Uniformly picking a finite discreet distribution

1. Introduction to the problem

This problem occurred when I was programming an environment to simulate the performance of the betting bot. I had 8 horses and I wanted to generate a distribution that tells me the probability of winning for each horse. I wanted to generate this "truly random" distribution so that no distribution has a higher chance of being chosen.

Problem:

" Let D be set of all discreet distribution with 8 outcomes (1,2,3,…,8). Now we need to find a way to uniformly choose a random element from D"

2. Generalizing the problem and writing it with equations

First, let's switch 8 with n. Now we can create a bijective function $f: D \to R^n$

$$f(x) = (p_1, p_2, p_3, \dots, p_n) \ \ where \ \ x \in D, P(x = i) = p_i \ \forall i \ \in \{1,2,3, \dots, n\}$$

now we need to find a way to choose a random point from S where:

$$S = \{(x_1, x_2, x_3, \dots, x_n) | \sum_{i=1}^{n} x_i = 1 \ and \ x_i \geq 0, \forall i \in \{1, \dots n\}\}$$

3. The first step

Watching the graph in 3 dimensions (for n=3) I saw that my S is a triangle with {(0,0,1), (0,1,0), (1,0,0)} as vertices. My first intuition was "If I can uniformly choose a point from a triangle's orthogonal projection on xy plane (that projection is a triangle with points {(0,0,0), (0,1,0), (1,0,0)}) I can just inverse projection back to the original triangle with function g(x,y)=(x,y,1-x-y). I need to prove this and generalize.

Proposition: If I can uniformly choose a random point from S_p where:

$$S_p = \left\{(x_1, x_2, x_3, \dots, x_{n-1}) \middle| \sum_{i=1}^{n-1} x_i \leq 1 \ and \ x_i \geq 0, \forall i \in \{1, \dots n-1\}\right\}$$

Then I can uniformly choose a random point from S.

Proof: Let X be a random variable that uniformly chooses a point from S_p. Now I define a function $g: S_p \to S \ where \ g(x_1, x_2, x_3, \dots, x_{n-1}) = (x_1, x_2, x_3, \dots, x_{n-1}, 1 - \sum_{i=1}^{n-1} x_i)$. Now I want to prove that random variable g(X) uniformly chooses a point in S.

Let's choose some measurable $C \subseteq S$ and let's call his orthogonal projection on xy plane C_p. With that $P(g(x) \in C) = \frac{A(C)}{A(S)} = \frac{\int_{C_p} \sqrt{n} dv}{\int_{S_p} \sqrt{n} dv} = \frac{\int_{C_p} 1 dv}{\int_{S_p} 1 dv} = \frac{V(C_p)}{V(S_p)} = P\left(x \in C_p\right)$.

- Where in the first equality I just expressed uniform probability as a ratio of the areas.
- In the second equality I used the area definition of the k-dimensional mapping in n-dimensional space for function g.

- Definition is : $A(S) = \int_Q \sqrt{\det(\nabla F(u)^{\perp} \nabla F(u))} du \ \ where \ function \ F \in C^1(Q, S) \ is \ bejection$.
- The determinant of a matrix with 2 for i=j and 1 for i≠j is calculated by subtracting the first row from every other, then from the first row subtract every other row. You are left with a triangular matrix with the determinant equal product of diagonal elements. In this case, the solution is n.
- In the fourth equality I used the volume definition of a n-dimensional set in n-dimensions

That proves that I can reduce my problem to finding an algorithm that uniformly chooses point from S_p.

4. Let's explore some trivial approaches that don't yield good results.

a. The first approach that comes to my mind is uniformly picking x_1 between [0,1], then x_2 between [0,1-x_1], x_3 between [0,1$-x_1 - x_2$]… after generating all the variables randomly shuffle them.

Most programming languages have a function for generating numbers from the uniform distribution (If not, then you can easily construct it with time).

This approach yields a distribution that favors critical values. I will construct a contradiction with uniform distribution in n=3.

$$P_{aproach\ a}(\max(x_1, x_2, x_3) \geq 0.9) > 0.1$$

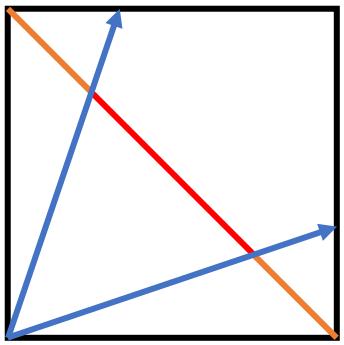
This holds because some variable is drawn from the uniform [0,1] distribution, and has a 10% chance of being ≥ 0.9

$$P_{uniform}(\max(x_1, x_2, x_3) \geq 0.9) = \frac{\frac{0.1 * 0.1}{2} + \frac{0.1 * 0.1}{2} + \frac{0.1 * 0.1}{2}}{\frac{1}{2}} = 0.03$$

And this is just adding areas of the triangle where $x_1, x_2, x_3 \geq 0.9$ and dividing it with an area of the triangle.

b. Intuitively next approach is generating n random variables and scaling it so their sum is equal to 1. Unfortunately, with this method for big n, it is almost impossible to get extreme values (values where one component is relatively bigger than the others).

It is easy to construct an intuitive contradiction for n=2 using geometry.



So basically, in our method, we are choosing a random point (or vector) in [0,1]X[0,1] square, and scaling it down till it hits the diagonal (which is actually our $S = \{x_1 + x_2 = 1 \; and \; x_1, x_2 \geq 0\}$ on x_1, x_2 plot).

I will prove that there is more chance to hit the middle 50% of the line.

If we choose a dot in S uniformly, the probability of getting a point in 50% middle data (red segment) is 0.5. So the probability of getting into the orange segment is also 0.5.

For our method, we will be guided by the sketch.

$$P_{method\ b}(\max(x_1, x_2) \geq 0.75) = \frac{\frac{1}{3} * 1}{2} + \frac{\frac{1}{3} * 1}{2} = \frac{1}{3} \neq \frac{1}{2} = P_{uniformly}(\max(x_1, x_2) \geq 0.75)$$

$-\ \max(x_1, x_2) \geq 0.75$ is equivalent to orange segment

- For the probability above I just added areas of the two right triangles with right angles in points (0,1), (1,0) and blue arrows as hypotenuse. Since only in them our method yields dots in orange segments.

c. Third, this time correct approach could be just trying to generate a point by generating all x_i uniformly from [0,1] till it lands in S. The only problem is the probability of landing a dot on S (which is hyperplane) by generating random dot from n-dimensional cube is 0.
Luckily, we can fix this by applying our proposition form third segment. Now we can uniformly generate points from an n-1-dimensional cube till it lands in S_p.
This approach works perfectly well and I was ready to accept it if I didn't find a better way.
I didn't settle on this because this approach has a big dimensionality scaling problem. The area of our S_p is $\frac{1}{(n-1)!}$ If we are solving the original problem in n dimensions. So probability of getting a point in S_p quickly becomes extremely small. I don't think I would be able to run my simulations with 8 horses.

5. Constructing an algorithm for our problem

The main idea is to recursively generate x_i one by one with the "fair" density function.

For fix $x_1 = c_1$ Let

$$S_{n-1,c_1} = \left\{(x_2, x_3, x_4, \dots, x_n) \middle| \sum_{i=2}^{n} x_i \leq 1 - c_1 \, , x_i \geq 0, \forall i \in \{2, \dots n\}\right\}$$

S_{n-1,c_1} represents set with all dots with c_1 as the first component.
Now I want to construct a density function for x_1 where the dots with relatively bigger $S_{n-1,c}$ will have bigger $f_{x_1}(c)$. Precisely, I want $\frac{A(S_a)}{A(S_b)} = \frac{f_{x_1}(a)}{f_{x_1}(b)}$ to be true for every $a, c \in S$. If I fixate $f_{x_1}(0) = T$ whole function is defined with $f_{x_1}(a) = \frac{A(S_{n-1,a})}{A(S_{n-1,0})} * T = A(S_{n-1,a}) * T'$. We can show that $A(S_{n,x}) = \frac{(1-x)^n}{n!}$, with that $f_{x_1}(a) = \frac{(1-a)^{n-1}}{n-1!} * T'$. Now we choose $T' = n!$ so that $\int_0^1 f_{x_1}(a)\, da = 1$.

With this, we got the density function which takes into account "thickness/size" of the set that has a for the first component.

For the rest of the variables, we need to repeat the process with:

$$S_{n-2,c_1+c_2} = \{(x_3, x_4, \dots, x_n) | \sum_{i=3}^{n} x_i \leq 1 - c_1 - c_2 \, , x_i \geq 0, \forall i \in \{3, \dots n\}\}$$

Where c_1 is constant we got after generating the first component using the function above:

$$f_{x_2}(a) = A(S_{n-2,a+c_1}) * T'$$

Calculating areas we get:

$$f_{x_2}(a) = \frac{(1 - c_1 - a)^{n-2}}{n-2!} * T'$$

Solving for $\int_0^{1-c_1} f_{x_2}(a) = 1$ we get $T' = n - 1! * \frac{1}{(1-c_1)^{n-1}}$.

The final generalization of this is: After generating $k \in \{1,2, n-2\}$ variables using the correct density function, let s be the sum of generated variables. Density function for generating k+1 variable is:

$$f_{x_{k+1}}(a) = (1 - s - a)^{n-(k+1)} * (n - k) * \frac{1}{(1-s)^{n-k}}$$

With domain = [0,1- s].

For the $k = n - 1$ the last variable is chosen by using the proposition from the third part ($x_n = 1 - s$).

6. Generating numbers from probability density function

In python library there is a built-in function for generating numbers from well-known distributions, but I found no premade way of generating numbers from arbitrary density functions.

Luckily for us, there is a well-known elegant solution to our problem, which is based on the following Lemma.

Lemma: "If F_X is cumulative distribution function of a continuous random variable X, then $Y = F_X(X)$ is a random variable uniformly distributed on the interval $[0,1]$."

Proof: First we need to assume that F_X is strictly increasing. This implies existence of inverse which is also increasing function. Now for every $y \in [0,1]$ holds:

$$F_Y(y) = P(Y \leq y) = P(F_x(X) \leq y) = P\left(X \leq F_X^{-1}(y)\right) = F_X\left(F_X^{-1}(y)\right) = y$$

Because F_Y is increasing, we can deduct that $F_Y(y) = 0$ for $y \leq 0$ and $F_Y(y) = 1$ for $y \geq 1$. Because uniform distribution has the same cumulative density function, we can say $Y \sim U(0,1)$.

By using this lemma, we can: $X \sim F_X^{-1}\left(F_X(X)\right) \sim F_X^{-1}(U(0,1))$.
As we can see, by using only uniform distribution (which is built-in function in almost all programing languages) we can generate our random variable X.

Our density functions are (for x_1 $s = 0$):

$$F_{x_k}(a) = 1 - \frac{(1 - s - a)^{n-k}}{(1 - s)^{n-k}}$$

Function inverse is:

$$F_{x_k}^{-1}(a) = (1 - s)\left(1 - (1 - a)^{\frac{1}{n-k}}\right)$$

With this we need to repeat three steps n-1 times (starting with $s = 0$ and k=1) to uniformly generate one dot from S.
1. Generate a number a from $U(0,1)$
2. $x_k = F_{x_k}^{-1}(a)$
3. Increase s by $x_k$and k by 1

After n-1 times of repeating this process, the last step is setting $x_n = 1 - s$.

7. Statistically testing my algorithm

After solving this problem there is always some fear that I made a mistake somewhere. This motivated me to statistically test this algorithm. The only way I could think of to statistically test my algorithm is using "chi-square goodness of fit" test.
The idea was to divide S_p into subsets and count how many generated dots ended up in every set. For the theoretical distribution, we use $\frac{V(subset)}{V(S_p)} * G$ where G is a number of generated dots. For this testing, I picked n=4 (so S_p will be in the third dimension).
My choice of dividing S_p is to use cubes that look like this:

$$C_{a,b,c} = [a, a + 0.1]\mathrm{X}[b, b + 0.1]\mathrm{X}[c, c + 0.1] \; for \; a, b, c \in \{0{,}0.1{,}0.2, \dots 0{,}7\}$$

with the constraint $a + b + c \leq 0.7$ because I want the whole cube to be in the S_p.
Besides these cubes we have special subset $\lambda = {S_p}/{\bigcup C_{a,b,c}}$.
It is obvious that $V(C_{a,b,c}) = 0.01$. While the volume of the $V(\lambda) = V(S_p) - N * 0.01$ where is N number of $C_{a,b,c}$ inside S_p.We can get N using "combinations with repetition" formula for every $a + b + c = 0.1 * i$. At the end we get:
$N = \sum_{i=2}^{9} \binom{i}{2} = 120$ and $V(\lambda) = \frac{1}{3!} - 120 * 0.01 \approx 0.04666$. Let's choose $G = 100000$ and applying the formula we get the theoretical value for $C_{a,b,c}$ is 60 dots and for the λ is 28000.

Putting everything in python and running a program a couple of times, I always get p-value around 0.5.

Now I feel more confident that my solution is correct.