walkr

by Andy Yao, David Kane

Abstract The `walkr` package uses random walks to sample points from the intersection of the N simplex with M hyperplanes. Mathematically, the sampling space is all vectors x that satisfy $Ax = b$, $\sum x = 1$, and $x_i \geq 0$. The sampling algorithms implemented are hit-and-run and Dikin walk, both of which are MCMC (Monte-Carlo Markov Chain) random walks. `walkr` also provide tools to examine the convergence properties of the random walks.

Introduction

Consider all possible vectors x that satisfy the underdetermined matrix equation $Ax = b$, such that every component of x is ≥ 0 and sum to 1. How do we generate a diverse sample of such x's? The `walkr` package uses MCMC (Monte-Carlo Markov Chain) random walks to generate such a sample.

`walkr` contains two MCMC random walks. Our first random walk is hit-and-run. Hit-and-run is a widely used MCMC sampling method that guarantees uniform sampling asympotically, but mixes slower and slower as the dimensions of A increase. Our second random walk is the Dikin Walk, which generates a nearly uniform sample and exhibits much stronger mixing. `walkr` uses `Rcpp`(Eddelbuettel and François (2011)) and `RcppEigen`(Bates and Eddelbuettel (2013)) for its core matrix calculations which take up most of the run-time.

`walkr` also provides statistical diagnostics of the mixing and convergence properties of a MCMC random walk.

Mathematical Background of Sampling Space

In this section, we go through the mathematical background of the space from which we are sampling – the intersection of the N-simplex and hyperplanes. The reader does not need to read this section in order to use our package or understand the sampling algorithms. However, this section should provide the reader a better sense of what the sample space is geometrically and mathematically.

1) $\mathbf{Ax} = \mathbf{b}$ represent a system of M linear equations with N variables ($M \ll N$). Hence, A is a $M \times N$ matrix (M variables and N constraints), x is a $N \times 1$ vector, and b is a $M \times 1$ vector. Every row in A represents a hyperplane in $\mathbb{R}^N$.

2) The N-dimensional unit simplex (N-Simplex) is described by:

$$x_1 + x_2 + x_3 + ... + x_N = 1$$
$$x_i \geq 0$$

Sampling space: simple 3D case

Let's begin with the simplest case – one linear constraint in 3 dimensional space.

$$x_1 + x_3 = 0.5$$

We can express this in terms of matrix equation $Ax = b$, where:

$$A = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}, \quad b = 0.5, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

In addition, we require the solution space to be intersected with the 3D simplex:

$$\sum x_i = 1$$

$$x_i \geq 0$$

In the following graph, we draw the intersection of the two. The orange equilateral triangle represents the 3D simplex, and the blue rectangle represents the plane $w_1 + w_3 = 0.5$. The intersection of the hyperplane (blue) with the simplex (orange) is the red line segment, which is our sampling space.

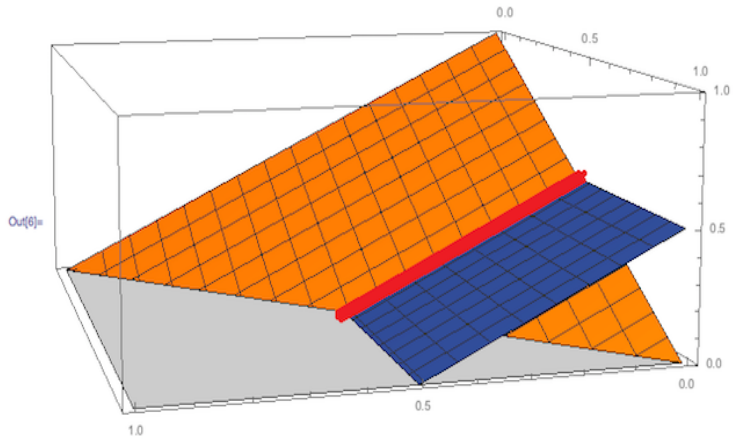


Figure 1: The sampling space is the line segment (red), which is the intersection of the 3D simplex (orange) and a hyperplane (blue)

Matrix Representation of Multiple Hyperplanes

Every hyperplane is described by one linear equation. Thus, a system of linear equations is the intersection of hyperplanes. In general, if we have M linear equations and N variables, then $Ax = b$ would look like:

$$A_{M\times N} = \underbrace{\left[\begin{array}{c} \\ \cdots \\ \\ \end{array}\right]}_{\text{N columns (variables)}} \Big\} \text{M rows (constraints)}$$

Again, b is a $M \times 1$ vector, and x is a $N \times 1$ vector.

4D space

Just like how the 3D simplex is a 2D surface living in 3D space, the 4D simplex (i.e. $x_1 + x_2 + x_3 + x_4 = 1$, $x_i \geq 0$) could be viewed as a 3D object. Specifically, the 4D simplex is the following tetradhedron when viewed from 3D space, with verticies $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, $(0, 0, 1, 0)$, and $(0, 0, 0, 1)$.

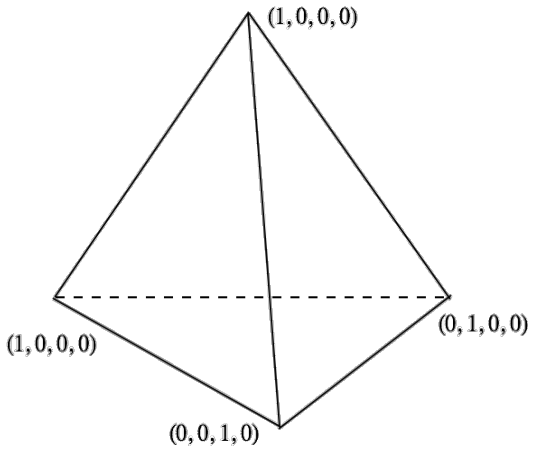


Figure 2: 4D Simplex Can Live in 3D Space

Now imagine the intersection of the 4D simplex with one hyperplane in 4D (1 equation, or 1 row in $Ax = b$). For a specific A and b, we demonstrate the intersection in the figure below. The resulting shape is a trapezoid in 4D space.

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 22 & 2 & 2 & 37 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 16 \end{bmatrix}$$

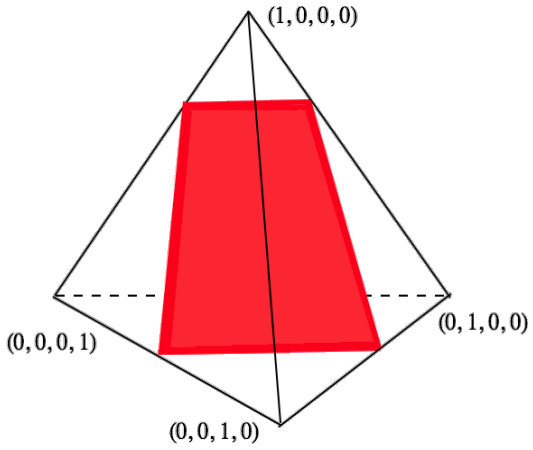


Figure 3: Intersection of a 4D Simplex and a Hyperplane

In higher dimensions, the same logic applies. Each row in $Ax = b$ is a hyperplane living in $\mathbb{R}^N$ (given N variables). Thus, geometrically, our sampling space is: **the intersection of hyperplanes with the N-simplex**.

From $Ax = b$ and the N-simplex to $Ax \leq b$

Our sampling space is bounded (i.e. has finite volume in $\mathbb{R}^N$). More formally, our sampling space is known as a **convex-polytope** in $\mathbb{R}^N$. Convex-polytopes are commonly described in the literature by a generic $Ax \leq b$. Here, we present a simple linear transformation which transforms the intersection of $Ax = b$ and the N-simplex to the form $Ax \leq b$.

First, note that the equality part of the simplex constraint ($\sum x = 1$) could be added as an extra row in $Ax = b$

$$A = \begin{bmatrix} & ..... & \\ & ..... & \\ & ..... & \\ 1 & 1 & ..... & 1 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} ... \\ ... \\ ... \\ 1 \end{bmatrix}$$

Second, to find the complete solution to the new $Ax = b$ (i.e. the set of all possible x's that satisfy $Ax = b$), we must find the Null Space of A(all x's that satisfy $Ax = 0$), then add on any particular solution to $Ax = b$ (This procedure can be found in any Linear Algebra textbook).

Mathematically, if the original A was $M \times N$, then after adding on the extra row from the simplex, the basis vectors, each with N components, which span the Null Space of our new A will be:

$$\left\{ v_1, \quad v_2, \quad v_3, \quad ..... \quad , \quad v_{N-(M+1)} \right\}$$

Using any particular solution, $v_{particular}$, the complete solution to the new $Ax = b$ will be

$$\left\{ v_{particular} + \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + ... + \alpha_{N-(M+1)} v_{N-(M+1)} \quad | \quad \alpha_i \in \mathbb{R} \right\}$$

Lastly, we tag on the $x_i \geq 0$ constraints, and with some algebraic manipulations:

$$v_{particular} + \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + ... + \alpha_{N-(M+1)} v_{N-(M+1)} \quad \geq \begin{bmatrix} 0 \\ 0 \\ ... \\ ... \\ ... \\ 0 \end{bmatrix}$$

$$\alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + ... + \alpha_{N-(M+1)} v_{N-(M+1)} \quad \geq -v_{particular}$$

$$V\alpha \geq -v_{particular}, \quad \text{where:} \quad V = \begin{bmatrix} v_1 & v_2 & ... & v_{N-(M+1)} \end{bmatrix}, \quad \alpha = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_{N-(M+1)} \end{bmatrix}$$

And finally, we arrive at the form $Ax \leq b$.

$$-V\alpha \leq v_{particular}$$

We have performed a **transformation** from "x-space" (coordinates described by $x_1, x_2, ...$) to "α-space" (coordinates described by $\alpha_1, \alpha_2, ...$). The geometric object described is still the same convex polytope. In fact, `walkr` internally performs this transformation, samples the α's, maps them back to "x-space, and then returns the sampled points.

The user need not be concerned with this transformation affecting the uniformity or mixing properties of our MCMC sampling algorithms. This is because the transformation above is an affine transformation, which preserves uniformity. Simply put, sampling in either space is equivalent.

Having understood that the intersection of $Ax = b$ with the unit-simplex is a convex polytope, we are ready to dive into the core of `walkr` – MCMC random walks.

Random Walk: How to pick starting points?

MCMC random walks need a starting point, x_0, in the interior of the convex polytope. `walkr` generates such starting points using linear programming. Specifically, the `lsei` function of `limSolve` finds x which:

$$\begin{aligned} \text{minimizes} \quad & |Cx - d|^2 \\ \text{subject to} \quad & Ax \leq b \end{aligned}$$

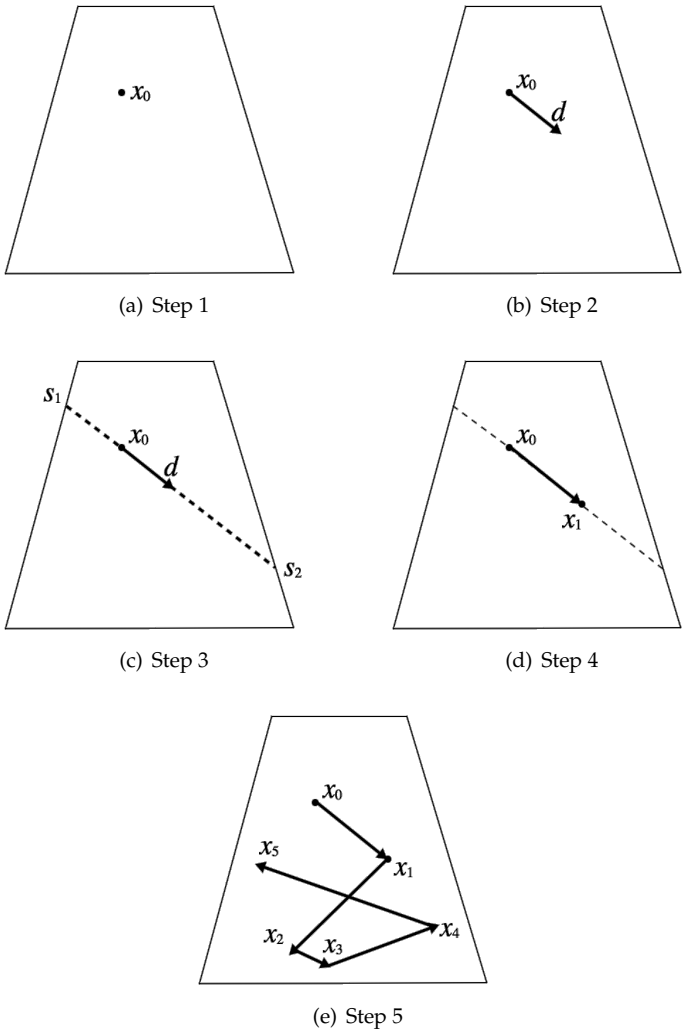
Thus, we randomly generate C and d obtaining x which satisfy $Ax \leq b$. We discovered that the x's generated this way fall randomly on the boundaries of our convex polytope, due to the minimizing property of linear programming. Thus, we repeat this for say, 10 times, and then take an average of the x's generated. This averaged point is x_0, our starting point.

Random Walk: Hit-and-run

The hit-and-run algorithm is as follows:

1. Set starting point x_0 as current point
2. Randomly generate a direction $\vec{d}$. If we are in N dimensions, then d will be a vector of N components. Specifically, d is a uniformly generated unit vector on the N dimensional unit-sphere
3. Find the chord S through x_0 along the directions $\vec{d}$ and $-\vec{d}$. We find end points s_1 and s_2 of the chord by going through the rows of $Ax_0 \leq b$ one by one, setting the inequality to equality (so we hit the surface). Then, parametrize the chord along x_0 by $s_1 + t(s_2 - s_1)$, where $t \in [0, 1]$
4. Pick a random point x_1 along the chord S by generating t from `Uniform[0,1]`
5. Set x_1 as current point
6. Repeat algorithm until number of desired points sampled

Here is a picture of the hit-and-run algorithm:



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

(e) Step 5

Figure 4: The hit-and-run algorithm begins with an interior point x_0 (Step 1). A random direction is selected (Step 2), and the chord along that direction is calculated (Step 3). Then, we pick a random point along that chord and move there as our new point (Step 4). The algorithm is repeated to sample many points (Step 5)

The core implementation of the hit-and-run algorithm calls the `har` function from the `hit-and-run` package on CRAN (van Valkenhoef and Tervonen). `walkr` serves as a convenient wrapper with MCMC diagnostics and multiple chains.

Random Walk: Dikin Walk

Preliminary Definitions

Recall, our sampling space is a convex polytope. We call this convex polytope K, which can be described in the form $Ax \leq b$.

For the definitions below, let a_i represent a row in A, x_i, b_i represent the i^{th} element of x and b. Also recall that A is a $M \times N$ matrix.

Log Barrier Function ϕ:

$$\phi(x) = \sum -\log(b_i - a_i^T x)$$

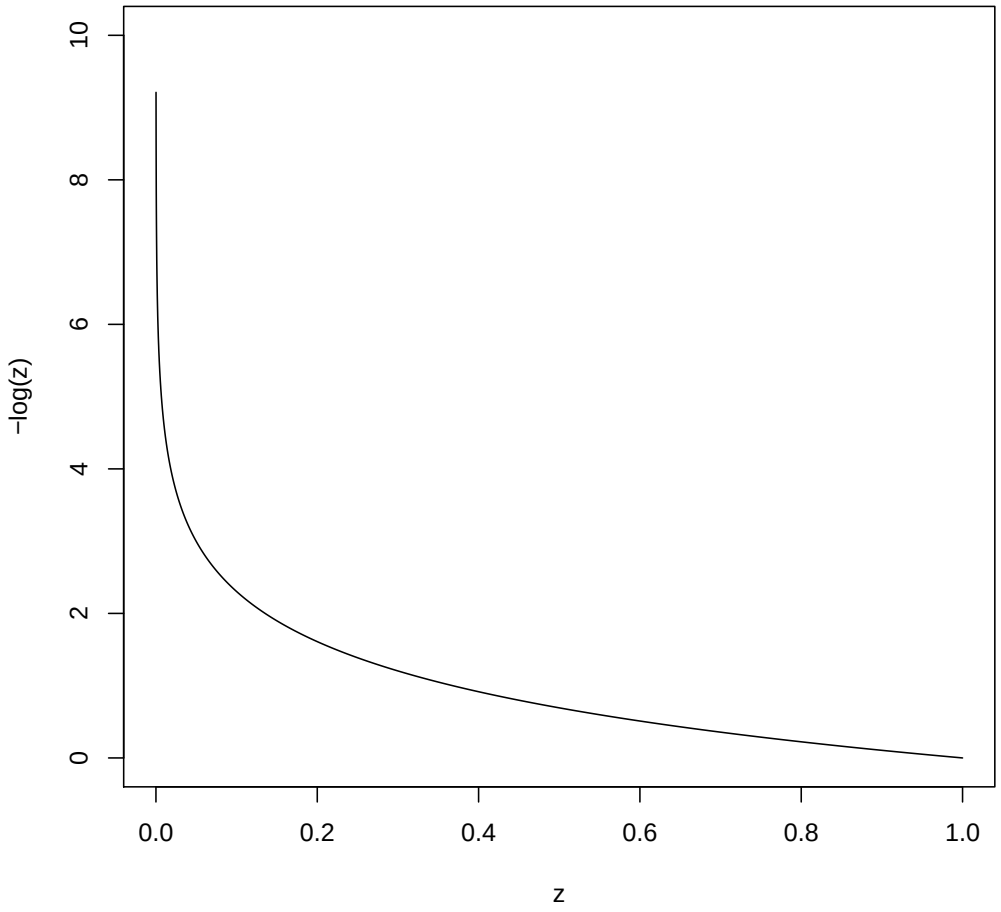
The log-barrier function of $Ax \leq b$ measures how "extreme" or "close-to-the-boundary" a point $x \in K$ is. This is because the value of ϕ tends to ∞ as x tends to the boundary of the polytope K. Mathematically, the polytope could be described by:

$$Ax \leq b$$
$$b - Ax \geq 0$$

Also recall that the negative logarithmic graph looks like this:

```
z <- seq(0, 1, 0.0001)

plot(-log(z)~z, type = 'l', ylim = c(0,10))
```



Thus, the higher the value of the log-barrier function ($\phi(x)$), the closer the point x is to the boundary of the convex polytope K (as $(b_i - a_i x) \to 0$).

Hessian of Log Barrier H_x:

$$H_x = \nabla^2 \phi(x) = ... = A^T D^2 A \quad , \quad \text{where:}$$
$$D = diag(\frac{1}{b_i - a_i^T x})$$

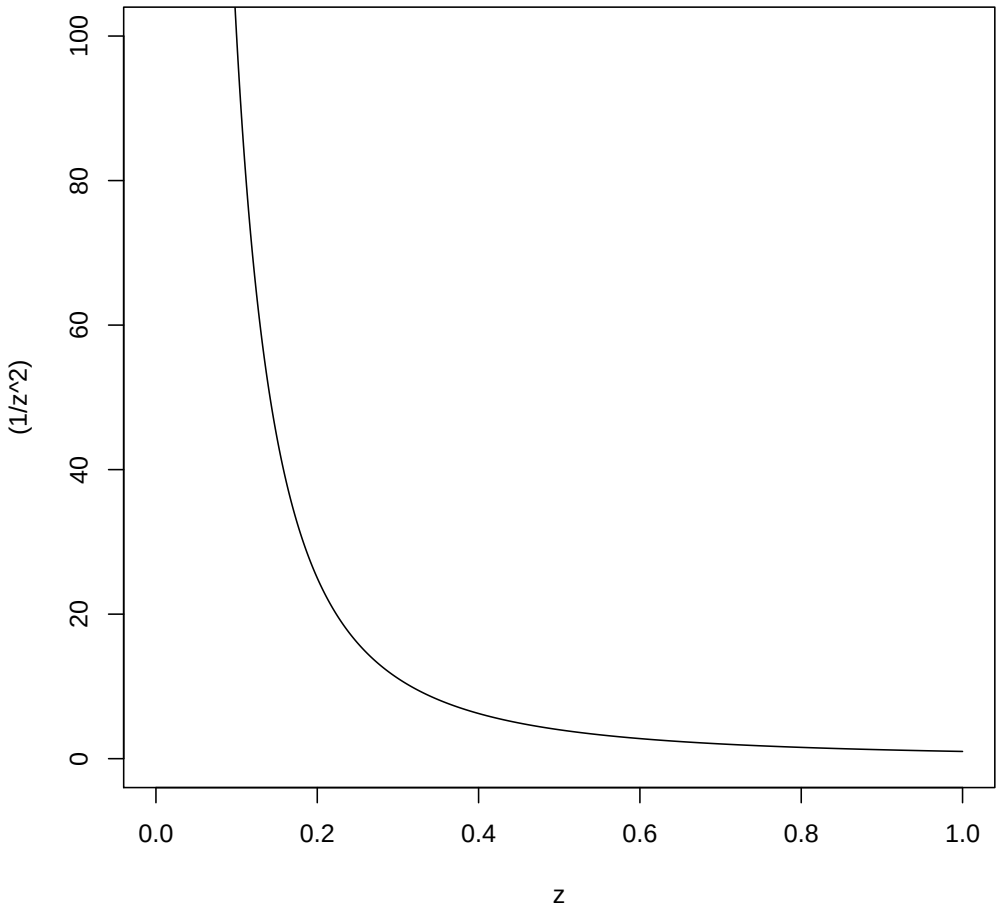
Note: H_x is a $N \times N$ linear operator. D is a $M \times M$ diagonal matrix.

The Hessian matrix($H_x = \nabla^2\phi(x)$) contains all the second derivatives of the function $\phi(x)$ with respect to vector x. In the land of optimization, the Hessian contains information on how the landscape is shaped, and also on extreme values of the landscape. To develop a better understanding of our intuition for the Hessian, we can think of it as a generalized notion of the second derivative.

Let z denote $b_i - a_i x$. Note that the second derivative of $-\log(z)$ is $\frac{1}{z^2}$, which looks like this:

```
z <- seq(0, 1, 0.0001)

plot((1/z^2)~z, type = 'l', ylim = c(0,100))
```



As we could see from the plot, as $z \to 0$ ($x \to$ boundary), the rate at which the Hessian increases becomes faster and faster.

Definition - Dikin Ellipsoid $D^r_{x_0}$

$D^r_{x_0}$, the Dikin Ellipsoid centered at x_0 with radius r is defined as:

$$D^r_{x_0} \quad = \quad \{y \quad | \quad (y - x_0)^T H_{x_0}(y - x_0) \leq r^2\}$$

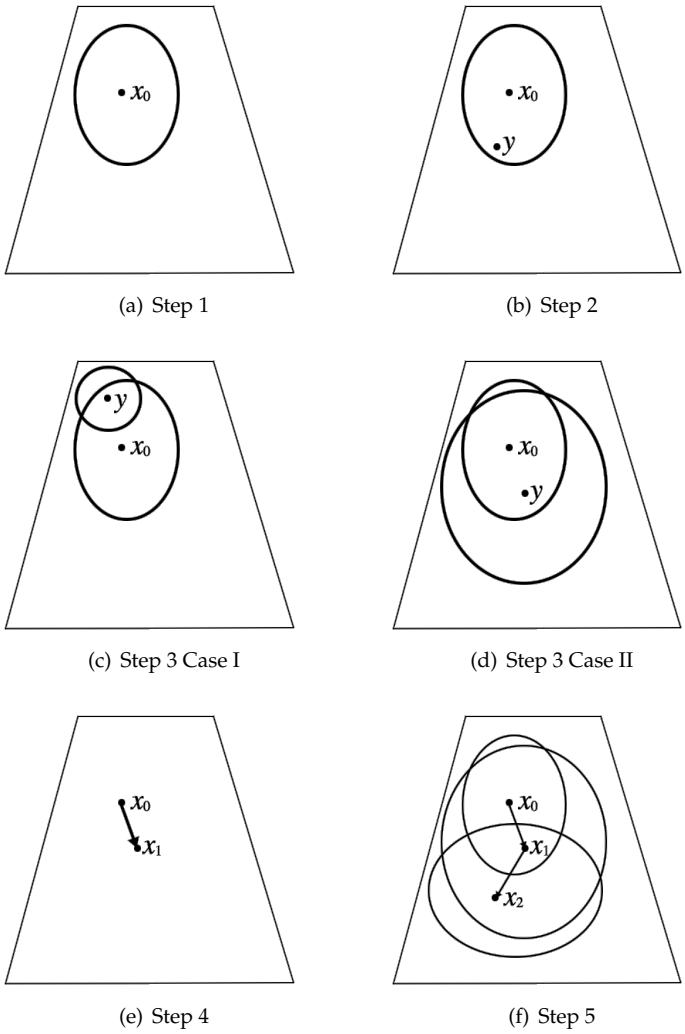
The Dikin Ellipsoid with radius $r = 1$ is the unit ball centered at x_0 with respect to the "Hessian norm" as the notion of length. The "Hessian norm" we could think as a δy on the previous $\frac{1}{z^2}$ graph. For x_0 that are far away from the boundary, given an allowed δy (captured in r), the range for allowed value of z is very large. This corresponds to having a large ellipsoid.

To rephrase, the Dikin Ellipsoid is the collection of all points whose difference with the current

point($y - x_0$) is within the unit threshhold of $r = 1$. When the center x_0 of the Dikin Ellipsoid is far from the boundary of the polytope, the ellipsoid is larger, as a large $\delta(z - x)$ corresponds to a small δy. As the center point x_0 approaches the boundary of the polytope, the ellipsoid becomes smaller and smaller. Therefore, the ellipsoid is able to reshape itself as it surveys through the polytope K, biasing away from the corners of the region(Kannan and Narayanan).

Algorithm `Dikin`

1. Begin with a point $x_0 \in K$. This starting point must be in the polytope.
2. Construct D_{x_0}, the Dikin Ellipsoid centered at x_0
3. Pick a random point y from D_{x_0}
4. If $x_0 \notin D_y$, then reject y (be careful, this condition is counter-intuitive)
5. If $x_0 \in D_y$, then accept y with probability $\min(1, \sqrt{\frac{det(H_y)}{det(H_{x_0})}})$ (the big picture is that the ratio of the determinants are equal to the ratio of volumes of the ellipsoids centered at x_0 and y. Thus, the geometric argument would be that this way the Dikin walk can avoid extreme corners of the region)
6. repeat until obtained number of desired points



(a) Step 1 (b) Step 2

(c) Step 3 Case I (d) Step 3 Case II

(e) Step 4 (f) Step 5

Figure 5: The Dikin Walk begins by constructing the Dikin Ellipsoid at the starting point x_0 (Step 1). An uniformly random point y is generated in the Dikin Ellipsoid centered at x_0 (Step 2). If point x_0 is not in the Dikin Ellipsoid centered at y, then reject y (Step 3 Case I). If point x_0 is contained in the Dikin Ellipsoid centered at y, then accept y with probability $\min(1, \sqrt{\frac{det(H_y)}{det(H_{x_0})}})$ (Step 3 Case II). Once we've successfully accepted y, we set y as our new point, x_1 (Step 4). Algorithm repeats (Step 5)

How to pick a random point uniformly from a Dikin Ellipsoid?

Let's say, we now have D_x^r, the Dikin Ellipsoid centered at x with radius r.

1. generate ζ from the n dimensional Standard Gaussian (i.e. `zeta = rnorm(n,0,1)`)
2. normalize ζ to be on the n dimensional ball with radius r, that is:
$$\zeta \quad = \quad < x_1, x_2, ..., x_n > \quad \rightarrow \quad < \frac{rx_1}{\sqrt{x_1^2+x_2^2+...+x_n^2}}, \frac{rx_2}{\sqrt{x_1^2+x_2^2+...+x_n^2}}, ..., \frac{rx_n}{\sqrt{x_1^2+x_2^2+...+x_n^2}} >$$
3. Solve for d in the matrix equation $H_x d = A^T D\zeta$ (note, as long as x_0 is not on the boundary of our polytope K, H_x will be non-singular, thus, d will always be unique)
4. $y = x_0 + d$ is our randomly sampled point from D_x^r

Important Theorem

In algorithm `Dikin`, what if the point y we accept is outside of our polytope K? Luckily, there is no need to worry about that because of the following theorem:

Theorem – If $x_0 \in K$, then $D^1_{x_0} \subseteq K$. That is, if our starting point x_0 is in our polytope K, then the Dikin Ellipsoid centered at x_0 with radius 1 will always be contained in K.

Thus, if we set $r = 1$ (or $r \leq 1$), then our algorithm will guarantee to sample points only within the polytope K.

Using `walkr` to sample points

`walkr` has one main function `walkr` which sample points.

First, the user must specify the A and b in the $Ax = b$ hyperplanes equation. For example, for a 5 dimensional case:

```
library(walkr)
A <- matrix(c(1, 0, 1, 0, 1), ncol = 5)
b <- 0.5
A
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    1    0    1
```

```
b
```

```
## [1] 0.5
```

The `walkr` function takes care of the intersection with the simplex for the user. However, if the user just want to sample from the simplex with no hyperplanes, she can simply specify the simplex equation, as `walkr` will remove linearly dependent rows internally.

```
A <- matrix(1, ncol = 5)
```

After we have `A` and `B`, we are now ready for sampling. `walkr` has the following parameters:

- `A` is the right hand side of the matrix equation $Ax = b$
- `b` is the left hand side of the matrix equation $Ax = b$
- `method` is the method of sampling – can be either `"hit-and-"run"` or `"dikin"`
- `thin` is the thinning parameter. Every `thin`-th point is stored into the final sample
- `burn` is the burning parameter. The first `burn` points are deleted from the final sample
- `chains` is the number of chains we want to sample. Every chain is an element of the list which `walkr` eventually returns

`walkr` warns the user if the chains have not mixed "well-enough" or have not converged to a stationary distribution according to the Gelman-Rubin Diagnostics (Gelman and Rubin (1992)).

```
## sampling from the 3D simplex
library(walkr)
A <- matrix(1, ncol = 3)
b <- 1
answer <- walkr(A = A, b = b, points = 10, method = "hit-and-run",
                thin = 1, burn = 0, chains = 5)
```

```
## Warning in walkr(A = A, b = b, points = 10, method = "hit-and-run", thin = 1, : there
are parameters with rhat > 1.1, you may want to run your chains for longer
```

```
answer
```

```
##           [,1]       [,2]         [,3]       [,4]      [,5]       [,6]
## [1,] 0.1867770 0.04642564 0.9384501529 0.92146693 0.5671380 0.34476057
## [2,] 0.5682799 0.50464516 0.0612996080 0.04743140 0.3277182 0.63232867
## [3,] 0.2449431 0.44892920 0.0002502391 0.03110167 0.1051438 0.02291075
##           [,7]      [,8]      [,9]      [,10]
## [1,] 0.4254227 0.4418168 0.1234434 0.11365284
## [2,] 0.4368319 0.4395879 0.4989029 0.82395547
## [3,] 0.1377454 0.1185953 0.3776536 0.06239168
```

As we see from the code chunk above, `walkr` returns a warning message that the $\hat{R}$ are not satisfying, since we only have 10 points. However, if we run the chain for long (or increasing thinning), then the statistical tests pass.

```
library(walkr)
answer <- walkr(A = A, b = b, points = 1000, method = "hit-and-run",
                thin = 1, burn = 0, chains = 5)
```

Dikin versus Hitandrun

	`hit-and-run`	`Dikin Walk`
Uniform Sampling	Yes, needs $O(N^3)$ points, where N is the dimension of the polytope	No, concentrates in the interior
Mixing	$O(\frac{N^2R^2}{r^2})$ *, slows down substantially as dimension of polytope increases and polytope becomes "skinnier"	$O(MN)$, where A is $M \times N$; much stronger mixing.
Cost of One Step	$O(MN)$	$O(MN^2)$, in practice, one step of Dikin is much more costly than hit-and-run
Rejection Sampling	No	Yes (see probability formula and $x \notin D_y$), but rejection rate not high

*R is the radius of the smallest ball that contains the polytope K. r is the radius of the largest ball that is contained within the polytope K. Thus, $\frac{R}{r}$ increases as the polytope is "skinnier"(Kannan and Narayanan).

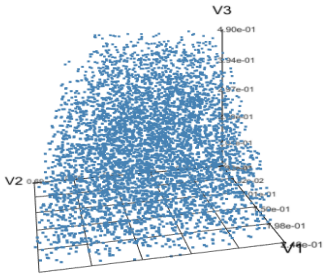


Figure 6: Dikin concentrates at the center

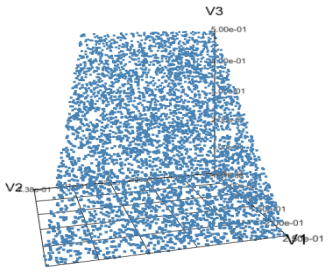


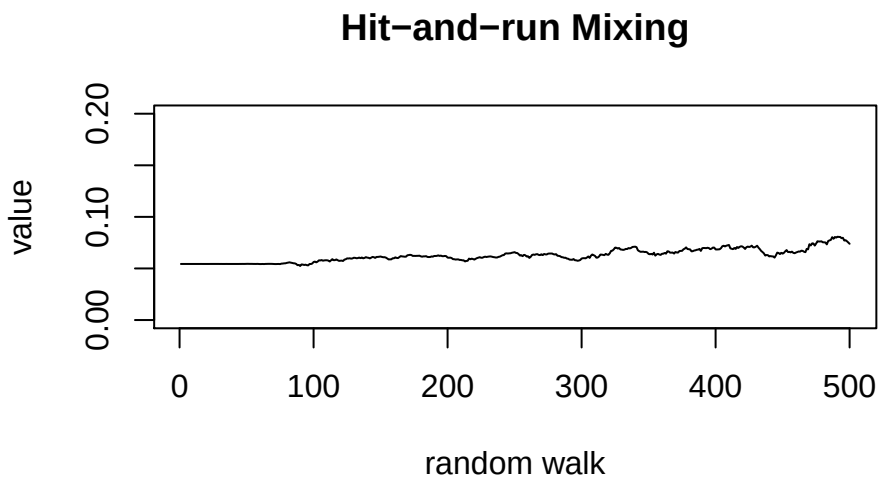
Figure 7: Hit-and-run samples the space thoroughly

As we can in the two trace-plots below, the mixing for Dikin is much better than hit-and-run given the same set of parameters

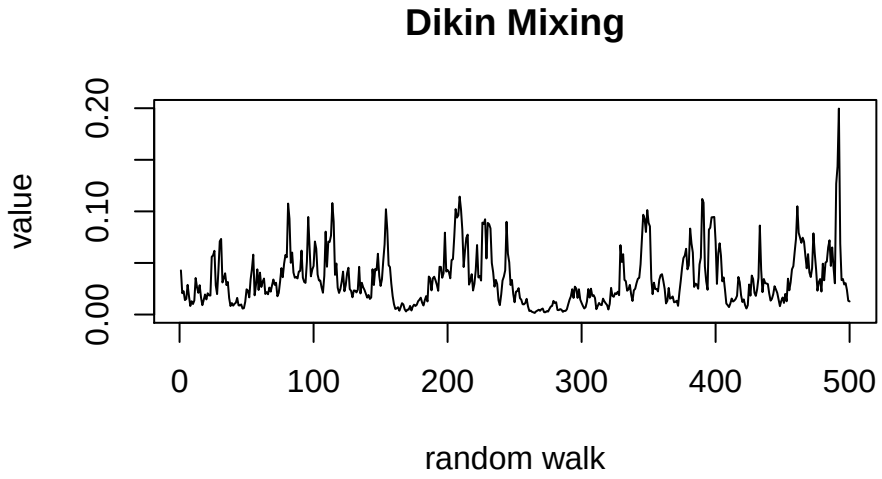
```
set.seed(314)
N <- 50
A <- matrix(sample(c(0,3), N, replace = T), nrow = 1)
A <- rbind(A, matrix(sample(c(0,3), N, replace = T), nrow = 1))
b <- c(0.7, 0.3)

answer_hitandrun <- walkr(A = A, b = b, points = 500, method = "hit-and-run",
                thin = 10, burn = 0, chains = 1)
answer_dikin <- walkr(A = A, b = b, points = 500, method = "dikin",
                thin = 10, burn = 0, chains = 1)

plot(y = answer_hitandrun[50,], x = 1:500,
     xlab = "random walk", ylab = "value", type = 'l',
     main = "Hit-and-run Mixing",
     ylim = c(0, 0.2))
```




```
plot(y = answer_dikin[50,], x = 1:500,
     xlab = "random walk", ylab = "value", type = 'l',
     main = "Dikin Mixing",
     ylim = c(0, 0.2))
```



Conclusion

`walkr` samples x that satisfy the underdetermined matrix equation $Ax = b$, every component of x is non-negative, and every component sum to 1. `walkr` internally performs the affine transformation which transforms $Ax = b$ and the simplex constraints into the $Ax \leq b$ form. Then, `walkr` samples points under this transform, takes the inverse transform, and finally returns the x's sampled.

The two methods that the user may specify is `"dikin"` and `"hit-and-run"`. Hit-and-run is a widely used sampling method that converges to the uniform sampling asymptotically, but fails to mix well in higher dimensions. Dikin generates a nearly uniform sample, but mixes stronger than hit-and-run, especially in higher dimensions. In addition, `walkr` will warn the user if the mixing is not strong enough according to the Gelman-Rubin Diagnostics.

Authors

Andy Yao
Mathematics and Physics
Williams College
Williamstown, MA, USA
ay3@williams.edu

David Kane
Managing Director
Hutchin Hill Capital
101 Federal Street, Boston, USA
dave.kane@gmail.com

Bibliography

D. Bates and D. Eddelbuettel. Fast and elegant numerical linear algebra using the RcppEigen package. *Journal of Statistical Software*, 52(5):1–24, 2013. URL http://www.jstatsoft.org/v52/i05/. [p1]

D. Eddelbuettel and R. François. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011. URL http://www.jstatsoft.org/v40/i08/. [p1]

A. Gelman and D. B. Rubin. Inference from iterative simulation using multiple sequences. *Statistical Science*, 7(4):457–511, 1992. URL https://projecteuclid.org/download/pdf_1/euclid.ss/1177011136. [p11]

R. Kannan and H. Narayanan. Random Walks on Polytopes and an Affine Interior Point Method for Linear Programming. *Mathematics of Operations Research*. [p9, 12]

G. van Valkenhoef and T. Tervonen. *hitandrun: "Hit and Run" and "Shake and Bake" for Sampling Uniformly from Convex Shapes*. CRAN. [p6]