Sampling from a Convex Polytope

Mike Flynn

June 28, 2013

Hit-and-Run algorithm

To sample from the hull of a convex polytope, a random-walk algorithm is typically used. The "hit-and-run" type algorithm uses the following steps:

1. Start from an intial solution x_0
2. Pick a random direction in the polytope, u
3. Isolate the segment s connecting x_0 to a wall of the polytope in that direction u
4. Sample uniformly from the segment s

In Detail:

The polytope is defined as the set: $\{x|Ax = Ax_0, x > 0\}$ where x is a vector of length n, A is a $m \times n$ matrix of constraints and x_0 an original solution. This set is geometrically an intersection of m n-planes defined by the rows of A and the half spaces $x_i > 0$. An easy case to picture is the 1-row A: $[1, 1, 1]$ with initial solution $(.3, .3, .4)$. This corresponds the the plane $x + y + z = 1$. The intersection of this plane with $x, y, z > 0$ leaves only the part in the first octant, a triangle.

Because we must at another solution to $Ax = Ax_0$ in the end, picking a "random" direction will not be a random direction in the space of x but rather a random direction in the k-plane that is $Ax = Ax_0$. This is done by finding an orthogonal basis of the null space of A: $Z_1, Z_2, \ldots, Z_k$, which will necessarily be orthogonal vectors in the k-plane. These basis vectors are weighted uniformly by sampling their weights from an exponential distribution and dividing by their sum. Therefore:

$$u = \sum_{j=0}^{k} Z_j r_j$$

where r_j is the normalized, exponentially distributed random weight.

We sample along the segment s by saying that $x_{i+1} = x_i + t * u$ where t is some scalar parameter, bounded by the limits of s. To sample uniformly on s,

we merely must sample uniformly on t, bounded by the limits of s. To find the limits of t we must recognize that for each index i:

$$x_i + t * u_i > 0$$

of which there are only 2 important cases: when $u_i > 0$ and when $u_i < 0$. This is because when we solve for the limits of t we simply divide by u_i to get 2 equations:

$$t_i > -\frac{x_i}{u_i} \text{ for } u_i > 0$$

and

$$t_i < -\frac{x_i}{u_i} \text{ for } u_i < 0$$

Therefore the largest t can be is large enough so that it is still less than the smallest such right hand side for the second equation, set by x_i, and likewise, must be greater than the largest right hand side for the first equation. Formally:

$$t_{\max} = \text{Min}(-\frac{x_i}{u_i}) \text{ for } u_i < 0$$

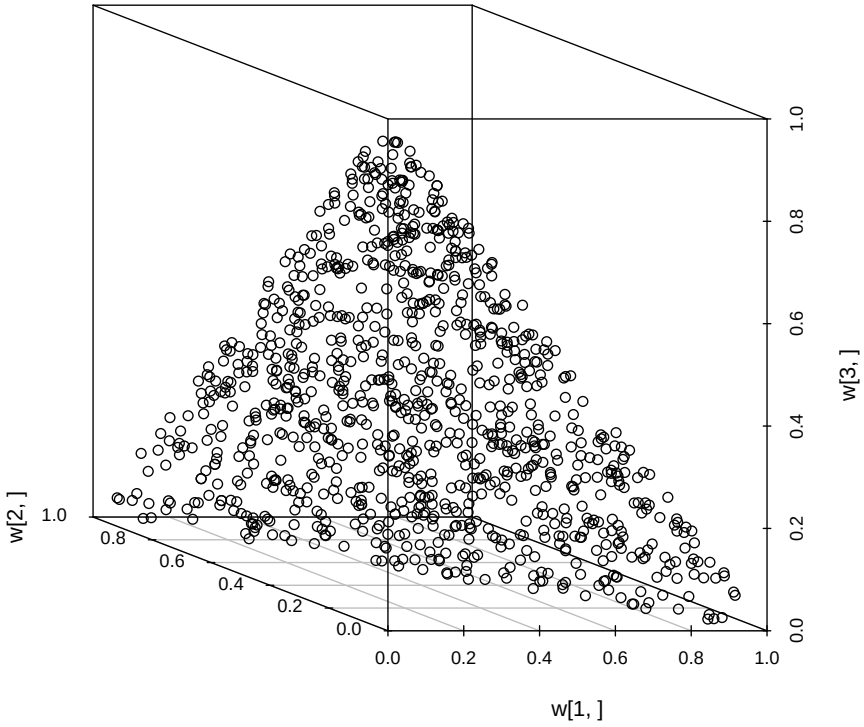
and

$$t_{\min} = \text{Max}(-\frac{x_i}{u_i}) \text{ for } u_i > 0$$

After these are figured out, t can be drawn from a uniform distribution between $t_{\min}$ and $t_{\max}$ to walk randomly on the simplex.

A demonstration:

```
require(MASS)
require(scatterplot3d)
#' Uniformly samples from {A*x=A*x0} U {x>0}
getWeights.hnr <- function(A, x0, n, discard) {

    y = x0
    ## resolve weird quirk in Null() function
    if (ncol(A) == 1) {
        Z = Null(A)
    } else {
        Z = Null(t(A))
    }
    X = matrix(0, nrow = length(x0), ncol = n + discard)
    for (i in 1:(n + discard)) {
        ## u is a random unit vector
        r = rexp(ncol(Z))
        r = r/sum(r)

        ## d is a unit vector in the appropriate k-plane pointing in a random
        ## direction
        u = Z %*% r
        c = y/u
        ## determine intersections of x + t*u with edges
        tmin = max(-c[u > 0])
        tmax = min(-c[u < 0])

        ## writeLines(paste('tmin: ', tmin, '\ntmax: ', tmax, '\n', sep = ''))
        ## chose a point on the line segment
        y = y + (tmin + (tmax - tmin) * runif(1)) * u
        X[, i] = y
    }
    return(X[, (discard + 1):ncol(X)])
}


## Sample from the triangle {x+y+z =1} U {x>0}
Amat = matrix(c(1, 1, 1), ncol = 3, nrow = 1)
x0 = c(0.3, 0.2, 0.5)
w = getWeights.hnr(Amat, x0, 1000, 5)

scatterplot3d(x = w[1, ], y = w[2, ], z = w[3, ], angle = 160)
```


w[2,]
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
w[1,]
0.0
0.2
0.4
0.6
0.8
1.0
w[3,]

1 Finding x_0

Minimizing distance from the origin

The thought process behind this idea was so: We need to get a point from the inside of a polytope, given the constraint equations. These equations give us a k-plane that will always go through the plane: $x_1 + x_2 + \cdots + x_n = 1$. Since the "triangle" defined by the intersection of this plane and the constraints $x_i > 0$ is already very close to zero, I assumed that any intersection of this plane with another constraint would automatically have it's closest point within the "triangle", however, this is only true if the intersection "goes through the triangle" in the first place, or so it seems.

The derivation is as follows:

The contraint equation $Ax = b$ can be though of geometrically as sequentially restricting x to plane by plane, with each row of A being a plane. It is a basic result of linear algebra that any plane can be represented by the equation $n * x = c$ where n is a vector that is normal to the plane, and c is some constant (i.e. $x + y + z = [1, 1, 1] * [x, y, z] = 1$). This is exactly what the rows of A are, normal vectors.

$$A = \begin{bmatrix} \mathbf{n_1} \\ \mathbf{n_2} \\ \vdots \\ \mathbf{n_n} \end{bmatrix}$$

This will help because the shortest distance between a point and a plane is the perpendicular distance between them, and therefore this distance vector must be in line with a normal vector. For this distance from the origin, this vector must be the position. It follows that:

$$x = \sum_{j=0}^{n} \mathbf{n_j} c_j = A^T c$$

We can now use the two equations for x to solve for it:

$$Ax = b$$
$$x = A^T c$$

Therefore:

$$c = (AA^T)^{-1} b$$

Finally:

$$x = A^T (AA^T)^{-1} b$$

I implemented this code as follows, which works fine in low dimensions but stop's being so good in higher dimensions:

```
find_x0 <- function(A, b) {
    return(t(A) %*% solve(A %*% t(A)) %*% b)
}
A = matrix(1, ncol = 3, nrow = 1)
A = rbind(A, rnorm(3))  # A random constraint, mimicking value
b = A %*% c(0.2, 0.3, 0.5)
x0 = find_x0(A, b)
b
```

```
##          [,1]
## [1,] 1.00000
## [2,] 0.03647
```

```
A %*% x0
```

```
##          [,1]
## [1,] 1.00000
## [2,] 0.03647
```

```
x0
```

```
##         [,1]
## [1,] 0.2418
## [2,] 0.4024
## [3,] 0.3558
```

```
## This stops working nicely for jan
data(jan)
A = matrix(1, ncol = nrow(jan), nrow = 1)
A = rbind(A, jan$value)
A = rbind(A, jan$growth)
b = A %*% jan$portfolio
b
```

```
##          [,1]
## [1,]  1.0000
## [2,]  1.9916
## [3,] -0.2689
```

```
x0 = find_x0(A, b)
length(which(x0 > 0))
```

```
## [1] 2113
```

```r
length(which(x0 < 0))
```

```
## [1] 887
```

```r
A %*% x0
```

```
##          [,1]
## [1,]  1.0000
## [2,]  1.9916
## [3,] -0.2689
```

As you can see, some of the weights are negative, which we do not want.

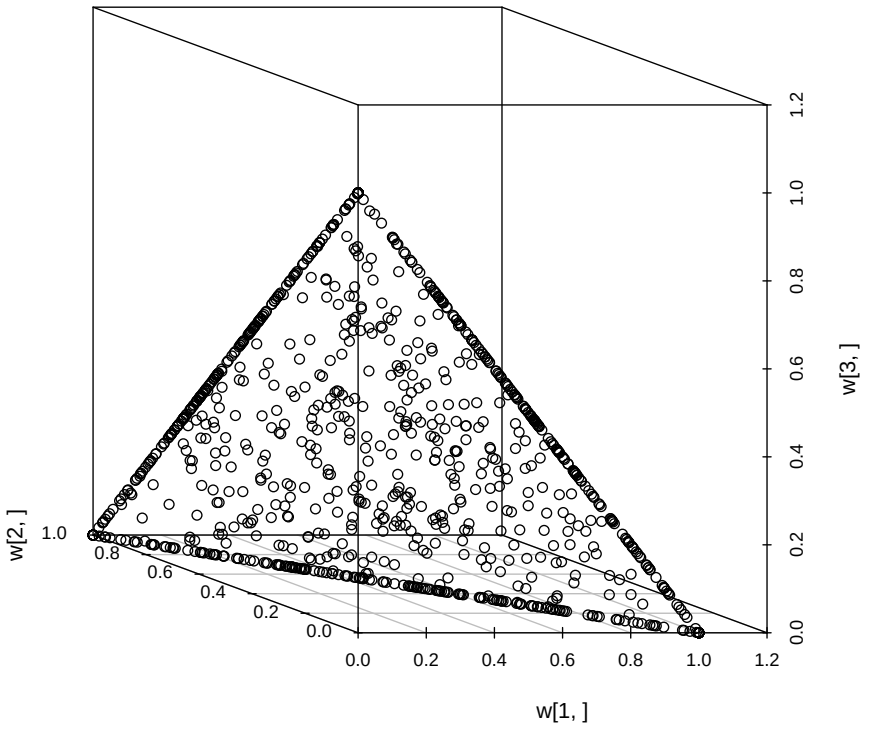
2 Changing up: going to quadratic programming

After suggestions from colleagues, we move back to an attempt at quadratic programming to solve our problem. The package `quadprog` will solve the following problem for b:

$$\text{Min}(-d^T b + 1/2 b^T D b) \text{ given that } A^T b >= b_0$$

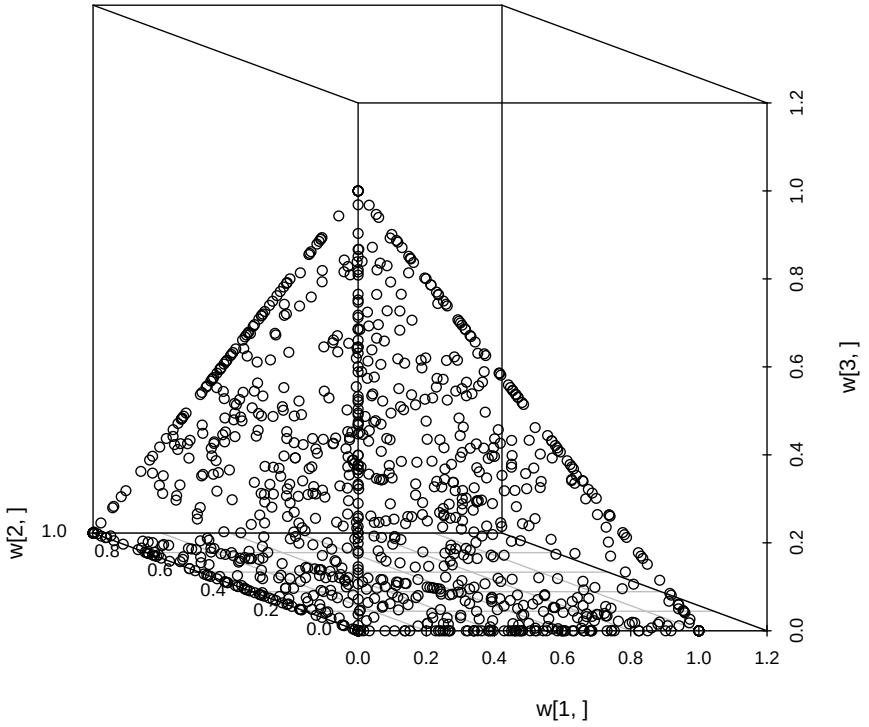
Since it similar to linear programming, it can be assumed that this method will bias towards the edges, which in fact it does, however, it does not exclusively go to corner solutions, like linear programming does, and so might be a better way to go about solving this. An example in 3D:

```
getWeights.quad <- function(A, b, n, verbose = FALSE) {

    w = matrix(0, ncol = n, nrow = ncol(A))
    if (verbose)
        cat("Portfolios created: 0")
    for (i in 1:n) {
        sol = solve.QP(Dmat = diag(2, ncol(A)), dvec = rnorm(ncol(A)), Amat = t(rbind(A,
            diag(1, ncol(A)))), bvec = c(b, rep(0, ncol(A))), meq = nrow(A))
        w[, i] = sol$solution
        if (verbose) {
            for (j in 1:nchar(paste(i - 1))) cat("\b")
            cat(paste(i))
        }
    }
    return(w)
}

A = matrix(1, ncol = 3, nrow = 1)
b = 1
w = getWeights.quad(A, b, 1000)
scatterplot3d(x = w[1, ], y = w[2, ], z = w[3, ], angle = 160)
```



As you can see we get many samples from the edges, but not all of them. Turning the dimensions up to 4 and looking at a 3d cross-section:

```
A = matrix(1, ncol = 4, nrow = 1)
b = 1
w = getWeights.quad(A, b, 1000)
scatterplot3d(x = w[1, ], y = w[2, ], z = w[3, ], angle = 160)
```



This is an interesting pyramid shape and despite its bias towards the edges, it seems like something that we would want. Unfortunately, trying it out on `jan` does not seem to be a productive use of time, as it takes quite a while.