Quantum computing – A QPL approach to algorithms & fidelity

Revised 12/14/2021 11:01:00

Chase Sadri
University of Washington, Department of Applied Physics
3910 15th Ave NE, Seattle, WA USA 98195-1560
e-mail address: csadri@uw.edu

When people think, "The world of tomorrow," Quantum computing is very often the technology at the forefront of their mind. In recent history we have gained the ability to reliably and consistently harness quantum mechanical properties of particles. Initializing, altering, and measuring these, "qubits," has opened the door to what very well might be the final frontier of the Information Era. Heralding in what we might begin to refer to as, the *Quantum* Era. In the following we consider a variety of quantum programming languages and analyze their performance and efficiency in the form of runtime and fidelity analysis. Running simulations on physical quantum hardware provided by IonQ and IBM, we find satisfying and exciting results for readily available quantum hardware.

Keywords: quantum computing,
PACS: ?

I. INTRODUCTION

A fantastic byproduct of societal growth and human evolution is the advent of technology. As we slowly make our way towards technological zenith, our needs for incredibly advanced *supportive* technologies continues to grow. And, as the value of information continues to rise, we continually need more and more advanced ways to protect and secure our bits and bytes. Enter quantum cryptography and key distribution.

RSA Encryption

Any proper discussion of quantum cryptography and quantum key distribution should unequivocally begin with a reasonable explanation/exploration of modern encryption standards. This is where the Rivest-Shamir-Adleman (RSA) algorithm comes into play. There are four major components of the RSA algorithm that we should observe. These can be summed up as key generation, key distribution, encryption and decryption. See the following figure for a complete overview of the steps behind distributing keys and encrypting/decrypting. We will use familiar names for the purpose of a reasonable working example.

It is important to note: RSA encryption is designed around the computational difficulty of finding prime factors for an incredibly large number. The most efficient classical algorithms generally have a runtime between $poly(N)$ and $2^{poly(N)}$. In other words, sub exponential runtime of the order…

$$O(N^k) < factoring\ algo < O(e^N).$$

Considering a standard RSA key of 255 digits, it would take an incredible amount of time to algorithmically break RSA encryption. **[3]**

1. Chase wants to send The Professor a message. The Professor generates a **public key** by choosing two prime integers p and q such that $N = p * q$ along with some integer $c > 1$ such that c has no common divisor with $(p-1) * (q-1)$

2. The Professor generates a **private key** d by computing the inverse of c for mod multiplication in the congruence relation…

$$c * d \equiv 1 \mod (\mathrm{p}-1)(\mathrm{q}-1)$$

3. Chase encrypts his message m with the following formula…

$$b = (m^c) \mod N$$

4. The Professor decrypts Chase's message by computing…

$$m = (b^d) \mod N$$

Fig. 1: RSA Encryption Algorithm

Shor's Algorithm

In the pursuit of greater security, some of the strongest minds of our generation went to the task of finding means with which to wrest control from the icy grips of Rivest, Shamir, and Adleman. Enter: Peter Shor – a Caltech and MIT graduate, Putnam Fellow, Macarthur Fellow, Dirac Medalist, MIT Jr. Professor, and – most prominently – the father of

Shor's algorithm. We will begin to discuss the algorithm in more detail in our attempts at implementation and runtime analysis but it is of significant note that the algorithm of Shor reduces our integer factorization runtime to…

$$O\big((logN)^2(loglogN)(logloglogN)\big).$$

An efficiency increase of this magnitude has the power to exponentially reduce the time it would take an individual to break into an encrypted system. **[1]**

II. QUANTUM PROGRAMMING

As we further our understanding of quantum computers and their implications, the need for further development of related technologies increases almost daily. Of these, the most notable for the avid and interested learner is the quantum programming language (QPL). In general, QPLs are development kits, APIs, and libraries that allow for simulating quantum algorithms/circuits and/or running these on a real physical quantum computer.

*These experiments were performed in an Integrated Development Environment (IDE) provided by Spyder and NOT directly tied to IBM Labs Experiment software.

A. Qiskit (IBM)

The Quantum Information Software Kit (QISKit) for Quantum Computation is the lovechild of IBM Research and a large community of individuals who believe in the importance of making accessible resources for furthering the growth of quantum computation. These efforts have provided us with the ability to program quantum algorithms and circuits that can then be run on quantum hardware of up to 5000 qubits! Despite minor issues with fidelity and noise, it is hard not to overstress the incredulous power of this newfound gift of ready access to physical quantum hardware.

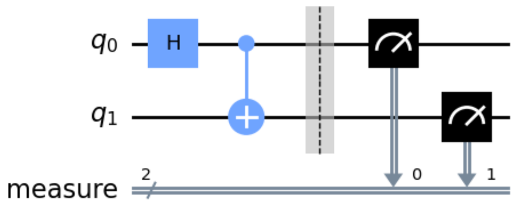


Fig.2 (a): A 2 qubit, 2 gate quantum circuit for Bell State generation

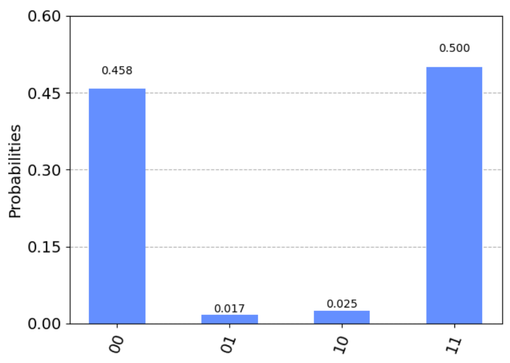


Fig. 2 (b): Associated histogram of output states to demonstrate noise/fidelity of IBM quantum hardware

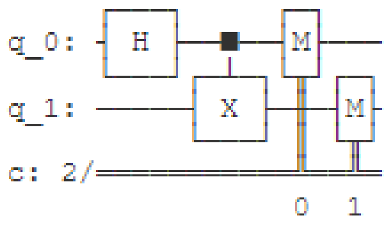
Running a premature error rate analysis with the following formula for on several passes over the same IBM quantum hardware ('ibmq-manila'); we observe the following

IBM: 2 qubits, 2 gates	
Mean accuracy	94.86%
Average queue + runtime	~1900s
Average runtime	350s

B. Q# (Microsoft)

Similar to Qiskit, Q# is a tool that can be utilized to run quantum algorithms (on quantum hardware or otherwise) from the comfort of your personal computer. The exciting nature of Q# is that it comes packaged as its own type of programming language based on the syntax and style of familiar languages like Python and C#. One major downside: integrating Q# outside of Azure (i.e. in development environments that we use very frequently) can be very difficult and time consuming. Another thing to consider: Algorithms run on ionq hardware are charged per run and can rack up quite a bill after quite a few operations. For this reason alone, we won't have the ability to pursue Q# integrated with ionq's hardware too far beyond fidelity analysis.

Simulating the same circuit as in Section A, we can follow the same process for building a quantum circuit and determining fidelity/runtime. In doing so, we generate the following:



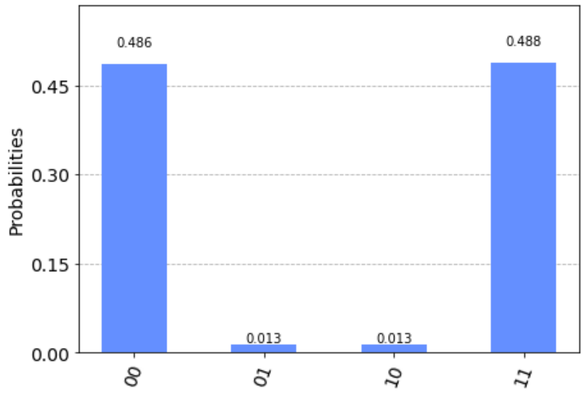


Fig. 3: The same quantum circuit run on ionq hardware. We observe noticeably higher fidelity and runtime with these devices when stacked up against competitors.

This yields the following accuracy and runtime for analysis/comparison

2 qubits, 2 gates	
Mean accuracy	97.04%
Average queue + runtime	323.3333 s

Although the jump in runtime and fidelity we observe going from IBM hardware to ionq is somewhat small in this particular example, it is very important to keep in mind that this is an introduction to quantum programming and we are implementing an algorithm that utilizes a meager 2 qubits and 2 gates. As we scale to larger and more complex algorithms that require a vast increase in qubit quantity along with more advanced gating operations, we should note the probability of runtime increasing and fidelity decreasing will likely increase linearly at the very least when we consider the difficulties of minimizing noise and environmental coupling for larger qubit systems.

Interestingly enough, it is not considerably difficult to find research wherein the investigators are interested in fidelity of quantum hardware from industry titans. Some fellow researchers from China's University of Mining and Technology have decided to ask similar questions about an algorithm known as HHL. Simply put, this algorithm uses quantum logic to solve systems of linear equations and, when implemented properly, provides a runtime increase of the following order:

$$O(N) \rightarrow O(logN)$$

Considering the prevalence of need for linear equation solvers in applied mathematics and computational science, we should be quite interested in this algorithm. In fact, we find it necessary to solve systems of linear equations in nearly every type of programmable simulation! Unfortunately, these authors **[4]** find quite a significant decrease in algorithmic fidelity as they increase the complexity of their systems of equations. Fidelity rapidly drops from over ninety-nine percent to just around sixty-five. Now, it appears as though IBM's quantum simulation software has vastly improved over the past year – running HHL on the same complex system of equations gives a fidelity of somewhere closer to seventy percent. My approach differed from the authors in how we approached designing the algorithm. They built their circuit by hand and I simply fell back on using IBM's HHL library.

C. Cirq (Google)

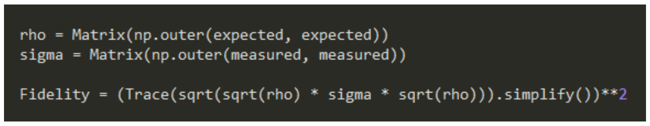
Cirq is the ideal tool for a Python developer that knows a little something about quantum computation to get familiarized with quantum programming. With friendly documentation, strong APIs, built-in simulators, and access to constructs that represent the constraints of a quantum processor, we highly recommend Cirq to individuals that are interested in quantum programming on a familiar language.

D. Runtime & Fidelity Analysis

Now that we've taken time to develop a strong familiarity and bond with a few of the most prevalent quantum programming tools available, it's time to peer in and take the time to perform some in-depth analysis on the fidelity and runtime of our various resources. Before we even breach the subject though, we shall be taking a moment to consider the following: What is fidelity?

$$Fidelity(\sigma, \rho) \coloneqq \left(Trace\left(\sqrt{\sqrt{\sigma}\rho\sqrt{\sigma}}\right)\right)^2$$

Eq. 1: Fidelity of σ and ρ: density matrices found by taking the outer product of a state vector with itself. For our purposes thus far, consider these to be the true Bell state versus what we measure on quantum hardware. A Pythonic approach follows below



```
rho = Matrix(np.outer(expected, expected))
sigma = Matrix(np.outer(measured, measured))

Fidelity = (Trace(sqrt(sqrt(rho) * sigma * sqrt(rho))).simplify())**2
```

I like to think of fidelity as a measurement of how noisy our quantum circuit is. For the very simple example of Bell State generation that we've been working with thus far (Hadamard gate on qubit 1 followed by a CNOT gate to generate an entangled state), we should expect to see an approximately equal superposition of the |00> and |11> states. However, we inevitably run into some difficulties finding an equal superposition; this is okay! Due to the probabilistic nature of particle states, we will almost always observe a superposition that's skewed in one direction or the other. But, as a young quantum student, I was quite troubled by the |01> and |10> states obtained in our preliminary analyses. That is, until I reviewed the earlier work of this paper's very author. **[5]**

Taking a step back to consider the computational complexity of these systems and the depth of algorithm we can run on a single IBM quantum computer brings forth not only an appreciation for a place at the forefront of a quantum developer's toolbox, but a shocking realization that these qubits are working incredibly hard. Sometimes, albeit not as often as hoped, we like to take a step back and appreciate the incredible applications of the very essential properties observed by atomic particles. Quantum mechanics is a cruel mistress and, in its nature, seems to disobey what we'd consider is natural. However, we must be grateful to the shoulders we stand on. For what are we if not quantum people.

Following this short detour of appreciation for quantum, let's talk about coupling! As with most things, the qubits behind quantum computation are closely tied to charged particles. Seeing that gates required for executing quantum algorithms operate very carefully on these particles, we shouldn't be surprised to learn that these particles are very sensitive to fluctuations and external perturbations. As a physicist, one might picture an atomic particle in some external field that we will refer to as unavoidable for now. Although, the magnitude of this field may not seem too significant; the atomic mass of our particle makes it such that there is a noticeable acceleration. A more directed mathematical approach to coupling is provided for reference.

Imagine a proton spinning around an electron (in the electron's frame of course), we would observe a magnetic field, **B**

$$B = \frac{\mu_0 I}{2r},$$

generated by what is effectively a current loop in the form of an orbiting proton. Simultaneously considering the motion of the electron with the proton as our frame of rest, we observe angular momentum, **L**

$$L = rmv,$$

that points in the same direction. Hence, we have…

$$\mathbf{B} = \frac{1}{4\pi\epsilon_0}\frac{e}{mc^2r^2}\mathbf{L}$$

Pairing this with the magnetic dipole moment of the electron produced by its own spin,

$$\mu = \frac{eg}{2m},$$

we find that the magnetic moment experiences a torque (τ) in the presence of an externally applied magnetic field:

$$\boldsymbol{\tau} = \mathbf{B} \times \boldsymbol{\mu}$$

As a result of this torque, we can calculate the resulting energy (in the case of a stationary proton) using the following work-torque relationship

$$W = \int_0^{2\pi} \boldsymbol{\tau}\, d\theta$$

Considering that total work done on a system is equivalent to its energy, we observe the energy of a spin-orbit coupled system to be represented by the following

$$E = 2\pi\mathbf{B} \times \boldsymbol{\mu}$$

Fig. 4: Spin-orbit coupling **[5], [9]**

Now that we have a reasonable understanding of one possible source of fidelity, let's consider another one. IBM quantum hardware is all based on superconducting qubit technology. By cooling a circuit to incredibly low temperatures (T_C), the electrical resistance of constituent components drops abruptly to zero. **[8]** Superconducting qubits can be defined as probability amplitudes associated with Cooper pair density on either side of a Josephson junction. Ideally, our temperature will remain below the superconducting transition and we will observe limited, if any, dissipation in the circuit. Nevertheless, repeated simulations run on these devices will undoubtedly increase their energy. For a superconducting qubit, this very well could result in reaching gap energy – causing a broken cooper pair and electron dissipation in the circuit. **[8]**

In collecting data for Figure x, we wound up running the same algorithm twice on a single computer. Curiously, the second pass unfailingly resulted in a lower fidelity. Possible explanations are the following:

1. Heat generated by quantum computation raises energy to the point where Cooper pairs are broken.
2. IBM developers don't like runtime experiments on their quantum computers.

Now, moving along we would very much like to analyze Ionq hardware. We will be doing so through the implementation of a Microsoft Azure Quantum Workspace. Ionq is heavily invested in trapped ion technology and believes this to be the best route towards scalable, high-fidelity readout. Without going into too much detail, I will now make an attempt to offer a succinct explanation of trapped ions as I understand them.

Prior to initializing trapped ions in the ground state, it is of the utmost importance that we provide a sufficiently high vacuum chamber to reduce fluctuations in the environment. Cooling ions with lasers and trapping them in a potential well has the effect of producing a discernible structure where each ion can be targeted individually for gating operations or measurement.

And lastly, our pursuits through this section on introduction to quantum programming yields the following data for in-depth runtime and fidelity analysis

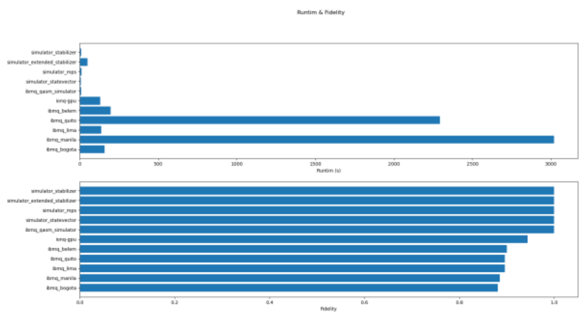


Fig. 5: Runtime & Fidelity Analysis.

Although Python plotting tends to be very disagreeable, we have obtained the following very useful information for simulating quantum circuits on real, physical hardware and quantum simulators (notice 100% fidelity and fast runtime). Since the charts are quite difficult to read, I'll summarize the findings here. Keep in mind, this research only applies to the 2 qubit, 2 gate Bell State construction circuit. We will require further investigation to see how runtime changes for higher complexity algorithms.

Most notably, we find the ionq-gpu backend in the running for shortest runtime with far and away the highest fidelity

III. GROVER'S ALGORITHM

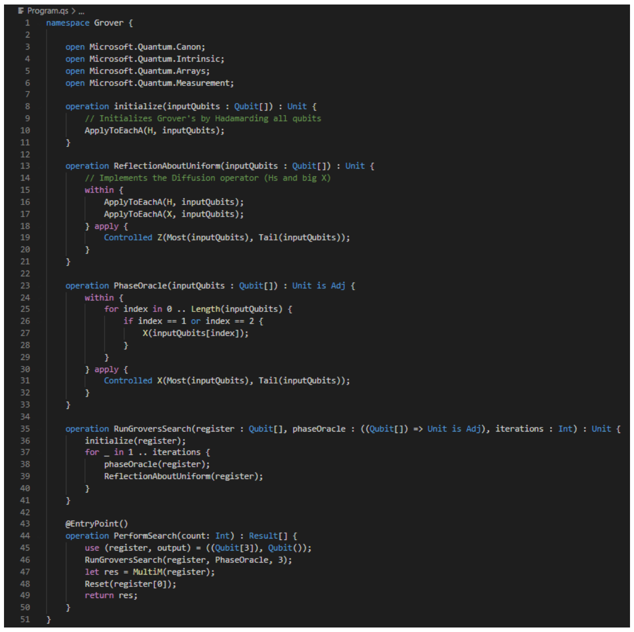
Having taken a spell to break away from the terribly complex and live in the world of simple quantum algorithms for a moment, it is high time that we jump back into the driver's seat and get going with increasingly complex quantum algorithms. This is where Grover's Algorithm comes in.

A. Unstructured Search

Sometimes, it's a fun activity to consider the average time we spend with certain activities. Picture this, you wake up and, all of a sudden, all of the books on your shelves have rearranged themselves! Now you don't know where to find anything anymore ☹. Frighteningly, if we were still living in the classical age of information, we would have no choice but to poke through an average of $N/2$ books with a maximum of touching all of them. For the quantum believer though, this is a thing of the past. Grover's developments have made it so that we don't have to leaf through too many more than $\sqrt{N}$ books!

B. First attempts at Q# programming

Saving ourselves from what could turn out to be going slightly overboard on runtime analysis, we will go no further than consider the following Q# snippet to observe the exciting applications of this new programming language (that, to our chagrin, has less than desired when it comes to documentation)



```
namespace Grover {

    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Arrays;
    open Microsoft.Quantum.Measurement;

    operation initialize(inputQubits : Qubit[]) : Unit {
        // Initializes Grover's by Hadamarding all qubits
        ApplyToEachA(H, inputQubits);
    }

    operation ReflectionAboutUniform(inputQubits : Qubit[]) : Unit {
        // Implements the Diffusion operator (Hs and big X)
        within {
            ApplyToEachA(H, inputQubits);
            ApplyToEachA(X, inputQubits);
        } apply {
            Controlled Z(Most(inputQubits), Tail(inputQubits));
        }
    }

    operation PhaseOracle(inputQubits : Qubit[]) : Unit is Adj {
        within {
            for index in 0 .. Length(inputQubits) {
                if index == 1 or index == 2 {
                    X(inputQubits[index]);
                }
            }
        } apply {
            Controlled X(Most(inputQubits), Tail(inputQubits));
        }
    }

    operation RunGroversSearch(register : Qubit[], phaseOracle : ((Qubit[]) => Unit is Adj), iterations : Int) : Unit {
        initialize(register);
        for _ in 1 .. iterations {
            phaseOracle(register);
            ReflectionAboutUniform(register);
        }
    }

    @EntryPoint()
    operation PerformSearch(count: Int) : Result[] {
        use (register, output) = ((Qubit[3]), Qubit());
        RunGroversSearch(register, PhaseOracle, 3);
        let res = MultiM(register);
        Reset(register[0]);
        return res;
    }
}
```

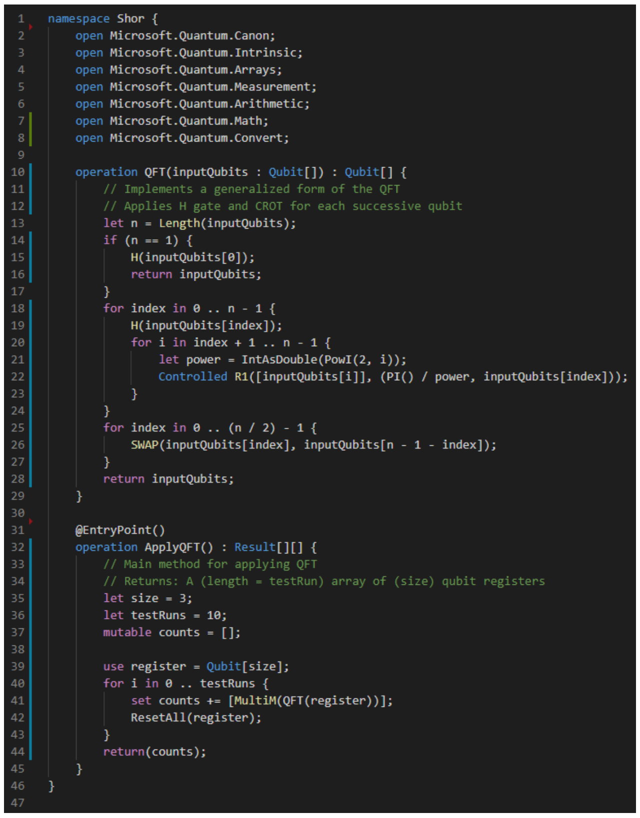
Fig. 6: Q# Implementation of Grover's algorithm

IV. SHOR'S ALGORITHM

Despite touching Shor earlier on in this endeavor, I feel it prudent to revisit and properly analyze the algorithm before we start to wrap everything up. As a challenge, I took it upon myself to attempt programming this algorithm in Q#. When compared to the abundance of available resources for programming Shor's algorithm in Qiskit, this choice would seem to make substantially less sense.

A. Quantum Fourier Transform

For this implementation, we heavily consulted the Le Bellac text **[8]** and Microsoft documentation **[10]** to produce a solution in a timely manner. Fortunately, our efforts produced fruitful in building a generalized QFT for a qubit register of any size.



```
namespace Shor {
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Arrays;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Arithmetic;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Convert;

    operation QFT(inputQubits : Qubit[]) : Qubit[] {
        // Implements a generalized form of the QFT
        // Applies H gate and CROT for each successive qubit
        let n = Length(inputQubits);
        if (n == 1) {
            H(inputQubits[0]);
            return inputQubits;
        }
        for index in 0 .. n - 1 {
            H(inputQubits[index]);
            for i in index + 1 .. n - 1 {
                let power = IntAsDouble(PowI(2, i));
                Controlled R1([inputQubits[i]], (PI() / power, inputQubits[index]));
            }
        }
        for index in 0 .. (n / 2) - 1 {
            SWAP(inputQubits[index], inputQubits[n - 1 - index]);
        }
        return inputQubits;
    }

    @EntryPoint()
    operation ApplyQFT() : Result[][] {
        // Main method for applying QFT
        // Returns: A (length = testRun) array of (size) qubit registers
        let size = 3;
        let testRuns = 10;
        mutable counts = [];

        use register = Qubit[size];
        for i in 0 .. testRuns {
            set counts += [MultiM(QFT(register))];
            ResetAll(register);
        }
        return(counts);
    }
}
```

Fig. 7: Q# Implementation of the Quantum Fourier Transform

In these pursuits, we find the implementation of the Quantum Fourier Transform to be very straightforward and efficient in Q#. Runtime on quantum hardware is incredibly high and we are able to produce an equal superposition state of a 3 bit quantum register in the range of $\mu - m$ seconds.

B. Period Finding

As it stands Shor's algorithm has proven to be the most efficient algorithm for prime factorization that we know of. We can accomplish this factorization of x using a register $|x\rangle$ and a **function** $f(x)$ in a register $|z\rangle$.Through a Unitary transformation U_f, we build an input register of the following form.

$$|\Psi_0\rangle = \frac{1}{\sqrt{K}} \sum_{k=0}^{K-1} |x_o + kr\rangle$$

Where $K \cong 2^n/r$ and r is the period of a function such that $f(x+pr) = f(x)$ when p is an integer. After rigorous computation and analysis, we find that determination of the period is enough to crack RSA encryption.

V. CONCLUSIONS

Our efforts in the preceding sections have taken us from analyzing the performance of a wide array of quantum programming languages up to the implementation of highly relevant algorithms that have the potential to change the way we operate on an informational level. Since quantum computing was theorized and introduced 20 years and some change ago, the integration of quantum computing solutions is finally beginning to make its way into the mainstream.

Since their inception, we have increased the quantity of qubits in our registers 10-fold and its hard to imagine that this exponential growth will slow down any time in the near future.

1. ACKNOWLEDGEMENTS

I'd like to thank this department that I am a part of for pushing me to the limit of my ability and my teachers for fostering my appreciate for learning. In particular, The Professor, Dr. Blinov has been incredibly supportive in my academic career. The beautiful struggle we endure in lecture – beginning to understand new material and on homework – applying what we've learned; is unprecedented. My classmates, peers, and mentors have been and continue to be fundamental in my growth. As I grow, I will work harder to meet up to the incredible examples set before me.

[**1**] Gidney, C. and Ekera, M. (2021) How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum 5*, 433.
[2] Tianqi Zhou, Jian Shen, Xiong Li, Chen Wang, and Jun Shen. (2017) Quantum Cryptography for the Future Internet and the Security Analysis. *Hindawi* 2018.
[3] Ramakrishnan, G. (2019) Design and Verification of an RSA Encryption Core. Thesis. Rochester Institute of Technology.
[4] Wen Ji, Xiangdong Meng. (2021) Demonstration of quantum linear equation solver on the IBM qiskit platform. *19th International Symposium on Distributed Computing and Applications for Business Engineering and Science*.
[5] Sadri, C. (2021) Qubits: Function, Fabrication and Quantum Dots. *PHYS 324 with The Professor*.
[6] Qiskit Development Team. (2021) Qiskit Documentation. *Qiskit*.
[7] Cirq Developers. (2021). Cirq (v0.12.0). *Google*. https://doi.org/10.5281/zenodo.5182845
[8] Le Bellac, Michel. *A Short Introduction to Quantum Information and Quantum Computation*. Cambridge University Press, 2006.
[9] Griffiths, David. *Introduction to Quantum Mechanics, Third Edition*. New York, Cambridge University Press, 2018.
[10] Microsoft. *Tutorial: Implement Grover's search algorithm in Q#*. Microsoft, 2021.
[11] cgranade. *Searching with Grover's Algorithm*. Github 2021.