walkr: MCMC Sampling from Convex Polytopes

by Andy Yao, David Kane

Abstract Consider the intersection of two spaces: the complete solution space to $Ax = b$ and the N-Simplex, described by $\sum_{i=1}^{N} x_i = 1$ and $x_i \geq 0$. The intersection of these two spaces is a convex polytope. The R package **walkr** samples from this intersection using two Monte-Carlo Markov Chain (MCMC) methods: hit-and-run and Dikin walk. **walkr** also provide tools to examine sample quality.

Introduction

Consider all possible vectors x that satisfy the matrix equation $Ax = b$, where A is $M \times N$, x is $N \times 1$, and b is $M \times 1$. The problem is only interesting when there are more rows than columns ($M < N$). If $M = N$, then there is a single solution, and if $M > N$, there then are, in general, no solutions. If the rows of A are linearly dependent, the rows can be reduced until they are linearly independent without affect the solution space. Therefore, we can assume that the rows are linearly independent going forward.

Geometrically, every row in $Ax = b$ describes a hyperplane in $\mathbb{R}^N$. Therefore, $Ax = b$ represents the intersection of M unbounded hyperplanes in $\mathbb{R}^N$. We bound the sample space by also requiring vector x to be in the N-simplex. The **N-simplex**:

$$\begin{aligned} x_1 + x_2 + x_3 + ... + x_N &= 1 \\ x_i \geq 0, \qquad &\forall i \in \{1, 2, ..., N\} \end{aligned}$$

The N-simplex is a $N - 1$ dimensional object living in N dimensional space. For example, the 3D simplex is a 2 dimensional triangle in 3D space (Figure 1).

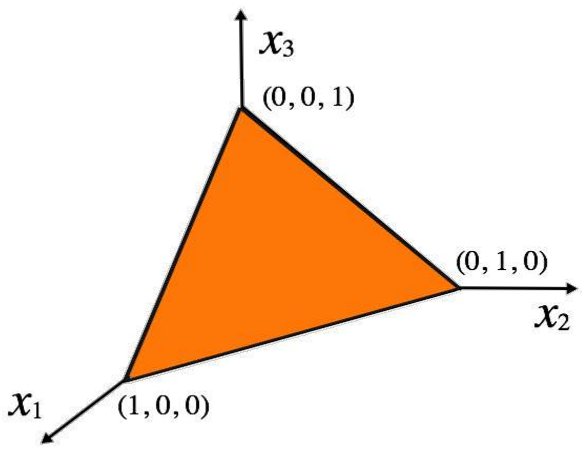


Figure 1: The 3D simplex is a 2 dimensional triangle in 3 dimensional space. The vertices of the simplex are $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$.

The intersection of the complete solution of $Ax = b$ and the N-simplex is a non-negative convex polytope. Sampling from such a convex polytope is a difficult problem, and the common approach is to run Monte-Carlo Markov Chains (MCMC) in the polytope (citation to some math book). **walkr** includes two MCMC algorithms: hit-and-run and Dikin walk. Hit-and-run is guarantees uniform sampling asympotically, but mixes increasingly slowly for higher dimensions of A (cite hitandrun here). Dikin walk generates a "nearly" uniform sample — favoring points away from the edges of the polytope — but exhibits much faster mixing (Kannan and Narayanan).

MCMC methods generally involve the creation of multiple random walks from different starting points, each of which is an indepedent "chain". To check for the quality of the samples, one step would be to check for signs that the chains have mixed well enough with each other. **walkr** allows the user to examine the quality of the samples

3 Dimensional Example

Consider one linear constraint in three dimensions.

$$x_1 + x_3 = 0.5$$

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

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

Figure 2 shows the intersection of the 3D simplex with $Ax = b$.

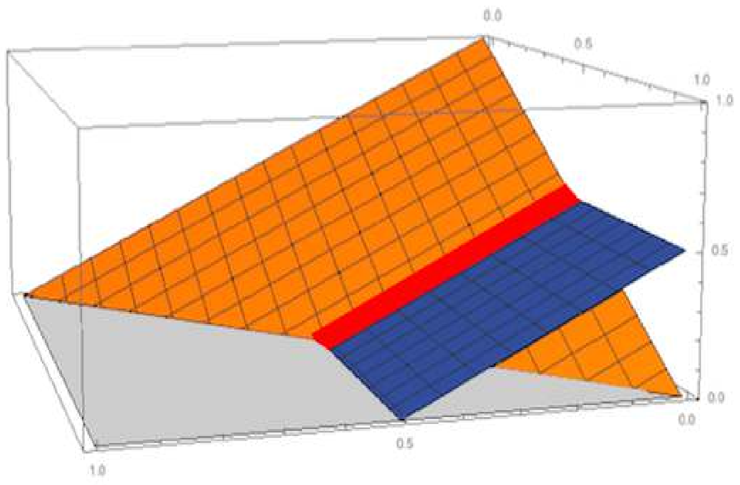


Figure 2: The orange triangle is the 3D-simplex. The blue plane is the hyperplane $x_1 + x_3 = 0.5$. The red line is their intersection, which is the space we're interested in sampling from. The end points are $(0, 0.5, 0.5)$ and $(0.5, 0.5, 0)$

4 Dimensional Example

Just as the 3D simplex is a 2D surface living in 3D space, the 4D simplex (i.e. $x_1 + x_2 + x_3 + x_4 = 1$, $x_i \geq 0$) can be viewed as a 3D object, as in Figure 3. 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 $(0, 0, 0, 1)$.

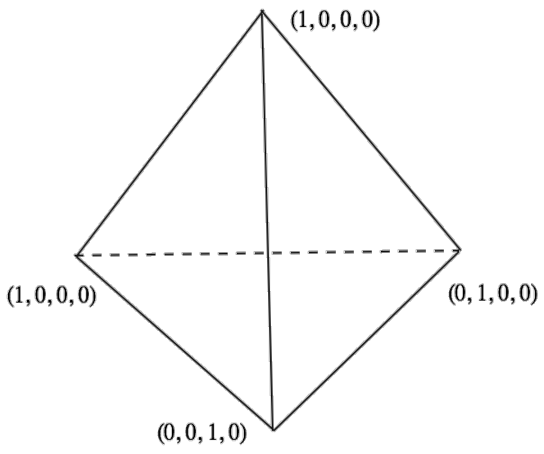


Figure 3: The 4D simplex exists in 4D space, but can be viewed as a 3D object. Specifically, the 4D simplex is a tetrahedron, with all four sides equilateral triangles.

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 Figure (below). The resulting shape is a trapezoid in 4D space.

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

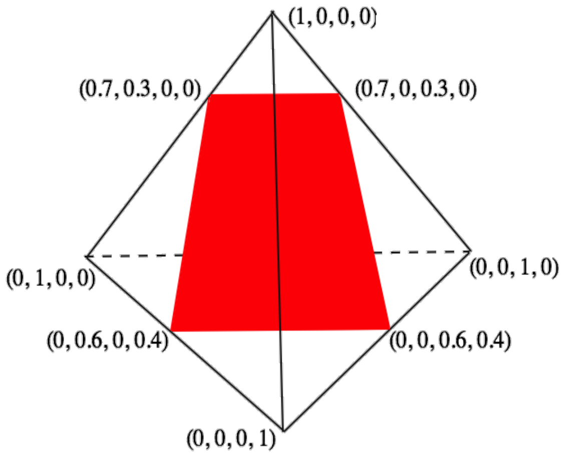


Figure 4: The 4D simplex is the tetrahedron. The hyperplane cuts through the tetrahedron, forming a trapezoid as the intersection (in red). This trapezoid is our sample space, as it is the intersection of the hyperplane with the 4D simplex. The vertices of the trapezoid are $(0.7, 0.3, 0, 0)$, $(0.7, 0, 0.3, 0)$, $(0, 0.6, 0, 0.4)$, and $(0, 0, 0.6, 0.4)$

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 x-space to α-space

Our sample space is a bounded, non-negative convex polytope. In the literature, convex polytopes are commonly described by a generic $Ax \leq b$. In this section, we present a 3 step procedure which transforms the intersection of $Ax = b$ and the N-simplex into the $Ax \leq b$ form (**note**: the A and b in $Ax \leq b$ is not the same as those which we began with in $Ax = b$. We use A and b in both occasions because it is standard notation for describing both a matrix equation and a convex polytope.)

Step 1: Combining Simplex Equality with $Ax = b$

Recall that A in $Ax = b$ is $M \times N$:

$$A_{M\times N} = \overbrace{\begin{bmatrix} & & \\ & \dots & \\ & & \end{bmatrix}}^{\text{N columns}} \left.\vphantom{\begin{bmatrix} \\ \\ \\ \end{bmatrix}}\right\}\text{M rows}$$

Add an extra row in $Ax = b$ which captures the equality part of the simplex constraint ($x_1 + x_2 + ... + x_N = 1$). Call this new matrix A':

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

Step 2: Solving for the Null Space – Transforming to α-space

Second, we find all possible x's that satisfy $A'x = b'$. To do so, we must first compute the null space of A' and then add on any particular solution that satisfy $A'x = b'$. See Leon's Linear Algebra textbook for a review of these basic results (Leon (2014)).

The Null Space of A' can be represented by $N - (M + 1)$ basis vectors, since we have N variables and $M + 1$ constraints in A'. Every basis vector, v_i, has N components:

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

Once we have the basis vectors, we could express the set of all x's that satisfy $A'x = b'$ in terms of coefficients α_i. The intuition would be that the basis vectors form a coordinate system in the complete solution space, and that the coefficients α_i's represent linear combinations of basis vectors to cover the whole space. The complete solution to $A'x = b'$ could be expressed as the set:

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

Step 3: Including the Simplex Inequalities

We add the inequality constraints from the N-simplex. We require every element of vector x to be ≥ 0:

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

We express all coefficients α_i as a vector α:

$$\alpha = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_{N-(M+1)} \end{bmatrix}$$

We can also express the set of basis vectors as columns of matrix V:

$$V = \begin{bmatrix} v_1 & v_2 & ... & v_{N-(M+1)} \end{bmatrix}$$

Therefore, the inequality now becomes:

$$v_{particular} + V\alpha \geq \begin{bmatrix} 0 \\ 0 \\ ... \\ ... \\ ... \\ 0 \end{bmatrix}$$

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

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

Here is the generic $Ax \leq b$ representation of a convex polytope (note again that the A and b in $Ax \leq b$ are different from those in $Ax = b$). We have performed a **transformation** from x-space to α-space. In fact, **walkr** internally performs this transformation, samples α, then maps it back to x-space.

We have performed a transformation, going from describing the polytope in terms of the intersection of $Ax = b$ and the N-simplex to the general $Ax \leq b$ form. This is necessary because the following sample algorithms we are going to present rely on the fact that the polytope is described in the $Ax \leq b$ form.

Algorithms

Important: Before moving to sampling algorithms, we clear up on some nomenclature. First, as established above, the sampling space is a convex polytope. We shall refer to this convex polytope as K. Moreover, because it is very standard notation in the math and sampling literature to describe a convex polytope as $Ax \leq b$, we describe K with $Ax \leq b$. However, we keep in mind that the A, x, and b are not the same as those in the original problem statement of $Ax = b$ and N-simplex. Instead, as described in the section above, we are actually sampling coefficients α's, and then eventually transforming it back to x-space. Again, $Ax \leq b$ describe the polytope in terms of transformed coordinates α.

Picking Starting Point

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

$$\begin{aligned} \text{minimizes} \quad & |Cx - d|^2 \\ \text{subject to} \quad & Ax \leq b, \quad \text{this is our polytope } K \end{aligned}$$

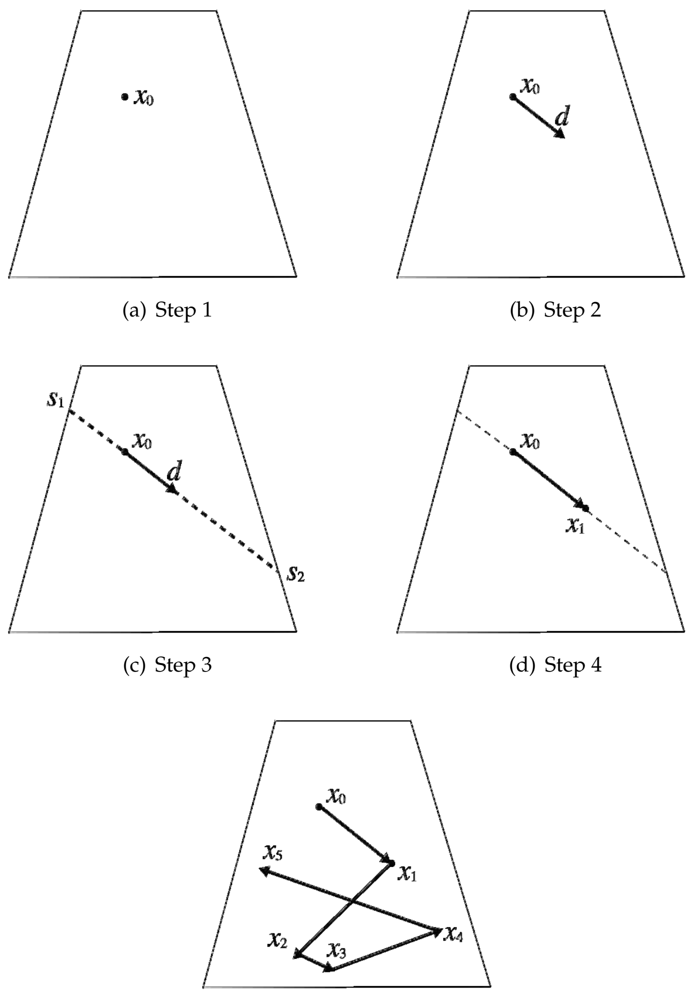
We randomly generate C and d, obtaining points that minize the random objective function. Because we are performing a minimization, the points that are obtained this way often fall onto the boundaries of our polytope K. Therefore, we repeat the process 30 times in **walkr** and then take an average of those points. This averaged point ought to be still in the polytope K because our polytope is convex. This procedure generates one starting point x_0.

Often times in MCMC random walks, it is beneficial to have multiple starting points at different locations in K. We call each random walk at a starting point a single chain. For example, if we have 5 different starting points, each running its own random walk, we would have 5 chains. Once we have multiple chains, we could examine whether the chains have all converged to the same stationary distribution. The correlation between different chains is also of interest. Ideally, we hope to see a low correlation which reflects that the chains are mixing well with each other.

Hit-and-run

We point out that the $Ax \leq b$ described below is the transformed polytope in α-space (discussed in section above). The A, x, and b are different from those in the original $Ax = b$ and N-Simplex. Once we sampled points, we perform the transformation which maps the points back into the original x-space.

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



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

(e) Step 5

Figure 5: 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)

walkr uses the `har` function from the **hitandrun** package on CRAN (van Valkenhoef and Tervonen) to implement hit-and-run.

The hit-and-run algorithm asymptotically generates an uniform sample in the convex polytope. The cost of each step for hit-and-run is also relatively inexpensive. However, as the dimensions of the polytope increases, the mixing of hit-and-run becomes increasingly slower.

Dikin Walk

Dikin walk is another method of MCMC random walks. In higher dimensions, the Dikin walk mixes much stronger than hit-and-run does. This desirable mixing effect is due to the fact that Dikin walk favors points that are far away from the edges of the polytope, thereby sampling a "nearly uniform" sample (Kannan and Narayanan).

Definitions

Recall, our sampling space is a convex polytope. We call this convex polytope K, which can be described in the form $Ax \leq b$. We point out that this $Ax \leq b$ is the transformed polytope in α-space (discussed in transformation section above). The A, x, and b are different from those in the original $Ax = b$ and N-Simplex.

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, because the negative log function tends to infinity as its argument tends to zero. The value of ϕ gets larger and larger as $a_i^T x$ gets closer and closer to b_i.

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. The second derivative of $-\log(z)$ is $\frac{1}{z^2}$, which also tends to infinity as z tends to zero.

Dikin Ellipsoid $D_{x_0}^r$

Define $D_{x_0}^r$, the Dikin Ellipsoid centered at x_0 with radius r as:

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

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 $\Delta \frac{1}{z^2}$ on the $\frac{1}{z^2}$ graph. For x_0's that are far away from the boundary, given an allowed $\Delta \frac{1}{z^2}$ (captured in r), the range of allowed values of $y - x_0$ is very large. This corresponds to having a large Dikin ellipsoid. Alternatively, if x_0 is near the boundary, then given an allowed $\Delta \frac{1}{z^2}$, only a small range of $y - x_0$ is acceptable.

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 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.

Algorithm

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. This is counter-intuitive because normally we would think that y must be contained in the Dikin ellipsoid of x_0 to be accepted. However, this step actually says that if the current point x_0 is not in the Dikin Ellipsoid of the potential point y, then we reject the point y.
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

Using walkr

The **walkr** package has one main function walkr which samples points. 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
- ret.format is the return format of the sampled points. If "list", then a list of chains is returned, with each chain as a matrix of points. If "matrix", then a single matrix of points is returned. A column is a sampled point.

Simple 3D Simplex

To sample from the 3D simplex, the user can simply specify the simplex equation in $Ax = b$, as walkr will remove linearly dependent rows internally.

```
library(walkr)

## Loading required package: ggplot2

A <- matrix(1, ncol = 3)
b <- 1
sampled_points <- walkr(A = A, b = b, points = 1000,
                        method = "hit-and-run", chains = 5, ret.format = "list")
```

Now, sampled_points contain 1000 points from the 3D simplex. We can visualize the MCMC random walks by calling the explore_walkr function, which launches a shiny interface from **shinystan**.

```
explore_walkr(sampled_points)
```

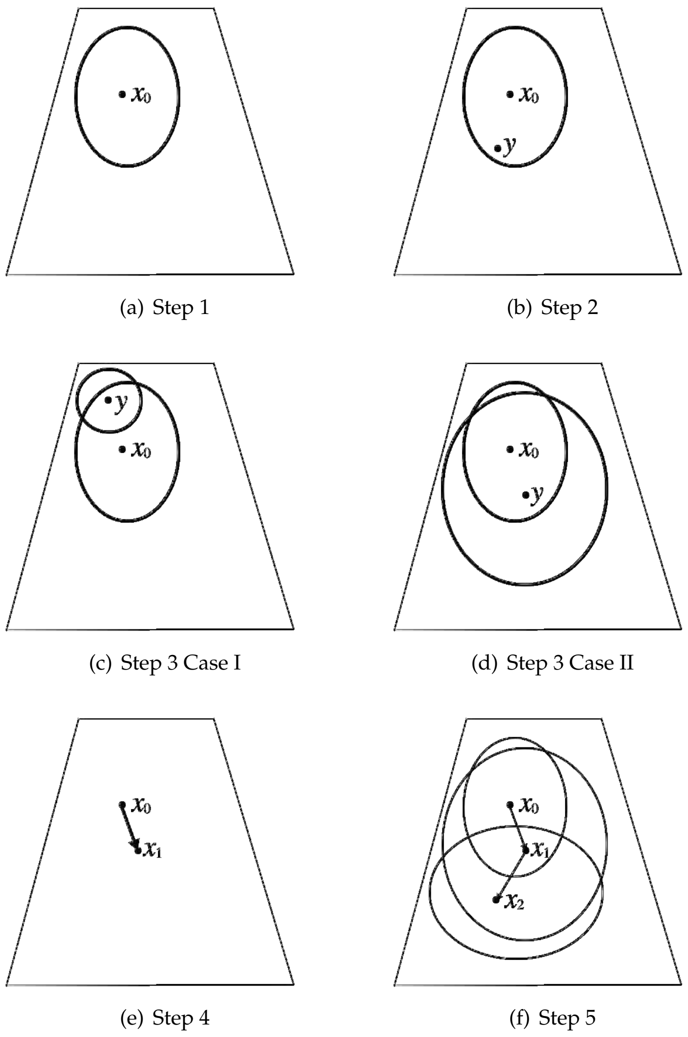


Figure 6: The Dikin Walk begins by constructing the Dikin Ellipsoid at the starting point x_0 (Step 1). An uniformly random point y is generated in the Dikin Ellipsoid centered at x_0 (Step 2). If point x_0 is not in the Dikin Ellipsoid centered at y, then reject y (Step 3 Case I). If point x_0 is contained 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)

Higher Dimensions with Constraints

Sampling from higher dimensions follows the same syntax. The user just has to specify an A and b. Note that `walkr` automatically intersects $Ax = b$ with the N-Simplex, so that the user does not have to include the simplex equation in $Ax = b$.

```
## two 20 dimensional constraints

set.seed(314)
A <- matrix(sample(c(0,1,2), 40, replace = TRUE), ncol = 20)
b <- c(0.5, 0.3)
sampled_points <- walkr(A = A, b = b, points = 1000, chains = 5,
                        method = "hit-and-run", ret.format = "matrix")

## Warning in walkr(A = A, b = b, points = 1000, chains = 5, method = "hit-and-run", :
there are parameters with rhat > 1.1, you may want to run your chains for longer
```

As we could see from the warning message above, `walkr` internally 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)). It is suggested that if any of the parameter's

$\hat{R}$ value is above 1.1, then the chains should run longer, or equivalently, we could increase the `thin` parameter.

```r
set.seed(314)
sampled_points <- walkr(A = A, b = b, points = 1000, chains = 5, thin = 100,
                        method = "hit-and-run", ret.format = "matrix")
sampled_points <- walkr(A = A, b = b, points = 1000, chains = 5, thin = 10,
                        method = "dikin", ret.format = "matrix")
```

As we could see from the code chunk above, Dikin walk only required `thin` to be 10, whereas hit-and-run needed a `thin` parameter of 100. This is a sign that Dikin mixes faster than hit-and-run does. As dimensions ramp up to the hundreds, this rapid mixing behavior of Dikin compared to hit-and-run is even more obvious.

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

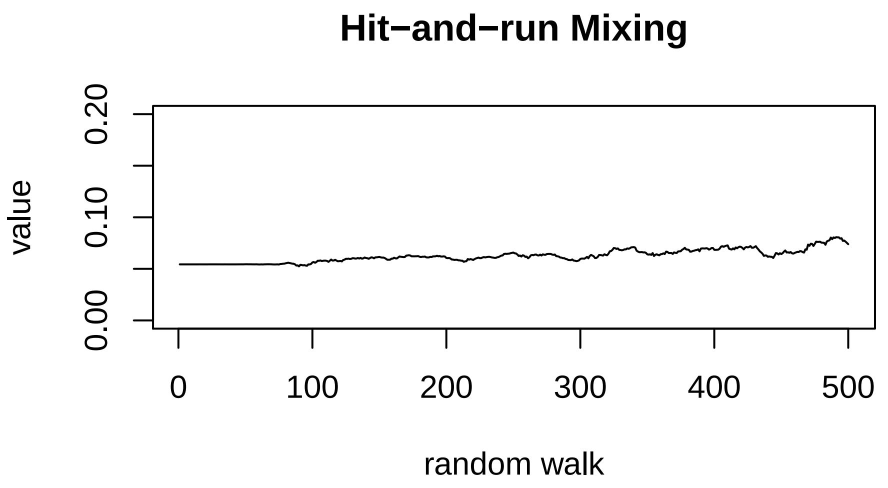
*R is the radius of the smallest ball that contains the polytope K. r is the radius of the largest ball that is contained within the polytope K. Thus, $\frac{R}{r}$ increases as the polytope is "skinnier"(Kannan and Narayanan).

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

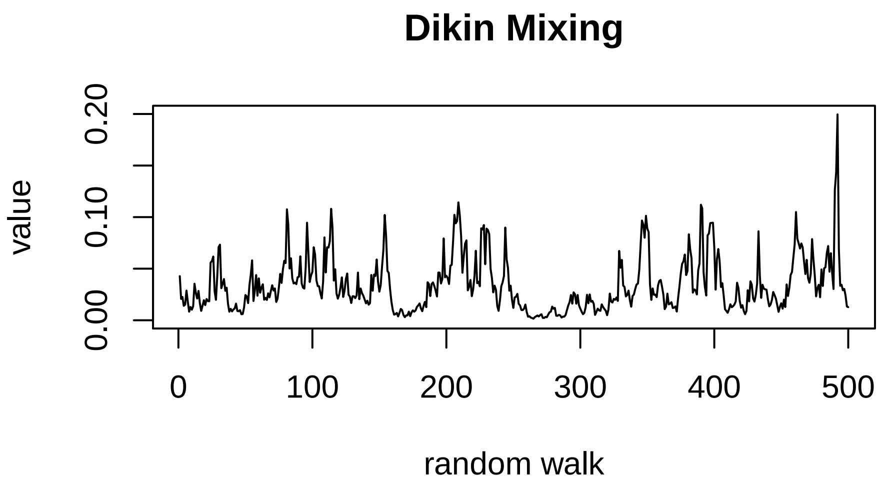
```r
set.seed(314)
N <- 50
A <- matrix(sample(c(0,3), N, replace = T), nrow = 1)
A <- rbind(A, matrix(sample(c(0,3), N, replace = T), nrow = 1))
b <- c(0.7, 0.3)

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 uses MCMC random walks to sample x from the intersection of two spaces. The first space is the complete solution space to the underdetermined matrix equation $Ax = b$, where A is a $M \times N$ matrix, with $M < N$. The second space is the N-Simplex, described by equation $x_1 + x_2 + ... + x_N = 1$ and inequalities $x_i \geq 0$ for all $i \in 1, 2, ..., N$. This intersection is a convex polytope.

In order to sample from this convex polytope, we perform an affine transformation which transforms from x-space to α-space, in which the α's are coefficients of basis vectors. Through this transformation, we are able to re-express our sampling space as a generic $Ax \leq b$ convex polytope, which allows us to perform our sampling algorithms. Once the sampling is completed, **walkr** maps the points back to the original coordinate system and returns them to the user. This is all done internally by **walkr**, as the user only need to specifcy the original A and b in $Ax = b$.

walkr implements two MCMC sampling algorithm – hit-and-run and Dikin walk. The package also provides MCMC convergence diagnostics of the random walk's mixing, as well as a link to the **shinystan** package which enables visualization. Hit-and-run is a widely used MCMC algorithm that guarantees uniform convergence asymptotically and has a relatively low computation cost for one step. However, as the dimension of our convex polytope increases, hit-and-run mixes increasingly slower. Dikin walk is an alternative MCMC algorithm that samples nearly uniformly, favoring points away from the edges of the polytope. While Dikin walk is only nearly uniform, it exhibits much stronger mixing in high dimensions than hit-and-run does.

Authors

Andy Yao
Mathematics and Physics
Williams College
Williamstown, MA, USA
andy.yao17@gmail.com

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

Bibliography

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. [p9]

R. Kannan and H. Narayanan. Random Walks on Polytopes and an Affine Interior Point Method for Linear Programming. *Mathematics of Operations Research*. [p1, 7, 10]

S. J. Leon. *Linear Algebra with Applications*. Pearson, 2014. [p4]

G. van Valkenhoef and T. Tervonen. *hitandrun: "Hit and Run" and "Shake and Bake" for Sampling Uniformly from Convex Shapes*. CRAN. [p7]