walkr: MCMC Sampling from Non-Negative 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 non-negative 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 provides 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 interesting when there are more rows than columns ($M < N$). In general, if $M = N$, then there is a single solution, and if $M > N$, then there are no solutions. If the rows of A are linearly dependent, rows can be eliminated until they are linearly independent without changing the solution space. Assume that the rows of A are linearly independent going forward.

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

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

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

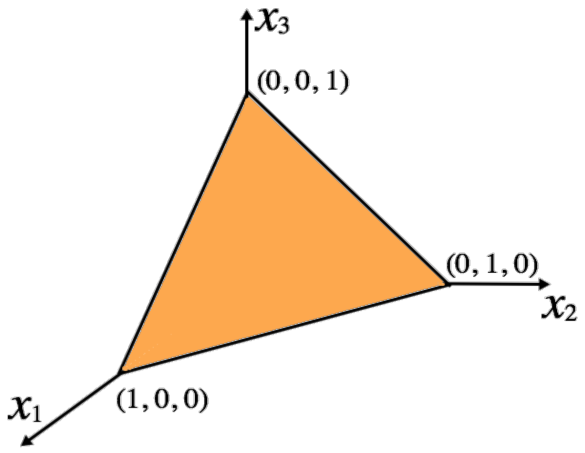

Figure 1: The 3D simplex is a two dimensional triangle in three dimensional space. The vertices of the simplex are $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. x_1, x_2, and x_3 are all greater than or equal to 0, and for all points on the simplex, the sum of x_1, x_2 and x_3 equals 1.

The intersection of the complete solution to $Ax = b$ and the N-simplex is a non-negative convex polytope. Sampling from such an object is a difficult problem, and the common approach is to run Monte-Carlo Markov Chains (MCMC) (Kannan and Lovasz (1997)). MCMC methods begin at a starting point in the solution space and then randomly "wander" through the space according to an algorithm. An important feature of MCMC is that every step depends only on the current location and not on the steps taken previously.

MCMC sampling generally involve the creation of multiple random walks from different starting points, each of which is an independent "chain". A key aspect of running multiple chains from different starting points is to examine the "mixing" of the sample. Good mixing means that the different chains (from different starting points) have overlapped with each other, suggesting that they have thoroughly moved around the sample space. While good mixing does not guarantee a correct sample of the polytope, poor mixing means poor sampling. **walkr** examines the quality of MCMC samples.

walkr includes two MCMC algorithms: hit-and-run and Dikin walk. Hit-and-run guarantees uniform sampling asympotically, but mixes more slowly as the number of columns in A increase (Vempala (2005)). Dikin walk generates a non-uniform sample — favoring points away from the edges of the polytope — but exhibits much faster mixing (Kannan and Narayanan (2009)).

Three 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 $Ax = b$.

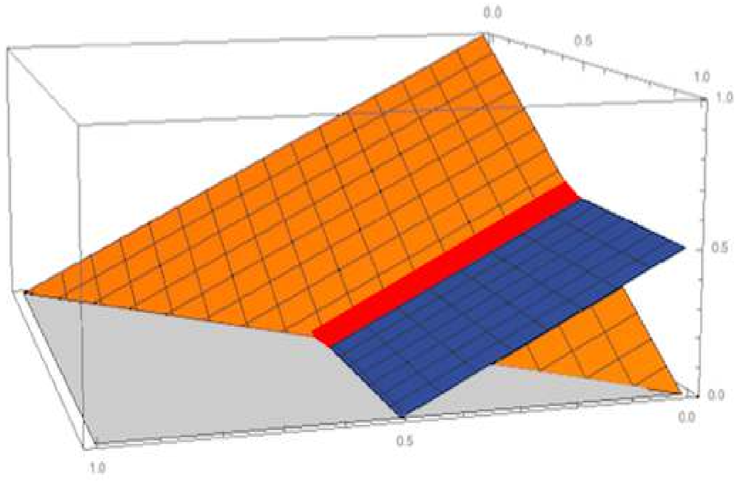

Figure 2: The orange triangle is the 3D-simplex. The blue plane is the hyperplane $x_1 + x_3 = 0.5$. The red line segment is their intersection, which is our sample space. The end points of the line segment are $(0.5, 0.5, 0)$ and $(0, 0.5, 0.5)$.

Four dimensional example

Just as the 3D-simplex is a 2D surface, 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 a tetradhedron when viewed from 3D space, with verticies $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, $(0, 0, 1, 0)$, and $(0, 0, 0, 1)$.

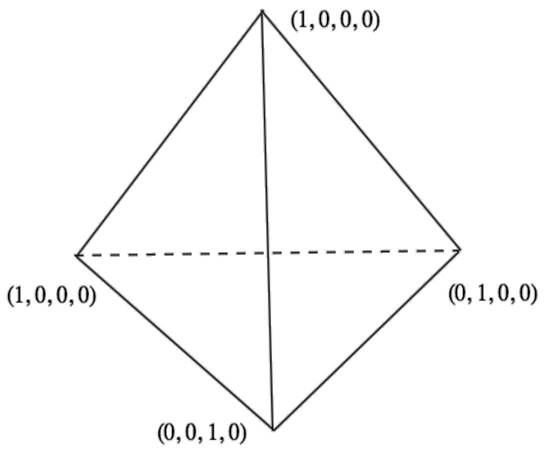

Figure 3: The 4D-simplex exists in 4D space, but can be viewed as a 3D object. When viewed from three dimensional space, the 4D-simplex is a tetrahedron, with all four sides equilateral triangles.

Figure 4 shows the intersection of the 4D-simplex with a hyperplane (1 equation, or 1 row in $Ax = b$). The resulting convex polytope is a 2D trapezoid in 4D space. Note that this convex polytope is $4 - (1 + 1) = 2$ dimensional. This is because we began with 4 dimensions, and the constraint and the simplex each reduced the dimension of the solution space by 1.

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

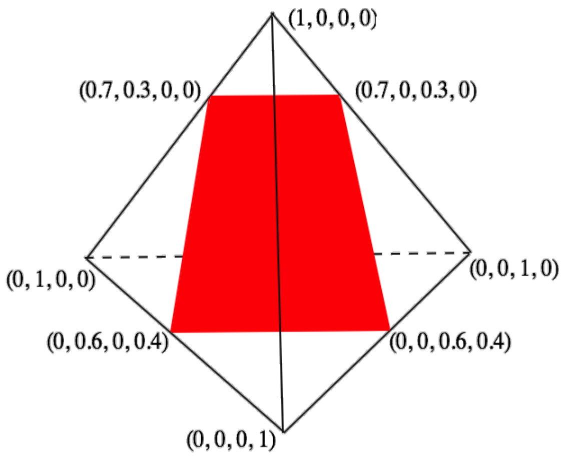

Figure 4: The 4D-simplex is a tetrahedron when projected to 3D space. The hyperplane $22x_1 + 2x_2 + 2x_3 + 37x_4 = 16$ cuts through the tetrahedron, forming a trapezoid as the intersection (in red). This trapezoid is our sample space, as it is the intersection of the hyperplane with the 4D-simplex. 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 $\mathbb{R}^N$. Geometrically, our sample space is the intersection of M hyperplanes with the N-simplex, which will be $N - (M + 1)$ dimensional.

x-space and 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 order to use the sampling algorithms, we must first re-express the sample space in the form $Ax \leq b$ (with different A, x and b) [1].

Recall that our sample space is the intersection of the complete solution to $Ax = b$ and the N-simplex, which consists of three components: first, the matrix equation $Ax = b$, second, the simplex constraint $x_1 + x_2 + ... + x_N = 1$ and, third, the non-negative inequalities, $x_i \geq 0$. In this section, we combine all three parts into one single inequality of the form $Ax \leq b$.

Step 1: Combine the simplex equality with the original $Ax = b$

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

$$A = \overbrace{\begin{bmatrix} & & \\ & ... & \\ & & \end{bmatrix}}^{\text{N columns}} \Big\} \text{M rows}$$

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: Solve for the null space and transform to α-space

Find x that satisfies $A'x = b'$. First, solve for the null space of A', defined as all x that satisfy $A'x = 0$. The null space is spanned by $N - (M + 1)$ basis vectors, because that is the dimension of the sample space (our polytope). Any vector formed by a linear combination of the basis vectors will still be in the null space.

Second, find a particular solution. Think of the null space as constructing a coordinate system for $A'x = 0'$ and of the particular solution as an offset from the origin to fit $A'x = b'$. See Leon (2014) for a review of the specifics of finding null spaces and particular solutions.

The null space of A' can be represented by $N - (M + 1)$ basis vectors. Because we are in $\mathbb{R}^N$, 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 null space basis vectors and a particular solution, $v_{particular}$, we can express the set of all x's that satisfy $A'x = b'$ in terms of coefficients α_i. The complete solution to $A'x = b'$ can be expressed as the set:

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

This space is now described in terms of α_i's, the coefficients of the basis vectors. We call this "α-space".

Step 3: Include the simplex inequalities

We add the inequality constraints from the N-simplex, requiring every element of vector x to be greater than or equal to 0:

[1]This is a total abuse of notation. The A in $Ax = b$ is very different from the A in $Ax \leq b$. This new A is N by $N - (M + 1)$. x and b are also different. The mathematical literature for linear equations uses $Ax = b$, and the litearture on convex polytopes uses $Ax \leq b$, so it seemed best to use the same notation in both places in order to make connections to the existing literature clearer.

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

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:

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

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

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

Finally, our convex polytope is in the desired form $Ax \leq b$, which is $-V\alpha \leq v_{particular}$.

Four dimensional transformation example

Consider the four dimensional example from Figure 4.

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

Step 1: Add an extra row in $Ax = b$ to capture the simplex equality.

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

Step 2: The null space basis contain 2 vectors, as $M - (N+1) = 4 - (1+1) = 2$ is the dimension of our solution space. The null space basis vectors (to three decimal places) are:

$$v_1 = \begin{bmatrix} -0.103 \\ -0.680 \\ 0.723 \\ 0.059 \end{bmatrix}, \quad v_2 = \begin{bmatrix} -0.833 \\ 0.265 \\ 0.092 \\ 0.476 \end{bmatrix}$$

A particular solution to $A'x = b'$ is (any particular solution works):

$$v_{particular} = \begin{bmatrix} 0.212 \\ 0.147 \\ 0.359 \\ 0.274 \end{bmatrix}$$

Step 3: We add on the simplex inequalities:

$$v_{particular} + \alpha_1 v_1 + \alpha_2 v_2 \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Finally, we re-express the inequalities as $-V\alpha \leq v_{particular}$, which is of the form $Ax \leq b$

$$V = \begin{bmatrix} -0.103 & -0.833 \\ -0.680 & 0.265 \\ 0.723 & 0.092 \\ 0.059 & 0.476 \end{bmatrix}$$

$$\alpha = \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix}$$

$$\begin{bmatrix} 0.103 & 0.833 \\ 0.680 & -0.265 \\ -0.723 & -0.092 \\ -0.059 & -0.476 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} \leq \begin{bmatrix} 0.212 \\ 0.147 \\ 0.359 \\ 0.274 \end{bmatrix}$$

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

We sample α's according to our MCMC sampling algorithms. Then, we map the α's back as x's in the original coordinate system by:

$$x = v_{particular} + V\alpha$$

For example, an α we sample can be:

$$\alpha = \begin{bmatrix} -0.149 \\ -0.372 \end{bmatrix}$$

Apply the map to α, obtaining x in the original problem statement.

$$x = v_{particular} + V\alpha = \begin{bmatrix} 0.212 \\ 0.147 \\ 0.359 \\ 0.274 \end{bmatrix} + \begin{bmatrix} -0.103 & -0.833 \\ -0.680 & 0.265 \\ 0.723 & 0.092 \\ 0.059 & 0.476 \end{bmatrix} \begin{bmatrix} -0.149 \\ -0.372 \end{bmatrix} = \begin{bmatrix} 0.539 \\ 0.152 \\ 0.219 \\ 0.090 \end{bmatrix}$$

Indeed, the mapped point satisfies the original $Ax = b$ and the N-simplex.

Recall that we began with the intersection of the complete solution to $Ax = b$ and the N-simplex, represented as three different parts. First, we add the $x_1 + x_2 + ... + x_N = 1$ part of the simplex equation as an extra row to $Ax = b$. Second, we solve for a particular solution and the the null space of A', obtaining the matrix V which contains the null space basis vectors. Finally, we add the inequality constraints $x_i \geq 0$ to obtain $-V\alpha \leq v_{particular}$.

Going forward, we will not use the $-V\alpha \leq v_{particular}$ notation and will instead use $Ax \leq b$ to denote the same matrix inequality. This is a total abuse of notation, as the A, x and b in $Ax \leq b$ are actually $-V$, α and $v_{particular}$. We do this because in the convex polytopes literature, it is standard to represent the polytope as $Ax \leq b$, so it seemed best to use the same notation in order to make connections to existing literature easier.

Algorithms

Define non-negative convex polytope K to be the solution to $Ax \leq b$. We are interested in sampling uniformly from K.

The two Monte-Carlo Markov Chain (MCMC) sampling methods we implement are hit-and-run and Dikin walk. MCMC methods begin at a starting point in K and wander through K according to a specified algorithm. Every MCMC step depends only on the current location.

To test the quality of our sample, we create multiple, independent "chains" from different starting points in K and observe their "mixing". The chains have mixed well if their values have repeatedly overlapped with each other. If the chains have not mixed well, then we need to run the chains for longer (i.e. sample more points). While good mixing does not guarantee that the sample is perfect, poor mixing alone indicates a problem.

Starting point

MCMC random walks need a starting point, x_0 in K. **walkr** generates starting points using linear programming. Specifically, the `lsei` function of the **limSolve** package (den Meersche et al. (2009)) 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 matrix C and vector d. Solving this system generates an x which will usually fall on the boundary of polytope K. We repeat this process 30 times and take an average of those points, thereby generating one starting point x_0.

Hit-and-run

Vempala (2005) provides an overview of the hit-and-run algorithm:

1. Set starting point x_0 as current point.
2. Generate a random direction d from the N dimensional unit-sphere.
3. Find the chord S through x_0 along the directions d and $-d$. Define end points s_1 and s_2 as the intersection of the chord S with the edges of K. Because K is convex, the chord S will only intersect it at two points. Parametrize the chord S 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 $U[0, 1]$.
5. Set x_1 as current point.
6. Repeat algorithm until number of desired points sampled.

See Figure 5. **walkr** uses the `har` function from the **hitandrun** package (van Valkenhoef and Tervonen (2015)) to implement hit-and-run. The hit-and-run algorithm asymptotically generates an uniform sample in the convex polytope K. However, the mixing of hit-and-run becomes slower with increasing dimensions in K. Slow mixing is observed when we run multiple chains from different starting points. Instead of overlapping with each other, the individual chains tend to stay near their initial values. To solve this problem, we must let the individual chains run for longer (i.e. sample more points). However, as the dimensions of the polytope increase, the number of points we need to sample to ensure adequate mixing increases exponentially.

Dikin walk

A Dikin walk is the second of two MCMC methods implemented in the **walkr** package. A Dikin walk begins from a random starting point within the convex polytope K and then creates a Dikin ellipsoid centered at the current point. It then samples a random point from that ellipsoid, one whose shape and size are determined by both the current point and the shape of K.

Unlike hit-and-run, the Dikin walk does not sample uniformly over K. Instead, it is biased towards points that are way from the edges of K (Kannan and Narayanan (2009)). This bias allows the chains to mix more quickly than hit-and-run, and therefore, to work better in higher dimensions.

For non-negative convex polytope K, defined as all x for which $Ax \leq b$, define a_i as the i^{th} row of A. Define x_i and b_i as the i^{th} element of x and b respectively. The dimensions of A are $m = N$ by $n = N - (M + 1)$, where M and N are the dimensions of A in $Ax = b$, the original problem statement.

Log Barrier Function ϕ:

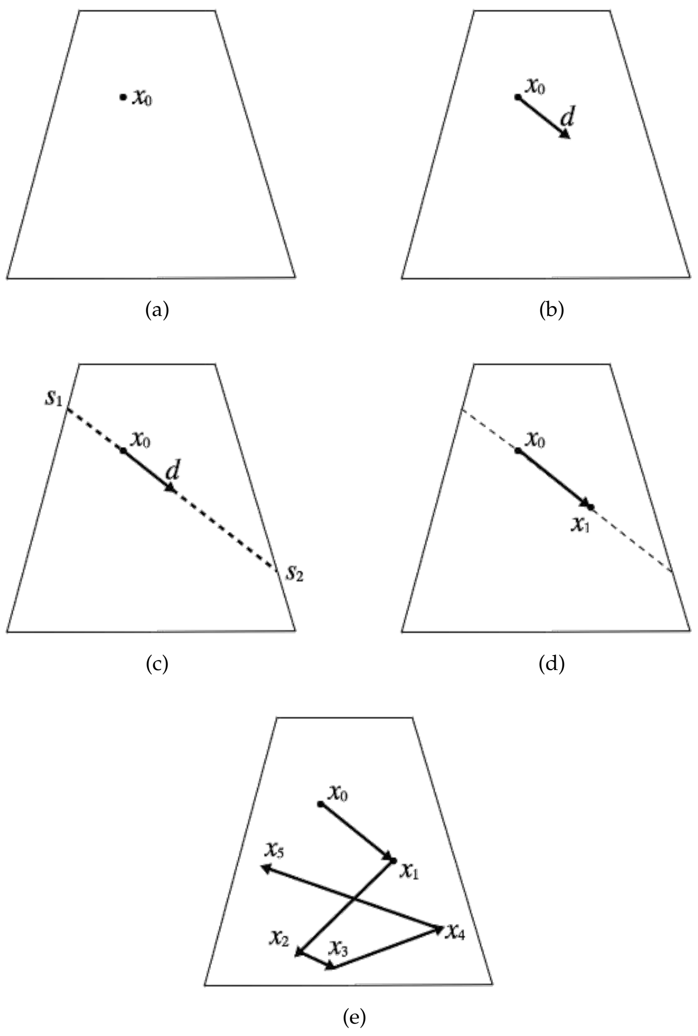

Figure 5: (a) The hit-and-run algorithm begins with an interior point x_0. (b) A random direction is selected. (c) The chord along that direction is calculated. (d) Then, pick a random point along that chord and move there as the new point. (e) The algorithm is repeated to sample many points.

$$\phi(x) = \sum_{i=1}^{m} -\log(b_i - a_i^T x)$$

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 tends to infinity as its argument goes to zero. Since $Ax \leq b$, for every row in $Ax \leq b$, $b_i > a_i^T x$. Therefore, as x approaches the boundary of K (as $a_i^T x$ approaches b_i), the value of ϕ approaches infinity. Hence, this is a barrier function. It is also exactly because of this that the starting point cannot on the boundary of K, where $b_i = a_i^T x$, but must be in the interior of K.

Hessian of Log Barrier H_x:

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

H_x is a $n \times n$ linear operator. D is a $m \times m$ diagonal matrix. The Hessian matrix (H_x) contains the second derivatives of the function $\phi(x)$ with respect to the vector x. The Hessian describes the shape of the local landscape at x.

Dikin ellipsoid $D^r_{x_0}$

Define the Dikin ellipsoid centered at x_0 with radius r as:

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

The Hessian H_{x_0} at x_0 is used as a local norm, which we call the "Hessian norm". The Dikin ellipsoid with radius 1 is the collection of all the points around x_0 whose difference with x_0 $(y - x_0)$ is within the unit threshold with respect to the Hessian norm.

The closer the point x_0 is to the boundary of polytope K, the larger the value of the Hessian norm, and thus, the smaller the range of allowed points given a unit threshold, which leads to a smaller Dikin ellipsoid. The further the point x_0 is from the boundary of polytope K, the smaller the Hessian norm and, therefore, the larger the Dikin ellipsoid.

For intuition, consider the single variable case. Recall that the log barrier function is of the form $-\log(z)$, where $z = a_i^T x - b_i$. The Hessian is the generalized second derivative, and the second derivative of $-\log(z)$ is $\frac{1}{z^2}$. The closer z is to zero (i.e., the closer x is to the boundary), the larger the norm.

Algorithm

1. Begin with a point $x_0 \in K$. This starting point must be in the polytope and not on its edge. If x_0 is on the boundary, then $a_i x_0 = b_i$ for some i, and consequently, the log-barrier and its Hessian would be infinity.
2. Construct D_{x_0}, the Dikin ellipsoid centered at x_0.
3. Pick a random point y from D_{x_0}.
4. Construct D_y, the Dikin ellipsoid centered at y.
5. If $x_0 \notin D_y$, then reject y. That is, if the current point x_0 is not in the Dikin ellipsoid of the potential point y, then we reject the point y. The purpose of this check is to avoid making a step that is too large.
6. If $x_0 \in D_y$, then accept y with probability $\min(1, \sqrt{\frac{det(H_y)}{det(H_{x_0})}})$. $\sqrt{\frac{det(H_y)}{det(H_{x_0})}}$ is equal to $\frac{\text{volume of } D_{x_0}}{\text{volume of } D_y}$. Recall that the volume of a Dikin ellipsoid reflects how close to the boundary its center is. The closer its center is to the boundary, the smaller its volume. This transition probability prevents the Dikin walk from concentrating in the "central region" of the polytope. Because we already know that $x_0 \in D_y$, the step is not too extreme. If the ratio of the volumes is greater than 1, that means the potential point y is closer to the boundary than x_0 is. In this case, we accept y with probability 1. If the ratio is smaller than 1, then x_0 is closer to the boundary than y is. In this case, we accept y depending on the ratio of their ellipsoids' volumes [2].
7. Repeat until obtained number of desired points.

[2]We do not need to worry that the accepted point y is not in K, because when we set $r = 1$, any Dikin ellipsoid centered at $x_0 \in K$ (x_0 not on the boundary) will be fully contained in K (see Kannan and Narayanan (2009) section 2.1.4)

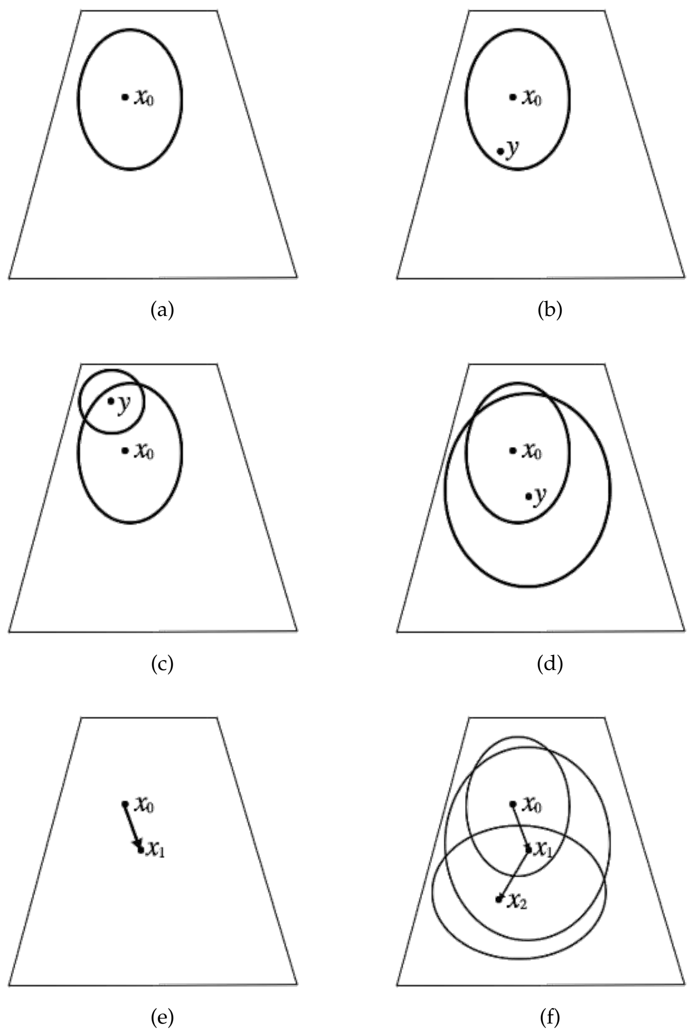

Figure 6: (a) The Dikin walk begins by constructing the Dikin ellipsoid at the starting point x_0. This point cannot be on the boundary of the polytope, otherwise the log-barrier and its Hessian would both be infinity. (b) An uniformly random point y is generated in the Dikin ellipsoid centered at x_0. (c) If point x_0 is not in the Dikin ellipsoid centered at y, then reject y. (d) If point x_0 is contained in the Dikin ellipsoid centered at y, then accept y with probability $\min(1, \sqrt{\frac{det(H_y)}{det(H_{x_0})}})$. (e) Once we've successfully accepted y, we set y as our new point, x_1. (f) The algorithm is repeated to sample many points.

See Figure 6. Dikin mixes much faster than hit-and-run does, especially in higher dimensions. This is because the mixing of Dikin is independent of the geometry of polytope K, whereas hit-and-run mixes slower in "skinny" regions of K (Kannan and Narayanan (2009)). Dikin's quick mixing comes at the cost of non-uniform sampling. Because the log-barrier function and Hessian prevent the Dikin walk from reaching points that are very close to the boundary of K, the resulting sample is concentrated in regions that are away from the boundary. See Figure 7 for an illustration.

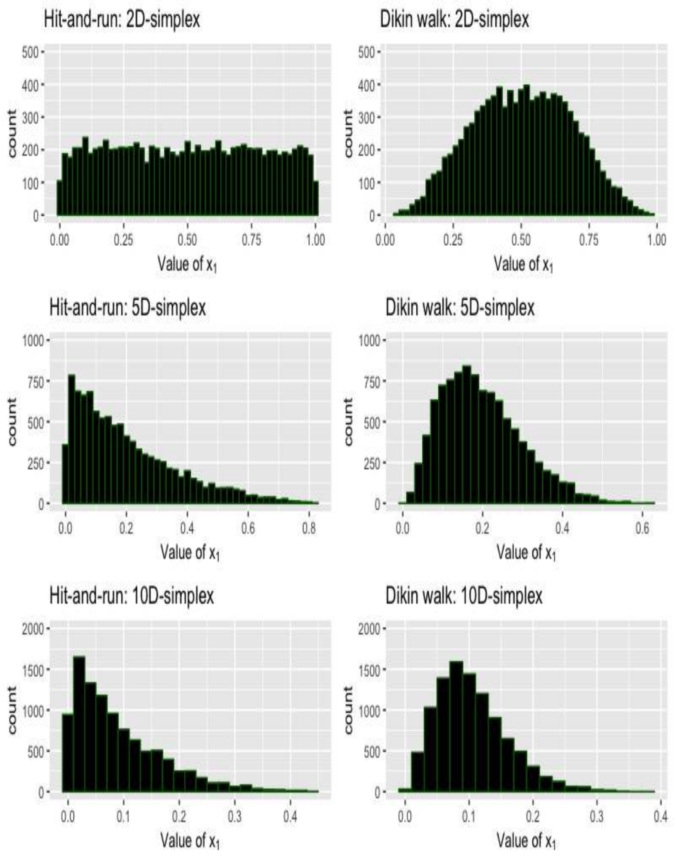

Figure 7: We use the two sampling algorithms, hit-and-run and Dikin walk, on the 2D, 5D, and 10D-simplex. We show the histograms for the first parameter, x_1, because the distribution for every parameter should be the same (there is nothing special about x_1). The 2D-simplex is the line segment described by $x_1 + x_2 = 1$ and x_i greater equal to 0. For the 2D-simplex, we see that hit-and-run samples uniformly across [0,1], while Dikin walk concentrates in regions away from the center. For higher dimensions (5D and 10D histograms), consider the 3D-simplex analogy. The 3D-simplex (Figure 1) is a triangle in three dimensional space. If we look at the distribution for x_1, that is equivalent to projecting this triangle onto the x_1 axis. As the samples are drawn uniformly from the 3D-simplex, there are more points near 0 than near 1. Therefore, we see this downward sloping distribution. In the 5D and 10D-simplex cases, we see that hit-and-run samples uniformly, while Dikin again concentrates in regions away from the edges.

```
library(walkr)
set.seed(314)
## initialize matrix

result <- matrix(1, ncol = 6, nrow = 10000)
## 2D simplex
A <- matrix(1, ncol = 2)
b <- 1
## hitandrun and dikin
result[,1] <- walkr(A = A, b = b, points = 10000, thin = 1, burn = 0, method = "hit-and-run")[1,]
result[,2] <- walkr(A = A, b = b, points = 10000, thin = 1, burn = 0, method = "dikin")[1,]
## 5D simplex
A <- matrix(1, ncol = 5)
b <- 1

## hitandrun and dikin
result[,3] <- walkr(A = A, b = b, points = 10000, thin = 1, burn = 0, method = "hit-and-run")[1,]
result[,4] <- walkr(A = A, b = b, points = 10000, thin = 1, burn = 0, method = "dikin")[1,]
## 10D simplex
A <- matrix(1, ncol = 10)
b <- 1
## hitandrun and dikin
```

```
result[,5] <- walkr(A = A, b = b, points = 10000, thin = 1, burn = 0, method = "hit-and-run")[1,]
result[,6] <- walkr(A = A, b = b, points = 10000, thin = 1, burn = 0, method = "dikin")[1,]
df <- as.data.frame(result)
colnames(df) <- c("har2","dikin2", "har5", "dikin5", "har10", "dikin10")
library(grid)
library(gridExtra)

m1 <- ggplot(df, aes(x=har2))
m2 <- ggplot(df, aes(x=dikin2))
m3 <- ggplot(df, aes(x=har5))
m4 <- ggplot(df, aes(x=dikin5))
m5 <- ggplot(df, aes(x=har10))
m6 <- ggplot(df, aes(x=dikin10))
m1 <- m1 + geom_histogram(binwidth = 0.02, colour = "darkgreen", fill = "black") +
      xlab(expression(paste("Value of ", x[1]))) +
      ggtitle("Hit-and-run: 2D-simplex") + scale_y_continuous(limits = c(0,500))
m2 <- m2 + geom_histogram(binwidth = 0.02, colour = "darkgreen", fill = "black") +
      xlab(expression(paste("Value of ", x[1]))) +
      ggtitle("Dikin walk: 2D-simplex") + scale_y_continuous(limits = c(0,500))
m3 <- m3 + geom_histogram(binwidth = 0.02, colour = "darkgreen", fill = "black") +
      xlab(expression(paste("Value of ", x[1]))) +
      ggtitle("Hit-and-run: 5D-simplex") + scale_y_continuous(limits = c(0,1000))
m4 <- m4 + geom_histogram(binwidth = 0.02, colour = "darkgreen", fill = "black") +
      xlab(expression(paste("Value of ", x[1]))) +
      ggtitle("Dikin walk: 5D-simplex") + scale_y_continuous(limits = c(0,1000))
m5 <- m5 + geom_histogram(binwidth = 0.02, colour = "darkgreen", fill = "black") +
      xlab(expression(paste("Value of ", x[1]))) +
      ggtitle("Hit-and-run: 10D-simplex") + scale_y_continuous(limits = c(0,2000))
m6 <- m6+ geom_histogram(binwidth = 0.02, colour = "darkgreen", fill = "black") +
      xlab(expression(paste("Value of ", x[1]))) +
      ggtitle("Dikin walk: 10D-simplex") + scale_y_continuous(limits = c(0,2000))

## plot it 3 by 2
grid.arrange(m1,m2,m3,m4,m5,m6, ncol = 2)
```

The **walkr** package uses **Rcpp** and **RcppEigen** (Eddelbuettel and François (2011), Bates and Eddelbuettel (2013)) to implement Dikin walk because of their support for faster matrix multiplication, inversion, and determinant calculation. This improvement in speed is especially important when sampling from A with many columns, corresponding to polytopes in higher dimensions.

To improve the mixing of a sample, there are two main techniques: thinning and burn-in. First, to "thin," we only save each thin$^{\text{th}}$ sample. Second, the "burn-in" is the portion of the total sample that is discarded. This is an effective technique when the starting point is in a corner or narrow region in the polytope. We must give time for the random walk to escape the corner and reach other parts of the sample space. For a discussion about techniques to improve MCMC sampling, see Stan Development Team (2015).

To quantitatively examine the mixing, we use the Gelman-Rubin diagnostic on multiple chains from diverse starting points (Gelman and Rubin (1992)). The general idea is that we measure the variance within each chain and the variance between the chains. If the variance between the chains is substantially larger than the variance within each chain, then the Gelman-Rubin diagnostic ($\hat{R}$) indicates that the mixing is poor and that, therefore, the chains should be longer. That is, poor mixing means that we need larger samples.

Using walkr

The **walkr** package has one main function `walkr` which samples points. `walkr` has the following parameters:

- `A` is the left hand side of the matrix equation $Ax = b$.
- `b` is the right hand side of the matrix equation $Ax = b$.
- `points` is the number of points returned. The total number of points sampled may be more than this because of thinning and burn-in.

- `method` is the method of sampling: either `"hit-and-run"` or `"dikin"`.
- `thin` is the thinning parameter. Every `thin`th point is returned. Default is `1`.
- `burn` is the burn-in parameter (as a percentage). The first `burn` points are deleted from the final sample. Default is `0.5`, for 50%.
- `chains` is the number of indepedent random walks we create, each from a different starting point. By default, walkr returns a matrix which consists of the individual chains combined together. Every column is a sampled point.
- `ret.format` is the return format of the sampled points. If `"matrix"` (the default), then a single matrix of points is returned. If `"list"`, then a list of chains is returned, with each chain as a matrix of points. Every column is a sampled point.

Consider the 3D simplex:

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

Sampling from higher dimensions follows the same syntax. Note that `walkr` automatically intersects $Ax = b$ with the N-simplex, so that the user does not have to include the simplex constraint in $Ax = b$. In this way, **walkr** is not a general tool for sampling from convex polytopes. Instead, it specializes in solving a special kind of convex polytope, one created by the intersection of $Ax = b$ and the N-simplex.

```
A <- matrix(sample(c(0,1,2,3,4,5), 40, replace = TRUE), ncol = 20)
b <- c(0.5, 0.3)
sampled_points <- walkr(A = A, b = b, points = 100, chains = 5,
                        method = "hit-and-run", ret.format = "list")
```

`walkr` warns the user if the chains have not mixed "well-enough" according to the Gelman-Rubin $\hat{R}$ values. We can ensure better mixing by increasing the amount of thinning, and hence the number of total points sampled.

```
sampled_points <- walkr(A = A, b = b, points = 1000, chains = 5, thin = 500,
                        method = "hit-and-run", ret.format = "list")
```

Alternatively, we could use Dikin, which mixes better.

```
sampled_points <- walkr(A = A, b = b, points = 1000, chains = 5, thin = 10,
                        method = "dikin", ret.format = "list")
```

Dikin walk only required `thin` to be `10`. Running hit-and-run with a `thin` of `10`, `100`, or even `250` would have produced a warning. This is evidence of Dikin mixing faster than hit-and-run. For higher dimensions of A, Dikin requires fewer points (or equivalently, a lower value for `thin`) to satisfy $\hat{R}$ than hit-and-run does.

Now, `sampled_points` contain 1000 sampled points. We can visualize the MCMC random walks by calling the `explore_walkr` function, which launches a `shiny` interface from **shinystan** (Gabry (2015)). Note that when calling `explore_walkr`, the `"ret.format"` argument from `walkr` must be `"list"`, because the individual chains must be separated out. Figure 8 shows a traceplot from the **shinystan** interface.

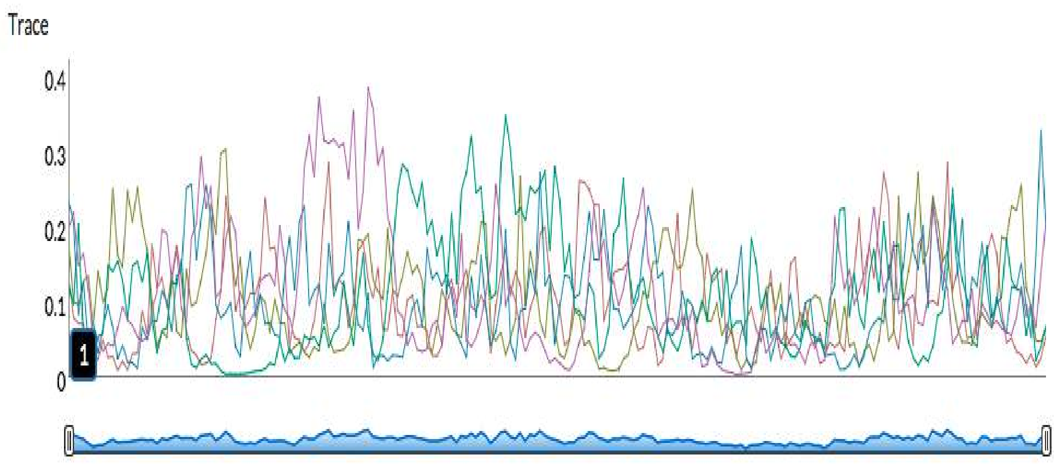

Figure 8: A screenshot from the **shinystan** interface called from `explore_walkr`. The traceplot is a plot of the value of different chains against the iteration number. It allows us to visualize the mixing of different chains. This particular sample comes from the `sampled_points` using the Dikin walk.

Conclusion

The **walkr** package samples from the intersection of two spaces. The first space is all possible vectors x that satisfy matrix equation $Ax = b$ (A is $M \times N$, with $M < N$), which defines M unbounded hyperplanes in $\mathbb{R}^N$. The second space is the N-simplex, defined as $x_1 + x_2 + x_3 + ... + x_N = 1$ and $x_i \geq 0$. The intersection of these two spaces is a non-negative convex polytope.

walkr samples from a non-negative convex polytope using two Monte-Carlo Markov Chain (MCMC) algorithms: hit-and-run and Dikin walk. Hit-and-run guarantees a uniform sample asymptotically, but mixes more slowly. Dikin walk samples non-uniformly, avoiding points near the boundary of the polytope.

MCMC methods begin at a starting point within the polytope and "wander" through the solution space. Every MCMC step depends only on the current location. To examine the quality of the samples, we create multiple chains, each from a different starting point. The "mixing" of different chains is one way of examining the quality of the samples. Dikin mixes much faster than hit-and-run does, especially in higher dimensions.

The major problem with our current implementaton is that run-time becomes unwieldy as the number of columns N in A increases. For lower dimensions of A (below 50) hit-and-run and Dikin can both generate a well-mixed sample within a few minutes. However, for dimensions near 500, it takes Dikin a few hours to generate a good sample, and hit-and-run much longer. In applications, we recommend using Dikin walk instead of hit-and-run for values of N greater than 50, assuming that a bias against the edges of the polytope is acceptable.

One possible extension to **walkr** involves parallelization. Especially for Dikin, the majority of the run-time is spent on matrix multiplication and inversion. Since matrix multiplication can be parallelized, the run-time issue in higher dimensions could be mitigated by extending the code to allow for the use of multiple cores.

Authors

Andy Yao
Mathematics and Physics
Williams College
3123 Paresky Center
Williamstown, MA 01267
United States
andy.yao17@gmail.com

David Kane
Harvard University
IQSS
1737 Cambridge Street
CGIS Knafel Building, Room 350
Cambridge, MA 02138
United States
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/. [p12]

K. V. den Meersche, K. Soetaert, and D. V. Oevelen. xsample(): An r function for sampling linear inverse problems. *Journal of Statistical Software, Code Snippets*, 30(1):1–15, 2009. ISSN 1548-7660. URL http://www.jstatsoft.org/v30/c01. [p7]

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/. [p12]

J. Gabry. *Interactive Visual and Numerical Diagnostics and Posterior Analysis for for Bayesian Models*, 2015. [p13]

A. Gelman and D. Rubin. Inference from iterative simulation using multiple sequences. *Statistical Science*, 7(4):457–511, 1992. [p12]

R. Kannan and L. Lovasz. Random Walks and a Volume Algorithm for Convex Bodies. *Random Structures and Algorithms*, 1997. [p1]

R. Kannan and H. Narayanan. Random Walks on Polytopes and an Affine Interior Point Method for Linear Programming. *Mathematics of Operations Research*, 2009. [p2, 7, 9, 10]

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

Stan Development Team. *Stan Modeling Language: User's Guide and Reference Manual*. Stan, 2015. [p12]

G. van Valkenhoef and T. Tervonen. *hitandrun: "Hit and Run" and "Shake and Bake" for Sampling Uniformly from Convex Shapes*. CRAN, 2015. [p7]

S. Vempala. Geometric random walks: A survey. *Combinatorial and Computational Geometry*, 52:573–612, 2005. [p2, 7]