SAND2015-9617R

SAND20XX-XXXXR
LDRD PROJECT NUMBER: 15-2618
LDRD PROJECT TITLE: Versatile Formal Methods Applied to Quantum Information

PROJECT TEAM MEMBERS: Wayne Witzel, Mohan Sarovar, Kenneth Rudinger

ABTRACT:

Using a novel formal methods approach, we have generated computer-verified proofs of major theorems pertinent to the quantum phase estimation algorithm. This was accomplished using our Prove-It software package in Python.

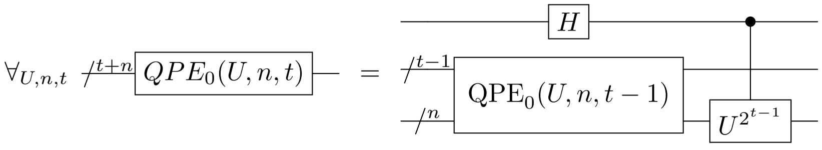
While many formal methods tools are available, their practical utility is limited. Translating a problem of interest into these systems and working through the steps of a proof is an art form that requires much expertise. One must surrender to the preferences and restrictions of the tool regarding how mathematical notions are expressed and what deductions are allowed. Automation is a major driver that forces restrictions. Our focus, on the other hand, is to produce a tool that allows users the ability to confirm proofs that are essentially known already. This goal is valuable in itself.

We demonstrate the viability of our approach that allows the user great flexibility in expressing statements and composing derivations. There were no major obstacles in following a textbook proof of the quantum phase estimation algorithm. There were tedious details of algebraic manipulations that we needed to implement (and a few that we did not have time to enter into our system) and some basic components that we needed to rethink, but there were no serious roadblocks. In the process, we made a number of convenient additions to our Prove-It package that will make certain algebraic manipulations easier to perform in the future. In fact, our intent is for our system to build upon itself in this manner.

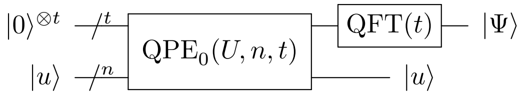
INTRODUCTION:

The quantum phase estimation algorithm solves the following problem. Given a unitary operator U and quantum state $|u\rangle$ such that $U|u\rangle = \mathrm{e}^{2\pi i\varphi}|u\rangle$, estimate φ. The algorithm uses a prepared register of t quantum bits and an input register containing $|u\rangle$. It puts the t-qubit register into a state that, when measured, produces a binary expansion that approximates φ. It requires the ability to apply qubit-controlled applications of $U^{2^0}, U^{2^1}, ..., U^{2^{t-1}}$. If these operations cannot be performed efficiently, e.g., if U^{2^k} is implemented in $O(2^k)$ time, then this quantum algorithm offers no advantage over the classical algorithm. However, for special instances of U, it is possible to implement U^{2^k} efficiently, in $O(k)$ time, and then there is a quantum speedup. The most well known application is as a main component of Peter Shors quantum factoring algorithm used to factor numbers in polynomial time. There is no known classical algorithm to accomplish this feat.

The following equations define the quantum phase estimation (QPE) algorithm using the quantum Fourier transform algorithm (QFT) as a sub-component.



(1)



(2)

Equation (1) defines a main sub-component of the algorithm, denoted QPE_0, through recursion over the register of t qubits. The top quantum circuit wire on the left of the equation is the t-th qubit in this register. A Hadamard operation is applied to this qubit and then it is used as a control qubit for conditionally applying $U^{2^{t-1}}$ to the bottom n quantum wires (the other qubit register). The main algorithm is shown in Eq. (2). It applies QPE_0 to both qubit registers, with the top register initialized to the $|0\rangle^{\otimes t}$ state and the bottom register initialized to the the $|u\rangle$ state. It then applies QFT to the top register to produce the output of the algorithm. Measuring the output of the top register is meant to give an estimate of φ.

Since $|u\rangle$ is supposed to be an eigenvalue of U, the state of the bottom register should not be altered by this algorithm (ideally applied). When $U^{2^{k-1}}$ is (conditionally) applied, although the state of the bottom register is unchanged, the overall state is (conditionally) multiplied by $e^{2^k \pi i \varphi}$. When the corresponding control qubit is in a quantum superposition state, the state becomes a superposition of acquiring and not acquiring this scalar factor (one of those odd quantum phenomena). When a Hadamard gate is applied to a $|0\rangle$ input, it transforms into the superposition state $(|0\rangle + |1\rangle)/\sqrt{2}$. When this state is used as the control in a controlled-U^{2^k} gate, the resulting state is the entangled superposition $\left(|0\rangle \otimes |u\rangle + e^{2^k \pi i \varphi}|1\rangle \otimes |u\rangle\right)/\sqrt{2}$ (where the tensor product, $\otimes$, is only between the control qubit of interest and the bottom register). After applying the controlled operation for all of the top register qubits, this will be a superposition over all computational states (all combinations of $|0\rangle$ and $|1\rangle$ states), with different scalar factor that depends upon φ. For the computation state indicated by x_k, where each x is 0 or 1, this factor is $\prod_{k=1}^{t} e^{2^k x_k \pi i \varphi}$. This quantum state encodes φ information, but measuring this state in the computational basis will yield an equal distribution of results since the square of the absolute value of these factors is each equal to one (it is the square of the absolute value of these quantum state amplitudes that determines outcome probabilities). The QFT component transforms this encoding into something useful for the quantum measurement. After this is applied, the top register becomes

$$|\Psi\rangle = \sum_{j=0}^{2^t-1}\left(\frac{1}{2^t}\cdot\left(\sum_{k=0}^{2^t-1}\left(e^{\frac{-(2\cdot\pi\cdot i\cdot k\cdot j))}{2^t}}\cdot e^{2\cdot\pi\cdot i\cdot\varphi\cdot k}\right)\right)\right). \tag{3}$$

Using Prove-It, we will prove that the distribution of measurement outcome probabilities of this t-register state is localized around the approximate binary encoding of φ. In fact, there is a $O(1)$ probability (greater than half) that the measured result is within $3/2^t$ of φ. With the number of qubits and algorithm time scaling polynomially with t (assuming that we can implement U^{2^k} in polynomial time), this is exponentially more efficient than the best known classical algorithm. We have followed the proof on pages 223-224 of Nielsen and Chuang, Quantum Computation and Quantum Information which is a standard reference in the field.

DETAILED DESCRIPTION OF EXPERIMENT/METHOD:

Prove-It is a code base written in the Python programming language that allows one to generate proofs in Python. A proof in the Prove-It system is specified as a derivation tree with a finite number of allowed derivation step types.

Before getting into some details of the Prove-It system, we shall discuss some background philosophy. The following is a list of guiding principles used in the development of the system.

Freedom of expression

Ideally, one should be able to express anything in the Prove-It system that is unambiguous and is standard notation in any field. The internal representation of this expression in Prove-It should be as close as possible to a direct translation to the written expression. We have had to alter some notation that may be standard but is somewhat ambiguous. One example is the use of ellipses () for which we have created our own notation that is unambiguous. But we strive, to the extent possible, to allow a user to be flexible in how they express statements, and allow these statements to be powerful. A theorem is a statement, represented by an expression, which applies generally to a variety of specialized cases. This is how statements may be powerful, and our goal is to allow Prove-It statements to be as limitless as possible.

New notation is defined via new axioms

To be versatile in expressing anything in any standard notation, a user must be able to define their notation independently of the Prove-It core. One does this by adding axioms to the system. Axioms may be any statements that the user deems fit to provide defining properties for their notation and mathematical constructs. Although it is easy to add an axiom that introduces a logical contradiction or is otherwise incorrect, our philosophy is to allow the user to do as they wish but make a clear disclaimer that the proof of any theorem may only be as trusted as the axioms employed in the proof. The axioms employed in a proof are easily tracked and should be clearly indicated (grouped into packages for convenience). As Prove-It gains popularity, the validity of axiom packages will be tested through crowd sourcing.

Prove-It should have a lightweight core

Since most of the notation that is used in Prove-It is defined via axioms that are added to the system, the core of Prove-It can be very lightweight. The Prove-It core only needs to understand a few core expression types and derivation rules in order to derive virtually anything from the use of powerful axiom and theorem statements.

Harmless nonsense is... well... harmless

To truly be flexible and robust, axioms that allow one to derive harmless nonsense should be embraced rather than restricted. It can be very cumbersome to make restrictions that prevent nonsensical statements from being derived. Being overly restrictive can limit the utility of the system. On the other hand, nonsense can be completely harmless. Consider the statement $\forall_{A,B} A \land B \Rightarrow A$ where $\forall$ is the universal quantifier (forall), $\land$ is the logical and operator, and $\Rightarrow$ is the implies operator (logical implication). Since $\land$ is an operation that is only defined when applied to Booleans, the quantifier should perhaps be restricted to A and B being in the set of Booleans (True or False): $\forall_{A,B \in \text{BOOLEANS}} A \land B \Rightarrow A$. However, if one specializes the original statement to non-Boolean objects, it derives a harmless statement because the hypothesis of the implication cannot be proven. For example, specializing A to 5 and B to 10 will produce $5 \land 10 \Rightarrow 5$. Fine. This doesnt make sense, but since we cannot prove that $5 \land 10$ is a true statement (with reasonable axioms), then this does not allow one to prove that 5 is a true statement. It is therefore harmless. While it is possible to use Prove-It only with strict axioms that prevent nonsense, it is designed to be flexible.

Statements are represented with expression trees. Each expression (including sub-expressions) is an instance of a core Prove-It expression type. The core of Prove-It only needs to understand expressions in the context of core expression types. Most employed notation is defined through axioms. The Prove-It core only needs to know how to apply (specialize) these axioms. Beyond that, the core is agnostic to ones notation (with just a couple of exceptions). The following is a complete list of the core expression types with descriptions:

Variable

In Prove-It, a *variable* is a label without any inherent meaning. The meaning of a statement should not ever change if one replaces a *variable* with any other variable, as long as distinct variables remain distinct.

Literal In contrast to *variable*, *literal*s are labels that do have contextual meaning. They are defined through axioms (with just a couple of exceptions for special *literal*s that are understood at the core level). Specific operators are *literal*s, and so are numbers, labeled sets (reals, integers, ...), etc. If it is a label with a meaning, it must be a *literal* in Prove-It.

Operation

An *operation* consists of an operator and operands. Each of these is a sub-expression. The operator is often a *literal*, but it can also be a *variable* or a Lambda function. For example, in the statement $\forall_{f,x,y}(x = y) \Rightarrow (f(x) = f(y))$ (the substitution axiom), f is a variable acting as an operator because it is a stand-in for any operator. The operands is an *expression-list* or *expression-tensor*.

Lambda

A *lambda* function consists of arguments and the lambda expression. It represents a mapping from the arguments to an expression that involves the arguments. For example, $(x, y) \rightarrow x + y$. The arguments is an *expression-list*.

Named-Expressions

A *named-expressions* expression maps each of a set of names (any string) to a sub-expression. Essentially, this simply labels the sub-expressions and is useful for making the internal representation of the expression unambiguous (e.g., when the order of the operands does not intrinsically distinguish their role).

Expression-List

An *expression-list* is an ordered list of any number of sub-expressions.

Expression-Tensor

An *expression-tensor* maps lists of indices to sub-expressions. This is useful for expressing matrices, quantum circuits, or anything with a 2-dimensional representation (or higher dimensional). The tensor may be sparse (not every combination of indices needs a sub-expression), but the dimensionality must be consistent (each list of indices must have the same length).

Multi-Variable

A *multi-variable* is a stand-in for any number of *variable*s. It is denoted with a box index, such as $x_\square$. This notation is meant to convey the notion that there is an x_1 ,x_2, ... except that the actual index labels are irrelevant and the number of them is unspecified (it may be replace with zero *variable*s, in fact). A *multi-variable* must be a sub-expression of an *etcetera*, described below.

Etcetera

An *etcetera* expression is a placeholder that may be expanded to any number of sub-expressions within an *expression-list*. This is our way to represent ellipses (...) but without any ambiguity. The following are examples of what one may do using *etcetera* (and *multi-variable*s):
$\forall_{...,x_\square,...,y_\square,...} .. + x_\square + ... + y_\square + .. = .. + y_\square + ... + x_\square + ..$
may be specialized, for example, to $a + c + d = c + d + a$ in one step.
$\forall_{x,..,y_\square,..} x \cdot (.. + y_\square + ..) = .. + x \cdot y_\square + ..$
may be specialized, for example, to $a \cdot (b + c + d) = a \cdot b + a \cdot c + a \cdot d$ in one step.

Block

Block is similar to *etcetera* but expands into an *expression-tensor* rather than an *expression-list*. This is useful in the context of quantum circuits for substituting a multi-qubit gate for an entire sub-circuit.

The concepts for the *expression-tensor*, *multi-variable*, *etcetera*, and *block* expression types were developed over the course of this project. Working out an appropriate treatment for these concepts was a considerable challenge that we encountered. These concepts may evolve further in the future.

Ideally, there is a direct translation between internal and external representations within Prove-It. However, this is not enforced in any way. The translation from the internal to the external representation is at the discretion of the user and is very flexible in terms of formatting expressions using LaTeX. As with the freedom to add axioms at will, this brings potential danger but we resolve to allow users to proceed as they see fit with clear disclaimers. The external representations are only for convenience and may only be trusted to the extent that they faithfully convey the internal representation. One must not only check the axioms employed in a proof for their validity, but also check the internal representations of the axioms and the internal representation of the theorem being proven. To the extent that the internal and external representations are direct translations of each other, this is relatively straightforward.

A proof in Prove-It is a derivation tree that deduces a theorem (the root of a tree) from a set of axioms and/or theorems (the leaves of the tree). Each derivation step indicates how a particular statement is proven under a set of assumptions given previously proven statements (possibly with other assumptions). The following is a list of the recognized derivation step types with descriptions:

Axiom/theorem invocation

Any axiom or theorem can be invoked within a proof and will be accepted as truth in the context of a proof. An axiom or theorem may be any Prove-It expression that contains no free *variable*s (meaning that all *variable*s must be bound explicitly as *lambda* arguments). For any proof, it is possible to trace back all used axioms (directly or indirectly via the proof of a used theorem) and all used unproven theorems. A proof is not complete unless all of theorems that it uses directly or indirectly have complete proofs for themselves.

Assumption

Any statement may be taken to be true by assumption. The assumption must be carried up toward the root of the derivation tree, in the set of required assumptions, until it is eliminated through hypothetical reasoning or generalization conditions.

Relabeling

Changes *variable*s. Since *variable*s are labels with no intrinsic meaning, the meaning of the statement is unchanged by relabeling except when changing which *variable*s are distinct from each other. For example, relabeling $P(x, y)$ into $P(a, a)$ does have a different meaning. However, the statement is only weakened by such a change, so this derivation step is allowed.

Specialization Eliminates universal quantification (a forall operation). Substitutes each quantified *variable* with an expression and each quantified *multi-variable* with an *expression-list* (using the *etcetera*/*block* machinery described above). Requires proof of all conditions placed upon those variables from the original forall operation. These are added as branches in the derivation tree. For example, one may specialize $\forall_{A \in \text{BOOLEANS}} A \Rightarrow (A \vee B)$ by replacing A with $P(x)$ as long as one can also satisfy the condition that $P(x) \in$ BOOLEANS. The process of specialization may introduce unbound variables that are taken as arbitrary variables (e.g., P and x in the previous example would be unbound, arbitrary *variable*s, unless they happen to be *literal*s). These may be bound further up the derivation tree using *generalization*.

Generalization

Introduces universal quantification (a forall operation) over unbound Variables as desired. May apply any domain restriction or condition on this universal quantification. These only weaken the generalized statement, making it no less proven than the unconditional forall statement. Applied domain restrictions or conditions may serve to eliminate assumptions. For example, $\forall_{x \in S} P(x)$ eliminates the assumption $x \in S$ because this statement is true under the set of assumptions Ω as long as we can prove $P(x)$ under the assumptions $\Omega \bigcup \{x \in S\}$.

Implication (modus ponens)
Proves that some statement B is true after proving statements of the form $A \Rightarrow B$ and A.

Hypothetical reasoning
Proves a statement of the form $A \Rightarrow B$ after proving statement B using statement A as an assumption. Eliminates the hypothesis (e.g., statement A) as an assumption.

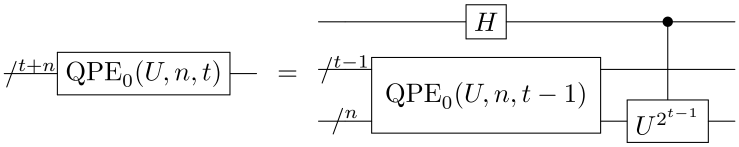
Axiom elimination
Transforms a set of axioms into a set of assumptions. When tracking the axioms that are employed in a proof, these ones may not be counted (unless used in another branch of the derivation tree) because they are ultimately not necessary for the proof if they can be transformed into assumptions and then eliminated. In the process, *literal*s that only appear in the axioms being eliminated are transformed into Variables (their axiomatic, contextual meaning has been lost except within the assumptions that are carried along in the derivation tree). This is useful since all definitions in Prove-It are made via *axiom*s involving *literal*s and some definitions are only a temporary convenience. In our quantum phase estimation proof, we employ this to define some useful *literal*s for the problem set-up (such as U, u, φ, t, etc.) that are used in multiple theorems but later transformed into Variables and quantified over universally (i.e., $\forall_{U,u,\varphi,t,\ldots}$) in the final theorem.

RESULTS:

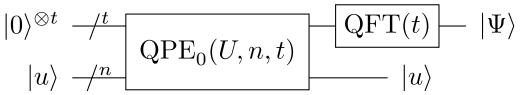
We begin our verification of the quantum phase estimation by define a set of axioms that defines the problem. This involves defining both temporary and permanent *literal*s. By temporary *literal*s, we mean convenient labels for setting up the problem and using throughout the proof, but ones that we ultimately would like to quantify over (as *variable*s) for a final theorem that may be used outside of this context. This would be done through *axiom elimination* (which has not yet been implemented in the system, but the concept is straightforard). The permanent *literal*s, in contract, are necessary in order to define what the proof means. In particular, a label for the algorithm is a permanent *literal*. Its definition is required in order to interpret the proof.

The following is a list of all of the axioms asserted within the quantum phase estimation context. We provide some further descriptions below.

1. $U \in SU(n)$
2. $\varphi \in [0, 1)$
3. $(U|u\rangle) = \left(e^{2 \cdot \pi \cdot i \cdot \varphi}|u\rangle\right)$
4. $t \in \mathbb{N}^{+}$
5. $\forall_{U,n,t}$



6.



7. $m = \mathcal{M}\left(|\Psi\rangle\right)$
8. $\varphi_m = \frac{m}{2^t}$
9. $b = \lfloor \varphi \cdot 2^t \rfloor$
10. $\delta = \left(\varphi - \frac{b}{2^t}\right)$
11. $\forall_{\varepsilon\in\mathbb{Z}}\left(P_{\text{success}}\left(\varepsilon\right) = Pr[(|(m-b)|_{\text{mod } 2^t} \leq \varepsilon)]\right)$
12. $\forall_{\varepsilon\in\mathbb{Z}}\left(P_{\text{fail}}\left(\varepsilon\right) = \left(1 - P_{\text{success}}\left(\varepsilon\right)\right)\right)$
13. $\forall_{a,b\in\mathbb{Z}}\left(\left(a \oplus b\right) = \left(\left(a+b\right) \mod 2^{\mathrm{t}}\right)\right)$
14. $\forall_{l\in\mathbb{Z}}\left(\alpha_l = \left(\langle b \oplus l||\Psi\rangle\right)\right)$

Axioms 1-3 set up the basic quantum phase estimation problem. The goal of the algorithm is to estimate φ for a given U (unitary quantum operation) and $|u\rangle$ (ket, or quantum state). The U operation involves n qubits. Axioms 4-8 defines the quantum phase estimation algorithm using a quantum circuit represantation and applies it to the problem of interest for a particular number of register qubits t. The precision of the estimate is determined by t. This is the main fact that we seek to prove. Axiom 5 [the same as Eq. (1] defines one component of the quantum phase estimation circuit denoted QPE_0 using a recursive definition. In this definition, U, n, and t are *variable*s that are universally quantified (they happen to have the same representation as *literal*s with which they correspond). Axiom 6 [the same as Eq. (2)] defines the rest of the quantum circuit that involves the quantum Fourier transform as a component denoted QFT. The quantum Fourier transform algorithm is a sub-component that requires its own independent proof and is out of our scope. It provides an exponential speedup over classical algorithm on its own. Axiom 6 implicitly defines $|\Psi\rangle$ as the output quantum state of the algorithm. Axiom 7 defines m to be the random variable outcome of the quantum measurement of $|\Psi\rangle$, the output of the quantum algorithm ($\mathcal{M}$ denotes quantum measurement). Axiom 8 defines φ_m to be the random variable estimate of φ. Axioms 9-12 are all related to defining success versus failure of the quantum algorithm output. Axiom 9 defines b to be the outcome of m that would give us the closest estimate to φ without exceeding it. Axiom 10 defines δ to be the difference between φ and this closest undershooting estimate. Axiom 11 defines the probability that the outcome succeeds in being within some ε of b, and Axiom 12 defines the corresponding probability of failure. Axioms 13 and 14 are convenient definitions within the proof. Axiom 13 defines a short-hand for adding integers modulo 2^t. Axiom 14 defines α_l to be the amplitude of the outcome state $|\Psi\rangle$ to a state denoted by l that is relative to b.

Below we list each of the theorems that we have either proven, or intend to prove, within the quantum phase estimation context. For each theorem, we indicate whether or not we have produced the proof, the lines of code to generate the proof, the number of theorems/axioms that this proof employed, and the number of unique nodes in its derivation tree (nodes are often repeated in derivation trees but we will not count them separately). We also provide brief notes about each theorem to indicate theorems/axioms that it derives from and how it fits into the larger picture. For unproven theorems, we indicate what it would require to finish them. Strictly speaking, none of our theorems are *complete* because they all rely upon theorems in other contexts that we have not yet proven in the Prove-It system (for algebraic manipulation and various other well-known facts).

1. $2^t \in \mathbb{N}^+$
   status: proven | **lines of code:** 10 | **used theorems/axioms:** 8 | **unique nodes:** 19
   Derives from Axiom 4 ($t \in \mathbb{N}^+$) and number set properties.
2. $2^{t-1} \in \mathbb{N}^+$
   status: proven | **lines of code:** 9 | **used theorems/axioms:** 12 | **unique nodes:** 26
   Derives from Axiom 4 and number set properties.

3. $(2^t - 1) \in \mathbb{N}^+$
   status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
   Derives from Axiom 4. Since $t \geq 1$ in order to be in $\mathbb{N}^+$, $(2^t - 1) \geq \left((2^1 - 1) = 1\right)$. This can be done easily, we just did not have time.

4. $2^t \neq 0$
   status: proven | **lines of code:** 9 | **used theorems/axioms:** 8 | **unique nodes:** 26
   Derives from Axiom 4 and number set properties.

5. $\forall_{a,b\in\mathbb{Z}}\left((a \oplus b) \in \mathbb{Z}\right)$
   status: proven | **lines of code:** 14 | **used theorems/axioms:** 6 | **unique nodes:** 16
   Derives from Axiom 13 (the definition of our $\oplus$ notation), substitution, and number set properties.

6. $\varphi \in \mathbb{R}$
   status: proven | **lines of code:** 6 | **used theorems/axioms:** 5 | **unique nodes:** 10
   Derives from Axiom 2 ($\varphi \in [0, 1)$) and number set properties.

7. $b \in \mathbb{Z}$
   status: proven | **lines of code:** 7 | **used theorems/axioms:** 13 | **unique nodes:** 30
   Derives from Axiom 9 (the definition of b), Theorem 1, Theorem 6, substitution, and number set properties.

8. $\forall_{\varepsilon\in\mathbb{N}^+}\left(\forall_{l\in\{(\varepsilon+1)\ldots 2^{t-1}\}}\left(l \in \{\left(\left(-2^{t-1}\right)+1\right)\ldots 2^{t-1}\}\right)\right)$
   status: proven | **lines of code:** 25 | **used theorems/axioms:** 21 | **unique nodes:** 67
   Derives from Theorem 3 and number set properties as well as some properties of ordering relations (less/greater than, etc.).

9. $\forall_{\varepsilon\in\mathbb{N}^+}\left(\forall_{l\in\{((-2^{t-1})+1)\ldots(-(\varepsilon+1))\}}\left(l \in \{\left(\left(-2^{t-1}\right)+1\right)\ldots 2^{t-1}\}\right)\right)$
   status: proven | **lines of code:** 25 | **used theorems/axioms:** 20 | **unique nodes:** 69
   Derives from Theorem 3 and number set properties as well as some properties of ordering relations (less/greater than, etc.).

10. $(2^t \cdot \delta) \in [0, 1)$
    status: proven | **lines of code:** 29 | **used theorems/axioms:** 6 | **unique nodes:** 13
    Derives from Axiom 9 and 10 (the definition of δ) as well as Theorems 1, 6, and 7 along with number set properties and algebraic manipulations.

11. $\delta \in \mathbb{R}$
    status: proven | **lines of code:** 7 | **used theorems/axioms:** 13 | **unique nodes:** 26
    Derives from Axiom 10 (the definition of δ) and Theorems 1, 6, and 7 along with substitution and number set properties.

12. $\forall_{l\in\mathbb{Z}}\left(\alpha_l \in \mathbb{C}\right)$
    status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
    Derives from Axiom 14 (the definition of α_l) and a fundamental property of quantum state projections. This is straightforward, but we did not have time to get to it.

13. $\forall_{l\in\mathbb{Z}}\left((|\alpha_l| \in \mathbb{R}) \wedge (|\alpha_l| \geq 0)\right)$
    status: proven | **lines of code:** 18 | **used theorems/axioms:** 4 | **unique nodes:** 10
    Derives from Theorem 12 and number set properties.

14. $\forall_{l\in\mathbb{Z}\ \mid\ l\neq 0}\left(\left(2^t\cdot\delta\right)\neq l\right)$
status: proven | **lines of code:** 40 | **used theorems/axioms:** 22 | **unique nodes:** 66
Derives from Theorem 10, number set properties, and some algebraic and logical manipulations. This is a proof by contradiction.

15. $\forall_{l\in\mathbb{Z}\ \mid\ l\neq 0}\left(\delta\neq\frac{l}{2^t}\right)$
status: proven | **lines of code:** 29 | **used theorems/axioms:** 18 | **unique nodes:** 53
Derives from Theorem 1 and 14, number set properties, and some algebraic and logical manipulations. This is a proof by contradiction as well.

16. $\forall_{l\in\{((-2^{t-1})+1)\ldots 2^{t-1}\}}\left(\left(\delta-\frac{l}{2^t}\right)\in\left[\left(-\frac{1}{2}\right),\frac{1}{2}\right)\right)$
status: proven | **lines of code:** 69 | **used theorems/axioms:** 60 | **unique nodes:** 224
Derives from Axiom 4 ($t\in\mathbb{N}^+$) and Theorems 1, 2, 6, and 10, as well as number set properties and algebraic manipulations.

17. $\forall_{l\in\{((-2^{t-1})+1)\ldots 2^{t-1}\}}\left(\left(2\cdot\pi\cdot\left(\delta-\frac{l}{2^t}\right)\right)\in\left(\left(-\pi\right),\pi\right)\right)$
status: proven | **lines of code:** 36 | **used theorems/axioms:** 25 | **unique nodes:** 79
Derives from Theorem 16, number set properties, and algebraic manipulations.

18. $\forall_{l\in\{((-2^{t-1})+1)\ldots 2^{t-1}\}\ \mid\ l\neq 0}\left(\left(\delta-\frac{l}{2^t}\right)\notin\mathbb{Z}\right)$
status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
Derives from Theorem 16 in a fairly straightforward manner. We simply did not have time to implement this.

19. $\forall_{\varepsilon\in\mathbb{N}}\left(P_{\text{success}}\left(\varepsilon\right)=\left(\sum_{l=-\varepsilon}^{\varepsilon}Pr((|(m-b)|=l))\right)\right)$
status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
Derives from Axiom 11 (the definition of P_{success}), modular arithmetic, and basic probability theory (summing the probabilities of independent events). This would not be difficult to implement, but we did not have time.

20. $\forall_{\varepsilon\in\mathbb{N}^+}\left(P_{\text{fail}}\left(\varepsilon\right)=\left(\left(\sum_{l=(-2^{t-1})+1}^{-(\varepsilon+1)}\left|\alpha_l\right|^2\right)+\left(\sum_{l=\varepsilon+1}^{2^{t-1}}\left|\alpha_l\right|^2\right)\right)\right)$
status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
Derives from Axiom 12 (the definition of P_{fail}), 14, and 15 (the definition of α_l), and from Theorem 19 along with algebraic manipulations and the fact that sum of all possible, distinct outcome probabilities is equal to one. This would require a little work, but there is no significant obstacle to producing this proof.

21. $\forall_{a,b\in\mathbb{Z}}\left(e^{\frac{2\cdot\pi\cdot i\cdot(a\oplus b)}{2^t}}=e^{\frac{2\cdot\pi\cdot i\cdot(a+b)}{2^t}}\right)$
status: proven | **lines of code:** 25 | **used theorems/axioms:** 13 | **unique nodes:** 35
Derives from Axioms 4 and 13 (the definition our $\oplus$ notation) as well as some algebraic manipulations and the trigonometry-related identity that $\forall_{x,r\in\mathbb{R}}\left(e^{\frac{2\cdot\pi\cdot i\cdot(x \bmod r)}{r}}=e^{\frac{2\cdot\pi\cdot i\cdot x}{r}}\right)$.

22. $\forall_{l\in\mathbb{Z}}\left(\alpha_l=\left(\frac{1}{2^t}\cdot\left(\sum_{k=0}^{2^t-1}\left(e^{\frac{-(2\cdot\pi\cdot i\cdot k\cdot(b\oplus l))}{2^t}}\cdot e^{2\cdot\pi\cdot i\cdot\varphi\cdot k}\right)\right)\right)\right)$
status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
Derives from the definition of the quantum phase estimation algorithm defined in Axioms 5 and 6 as well as the definition of α_l in Axiom 14, and the definition of our modular arithmetic shorthand in Axiom 13. This is not completely trivial, but comes from a relatively direct translation of the quantum circuit operations on the input states. There is no significant obstacle here, but we did not have time to do this.

23. $\varphi = \left(\frac{b}{2^t} + \delta\right)$
 status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
 This is very simply derived from Axiom 10 (the definition of δ) but we did not have time.

24. $\forall_{l \in \mathbb{Z}} \left(\alpha_l = \left(\frac{1}{2^t} \cdot \frac{1-e^{2\cdot\pi\cdot i\cdot((2^t\cdot\delta)-l)}}{1-e^{2\cdot\pi\cdot i\cdot\left(\delta-\frac{l}{2^t}\right)}}\right)\right)$
 status: proven | **lines of code:** 92 | **used theorems/axioms:** 62 | **unique nodes:** 285
 Performs the summation in the expression for α_l in Theorem 22 as a finite geometric series. Also uses the definition of our $\oplus$ notation via Axiom 13, $t \in \mathbb{N}^+$ from Axiom 4, $(2^t - 1) \in \mathbb{N}^+$ from Theorem 3, $b \in \mathbb{Z}$ from Theorem 7, φ and $\delta \in \mathbb{R}$ from Theorems 6 and 11, the identity of Theorem 21, and the relation between b, φ, and δ from Theorem 23. There is also various algebraic manipulations, substitutions, and number set properties employed.

25. $\forall_{l \in \mathbb{Z}} \left(|\alpha_l| = \frac{\left|1-e^{2\cdot\pi\cdot i\cdot((2^t\cdot\delta)-l)}\right|}{2^t\cdot\left|\left(1-e^{2\cdot\pi\cdot i\cdot\left(\delta-\frac{l}{2^t}\right)}\right)\right|}\right)$
 status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
 Very easy to derive from Theorem 24, but we did not have time to do this.

26. $\forall_{l \in \{((-2^{t-1})+1)\ldots 2^{t-1}\}\ |\ l \neq 0} \left(|\alpha_l|^2 \leq \frac{1}{4\cdot(l-(2^t\cdot\delta))^2}\right)$
 status: proven | **lines of code:** 197 | **used theorems/axioms:** 74 | **unique nodes:** 425
 Bounds $|\alpha_l|^2$ using the expression for $|\alpha_l|$ from Theorem 25. It uses Axiom 4 and Theorem 2 ($t \in \mathbb{N}^+$ and $2^{t-1} \in \mathbb{N}^+$), Theorem 12 and 13 ($\alpha_l \in \mathbb{C}$ and related properties), Theorem 11, 15, 17, and 18 ($\delta \in \mathbb{R}$ and more specific constraints that avoid division by zero and enable the upper bounding). Specifically, we bound $|\alpha_l|^2$ using $\forall_{\theta \in [(-\pi),\pi]} \left(\left|\left(1-e^{i\cdot\theta}\right)\right| \geq \frac{2\cdot|\theta|}{\pi}\right)$ and $\forall_{\theta \in \mathbb{R}} \left(\left|\left(1-e^{i\cdot\theta}\right)\right| \leq 2\right)$ (theorems that we have not proven in our system). We also employ various algebraic manipulations, including manipulations of inequalities.

27. $\forall_{\varepsilon \in \{1\ldots(2^{t-1}-2)\}} \left(P_{\text{fail}}\left(\varepsilon\right) \leq \left(\frac{1}{2} \cdot \left(\frac{1}{\varepsilon} + \frac{1}{\varepsilon^2}\right)\right)\right)$
 status: proven | **lines of code:** 303 | **used theorems/axioms:** 115 | **unique nodes:** 658
 Uses the expression for $P_{\text{fail}}(\varepsilon)$ from Theorem 20 and the bound of $|\alpha_l|^2$ from Theorem 26 along with number set restrictions of Axiom 4 and Theorems 2, 8, 9, 10, 11, 12, 13, and 14. Employs various algebraic manipulations, including manipulations of inequalities and summations (e.g., splitting summations apart over separate ranges, and the fact that an inequality for all summand instances implies the inequality of the summations). It also uses the fact that $l \rightarrow 1/l^2$ is an even function so that $\forall_{a,b \in \mathbb{Z}} \left(\left(\sum_{l=a}^{b} \frac{1}{l^2}\right) = \left(\sum_{l=-b}^{-a} \frac{1}{l^2}\right)\right)$, that it is a monotonically decreasing function so that $\left(\sum_{l=\varepsilon}^{2^{t-1}-1} \frac{1}{l^2}\right) \leq \left(\frac{1}{\varepsilon^2} + \int_{\varepsilon}^{2^{t-1}-1} \frac{1}{l^2} dl\right)$. These facts come from theorems not yet proven in the Prove-It system. We assert (as an unproven Prove-It theorem) that $\forall_{a,b \in \mathbb{R}^+\ |\ a \leq b} \left(\int_a^b \frac{1}{l^2} dl \leq \frac{1}{a}\right)$.

28. $P_{\text{fail}}(2) \leq \frac{3}{8}$
 status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
 Simple arithmetic applied to Theorem 27, but we did not have time to do this.

29. $P_{\text{success}}(2) \geq \frac{5}{8}$
 status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
 Simple arithmetic applied to Axiom 12 (relating P_{success} and P_{fail}) and Theorem 28, but we did not have time to do this.

30. $Pr\left[|(\varphi_m - \varphi)|_{\text{mod } 1} < \frac{3}{2^t}\right] \geq \frac{5}{8}$
status: unproven | **lines of code: -** | **used theorems/axioms: -** | **unique nodes: -**
Derives from Theorem 29 bounding P_{success} and Theorem 10 bounding δ. Theorem 29 indicates that there at least 5/8 probability that m is within 2 units of b. Theorem 10 indicates that b is less than one unit from the true answer. Therefore, we have at least a 5/8 probability of measuring an answer that is within three $1/2^t$ units from the true answer.

An additional theorem, not listed, would be to prove the success probability (or probability distribution) of the algorithm quantified over all appropriate values of U, $|u\rangle$, and t via *axiom elimination*. We have not yet implemented *axiom elimination* in the Prove-It system, but this step would be straightforward. Furthermore, to be complete, we would want to prove that the depth (time) and width (number of qubits) of the quantum circuit for the quantum phase estimation algorithm both scale linearly with t. This fact is obvious and would not be difficult to prove in our system.

DISCUSSION:

In the course of this work, we added/modified several thousands of lines of code in the Prove-It software package in addition to the lines of code indicated above for each theorem proof. Proof code is specific to the corresponding proof but it relies upon the broader software package. The core of Prove-It is not meant to expand significantly (though it did undergo a few modications). There is a layer in between the core and proof code that is intended to expand. This layer includes the axioms and theorems of an expanding suite of packages and code that makes it convenient to exploit this background knowledge. A particular convenience is the use of object oriented programming in Prove-It. Typically, each type of *operation* (identified by its *literal* operand) has a corresponding Python class with member functions that conveniently apply theorems that or specific to that type of *operation*. This is the best way to utilize prove it. It avoids having to remember over one hundred theorem names (as required in the proof of Theorem 27, for example) for different uses and special cases. Instead, one learns method names associated with *operation* types for performing general functions or transformations that could invoke one of more of several possible theorems depending upon the specific case. An instance of an *operation* class will know about its expression form and can “decide” what theorem(s) is/are appropriate to perform the desired function/transformation. The idea is to mimic the way that a mathematician, scientist, or engineer organizes their knowledge of math. We group related procedures together that can take several forms. For example, cancelation has a general meaning but takes on many forms. It can be cancelation with respect to a fraction with one or more factors in the numerator and one or more factors in the denominator. Or it can be cancelation with respect to subtraction with one or more terms on either side of the minus sign.

In the course of the current project, we performed a mix of this “best” practice as well as some less ideal implementations for expediency. We implemented good, convenient methods for different types of factoring, distributing, and many cases of cancellation. For many of the manipulations of ordering relation inequalities, in contract, we used theorems on a case by case basis. There were also various specialty theorems that we used on a case by case basis in order to accomplish our task in the short time that we had. This was planned from the beginning (in the proposal). For special cases that are not generally useful as named theorems on their own, it would be better to write code that automates the proofs. For example, we could automate the evaluation of simple arithmetic, of basic integrals, or other instances where procedures are straightforward to automate. Along these lines, we did write some useful code for automating various number set proofs, automatically applying “closure” theorems. For example, for an arbitrary arithmetic expression that is well defined (without division by zero, etc.), we can automatically deduce that it is in the set of complexes, denoted $\mathbb{C}$, if all of its lowest-level components are known to be in the set of complexes, or any of its subsets. This automation is quite versatile and was extremely valueable for producing these proofs. That said, it could certainly use some improvements. Our proofs could easily be simplified (using fewer lines of code and made to be more transparent) with improvements in the code base. However, we do have a very nice demonstration of our theorem capabilities as it is.

Another improvement for the future will be to add more tools to aid the user in exploring the graph of derivations that Prove-It uses to produce a derivation tree. It can be very frustrating for a user when Prove-It is not able to prove a statement for reasons that are not clear. It takes time to figure out what steps are missing. With tools to explore the graph that Prove-It uses internally, missing steps will become apparent much more quickly.

Future work is also needed to fully implement a proof certification system. We have a plan that would allow communication of proofs from untrusted parties. The user would have control over axioms/theorems that are added or changed, and the derivation trees of proofs would be exported and imported without running the other party's proof code.

ANTICIPATED IMPACT:

Prove-It is a useful, evolving tool for a wide array of formal methods applications. Many types of critical systems are developed at Sandia. Engineering systems that are formally verified is of paramount importance to its mission. Investments in quantum technology and other promising cutting-edge research can be made wisely and with more confidence using formal verification of key, theoretical insights. Across the diverse areas of research at Sandia, trust and assurance is a greatly valued asset.

As the impact of Prove-It becomes broader, its inherent value also increases. As more people use Prove-It, it will inevitably have more contributors. As more people contribute to Prove-It, it becomes more useful. As it becomes more useful, more people will use it. Thus, there is an intrinsic positive feedback. At some point, it (or something like it) will hopefully reach a self-sustaining critical mass. Wikipedia reached such a critical mass many years ago and now it is regarded as an invaluable resource. Ideally, Prove-It (and/or other similar tools) will become the backbone to some kind of "wiki-qed" portal of shared mathematical knowledge. This system could also be used for private (proprietary or classified) mathematical and engineering knowledge, but having a shared resource of public knowledge for theorem proving would really be invaluable for researchers.

This is the dream. It draws inspiration from the QED Manifesto, a document published in 1994 pushed by Robert Boyer with input from several researchers. This document argues for a QED system that is a repository of formalized mathematical knowledge. In their vision, it would arise from a large scale, collaborative effort orchestrated by a research agency. However, as proven by the success of Wikipedia, it is possible to build a critical mass with a modest investment. If communities of researchers perceive that it is a useful tool, they will use it, contribute to it, share, and the system will grow.

Our modest demonstration of the quantum phase estimation algorithm is not likely sufficient to garner the broad interest required for a snowball effect of gaining contributors. Hopefully, it will draw some interest from the quantum information community. This is a very intelligent community that is open and receptive to creative, new ideas. We will promote our work to this community. We are considering the development of quantum circuit manipulation tools that would be useful to this community.

Before this project began, Prove-It was developed by Wayne Witzel (with ideas, encouragement, and support from Robert Carr, a Sandia employee at the time) on "sweat equity" during non-working hours. As a test of the system, we added a Boolean algebra framework (external to the core) with theorems proven completely down to base axioms (facts that cannot be derived from anything that is more fundamental). We built theorems up to point at which any Boolean expression, including quantification over Boolean values, can be automatically evaluated (with no claims to efficiency, unlike SAT solvers). The ability to evaluate Boolean expressions is not particularly impressive, but proving the evaluation to the level of base axioms was a valuable demonstration of our approach.

Using the support from this project, we have extended the Prove-It system to treat numbers (integers, reals, and complexes). Our number theory framework is far from complete. A much larger effort would be required to make complete a set of useful theorems and procedures for algebraic manipulations. With a relatively modest effort, we could extend the algebraic manipulation features of the system to make theorem proving involving numerical systems much more efficient in Prove-It. The difference is that the modest investment would not complete proofs down to base axioms. Our demonstration regarding the quantum phase estimation algorithm shows the value of a flexible system that allows "shallow" proofs. Our proofs are not complete down to base axioms, but we have the list of unproven theorems (as well as the axioms that we assert to be true). Each of these can be can be checked and proven to a deeper level when there is down.

Building the system from the bottom (axioms) to the top (high-level theorems) is satisfying. However, allowing users to work in either direction may be the key to gaining users without a substantial investment. In this manner, the system is useful far beyond the extent to which it is complete (with respect to having proofs down to the axiom level). Thus, there is hope. If we can make the system useful with simple tricks, we can attract users that will help with the arduous taks of completing the foundation.

CONCLUSION:

We have demonstrated the utility and flexibility of our Prove-It system by producing the major components for the computer-validated proof of the quantum phase estimation algorithm based upon a less formal proof presented in "Quantum Computation and Quantum Information" by Isaac Chuang and Michael Nielsen. We have presented the theorems, particular to this problem, that we have proven and those that remain to be proven and have described their dependencies. For the theorems that we have proven, we indicate the size of the program that builds the proof, the number of theorems and axioms it required, and size of the its derivation tree. For the unproven theorems, we indicate the effort we expect is required to program the proof. Certainly, we have fished the most challenging parts of the proof. The remaining pieces are relatively straightforward.

None of our proofs are "complete" in the sense that they rely upon other theorems that are not yet proven in this system. There are a large number of facts with respect to arithmetic, algebra, trigonometry, calculus, and quantum mechanics that we accept without proof for the purposes of this demonstration. This is consistent with our proposal and intention. It demonstrates a flexibility to work from top to bottom as well as bottom to top with respect to low-level axioms versus high-level theorems. This flexibility makes Prove-It be useful beyond the extent to which it is strictly complete. This may be the key to building a user base without a substantial investment. When the user base reaches a critical mass, the system will be an invaluable tool for a diverse array of researchers.