JQuantum for Abstract Experimentation of Quantum Computation

Chase Norman

July 26, 2018

Abstract

The literature of quantum computing explains algorithms in terms engineers and quantum physicists will understand. This paper seeks to explain the mathematics of quantum computation to a computer science audience. This paper is centered around the java library JQuantum, which has several advantages to other methods of learning quantum computation. It is accessible to all programmers, and allows for repeatable, dependable, and efficient calculations involving complex states. This library implements numerous popular quantum algorithms that can be used alongside classical java code. Through an explanation of quantum algorithms and their possibilities, this paper seeks to make programmers better prepared for the large-scale implications of quantum computing once quantum hardware is widely available.

1 Introduction

For all of the far-reaching possibilities that quantum computing promises, programmers tend to know very little about it. Quantum computation is incredibly inaccessible and reliant on engineering and physics knowledge to learn. The literature on quantum computation is not particularly helpful at aiding this problem, as it uses circuit notation or physics descriptions of quantum states to explain algorithms. It is near impossible for a programmer to interface with a large-scale quantum system in an abstract manner to test ideas.

This paper explains the mathematics of quantum programming through the use of the java library JQuantum. This paper serves as documentation for the library's syntax and also provides a detailed description of some of the most popular quantum algorithms and their uses. JQuantum was designed with encapsulation, clarity, and scalability in mind, making it exceptionally easy to program with.

With the ability to efficiently test embedded quantum systems, I expect quantum computing to become a much easier field to learn and produce meaningful results from. In simplest terms, this paper is a one-stop shop for quantum computing, designed for programmers.

2 Possibilities of Quantum Computing

Quantum computers are exceptional at solving optimization problems that require brute force (or a similarly inefficient algorithm) on classical computers. Several NP problems have been shown to be solvable in polynomial time by a quantum computer, although it is very unlikely that this is possible with an NP-complete problem. Among the problems that quantum computers have the capacity to solve more efficiently are factoring large composite numbers, optimization, and database searching.

Such an ability is possible because of the quantum mechanical properties that quantum computation exploits. Quantum computers are in a gigantic superposition of all possible combinations of its bits, called qubits, with varying coefficients. Quantum computation acts probabilistically; quantum algorithms attempt to gradually modify the superposition in order to increase the chances of a meaningful output.

Quantum computing is making headway, but is still in the early stages of development. As of 2018, the largest quantum computer has 72 qubits and is owned by Google. This is hardly enough to perform any meaningful computation, let alone to outperform the super computers Google already has at its disposal. At the time being, quantum computation is also very unpredictable. Quantum computation is done with a large margin of error and quantum states spontaneously decay if unmeasured. Most quantum computers also have a limitation in which qubits can only be put in a superposition with their nearest neighbors. This drastically reduces the computation power.

Quantum computing makes no pretense to replace classical computing or find solutions in a more deterministic way to classical computing. Quantum computing, rather, abides by a completely foreign logical structure, which is useful on a select set of problems in reducing the complexity of algorithms. Quantum computers are able to run logic gates on all superpositions simultaneously, in $O(1)$ allowing for an exponential speed-up in certain problems.

I expect quantum computing to increase in prevalence dramatically in the coming years. The limitations of classical computation has become quite obvious to computer scientists and quantum computing opens a door to solve some unanswered questions. Most famously, quantum computing has the capability to break all of encryption technology used online today. With its ability to check all possibilities simultaneously, quantum computation also looks promising for optimization problems and the future of machine learning. For these reasons, it is very important that future computer scientists learn how to use these tools that are so powerful.

3 The Math of Quantum Computing

Quantum computing is defined through complex-number matrix math. States are represented as matrices which are operated on by gates through matrix multiplication.

$$|\psi\rangle = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{bmatrix} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{bmatrix} = a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle$$

Figure 1: An example of a two qubit quantum state shown as a ket, and then as a linear combination of all two bit basis states. a, b, c, and d are complex coefficients of each of the basis states.

$$|\psi\rangle \Rightarrow \begin{bmatrix} \|a\|^2 : |00\rangle \\ \|b\|^2 : |01\rangle \\ \|c\|^2 : |10\rangle \\ \|d\|^2 : |11\rangle \end{bmatrix}, \|a\|^2 + \|b\|^2 + \|c\|^2 + \|d\|^2 = 1$$

Figure 2: diagram of the probability of a two qubit state being measured as any of the two bit basis states, based on the complex coefficients a, b, c, and d. This probability statement shows that the absolute squares of the coefficients must sum to 1.

3.1 States

In classical computing, the states of the computer are limited to the binary combinations of the bits in the system. In quantum computing, the state of the computer is made up of a linear combination of all of these states, called basis states. Each basis state has a complex coefficient denoting its magnitude and phase. Quantum states are represented with kets, a symbol with a bar on the left, the state name in the middle, and a angle bracket on the right. Basis states can be represented as the binary equivalent of the state within the ket.

Quantum states are represented as tall matrices containing a complex coefficient for each basis state, in binary counting order. To be measured, a quantum state must collapse into one of the basis states. This basis state is chosen probabilistically, with each basis state having a probability of the absolute square of the coefficient. All of the absolute squares of the coefficients therefore must sum to one.

3.2 Gates

In quantum computation, gates operate on one or more qubits to change their state. This operation is represented by a square unitary matrix of complex numbers. A matrix is defined as unitary when its inverse is its conjugate transpose. The conjugate transpose operation performs a transpose flip on the matrix followed by taking the complex conjugate of all of the elements in the matrix. This fact about unitary matrices allows quantum gates to easily be inverted through this operation. The conjugate transpose operation is generally denoted through a dagger symbol.

$$U^\dagger \equiv U^{-1}$$

Figure 3: A property of all quantum gates, as a result of being unitary. The dagger represents the conjugate transpose operation.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} a \\ c \\ b \\ d \end{bmatrix}$$

Figure 4: the application of the two qubit swap gate on a two qubit quantum state in order to produce a new quantum state. Gates should be of the same height as the quantum state. That is, 2^n where n is the amount of qubits.

Multiply a quantum gate to a state in order to apply it. Should the gate be smaller than the state it is being applied to, this multiplication must be done separately for every basis combination of non-operand qubits. Quantum mechanics causes these multiplications to be done $O(1)$, whereas classical computation must instead perform all multiplications $O(2^n)$.

It is worth mentioning that the bra symbol, which looks like the opposite of a ket, with a angle bracket on the left and a bar on the right, is representative of the conjugate transpose operation on a quantum state, creating a long matrix of opposite phase.

3.3 JQuantum implementation

The `QuantumState` class handles all of the math of combination and simplification of states along with the application of gates. This class is entirely hidden from the user, and performs all of the mathematics behind the scenes. It is always most efficient for a quantum state to be split up, should some of the qubits not be entangled with one another. After every gate the quantum state attempts to simplify itself to possibly split into multiple states. If a gate is applied onto qubits of different states, the two states are combined using the

$$\begin{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} b \\ a \\ d \\ c \end{bmatrix}$$

Figure 5: the application of the single qubit NOT operation on the most significant bit of a two qubit state. This requires two multiplications of the gate matrix. The matrices within matrices are to be interpreted as groupings, showing on which states the gate is applied.

$$\langle\psi| = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}^{\dagger} = \begin{bmatrix} \bar{a} & \bar{b} & \bar{c} & \bar{d} \end{bmatrix}$$

Figure 6: mathematical definition of the bra symbol.

$$\begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix}$$

Figure 7: Tensor product operation showing the splitting and joining of two single-qubit quantum states

tensor product operation, and the gate is applied to the newly created state.

The `QuantumState` class also handles all measurement mechanics, and is the only class with access to the complex coefficients of any particular state. Because the complex coefficients are stored on a classical computer, the measurement math could be invoked without causing a collapse to a basis state for testing purposes. This hypothetical operation is called a *sample*, and is one of the advantages of JQuantum. This class is intentionally impossible to access through normal means as a way to abstract the processes such that the programmer can interface with the qubits directly and the entanglement is handled underneath. Each `Qubit` has a `QuantumState` delegate, which will be the same for entangled `Qubits`.

The `QuantumGate` class serves to store quantum gates as two dimensional arrays of complex numbers. The class comes with statically defined gates that are commonly implemented on quantum hardware. The class comes with an invert method which applies the conjugate transpose operation on the unitary gate matrix. The quantum gates and their respective matrices are listed in the documentation.

4 Advantages of JQuantum

JQuantum is scalable. Because the qubits are abstractly referenced, adding and removing qubits takes only an initialization. Any function can be written to use a variable amount of qubits decided at runtime. Unlike real quantum hardware JQuantum is designed to make qubits universally entangleable. Quantum functions can be of any complexity or size, as the quantum states do not decay over time. The implementations of hardware are not the concern of the JQuantum programmer, such as the allocation of qubits. Qubits can also be entangled with any other qubit, as there is no nearest neighbor rule. The only limitation to the scalability of the system is the 32 bit size of java integers, forcing the maximum

number of entangled qubits to 32.

JQuantum is accessible. Java is one of the most popular programming languages. JQuantum requires little to know prior knowledge in order to start toying with. All algorithms are explained in the javadocs. JQuantum also allows for the embedding of quantum code alongside classical code. No setup is required to begin programming in quantum. All quantum algorithms are written in code, with explanatory documentation, as opposed to with circuit notation.

JQuantum is encapsulated. JQuantum focuses on the possibilities of quantum computing but not on their details. This allows a programmer to simply code while having the library take control of the mathematics. Qubits in separate objects can be entangled behind the scenes, but to the programmer they can be treated as separate entities. JQuantum presents qubits just as classical computers present classical bits.

All calculations performed through JQuantum are reproducable and exact to double precision. JQuantum programs, unlike real quantum hardware, can be run with the expectation of accurate results, even when applying algorithms of high depth. Quantum registers can be remeasured, or even analyzed for their basis state probabilities.

5 Documentation

The following information is available in the javadocs.

5.1 Gates

These are the statically defined gates of the `QuantumGate` class.

5.1.1 Hadamard

This gate transforms the $|0\rangle$ basis state into an equal superposition of $|0\rangle$ and $|1\rangle$, while transforming the $|1\rangle$ basis state into the same superposition, except with the opposite phase on the $|1\rangle$.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

5.1.2 Pauli-X

Acts as a NOT gate on a single Qubit, transforming $|0\rangle$ to $|1\rangle$ and vice versa.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

5.1.3 Pauli-Y

Represents a π radian rotation about the Y-axis

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

5.1.4 Pauli-Z

Represents a π radian phase shift in a single Qubit.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

5.1.5 SWAP

Acts on two Qubits, swapping their respective states.

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.1.6 $\sqrt{SWAP}$

When performed twice, this gate is equivalent to a swap, therefore this gate is the equivalent of the square root of a swap.

$$SQRT\_SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 \\ 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.1.7 Controlled-NOT

Operates on two Qubits, performing NOT on the first iff the second is $|1\rangle$

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

5.1.8 $\sqrt{\neg}$

When performed twice, this gate is equivalent to a NOT (X) gate, so it considered the square root of the not gate.

$$SQRT\_NOT = \frac{1}{2} \begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}$$

5.1.9 Phase Shift Gates $R(k)$

Creates a phase-shift gate, on one Qubit, of order k, such that the shift is determined by $2\pi/(2^k)$ radians. $R(1) = Z$

$$R(k) = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi}{2^k}i} \end{bmatrix}$$

5.1.10 Controlled Gates

Creates a gate that is controlled by the last parameter passed in. This means the gate will perform iff the last Qubit is $|1\rangle$

$$C(U) = \begin{bmatrix} I_n & 0_n \\ 0_n & U \end{bmatrix}$$

5.1.11 Inverse

Creates an inverse gate, that when applied before or after the gate will result in the identity matrix. Because the matrices in quantum gates are unitary, this is equivalent to the conjugate transpose.

$$U.inverse() = U^\dagger$$

5.2 Qubit and QubitRegister

A `Qubit` can be created through a default constructor or a constructor with a `boolean` representing the basis state. The `measure` and `sample` methods return a result of this `Qubit`, but the measure command incurs a collapse, separating the `Qubit` from its state. The `probabilityOf` method returns the probability that a particular basis state is found when collapsed. These probabilities are shown in the `toString` method. The `isEntangledWith` method will check if this `Qubit` is owned by the same `QuantumState` as another `Qubit`

The `QubitRegister` works in much the same way, except all of its methods use `int` instead of `boolean`. A `QubitRegister` can be initialized with an `int` length parameter, along with an optional `int` value parameter representing its starting basis state. The `toString` method will print the probability of every state. Each `Qubit` of the `QubitRegister` can be found in the public `qubits` array instance variable.

Gates can be used through their `accept` method on a varargs of `Qubit` or on a `QubitRegister`. A gate can be initialized with a `Complex[][]`.

```
public static void main(String[] args){
    Qubit a = new Qubit();
    Qubit b = new Qubit(true);
    QubitRegister qr = new QubitRegister(5);
    QubitRegister qr2 = new QubitRegister(5, 3);

    H.accept(a);
    CNOT.accept(b,a);

    QuantumAlgorithm.QFT(qr2);

    System.out.println(a.probabilityOf(a.sample()) + ", " +  a.isEntangledWith(b));
```

```
    System.out.println(a.measure() + ", " + b.measure());
}
```

5.3 Algorithms

5.3.1 QFTADD

Changes the state of a `QubitRegister` from QFT(a) to QFT(a + b). The first parameter of this *MUST* have gone through the QFT function beforehand, and the result will only be found by performing the inverse QFT function.

- qft: a `QubitRegister` for the function to be applied on, already passed through the QFT function
- b: value to be added to the register

```
public static void QFTADD(QubitRegister qft, int b){
    for(int q = qft.qubits.length-1; q >= 0; q--)
        for(int i = 0; i <= q; i++)
            if (BitUtils.bit(b,q-i))
                R(i+1).accept(qft.qubits[(qft.qubits.length-1) - q]);
}
```

5.3.2 CQFTADD

Changes the state of a `QubitRegister` from QFT(a) to QFT(a + b) if the control Qubit is set. The first parameter of this *MUST* have gone through the QFT function beforehand, and the result will only be found by performing the inverse QFT function.

- qft: a `QubitRegister` for the function to be applied on, already passed through the QFT function
- b: value to be added to the register
- c: the control `Qubit`

```
public static void CQFTADD(QubitRegister qft, int b, Qubit c){
    for(int q = qft.qubits.length-1; q >= 0; q--)
        for(int i = 0; i <= q; i++)
            if (BitUtils.bit(b,q-i))
                C(R(i+1)).accept(qft.qubits[(qft.qubits.length-1) - q], c);
}
```

5.3.3 QFTMAC

the Multiplier Accumulator. Changes the state of a `QubitRegister` from QFT(a) to QFT(a + x*y)

- qft: a `QubitRegister` for the function to be applied on, already passed through the QFT function
- x: operand to be multiplied
- y: multiplier constant

```
public static void QFTMAC(QubitRegister qft, QubitRegister x, int y){
    for(int i = 0; i < x.qubits.length; i++)
        CQFTADD(qft,y*(1<<i),x.qubits[i]);
}
```

5.3.4 genericQFTADD

Changes the state of a `QubitRegister` from QFT(a) to QFT(a + b). The first parameter of this *MUST* have gone through the QFT function beforehand, and the result will only be found by performing the inverse QFT function.

- qft: a `QubitRegister` for the function to be applied on, already passed through the QFT function
- b: quantum value to be added to the register

```
public static void genericQFTADD(QubitRegister qft, QubitRegister b){
    for(int q = qft.qubits.length-1; q >= 0; q--)
        for(int i = 0; i <= q; i++)
            if (q-i < b.qubits.length)
                C(R(i + 1)).accept(qft.qubits[(qft.qubits.length - 1) - q], b.qubits[q - i]);
}
```

5.3.5 QFT

Applies the Quantum Fourier Transform on a register

- qr: operand

```
public static void QFT(QubitRegister qr){
    for(int x = 0; x < qr.qubits.length; x++){ //0..<length
        H.accept(qr.qubits[qr.qubits.length-1-x]);
        for(int y = 0; y < qr.qubits.length-1-x; y++)
            C(R(y+2)).accept(qr.qubits[qr.qubits.length-1-x],qr.qubits[qr.qubits.length-x-y-2]);
    }
    reverse(qr);
}
```

5.3.6 inverseQFT

Applies the inverse Quantum Fourier Transform on a register

- qr: operand

```
public static void inverseQFT(QubitRegister qr) {
    for(int x = 0; x < qr.qubits.length; x++){ //0..<length
        for(int y = 0; y < x; y++)
            C(R(1+x-y)).inverse().accept(qr.qubits[qr.qubits.length-1-x],qr.qubits[qr.qubits.length-1-y]);
        H.accept(qr.qubits[qr.qubits.length-1-x]);
    }
    reverse(qr);
}
```

5.3.7 H

Applies the hadamard gate on the entirety of a `QubitRegister`.

- qr: register to apply the Hadamard gate to.

```
public static void H(QubitRegister qr){
    for(Qubit q : qr.qubits)
        H.accept(q);
}
```

5.3.8 reverse

reverses the qubits of a `QubitRegister` by directly reordering them.

- qr: `QubitRegister` to be reversed

```
public static void reverse(QubitRegister qr){
    Qubit[] q = new Qubit[qr.qubits.length];
    System.arraycopy(qr.qubits,0,q,0,qr.qubits.length);

    for(int i = 0; i < qr.qubits.length; i++)
        qr.qubits[i] = q[qr.qubits.length-i-1];
}
```

5.3.9 grovers

Performs Grovers search algorithm. Finds a particular value of an oracle function in O(sqrt(N)) evaluations

- oracle: function to be searched. This oracle function should flip the polarity of whichever state is to be searched for.
- length: the amount of bits in the result to be searched

```
public static int grovers(Consumer<QubitRegister> oracle, int length){
    QubitRegister qr = new QubitRegister(length);

    Consumer<QubitRegister> nand = oracle(x->x!=0);

    H(qr);
    for(int invocations = 0; invocations < Math.PI*0.25*Math.sqrt(1<<length)+1; invocations++){
        oracle.accept(qr);
        H(qr);
        nand.accept(qr);
        H(qr);
    }

    return qr.measure();
}
```

5.3.10 deutschJozsa

Determines whether a predicate function is constant or balanced. The function should flip the phase of any "true" values and leave others alone.

- oracle: predicate function
- length: length of `QubitRegister` this function should operate on

returns `true` if constant, and `false` if balanced

```
public static boolean deutschJozsa(Consumer<QubitRegister> oracle, int length){
    QubitRegister qr = new QubitRegister(length);

    H(qr);
    oracle.accept(qr);
    H(qr);

    return qr.measure()==0;
}
```

5.3.11 oracle

Creates a quantum function that flips the phase of any basis state coefficients as determined by the predicate specified.

- function: predicate to determine which basis state coefficients are phase-flipped

returns a quantum oracle function representation of the predicate given

```
public static Consumer<QubitRegister> oracle(Predicate<Integer> function){
    return qr->{
        Complex[][] matrix = new Complex[1 << qr.qubits.length][1 << qr.qubits.length];

        for (int x = 0; x < 1 << qr.qubits.length; x++) {
            boolean test = function.test(x);
            for (int y = 0; y < 1 << qr.qubits.length; y++) {
                if (x == y && test)
                    matrix[x][y] = new Complex(-1);
                else if (x == y)
                    matrix[x][y] = ONE;
                else
                    matrix[x][y] = ZERO;
            }
        }

        new QuantumGate(matrix).accept(qr);
    };
}
```

5.3.12 quantumFunction

Creates a quantum function version of a given integer operator. The resulting function will take two registers as input, one to be used as input for the function, and the other to be used as a regiser for the output. Any data in the output register will be ignored. The resulant function will instantiate a gate of arbitrary size, that will execute the function given.

- function: The function to be mapped into a quantum function

returns the quantum function created

```
public static BiConsumer<QubitRegister, QubitRegister> quantumFunction(UnaryOperator<Integer> function){
    return (input, output)->{
```

```
        Qubit[] qubits = new Qubit[input.qubits.length+output.qubits.length];
        System.arraycopy(input.qubits,0,qubits,0,input.qubits.length);
        System.arraycopy(output.qubits,0,qubits,input.qubits.length,output.qubits.length);

        Complex[][] matrix = new Complex[1 << qubits.length][1 << qubits.length];

        int mask = (1<<input.qubits.length)-1;

        for (int x = 0; x < 1 << qubits.length; x++) {
            for (int y = 0; y < 1 << qubits.length; y++) {
                int value = function.apply(y&mask)%(1<<output.qubits.length);
                matrix[x][y] = ZERO;
                //if x is the output of y
                if((x&mask)==(y&mask) && (x>>>input.qubits.length)==value)
                    matrix[x][y] = ONE;
            }
        }

        new QuantumGate(matrix).accept(qubits);
    };
}
```

5.3.13 periodFinder

More likely to evaluate to a possible period of a quantum function.

- oracle: function that takes in two registers. This function changes the value of the second to the output of a function with the first.
- in: length of the input register of the function
- out: length of the output register of the function

reurns a `QubitRegister` with higher probability to evaluate to the period of the function

```
public static QubitRegister periodFinder(BiConsumer<QubitRegister,QubitRegister> oracle, int in, int out){
    QubitRegister input = new QubitRegister(in);
    QubitRegister output = new QubitRegister(out);
    H(input);
    oracle.accept(input,output);
    inverseQFT(input);
    return input;
}
```

5.3.14 shorsAlgorithm

Performs a rudimentary, inefficient version of Shor's factorization algorithm. This will print out debug information, and will occasionally reproduce the value to be factored. Should really only be tested on the values 15 or 21, as this is unusable with square numbers.

- N: value to be factored, probably should not exceed 21

returns a factor of N

```
public static int shorsAlgorithm(int N){
    int r;
    int a;
    while(true){
        a = (int)(Math.random()*N);
        if(BigInteger.valueOf(a).gcd(BigInteger.valueOf(N)).intValueExact()!=1) {
            System.out.println("found solution trivially :(");
            return BigInteger.valueOf(a).gcd(BigInteger.valueOf(N)).intValueExact();
        }
        r = shorsQuantumSubroutine(N,a);
        if(r%2==0 && ((int)Math.pow(a,r/2))%N != N-1)
            return BigInteger.valueOf((int)Math.pow(a,r/2) + 1).gcd(BigInteger.valueOf(N)).intValueExact();
    }
}

private static int shorsQuantumSubroutine(int N, int a){
    int evaluations = 0;
    int n = log2(N);
    int q = log2(N*N);
    int Q = 1<<q;
    QubitRegister qr = periodFinder(quantumFunction(x -> ((int) Math.pow(a, x)) % N), q, n);
    while(true) {
        evaluations++;
        int y = qr.sample();
        int[][] cfe = fractionOf(y / (double) Q, 30);
        //fractionOf() produces an array of continued fraction expansions of a given double.
        for (int x = 0; x < cfe[0].length; x++) {
            int k = 1;
            while(cfe[1][x]*k < N){
                if(Math.pow(a,cfe[1][x]*k)%N==1){
                    System.out.println("candidate found in " + evaluations + " evaluations.");
                    return cfe[1][x]*k;
                }
                k++;
            }
        }
    }
}
```

6 Conclusion

JQuantum allows programmers to interface with quantum systems abstractly. With the increasing prevalence of quantum computation, such an ability is be-

coming increasingly important. JQuantum can be used alongside classical java code, and the library can be found at `https://github.com/unknownchasen/JQuantum`.