Quantum Computing Wavefunctions in Qiskit

Andreas Tsantilas
For Professor Javad Shabani

June, 2020

1 The Problem

On May 7, 1981, Richard Feynman delivered a speech [3] at Caltech proposing a new model of computation, one which utilized quantum phenomena to execute computations, rather than the classical model of a Turing Machine. In particular, Feynman pondered how one could run simulations of physical systems—not simply by numerical approximations of differential equations, but by making an *exact* simulation of physics. He considered the question of whether or not a quantum system could be simulated on a classical computer (i.e, one that gives the same probabilities as a quantum system does). Feynman's answer was an emphatic no; this is the so-called *hidden-variable* problem, which asserted that it was impossible to represents the results of quantum mechanics with a classical device. Moreover, such simulation on a classical machine is very inefficient. He demonstrated that a classical system would experience a slowdown that scaled exponentially with the input [8].

The goals for simulating a quantum system is twofold; one is to simulate time-dependent physical properties (e.g., scattering amplitudes), and the other is to compute spectral properties, such as eigenvalues [7]. In general, we can divide the process for simulating a quantum system into three stages. Foremost, we must efficiently[1] prepare the qubits into a state corresponding to some distribution. Next, we apply a time-dependent unitary evolution; lastly, we read out the desired information from this state [5].

The contents of this paper focus on the first step, which is preparing a wavefunction on a quantum computer. The discretized wavefunction is encoded in the (exponentially numerous) amplitudes of the basis states of the qubits, and the system is evolved by operators in parallel. Clearly, this corresponds to an exponential increase in time and space for a classical computer. However, for a quantum computer, this can be done with efficient time, memory, and circuit depth.

First, I begin by discussing a general method to prepare states $|\psi\rangle$ corresponding to positive, real-valued, efficiently-integrable probability distributions. This appears to have been first discovered by Zalka, followed by Grover and Rudolph, as well as Kaye and Mosca, who all seemed to have come across this algorithm independently. Next, I discuss the algorithm of A Kitaev and Webb, which improves upon the prior methods in constructing a Gaussian wavefunction in particular. Finally, I make a step-by-step guide to implementing this circuit in Qiskit, a Python-based quantum computing software development framework, made by IBM.

[1] We call a state consisting of N qubits "efficiently preparable" if it can be prepared within an error of ϵ using $poly(N, 1/\epsilon)$ one and two qubit gates.

2 Efficiently-Integrable Probability Distributions

2.1 Generating the Superposition

The work of Grover and Rudolph [4] in 2002 (and others) proposed an algorithm for preparing a discretization of an *efficiently-integrable* probability distribution. That is, given a probability distributon $\{p_i\}$, we would like to make a quantum superposition of the form

$$|\psi(\{p_i\})\rangle = \sum_{i=0}^{2^N-1} \sqrt{p_i}|i\rangle \tag{2.1}$$

where each $|i\rangle$ represents the corresponding configuration of qubits in the computational basis states in increasing lexicographic order. More concretely, the state $|0\rangle$ corresponds to $|0\dots0\rangle$, and $|2^N-1\rangle$ is $|1\dots1\rangle$, and the state $|i\rangle$ is $|i$ in binary$\rangle$ with the appropriate amount of 0's in front of it. With this aim in mind, we can sketch a rough outline of how to achieve it.

The argument proceeds inducively. Suppose we already have the m-qubit state

$$|\psi_m\rangle = \sum_{i=0}^{2^m-1} \sqrt{p_i^{(m)}}|i\rangle \tag{2.2}$$

where $|\psi_m\rangle$ denotes the state corresponding to the discretization done with m qubits, and p_i is the probability of the random variable to lie in the ith region of the distribution. For instance, p_0 is the probability for x to lie in the far leftmost region, p_1 is the probability of x to lie in the next region, and so on. In total, we have divided the distribution into 2^m evenly-spaced regions.

When we add another qubit, we achieve 2^{m+1} regions. Each region is cut in half, with the random variable x having a probability α_i to lie in the left half of the region i, and β_i to be the probability x is in the right half. Equivalently, we say that we achieve the state $|i\rangle\otimes|0\rangle$ and $|i\rangle\otimes|1\rangle$ with probability α_i and β_i, respectively.

Expressed as a transformation,

$$\sqrt{p_i^{(m)}}|i\rangle \mapsto \sqrt{p_i^{(m)}}\left(\sqrt{\alpha_i}|i\rangle\otimes|0\rangle + \sqrt{\beta_i}|i\rangle\otimes|1\rangle\right). \tag{2.3}$$

If we can achieve this evolution, we end up with the state

$$|\psi_{m+1}\rangle = \sum_{i=0}^{2^{m+1}-1} \sqrt{p_i^{(m+1)}}|i\rangle \tag{2.4}$$

where we can then repeat this process until $m=N$, i.e., until we have created the discretization over all 2^N states.

How can we actually go about implementing this? Grover and Rudolph propose that a function f is defined such that given that x lies in the region i, it is also in the left half of i. That is, $f(i)=\alpha_i$. Let x_L^i denote the left boundary of the region i, and x_R^i to be the right boundary. The probability of our random variable to lie in the left half of the region i is just

$$f(i) = \frac{\displaystyle\int_{x_L^i}^{x_L^i+(x_R^i-x_L^i)/2} p(x)dx}{\displaystyle\int_{x_L^i}^{x_R^i} p(x)dx} = \alpha_i. \tag{2.5}$$

If $p(x)$ is in the class of efficiently-integrable functions, then clearly finding this function is also efficient. Once we have computed $f(i)$, we can complete the transformation specified in equation (2.3) by rotating the $m+1$th qubit by a specific angle. In particular, the desired angle is

$$\theta_i \equiv \cos^{-1}\left(\sqrt{f(i)}\right) = \cos^{-1}\left(\sqrt{\alpha_i}\right). \tag{2.6}$$

Applying this rotation operator to our new qubit, we get the evolution

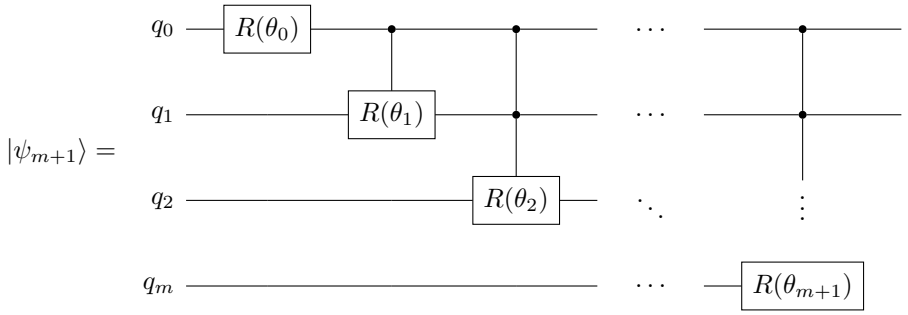
$$\sqrt{p_i^{(m)}}|i\rangle \otimes |0\rangle \mapsto \sqrt{p_i^{(m)}}|i\rangle \otimes (\cos(\theta_i)|0\rangle + \sin(\theta_i)|1\rangle) \tag{2.7}$$

which, if you recall the definition of θ_i in equation (2.6), is simply

$$\sqrt{p_i^{(m)}}\left(\sqrt{\alpha_i}|i\rangle \otimes |0\rangle + \sqrt{\beta_i}|i\rangle \otimes |1\rangle\right) \tag{2.8}$$

as desired.

Of course, the region i is really determined by some sequence of qubits in the states 0 and 1. Indeed, the transformation described in equations (2.3) and (2.7) is really a function of the states of the preceding qubits (albeit in decimal representation). What that means is that the rotation of the $m+1$st qubit controlled by all of the preceding m qubits. As a concrete example, should we consider the input $f(3)$, we know that the operation proceeds as if the first two qubits, q_0 and q_1, are in the $|1\rangle$ state, and the rest are in the $|0\rangle$ state. We can present the following circuit diagram:



It is worth noting that the gates are not only activated if all of its control qubits are $|1\rangle$. Rather, the gate always performs a rotation, but the rotation angle will vary based on the sequence of control qubits; viz., we take the state of the previous qubits in binary, convert it to decimal, and input it into our function f to obtain θ_i.

2.2 Storing the Rotation Angle

One possible method of computing the angle θ_i is by storing its binary representation on k supplementary qubits, or to $O(-\log(\epsilon))$ bits of accuracy. That is, we can initialize an ancilla register in the state $|0\ldots0\rangle$, and, because the function $p(x)$ is efficiently computable, we can construct a circuit which efficiently performs the computation

$$\sqrt{p_i^{(m)}}|i\rangle \otimes |0\ldots0\rangle \mapsto \sqrt{p_i^{(m)}}|i\rangle \otimes |\theta_i\rangle. \tag{2.9}$$

Once this has been completed, we are then at liberty to perform the transformation delineated in equation (2.7):

$$\sqrt{p_i^{(m)}}|i\rangle \otimes |0\ldots0\rangle \otimes |0\rangle \mapsto \sqrt{p_i^{(m)}}|i\rangle \otimes |\theta_i\rangle \otimes (\cos(\theta_i)|0\rangle + \sin(\theta_i)|1\rangle). \tag{2.10}$$

We can then uncompute the k-bit ancilla register containing θ_i, leaving us with a state of the form (2.7), as desired.

3 The Gaussian in One Dimension

In this section, we present the algorithm delineated in the 2008 paper by Kitaev and Webb [6]. Recall from a first course in probability that the Gaussian or Normal distribution is a function on x that takes two parameters: the mean and standard deviation (given by μ and σ, respectively). That is, we have $G : \mathbb{R}^3 \to \mathbb{R}$, where

$$G(x, \mu, \sigma) = e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{3.1}$$

Holding μ and σ fixed, and multiplying by a normalizing constant, we get that the Gaussian is a function $\psi_{\mu,\sigma} : \mathbb{R} \to \mathbb{R}$ given by

$$\psi_{\mu,\sigma}(x) = \frac{1}{\sqrt{\sigma\sqrt{\pi}}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{3.2}$$

Squaring it, we obtain

$$\psi^2_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{\pi}} e^{-\frac{(x-\mu)^2}{\sigma^2}}. \tag{3.3}$$

One can see that this function ψ^2 obeys the normalization property:

$$\int_{-\infty}^{\infty} \psi^2_{\mu,\sigma}(x)dx = 1. \tag{3.4}$$

Of course, we wish to consider the Gaussian distribution as a function of an integer value, since we are discretizing it. Therefore, we have the function $\widetilde{\psi}_{\mu,\sigma} : \mathbb{Z} \to \mathbb{R}$ which is a discrete approximation of the Gaussian given by

$$\widetilde{\psi}_{\mu,\sigma}(j) = \frac{1}{\sqrt{g(\sigma, \mu)}} e^{-\frac{(j-\mu)^2}{2\sigma^2}} \tag{3.5}$$

The function $g(\sigma, \mu)$ is a normalization factor, which is the sum over all the integers of $\widetilde{\psi}^2$. That is,

$$g(\sigma, \mu) = \sum_{k\in\mathbb{Z}} e^{-\frac{(k-\mu)^2}{\sigma^2}} = \widetilde{\psi}^2_{\mu,\sigma}(\mathbb{Z}) \tag{3.6}$$

so that the discrete Gaussian is

$$\sum_{j\in\mathbb{Z}} \widetilde{\psi}^2_{\mu,\sigma}(j) = 1 \tag{3.7}$$

However, the algorithm of Kitaev and Webb does *not* produce a Gaussian. In fact, a series of approximations are made to improve the efficiency of the algorithm. What we create is actually a *periodic* descrete Gaussian quantum state. This state has the advantage of having a more natural normalization, as well as the ability to be decomposed into a sum [2].

Now, if $\sigma \gg 1$, then

$$g(\sigma, \mu) \approx \int_{-\infty}^{\infty} e^{-\frac{(x-\mu)^2}{\sigma^2}} dx = \sigma\sqrt{\pi} \tag{3.8}$$

because the integer values in equation (3.6) are closer to small changes in a variable x, like an integral is. With this in mind, we are now ready to define a function that takes the parameters of mean,

standard deviation, and the number of qubits N. We denote by $\xi_{\mu,\sigma,N} : \mathbb{Z}/2^N\mathbb{Z} \to \mathbb{R}^+$ characterized by

$$\xi^2_{\mu,\sigma,N}(i) = \widetilde{\psi}^2_{\mu,\sigma}(i + 2^N\mathbb{Z}) = \sum_{t\in\mathbb{Z}} \widetilde{\psi}^2_{\mu,\sigma}(i + 2^N t). \tag{3.9}$$

Note that this function is periodic with period 2^N, which means it will produce an infinite number of curves that, under the right conditions, very nearly approximate a Gaussian wavefunction. Bear in mind that equation (3.9) is the *square* of the function ξ. This is important, because it will correspond to the probability of achieving a certain configuration of qubits. This allows us to define the desired quantum state $|\xi_{\mu,\sigma,N}\rangle$:

$$|\xi_{\mu,\sigma,N}\rangle \equiv \sum_{i=0}^{2^N-1} \xi_{\mu,\sigma,N}(i)|i\rangle. \tag{3.10}$$

For $\mu, 2^N - \mu \gg \sigma \gg 1$, this is a very close approximation to a Gaussian. This state is already normalized, since we took $\widetilde{\psi}^2$ to be normalized. This can be proven rigorously (see appendix?), but roughly the proof states that when N is large, μ is relatively far away from the borders, and σ is larger than one but not too much, then the periodic discrete and non-periodic discrete Gaussians are very close in total variation distance [2].

Of course, we have seen a lot of equations thrown about with not very much explanation, so it is important to conceptually understand what is occuring as well. We first considered a normalized Gaussian function in equation (3.2), and considered a discrete approximation (3.5) which was also normalized to unity. The function $f(\sigma, \mu)$ is the sum over all integers of the equation in (3.5) squared, such that we get the desired result in equation (3.7). We then consider a function ξ which satisfies equation (3.9); this is where the periodicity that we mentioned before applies. We are saying that for a fixed i,

$$\cdots = \xi_{\mu,\sigma,N}(i) = \xi_{\mu,\sigma,N}(i + 2^N) = \xi_{\mu,\sigma,N}(i + 2(2^N)) = \ldots \tag{3.11}$$

so if we changed our μ to $\mu + k \cdot 2^N$, for $k \in \mathbb{Z}$, we would get the same exact curve. Moreover, if we changed the upper bound of summation in (3.10) to $2 \times (2^N - 1)$, then for $\xi_{\mu,\sigma,N}$ we would obtain two discrete approximate Gaussian curves, one next to the other.

The key idea of the Kitaev-Webb algorithm is that there is a natural recursive definition for the state $|\xi_{\mu,\sigma,N}\rangle$, which leads almost immediately to an algorithm for preparing it. We will now see the importance of the periodic nature of the function ξ.

Lemma 1. The state of the periodic discrete Gaussian can be expressed recursively by

$$|\xi_{\mu,\sigma,N}\rangle = |\xi_{\frac{\mu}{2},\frac{\sigma}{2},N-1}\rangle \otimes \cos(\alpha)|0\rangle + |\xi_{\frac{\mu-1}{2},\frac{\sigma}{2},N-1}\rangle \otimes \sin(\alpha)|1\rangle \tag{3.12}$$

where the angle α is given by

$$\alpha = \cos^{-1}\left(\sqrt{\frac{g(\mu/2, \sigma/2)}{g(\mu, \sigma)}}\right) \tag{3.13}$$

Proof. We begin the proof by splitting the state in equation (3.10) into a sum over even integers and a sum over the odd integers:

$$|\xi_{\mu,\sigma,N}\rangle = \sum_{j=0}^{2^{N-1}-1} \xi_{\mu,\sigma,N}(2j)|j\rangle \otimes |0\rangle + \sum_{j=0}^{2^{N-1}-1} \xi_{\mu,\sigma,N}(2j+1)|j\rangle \otimes |1\rangle.$$

It now suffices to show that

$$\xi_{\mu,\sigma,N}(2j) = \cos(\alpha) \cdot \xi_{\frac{\mu}{2},\frac{\sigma}{2},N-1}(j)$$

and that

$$\xi_{\mu,\sigma,N}(2j+1) = \sin(\alpha) \cdot \xi_{\frac{\mu-1}{2},\frac{\sigma}{2},N-1}(j).$$

We will first prove this for the even sum:

$$\begin{aligned}
\xi^2_{\mu,\sigma,N}(2j) &= \widetilde{\psi}^2_{\mu,\sigma}(2j + 2^N\mathbb{Z}) \\
&= \frac{\psi^2_{\mu,\sigma}(2j + 2^N\mathbb{Z})}{g(\sigma,\mu)} \\
&= \frac{\psi^2_{\frac{\mu}{2},\frac{\sigma}{2}}(j + 2^{N-1}\mathbb{Z})}{g(\sigma,\mu)} \\
&= \frac{\psi^2_{\frac{\mu}{2},\frac{\sigma}{2}}(\mathbb{Z})}{g(\sigma,\mu)} \cdot \xi^2_{\frac{\mu}{2},\frac{\sigma}{2},N-1}(j) \\
&= \frac{g\left(\frac{\mu}{2},\frac{\sigma}{2}\right)}{g(\sigma,\mu)} \cdot \xi^2_{\frac{\mu}{2},\frac{\sigma}{2},N-1}(j).
\end{aligned}$$

Taking the square root,

$$\sqrt{\frac{g(\mu/2,\sigma/2)}{g(\mu,\sigma)}} \cdot \xi_{\frac{\mu}{2},\frac{\sigma}{2},N-1}(j)$$

which is, by definition,

$$\cos(\alpha) \cdot \xi_{\frac{\mu}{2},\frac{\sigma}{2},N-1}(j).$$

A similar argument can be made to satisfy the second part of the recurrence relation. By doing the same steps, we arrive at

$$\sqrt{\frac{g((\mu-1)/2,\sigma/2)}{g(\mu,\sigma)}} \cdot \xi_{\frac{\mu-1}{2},\frac{\sigma}{2},N-1}(j)$$

Notice that

$$\frac{g(\mu/2,\sigma/2)}{g(\mu,\sigma)} + \frac{g((\mu-1)/2,\sigma/2)}{g(\mu,\sigma)} = 1 = \cos^2(\alpha) + \frac{g((\mu-1)/2,\sigma/2)}{g(\mu,\sigma)}$$

from which we conclude

$$\sin(\alpha) = \sqrt{\frac{g((\mu-1)/2,\sigma/2)}{g(\mu,\sigma)}}$$

which completes the proof.

■

Following from this recursive description, we can specify a relatively simple algorithm [2] incorporating this lemma.

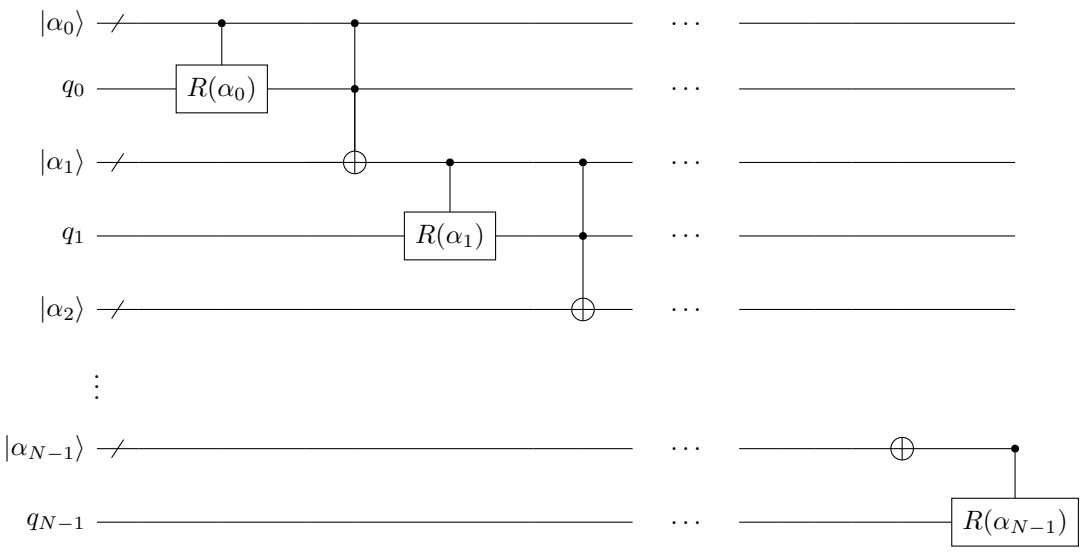
As the authors remark, it can be a bit confusing to understand quantum recursion. This schema might be better visualised as a quantum circuit. We will be using notation similar to the one that occurs in the Kitaev-Webb paper [6].

The wires of the circuit with a slash (/) on the far left indicate the k-qubit register that store the rotation angle at each step of the recursion. Note that each such wire represents the same qubits, albeit in different stages of the computation. Each gate is fed the angle α_i from the ancilla register above it. The lines with two nodes, terminating on the $\oplus$ symbol represent the fact that in our computation of α_i, we use $(\sigma_{i-1}, \mu_{i-1})$ and the state of q_{i-1} to control the rotation of the qubit i. However, since the previous qubit is in a superposition state, each rotation angle α_i *is always entangled with the previous qubits.*

Algorithm 1 Kitaev-Webb

Input: The parameters $\mu, \sigma \in \mathbb{R}^+$, $k \in \mathbb{N}$, and $N \in \mathbb{N}$.
Output: A discrete approximation of the periodic state $|\xi_{\mu,\sigma,N}\rangle$.

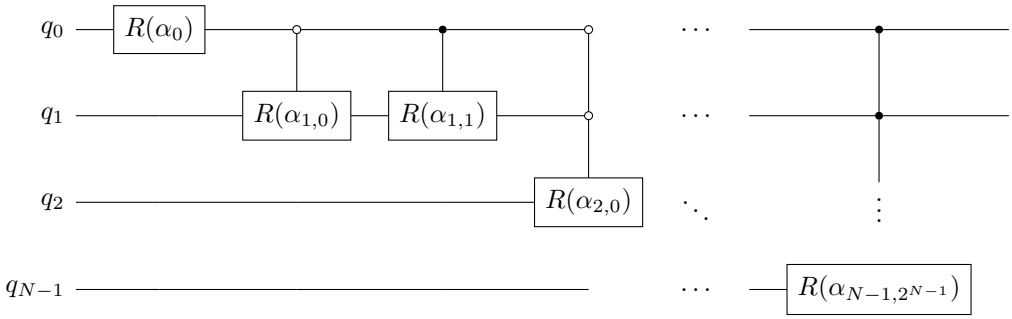
1. **Initial State:** $|0^k\rangle|\mu,\sigma,N\rangle|0^N\rangle$;
2. **Compute** α in the first register, yielding the state $|\alpha\rangle|\mu,\sigma,N\rangle|0^N\rangle$;
3. **Apply the rotation** α **on** q_0, yielding the superposition state
   $|\alpha\rangle|\mu,\sigma,N\rangle|0^{N-1}\rangle \otimes (\cos(\alpha)|0\rangle + \sin(\alpha)|1\rangle)$
4. **Uncompute** α from the register containing it, obtaining
   $|0^k\rangle|\mu,\sigma,N\rangle|0^{N-1}\rangle \otimes (\cos(\alpha)|0\rangle + \sin(\alpha)|1\rangle)$
5. **Execute a change of parameters controlled by** q_0, given by
   $\cos(\alpha)|0^k\rangle|\frac{\mu}{2},\frac{\sigma}{2},N-1\rangle|0^{N-1}\rangle \otimes |0\rangle + \sin(\alpha)|0^k\rangle|\frac{\mu-1}{2},\frac{\sigma}{2},N-1\rangle|0^{N-1}\rangle \otimes |1\rangle$
6. **Whenever** $N > 1$**, Apply steps 2-5 on all qubits except the last**, giving
   $\cos(\alpha)|0^k\rangle|\frac{\mu}{2},\frac{\sigma}{2},N-1\rangle|\xi_{\frac{\mu}{2},\frac{\sigma}{2},N-1}\rangle \otimes |0\rangle + \sin(\alpha)|0^k\rangle|\frac{\mu-1}{2},\frac{\sigma}{2},N-1\rangle|\xi_{\frac{\mu}{2},\frac{\sigma}{2},N-1}\rangle \otimes |1\rangle$
7. **Reverse the parameter change**, yielding
   $|0^k\rangle|\mu,\sigma,N\rangle \otimes \Big(\cos(\alpha)|\xi_{\frac{\mu}{2},\frac{\sigma}{2},N-1}\rangle|0\rangle + \sin(\alpha)|\xi_{\frac{\mu}{2},\frac{\sigma}{2},N-1}\rangle \otimes |1\rangle\Big)$
   $= |0^k\rangle|\mu,\sigma,N\rangle|\xi_{\mu,\sigma,N}\rangle$
   as desired.



4 Combined Approach in Qiskit

Now that we have covered the Kitaev-Webb algorithm, we can alter it slightly to be more apt to code. Namely, our implementation will be a hybrid of both Kitaev-Webb and Grover-Rudolph. Moreover, at each step of the recursion, we will compute the rotation angle classically. Creating separate quantum circuits to do the computations, and then store those on supplementary bits, is a computational hassle that distracts from the true purpose of this project. The main goal is not to gain ascendancy trhough a more efficeint algorithm; it is to prepare a physical state that obeys the laws of quantum mechanics and can be evolved accordingly. Therefore, we will take a slightly less efficient approach, that uses the functions delineated in Section 3, while having a process similar to the ones outlined in Section 2.

Because we can only specify one control sequence per gate in Qistkit, at each level i, we must specify 2^i gates, such that we account for all 2^i possible states of the qubits that come before. Written in circuit form, we have



where $\circ$ denotes activation by the $|0\rangle$ state, and $\alpha_{i,j}$ represents the rotation angle on level i obtained by the control sequence j in binary.

Before we do any coding, we have to make sure we have the relevant imports and version of Qiskit. One crucial feature that is only manifest in the later versions of Qiskit is that of the controlled gate, whereby we can specify a specific sequence of control qubits that triggers the operation.

```
from qiskit import *
from qiskit.circuit import *
from qiskit.extensions import RYGate
import math
import numpy as np
import matplotlib.pyplot as plt
import itertools

qiskit.__qiskit_version__
```

```
{'qiskit-terra': '0.13.0',
 'qiskit-aer': '0.4.0',
 'qiskit-ignis': '0.2.0',
 'qiskit-ibmq-provider': '0.6.0',
 'qiskit-aqua': '0.6.6',
 'qiskit': '0.18.0'}
```

Having done this, we now create the function $g(\sigma, \mu, \text{lim})$, which is the normalization function for $\widetilde{\psi}$. We include the extra parameter to specify the bounds for a (large) finite sum.

```
def g_(sigma_, mu_, lim):
    # normalization function, equal to psi tilde squared, summed over all the integers.
    # in lieu of an infinite sum, we can simply make "lim" sufficiently high.
    return np.sum(np.exp((-(np.arange(-lim, lim+1, 1) - mu_)**2)/float((2)*sigma_**2)))
```

The angle α is calculated as a function of g:

```
def angle_(sigma_, mu_, lim=1000):
    # Calculates the angle, based on the square root of probability.
    # cutoff the infinite sum in g_(...) at 1000 by default
    return np.arccos(np.sqrt(g_(sigma_/2., mu_/2., lim)/g_(sigma_, mu_, lim)))
```

At each level ℓ of the circuit (starting from zero), there are exactly ℓ levels above it. Each gate we place on this level is therefore controlled by ℓ qubits; so we have to have 2^ℓ controlled rotation gates at each level, whose control sequences are given by the range 0 through 2^ℓ in binary. For instance, at level 2, the first gate is triggered if and only if the qubits above are in the state $|00\rangle$, the second gate is triggered by $|01\rangle$, the third by $|10\rangle$, and the last by $|11\rangle$. Ideally, we would only have one gate whose function changes based on the ℓ-bit input, but this does not appear to be possible in the version of Qiskit at the time of this paper.

```
def ctrl_states(n):
    states = []
    for i in range(0,2**n):
        s = bin(i).replace('0b', '')
        states.append(''.join(['0' for j in range(0,n-len(s))])+s)
    return states

print(ctrl_states(3))
```

The output of `ctrl_states(3)` is

`['000', '001', '010', '011', '100', '101', '110', '111'].`

Recall that our calculation of the angle depends on all the other states before it. This is due to the parameter changes we apply. For σ, we divide by 2^ℓ. However, we subtract 1 or 0 from the mean (depending on the states of the previous qubits), we need to specify a function that will return the new mean (μ') we input into g and then into the angle $\alpha(\mu', \sigma)$.

```
def new_mu(qub, mu):
    # calculate \mu' for n-bit string qubit
    # i.e., we have c(b, mu) = (mu - b)/2, b \in {0,1}, and c('',mu) = mu
    # e.g., h('101101', mu) = c('1',c('0',c('1',c(...)))) etc.
    new_mean = mu
    for bit in qub[::-1]:#reversed, as we consider the qubits increasingly further back
        new_mean = (new_mean/2.) - ((1/2.)*int(bit))
    return new_mean
```

Now that we have all our functions that aid in the calculation of the rotation angle, we must now consider the rotation operator that we wish to use. The obvious choice of this operator would be one that maps $R(\theta) : |0\rangle \mapsto \cos(\theta)|0\rangle + \sin(\theta)|1\rangle$. This is accomplished by the R_y gate, which in Qiskit is

$$R_y(\theta) = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \tag{4.1}$$

Notice that this gate halves the angle, so somewhere in our program we must introduce a corrective factor of 2.

In order to create our circuit, we will first create a function that takes the parameters N, μ_0, σ_0. First, we create an empty circuit that has N qubits. We then initialize the angle α_0 by plugging it into our function `g_(·)`. To the 0th qubit, we add the gate $R_y(2\alpha_0)$. We then add 2^i gates to each level i, as we depicted before. Each gate on this level is controlled by a specific sequence of the previous qubits, so it is therefore entangled with them. We repeat this process for all qubits $q_1, \ldots, q_{N-1}$. Lastly, we return the circuit.

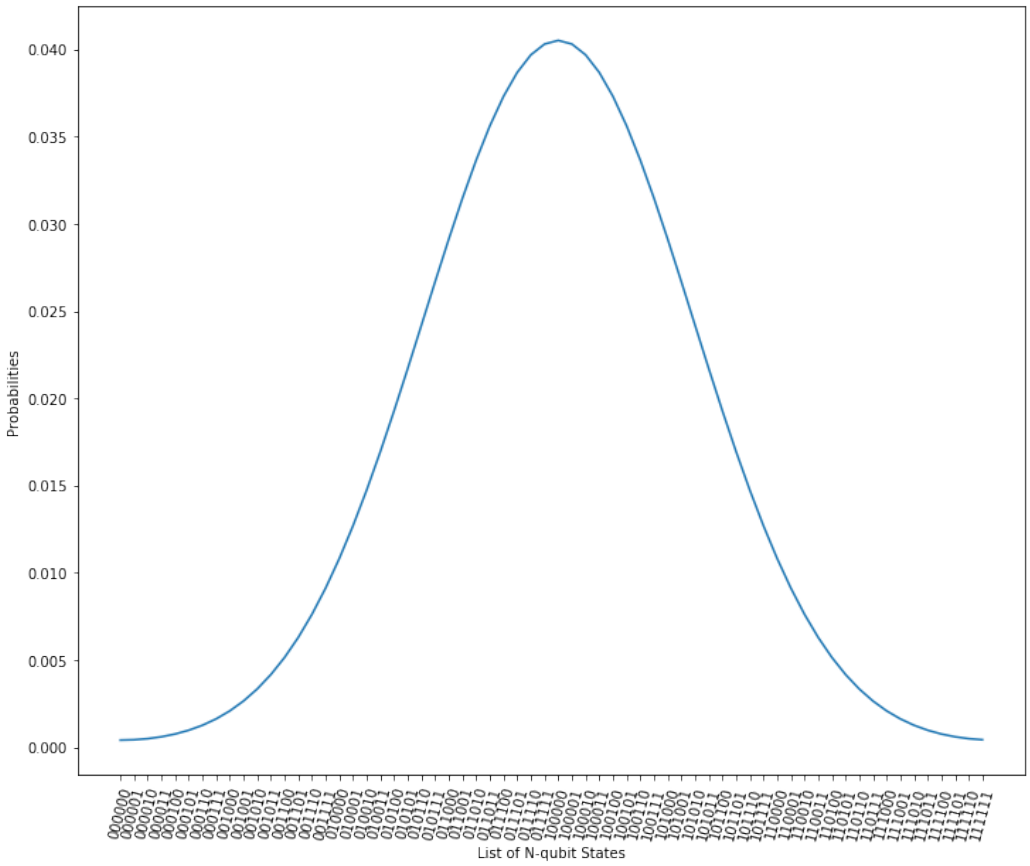
```
def create_circ(N, mu_, sig_,):
    qr = QuantumRegister(N, 'q')
    qc = QuantumCircuit(qr)# Generate quantum circuit with a register of qubit objects
    alpha_0 = angle_(sig_, mu_) # multiply by 2, because the Ry gate rotates by alpha/2
    qc.ry(2*alpha_0,0) # apply a rotation angle of 2*alpha_0
    for i in range(1,N): # Steps to be done at level q_i
        qstring = ctrl_states(i) # create list of 2^i strings of length i
        for k in qstring:
            alpha_ = angle_(sig_/(2**i), new_mu(k, mu_)) # Calculate angle via \mu'
            new_gate = RYGate(2*alpha_).control(num_ctrl_qubits = i,
                                                label = None,
                                                ctrl_state=k) #returns controlled version
            qc.append(new_gate, qr[:i+1]) # add Ry gate to level

    return qc #return the circuit we have generated
```

We now have everything we need. Recall, however, that we must judiciously choose a σ and μ which satisfy the limit imposed by Kitaev and Webb (viz. $\mu, 2^N - \mu \gg \sigma \gg 1$). It is worth tinkering around to see what different values of μ and σ [1].

```
N = 6
sigma = (2**N)/6
mu = 2**(N-1)

qc = create_circ(N,mu,sigma)


backend = Aer.get_backend('statevector_simulator')
state = execute(qc, backend).result().get_statevector()

print(state)

probs = [x*x for x in state]
print('sum : '+ str(sum(probs))) #This sum should approximately be 1

# plot the probabilities
wavefcn_xlist = list(ctrl_states(N))
wavefcn_ylist = list(probs)
plt.figure(figsize=(12, 10))
plt.plot(range(len(wavefcn_ylist)), wavefcn_ylist)
plt.xticks(range(len(wavefcn_xlist)), wavefcn_xlist, rotation=75)
plt.xlabel('List of N-qubit States')
plt.ylabel('Probabilities')
plt.show()
```

```
[0.02030149+0.j 0.02079833+0.j 0.02224751+0.j 0.02454891+0.j
0.02759022+0.j 0.03127888+0.j 0.03555048+0.j 0.04036378+0.j
0.04569234+0.j 0.05151675+0.j 0.05781893+0.j 0.06457802+0.j
0.07176757+0.j 0.07935376+0.j 0.08729417+0.j 0.09553724+0.j
0.10402198+0.j 0.11267829+0.j 0.12142744+0.j 0.13018299+0.j
0.138852  +0.j 0.1473365 +0.j 0.15553523+0.j 0.16334559+0.j
0.17066568+0.j 0.17739654+0.j 0.1834443 +0.j 0.18872241+0.j
0.19315361+0.j 0.19667192+0.j 0.19922416+0.j 0.20077138+0.j
0.20128978+0.j 0.20077138+0.j 0.19922416+0.j 0.19667192+0.j
0.19315361+0.j 0.18872241+0.j 0.1834443 +0.j 0.17739654+0.j
0.17066568+0.j 0.16334559+0.j 0.15553523+0.j 0.1473365 +0.j
0.138852  +0.j 0.13018299+0.j 0.12142744+0.j 0.11267829+0.j
0.10402198+0.j 0.09553724+0.j 0.08729417+0.j 0.07935376+0.j
0.07176757+0.j 0.06457802+0.j 0.05781893+0.j 0.05151675+0.j
0.04569234+0.j 0.04036378+0.j 0.03555048+0.j 0.03127888+0.j
```

```
0.02759022+0.j 0.02454891+0.j 0.02224751+0.j 0.02079833+0.j]

sum : (0.9999999999999691+0j)
```



We now have our Gaussian, as desired.

References

[1] M. Sohaib Alam. Quantum computing a gaussian wavefunction. Medium, 2018.

[2] Koen de Boer, Léo Ducas, and Serge Fehr. On the quantum complexity of the continuous hidden subgroup problem. Cryptology ePrint Archive, Report 2019/716, 2019.

[3] Richard P. Feynman. Simulating physics with computers. *Int J Theor Phys*, 21, 1982.

[4] Lov Grover and Terry Rudolph. Creating superpositions that correspond to efficiently integrable probability distributions, 2002.

[5] Ivan Kassal, James D. Whitfield, Alejandro Perdomo-Ortiz, Man-Hong Yung, and Alán Aspuru-Guzik. Simulating chemistry using quantum computers. *Annual Review of Physical Chemistry*, 62(1):185–207, May 2011.

[6] Alexei Kitaev and William A. Webb. Wavefunction preparation and resampling using a quantum computer, 2008.

[7] Rolando D. Somma. Quantum simulations of one dimensional quantum systems, 2015.

[8] Nicholas J. Ward, Ivan Kassal, and Alán Aspuru-Guzik. Preparation of many-body states for quantum simulation. *The Journal of Chemical Physics*, 130(19):194105, May 2009.