`vws`: Vertical Weighted Strips in R using C++

Andrew M. Raim

Abstract

The `vws` package facilitates the vertical weighted strips (VWS) method in R. VWS is an approach to construct proposals for rejection sampling which is explored in Raim et al. (2025). The `vws` package provides an API to program the necessary components for the proposal and carry out rejection sampling. The API of the `vws` package is in C++ and makes use of templates and object-oriented programming; a working understanding of these techniques is assumed. The primary intended usage is that sampler logic is implemented in C++ as a variate generation function; this is exposed in R for use in applications. This vignette presents the API and several complete examples; multiple variations of VWS samplers are presented with each example. The `vws` package and this vignette have been written primarily for univariate targets; however, it is intended that the framework can be used to handle multivariate problems as well.

Table of contents

For correspondence: andrew.raim@gmail.com. Center for Statistical Research & Methodology, U.S. Census Bureau, Washington, DC, 20233, U.S.A. Document was compiled 2026-06-14 07:10:25 UTC.

Disclaimer and Acknowledgments

This document is released to inform interested parties of ongoing research and to encourage discussion of work in progress. Any views expressed are those of the authors and not those of the U.S. Census Bureau.

The author is grateful to Drs. James Livsey and Kyle Irimata for discussions that motivated development of this package. Thanks to Drs. Kyle Irimata and Tommy Wright for their reviews of this manuscript.

Although there are no guarantees of correctness of the `vws` package, reasonable efforts will be made to address shortcomings. Comments, questions, corrections, and possible improvements can be communicated through the project's Github repository (https://github.com/andrewraim/vws).

1 Introduction

The `vws` package facilitates vertical weighted strips (VWS) sampling (Raim et al. 2025). VWS is an approach to construct proposal distributions for rejection sampling. Here, the target density is regarded as a weighted density - the product of a nonnegative weight function and a base density. The weight function is majorized by a given function (which bounds it above) to yield an envelope which is recombined with the base density to form a proposal distribution. If it is convenient to generate draws from the proposal, and the distance between the original and majorized weight function is not too large, such a proposal can be effectively used in rejection sampling. When the weight function is majorized in a piecewise manner over the support, the resulting proposal can be regarded as a finite mixture. Such a mixture can be refined to reduce the rejection probability to an acceptable tolerance. The term "vertical weighted strips" emphasizes that it is an extension of a method referred to as "vertical strips" in Section 3.6.1 of Martino et al. (2018), and was previously described in Chapter VIII of Devroye (1986). Partitioning in VWS is done to facilitate majorization of the weight function rather than to majorize the entire target density as in vertical strips.

The `vws` package provides tools to construct and utilize VWS proposal distributions. Variate generation functions are constructed in C++ and may be exposed for use in R via the Rcpp package (Eddelbuettel 2013). C++ is taken as the primary programming language because of its efficiency as a compiled language and its formal approach to object-orientation and template programming. The `fntl` package (Raim 2024) is integral to `vws` usage; it provides a simplified C++ interface to useful routines in the R application programming interface (API) which is described in R Core Team (2025), as well as other useful numerical routines, where functions are supplied as lambdas. The `vws` package has been designed primarily to generate from univariate distributions (both continuous and discrete). It is also possible to construct proposals for multivariate distributions using the VWS approach; the `vws` package is intended to support code development for such settings as well.

This vignette presents an overview of the package, its API, and several examples of working samplers. An `R>` prompt is shown in some code displays to emphasize interaction via the console. Examples make liberal use of the `tidyverse` (Wickham et al. 2019), especially `ggplot2` to plot and `dplyr` to manipulate tables. Some of the lengthier codes for the examples are not presented in this document;

however, complete codes are provided with the package. They can be accessed in the `doc/examples` folder of the installed `vws` package. This path can be accessed in R with the following command.

```
> file.path(path.package("vws"), "doc", "examples")
```

The remainder of the vignette proceeds as follows. Section 2 briefly reviews VWS sampling. Section 3 gives an overview of the package, its major components, and important preliminaries for programming. Section 4 provides a detailed specification of the C++ API. Section 5 develops several VWS samplers to generate from the Von Mises Fisher (VMF) distribution following the application in Raim et al. (2025). Section 6 develops samplers in a disclosure avoidance setting relevant to the U.S. Census Bureau, which is considered by Raim (2021), Irimata et al. (2022), and Janicki et al. (2025+). Section 7 develops VWS samplers for the Bessel count distribution (Devroye 2002). In some applications, it may be desirable to have a persistent VWS proposal object that can gradually be refined and reused to draw samples; Section 8 presents several mechanisms to achieve this.

> Quick Start
>
> Readers looking for a quick start can go directly to the first example in Section 5.1 whose contents are described with the highest amount of detail.

2 A Brief Review of Vertical Weighted Strips

The objective of VWS is to sample from a weighted target density

$$f(x) = f_0(x)/\psi, \quad f_0(x) = w(x)g(x), \quad \psi = \int_\Omega f_0(x)d\nu(x), \tag{1}$$

where Ω is the support, ν is a dominating measure, g is assumed to be a normalized density, $w(x)$ is a nonnegative weight function, and ψ is a normalizing constant. We will construct a proposal of the form

$$h(x) = h_0(x)/\psi_N, \quad h_0(x) = \overline{w}(x)g(x), \quad \psi_N = \int_\Omega h_0(x)d\nu(x).$$

We will say that a function ϕ *majorizes* w on $A \subseteq \Omega$ if $\phi(x) \geq w(x)$ for $x \in A$. Let $\mathrm{I}\{x \in A\}$ denote the indicator function for $x \in A$. Suppose Ω is partitioned into regions $\mathscr{D}_1, \ldots, \mathscr{D}_N$ and there are corresponding functions $\overline{w}_1, \ldots, \overline{w}_N$ where $\overline{w}_j$ majorizes w on $\mathscr{D}_j$. Then $\overline{w}(x) = \sum_{j=1}^N \overline{w}_j(x)\,\mathrm{I}\{x \in \mathscr{D}_j\}$ majorizes w on Ω and the unnormalized proposal becomes

$$h_0(x) = g(x)\sum_{j=1}^N \overline{w}_j(x)\,\mathrm{I}\{x \in \mathscr{D}_j\}.$$

With this construction, $f_0(x) \leq h_0(x)$ for all $x \in \Omega$. Therefore, classical rejection sampling can be carried out by drawing u from Uniform$(0,1)$, x from h, and accepting x as a draw from f if $u \leq f_0(x)/h_0(x)$. A benefit of this construction is that the functional form of the density can help to guide proposal selection. The normalized h can be obtained by defining

$$\overline{\xi}_j = \int_{\mathscr{D}_j} \overline{w}_j(x)g(x)d\nu(x)$$

and

$$\psi_N = \sum_{j=1}^{N} \overline{\xi}_j,$$

giving the finite mixture

$$h(x) = h_0(x)/\psi_N = \sum_{j=1}^{N} \pi_j g_j(x), \tag{2}$$

a finite mixture with mixing weights and component densities,

$$\pi_j = \frac{\overline{\xi}_j}{\sum_{\ell=1}^{N} \overline{\xi}_\ell}, \quad \text{and} \quad g_j(x) = \overline{w}_j(x) g(x)\, \mathrm{I}\{x \in \mathscr{D}_j\}/\overline{\xi}_j,$$

respectively. The g_j are truncated and reweighted versions of base distribution g. In addition to the majorizer, suppose that $\underline{w}_j$ is a *minorizer* of w so that $0 \le \underline{w}_j(x) \le w(x)$ for all $x \in \mathscr{D}_j$, and let $\underline{\xi}_j = \int_{\mathscr{D}_j} \underline{w}_j(x) g(x) d\nu(x)$. Note that the majorizer and minorizer need not have the same functional form. When h is used as a proposal in rejection sampling, an upper bound for the probability of rejection is

$$\rho_+ = \sum_{j=1}^{N} \rho_j, \quad \text{where} \quad \rho_j = \frac{\overline{\xi}_j - \underline{\xi}_j}{\sum_{j=1}^{N} \overline{\xi}_\ell}. \tag{3}$$

The quantities $\rho_1, \ldots, \rho_N$ are the contributions of each region to ρ_+. The bound ρ_+ can be used to determine whether a VWS proposal will be viable for rejection sampling; if it is not much less than 1, rejections may be too frequent for practical use. The proposal may be refined by altering the partitioning or considering a different majorizer. Several specific choices of majorizer are considered by Raim et al. (2025) and will be reviewed in the remainder of this section. Section 2.1 discusses the use of a constant majorizing function. A linear majorizing function is discussed in Section 2.2. Section 2.3 reviews several knot selection methods.

2.1 Constant Majorizer

Suppose $\Omega = (\alpha_0, \alpha_N]$ is an interval whose endpoints may or may not be finite and $w(x) < \infty$ on Ω. We can majorize w using a constant $\overline{w}_j \ge \sup_{x \in \mathscr{D}_j} w(x)$ on each $\mathscr{D}_j$ so that the majorizer is $\overline{w}(x) = \sum_{j=1}^{N} \overline{w}_j\, \mathrm{I}\{x \in \mathscr{D}_j\}$. Here we obtain component densities of finite mixture (2) as

$$g_j(x) = g(x)\, \mathrm{I}\{x \in \mathscr{D}_j\}/\, \mathrm{P}(T \in \mathscr{D}_j), \quad \text{with} \quad \overline{\xi}_j = \overline{w}_j\, \mathrm{P}(T \in \mathscr{D}_j).$$

Furthermore, if we assume a constant minorizer $\underline{w}_j \le \inf_{x \in \mathscr{D}_j} w(x)$, then $\underline{\xi}_j = \underline{w}_j\, \mathrm{P}(T \in \mathscr{D}_j)$.

2.2 Linear Majorizer

Suppose $\Omega = (\alpha_0, \alpha_N]$ is an interval whose endpoints may or may not be finite and $0 < w(x) < \infty$ on Ω. Furthermore, suppose Ω is partitioned into regions $\mathscr{D}_j = (\alpha_{j-1}, \alpha_j]$ where w is entirely

either log-convex or log-concave for each $j = 1 \ldots, N$. An exponentiated linear function $\overline{w}_j(x) = \exp\{\overline{\beta}_{0j} + \overline{\beta}_{1j}x\}$ can be used to majorize w on $\mathscr{D}_j$ with an appropriate choice of coefficients $\overline{\beta}_{0j}$ and $\overline{\beta}_{1j}$. Therefore, the piecewise linear function $\overline{w}(x) = \sum_{j=1}^{N} \overline{w}_j(x)\, \mathrm{I}\{x \in \mathscr{D}_j\}$ is a majorizer on Ω. Here the component densities of finite mixture (2) are

$$g_j(x) = \exp\{\overline{\beta}_{0j} + \overline{\beta}_{1j}x\} g(x)\, \mathrm{I}\{x \in \mathscr{D}_j\}/\overline{\xi}_j, \quad \text{with} \quad \overline{\xi}_j = e^{\overline{\beta}_{0j}} \int_{\alpha_{j-1}}^{\alpha_j} e^{\overline{\beta}_{1j} \cdot x} g(x) d\nu(x).$$

In certain cases, such as where g is in an exponential family, the integral in $\overline{\xi}_j$ can be simplified and g_j is seen to be a familiar distribution. We may also assume a linear minorizer $\overline{w}(x) = \sum_{j=1}^{N} \underline{w}_j(x)\, \mathrm{I}\{x \in \mathscr{D}_j\}$ with $\overline{w}_j(x) = \exp\{\underline{\beta}_{0j} + \underline{\beta}_{1j}x\}$ for appropriate coefficients $\underline{\beta}_{0j}$ and $\underline{\beta}_{1j}$.

To identify coefficients $\overline{\beta}_{0j}$, $\overline{\beta}_{1j}$, $\underline{\beta}_{0j}$, and $\underline{\beta}_{1j}$, consider the following remarks corresponding to the log-concave and log-convex cases, respectively.

Remark 2.1. Suppose w is log-concave on $\mathscr{D}_j$. For any $c \in \mathscr{D}_j$,

$$\begin{aligned} \log w(x) &\leq \log w(c) + (x - c)\nabla(c) \\ &= \overline{\beta}_{0j} + \overline{\beta}_{1j} \cdot x, \end{aligned} \tag{4}$$

so that $\overline{w}_j(x) = \exp\{\overline{\beta}_{0j} + \overline{\beta}_{1j} \cdot x\}$ with $\overline{\beta}_{0j} = \log w(c) - c \cdot \nabla(c)$, $\overline{\beta}_{1j} = \nabla(c)$, and $\nabla(x) = \frac{d}{dx} \log w(x)$. For an accompanying minorizer, let $\lambda \in [0, 1]$ for a given $x \in \mathscr{D}_j$ so that $x = (1 - \lambda)\alpha_{j-1} + \lambda\alpha_j$. Concavity of $\log w(x)$ gives

$$\begin{aligned} \log w(x) &\geq (1 - \lambda) \log w(\alpha_{j-1}) + \lambda \log w(\alpha_j) \\ &= \log w(\alpha_{j-1}) + \frac{x - \alpha_{j-1}}{\alpha_j - \alpha_{j-1}} [\log w(\alpha_j) - \log w(\alpha_{j-1})] \\ &= \underline{\beta}_{0j} + \underline{\beta}_{1j} \cdot x, \end{aligned} \tag{5}$$

so that $\underline{w}_j(x) = \exp\{\underline{\beta}_{0j} + \underline{\beta}_{1j} \cdot x\}$ with $\underline{\beta}_{0j} = \log w(\alpha_{j-1}) - \alpha_{j-1}\underline{\beta}_{1j}$ and $\underline{\beta}_{1j} = \{\log w(\alpha_j) - \log w(\alpha_{j-1})\}/\{\alpha_j - \alpha_{j-1}\}$. ■

Remark 2.2. Suppose w is log-convex on $\mathscr{D}_j$. A majorizer and minorizer are obtained from (5) and (4), respectively. Note that they have switched roles from Remark 2.1. ■

The following remark describes one possible choice for the expansion point c which will be utilized in the examples in Sections 5.3, 6.3, and 7.3.

Remark 2.3. For a majorizer in the log-concave case, we choose c to minimize the L1 distance between unnormalized densities,

$$\begin{aligned} c^* &= \operatorname*{argmin}_{c \in \mathscr{D}_j} \int_{\mathscr{D}_j} [\overline{w}_j(x) - w(x)] g(x) d\nu(x) \\ &= \operatorname*{argmin}_{c \in \mathscr{D}_j} \left\{ w(c) \exp\{-c\nabla(c)\} \int_{\mathscr{D}_j} \exp\{x\nabla(c)\} g(x) d\nu(x) \right\} \\ &= \operatorname*{argmin}_{c \in \mathscr{D}_j} \Big\{ \log w(c) - c\nabla(c) + \log M_j(\nabla(c)) \Big\}, \end{aligned} \tag{6}$$

where $\overline{w}_j(x) = \exp\{\overline{\beta}_{0j} + \overline{\beta}_{1j} \cdot x\}$, $\overline{\beta}_{0j} = \log w(c) - c \cdot \nabla(c)$, $\overline{\beta}_{1j} = \nabla(c)$, $\nabla(x) = \frac{d}{dx} \log w(x)$, and $M_j(s) = \int_{\alpha_{j-1}}^{\alpha_j} e^{xs} g(x) d\nu(x)$. A similar choice of c for a minorizer in the log-convex case is

$$\begin{aligned} c^* &= \operatorname*{argmin}_{c \in \mathscr{D}_j} \int_{\mathscr{D}_j} [w(x) - \underline{w}_j(x)] g(x) d\nu(x) \\ &= \operatorname*{argmax}_{c \in \mathscr{D}_j} \int_{\mathscr{D}_j} \underline{w}_j(x) g(x) d\nu(x) \\ &= \operatorname*{argmax}_{c \in \mathscr{D}_j} \Big\{ \log w(c) - c\nabla(c) + \log M_j(\nabla(c)) \Big\}. \end{aligned} \tag{7}$$

where $\overline{w}_j(x) = \exp\{\underline{\beta}_{0j} + \underline{\beta}_{1j} \cdot x\}$, $\underline{\beta}_{0j} = \log w(c) - c \cdot \nabla(c)$, $\underline{\beta}_{1j} = \nabla(c)$, $\nabla(x) = \frac{d}{dx} \log w(x)$. ■

2.3 Knot Selection

After selecting a decomposition of the target into weighted form (1), the choice of knots is important to obtain an efficient proposal distribution. One option is to specify knots at known locations. For example, we can consider using evenly spaced points on a given interval within the support; however, this may not take into account important features of the target and add wasteful components to the finite mixture which do not contribute much to the quality of the approximation. An automated option that often achieves much better results is the rule of thumb proposed by Raim et al. (2025), shown here as Algorithm 1. This method sequentially refines an initial partition of the support consisting of one or more regions. At each step a region is selected and bifurcated at its midpoint. The selection is probabilistic, with probabilities proportional to $\rho_1, \ldots, \rho_N$. A deterministic ("greedy") variation can be considered by instead always selecting a region ℓ whose value ρ_ℓ is largest.

Bifurcation at a midpoint is mostly straightforward in univariate settings where the support is a subset of the real line. However, there may be multiple ways to characterize this operation when the support is multivariate, perhaps with more structure like a multinomial sample space. The necessary operations can be coded for such settings within the `vws` framework; however, this document does not consider such cases in detail.

Algorithm 1 Probabilistic rule of thumb for sequential knot selection.

Input: maximum number of knots to add N.
Input: initial vector of internal knots $\alpha_1, \ldots, \alpha_{N_0-1}$; may be empty with $N_0 = 0$.
Input: tolerance $\epsilon > 0$.

$j \leftarrow 0$
while $j \leq N$ **do**
  Let $\mathscr{D}_\ell$ with ρ_ℓ for $\ell \in \{1, \ldots, N_0 + j\}$ be current regions.
  If $\sum_{\ell=1}^{N_0+j} \rho_\ell < \epsilon$, break from the loop.
  Draw $\ell \in \{1, \ldots N_0 + j\}$ from Discrete$(\rho_1, \ldots, \rho_{N_0+j})$.
  Let α^* be midpoint of $\mathscr{D}_\ell$; add α^* to vector of knots.
  Let $j \leftarrow j + 1$.
end while
return $(\alpha_0, \ldots, \alpha_{N_0+j})$.

3 Overview of Package

The `vws` package aims to support the methodology which was described in the previous section. The present section will describe tools in the package which can be used to formulate a problem, construct a proposal, and generate samples.

Use of the `vws` package centers on two classes. The `FMMProposal` class represents a finite mixture of the form (2) and encapsulates the operations needed for rejection sampling. The `Region` class represents region $\mathscr{D}_j$ and the operations required for VWS. `Region` itself is an abstract base class which defines an interface needed by the framework. All problem-specific logic is coded within a subclass of `Region`: either by using one of the provided subclasses or by coding a new subclass. The subclasses `RealConstRegion` and `IntConstRegion` provide implementations of the constant majorizer from Section 2.1, for continuous and discrete univariate distributions, respectively. The linear majorizer described in Section 2.2 requires more customization than the constant majorizer; here, users should create a subclass of `Region`. An `FMMProposal` object is created from $N \geq 1$ `Region` objects that represent the partition $\mathscr{D}_1, \ldots, \mathscr{D}_N$ of Ω.

The `rejection` function takes an object `h` of class `FMMProposal` and carries out rejection sampling to obtain n draws. The `rejection` function returns a vector of accepted draws along with information about the number of rejections. Figure 3.1 displays a diagram of the high-level design just described.

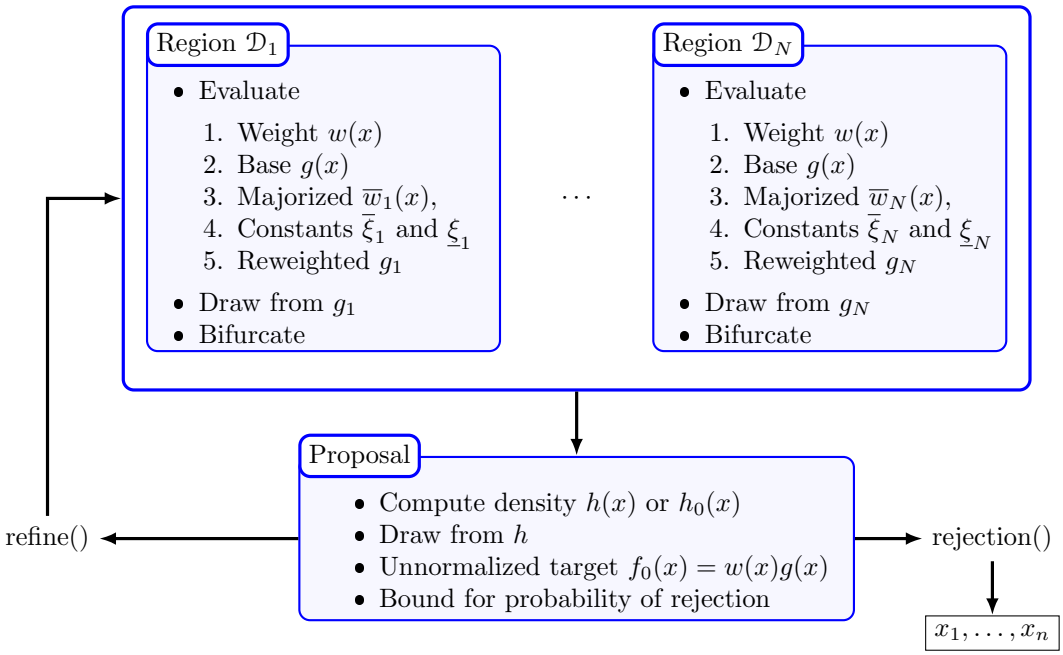


Figure 3.1: Design of `vws` package.

The following display outlines a typical workflow for VWS rejection sampling in C++. Some of the lines are labeled with numbers; descriptions of these labels are given after the code.

```
// [[Rcpp::depends(vws, fntl)]]                                    ①
#include "vws.h"                                                   ②
```

```cpp
#include "MyRegion.h"

// [[Rcpp::export]]                                                    ③
Rcpp::List sample(unsigned int n)
{
    MyRegion supp( ... );                                             ④
    vws::FMMProposal<double, MyRegion> h(supp);                       ⑤
    h.refine(N - 1, 0.01);                                            ⑥
    auto out = vws::rejection(h, n);                                  ⑦

    return Rcpp::List::create(                                        ⑧
        Rcpp::Named("draws") = out.draws,
        Rcpp::Named("rejects") = out.rejects
    );
}
```

① This annotation ensures that `vws` and `fntl` packages are linked during compilation. `fntl` is not explicitly used in this example, but it is needed for linking by the `vws` library regardless.
② Include the main header for the `vws` package so that the API is accessible.
③ This annotation exports the `sample` function for use in R. This is not needed if the `sample` function will be used only in C++.
④ Construct a region of type `MyRegion` which we have presumably defined in `MyRegion.h`. The symbol `...` here is a placeholder for any arguments needed by the constructor.
⑤ Construct an `FMMProposal` based on a single region `supp`. Note that it is a template class that needs two template arguments: the first (`double`) indicates the data type of the distribution and the second (`MyRegion`) indicates the type of region it will contain.
⑥ Refine the proposal using Algorithm 1 up to $N - 1$ times or until tolerance `0.01` is achieved for (3).
⑦ Carry out rejection sampling with the refined proposal `h`.
⑧ Pack elements of the struct `out` into an `Rcpp::List` and return them to the caller.

Note that the `FMMProposal` class, the `rejection` function, and other elements of the `vws` API are accessed in the `vws` namespace. Details for the API are given in Section 4. Once the `sample` function is successfully exported as an R function, we may invoke it as usual. Suppose the code in the previous display is saved is in a file named `sample.cpp`.

```
> Rcpp::sourceCpp("sample.cpp")
> out = sample(n = 100)
> head(out$draws)
> head(out$rejects)
```

Working Examples

A complete VWS example is given in Section 5.1; this demonstrates the use of `RealConstRegion` to implement a sampler with a minimal amount of coding. Section 5.2 is an improvement

which requires more coding; here we provide functions to determine values of the constant majorizer and minorizer analytically. Finally, Section 5.3 demonstrates how to subclass `Region` to implement a linear majorizer, which requires more intricate coding. Subsequent examples are given in Section 6 and Section 7, but with less explanation.

It is useful to mention several other preliminaries for the `vws` package; see the following remarks.

Remark 3.1. Lambdas in C++ are functions which can be defined within the course of a program, regarded as objects which can be passed to other functions, and capture variables in the environment so that they do not need to be passed as arguments. They should feel very familiar to R programmers who likely use R functions in this way. Here is a snippet with several examples of lambdas.

```
double z = 3;
typedef function<double(double)> dfdd;
dfdd f1 = [&](double x, double y) -> double { return x*y*z; };
dfdd f2 = [&](double x, double y) { return x*y*z; };
dfdd f3 = [=](double x, double y) { return x*y*z; };
dfdd f4 = [](double x, double y) { return x*y*3; };
```

These four functions carry out the same operation and may be invoked in the usual way.

```
double out = f1(1, 2);
```

The typedef `dfdd` is given as a shorthand for a Standard Template Library (STL) `function` which takes two `double` variables as arguments and returns a `double`. An STL `function` can be passed to other functions like any other variable. Function `f1` explicitly states the return type of the lambda using `-> double` while it is omitted in the others to be inferred by the compiler. The variable `z` is *captured* by `f1`, `f2`, and `f3`; in other words, it is "baked in" to the function definition without needing to be passed as an argument. Functions `f1` and `f2` capture `z` by reference while `f3` captures `z` by value, indicated with the syntaxes `[&]` and `[=]` respectively. Function `f4` uses empty square brackets `[]` to indicate that it does not capture any outside variables.

The `fntl` package defines several function typedefs used in the `vws` package, and also provides access to numerical tools such as integration, differentiation, and optimization which take STL `functions` as arguments. ■

Remark 3.2. Many of the computations in this vignette and within the `vws` package are carried out on the logarithmic scale. This is to avoid loss of precision when working with very large or small magnitude numbers. Several functions to support log-scale arithmetic are given in Section 4.9. ■

Remark 3.3. To help automate computation of constant majorizers and minorizers for use with `RealConstRegion` and `IntConstRegion`, the default is to use numerical optimization to find appropriate constants on each region in the partition. The particular optimization method is presented in Section 4.8. This can be somewhat of a computational burden; it can sometimes be avoided when there is a closed form solution or other informed way to arrive at maximizing / minimizing constants. The API for `RealConstRegion` and `IntConstRegion` accepts user-specified lambdas to specify alternative methods. ■

4 C++ API

This section documents functions, classes, and other components in the C++ API. Note that some functions in the C++ API are also exposed as R functions; see the `vws` package manual for these.

4.1 Typedefs

The following typedef is used to represent functions that take `double` and `bool` arguments and return a `double`. It is used as the type for weight functions in particular, with `x` the argument of the weight function and `log = true` if the result is to be returned on the log-scale.

```
typedef std::function<double(double x, bool log)> dfdb;
```

The following typedef represents a function that optimizes (i.e., maximizes or minimizes) a weight function on a given interval.

```
typedef std::function<double(
    const dfdb& w,                                    ①
    double lo,                                        ②
    double hi,                                        ③
    bool log                                          ④
)> optimizer;
```

① A weight function.
② Lower bound of the interval; may be `R_NegInf`.
③ Upper bound of the interval; may be `R_PosInf`.
④ Logical; `log = true` specifies that the result should be the optimal value $\log w(x^*)$, given on the log-scale. Otherwise, the result should be the optimal value $w(x^*)$ on the original scale.

The following typedef represents a function that computes the midpoint of a given interval. Note that this type name can clash with the `midpoint` function in the STL; therefore, it is referenced using its namespace `vws::midpoint` within the `vws` package.

```
typedef std::function<double(double a, double b)> midpoint;
```

4.2 Finite Mixture Proposal

The class `FMMProposal` represents a VWS proposal in the form of (2).

4.2.1 Class Definition

`FMMProposal` has two template arguments: `T` is the data type for the underlying distribution (e.g., `T = double` for a real-valued univariate random variable) and `R` is the type of region which the proposal will be based upon.

```
template <class T, class R>
class FMMProposal { ... };
```

4.2.2 Constructor

The primary constructor takes a vector of regions of type `R`. There must be at least one region in the vector. These regions are expected to be a partition of the support Ω.

```
FMMProposal(const std::vector<R>& regions);
```

A second constructor takes a single region. This is a special case of the previous one which is provided for convenience.

```
FMMProposal(const R& region);
```

4.2.3 Distribution Methods

The following functions make use of the proposal as a distribution. They are especially utilized in rejection sampling.

```
std::vector<T> r(unsigned int n = 1) const;                                    ①
std::pair<std::vector<T>, std::vector<unsigned int>>                           ②
    r_ext(unsigned int n = 1) const;
double d(const T& x, bool normalize = true, bool log = false) const;          ③
double w_major(const T& x, bool log = true) const;                            ④
double d_target_unnorm(const T& x, bool log = true) const;                    ⑤
```

① Draw n variates of type `T` from the proposal.
② Draw n variates of type `T` from the proposal and retain the indices of the regions used in each draw. The result is an STL pair whose first element is the vector of draws and second element is the vector of indices.
③ Evaluate the density h on the given x. If `normalize = false`, $h_0(x)$ is computed; otherwise $h(x)$ is computed.
④ Evaluate the majorized weight function $\overline{w}(x)$.
⑤ Evaluate the unnormalized target $f_0(x) = w(x)g(x)$.

For the `log` argument, the value on the log-scale is returned if `true`; otherwise, the value on the original scale is returned.

4.2.4 Accessors

The following accessors are provided.

```
Rcpp::NumericVector xi_upper(bool log = true) const;                    ①
Rcpp::NumericVector xi_lower(bool log = true) const;                    ②
Rcpp::LogicalVector bifurcatable() const;                               ③
Rcpp::NumericVector pi(bool log = false) const;                         ④
Rcpp::NumericVector bound_contrib(bool log = false) const;              ⑤
double bound(bool log = false) const;                                   ⑥
double nc(bool log = false) const;                                      ⑦
unsigned int size() const;                                              ⑧
```

① Get the constants $\overline{\xi}_1, \ldots, \overline{\xi}_N$.
② Get the constants $\underline{\xi}_1, \ldots, \underline{\xi}_N$.
③ Get a vector of N logical values indicating whether the corresponding regions can be bifurcated. For example, when the support `T` is `int`, a region $(a, b]$ containing one integer should not be bifurcated because one of the two resulting regions will not contain any points of the support.
④ Get the mixing proportions $\pi_1, \ldots, \pi_N$.
⑤ Get the contributions $\rho_1, \ldots, \rho_N$ to bound (3) corresponding to the N regions.
⑥ Get the overall rejection bound (3).
⑦ Get the normalizing constant ψ_N
⑧ Get the number of regions N.

Methods above with the a `log` argument return values on the log-scale when `log = true`; values are returned on the original scale otherwise.

4.2.5 Iterators

The following methods can be used to get (read-only) iterators to internal data structures. These can be more efficient than the accessors in Section 4.2 because they do not make a copy of the data.

```
std::set<R>::const_iterator regions_begin() const;                      ①
std::set<R>::const_iterator regions_end() const;
Rcpp::NumericVector::const_iterator log_xi_upper_begin() const;         ②
Rcpp::NumericVector::const_iterator log_xi_upper_end() const;
Rcpp::NumericVector::const_iterator log_xi_lower_begin() const;         ③
Rcpp::NumericVector::const_iterator log_xi_lower_end() const;
Rcpp::LogicalVector::const_iterator bifurcatable_begin() const;         ④
Rcpp::LogicalVector::const_iterator bifurcatable_end() const;
```

① The start and end of the set of regions (of template type `R`) in the proposal.
② The start and end of the vector $(\overline{\xi}_1, \ldots, \overline{\xi}_N)$.
③ The start and end of the vector $(\underline{\xi}_1, \ldots, \underline{\xi}_N)$.

④ The start and end of the vector of indicators for whether regions are bifurcatable.

4.2.6 Refining the Proposal

Two functions are provided to refine the proposal from $\mathscr{D}_1, \ldots, \mathscr{D}_N$ into a finer partition. Both variants return a vector $\rho_+^{(0)}, \ldots, \rho_+^{(N_0)}$ which represents values of the bound (3) at each refinement step; N_0 is the number of steps taken, $\rho_+^{(j)}$ represents the value at refinement $j = 1, \ldots, N_0$, and $\rho_+^{(0)}$ represents the initial value. The first variant partitions at a given vector of knots.

```
Rcpp::NumericVector refine(
    const std::vector<T>& knots,                                    ①
    bool log = true                                                 ②
);
```

① A vector of knots.
② If `log = true` return bound values on the log-scale.

The second variant uses the rule of thumb for sequential knot selection from Raim et al. (2025). Refining will halt when (3) reduces below `tol`; which is possible when `tol` is positive; otherwise, refining halts after `N` steps.

```
Rcpp::NumericVector refine(
    unsigned int N,                                                 ①
    double tol = 0,                                                 ②
    bool greedy = false,                                            ③
    unsigned int report = fntl::uint_max,                           ④
    bool log = true                                                 ⑤
);
```

① Maximum number of refinements to make.
② Tolerance for (3); refinement will halt if ρ_+ reaches this.
③ If `greedy = true`, the region with the largest ρ_ℓ is always selected for partitioning; otherwise, regions are selected with probabilities proportional to $\rho_1, \ldots, \rho_N$.
④ If `log = true` return bound values $\rho_+^{(0)}, \ldots, \rho_+^{(N)}$ on the log-scale.
⑤ The period at which progress is reported to the console. E.g., use `period = 2` to report progress every two selections.

The `seq` function is provided as a convenience to generate equally-spaced knots for univariate real-valued intervals.

```
std::vector<double> seq(double lo, double hi, unsigned int N,
    bool endpoints = false);
```

4.2.7 Summary Methods

Several methods are provided to summarize the regions in the proposal. Table 4.1 describes the columns in the summary and rows correspond to the N regions.

```
Rcpp::DataFrame summary() const;                    ①
void print(unsigned int n = 5) const;               ②
```

① Get a data frame with the summary.
② Print summary to the console.

Table 4.1: Data frame returned by summary method of `FMMProposal`.

Column	Description
`Region`	String describing region j.
`log_xi_upper`	$\log \overline{\xi}_j$
`log_xi_lower`	$\log \underline{\xi}_j$
`log_volume`	$\log \rho_j$

4.3 Region Base Class

`Region` is an abstract base class whose interface represents the problem-specific logic that must be coded to implement VWS. Users create a subclass of this method to construct a proposal for a given problem. However, for the most common application of VWS - univariate support with a constant majorizer - users may be able to leverage the provided subclasses in Sections 4.4 and 4.5.

The class has one template argument `T`, which is the data type for the underlying distribution.

```
template <class T>
class Region { ... };
```

The interface consists of the following public methods. These are abstract and must be implemented in a subclass.

```
virtual double d_base(const T& x, bool log = false) const = 0;      ①
virtual std::vector<T> r(unsigned int n) const = 0;                 ②
virtual bool s(const T& x) const = 0;                               ③
virtual double w(const T& x, bool log = true) const = 0;            ④
virtual double w_major(const T& x, bool log = true) const = 0;      ⑤
virtual bool is_bifurcatable() const = 0;                           ⑥
virtual double xi_upper(bool log = true) const = 0;                 ⑦
virtual double xi_lower(bool log = true) const = 0;                 ⑧
virtual std::string description() const = 0;                        ⑨
```

① Evaluate the density function g of the base distribution.
② Generate a vector of n draws from g_j specific to this region.
③ Indicator of whether x is in the support for g_j specific to this region.
④ The weight function w.
⑤ Majorized weight function $\overline{w}_j$ for this region.
⑥ Indicator of whether this region is bifurcatable into two smaller regions. This is used when refining a proposal; see Section 2.3. One reason that a region should not be bifurcated is when one of the resulting regions will not have any points in the support.
⑦ The quantity $\overline{\xi}_j$ for this region.
⑧ The quantity $\underline{\xi}_j$ for this region.
⑨ A string that describes this region; used for printing to the console.

The argument `log = true` in the methods above requests values to be returned on the log-scale.

Subclasses of `Region` are expected to provide several additional functions which are not specified in the interface. Suppose `MyRegion` is a subclass of `Region` whose elements are variables of type `type` (e.g., `double`); its definition should include the following.

```
class MyRegion : public Region<type>
{
public:
    std::pair<MyRegion,MyRegion> bifurcate() const;                    ①
    std::pair<MyRegion,MyRegion> bifurcate(const type& x) const;       ②
    MyRegion singleton(const type& x) const;                           ③
    bool operator<(const MyRegion& x) const;                           ④
    bool operator==(const MyRegion& x) const;                          ⑤
    const MyRegion& operator=(const MyRegion& x);                      ⑥
...
};
```

① Bifurcate the current region into two regions; this version takes no arguments and should include logic to decide where to make the split.
② Bifurcate the current region into two regions; the argument `x` is used as the location to make the split.
③ Return a region on the singleton set $(x, x]$ with the information in the current object.
④ Comparison operator; used to order regions of this type.
⑤ Equality operator; check whether two regions of this type are equal.
⑥ Assignment operator; assign information from `x` to this object.

4.4 Region on Real-Valued Support with Constant Majorizer

`RealConstRegion` is a subclass of `Region`, defined in Section 4.3, specifically for univariate problems with continuous support where $\overline{w}(x) = \sum_{j=1}^{N} \overline{w}_j \, \mathrm{I}\{x \in \mathscr{D}_j\}$ is constructed from constants $\overline{w}_1, \ldots, \overline{w}_N$. It assumes a constant minorizer $\underline{w}(x) = \sum_{j=1}^{N} \underline{w}_j \, \mathrm{I}\{x \in \mathscr{D}_j\}$ is constructed from constants $\underline{w}_1, \ldots, \underline{w}_N$. The $\overline{w}_j$ and $\underline{w}_j$ are obtained using numerical optimization by default; however, user-supplied methods can be supplied to identify these values when it is possible.

4.4.1 Constructors

`RealConstRegion` has the following constructors. The first creates a region based on the interval $(a, b]$; the second creates a region based on the singleton set $\{a\}$, which is intended primarily for internal use.

```
RealConstRegion(
    double a,                                                    ①
    double b,                                                    ②
    const dfdb& w,                                               ③
    const UnivariateHelper& helper,                              ④
    const optimizer& maxopt = maxopt_default,                    ⑤
    const optimizer& minopt = minopt_default,                    ⑥
    const midpoint& mid = midpoint_default                       ⑦
);

RealConstRegion(
    double a,                                                    ①
    const dfdb& w,                                               ③
    const UnivariateHelper& helper                               ④
    const optimizer& maxopt = maxopt_default,                    ⑤
    const optimizer& minopt = minopt_default,                    ⑥
    const midpoint& mid = midpoint_default                       ⑦
);
```

① Lower limit of interval that defines this region.
② Upper limit of interval that defines this region.
③ Weight function w for the target distribution.
④ A container with operations of the base distribution g.
⑤ A function of type `optimizer` that maximizes `w` on the given region.
⑥ A function of type `optimizer` that minimizes `w` on the given region.
⑦ A function of type `midpoint` to compute the midpoint of region's interval.

The default optimizers, `maxopt_default` and `minopt_default`, use a hybrid numerical optimization method in Section 4.8 to optimize `w`.

The function `midpoint_default` is a version of the arithmetic midpoint with special handling for infinite endpoints. It is used primarily to bifurcate a given region into two smaller regions. In particular, the current implementation is

$$\text{midpoint}(a,b) = \begin{cases} 0, & \text{if } a = -\infty \text{ and } b = \infty, \\ b \cdot 2^{-\text{sign}(b)} - 1, & \text{if } a = -\infty \text{ and } b < \infty, \\ a \cdot 2^{\text{sign}(a)} + 1, & \text{if } a > -\infty \text{ and } b = \infty, \\ (a+b)/2, & \text{otherwise,} \end{cases}$$

where $\text{sign}(x) = \text{I}(x > 0) - \text{I}(x < 0)$. If both endpoints are infinite, zero is returned. When one endpoint is infinite, the finite endpoint is reduced in magnitude by a factor of $1/2$ and brought

in by an additional unit of one. The exponential factor helps to avoid large numbers of wasted regions during knot selection. The additive factor accommodates endpoints that are near zero; without it, midpoint(a, b) only brings the finite endpoint closer to zero which likely to create wasteful bifurcations.

4.4.2 Methods

The `midpoint` method returns a midpoint for the current region.

```
double RealConstRegion::midpoint() const
```

The `bifurcate` method returns two disjoint regions whose union is the current region. The result is given as an STL pair. The first version bifurcates at the midpoint of the current region, determined by the `midpoint` method. The second version partitions at the given x.

```
std::pair<RealConstRegion,RealConstRegion> bifurcate() const;
std::pair<RealConstRegion,RealConstRegion> bifurcate(const double& x) const;
```

The `is_bifurcatable` function always returns `true` in the case of a continuous support.

```
bool is_bifurcatable() const;
```

The `singleton` method returns a singleton interval $(x, x]$, using the current object's member data.

```
RealConstRegion singleton(const double& x) const;
```

The following methods determine an ordering of the current region and an another region specified as argument `x`. Region $(a_1, b_1]$ is considered "less than" $(a_2, b_2]$ if $b_1 \leq a_2$. The regions are considered equal if $a_1 = a_2$ and $b_1 = b_2$. Note that other elements such as w are g are not explicitly checked, and are assumed to be the same.

```
bool operator<(const RealConstRegion& x) const;
bool operator==(const RealConstRegion& x) const;
```

The following method assigns the current region to be equal to the argument `x`.

```
const RealConstRegion& operator=(const RealConstRegion& x);
```

4.5 Region on Integer-Valued Support with Constant Majorizer

The class `IntConstRegion` is a subclass of `RealConstRegion` that is specialized to integer-valued distributions.

```
class IntConstRegion : public RealConstRegion { ... };
```

4.5.1 Constructors

`IntConstRegion` has the following constructors.

```
IntConstRegion(
    double a,                                                    ①
    double b,                                                    ②
    const dfdb& w,                                               ③
    const UnivariateHelper& helper,                              ④
    const optimizer& maxopt = maxopt_default,                    ⑤
    const optimizer& minopt = minopt_default,                    ⑥
    const midpoint& mid = midpoint_default                       ⑦
);

IntConstRegion(
    double a,                                                    ①
    const dfdb& w,                                               ③
    const UnivariateHelper& helper                               ④
    const optimizer& maxopt = maxopt_default,                    ⑤
    const optimizer& minopt = minopt_default,                    ⑥
    const midpoint& mid = midpoint_default                       ⑦
);
```

① Lower limit of interval that defines this region.
② Upper limit of interval that defines this region.
③ Weight function w for the target distribution.
④ A container with operations of the base distribution g.
⑤ A function of type `optimizer` that maximizes `w` on the given region.
⑥ A function of type `optimizer` that minimizes `w` on the given region.
⑦ A function of type `midpoint` to compute the midpoint of region's interval.

The arguments here are analogous to those in Section 4.4.1. Note that `a` and `b` need not be integers here.

4.5.2 Methods

The following methods are specialized from `RealConstRegion`.

```
std::pair<IntConstRegion,IntConstRegion> bifurcate() const;
std::pair<IntConstRegion,IntConstRegion> bifurcate(const double& x) const;
```

The `is_bifurcatable` function returns `false` for regions whose width is smaller than 1.

```
bool is_bifurcatable() const;
```

The `singleton` method returns a singleton interval $(x, x]$, using the current object's weight function, base distribution, etc.

```
IntConstRegion singleton(const double& x) const;
```

The following methods determine an ordering of the current region and an another region specified as argument `x`. Region $(a_1, b_1]$ is considered "less than" $(a_2, b_2]$ if $b_1 \leq a_2$. The regions are considered equal if $a_1 = a_2$ and $b_1 = b_2$. Note that other elements such as w are g are not explicitly checked, and are assumed to be the same.

```
bool operator<(const IntConstRegion& x) const;
bool operator==(const IntConstRegion& x) const;
```

The following method assigns the current region to be equal to the argument `x`.

```
const IntConstRegion& operator=(const IntConstRegion& x);
```

4.6 Univariate Helper

The class `UnivariateHelper` is intended for use with `RealConstRegion` and `IntConstRegion`. It encapsulates several operations needed from the base distribution g. These operations are specified as lambdas in the constructor.

```
UnivariateHelper(
    const fntl::density& d,                                    ①
    const fntl::cdf& p,                                        ②
    const fntl::quantile& q                                    ③
);
```

① A function to evaluate density g.
② A function to evaluate the cumulative distribution function (CDF) G.
③ A function to evaluate the quantile function G^-.

The following methods on `UnivariateHelper` utilize the lambdas specified above.

```
double d(double x, bool log = false) const;                    ①
double p(double q, bool lower = true, bool log = false) const; ②
double q(double p, bool lower = true, bool log = false) const; ③
```

① Evaluate the density function at argument x. Result is on the log-scale if `log = true`.
② Evaluate the CDF at argument q. Result is on the log-scale if `log = true`. Result represents $P(X \leq q)$ if `lower = true` and $P(X > q)$ otherwise.

③ Evaluate the quantile function at argument p. Assume p is specified on the log-scale if `log = true`. Request p quantile if `lower = true` and $1 - p$ quantile otherwise.

4.7 Rejection Sampling

The following functions perform rejection sampling using a VWS proposal described in Section 4.2.

```
template <typename T, typename R>
rejection_result<T> rejection(
    const FMMProposal<T,R>& h,                                    ①
    unsigned int n,                                               ②
    const rejection_args& args                                    ③
);

template <typename T, typename R>
rejection_result<T> rejection(
    const FMMProposal<T,R>& h,                                    ①
    unsigned int n                                                ②
);
```

① An `FMMProposal` object to use as the proposal.
② The number of desired draws.
③ Additional arguments for rejection sampling. Default values are assumed in the second form.

Template arguments `T` and `R` correspond to the given proposal `h` and are described in Section 4.2. Additional arguments to `rejection` are provided via the following struct.

```
struct rejection_args {
    unsigned int max_rejects = std::numeric_limits<unsigned int>::max();    ①
    unsigned int report = std::numeric_limits<unsigned int>::max();         ②
    double ratio_ub = std::exp(1e-5);                                       ③
    fntl::error_action action = fntl::error_action::STOP;                   ④

    rejection_args() { };                                                   ⑤
    rejection_args(SEXP obj);                                               ⑥
    operator SEXP() const;                                                  ⑦
};
```

① Maximum number of rejections to tolerate overall (among all n attempted draws) before bailing out.
② Determines period at which progress is logged to the console.
③ Maximum allowed value for the ratio $f_0(x)/h_0(x)$, which may be slightly larger than 1 due to numerical precision. An error is thrown if the ratio is larger than this value.
④ The action to take if `max_rejects` rejections is exceeded; see Table 4.2. The definition of the enum `fntl::error_action` is given in Section 2 of Raim (2024).

⑤ Constructor that takes no arguments and initializes to default values.
⑥ Convert an `Rcpp::List` to a `rejection_args` struct.
⑦ Return a `Rcpp::List` from a `rejection_args` struct.

Table 4.2: Value of `action` used in `rejection_args.action` and its effect in `rejection`.

Action	Effect
`fntl::error_action::NONE`	Error condition is ignored.
`fntl::error_action::MESSAGE`	A message is emitted without halting.
`fntl::error_action::WARNING`	A warning is emitted without halting.
`fntl::error_action::STOP`	Halts when an error condition is encountered.

The return value of `rejection` is a struct of the following type.

```
template <typename T>
struct rejection_result
{
    std::vector<T> draws;                                    ①
    std::vector<unsigned int> rejects;                       ②

    operator SEXP() const;                                   ③
};
```

① Vector of draws.
② Vector of rejection counts; the ith element represents number of rejections observed before accepting the ith draw.
③ Return a `Rcpp::List` from a `rejection_args` struct.

If the maximum number og rejections are reached and `fntl::error_action::STOP`, the length of the vectors `draws` and `rejects` will be less than n.

4.8 Optimization on an Interval

The function `optimize_hybrid` is a hybrid optimization method for univariate functions $f(x) : [a, b] \to \mathbb{R}$ with bounds $x \in [a, b]$ whose endpoints may be finite or infinite. Uses Brent's method if both bounds are finite and BFGS otherwise. In the latter case, the bounds are enforced via a transformation.

```
optimize_hybrid_result optimize_hybrid(
    const fntl::dfd& f,                                      ①
    double init,                                             ②
    double lower,                                            ③
    double upper,                                            ④
    bool maximize,                                           ⑤
```

```
    unsigned maxiter = 100000                                    ⑥
);
```

① Objective function.
② Initial value used with BFGS.
③ Lower bound a of domain $[a, b]$.
④ Upper bound b of domain $[a, b]$.
⑤ Logical; if `true`, optimization will be a maximization. Otherwise it is a minimization.
⑥ Maximum number of iterations.

The result is a `optimize_hybrid_result` struct which is defined as follows.

```
struct optimize_hybrid_result {
    double par;                                                  ①
    double value;                                                ②
    std::string method;                                          ③
    int status;                                                  ④

    operator SEXP() const;                                       ⑤
};
```

① Final value of the optimization variable x.
② Final value of the objective function $f(x)$.
③ Description of the method used to obtain the result; see Table 4.3 for possible values.
④ Corresponds to a code from BFGS if it is used as `method`; otherwise 0.
⑤ Return an `Rcpp::List` from a `optimize_hybrid_result` struct.

Table 4.3: Possible values for `method` field of `optimize_hybrid_result` struct.

Message	Description
`"Brent"`	Brent optimization method was used.
`"BFGS"`	BFGS method was used.
`"Lower Limit Inf"`	For maximization, lower limit taken as `par` because its value is `inf`.
`"Upper Limit Inf"`	For maximization, upper limit taken as `par` because its value is `inf`.
`"Lower Limit NegInf"`	For minimization, lower limit taken as `par` because its value is `-inf`.
`"Upper Limit NegInf"`	For minimization, upper limit taken as `par` because its value is `-inf`.
`"Max at Lower Limit"`	Numerical maximization used, but larger value found at lower limit.
`"Max at Upper Limit"`	Numerical maximization used, but larger value found at upper limit.
`"Min at Lower Limit"`	Numerical minimization used, but smaller value found at lower limit.
`"Min at Upper Limit"`	Numerical minimization used, but smaller value found at upper limit.

4.9 Log-Scale Arithmetic

As mentioned in Remark 3.2, calculations in the package are carried out on the log-scale, where possible, to avoid issues from floating point numbers with very small or very large magnitudes.

Users may want to follow this convention when implementing their own sampling problems. Several included functions help to avoid explicit exponentiation.

The following computes the scalar $f(x) = \log\{\sum_{i=1}^{n} \exp(x_i)\}$ from a vector $x \in \mathbb{R}^n$ using the method described in this StackExchange post.

```
double log_sum_exp(const Rcpp::NumericVector& x);
```

The following functions carry out addition on the log scale using the property

$$\log(e^x + e^y) = t + \log\{1 + \exp[s - t]\}, \quad s = \min(x, y), \quad t = \max(x, y).$$

The first form takes scalar arguments x and y. The second and third forms take $x, y \in \mathbb{R}^n$ and produce a vector with elements $\log(e^{x_i} + e^{y_i})$, $i = 1, \ldots, n$.

```
double log_add2_exp(double x, double y);

std::vector<double> log_add2_exp(
    const std::vector<double>& x,
    const std::vector<double>& y
);
Rcpp::NumericVector log_add2_exp(
    const Rcpp::NumericVector& x,
    const Rcpp::NumericVector& y
);
```

The following carry out subtraction on the log scale using the property

$$\log(e^x - e^y) = x + \log\{1 - \exp[y - x]\}.$$

When x is smaller than y, the results is assumed to be `nan`. The first form takes scalar arguments x and y. The second and third forms take $x, y \in \mathbb{R}^n$ and produce a vector with elements $\log(e^{x_i} - e^{y_i})$, $i = 1, \ldots, n$.

```
double log_sub2_exp(double x, double y);

std::vector<double> log_sub2_exp(
    const std::vector<double>& x,
    const std::vector<double>& y
);
Rcpp::NumericVector log_sub2_exp(
    const Rcpp::NumericVector& x,
    const Rcpp::NumericVector& y
);
```

4.10 Generating from a Discrete Distribution

We make use of the Gumbel trick (e.g., Huijben et al. 2023) to draw from a discrete distribution with probabilities $p_1, \ldots, p_k$. This approach allows probabilities to be specified on the log-scale without the need to exponentiate or normalize them so that they sum to 1. The Gumbel trick generates a draw x from the desired discrete distribution via

$$X = \text{argmax}\{Z_1 + \log p_1, \ldots, Z_k + \log p_k\}, \quad Z_1, \ldots, Z_k \overset{\text{iid}}{\sim} \text{Gumbel}(0, 1),$$

where $\text{Gumbel}(0, 1)$ is a standard Gumbel distribution with density $f(x) = e^{-(x+e^x)}\, \text{I}(x > 0)$.

4.10.1 Categorical Distribution

The following functions generate a draw of X. The first form generates a single variate and the second form generates a sample of size n.

```
unsigned int r_categ(
    const Rcpp::NumericVector& p,                                    ②
    bool log = false,                                                ③
    bool one_based = false                                           ④
);
Rcpp::IntegerVector r_categ(
    unsigned int n,                                                  ①
    const Rcpp::NumericVector& p,                                    ②
    bool log = false,                                                ③
    bool one_based = false                                           ④
);
```

① Desired sample size.
② Vector of probabilities $p_1, \ldots, p_k$.
③ Logical; if `true`, argument `p` is interpreted as $\log p_1, \ldots, \log p_k$. Otherwise, it is interpreted as $p_1, \ldots, p_k$ on the original scale.
④ Logical; if `true`, support is assumed to be $\{1, \ldots, k\}$, where k is length of the given `p`. Otherwise it is assumed to be $\{0, \ldots, k-1\}$. The former is useful to generate indices in C++ while the latter is useful for indices in R.

4.10.2 Gumbel Distribution

The following functions are provided for the Gumbel distribution with location parameter μ and scale σ. They compute the density, CDF, quantile, and variate generation, respectively. The first group operates on scalars and the second are vectorized versions which operate on an independent and identically distributed sample.

```
double d_gumbel(
    double x,                                                        ①
```

```
    double mu = 0,                                                    ⑤
    double sigma = 1,                                                 ⑥
    bool log = false                                                  ⑦
);
double p_gumbel(
    double q,                                                         ②
    double mu = 0,                                                    ⑤
    double sigma = 1,                                                 ⑥
    bool lower = true,                                                ⑧
    bool log = false                                                  ⑦
);
double q_gumbel(
    double p,                                                         ③
    double mu = 0,                                                    ⑤
    double sigma = 1,                                                 ⑥
    bool lower = true,                                                ⑧
    bool log = false                                                  ⑦
);
double r_gumbel(
    double mu = 0,                                                    ⑤
    double sigma = 1                                                  ⑦
);

Rcpp::NumericVector d_gumbel(
    const Rcpp::NumericVector& x,                                     ①
    double mu = 0,                                                    ⑤
    double sigma = 1,                                                 ⑥
    bool log = false                                                  ⑦
);
Rcpp::NumericVector p_gumbel(
    const Rcpp::NumericVector& q,                                     ②
    double mu = 0,                                                    ⑤
    double sigma = 1,                                                 ⑥
    bool lower = true,                                                ⑧
    bool log = false                                                  ⑦
);
Rcpp::NumericVector q_gumbel(
    const Rcpp::NumericVector& p,                                     ③
    double mu = 0,                                                    ⑤
    double sigma = 1,                                                 ⑥
    bool lower = true,                                                ⑧
    bool log = false                                                  ⑦
);
Rcpp::NumericVector r_gumbel(
    unsigned int n,                                                   ④
```

```
    double mu = 0,                                                    ⑤
    double sigma = 1                                                  ⑥
);
```

① Argument to the density.
② Argument to the CDF.
③ Argument to the quantile function.
④ Number of draws.
⑤ Location parameter.
⑥ Scale parameter.
⑦ Logical; if `true`, probability arguments and return values are on the log-scale
⑧ Logical; if `true`, probability arguments and return values are lower tailed in the form $P[X \leq x]$; otherwise, $P[X > x]$.

5 Example: Von Mises Fisher Distribution

Let us consider generating from the von Mises Fisher (VMF) distribution as in Raim et al. (2025). VMF is useful in modeling directional data whose support is the sphere $\mathbb{S}^{d-1} = \{\boldsymbol{v} \in \mathbb{R}^d : \boldsymbol{v}^\top \boldsymbol{v} = 1\}$ (e.g., Mardia and Jupp 1999). A random variable $\boldsymbol{V}$ with distribution $\text{VMF}_d(\boldsymbol{\mu}, \kappa)$ has density

$$f_{\text{VMF}}(\boldsymbol{v}) = \frac{\kappa^{d/2-1}}{(2\pi)^{d/2} I_{d/2-1}(\kappa)} \exp(\kappa \cdot \boldsymbol{\mu}^\top \boldsymbol{v}) \cdot \text{I}\{\boldsymbol{v} \in \mathbb{S}^{d-1}\},$$

with modified Bessel function of the first kind $I_\nu(x) = \sum_{m=0}^{\infty} \{m! \cdot \Gamma(m + \nu + 1)\}^{-1} (\frac{x}{2})^{2m+\nu}$ and gamma function $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$. Parameters $\boldsymbol{\mu} \in \mathbb{S}^{d-1}$ and $\kappa > 0$ determine the orientation on the sphere and the concentration, respectively. First consider $\boldsymbol{\mu}_0 = (1, 0, \ldots, 0)$. A random variable $\boldsymbol{V}_0 \sim \text{VMF}_d(\boldsymbol{\mu}_0, \kappa)$ can be obtained from the transformation

$$\boldsymbol{V}_0 = \left(X, \sqrt{1 - X^2} \cdot \boldsymbol{U}\right), \tag{8}$$

where $\boldsymbol{U}$ is a uniform random variable on the sphere $\mathbb{S}^{d-2}$ and X has density

$$f(x) = \frac{(\kappa/2)^{d/2-1}(1 - x^2)^{(d-3)/2} \exp(\kappa x)}{\sqrt{\pi} \cdot I_{d/2-1}(\kappa) \cdot \Gamma((d-1)/2)} \cdot \text{I}(-1 < x < 1). \tag{9}$$

To obtain a draw from $\boldsymbol{V} \sim \text{VMF}_d(\boldsymbol{\mu}, \kappa)$ with an arbitrary $\boldsymbol{\mu}$, we can rotate $\boldsymbol{V} = \boldsymbol{Q}\boldsymbol{V}_0$ using an orthonormal matrix $\boldsymbol{Q}$ whose first column is $\boldsymbol{\mu}$. To generate from $\boldsymbol{V}_0$ in (8), we may draw $\boldsymbol{U} = \boldsymbol{Z}/\sqrt{\boldsymbol{Z}^\top \boldsymbol{Z}}$ from $\boldsymbol{Z} \sim \text{N}(\boldsymbol{0}, \boldsymbol{I}_{d-1})$ and X independently from (9). In the following, we consider the use of VWS to draw the univariate random variable X. Before proceeding, we identify a useful distribution.

Definition 5.1. Denote $X \sim \text{Exp}_{(a,b)}(\kappa)$ as a "doubly truncated" Exponential random variable with density

$$g(x) = \frac{\kappa e^{\kappa x}}{e^{\kappa b} - e^{\kappa a}} \cdot \text{I}(a < x < b),$$

where $-\infty < a < b < \infty$ and rate κ may be any real number. The CDF and quantile function corresponding to g are

$$G(x) = \frac{e^{\kappa x} - e^{\kappa a}}{e^{\kappa b} - e^{\kappa a}}\cdot, \quad x \in (a, b),$$
$$G^{-1}(\varphi) = \frac{1}{\kappa}\log\left[e^{\kappa a} + \varphi(e^{\kappa b} - e^{\kappa a})\right], \quad \varphi \in [0, 1].$$

■

Returning to target (9), let us decompose f into

$$f(x) \propto \underbrace{(1-x^2)^{(d-3)/2}}_{w(x)}\underbrace{\exp(\kappa x)\cdot \mathrm{I}(-1 < x < 1)}_{g_0(x)},$$

excluding terms from the normalizing constant, so that w is the weight function and g_0 is proportional to the density of $\mathrm{Exp}_{(-1,1)}(\kappa)$,

$$g(x) = \frac{\kappa e^{\kappa x}}{e^{\kappa} - e^{-\kappa}}\cdot \mathrm{I}(-1 < x < 1).$$

Section 5.1 obtains a VWS sampler with this decomposition using a constant majorizer. Section 5.2 replaces the default numerical optimization with custom code, which reduces the amount of computational overhead. Section 5.3 considers a linear majorizer which is substantially more involved but also obtains substantially lower rejection rates with a moderate number of regions. Codes for this example are in the folder `examples/vmf`. C++ functions for the $\mathrm{Exp}_{(a,b)}(\kappa)$ distribution from Definition 5.1 are given in the file `examples/vmf/texp.h`.

Remark 5.1. A caveat of this decomposition is that, in the $d < 3$ case, $w(x) \to \infty$ as x approaches ± 1. One way to avoid this is by truncating the support to $(\alpha_0, \alpha_N] = (-1+\epsilon, 1-\epsilon]$ for a small $\epsilon > 0$. Rejection sampling can proceed using the truncated support if the exclusion of $(-1, -1+\epsilon] \cup (1-\epsilon, 1]$ is known to have a negligible impact on the result. Otherwise, Raim et al. (2025) mention another strategy where the support is initially truncated and gradually expanded as rejections are encountered. In this document, we assume $(\alpha_0, \alpha_N] = (-1, 1]$ for $d > 3$ and a fixed truncation $(\alpha_0, \alpha_N] = (-1+\epsilon, 1-\epsilon]$ for $d < 3$.

5.1 Constant Majorizer with Numerical Optimization

We now give our first example demonstrating the `vws` package. We consider a constant majorizer for w which uses the default numerical optimization routine to identify appropriate constants $\overline{w}_j$ and $\underline{w}_j$. Because this is our first example, the source file for the sampler (`examples/vmf/vmf-v1.cpp`) is displayed in its entirety as follows.

```
1 // [[Rcpp::depends(vws, fntl)]]                                    ①
2 #include "vws.h"                                                  ②
3 #include "texp.h"                                                 ③
4
```

```cpp
// [[Rcpp::export]]                                                    ④
Rcpp::List r_vmf_pre_v1(unsigned int n, double kappa, double d,
        unsigned int N, double tol = 0, unsigned int max_rejects = 10000,
        unsigned int report = 10000)
{
    vws::rejection_args args;                                          ⑤
    args.max_rejects = max_rejects;
    args.report = report;

    const vws::dfdb& w =                                               ⑥
    [&](double x, bool log = true) {
        double out = R_NegInf;
        if (std::fabs(x) < 1){
            out = 0.5 * (d - 3) * std::log1p(-std::pow(x, 2));
        }
        return log ? out : std::exp(out);
    };

    fntl::density df = [&](double x, bool log = false) {               ⑦
        return d_texp(x, kappa, -1, 1, log);
    };
    fntl::cdf pf = [&](double q, bool lower = true, bool log = false) {
        return p_texp(q, kappa, -1, 1, lower, log);
    };
    fntl::quantile qf = [&](double p, bool lower = true, bool log = false) {
        return q_texp(p, kappa, -1, 1, lower, log);
    };

    vws::UnivariateHelper helper(df, pf, qf);                          ⑧
    vws::RealConstRegion supp(-1, 1, w, helper);                       ⑨
    vws::FMMProposal<double, vws::RealConstRegion> h(supp);            ⑩

    auto lbdd = h.refine(N - 1, tol);                                  ⑪
    auto out = vws::rejection(h, n, args);                             ⑫

    return Rcpp::List::create(                                         ⑬
        Rcpp::Named("draws") = out.draws,
        Rcpp::Named("rejects") = out.rejects,
        Rcpp::Named("lbdd") = lbdd
    );
}
```

① Ensure that this code links with the `vws` and `fntl` packages during compilation.
② Include the header for C++ framework in `vws`.
③ Include `texp.h`, which provides functions described in Definition 5.1.
④ Define a C++ function which invokes the sampler and export it for use in R.

⑤ Prepare a struct with extra arguments for rejection sampling.
⑥ Define the weight function as a lambda. We are careful to avoid `nan` values that can occur when $x = \pm 1$. Computations are carried out on the log-scale to avoid numerical loss of precision.
⑦ Specify the density, CDF, and quantile function of the base distribution $\text{Exp}_{(-1,1)}(\kappa)$. The density, CDF, and quantile function types are defined in Section 4.1.
⑧ Create a "helper" object as a container for the distribution functions.
⑨ Construct a `RealConstRegion` which contains all problem-specific logic of the sampler. We construct one initial region which contains the entire support $(-1, 1]$.
⑩ Construct an `FMMProposal` based on our initial region `supp`. The first template argument specifies that the data type of the support is `double`; the second specifies that regions (which include the logic of the sampler) are of type `RealConstRegion`.
⑪ Request the proposal `h` to refine itself `N-1` times using Algorithm 1 so that there are `N` regions.
⑫ Carry out rejection sampling with proposal `h`.
⑬ Assemble an `Rcpp::List` to return to the caller. It contains the draws in element `draws`, a vector of rejection counts in `rejects` where the ith element represents the number of rejections for the ith draw, and a vector in element `lbdd` with the N bounds $\rho_+^{(1)}, \ldots, \rho_+^{(N)}$, where $\rho_+^{(j)}$ is the value achieved with j regions.

The name of the function `r_vmf_pre_v1` reflects that this is our first version of the sampler for target (9), which is a precursor to transformation (8) to obtain a VMF random variable. The following R snippet builds the C++ code and invokes the sampling function.

```
> Rcpp::sourceCpp("examples/vmf/vmf-v1.cpp")
> out1 = r_vmf_pre_v1(n = 1000, kappa = 5, d = 4, N = 50, tol = 0.10)
> head(out1$draws)
```

```
[1] 0.9056983 0.5062053 0.8663953 0.6000328 0.8686481 0.9627474
```

Figure 5.1 plots the bound for the rejection probability during the refinement process, which is captured in the variable `out$lbdd`. Figure 5.2 plots the empirical distribution of the draws and compares them to the density.

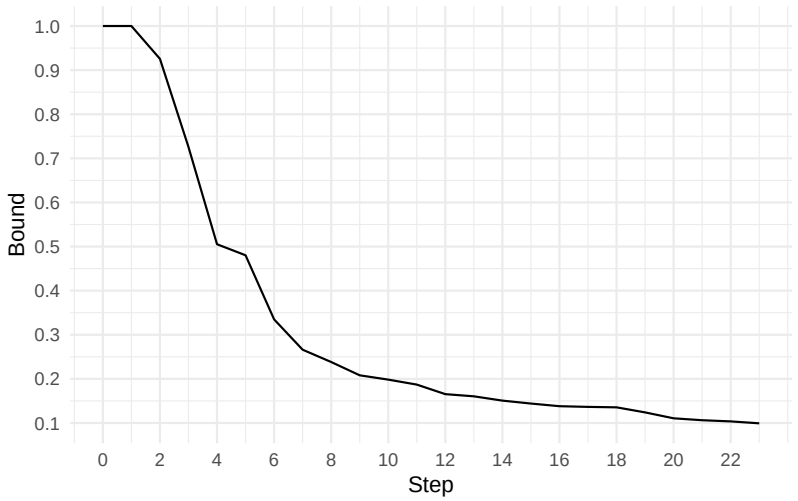


Figure 5.1: Refinement for VMF example with constant majorizer.

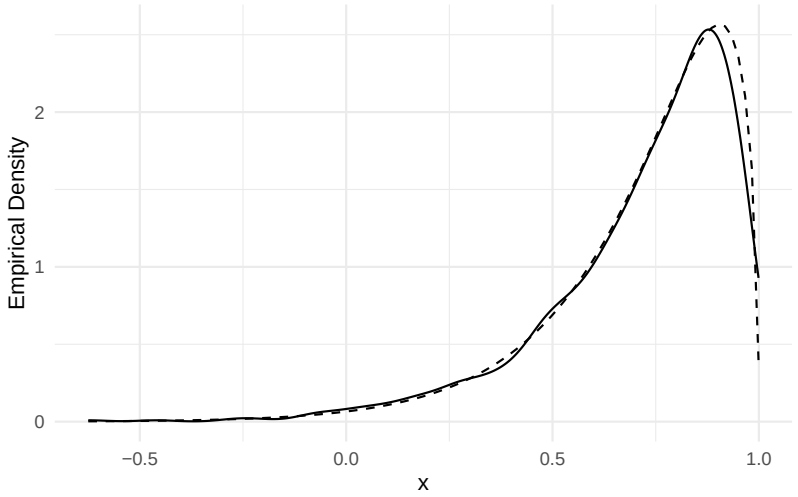


Figure 5.2: Empirical distribution of draws (solid) versus target (dashed) for VMF example with constant majorizer.

5.2 Constant Majorizer with Custom Optimization

It is not difficult to find the minimum and maximum of the function w on a given interval for this example. We can reduce some of the computational burden by providing functions to compute these two values.

We have

$$\log w(x) = \frac{d-3}{2}\log(1-x^2),$$
$$\frac{d}{dx}\log w(x) = -(d-3)\frac{x}{1-x^2},$$
$$\frac{d^2}{dx^2}\log w(x) = -(d-3)\frac{1+x^2}{(1-x^2)^2}.$$

Therefore, it is seen that $\log w(x)$ is concave when $d > 3$, convex when $d = 2$, and constant otherwise. When $d > 3$, $\frac{d}{dx}\log w(x)$ is positive for $x \in (-1, 0)$, negative for $x \in (0, 1)$, and has root $x = 0$; therefore, $\log w(x)$ is unimodal on $(-1, 1)$ with a maximum at $x = 0$. When $d = 2$, the point $x = 0$ is instead a minimum of $\log w(x)$. Finally, w is a constant in the case $d = 3$. The following code implements these maximization and minimization routines as lambdas.

```
vws::optimizer opt1 = [&](const vws::dfdb& w, double lo, double hi, bool log)
{
    double x = 0;
    if (hi < 0) {
        x = hi;
    } else if (lo > 0){
        x = lo;
    }
    double out = w(x, true);
    return log ? out : std::exp(out);
};
```

```
vws::optimizer opt2 = [&](const vws::dfdb& w, double lo, double hi, bool log)
{
    double w_lo = w(lo, true);
    double w_hi = w(hi, true);
    double out = std::min(w_lo, w_hi);
    return log ? out : std::exp(out);
};
```

The following snippet creates pointers `maxopt` and `minopt` for the maximizer and minimizer functions. The condition $d > 3$ is checked to determine which of `opt1` and `opt2` is the maximizer and which is the minimizer.

```
vws::optimizer* maxopt;
vws::optimizer* minopt;
if (d >= 3) {
    maxopt = &opt1;
    minopt = &opt2;
} else {
    minopt = &opt1;
```

```
    maxopt = &opt2;
}
```

Finally, we create the initial region `supp`. The `maxopt` and `minopt` functions are provided to the constructor as additional arguments. Note that we dereference our pointers and pass the function objects themselves (which are taken as references by the constructor).

```
vws::UnivariateHelper helper(df, pf, qf);
vws::RealConstRegion supp(-1, 1, w, helper, *maxopt, *minopt);
```

The following R snippet builds the C++ code and invokes the sampling function.

```
> Rcpp::sourceCpp("examples/vmf/vmf-v2.cpp")
> out2 = r_vmf_pre_v2(n = 1000, kappa = 5, d = 4, N = 50, tol = 0.10)
> head(out2$draws)
```

```
[1] 0.7548988 0.9020145 0.3765028 0.5477933 0.8755307 0.9491774
```

5.3 Linear Majorizer

We noted in Section 5.2 that w is log-convex when $d < 3$, log-concave when $d > 3$, and a constant otherwise. Therefore, we can majorize w with exponentiated linear functions of the form $\overline{w}_j(x) = \exp\{\overline{\beta}_{0j} + \overline{\beta}_{1j}x\}$. This yields the expression

$$\overline{\xi}_j = \int_{\alpha_{j-1}}^{\alpha_j} \overline{w}_j(x)g(x)dx = \frac{\kappa \exp\{\overline{\beta}_{0j}\}}{(\kappa + \overline{\beta}_{1j})(e^{\kappa\alpha_N} - e^{\kappa\alpha_0})} \left\{\exp\{(\kappa + \overline{\beta}_{1j})\alpha_j\} - \exp\{(\kappa + \overline{\beta}_{1j})\alpha_{j-1}\}\right\}.$$

The proposal h is then a finite mixture $h(x) = \sum_{j=1}^{N} \pi_j g_j(x)$ with

$$\begin{aligned} g_j(x) &= \overline{w}_j(x)g(x)\,\mathrm{I}\{x \in \mathscr{D}_j\}/\overline{\xi}_j \\ &= \frac{(\kappa + \overline{\beta}_{1j})\exp\{(\kappa + \overline{\beta}_{1j})x\}}{\exp\{(\kappa + \overline{\beta}_{1j})\alpha_j\} - \exp\{(\kappa + \overline{\beta}_{1j})\alpha_{j-1}\}} \cdot \mathrm{I}(\alpha_{j-1} < x \le \alpha_j), \end{aligned}$$

the density of $\mathrm{Exp}_{(\alpha_{j-1},\alpha_j]}(\kappa + \overline{\beta}_{1j})$. Remark 2.3 is used to select the expansion point c to determine coefficients for the majorizer in the log-concave case, with

$$M_j(s) = \int_{\alpha_{j-1}}^{\alpha_j} e^{sx}g(x)dx = \frac{e^{s\alpha_j} - e^{s\alpha_{j-1}}}{s(\alpha_j - \alpha_{j-1})}.$$

To compute (3), we assume the "trivial" minorizer $\underline{w}_j(x) = w(x)$ so that

$$\underline{\xi}_j = \int_{\alpha_{j-1}}^{\alpha_j} w(x)g(x)dx = \int_{\alpha_{j-1}}^{\alpha_j} \frac{(1-x^2)^{(d-3)/2}\kappa e^{\kappa x}}{e^{\kappa\alpha_N} - e^{\kappa\alpha_0}}dx,$$

which we compute using numerical integration. The proposal is implemented with the `vws` package by inheriting from the abstract `Region` base class (Section 4.3) and implementing each of the functions

using the expressions above. We name the resulting subclass `LinearVWSRegion`; its complete code is given in `examples/vmf/LinearVWSRegion.h`. The code in `examples/vmf/vmf-v3.cpp` instantiates a proposal with regions of type `LinearVWSRegion`, invokes rejection sampling, and returns an `Rcpp::List` with the results. This code is displayed below.

```
// [[Rcpp::depends(vws, fntl)]]
#include "vws.h"
#include "LinearVWSRegion.h"

// [[Rcpp::export]]
Rcpp::List r_vmf_pre_v3(unsigned int n, double kappa, double d, unsigned int N,
    double tol = 0, unsigned int max_rejects = 10000,
    unsigned int report = 10000)
{
    vws::rejection_args args;
    args.max_rejects = max_rejects;
    args.report = report;

    LinearVWSRegion supp(-1, 1, kappa, d);
    vws::FMMProposal<double, LinearVWSRegion> h(supp);

    auto lbdd = h.refine(N - 1, tol);
    auto out = vws::rejection(h, n, args);

    return Rcpp::List::create(
        Rcpp::Named("draws") = out.draws,
        Rcpp::Named("rejects") = out.rejects,
        Rcpp::Named("lbdd") = lbdd
    );
}
```

The following R snippet builds the C++ code and invokes the sampling function.

```
> Rcpp::sourceCpp("examples/vmf/vmf-v3.cpp")
> out3 = r_vmf_pre_v3(n = 1000, kappa = 5, d = 4, N = 50, tol = 0.01)
> head(out3$draws)
```

```
[1] 0.4913948 0.7346177 0.8208684 0.8185937 0.3202652 0.9625593
```

Remark 5.2. This code will fail if we attempt to use it with $d = 2$ as described in Remark 5.2. Here, $\log w(x) \to \infty$ as $x \to \pm 1$ so that the first and last regions cannot be bounded by an exponentiated linear form. This can be averted by truncating the support to $(-1 + \epsilon, 1 - \epsilon]$ for a small $\epsilon > 0$. ■

Figure 5.2 plots the empirical distribution of the draws and compares them to the density. Figure 5.4 plots the bound for the rejection probability for this sampler after each step of refining, along with

those from the previous two versions. A substantial improvement in efficiency is seen here; fewer regions are needed to achieve a small rejection probability. Although versions 1 and 2 of the sampler are based on the same proposal, the changes to their bounds are seen to differ due to the randomness in knot selection.

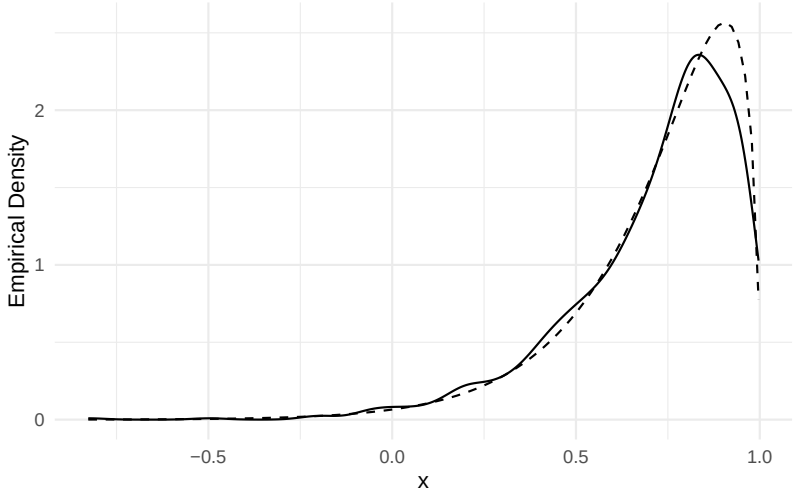


Figure 5.3: Empirical distribution of draws (solid) versus target (dashed) for VMF example with linear majorizer.

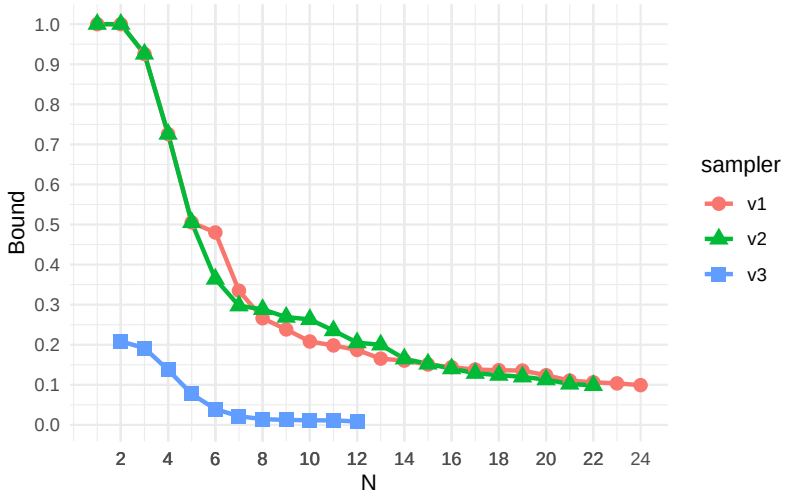


Figure 5.4: Refinement for VMF example with constant majorizer.

6 Example: Lognormal-Normal Conditional Distribution

Suppose an official statistics agency collects sensitive data from respondents in a survey or census. The agency then seeks to disseminate a tabulation to data users while protecting the privacy of the respondents. The agency can consider adding random noise γ to the tabulation Y and releasing

$Z = Y + \gamma$. This setting is the basis of a modeling scenario considered by Raim (2021), Irimata et al. (2022), and Janicki et al. (2025+) where the distribution for γ has been selected by the agency and the data have been disseminated; the objective in these works is to carry out inference on the unobserved Y given observed $Z = z$. The field of differential privacy studies the design of noise mechanisms (including distributions of γ in the present setting) to satisfy mathematical criteria for privacy (e.g., Dwork and Roth 2014). Abowd et al. (2022) describe work by the U.S. Census Bureau to implement differential privacy in the release of data from the decennial census. Our present motivation is to consider a simple but nontrivial sampling problem that arises in analysis of the released z.

Suppose Y and γ are independently distributed with $Y \sim \text{Lognormal}(\mu, \sigma^2)$ and $\gamma \sim \text{N}(0, \lambda^2)$. The variance λ^2 is assumed to be known and provided with the noisy data. Such transparency into the noise mechanism is often featured in differential privacy.

Suppose the target distribution is the conditional $[Y \mid Z = z, \mu, \sigma^2]$ whose density is given by

$$\begin{aligned} f(y \mid z, \mu, \sigma^2) &\propto \frac{1}{\lambda\sqrt{2\pi}} \exp\left\{-\frac{1}{2\lambda^2}(z-y)^2\right\} \cdot \frac{1}{y\sigma\sqrt{2\pi}} \exp\left\{-\frac{1}{2\sigma^2}(\log y - \mu)^2\right\} \text{I}(y > 0) \\ &\propto \underbrace{\frac{1}{\lambda\sqrt{2\pi}} \exp\left\{-\frac{1}{2\lambda^2}(z-y)^2\right\}}_{g(y)} \cdot \underbrace{\frac{1}{y} \exp\left\{-\frac{1}{2\sigma^2}(\log y - \mu)^2\right\} \text{I}(y > 0)}_{w(y)} . \end{aligned} \tag{10}$$

We have decomposed f into weight function $w(y) = \frac{1}{y} \exp\left\{-\frac{1}{2\sigma^2}(\log y - \mu)^2\right\} \text{I}\{y \in (0, \infty)\}$, from the Lognormal component—dropping some of the terms in its normalizing constant—and base distribution whose density g is $\text{N}(z, \lambda^2)$. The conditional $[\boldsymbol{y} \mid \sigma^2, \boldsymbol{\mu}, \boldsymbol{z}]$ in (10) would be encountered in a Gibbs sampler for the posterior distribution $[\boldsymbol{y}, \boldsymbol{\mu}, \sigma^2 \mid \boldsymbol{z}]$, based on an observed sample $\boldsymbol{z} = (z_1, \ldots, z_n)$ with augmented data $\boldsymbol{y} = (y_1, \ldots, y_n)$. The remaining conditionals $[\boldsymbol{\mu} \mid \sigma^2, \boldsymbol{y}, \boldsymbol{z}]$ and $[\sigma^2 \mid \boldsymbol{\mu}, \boldsymbol{y}, \boldsymbol{z}]$ may be straightforward to generate if a convenient prior distribution is selected; therefore, we will focus on (10).

Before proceeding, let us fix the following values for the parameters.

```
> mu = 5
> sigma = sqrt(0.5)
> lambda = 10
```

Jointly draw values of Y and Z from the model; Z will be observed by users of the data while Y is unobserved and the objective for inference.

```
> source("examples/ln-norm/functions.R")
> y_true = rlnorm(1, mu, sigma)
> z = rnorm(1, y_true, lambda)
> vws::printf("y_true = %g and z = %g\n", y_true, z)
```

```
y_true = 57.9052 and z = 62.9898
```

Coding the target density is helpful to evaluate the distribution of the draws. To compute the normalizing constant, we use Hermite quadrature via the `gauss.quad` function in the `statmod` package (Smyth 2005). The integral $\psi = \int_{-\infty}^{\infty} q(x)e^{-x^2}dx$ is approximated as $\psi \approx \sum_{j=1}^{Q} \omega_j q(x_j)$ using quadrature points $x_1, \ldots, x_Q$ and weights $\omega_1, \ldots, \omega_Q$; to identify the function q, we have

$$\begin{aligned}
\psi &= \int_{-\infty}^{\infty} w(y)g(y)dy \\
&= \int_{-\infty}^{\infty} \mathrm{I}\{y > 0\} \cdot \frac{1}{y} \exp\left\{-\frac{1}{2\sigma^2}(\log y - \mu)^2\right\} \frac{1}{\lambda\sqrt{2\pi}} \exp\left\{-\frac{1}{2\lambda^2}(z-y)^2\right\} dy \\
&= \int_{-\infty}^{\infty} \mathrm{I}\left(z > \sqrt{2}\lambda x\right) \cdot \frac{1}{z - \sqrt{2}\lambda x} \exp\left\{-\frac{1}{2\sigma^2}\left[\log\left(z - \sqrt{2}\lambda x\right) - \mu\right]^2\right\} \frac{1}{\sqrt{\pi}} e^{-x^2} dx,
\end{aligned}$$

using the transformation $y = z - \sqrt{2}\lambda x$, so that

$$q(x) = \mathrm{I}\left(z > \sqrt{2}\lambda x\right) \cdot \frac{1}{z - \sqrt{2}\lambda x} \exp\left\{-\frac{1}{2\sigma^2}\left[\log\left(x - \sqrt{2}\lambda x\right) - \mu\right]^2\right\} \frac{1}{\sqrt{\pi}}$$

is used with `gauss.quad`.

We will consider three variations of a VWS sampler, progressing from easier-to-implement to more computationally efficient. Section 6.1 considers a constant majorizer where the constant for each region is obtained by numerical optimization. Section 6.2 replaces numerical optimization with code for a closed-form solution. Section 6.3 makes use of a linear majorizer which subclasses the abstract `Region` class. All codes for this example are in the folder `examples/ln-norm`. The function `d_target` defined in `examples/ln-norm/functions.R` computes the target density.

6.1 Constant Majorizer with Numerical Optimization

We may adapt the code from Section 5.1 to the target density in the present problem. The present weight function may be coded as follows.

```
const vws::dfdb& w = [&](double x, bool log = true) {
    double out = R_NegInf;
    if (x > 0) {
        double sigma2 = std::pow(sigma, 2)
        out = -std::log(x) - std::pow(std::log(x) - mu, 2) / (2 * sigma2);
    }
    return log ? out : std::exp(out);
};
```

The base distribution's density, CDF, and quantile function are coded as lambdas and packaged into a `UnivariateHelper` object. Here we can make use of the implementations of the normal distribution in R's API.

```
fntl::density df = [&](double x, bool log = false) {
    return R::dnorm(x, z, lambda, log);
};
```

```
fntl::cdf pf = [&](double q, bool lower = true, bool log = false) {
    return R::pnorm(q, z, lambda, lower, log);
};
fntl::quantile qf = [&](double p, bool lower = true, bool log = false) {
    return R::qnorm(p, z, lambda, lower, log);
};

vws::UnivariateHelper helper(df, pf, qf);
```

The following R snippet builds the C++ code and invokes the sampling function.

```
> Rcpp::sourceCpp("examples/ln-norm/ln-norm-v1.cpp")
> out1 = r_ln_norm_v1(n = 1000, z, mu, sigma, lambda, N = 50, tol = 0.10)
> head(out1$draws)
```

```
[1] 63.80234 73.74232 63.03024 55.11748 65.16079 57.30657
```

Figure 6.1 plots the bound for the rejection probability during the refinement process, which is captured in the variable `out$lbdd`. Figure 6.2 plots the empirical distribution of the draws and compares them to the density. It displays an interval based on the 0.025 and 0.975 quantiles of the distribution $[Y \mid Z = z]$ approximated from the empirical quantiles of the draws. The value of the observed z and the latent y are also highlighted for reference.

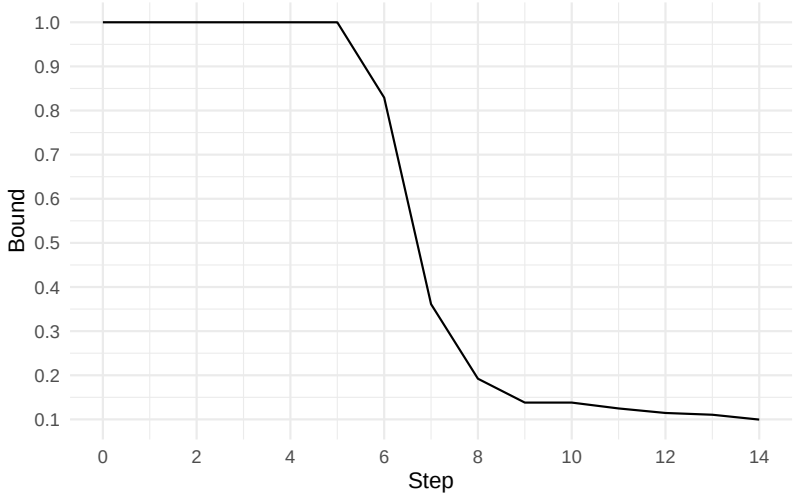


Figure 6.1: Refinement for lognormal-normal example with constant majorizer.

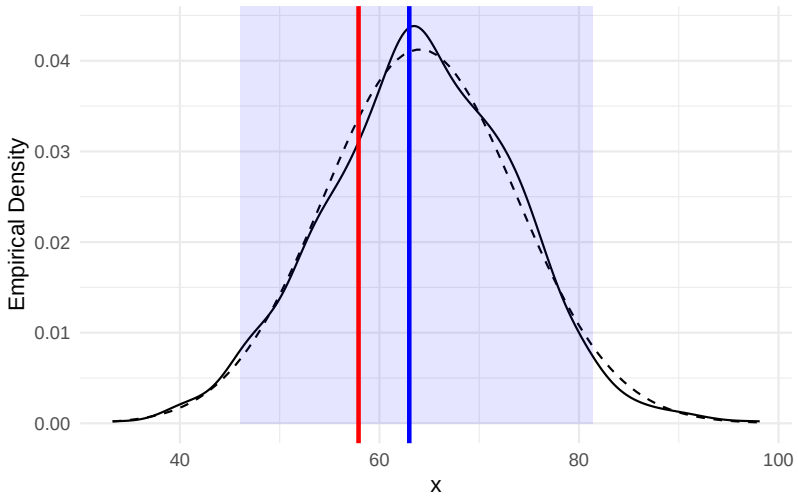


Figure 6.2: Empirical distribution of draws (solid black) versus target (dashed black) for lognormal-normal example with constant majorizer. Observed value of z (blue line) and latent value of y (red line) are displayed along with a 95% interval (blue ribbon) based on draws from $[Y \mid Z = z]$.

6.2 Constant Majorizer with Custom Optimization

The log-weight function can be both maximized and minimized in closed form. Coding it explicitly reduces computational overhead and avoids convergence issues from numerical optimization. This section will demonstrate how to override the default `optimize` method of `RealConstRegion`.

We have first derivative

$$\frac{d}{dy} \log w(y) = -\frac{1}{y}\left(1 + \frac{\log y - \mu}{\sigma^2}\right),$$

for $y \in (0, \infty)$. Let $y^* = \exp(\mu - \sigma^2)$; it is seen that $\frac{d}{dy} \log w(y)$ is positive when $y < y^*$, negative when $y > y^*$, and takes value zero at $y = y^*$. Therefore, $\log w(y)$ is unimodal and y^* maximizes $\log w(y)$ with

$$\begin{aligned}
\log w(y^*) &= -\log[\exp(\mu - \sigma^2)] - \frac{(\log[\exp(\mu - \sigma^2)] - \mu)^2}{2\sigma^2} \\
&= -\mu + \sigma^2 - \frac{(\sigma^2)^2}{2\sigma^2} \\
&= -\mu + \sigma^2/2.
\end{aligned}$$

Therefore, on a region $\mathscr{D}_j = (\alpha_{j-1}, \alpha_j]$ where both endpoints are smaller than y^*, the maximum of $\log w(y)$ is $\log w(\alpha_j)$ and the minimum is $\log w(\alpha_{j-1})$. On the other hand, for a region where both endpoints are larger than y^*, the maximum of $\log w(y)$ is $\log w(\alpha_{j-1})$ and the minimum is $\log w(\alpha_j)$. Figure 6.3 displays a plot of $\log w(y)$ with our selected μ and σ^2 values.

```
> y_star = exp(mu - sigma^2)
> w_star = -mu + sigma^2 / 2
> vws::printf("Maximizer y = %g obtains value log w(%g) = %g.\n",
+   y_star, y_star, w_star)
```

```
Maximizer y = 90.0171 obtains value log w(90.0171) = -4.75.
```

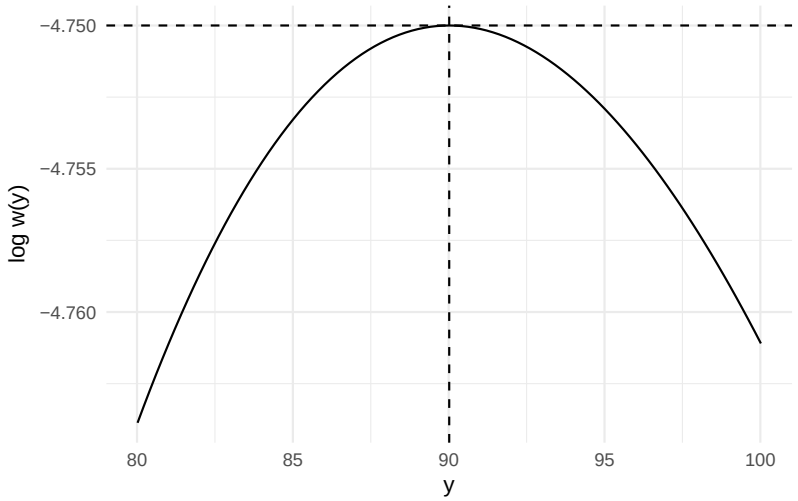


Figure 6.3: Weight function for Lognormal-Normal example on the log-scale, with the maximizer $y^* = \exp(\mu - \sigma^2)$ highlighted.

The maximizer may be coded as follows.

```
const vws::optimizer& maxopt = [&](const vws::dfdb& w, double lo,
    double hi, bool log)
{
    double y_star = exp(mu - sigma2);

    double y = y_star;
    if (y_star > hi) {
        y = hi;
    } else if (y_star < lo) {
        y = lo;
    }

    double out = w(y, true);
    return log ? out : exp(out);
};
```

Here is code for the minimizer.

```
const vws::optimizer& minopt = [&](const vws::dfdb& w, double lo,
    double hi, bool log)
{
    double lwa = w(lo, true);
    double lwb = w(hi, true);
    double out = std::min(lwa, lwb);
    return log ? out : exp(out);
};
```

We can construct the proposal using a single `RealConstRegion` that represents the support; the `maxopt` and `minopt` arguments are specified here.

```
vws::RealConstRegion supp(0, R_PosInf, w, helper, maxopt, minopt);
```

The following R snippet builds the C++ code and invokes the sampling function.

```
> Rcpp::sourceCpp("examples/ln-norm/ln-norm-v2.cpp")
> out2 = r_ln_norm_v2(n = 1000, z, mu, sigma, lambda, N = 50, tol = 0.10)
> head(out2$draws)
```

```
[1] 65.52465 54.89391 70.52306 53.60235 79.18636 69.74638
```

6.3 Linear Majorizer

We can obtain a linear majorizer by noting the convexity of $\log w(y)$. Its second derivative is seen to be

$$\frac{d^2}{dy^2} \log w(y) = -\frac{1}{y^2}\left(1 + \frac{\log y - \mu - 1}{\sigma^2}\right),$$

which has root $y_0 = \exp\{\mu - \sigma^2 + 1\}$. The weight function w is log-concave for $y < y_0$ and log-convex for $y > y_0$. This is plotted in Figure 6.4. We will assume that there are two initial regions $\mathscr{D}_1 = (0, y_0]$ and $\mathscr{D}_2 = (y_0, \infty]$ so that all partitions considered thereafter consist of regions on which $\log w(y)$ is entirely concave or convex.

```
> y0 = exp(mu - sigma^2 + 1)
> printf("Convexity changes at y = %g.\n", y0)
```

```
Convexity changes at y = 244.692.
```

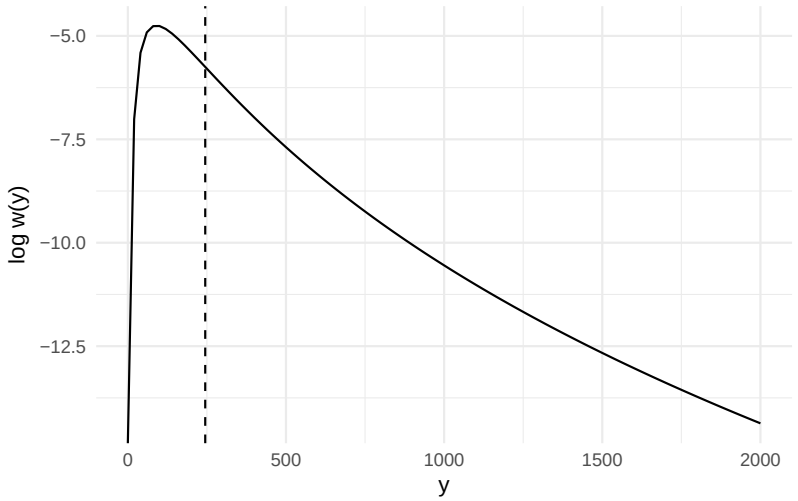


Figure 6.4: Weight function for Lognormal-Normal example on the log-scale, highlighting $y_0 = \exp(\mu - \sigma^2 + 1)$ where there is a change in convexity.

As in Section 5.3, we can majorize w with exponentiated linear functions of the form $\overline{w}_j(y) = \exp\{\overline{\beta}_{0j} + \overline{\beta}_{1j} y\}$. Here, we also assume a minorizer of the form $\underline{w}_j(y) = \exp\{\underline{\beta}_{0j} + \underline{\beta}_{1j} y\}$. Before proceeding, the following remark gives an integral that will be used several times.

Remark 6.1. Let $\phi(\cdot \mid \mu, \sigma^2)$ and $\Phi(\cdot \mid \mu, \sigma^2)$ be the density and CDF of $Y \sim \mathrm{N}(\mu, \sigma^2)$, respectively. If $a < b$ are scalars (possibly infinite), then

$$\begin{aligned}\int_a^b e^{ty} \phi(y \mid \mu, \sigma^2) dy &= \exp(\mu t + t^2\sigma^2/2) \left\{ \Phi(b \mid \mu + t\sigma^2, \sigma^2) - \Phi(a \mid \mu + t\sigma^2, \sigma^2) \right\} \\ &= \exp(\mu t + t^2\sigma^2/2)\, \mathrm{P}(a < T \leq b).\end{aligned}$$

where $T \sim \mathrm{N}(\mu + t\sigma^2, \sigma^2)$. The special case $a = -\infty$ and $b = \infty$ yields the moment-generating function $M(t) = \exp(\mu t + t^2\sigma^2/2)$ of Y. ■

Remark 6.1 yields the expressions

$$\begin{aligned}\overline{\xi}_j &= \int_{\alpha_{j-1}}^{\alpha_j} \overline{w}_j(y) g(y) dy \\ &= \exp\{\overline{\beta}_{0j}\} \int_{\alpha_{j-1}}^{\alpha_j} \exp\{\overline{\beta}_{1j} y\} \frac{1}{\lambda\sqrt{2\pi}} \exp\left\{ -\frac{1}{2\lambda^2}(y - z)^2 \right\} dy \\ &= \exp\left\{ \overline{\beta}_{0j} + z\overline{\beta}_{1j} + \frac{1}{2}\overline{\beta}_{1j}^2 \lambda^2 \right\} \mathrm{P}(\alpha_{j-1} < \overline{T} \leq \alpha_j),\end{aligned}$$

where $\overline{T} \sim \mathrm{N}(z + \overline{\beta}_{1j}\lambda^2, \lambda^2)$, and similarly,

$$\underline{\xi}_j = \exp\left\{ \underline{\beta}_{0j} + z\underline{\beta}_{1j} + \frac{1}{2}\underline{\beta}_{1j}^2 \lambda^2 \right\} \mathrm{P}(\alpha_{j-1} < \underline{T} \leq \alpha_j),$$

where $\underline{T} \sim \mathrm{N}(z + \underline{\beta}_{1j}\lambda^2, \lambda^2)$. The proposal h is a finite mixture $h(y) = \sum_{j=1}^{N} \pi_j g_j(y)$ with

$$\begin{aligned} g_j(y) &= \overline{w}_j(y) g(y) \, \mathrm{I}\{y \in \mathscr{D}_j\} / \overline{\xi}_j \\ &= \exp\{\overline{\beta}_{0j} + \overline{\beta}_{1j} y\} \frac{1}{\lambda\sqrt{2\pi}} \exp\left\{-\frac{1}{2\lambda^2}(y - z)^2\right\} \cdot \mathrm{I}(\alpha_{j-1} < y \le \alpha_j) / \overline{\xi}_j \\ &= \frac{\frac{1}{\lambda\sqrt{2\pi}} \exp\left\{-\frac{1}{2\lambda^2}(y - z - \lambda^2\overline{\beta}_{1j})^2\right\}}{\mathrm{P}(\alpha_{j-1} < \overline{T} \le \alpha_j)} \cdot \mathrm{I}(\alpha_{j-1} < y \le \alpha_j), \end{aligned}$$

which is the density of $\mathrm{N}(z + \lambda^2\overline{\beta}_{1j}, \lambda^2)$ truncated to the interval $(\alpha_{j-1}, \alpha_j]$. Remark 2.3 is used to select the expansion point c to determine coefficients for the majorizer in the log-concave case and the minorizer in the log-convex case, with

$$M_j(s) = \exp(zs + s^2\lambda^2/2) \frac{\Phi(\alpha_j \mid z + s\lambda^2, \lambda^2) - \Phi(\alpha_{j-1} \mid z + s\lambda^2, \lambda^2)}{\Phi(\alpha_j \mid z, \lambda^2) - \Phi(\alpha_{j-1} \mid z, \lambda^2)}.$$

Adapting the code from Section 5.3 to the present problem, we implement `LinearVWSRegion` as a subclass of the abstract `Region` base class (Section 4.3). See the file `examples/ln-norm/LinearVWSRegion.h`. The code in `examples/ln-norm/ln-norm-v3.cpp` instantiates a proposal from this region type and proceeds with rejection sampling. This code is displayed below.

```
// [[Rcpp::depends(vws, fntl)]]
#include "vws.h"
#include "LinearVWSRegion.h"

// [[Rcpp::export]]
Rcpp::List r_ln_norm_v3(unsigned int n, double z, double mu, double sigma,
    double lambda, double lo, double hi, unsigned int N, double tol = 0,
    unsigned int max_rejects = 10000, unsigned int report = 10000)
{
    vws::rejection_args args;
    args.max_rejects = max_rejects;
    args.report = report;

    // Initially partition at y0 where convexity of the weight function changes
    double y0 = exp(mu - std::pow(sigma, 2) + 1);
    LinearVWSRegion r1(lo, y0, z, mu, sigma, lambda);
    LinearVWSRegion r2(y0, hi, z, mu, sigma, lambda);
    vws::FMMProposal<double, LinearVWSRegion> h({r1, r2});

    auto lbdd = h.refine(N - 2, tol);
    auto out = vws::rejection(h, n, args);

    return Rcpp::List::create(
        Rcpp::Named("draws") = out.draws,
        Rcpp::Named("rejects") = out.rejects,
        Rcpp::Named("lbdd") = lbdd
```

```
    );
}
```

The following R snippet builds the C++ code and invokes the sampling function.

```
> Rcpp::sourceCpp("examples/ln-norm/ln-norm-v3.cpp")
> out3 = r_ln_norm_v3(n = 1000, z, mu, sigma, lambda, lo = 1e-8, 1e8,
+   N = 50, tol = 0.10)
> head(out3$draws)
```

```
[1] 58.33451 76.48139 65.70251 76.28704 70.20251 70.24626
```

Figure 6.5 plots the empirical distribution of the draws and compares them to the density. Figure 6.6 plots the bound for the rejection probability for this sampler after each step of refining, along with those from the previous two versions. A substantial improvement in efficiency is seem here; fewer regions are needed to achieve a small rejection probability. Although versions 1 and 2 of the sampler are based on the same proposal, the changes to their bounds are seen to differ due to randomness in knot selection.

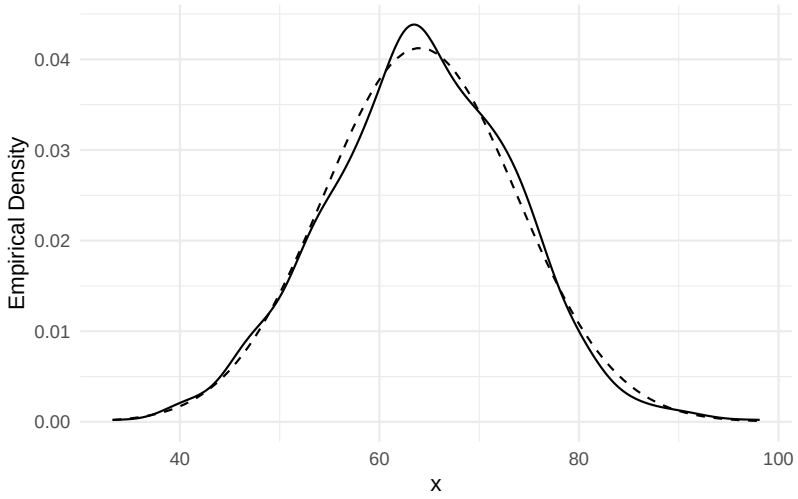


Figure 6.5: Empirical distribution of draws (solid) versus target (dashed) for lognormal-normal example with constant majorizer.

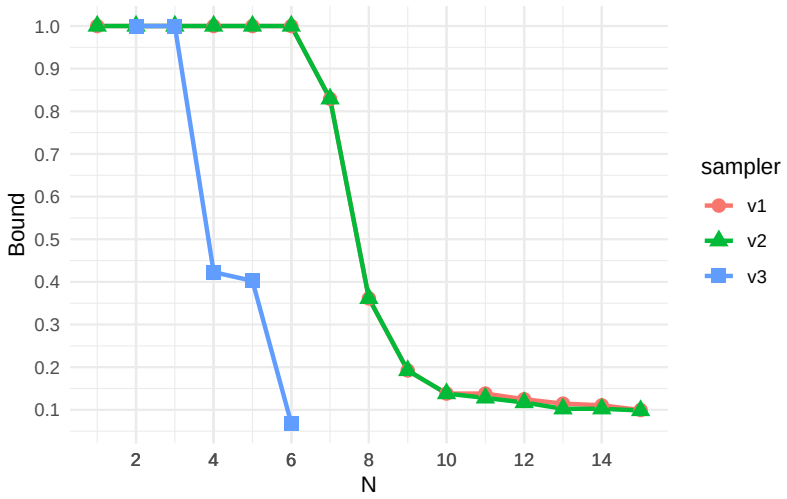


Figure 6.6: Refinement for lognormal-normal example with constant majorizer.

7 Example: Bessel Count Distribution

Where Sections 5 and 6 considered continuous distributions, we now generate from a discrete target. The Bessel distribution (Yuan and Kalbfleisch 2000) has density

$$f(x) = \frac{(\lambda/2)^{2x+\nu}}{I_v(\lambda) \cdot x! \cdot \Gamma(x+\nu+1)} \cdot \mathrm{I}\{x \in \mathbb{N}\}, \quad \nu > -1, \quad \lambda > 0,$$

where $\mathbb{N} = \{0, 1, 2, \ldots\}$ is the set of nonnegative integers and where

$$I_v(\lambda) = \sum_{x=0}^{\infty} \frac{(\lambda/2)^{2x+\nu}}{x! \cdot \Gamma(x+\nu+1)}$$

is a modified Bessel function of the first kind. Let $X \sim \text{Bessel}(\lambda, \nu)$ denote a random variable with density f. Devroye (2002) develops a rejection sampling method to generate draws of X using properties of the distribution. Here we implement several VWS samplers which do not require as much insight. Our approach is to decompose the density into

$$\begin{aligned} f(x) &= \frac{(\lambda/2)^{2x+\nu}}{I_v(\lambda) \cdot x! \cdot \Gamma(x+\nu+1)} \cdot \mathrm{I}\{x \in \mathbb{N}\} \\ &\propto \underbrace{\frac{1}{\Gamma(x+\nu+1)}}_{w(x)} \underbrace{\frac{(\lambda/2)^{2x} e^{-\lambda^2/4}}{x!} \mathrm{I}\{x \in \mathbb{N}\}}_{g(x)}, \end{aligned}$$

disregarding several terms in the normalizing constant, so that g is the density of a Poisson$(\lambda^2/4)$ distribution and the weight function is specified by $\log w(x) = -\log \Gamma(x+\nu+1)$.

Before proceeding, let us fix values of λ and ν. Let us also load some useful R functions from the script `examples/bessel/bessel.R`; primarily, `d_bessel` which computes the target density.

```
> source("examples/bessel/bessel.R")
> lambda = 10
> nu = 2
```

Three variations of VWS samplers are considered in subsections. Section 7.1 uses a constant majorizer obtained by numerical optimization. Section 7.2 replaces numerical optimization with a closed-form solution. Section 7.3 uses a linear majorizer and subclasses the abstract `Region` class. All codes for this example are in the folder `examples/bessel`. The function `d_bessel` defined in `examples/bessel/bessel.R` computes the target density.

7.1 Constant Majorizer with Numerical Optimization

The weight function may be coded as a lambda in C++ as follows.

```
const vws::dfdb& w = [&](double x, bool log = true) {
    double out = -std::lgamma(x + nu + 1);
    return log ? out : std::exp(out);
};
```

The base distribution's density, CDF, and quantile function may be supplied as a `UnivariateHelper` object using Poisson functions in the R API.

```
fntl::density df = [&](double x, bool log = false) {
    return R::dpois(x, mean, log);
};
fntl::cdf pf = [&](double q, bool lower = true, bool log = false) {
    return R::ppois(q, mean, lower, log);
};
fntl::quantile qf = [&](double p, bool lower = true, bool log = false) {
    return R::qpois(p, mean, lower, log);
};

vws::UnivariateHelper helper(df, pf, qf);
```

The following R snippet builds the C++ code and invokes the sampling function.

```
> Rcpp::sourceCpp("examples/bessel/bessel-v1.cpp")
> out1 = r_bessel_v1(n = 1000, lambda, nu, N = 50, tol = 0.10)
> head(out1$draws)
```

```
[1] 3 4 4 3 5 4
```

Figure 7.1 plots the bound for the rejection probability during the refinement process, which is captured in the variable `out$lbdd`. Figure 7.2 plots the empirical distribution of the draws and compares them to the target probability mass function (pmf).

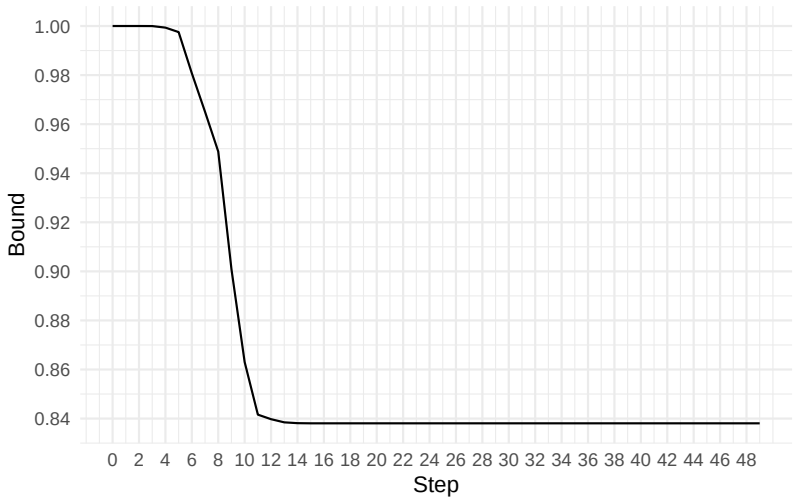


Figure 7.1: Refinement for lognormal-normal example with constant majorizer.

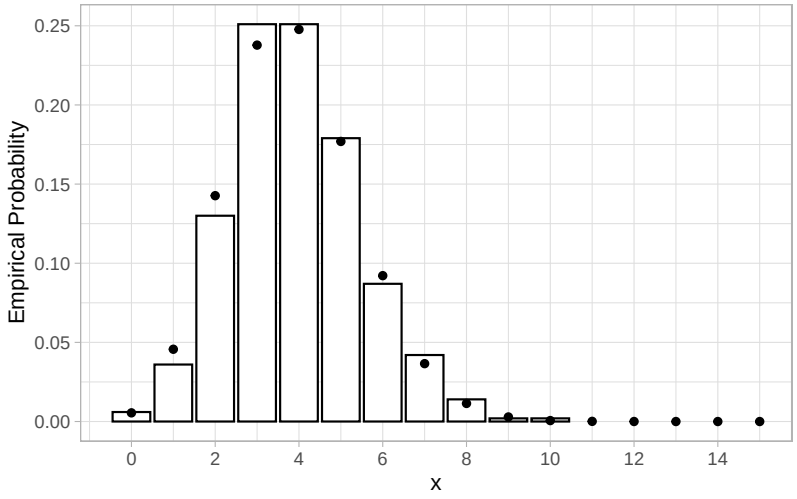


Figure 7.2: Empirical distribution of draws (bars) versus target (points) for Bessel example with constant majorizer.

7.2 Constant Majorizer with Custom Optimization

We can make several improvements to the default numerical majorizer by coding it explicitly. Note that $\log w(x) = -\log \Gamma(x + \nu + 1)$ is a decreasing function in x so that numerical optimization is not necessary. Therefore,

$$\log w(\alpha_j) \leq \log w(x) \leq \log w(\alpha_{j-1}), \quad x \in \mathscr{D}_j.$$

If the endpoints are not integers, we can improve the bounds by restricting consideration to integers, using $\lfloor \alpha_{j-1} \rfloor + 1$ and $\lfloor \alpha_j \rfloor$ to obtain the maximizer and minimizer, respectively. The following functions put this into practice.

```
const vws::optimizer& maxopt = [](const vws::dfdb& w, double lo,
    double hi, bool log)
{
    if (lo < 0 && hi < 0) { Rcpp::stop("Did not code this case"); }
    double x = (lo < 0) ? 0 : std::floor(lo) + 1;
    double out = w(x, true);
    return log ? out : std::exp(out);
};

const vws::optimizer& minopt = [](const vws::dfdb& w, double lo,
    double hi, bool log)
{
    if (lo < 0 && hi < 0) { Rcpp::stop("Did not code this case"); }
    double x = std::isinf(hi) ? hi : std::floor(hi);
    double out = w(x, true);
    return log ? out : std::exp(out);
};
```

We can construct the proposal using a single `IntConstRegion` that represents the support; the `maxopt` and `minopt` arguments are specified here. Also note that we have specified a lower endpoint α_0 below zero to ensure that the support $(\alpha_0, \alpha_N]$ contains zero.

```
vws::IntConstRegion supp(-0.1, R_PosInf, w, helper, maxopt, minopt);
```

The following R snippet builds the C++ code and invokes the sampling function.

```
> Rcpp::sourceCpp("examples/bessel/bessel-v2.cpp")
> out2 = r_bessel_v2(n = 1000, lambda, nu, N = 50, tol = 0.10)
> head(out2$draws)
```

```
[1] 5 4 3 5 1 5
```

7.3 Linear Majorizer

Our decomposition of f into the given w and g lends itself well to a linear majorizer because the function $\log w(x) = -\log \Gamma(x + \nu + 1)$ is log-concave. In fact, a theorem attributed to Bohr and Mollerup (1922) states that the function $q(x) = \Gamma(x)$ is uniquely characterized as the only function on $(0, \infty)$ which is log-convex with $q(1) = 1$ and $q(x + 1) = xq(x)$. The following remark is also useful to formulate the linear majorizer.

Remark 7.1. By rearranging equation 6.5.13 of Abramowitz and Stegun (1972), the cdf of $T \sim \text{Poisson}(\lambda)$ can be written as

$$F(x \mid \lambda) = e^{-\lambda} \sum_{k=0}^{n} \frac{\lambda^k}{k!} = \frac{\Gamma(x + 1, \lambda)}{\Gamma(x + 1)}, \quad x \in \{1, 2, \ldots\},$$

where $\Gamma(c,z) = \int_z^\infty t^{c-1}e^{-t}dt$ and $\Gamma(c) = \Gamma(c,0)$. Therefore, for $a, b \in \mathbb{R}$ with $0 \le a \le b$ and $T \sim \text{Poisson}(\lambda)$,

$$\begin{aligned}
\mathrm{P}(a < T \le b) &= \mathrm{P}(\lfloor a \rfloor + 1 \le X \le \lfloor b \rfloor) \\
&= \sum_{x=\lfloor a \rfloor + 1}^{\lfloor b \rfloor} \frac{e^{-\lambda}\lambda^x}{x!} \\
&= \sum_{x=0}^{\lfloor b \rfloor} \frac{e^{-\lambda}\lambda^x}{x!} - \sum_{x=0}^{\lfloor a \rfloor} \frac{e^{-\lambda}\lambda^x}{x!} \\
&= \frac{\Gamma(\lfloor b \rfloor + 1, \lambda)}{\Gamma(\lfloor b \rfloor + 1)} - \frac{\Gamma(\lfloor a \rfloor + 1, \lambda)}{\Gamma(\lfloor a \rfloor + 1)}.
\end{aligned}$$

To verify the first equality, if a is an integer then

$$\begin{aligned}
\mathrm{P}(a < T \le b) &= \mathrm{P}(T \in \{a+1, \ldots, \lfloor b \rfloor\}) \\
&= \mathrm{P}(T \in \{\lfloor a \rfloor + 1, \ldots, \lfloor b \rfloor\}) = \mathrm{P}(\lfloor a \rfloor + 1 < T \le \lfloor b \rfloor);
\end{aligned}$$

otherwise, if a is not an integer then

$$\begin{aligned}
\mathrm{P}(a < T \le b) &= \mathrm{P}(T \in \{\lceil a \rceil, \ldots, \lfloor b \rfloor\}) \\
&= \mathrm{P}(T \in \{\lfloor a \rfloor + 1, \ldots, \lfloor b \rfloor\}) = \mathrm{P}(\lfloor a \rfloor + 1 < T \le \lfloor b \rfloor).
\end{aligned}$$

■

We have the expression

$$\begin{aligned}
\overline{\xi}_j &= \int_{\alpha_{j-1}}^{\alpha_j} \overline{w}_j(x) g(x) d\nu(x) \\
&= \exp\{\overline{\beta}_{0j}\} \sum_{k \in (\alpha_{j-1}, \alpha_j]} \exp\{\overline{\beta}_{1j} x\} \frac{(\lambda^2/4)^x e^{-\lambda^2/4}}{x!} \\
&= \exp\left(\overline{\beta}_{0j} - \frac{\lambda^2}{4} + \frac{\lambda^2}{4} e^{\overline{\beta}_{1j}}\right) \mathrm{P}(\alpha_{j-1} < \overline{T} \le \alpha_j),
\end{aligned}$$

where $\overline{T} \sim \text{Poisson}(e^{\overline{\beta}_{1j}}\lambda^2/4)$, and similarly,

$$\underline{\xi}_j = \exp\left(\overline{\beta}_{0j} - \frac{\lambda^2}{4} + \frac{\lambda^2}{4} e^{\overline{\beta}_{1j}}\right) \mathrm{P}(\alpha_{j-1} < \underline{T} \le \alpha_j).$$

where $\underline{T} \sim \text{Poisson}(e^{\underline{\beta}_{1j}}\lambda^2/4)$. The proposal h is a finite mixture $h(x) = \sum_{j=1}^N \pi_j g_j(x)$ with

$$\begin{aligned}
g_j(x) &= \overline{w}_j(x) g(x) \, \mathrm{I}\{x \in \mathscr{D}_j\} / \overline{\xi}_j \\
&= \frac{\exp\{\overline{\beta}_{0j} + \overline{\beta}_{1j} x\} (\lambda^2/4)^x e^{-\lambda^2/4}/x!}{\exp\left(\overline{\beta}_{0j} - \frac{\lambda^2}{4} + \frac{\lambda^2}{4} e^{\overline{\beta}_{1j}}\right) \mathrm{P}(\alpha_{j-1} < \overline{T} \le \alpha_j)} \, \mathrm{I}\{x \in \mathscr{D}_j\} \\
&= \frac{1}{x!} \left(\frac{\lambda^2}{4} e^{\overline{\beta}_{1j}}\right)^x \exp\left(-\frac{\lambda^2}{4} e^{\overline{\beta}_{1j}}\right) \frac{1}{\mathrm{P}(\alpha_{j-1} < \overline{T} \le \alpha_j)} \, \mathrm{I}(\alpha_{j-1} < x \le \alpha_j),
\end{aligned}$$

which is the density of Poisson($\frac{\lambda^2}{4}e^{\overline{\beta}_{1j}}$) truncated to the interval $(\alpha_{j-1}, \alpha_j]$. Remark 2.3 is used to select the expansion point c to determine coefficients for the majorizer in the log-concave case, with

$$\begin{aligned} M_j(s) &= \int_{\alpha_{j-1}}^{\alpha_j} e^{sx} g(x) d\nu(x) \\ &= \exp(-\lambda^2/4) \exp(e^s \lambda^2/4) \sum_{x \in (\alpha_{j-1}, \alpha_j]} \frac{1}{x!} \exp\{-e^s \lambda^2/4\} (e^s \lambda^2/4)^x \\ &= \exp\left\{-\frac{\lambda^2}{4} + e^s \frac{\lambda^2}{4}\right\} \mathrm{P}(\alpha_{j-1} < T \leq \alpha_j), \end{aligned}$$

where $T \sim \text{Poisson}(e^s \lambda^2/4)$.

The following R snippet builds the C++ code and invokes the sampling function.

```
> Rcpp::sourceCpp("examples/bessel/bessel-v3.cpp")
> out3 = r_bessel_v3(n = 1000, lambda, nu, lo = -0.1, hi = 1e5, N = 50,
+   tol = 0.10)
> head(out3$draws)
```

```
[1] 5 7 3 2 3 3
```

Figure 7.3 plots the bound for the rejection probability for this sampler after each step of refining, along with that of the previous two versions. A substantial improvement in efficiency is seen here; fewer regions are needed to achieve a small rejection probability. Although versions 1 and 2 of the sampler are based on the same proposal, the changes to their bounds are seen to differ due to randomness in knot selection. Figure 7.4 and Figure 7.5 display the three versions of majorized weight functions and proposal distributions, respectively, after refinement. It is seen that the majorizer from Section 7.1 does not provide a tight bound, even with many knots, because our numerical optimization is not specific to the integers in the support. This is rectified by the manual optimization from Section 7.2. The linear majorizer provides a tight bound with fewer knots.

Because the mass of our selected Bessel(λ, ν) distribution is focused on a relatively small set of integers, the second and third versions of the proposal can be made practically equivalent to the target with a sufficient number of regions. After being refined to this point, candidates are rarely discarded in rejection sampling.

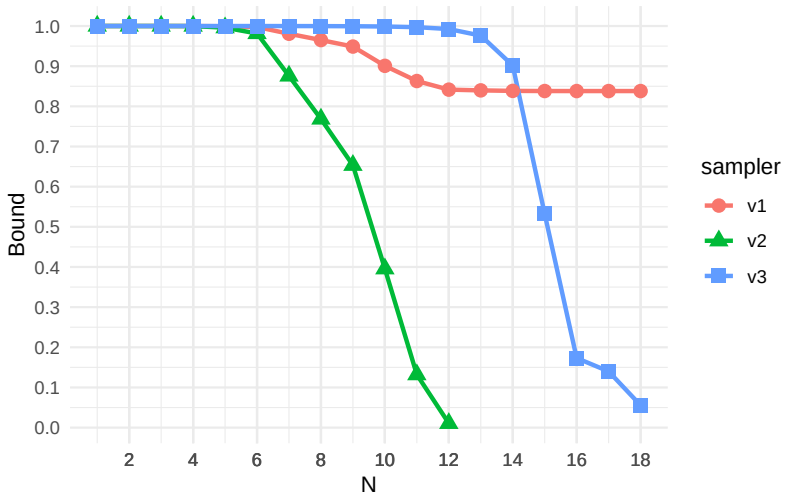
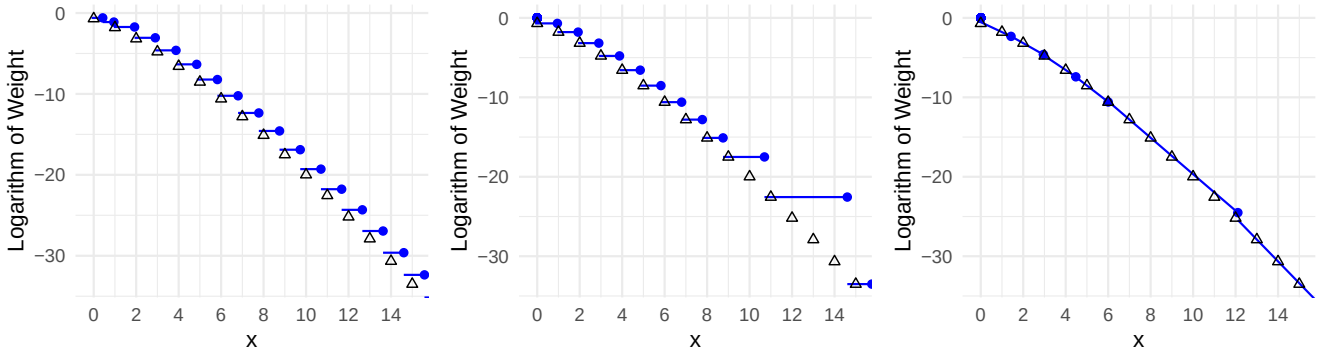
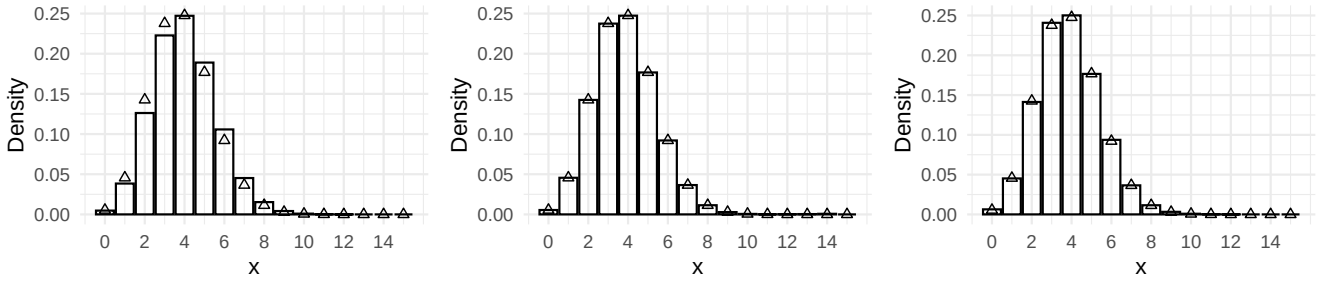


Figure 7.3: Refinement for Bessel example with three majorizers.



(a) v1. (b) v2. (c) v3.

Figure 7.4: Majorizers (on the log-scale) for Bessel example for three sampler versions. The blue curve is the majorizer with a blue dot marking the right endpoint of each region. The black triangle displays the value of $\log w(x)$ at integers $x \in \mathbb{N}$.



(a) v1. (b) v2. (c) v3.

Figure 7.5: Proposal density (bars) for Bessel example for three sampler versions. Triangles display the values of $f(x)$, $x \in \mathbb{N}$.

8 Persistent Proposals in R

The samplers developed in previous sections each expose a function in R which can be used to generate samples. A proposal is constructed each time the function is called which exists only for the duration of the call. For purposes of efficiency and otherwise, it may be desirable to have a persistent proposal that can be gradually refined and reused. This section demonstrates several Rcpp mechanisms to enable this workflow: first using external pointers then using using modules. The latter is based on object-oriented programming but recommended over the former as it is less susceptible to crashing R. We demonstrate the two approaches by revisiting the example from Section 5.1. For more information on external pointers and modules, refer to the vignette "Exposing C++ functions and classes with Rcpp modules" in the `Rcpp` package (Eddelbuettel et al. 2026) and section "External pointers and weak references" of the guide "Writing R Extensions" (R Core Team 2026).

8.1 External Pointers

An Rcpp external pointer (`XPtr`) can be used to construct objects in C++ and return them to R so that they persist and can be accessed in subsequent calls to C++. This is especially useful for use with complicated data structures that are not easy to directly represent as R objects. A VWS proposal is an example of such as structure. The full C++ code is given in `examples/vmf/vmf-v1-xptr.cpp`. Here we describe the components of that file.

The following typedefs provide more readable names for the proposal and its `XPtr`.

```
typedef vws::FMMProposal<double, vws::RealConstRegion> t_proposal;
typedef Rcpp::XPtr<t_proposal> t_proposal_xptr;
```

The following function creates an `XPtr` to a proposal. There are several notable changes from the implementation in Section 5.1.

```
// [[Rcpp::export]]
t_proposal_xptr vmf_pre_v1_xptr(double kappa, double d)                    ①
{
    vws::dfdb w =
    [=](double x, bool log = true) {                                       ②
        double out = R_NegInf;
        if (std::fabs(x) < 1){
            out = 0.5 * (d - 3) * std::log1p(-std::pow(x, 2));
        }
        return log ? out : std::exp(out);
    };

    fntl::density df = [=](double x, bool log = false) {
        return d_texp(x, kappa, -1, 1, log);
    };
```

```
    fntl::cdf pf = [=](double q, bool lower = true, bool log = false) {
        return p_texp(q, kappa, -1, 1, lower, log);
    };

    fntl::quantile qf = [=](double p, bool lower = true, bool log = false) {
        return q_texp(p, kappa, -1, 1, lower, log);
    };

    vws::UnivariateHelper helper(df, pf, qf);
    vws::RealConstRegion supp(-1, 1, w, helper);

    auto p = new t_proposal(supp);                                          ③
    auto out = t_proposal_xptr(p, true);
    set_tag(out, "t_proposal");                                             ④
    return out;
}
```

① The function returns an `XPtr` to a proposal rather than the result of sample generation.
② Capture by value with the lambdas here using `[=]`. Variables `kappa` and `d` may not be available in the environment when evaluating the lambdas later; this ensure that they can still be accessed by the lambdas.
③ Create a proposal object on the heap using `new` and wrap it in an `XPtr` to be returned to R.
④ The function `set_tag` a utility we will define; more explanation is provided below.

The following functions make use of the proposal `XPtr`. The `refine` function modifies the proposal by applying up to `N` refinement steps of Algorithm 1 until toleration `tol` is achieved. The `draw` function draws a sample of size `n`.

```
// [[Rcpp::export]]
Rcpp::NumericVector refine(
    t_proposal_xptr h,                                                      ①
    unsigned int N, double tol = 0)
{
    assert_tag(h, "t_proposal");                                            ②
    return h->refine(N - 1, tol);                                           ③
}

// [[Rcpp::export]]
Rcpp::List draw(t_proposal_xptr h, unsigned int n,
    unsigned int max_rejects = 10000, unsigned int report = 10000)
{
    assert_tag(h, "t_proposal");

    vws::rejection_args args;
    args.max_rejects = max_rejects;
```

```
    args.report = report;

    auto out = vws::rejection(*h, n, args);

    return Rcpp::List::create(
        Rcpp::Named("draws") = out.draws,
        Rcpp::Named("rejects") = out.rejects
    );
}
```

① The first argument is expected to be the `XPtr` obtained from `vmf_pre_v1_xptr`.
② The function `assert_tag` is a utility we will define; more explanation is provided below.
③ The `refine` member function is called on the object pointed to by `h`; this modifies it in place.

Something to be aware of with `XPtr` arguments is that type safety is not as well-established as with other Rcpp argument types. For example, consider attempting the operation `h->refine(N - 1, tol)` with `h` as an `XPtr` to something other than a `t_proposal`. At the time of this writing, this crashes R rather than the operation being prevented with an error message. To help avoid this issue, we use the tagging mechanism for `XPtr` which is in the Rcpp API. Define the following functions for convenience: `set_tag` applies a string tag to an `XPtr` and `assert_tag` throws an exception if an `XPtr` has no tag or has a tag different from the expected one.

```
template <typename T>
void assert_tag(Rcpp::XPtr<T> x, const char* tag)
{
    if (Rf_isNull(R_ExternalPtrTag(x))) {
        Rcpp::stop("XPtr does not have a tag");
    }

    if (Rcpp::as<Rcpp::String>(R_ExternalPtrTag(x)) != Rcpp::String(tag)) {
        Rcpp::stop("XPtr is not tagged as %s", tag);
    }
}

template <typename T>
void set_tag(Rcpp::XPtr<T> x, const char* tag)
{
    R_SetExternalPtrTag(x, Rcpp::wrap(Rcpp::String(tag)));
}
```

Now let us demonstrate use of these functions from R.

```
> Rcpp::sourceCpp("examples/vmf/vmf-v1-xptr.cpp")
> h = vmf_pre_v1_xptr(kappa = 5, d = 4)
> lbdd = refine(h, N = 50, tol = 0.10)
> out = draw(h, n = 1000, max_rejects = 10000)
```

```
> rm(h)
> print(lbdd)
```

```
 [1]  0.00000000  0.00000000 -0.07731969 -0.32052199 -0.68269398 -1.01037113
 [7] -1.21286339 -1.32381733 -1.43456513 -1.47975787 -1.62326086 -1.73784037
[13] -1.80570047 -1.83395461 -1.86544879 -1.93059057 -1.97740810 -2.07778562
[19] -2.16647471 -2.19334229 -2.21733803 -2.22633620 -2.23585806 -2.24137567
[25] -2.28215888 -2.32562545
```

```
> head(out$draws)
```

```
[1] 0.8999098 0.8701695 0.5357860 0.8131539 0.3903483 0.8255003
```

For demonstration purposes, let us define a C++ function that returns an `XPtr` to a vector. When an `XPtr` to a proposal is expected but an `XPtr` to a vector is encountered, ensure that an appropriate exception is thrown.

```
// [[Rcpp::export]]
Rcpp::XPtr<std::vector<double>> test_vector()
{
    std::vector<double>* p = new std::vector<double>();
    p->push_back(10);
    p->push_back(10);
    p->push_back(12);
    return Rcpp::XPtr<std::vector<double>>(p);
}
```

```
tryCatch({
    h = test_vector()
    refine(h, N = 50, tol = 0.10)
}, error = function(e) {
    print(e)
})
```

```
<Rcpp::exception in eval(expr, envir): XPtr does not have a tag>
```

Remark 8.1. In the current implementation of `vws`, lambdas `w`, `df`, `pf`, and `qf` are copied to all regions used in the sampler. The overhead created by copying will likely not be significant in most instances, but could be an issue when very large objects are captured by value. In such cases, users may consider using pointers rather than values to capture these objects. ■

8.2 Modules

The external pointer approach in Section 8.1 lets us operate on a proposal object from R through C++ functions. We can take this one step further with object-oriented programming so that the proposal appears as a class in R that includes its methods. The complete C++ code for this example is in `examples/vmf/vmf-v1-module.cpp`.

The following typedef provides a more readable name for the proposal.

```
typedef vws::FMMProposal<double, vws::RealConstRegion> t_proposal;
```

We now define a class named `RcppProposal` which will be exposed as an Rcpp module. This class inherits from ("is a") `t_proposal`. Its constructor, `refine` operation, and `draw` operation are analogous to those in Section 8.1.

```
class RcppProposal : t_proposal
{
public:
    RcppProposal(double kappa, double d)
    : t_proposal(supp(kappa, d))
    {
    }

    Rcpp::List draw(unsigned int n, unsigned int max_rejects = 10000,
        unsigned int report = 10000);

    Rcpp::NumericVector refine(unsigned int N, double tol = 0) {          ①
        return t_proposal::refine(N, tol);
    }

private:
    vws::RealConstRegion supp(double kappa, double d);                    ②
};
```

① `t_proposal::refine` is the method in the base class; it is possible to expose this directly from the module, but we opt to call it from `RcppProposal::refine` for simplicity.

② The `supp` function handles most of the constructor work; it returns a single region `supp` with all of the necessary operations. The `RcppProposal` constructor creates an instance of itself from region.

The following code exposes `RcppProposal` as an Rcpp module.

```
RCPP_MODULE(vws_module) {
    Rcpp::class_<RcppProposal>("RcppProposal")
    .constructor<double,double>()
    .method("draw", &RcppProposal::draw)
```

```
    .method("refine", &RcppProposal::refine)
    ;
}
```

See `examples/vmf/vmf-v1-module.cpp` for the implementations for the `draw` and `refine` methods.

Now let us demonstrate use of these functions from R.

```
> Rcpp::sourceCpp("examples/vmf/vmf-v1-module.cpp")
> h = new(RcppProposal, kappa = 5, d = 4)
> lbdd = h$refine(N = 50 - 1, tol = 0.10)
> out = h$draw(n = 1000, max_rejects = 1000, max_rejects = 10000)
> print(lbdd)
```

```
 [1]  0.00000000  0.00000000 -0.07731969 -0.32052199 -0.68269398 -1.01037113
 [7] -1.21286339 -1.30836222 -1.32780700 -1.45753144 -1.50403430 -1.60187880
[13] -1.65182032 -1.83314614 -1.91770903 -1.93799484 -1.97365142 -2.00433532
[19] -2.09825943 -2.13419055 -2.22214608 -2.26208634 -2.29684893 -2.34653030
```

```
> head(out$draws)
```

```
[1] 0.9549472 0.7856618 0.8659067 0.8060806 0.8843272 0.8934049
```

Remark 8.2. The lambdas `w`, `df`, `pf`, and `qf` in this code capture `kappa` and `d` by value as in Section 8.1, which can lead to unwanted overhead from copying mentioned in Remark 8.1. In the present case, we can consider storing a single copy of any large objects as member variables of the `RcppProposal` class and capturing them by reference (rather than by value) in the lambdas. ■

References

Abowd, John, Robert Ashmead, Ryan Cumings-Menon, et al. 2022. "The 2020 Census Disclosure Avoidance System TopDown Algorithm." *Harvard Data Science Review*, nos. Special Issue 2.

Abramowitz, Milton, and Irene A. Stegun. 1972. *Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables*. 10th Printing. Applied Mathematics Series 55. National Bureau of Standards.

Bohr, H., and J. Mollerup. 1922. *Lærebog i Kompleks Analyse*. III. Copenhagen.

Devroye, Luc. 1986. *Non-Uniform Random Variate Generation*. Springer.

Devroye, Luc. 2002. "Simulating Bessel Random Variables." *Statistics & Probability Letters* 57 (3): 249–57. https://doi.org/10.1016/S0167-7152(02)00055-X.

Dwork, Cynthia, and Aaron Roth. 2014. "The Algorithmic Foundations of Differential Privacy." *Foundations and Trends in Theoretical Computer Science* 9 (3–4): 211–407. https://doi.org/10.1561/0400000042.

Eddelbuettel, Dirk. 2013. *Seamless R and C++ Integration with Rcpp*. Springer. https://doi.org/10.1007/978-1-4614-6868-4.

Eddelbuettel, Dirk, Romain Francois, JJ Allaire, et al. 2026. *Rcpp: Seamless r and c++ Integration*. https://doi.org/10.32614/CRAN.package.Rcpp.

Huijben, Iris A. M., Wouter Kool, Max B. Paulus, and Ruud J. G. van Sloun. 2023. "A Review of the Gumbel-Max Trick and Its Extensions for Discrete Stochasticity in Machine Learning." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45 (2): 1353–71. https://doi.org/10.1109/tpami.2022.3157042.

Irimata, Kyle M., Andrew M. Raim, Ryan Janicki, James A. Livsey, and Scott H. Holan. 2022. *Evaluation of Bayesian Hierarchical Models of Differentially Private Data Based on an Approximate Data Model*. Research Report Series: Statistics #2022-05. Center for Statistical Research; Methodology, U.S. Census Bureau. https://www.census.gov/library/working-papers/2022/adrm/RRS2022-05.html.

Janicki, Ryan, Mikaela Meyer, Adam Hall, et al. 2025+. *Statistical Post-Processing to Improve the Accuracy and Utility of Differentially Private Counts in the Decennial Census. (Submitted)*.

Mardia, Kanti V., and Peter E. Jupp. 1999. *Directional Statistics*. Wiley. https://doi.org/10.1002/9780470316979.

Martino, Luca, David Luengo, and Joaquín Míguez. 2018. *Independent Random Sampling Methods*. Springer. https://doi.org/10.1007/978-3-319-72634-2.

R Core Team. 2025. *Writing R Extensions*. R Foundation for Statistical Computing. https://cran.r-project.org/doc/manuals/r-devel/R-exts.html.

R Core Team. 2026. *Writing R Extensions*. https://cran.r-project.org/doc/manuals/r-release/R-exts.html.

Raim, Andrew M. 2021. *Direct Sampling in Bayesian Regression Models with Additive Disclosure Avoidance Noise*. Research Report Series: Statistics #2021-01. Center for Statistical Research; Methodology, U.S. Census Bureau. https://www.census.gov/library/working-papers/2021/adrm/RRS2021-01.html.

Raim, Andrew M. 2024. *fntl: Numerical Tools for Rcpp and Lambda Functions*. https://doi.org/10.32614/CRAN.package.fntl.

Raim, Andrew M., James A. Livsey, and Kyle M. Irimata. 2025. *Rejection Sampling with Vertical Weighted Strips*. https://arxiv.org/abs/2401.09696.

Smyth, Gordon K. 2005. "Numerical Integration." *Encyclopedia of Biostatistics*, 3088–95.

Wickham, Hadley, Mara Averick, Jennifer Bryan, et al. 2019. "Welcome to the tidyverse." *Journal of Open Source Software* 4 (43): 1686. https://doi.org/10.21105/joss.01686.

Yuan, Lin, and John D. Kalbfleisch. 2000. "On the Bessel Distribution and Related Problems." *Annals of the Institute of Statistical Mathematics* 52 (3): 438–47. https://doi.org/10.1023/A:1004152916478.