Towards Higher-Level Abstractions for Quantum Computing

Adrian Cobb
Deakin University,
Geelong, VIC
Australia
alcobb@deakin.edu.au

Jean-Guy Schneider
Deakin University,
Geelong, VIC
Australia
jeanguy.schneider@deakin.edu.au

Kevin Lee
Deakin University,
Geelong, VIC
Australia
kevin.lee@deakin.edu.au

ABSTRACT

Quantum Computing (QC) has emerged as a field of ever-increasing activity as it promises to revolutionize computation and enable the solution of computational problems that we (realistically) cannot solve with Classical Computing to date. However, existing quantum programming environments mostly require an in-depth understanding of the basic QC building blocks, that is, quantum states, superposition, entanglement and measurement as well as the changing of quantum states using basic quantum gates. The present state of quantum programming reminds us of how Classical Computing was 60+ years ago when computing machines such as the ENIAC required significant effort to program solely using very basic digital building blocks. Over the decades, though, increasingly higher-level abstractions have been crated on top of the basic building blocks of Classical Computing and made computation much more accessible and wide-spread. In order to make Quantum Computing more accessible, we argue that Software Engineering for QC needs to embark on a similar journey and create abstractions that shield developers from the basic QC building blocks as much as possible so that they can focus their attention on solving problems and less on how to manipulate quantum sates using quantum cirquits. Based on our experience of developing a scaling quantum *n*-queens solver, this paper aims to formulate recommendations for raising the level of abstraction in Quantum Software Engineering.

KEYWORDS

Quantum Computing, Quantum Software Engineering, Programming Abstractions

ACM Reference Format:
Adrian Cobb, Jean-Guy Schneider, and Kevin Lee. 2021. Towards Higher-Level Abstractions for Quantum Computing. In *Australasian Computer Science Week Multiconference (ACSW '22), February 1–5, 2021, Virtual*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Quantum computing forms part of the larger field of *quantum information sciences* [15] and has been conjectured as the next major breakthrough in computing [20]. By harnessing the collective properties of quantum states, such as superposition, interference, and entanglement, it is conjectured that a new class of problems intractable on a classical computer suddenly becomes within reach [17].

The current state of Software Engineering tools for Quantum Computing requires a different level of knowledge compared to today's Classical Computing architectures [21]. Classically, information is stored in Bits that are represented logically by either a 0 or 1 and software development tools support levels of abstraction where a developer does not need to individually program state changes for each Bit. Fundamental to many Software Engineering tasks is that inspecting the state of a Bit does not affect its state, something we fundamentally take for granted during program inspection.

In Quantum Computing, Bits are replaced by Quantum Bits (or *qubits*) that have a certain probability of being $|0\rangle$ or $|1\rangle$, respectively [15]. The main task of a developer creating a quantum program is to understand how qubits interact with quantum gates through superposition and entanglement in order to create *quantum circuits* [27]. In contrast to Classical Computing, inspecting the "state" of a qubit (aka *measurement* of a qubit) destroys its coherence and irrevocably disturbs the superposition state the qubit was in [25]. It is currently essential that these concepts are understood by a software engineer when creating quantum circuits.

Like their digital counterparts, quantum circuits encode the solution to a *very specific problem*. As an example, consider the quantum circuit presented by Jha *et al.* [17] that computes all possible solutions to the 4 Queens puzzle, that is, placing 4 queens on a 4-by-4 chess board so that no two queens can attack each other [2]. This circuit has to be substantially modified to solve a Queens puzzle for a different number n of queens/sizes of chessboards.

In order to address the scalability issue of the 4-queens solver, we opted for a quantum code generation approach: using techniques from Classical Computing, we developed a generator that, taking the number of queens/sizes of chessboards n as input, produced the QASM code [9] for the resulting quantum circuit that we then could execute on existing quantum simulators. The lessons learnt during the development of the generator not only enhanced our understanding of the basic quantum technology building blocks, but also highlighted the need for higher-level abstractions to make quantum programming more accessible.

To give a concrete example: the quantum circuit presented by Jha *et al.* [17] relies on 4 qubits being entangled into a $|W\rangle 4$ state [6]. Such an entanglement, although not uncommon in quantum circuits, has to be explicitly constructed using a composition of quantum logic gates (Hadamard, Toffoli, CNOT and Not gates [15] – see Figure 12). As the solution requires 4 such $|W\rangle 4$ states, the same composition of quantum logic gates has to be repeated 4 times. From a programming perspective, it would be much easier if there

ACSW '22, February 14–18, 2022, Virtual

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

was an abstraction that allowed a developer to simply specify the need for a $|W\rangle$ n state, especially as the generalization of a $|W\rangle$ state to n entangled qubits is not straightforward.

Our main motivation is to investigate that as classical programming has raised to a level where developers no longer need to understand digital gates, the same level of abstraction should be made available for quantum software development. We do this by describing our experience from solving the 4 Queens puzzle with quantum code, and discussing the fundamentals of Quantum Computing at a level that software engineers may understand more easily than a quantum physics level that most documentation is written in. This paper investigates issues with creating quantum programs by discussing challenges and limitations faced by new developers to the area. Our findings may also lead themselves for future work to help simplify quantum computing for software engineers and integrate quantum Software Engineering into standard Software Engineering practices.

The remainder of this paper is organized as follows: the fundamentals of Quantum computing technology for software engineers is reviewed in Section 2. Section 3 discusses the breakdown of the *4*-queens solutions and the development of scaling quantum code. Section 4 introduces the idea of a scalable quantum code generator. Section 5 has some discussion on the topic of simplification of quantum coding. Finally, Section 6 presents some conclusions and future work.

2 QUANTUM COMPUTING TECHNOLOGY

Quantum Computing is the use of quantum mechanical phenomena such as superposition and entanglement to perform computations [23]. The fundamentals of QC and how quantum gates use quantum phenomena are described in Section 2.1. This includes a description of qubits and gates and how superposition and entanglement are used to formulate outputs. In Section 2.3 the current state of quantum computers and simulators and the limitations this has on quantum software development is discussed.

2.1 Fundamentals

As a new way of computing, Quantum Computing was first conceptualised in the early 1980's [3] as Feynman suggested that quantum systems could not be represented by classical computers [11]. The physical realisation of the idea did not occur until 1988 [16]. In 1997, Shor described an algorithm for factoring integers in polynomial time [28] which ignited massive interest in QC's potential.

A Quantum Bit is the fundamental unit of information within a quantum computer, similar to what a binary digit (Bit) is to a classical system [27]. Through superposition, a qubit is a two-state system and can be in the state of $|0\rangle$ and $|1\rangle$ simultaneously. Upon measurement or observation, the state collapses into one state and will only ever be observed as either $|0\rangle$ or $|1\rangle$ [19], respectively.

A software developer may not be concerned with the physics of superposition. However, they need to understand a qubit in superposition has an equal probability of being $|0\rangle$ or $|1\rangle$ when a circuit is executed (or *shot*). Depending on the size of the output, a circuit needs to execute thousands of shots to increase the chance of all combinations being calculated [30].

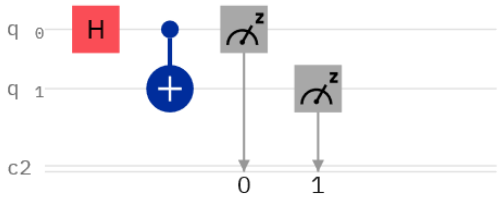

Figure 1: the 2 qubit Bell State – the simplest example of quantum entanglement on a circuit.

In QC, a shot is a user defined value that refers to how many executions of a circuit are run before termination. Because the measurement of a qubit in a superposition state is random, the outcome is sometimes $|0\rangle$, sometimes $|1\rangle$. The measurement must be repeated multiple times to determine the likelihood that a qubit is in a particular state.[1] Even though the quantum phenomena of superposition and entanglement are more concerned with the physical behaviour of qubits, Quantum software engineers will need to develop their understanding in these phenomena and how code is written to take advantage of these phenomena.

The physical makeup of a qubit should not be much of a concern for writing code and can be represented in many ways just like a digital bit. Where the magnetisation of a hard disk or voltage in a wire can determine a 0 or 1 for a Bit, some of the ways a qubit can represent $|0\rangle$ and $|1\rangle$ are through the spin of electrons [14] or the polarisation of photons [10].

For example, a 3 qubit register can represent all $2^3 = 8$ combinations of possible values at once. Quantum gates are hard coded in a quantum circuit to control the outputs for a desired result. This makes it difficult for people who write classical software to develop code for quantum computing as it requires a developer to code each entanglement at an assembly language level.

Gates used in quantum circuits can be categorised into quantum, classical, phase and non-unitary. The Hadamard (H) quantum gate creates superposition and is the fundamental quantum gate for most quantum algorithms.[1] A more detailed introduction of quantum gates is given in Section 2.2.

Figure 1 shows a simple example of quantum entanglement with a circuit using IBM Qiskit.[1] The x-axis represents the order of gates in the circuit and is generally read from left to right. The y-axis show how many qubits (two) are in this circuit followed by the c2 line which is interface between the qubits and classical Bits. In IBM's composer red gates are the quantum gates, blue gates are classical and grey are the non-unitary operators.

A H(adamard) gate on q0 is entangled with a CNOT gate on q1 and both qubits are measured to provide an output. The output of this circuit is refereed to as the Bell state, which is an even distribution of 2 outputs being $|00\rangle$ and $|11\rangle$, respectively.

A quantum software developer does not need to understand how qubits are physically entangled. However, how gates exploit entanglement to create quantum circuits that provide output should be understood.

[1] https://quantum-computing.ibm.com/composer/docs/iqx/operations_glossary

Quantum computing is analogous to when developers needed to understand the behaviour of binary gates and how they can be composed to solve computational problems. A higher level of abstraction in Quantum Software Engineering is required to provide QC with its ENIAC moment. ENIAC which was the first operational programmable computer was the transition between mechanical and electromechanical calculators and computers. This development allowed for digital computers to become commercially available [8].

In 2018, John Preskill described the current state of QC as the Noisy Intermediate Scale Quantum computing (NISQ) era [26]. This era is the intermediate step between theorising QC concepts and developing hardware that is superior to the most advanced classical computing technology. The NISQ era is concerned with experimenting through the development of small working examples and simulations that will form the knowledge base of QC.

The biggest hurdle that has to be overcome is the large amounts of noise and errors generated as more qubits are added to a quantum computer. Quantum noise can effect circuits and produce results that do not occur in simulation [22].

As quantum computers can only output sequences of binary values (1 value per qubit) there will always be a hybrid relationship with classical computing to process results. Measurement gates are required in a quantum circuit to measure qubit states after a shot and return the values to a classical computer.

While operational quantum technology is readily available to the general public current simulators can still be used to learn quantum Software Engineering. Researching the issues that will be faced by software engineers wanting to learn quantum software development will go some way for developing technology that can be deployed on real world problems.

QC technology has the potential to speed up the discovery of optimal solutions of NP-hard problems in exponential or quadratic time [13, 28]. However, it is debated if QC will ever be able to solve NP-hard problems in polynomial time [1, 24]. NP-hard problems are those that can be found in cryptography, data mining, vehicle routing, scheduling/planning, and configuration management. But for now Classical Computing and its probabilistic nature is still the best way for finding optimal solutions, even though not all possibilities would be calculated in a feasible time.

2.2 Quantum Gates

In QC, a quantum logic gate (or simply quantum gate) is a basic quantum circuit operating on a small number of qubits. Quantum gates are the building blocks of quantum circuits similar to how classical logic gates form the basis for conventional digital circuits. Unlike many classical logic gates, quantum logic gates are *reversible*, that is, the output of a logic gate can be use to restore the corresponding input [15].

It is possible to perform classical computing using only reversible gates. For example, the Toffoli gate (see below) can implement all Boolean functions, often at the cost of having to use auxiliary qubits. The Toffoli gate has a direct quantum equivalent, showing that quantum circuits can perform all operations performed by classical digital circuits.

Described below are some of the basic quantum gates that a developer will need to understand to build quantum circuits and are used throughout examples used in this paper. This list is not exhaustive and an introduction of more complex gates has been omitted due to space constraints.

The **Hadamard** (H) gate is the most basic of the quantum gates that exploit quantum mechanics. It puts a qubit into superposition with an equal probability of returning a $|0\rangle$ or $|1\rangle$. Exploiting this ability is fundamental to most quantum algorithms[1]. This gate ensures that both $|0\rangle$ and $|1\rangle$ can be tested over a number of shots. Entangled with other H gates increases the combinations of outputs that can be returned.

The **NOT/CNOT** gates are classic gates that are used within a quantum circuit. The state of a qubit will be flipped from $|0\rangle$ to $|1\rangle$ and vice versa. The controlled-NOT (CNOT) gate means that 2 qubits are entangled and the state of control qubit will change the target qubit. Even though the control qubit acts as an input to the target qubit its value continues along the circuit and can be further used by other gates.

Toffoli (or CCNOT) gates are used extensively through the n-queens solution. They are also known as the double controlled-NOT gate (CCX) and have three inputs made of two control qubits and one target. It only applies a NOT to the target when both controls are in an excited state ($|1\rangle$.) As Toffoli gates are universal they can build systems that can perform Boolean functions.

To rotate a qubit state on the y axis by a given angle an $\mathbf{R}_y$ gate is used. The R_y gate changes the probability of measuring the qubit as either 0 or 1. $\mathbf{R}_x$ and $\mathbf{R}_z$ gates do similar rotations for their respective axes. These gates provide simple rotations and do not introduce complex amplitudes.

To convert quantum output into a vector of readable, binary sequences, **Measurement** gates are placed in a circuit on each qubit. After each shot you get a sequence of '0' or '1' where each 0 and 1 represents the measurement of a specific qubit. These are essentially the interface between quantum and classical computing and the sequences can then be interpreted on a classical computer.

2.3 Quantum Simulators

To investigate how to create quantum software, we have familiarised ourselves with quantum software simulators as they are easily accessible and can run code that closely resemble quantum circuits. The complexities of the issues with operational quantum computers are outside the scope of this research.

Quantum software simulators differ from true quantum computers by simulating quantum phenomena on classical hardware. The benefits of simulators is quantum code can be trialled on online simulators or personal computers. However, simulators are limited by the amount of RAM available and, therefore, only circuits for solving "small" problems can be executed. This is because the value of each qubit and gate is stored in a Hilbert space and the values within a quantum computer are represented by the tensor product of all the components' Hilbert spaces [7]. As more qubits and quantum gates are added, an exponentially increasing amount of RAM is needed.

There are a number of different QC software simulators[2] where developers can build and test formulations on a small scale. Simulators from IBM, Microsoft and Silq were trialed for our research

[2] https://quantiki.org/wiki/list-qc-simulators/

Qubits	5	10	20	30	40	50
RAM	512B	16KB	16MB	16GB	16TB	16PB

Table 1: Quantum Simulation RAM Requirements for increasing number of qubits.

to understand their capabilities. IBM's Quantum Experience is the most accessible and capable as it provides a downloadable software package and access to an online portal to run larger quantum circuits on their more powerful quantum simulators. Circuits and results are also more easily visualised on the online portal.

IBM's simulators with larger numbers of qubits have restrictions on the types of gates used and a 10,000 second time limit on executions. Due to the properties of the Hilbert space in quantum systems, adding qubits to a quantum simulator running on classical hardware exponentially increases (16×2^n) the memory required [18]. Table 1 shows how much RAM is needed for the simulation of circuits with certain numbers of qubits.

This does not mean that actual quantum computers will need that much RAM, it is only a calculation for simulating on classical hardware. These calculations explain why we could not run anything greater than a 4x4 n-queens puzzle on one circuit as discussed in Section 4. With these limitations we experimented with quantum batch processing. This involved keeping the core "chessboard" qubits and $|W\rangle$ states intact for each batch, then cycling through each of the ancillary qubit checks. Keeping the number of qubits low allowed us to process the quantum circcuit within the time limit. Such an approach may not be universally applicable, though.

Microsoft Q# is available to download as an extension for MS Studio Code and Python.[3] The code is similar to C# and F# and lets users experiment with quantum phenomena on a small scale. Microsoft provides some libraries for experimentation. Although the code provides higher-level abstractions compared to IBM's Quantum Assembly Language (QASM), a user still has to familiarise themselves with qubits and entanglement. Microsoft also provide online simulators through Azure Quantum but are only free for the first hour of computing time.

Silq is a quantum programming language developed by ETH in Zurich that is similar to Q#. It can either be accessed via MS Studio Code or Silq's VSCodium which is a modified version of VSCode.[4] Their tool does simplify some coding compared to Q#, however, it still requires the user to understand qubits and entanglement. No simulator accessible online is available for Silq and hence all code must be run from a local computer [4].

3 WRITING SCALABLE QUANTUM CODE

To investigate the current level of abstraction of quantum software development, we investigated the quantum solution to the *n*-queens problem as described by Jha *et al.* [17]. The *n*-queens problem was chosen because its a well known classic backtracking coding exercise with many published solutions. Backtracking is useful for finding optimal results for combinatorial optimization problems. However, for large problems it is not feasible to test all combinations [12]. QC theory suggests it could run all combinations in a much shorter time frame. By expanding on Jha *et al.*'s code, we developed a quantum code generator that scales the quantum code for different values of *n*. In this section, we describe the 4 Queens solver, how the code for the scaling generator was created, and the lessons learned from developing the solution.

3.1 Problem Description

The *n*-queens puzzle places *n* chess queens on a *n*-by-*n* chessboard so that no two queens threaten each other. This is a well known computing problem that many Software Engineering students would have solved during their studies. Backtracking or branch-and-bound can be used to solve the problem classically, but to solve this problem with QC requires a different approach.

While many classical solutions have been developed, only a few quantum solutions have been documented. Besides the solution presented by Jha *et al.* [17], Torggler *et al.* [29] discuss a solution on the physical level using atoms in an optical lattice altered by lasers but is purely concerned with quantum mechanics.

Where classic methods step through each combination and criteria, the quantum method requires the code to find all the answers *at once*. Solving the *n*-queens puzzle with QC is about decreasing the search space, exhibits how the correct answer will not be the only output, and that quantum noise and incorrect answers will need to be dealt with via classical means.

Jha *et al.*'s approach to the problem was to have one qubit for each square on a chessboard and a number of ancillary qubits to indicate if the necessary criteria checks passed or failed. The 3 criteria to satisfy for this approach:

- there is always one queen in each row;
- 2 or more queens are not in the same row;
- no 2 queens face each other diagonally.

The aim of the experiment was to take the existing solution and generate the QASM code for an arbitrary value of *n*. On investigation Jha *et al.*'s 4 queens solution could not be scaled directly and required a redesign of the $|W\rangle$ state. Once we had a scalable $|W\rangle$ state, the remainder of the solution worked as described by Jha *et al.*.

3.2 Solution Description

As the number of qubits are fixed for each quantum circuit, we needed to determine the amount of qubits to be used in advance. After this we can proceed how the circuit is entangled. The solution can be broken down into 2 different sections, the chessboard and ancillary qubits. The representation of the chessboard squares is equal to n^2. The number of ancillary qubits Q_A is calculated with equation (1.)

$$Q_A = \frac{n^2}{2} + \frac{n}{2} - 1 \tag{1}$$

When $n = 4$ there are 16 qubits for each square on the board and 9 ancillary qubits are required. The first 4 qubits, therefore, represent the first row of a 4x4 board and the pattern is repeated with qubits 5 to 8, 9 to 12, and 13 to 16, respectively. The 9 ancillary

[3] https://azure.microsoft.com/en-us/resources/development-kit/quantum-computing/
[4] https://silq.ethz.ch/

qubits for $n = 4$ can be broken down to $n - 1 = 3$ for the column check indicators and $\frac{n^2-n}{2} = 6$ for the diagonal indicators.

Once the number of qubits has been determined for the circuit, it is time to place the quantum gates that represent the queens and perform the criteria checks. The representation of the queens use an n qubit-$|W\rangle$ state.

A $|W\rangle$ state is an entangled quantum state of three or more qubits which has a distribution of 1 qubit being in the excited state of $|1\rangle$ and the remainder in the ground state of $|0\rangle$ [6]. Figure 2 shows the circuit for a singular $|W\rangle 4$ state. An R_y gate, 2 controlled Hadamards, 3 controlled-NOT gates and a singular NOT gate create the entanglement for the $|W\rangle 4$ state. At the end there are the grey measurement gates which detect the state of the qubit and map it to a classical bit.

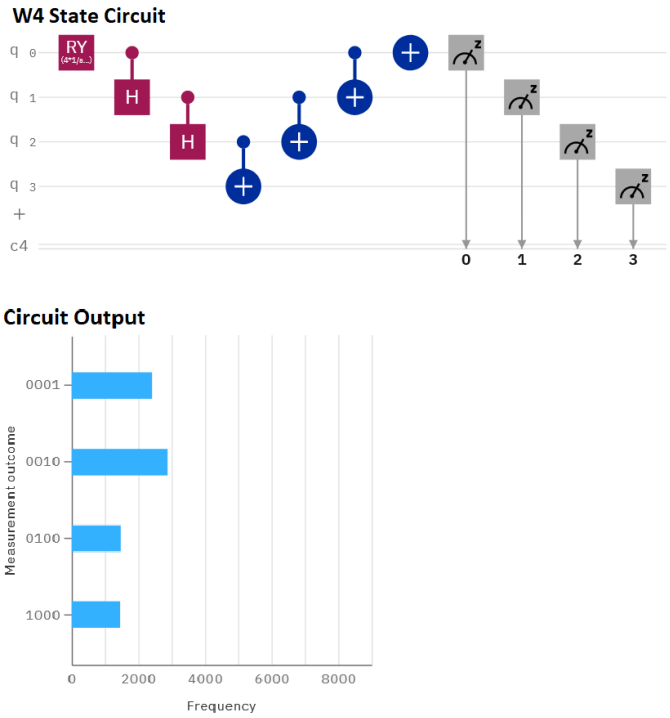

Figure 2: W4 Circuit and Sample Output Generated in IBM Quantum Composer.

The output in the same figure shows the distribution of results after the circuit has performed 8,192 shots. The 4 outcomes are the only outputs ever returned by the circuit. The output of this circuit can be used to represent a queen appearing on one square of a row for one shot.

The $|W\rangle 4$ state is then repeated for each group of 4 qubits (Figure 3) to get the possibility of every combination of 1 queen always being in each row. That ensures there is never less than or more than 4 queens on the board. Without using a $|W\rangle$ state the algorithm would output every possible combination of 0 and 1; with $n = 4$ that is 65,536 combinations. The $|W\rangle$ 4 State brings this down to a manageable 256 combinations, making the algorithm better than brute force.

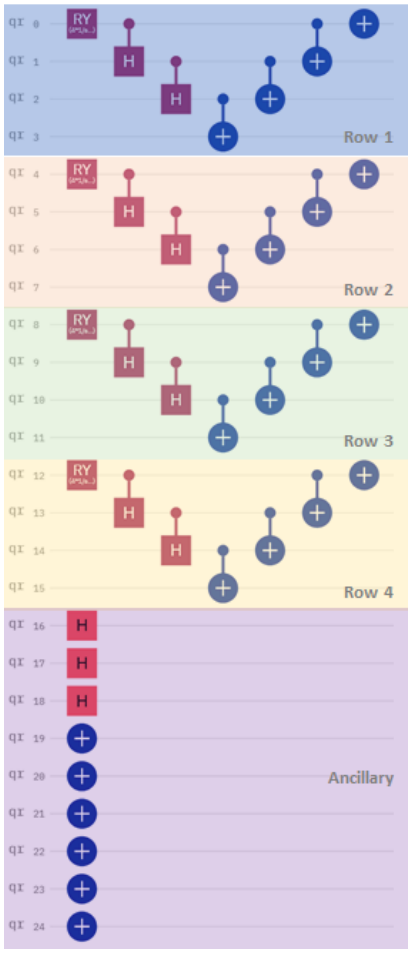

Figure 3: Start of 4 Queens Circuit. Showing 4 x $|W\rangle$ states and Ancillary qubit Groups.

Figure 4 shows the squares in the first column entangled with q16 using controlled U gates to form part of the column check criteria. The U gates placed like this will return a $|0\rangle$ in q16 if there are no or 2 or more queens in a column, and in turn a $|1\rangle$ if there is only one queen the whole column.

There is no need to check the nth column as the $|W\rangle$ state ensures there is always 1 queen in each row and if the column criteria is met then it is always true that the nth queen will be in the correct position.

The final criteria to meet is that no 2 queens meet each other diagonally. Figure 6 shows the example for 1 diagonal check entangled with Toffoli gates to q19. The first qubit for the diagonal indicators is a classic NOT gate (see Figure 3) meaning that they change the ground state of $|0\rangle$ to only ever equal $|1\rangle$ at that point of the circuit.

Toffoli gates return $|0\rangle$ if the first two control inputs equal $|1\rangle$. Therefore, if there were a queen on squares 0 and 5 then the Toffoli gate would change q19 to $|0\rangle$ and if only $|1\rangle$ were present q19 would remain as $|1\rangle$.

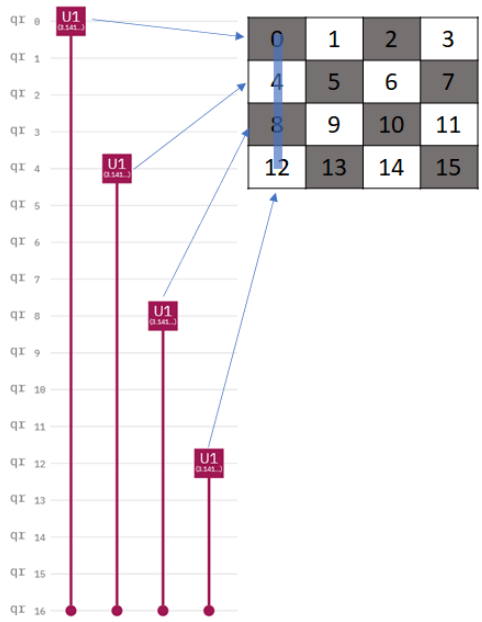

Figure 4: Example Column Check

```
# 4 Queens Column Check for 1 Column

# Record Positions of Queens of Squares
If(sum(Q0,Q4,Q8,Q12) != 1){
  # Set Ancillary Qubit
  Then Q16 = 0
  Else Q16 = 1
}
```

Figure 5: Pseudo Code for Column Check

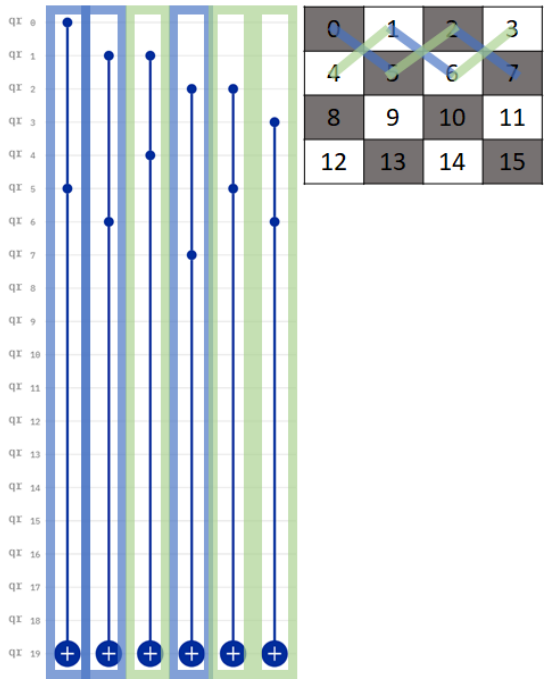

Figure 6: Example Diagonal Check.

```
# 4 Queens Diagonal Check for 1 Group

# Record Positions of Queens of Squares
If(sum(Q0,Q15) OR sum(Q8,Q12) > 1){
  # Set Ancillary Qubit
  Then Q21 = 0
  Else Q21 = 1
}
```

Figure 7: Pseudo Code for Diagonal Check.

The diagonal fragments works because only 2 queens are ever checked in a shot. Therefore, it can only ever fail once. When combined with the $|W\rangle$ state there will never be a combination of queens that would make it fail twice or more.

The design of this circuit will only ever output n^n measurement outcomes, which represent all the combinations of only one queen appearing in each row in every position of that row. When the circuit is run for $n = 4$, we will possibly get 256 different outputs. If we only set the number of shots to execute at around 256, due to the random nature of qubits not every combination will have the chance to be tested in the circuit. The limit of IBM Quantum Experience is 8,192 and this increases the possibility that each combination gets run through more than once.

To find the sequences of 0's and 1's that are correct solutions, each sequence needs to be read on a classical computer. The correct sequences are those that have all nine ancillary bits in the "excited state."

QC output size is determined by the amount of qubits in a circuit. If we have 25 qubits in a circuit then the sequence of 0's and 1's returned is precisely 25. The order of the output matches the position of the qubit. So q0 will be the first character in the sequence and q25 will be the last.

Figure 8 shows an example of a correct and incorrect solution found among the 256 outputs from the n-queens circuit. The first 16 characters represent the queens on the board and the final 9 show whether each criteria check was successful or not.

In the correct solution, all ancillary Bits equal to 1 and as illustrated, we can see that the criteria has been met. For the incorrect solution we can see the $|W\rangle$ states working by only having 4 queens in the board and 1 on each row. However, there is a queen in position 2 and 6 in turn this causes qubits q18 and q22 to return a 0 value.

Good circuit design is critical to limit the amount of outputs created and so the correct solutions can be found by classical means more easily. As some of the gates used to create a $|W\rangle$ state can cause an imbalance in values, it is important to understand how the parameters are tuned to achieve an even distribution of outputs. When there are finite amount of shots to run before termination it is possible for some correct solutions to be completely missed.

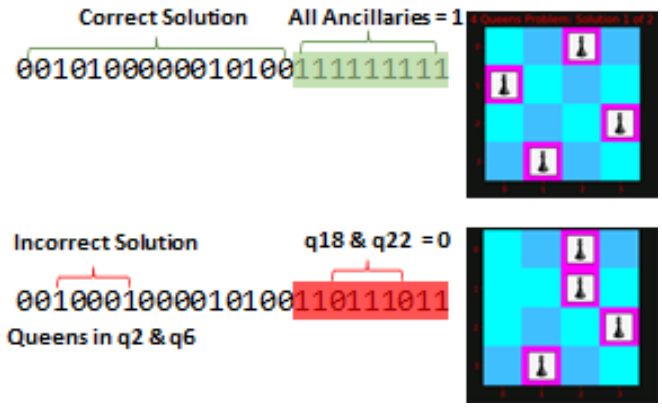

Figure 8: Example output showing a correct and incorrect solution

4 QUANTUM CODE GENERATION

Section 3 described the process of creating scalable quantum solutions from the point of view of the quantum code programmer. In particular, it highlighted the extensive skills required from a quantum code programmer. The programmer needs to know i) the complete details of the problem they are trying to solve, ii) how to write quantum code, iii) how to write solutions to their problem in quantum code, and iv) depending on the use case, how to write scalable quantum solutions. For non-quantum programming the developer generally doesn't have to worry about a lot of this detail as there is a lot of tool support to generate code. A desirable aim of software engineering for quantum computing should be to raise the level of abstraction to enable more rapid quantum code creation. One method of raising the level of abstraction to simplify the Software Engineering process for Quantum programming is through Automatic code generation using design patterns [5].

To be able to automatically generate quantum code a code generator needs to be created. This is a challenging problem, especially in quantum computing, so the focus of this paper is to create a quantum code generator for the n-queens problem as proof-of-content that this is viable. Applying this approach more generally will require domain knowledge for each specific problem. Through the development of a quantum n-queens solver this paper investigates if there should be similar automatic code generation tools available for all quantum computing problems.

Manually writing the code for each qubit and gate is time consuming and as the number of qubits in a system increases the chances of writing erroneous code also increases. Another issue with quantum code is each problem is solved with its own particular fixed circuit. Not only does the code need to change for each problem; even scaling a circuit for a problem needs major changes. Raising the level of abstraction from dealing with individual quantum gates and the sheer size required to achieve quantum supremacy is an important area of focus for the QC community.

The approach used in this paper was to create a Python program that will generate quantum code compatible for IBM's Qiskit toolkit specifically for the n-Queens problem. This allows the user to enter a value for n at the start, have the QASM code written to a file, execute the circuit then process and visualise the successful results. The aim of this is to allow a user with knowledge of the n-Queens problem to generate quantum code in the specific format to be executed on IBM Qiskit compatible simulators and hardware.

```
# Define number of queens
nQ = input('Integer greater than 1')

# Chessboard Size
nS = nQ**2

# Total Qubits Required
nQbs = int((3/2) *nQ**2 + nQ/2 -1)

# Diagonal Ancillaries
daQbs = int(((nQ**2)-nQ)/2)

# Column Ancillaries
colQbs = nQbs - daQbs - nS

# Build lists of combinations
# Hadamards required for preparation of W-states
# Define left and right column edge squares
whLst = squares in right and left columns

# Initialize the ancillas required for
#diagonal checks
dcwLst = daQbs + whLst

# Circuit for performing column check
#using ancillary qubits
colQbsChk = colQbs + column_position

# Create Table for all diagonal combinations
#and assigned ancillary qubit
diagLst = Q1 to Q2 positions + daQbs

#Write Code to QASM File
to_file('Nqueens.QASM', \
[whLst,dcwLst,colQbsChk,diagLst])
```

Figure 9: Pseudocode for the n-Queens quantum code generator

Our experience with creating quantum code for the 4-Queens problem, led the creation of a quantum code generator to solve the puzzle for any value of $n>2$. This quantum code generator is publicly available [5]. Figure 9 illustrates pseudocode for the quantum code generator.

To create this involved an understanding how the QASM code for the 4 Queens problem was constructed and how it should look for different values of n. Figures 10 and 11 illustrate the generated n-queens quantum code for 4-queens and 5-queens respectively.

As with Classical Computing, the complexity of the computation can be observed in terms of the resources it uses. For quantum

[5] https://www.qcse.net/n-queens-solver

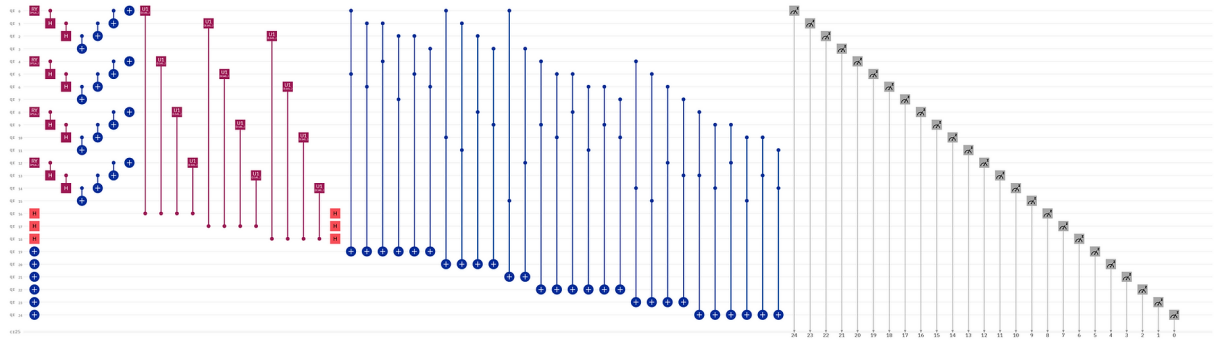

Figure 10: Visualisation of a Generated 4-Queens solver Quantum Circuit

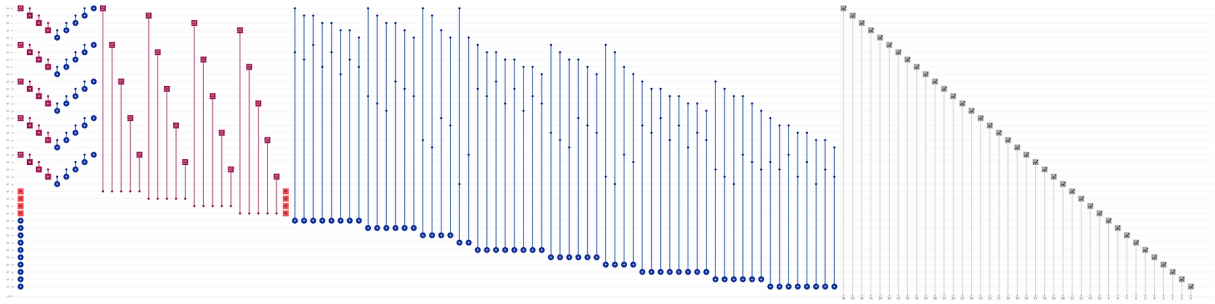

Figure 11: Visualisation of a Generated 5-Queens solver Quantum Circuit

computing code, this is generally measured in qubits. The total number of qubits for a circuit to solve for or any value is represented in the equation 2. As stated in Section 3, n^2 are reserved for the squares on the board, $n-1$ for the column criteria indicators and the remainder $\frac{n^2-n}{2}$ for the diagonal criteria indicators.

$$Q_t = \frac{3}{2}n^2 + \frac{n}{2} - 1 \tag{2}$$

The placement of the queens on the board requires a $|W\rangle$ state that can scale to ensure the right number of queens and to make sure each is placed somewhere along 1 row only, therefore decreasing the search space. Figure 12 are different configurations of $|W\rangle$ states. On the very right is Jha's $|W\rangle$ 4 state that works for n=4, however this doesn't easily scale for other values of n. There was more success with the $|W\rangle$ state in IBM's documentation [6] by adding extra controlled Hadamards and CNOT gates for each increment of n. Further improvements of this circuit can be made by balancing the distribution of results.

The column checks increase linearly with n, all but the last column gets checked. A controlled U gate is placed in each square of a particular column then entangled with an ancillary qubit. The diagonal checks are more complex with each combination where 2 queens could face each other diagonally needing to be checked. There is a polynomial increase of Toffoli gates to use as n increases.

Figure 6 illustrates a group of diagonal checks for one ancillary. These are the squares on the first row checking against those diagonals 1 square away. Figure 13 are the diagonal checks for those 2 squares away. This will check to see if there is a queen in squares 0 and 10, but it is not checking for square 5 which is in between.

[6] https://quantum-computing.ibm.com/composer/docs/iqx/example-circuits/w-state

Figure 12: Left $|W\rangle$4 State, Centre $|W\rangle$5 State, Right Jha's $|W\rangle$4 State

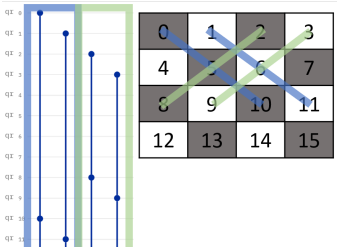

Figure 13: Diagonal Check

When the code generator worked as expected for n=4 the focus was moved onto trialling the 5 queens puzzle on both a PC and IBM's simulators. This is where limitations of Qiskit on both platforms become obvious. Due to these limitations, the 5 queens solution has to be performed in batches.

n	Qubits	Gates	Diagonal	Outputs	Total Solutions
3	14	54	10	27	0
4	25	105	28	256	2
5	39	182	60	3,125	10
8	99	569	280	16,777,216	92
27	1,106	16,044	12,402	4.43×10^{38}	2.349×10^{17}

Table 2: N Queens Circuit Statistics.

Solving the 4x4 puzzle is trivial for Classical Computing but is at the upper limits for current publicly available QC simulators. Its important to note that adding qubits to a quantum simulator running on classical hardware exponentially increases the memory required [18]. The 4x4 puzzle requires 25 qubits to solve and requires about 512MB RAM to simulate, whereas a 5x5 puzzle requires 39 qubits and approximately 8TB RAM to simulate, making it out of reach of most researchers. When trying to run the 5 queens code on the IBM simulator a a timeout error was received after 10,000 seconds.

As the probabilistic nature of quantum shots is random, throughout the run of the generated batched circuits not all combinations of the $|W\rangle$ state had the chance to be executed. For n=5 that is 3,125 and the limit of shots on IBM's servers is 8,192. There are also differences between the batches. Not all $|W\rangle$ state combinations occur in each batch and the combinations observed in each batch were also different. This means there was a decreased amount of $|W\rangle$ state combinations that had managed to complete all ancillary checks.

To ensure we were to observe all $|W\rangle$ state combinations, each batch would need to be run multiple times and the observations merged. Batch processing is not ideal, but it suggests that it could be a process used when the limits of a quantum simulator cannot process a circuit all at once. The classical computation costs would be greater as it is needed to process results while it waits for the results of each batch to run; but its possible to be able to calculate the more complex parts of a problem much quicker with QC.

Table 2 shows the statistics for the QASM code for selected values of n. The number of qubits and gates give an indication to how the lines of code need to increase for different values of n and is not just a case of reparametrisation of the a circuit. The diagonal column represents every way 2 queens could meet and is equal to the amount of Toffoli gates required. The outputs column shows the total number of combinations that each circuit generates if given enough "shots." This is the output a classical computer will have to sort through. Total solutions indicated the number correct solutions available for that value.

The selected values were chosen to highlight quantum computing processes and outputs and the difficulties with simulating the n-queens solution. n=3 was chosen because there is no correct solution but 27 output sequences will still be returned from the circuit. n=4 because its the first problem attempted and it was the largest circuit simulatable. n=5 is the next step up and where issues with simulation occurred. n=8 as it highlights what is required to solve the puzzle on a standard chessboard.

Thanks to the Q27 Project in 2016 [7] which was a massive parallel computation project that enumerated and counted all the valid solutions of the 27-Queens Puzzle.[8] We know how many total solutions are possible for up to *n=27*. The project took over a year to run to find all solutions and would be difficult to perform a quantum simulation of this size because of the inconceivable amounts of RAM required to produce results.

5 DISCUSSION

The current level of abstraction for writing quantum code takes us back to the 1940s-50s where computers needed to be hardwired to solve one specific task and changed again to scale or solve a completely different problem. Quantum programs now are essentially hardwired for a specific problem and require changes to scale or solve a different problem. When creating the quantum n-queens code generator we came to the conclusion that the level of understanding required by a quantum software engineer is much more complex than writing classical code.

As the number of qubits and gates required to solve a problem increases it will become impractical for a developer to manually write thousands of lines without error. Having a coding tool that could generate the required gates and entanglements from higher-level descriptions would provide a level of simplification that would allow an engineer to write code more efficiently. For example a developer should not have to create an evenly distributed $|W\rangle$ state with qubits and entanglement from scratch every time. A higher level quantum software development tool should deal with the QASM in the background.

Debugging tools for quantum computers is an area for further research. Due to the measurement issue it is difficult to debug a quantum circuit. It is possible to to debug on a simulator. However, if the problem is too big for a simulator more efficient simulators and debugging tools and techniques will need to be developed.

A higher level quantum development tool will also be necessary when the numbers of qubits in a computer increase to sizes that would be too time consuming to type individually while simultaneously trying to determine a program's outputs. Quantum program debugging is also an area that is underdeveloped and still being discussed. A higher level tool should provide some solutions for debugging a circuit that is far too large for a quantum simulator to inspect.

The n-queens solution could possibly be created with not so many ancillary qubits needed to confirm a result. Decreasing the search of correct solutions will be essential when dealing with potentially quadrillions of outputs.

Batch processing of quantum programs should be considered to process circuits which require more qubits than are available on a quantum device. When creating a solution for the 5 queens problem we found that IBM's simulators were not able to process the number of qubits and gates within their set time limit. We were able to find some correct solutions by running the circuit in batches and using classical means to concatenate the outputs. It is not ideal to run in batches but it can overcome some limitations in quantum hardware.

[7] https://github.com/preusser/q27.2016.TheQ27Project
[8] https://en.wikipedia.org/wiki/Eight_queens_puzzle#Counting_solutions

6 CONCLUSIONS AND FUTURE WORK

Software Engineering for quantum computing requires a vastly different approach to current classical Software Engineering. Using an existing solution developed for the 4 queens puzzle and its generalised scaling technique. We built a quantum code generator capable of writing and executing for values where $n>2$. During the development of the code we were able to identify some of the areas of quantum computing that need to be addressed for quantum Software Engineering to be accessible by more people.

If quantum computing hardware achieves its potential power, quantum Software Engineering cannot remain at the level of abstraction it is currently at. The quantum coding tools (IBM Qiskit, Q# and Silq) we trialled in our research require an engineer to code each qubit and its entanglements. As these tools require an understanding of complex quantum concepts they could prohibit the amount of people that can create quantum code and those that are capable would need to spend a lot of time potentially coding millions of individual qubits and gates.

For the foreseeable future there will be a need for quantum simulators as getting access time to quantum computers will be limited. Simulators are a great way to test out coding concepts before scaling them to more qubits on a real quantum computer.

As we discovered trying to solve the relatively small 5 queens puzzle on a simulator it would take 8TB RAM to solve on one circuit. Because of the limitations of current quantum simulators extending the number of qubits and gates mean that non-trivial problems cannot be simulated. Quantum simulators will need to improve on current capabilities for circuits to be tested more thoroughly.

When millions of qubits are required to solve a problem like the much smaller 27 queens problem, even 27^2 qubits with gates restricting outcomes will still output 4.43×10^{38} strings of data and Classical Computing will need to sort through those to find correct solutions.

Extending that logic to millions of qubits means efficient quantum algorithm development will be essential. Also any imbalances in the quantum code could see combinations not run, which may hinder finding all correct solutions.

Even though currently QC is still in its infancy and not yet capable of out performing Classical Computing. Researching quantum Software Engineering gaps and building tools and processes to make the field accessible for more people should be of high importance. When the time comes for commercially available quantum computers; having a well researched body of knowledge in place will provide those utilising the new technology an advantage.

REFERENCES

[1] Scott Aaronson. 2008. The limits of quantum. *Scientific American* 298, 3 (2008), 62–69.

[2] Jordan Bell and Brett Stevens. 2009. A survey of known results and research areas for n-queens. *Discrete Mathematics* 309, 1 (Jan. 2009), 1–31.

[3] Paul Benioff. 1980. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of Statistical Physics* 22, 5 (1980), 563–591. https://doi.org/10.1007/BF01011339

[4] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. [n. d.]. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. Association for Computing Machinery, 286–300.

[5] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. 1996. Automatic code generation from design patterns. *IBM Systems Journal* 35, 2 (1996), 151–171.

[6] Adán Cabello. 2002. Bell's theorem with and without inequalities for the three-qubit Greenberger-Horne-Zeilinger and W states. *Physical Review A* 65 (March 2002), 032108. Issue 3. https://doi.org/10.1103/PhysRevA.65.032108

[7] Gabriele Carcassi, Lorenzo Maccone, and Christine A Aidala. 2021. Four Postulates of Quantum Mechanics Are Three. *Physical Review Letters* 126, 11 (2021).

[8] James W. Cortada. 2006. The ENIAC's influence on business computing, 1940s-1950s. *IEEE Annals of the History of Computing* 28, 2 (2006), 26–28.

[9] Andrew W. Cross, Lev Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. *arXiv: Quantum Physics* (2017).

[10] Daniel Barbosa de Brito, José Cláudio Nascimento, and Rubens Viana Ramos. 2008. Quantum Communication With Polarization-Encoded Qubit Using Quantum Error Correction. *IEEE Journal of Quantum Electronics* 44, 2 (2008), 113–118.

[11] Richard P Feynman. 1981. Simulating physics with computers. *International Journal of Theoretical Physics* 21 (1981), 474. Issue 6/7. https://doi.org/10.1007/BF02650179

[12] Martin Grotschel and László Lovász. 1995. Combinatorial optimization. *Handbook of combinatorics* 2, 1541–1597 (1995), 4.

[13] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, 212–219.

[14] Ronald Hanson, J. M. Elzerman, Laurens H. Willems van Beveren, Lieven M. K. Vandersypen, and Leo P. Kouwenhoven. 2004. Electron spin qubits in quantum dots. In *IEDM Technical Digest. IEEE International Electron Devices Meeting*. 533–536.

[15] Jack D. Hidary. 2019. *Quantum Computing: An Applied Approach*. Springer.

[16] Kazuhiro Igeta and Yoshihisa Yamamoto. [n. d.]. Quantum mechanical computers with single atom and photon fields. In *International Conference on Quantum Electronics (OSA Technical Digest)*, H. Yajima T. Inaba and T. Ikegami (Eds.). Optical Society of America, TuI4. http://www.osapublishing.org/abstract.cfm?URI=IQEC-1988-TuI4

[17] Rounak Jha, Debaiudh Das, Avinash Dash, Sandhya Jayaraman, Bikash K Behera, and Prasanta K J arXiv preprint arXiv:.10221 Panigrahi. 2018. A novel quantum N-Queens solver algorithm and its simulation and application to satellite communication using IBM quantum experience. (2018).

[18] Tyson Jones, Anna Brown, Ian Bush, and Simon C. Benjamin. 2019. QuEST and High Performance Simulation of Quantum Computers. *Scientific Reports* 9, 1 (2019), 10736. https://doi.org/10.1038/s41598-019-47174-9

[19] Jacob R Mandel. 2021. Quantum Computing: Resolving Myths, From Physics to Metaphysics. *Digital Commons Calpoly* (Mar 2021).

[20] Dan C. Marinescu. 2005. The promise of quantum computing and quantum information theory – quantum parallelism. In *Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium*. https://doi.org/10.1109/IPDPS.2005.430

[21] David Matthews. 2021. How to get started in quantum computing. *Nature* 591, 7848 (2021), 166–167. https://www.nature.com/articles/d41586-021-00533-x

[22] Benjamin Nachman, Miroslav Urbanek, Wibe A. de Jong, and Christian W. Bauer. 2020. Unfolding quantum computer readout noise. *npj Quantum Information* 6, 1 (2020), 84. https://doi.org/10.1038/s41534-020-00309-7

[23] Engineering National Academies of Sciences and Medicine. 2019. *Quantum Computing: Progress and Prospects*. The National Academies Press, Washington, DC. 272 pages.

[24] Michael A. Nielsen and Isaac Chuang. 2002. Quantum computation and quantum information. *American Journal of Physics* 70 (2002), 558.

[25] P. Pearle and A. Valentini. 2006. Quantum Mechanics: Generalizations. In *Encyclopedia of Mathematical Physics*, Jean-Pierre Françoise, Gregory L. Naber, and Tsou Sheung Tsun (Eds.). Academic Press, Oxford, 265–276.

[26] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug 2018), 79. https://doi.org/10.22331/q-2018-08-06-79

[27] Benjamin Schumacher. 1995. Quantum coding. *Physical Review A* 51, 4 (1995), 2738–2747. https://doi.org/10.1103/PhysRevA.51.2738

[28] Peter W. Shor. 1997. Polynomial-Time Algorithms For Prime Factorization And Discrete Logarithms On A Quantum Computer. *SIAM J. Comput.* 26, 5 (Oct 1997), 1484.

[29] Valentin Torggler, Philipp Aumann, Helmut Ritsch, and Wolfgang J Quantum Lechner. 2019. A quantum n-queens solver. *Quantum* 3 (Jun 2019), 149.

[30] Ken X. Wei, Isaac Lauer, Srikanth. Srinivasan, Neereja Sundaresan, Douglas T. McClure, David Toyli, David C. McKay, Jay M. Gambetta, and Sarah Sheldon. 2020. Verifying multipartite entangled Greenberger-Horne-Zeilinger states via multiple quantum coherences. *Physical Review A* 101, 3 (2020).