Random Generation of Quantum States

Alec Roberson

Abstract

A brief overview of random generation protocol for both pure and mixed quantum states, with an emphasis on implementation.

1 Introduction

2 Random Pure States

Even though there is a method for randomly sampling n-qubit pure states, there are also many ways to mess up with implementing this method. In this section, we will look at one way to mess it up, and two ways to do it correctly.

Since any n-qubit pure state can be expressed in an orthonormal basis as a vector $\mathbf{v} \in \mathbb{C}^{2^n}$ where $|\mathbf{v}| = 1$, this is essentially our goal: generating random complex vectors with unit magnitude. With this in mind, here are some approaches I've tried.

2.1 Random Complex Numbers (Bad)

Since we can normalize any vector to have magnitude of unity, we might think that if we can evenly sample the complex numbers we can just do that a bunch of times and then normalize the resulting vector. One method to randomly sample the complex numbers is to use a gaussian distribution for r and a uniform distribution for $\theta \in [0, 2\pi)$ so that $z = re^{i\theta}$ is a random complex number. This seems like a reasonable approach, but implementing it reveals some glaring flaws. Figure 1 shows us 10,000 single-qubit states generated with this process. Note the clumping around the top and bottom of the Bloch sphere. This uneven distribution is even more evident looking at the histograms in fig. 2.

The reason for this uneven distribution can be tracked down if you go ahead and calculate $\langle\sigma_x\rangle$, $\langle\sigma_y\rangle$ and $\langle\sigma_z\rangle$ for a state $|\psi\rangle = r_1 e^{i\theta_1} |+\mathbf{z}\rangle + r_2 e^{i\theta_2} |-\mathbf{z}\rangle$. You will obtain $\langle\sigma_x\rangle = 2r_1 r_2 \cos\Delta\theta$, $\langle\sigma_y\rangle = 2r_1 r_2 \sin\Delta\theta$ and $\langle\sigma_z\rangle = r_1^2 - r_2^2$, which are clearly different distributions.

2.2 Random Probabilities (Good)

Instead of treating states as complex vectors in $\mathbb{C}^{2^n}$, this method begins with *probabilities*. Namely, since the probability of $|\psi\rangle$ being in the basis state $|e_j\rangle$ is $|z_j|^2$, we begin with a fair probability distribution of probabilities p_j that all sum to unity. Then the coefficients can be found as $z_j = \sqrt{p_j} e^{i\theta_j}$.

This raises the question: "what does it mean to find an even distribution of probabilities that sum to one?" This is a very important step that will come up again, and so the detailed answer to this question is given in appendix A.

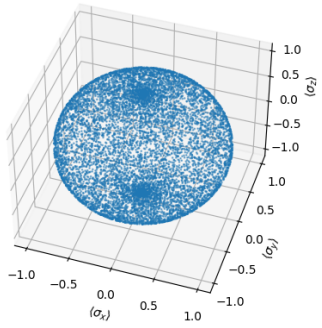

Fig. 1: These "random" states were generated with the method of generating random complex numbers with a gaussian distribution around the origin. Note the congregation of states around $|+\mathbf{z}\rangle$ and $|-\mathbf{z}\rangle$. This sampling is clearly not uniform across the bloch sphere.

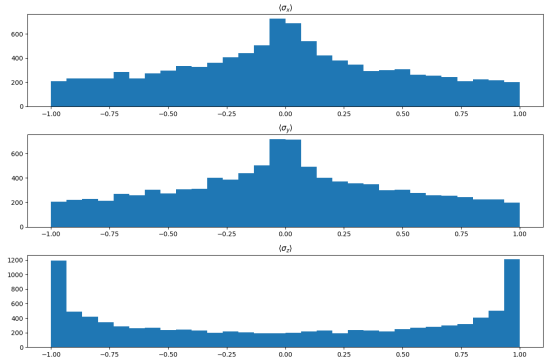

Fig. 2: Histograms of the x, y, and z positions for states generated with the second method (shown in fig. 1). With the same method of generating random complex numbers using a gaussian distribution about the origin, this plots histograms of the $\langle\sigma_z\rangle$, $\langle\sigma_y\rangle$, and $\langle\sigma_z\rangle$ for those states.

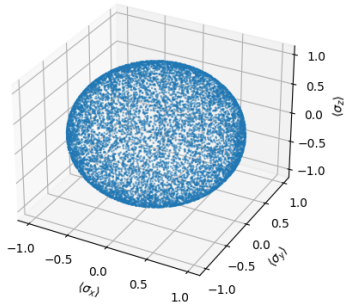

Fig. 3: 10,000 random states were generated using the second method discussed, taking the square root of the probability vector.

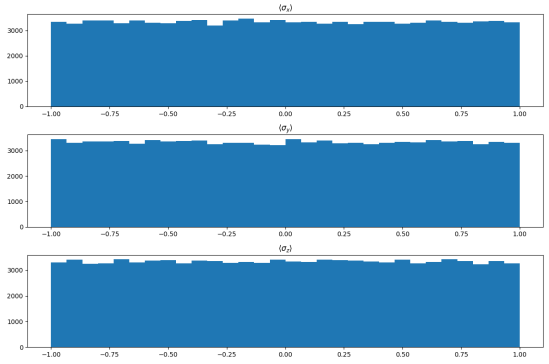

Fig. 4: Histograms of the x, y, and z positions for states generated with the second method (shown in fig. 3). We've increased the sampling to 100,000 to emphasizes the uniformity of these distributions.

2.3 Double Gaussian

Another method that appears to be equally as effective as the probability method is an entirely gaussian distribution for both $\operatorname{Re} z_i$ and $\operatorname{Im} z_i$ before normalizing the vector. This means that the `random_unit_vector` function would then be

```
def random_unit_vector(dim):
    x = np.random.randn(dim) + 1j * np.random.randn(dim)
    x = x / np.linalg.norm(x)
    return x
```

Generation of plots like those in fig. 1-4 shows this method to be effectively equivalent to the probabilities method, but we will be sticking with the probabilities method for generating random states for the remainder of the exploration. I will not include plots for this generation method, as they look identical to the aforementioned plots, but code to generate these plots is available in the file `2-3.py`.

3 Random Mixed States

3.1 Standard Method

3.2 Bloch-Parameterization Method

3.3 Pauli-Parameterization Method

3.4 Pauli-Parameterization Method

3.5 Trace Method

3.6 N-Trace Method

3.7 Over Parameterized Method

3.8

We might think that this task of generating

Expressed as vectors with complex components in an orthonormal basis, pure states will always have a magnitude of unity.

Pure states, expressed in an orthonormal basis, will all have a magnitude of unity. With this in mind, the only requirement for our sampling of pure state vectors is that it respect rotational symmetry in higher dimensions. There are many ways to do this, but perhaps

It is not as trivial as one might think to generate random pure quantum. Every n-dimensional quantum state is parameterized by n complex numbers $z_i \in \mathbb{C}$ that must satisfy $\sum_i |z_i|^2 = 1$. A naïve approach would be to attempt to sample the complex sphere in $\mathbb{C}^n$ using a gaussian distribution much like in $\mathbb{R}^n$, so the function would look like

```
def random_unit_vector(dim):
    x = np.random.randn(dim) + 1j * np.random.randn(dim)
    x = x / np.linalg.norm(x)
```

```
    return x
```

But there are some pretty fundamental problems with this sampling. Generating 10,000 pure states with this

A Random Probability Vectors

A probability vector in this context is some vector $\mathbf{v} \in \mathbb{R}^m$ such that $\mathbf{v} \cdot \mathbf{1} = 1$. This equation defines a plane of vectors that we wish to sample in some "fair" way. To get a feel for what's going on here, we'll look at the generation of random probability vector in $\mathbb{R}^3$.

A.1 Normalizing a Gaussian or Uniform Distribution (Bad)

One way to approach this (and the first answer given on stack overflow) is to just use your distribution of choice, gaussian or uniform, to generate a vector of numbers, and then normalize that vector according to it's sum. Code for this might look like

```
def random_prob_vector(dim):
    x = np.dist(dim) # dist is your chosen distribution
    x = x / np.sum(x)
    return x
```

However, this method causes issues. Let's look at the distribution of probability vectors in $\mathbb{R}^3$ that this method generates. Such distributions are shown in fig. 5 and fig. 6, and both are noticeably uneven, in that there are larger concentrations of probability vectors in certain areas of the plane than others.

The unevenness displayed in fig. 5 and fig. 6 is not necessarily a bad thing however, in fact, the correct probability distribution will also be "uneven" in this manner of speaking. What matters more is *how* they are uneven. I won't get too nitty-gritty here, but in higher dimensions when you have to pick random variables that sum to one, you are far more likely to get *small* numbers than large ones. However, in both the distributions shown in the aforementioned figures, the clumping of probability vectors occurs near the center of the triangle, where the probabilities are relatively large and evenly spaced. You can justify why this would be for yourself by considering the space spanned by the original distribution, and how it projects down to this plane when we take the norm.

A.2 Correct Method

The correct method for random probability generation in higher dimensions is described in [CITE THE PAPER]. First, we pick a random variable with a uniform distribution $p_1 \in [0, 1]$. Then for p_2 through p_{m-1}, you can use the following recursion relation to select the next probability with a uniform distribution

$$p_i \in \left[0, 1 - \sum_{j=1}^{i-1} p_j\right]$$

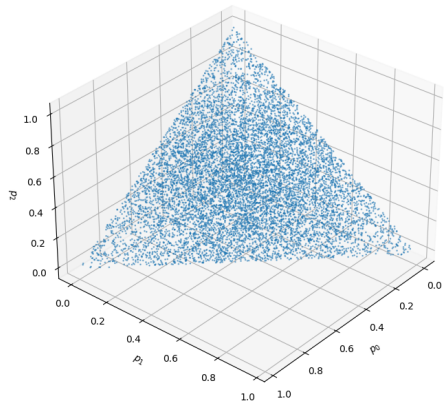

Fig. 5: 10,000 probability vectors generated by normalizing vectors from a **gaussian** distribution. Note the slight concentration of vectors towards the center of this triangle, as well as the more apparent lack of probability vectors near the vertices.

Finally, you select $p_m = 1 - \sum_{j=1}^{m-1} p_j$. This seems wildly unfair, for one thing you are virtually guaranteed for p_m to be very small, however by applying a random permutation to the probabilities, we can achieve a totally fair distribution that also captures the fact that you are simply more likely to get smaller probabilities from a fair distribution. A plot of probability vectors generated with this method is shown in fig. 7.

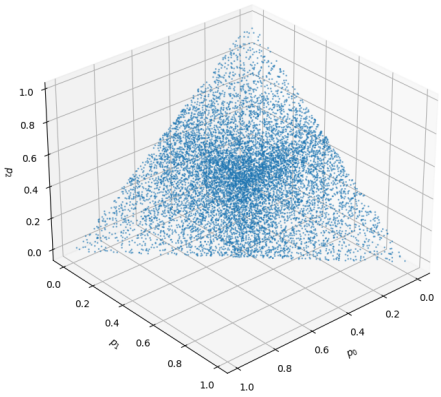

Fig. 6: 10,000 probability vectors generated by normalizing vectors from a **uniform** distribution. Note the overwhelming concentration of probability vectors in the center of this triangle, as well as on the edges.

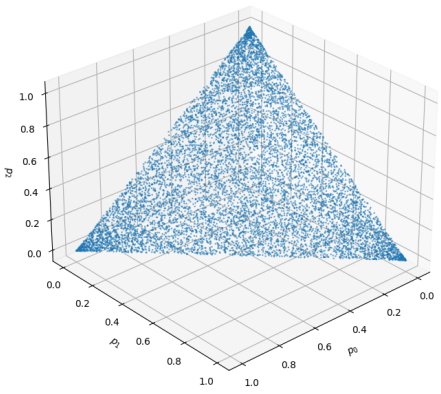

Fig. 7: 10,000 probability vectors generated using a "fair" method. Note the concentrations at the vertices of the triangle, indicating that smaller probabilities are more likely here, as we would expect in three dimensions or larger.