Simulating Quantum Algorithms with Q-Prog

Walter O. Krawec
walter.krawec@gmail.com

Abstract

Q-Prog (Quantum-Programmer) is a new programming language allowing users to write and then simulate quantum algorithms and to perform experiments with arbitrary dimensional quantum systems. Also, due to its design and its minimal dependencies, Q-Prog may be easily embedded into a C++ application and extended to support user/application specific functionality.

In this paper we describe how the interpreter works including its core functionality and also the portions specific to quantum simulation. We then describe specific examples which benefit from the use of Q-Prog . These include quantum random walks, and the evolution of quantum operators.

1 Introduction

This paper describes Q-Prog (Quantum-Programmer), a new interpreter permitting users to program and then simulate quantum algorithms and to perform experiments with arbitrary, finite dimensional quantum systems. Unlike past quantum simulators, Q-Prog has several unique features that are beneficial to certain applications of quantum information theory and quantum computation.

In this report, we describe the interpreter in detail. Beginning with its core functionality, we then move on to describe the portions specific to quantum simulation. We will describe how Q-Prog can handle arbitrary (finite) dimensional systems which are not necessarily powers of two (unlike similar prior work). We also explain how, in addition to running as a stand-alone system, Q-Prog can also be easily embedded into a C++ program and extended to support application specific functionality.

We next look at specific applications which benefit from the use of Q-Prog . This includes first, a simulation of quantum random walks as described in [9], [11], [5] (in this application, we are also able, by embedding Q-Prog into a custom-made graphing program, to view the changing distribution over time). Following this we will look into the evolution of quantum operators as described in [8]. This application will take full advantage of Q-Prog's ability to handle non power of two spaces and also its ability to be embedded into another C++ program.

We conclude with a survey of past simulators which are similar to our own and mention how Q-Prog is different. In particular, while there do exist several programs permitting a

user to write and then simulate a quantum algorithm, we are not aware of any stand-alone quantum simulator permitting the experimentation of non-power of two dimensional spaces (an important ability in some applications as we'll mention and also vital for preserving space-time complexity).

2 Language Design

The primary goal of `Q-Prog` is to allow the programmer to write and then simulate quantum algorithms involving spaces of arbitrary (not necessarily powers of two) dimension. This is important in many applications (e.g. quantum random walks and quantum decision making) where qubits are not explicitly used (while they may be used in a final application on a quantum computer, it is often easier to think and work with other dimensions). It is also important to reduce the space-time complexity (which grows exponentially with the number of subspaces) if an exact power of two is not needed for a particular application.

A second goal is to permit a user to easily embed the interpreter into a C++ program. To achieve this, `Q-Prog` has minimal dependencies (and in fact only requires the standard C++ libraries). It is also cross platform tested so far on Windows (MS VC++ 2010) and Mac (g++ v. 4.01).

Our third goal is extensibility. We achieve this by allowing a user to easily write new commands in C++ and "register" them with the interpreter. This is done simply by inheriting an abstract class and overwriting a single procedure. This permits a user to tailor `Q-Prog`'s functionality to a certain application without bloating the primary interpreter.

2.1 The Interpreter

`Q-Prog` is written in C++ and was designed to run stand-alone or to be easily embedded in another application. We choose to use a syntax style similar to Scheme/Lisp. This permits (in our opinion at least) a user to more readily learn the language. But more importantly, this style of language is particularly useful for experiments in genetic programming (for example, as in [13]).

There are two primary components: the store and the interpreter. `Q-Prog` permits users to declare typed variables in the store which simply maps a string name to an `Object`. An `Object` is an internal structure specifying the data type (as a string) along with a pointer to the variable data. Currently supported data types include string, integer, double, complex, and matrix. There are also several quantum related data types which are discussed later. For notation, if σ is a store, then $\sigma = \{(\texttt{name}_i, \texttt{data}_i*)\}_{i=1}^{|\sigma|}$ where $\texttt{data}_i*$ is a pointer to a data type. The interpreter begins by defining a global store $\sigma_0 = \emptyset$. This store may, however, grow as programs are run or may be pre-configured by the user.

The interpreter is accessed via a simple call to the C++ function:

```
evaluate(string, store, Object*, bool globalMode)
```

which will parse and then run the given string using the given store (or a copy of it depending on the value of `globalMode` as we discuss below). The returned value will be stored in the passed `Object` pointer; this will contain the result of the operation. If running as a stand-alone interpreter, this function is called whenever the user runs a command from the prompt (setting `store` $= \sigma_0$). Otherwise, a user application may call this command arbitrarily to access functionality in `Q-Prog`. This function works by first receiving a text string which constitutes a single program. This string is then parsed on-the-fly into an expression of the form `string-constant` or `command` $\{$`arguments`$\}^+$ where each `argument` is a string that may further be parsed into one of these two forms. The value of a `string-constant` (when we say "value of", we mean the data returned in the `Object` structure after running `evaluate`) is simply the string; the value of a command is, of course, the result of running the command given the value of each argument. If the command does not return anything, the `Object` output is simply `NULL`.

More specifically, when called, if `globalMode = false`, the `evaluate` function will create a copy of the passed store. It will then parse `line` into a series of $n \geq 1$ tokens. If $n = 1$, the function simply returns the string `line` (as an `Object`). If $n > 1$, the first token is assumed to be a command, while the rest are arguments. Functionality from this point depends on which command is run. `Q-Prog` contains several pre-build functions/commands and also permits a user to write new ones (if embedding in another application). While we do not list every supported command in this paper, we do mention a few.

The store may be altered using the `new` and `set` commands. Running `new TYPE NAME [ARGUMENTS]` will add to the copied store (or the given store if `globalMode = true`) a new variable with the appropriate name and type. Arguments are optional and may specify, for example, the number of rows/columns in a matrix. Basic data types (integers, doubles, complex, and matrices) include constructors such as `integer NUMBER` or `complex REAL IMAG` (where the value of `NUMBER`, `REAL`, and `IMAG` are simply strings). The value of the first is simply an integer assigned the value `NUMBER`; the output of evaluating the second is a complex type equal to `REAL` $+$ `IMAG`i. There are also several types specific to quantum simulation. These types are spaces, which represent an n dimensional Hilbert space, and systems, which are the tensor products of multiple (previously defined) spaces. The value of a `new` command is simply `NULL`. Notice that only the copied store is altered, hence once the evaluation of the `new` command is finished, this new variable will be deleted (unless running in global mode). The behavior however is different when pairing this command with either a `begin` or `while` block which we will describe shortly.

Arrays may also be created via: `new array NAME TYPE SIZE`. Such a call will add to the store `SIZE` new variables of type `TYPE` called `NAME.i` for `i` $= 0, 1, \cdots$ `SIZE` $- 1$. It will also add to the store an integer variable called `NAME.size` set appropriately. You may index an array using `index NAME` $\{$`INDICES`$\}^+$. The value of this command (if we assume the given set of `INDICES` is `INDEX`$_1$, `INDEX`$_2$ $\cdots$ `INDEX`$_n$) is a string equal to `NAME.INDEX`$_1$`.INDEX`$_2$ $\ldots$ `INDEX`$_n$. In this way, one can have multi-dimensional arrays; furthermore, one may organize variables with sub-names (that is, `INDEX`$_i$ may evaluate to either an integer or a string). This function is meant to be paired with the `get` command which, as we'll see, expects a string

as its first argument.

To modify the value of a variable in a store, the `set NAME VALUE` command may be used. The value of `NAME` should be a string while the value of `VALUE` should be an object of the same type as the variable `NAME` (e.g. integer, complex, etc.). Since data types in a store are pointers, this command will alter the value of this variable in all future evaluations. Finally, the value of a variable in the store may be returned using the `get NAME` command where `NAME` evaluates to a string. Evaluating this command will return an `Object` structure containing a new copy of the requested variable's data (not a pointer to the original, but to a new, temporary copy).

To chain together multiple commands in a single program, a `begin` block is used. Here we have the value of `begin` $\texttt{command}_1$ $\texttt{command}_2$ $\cdots$ $\texttt{command}_n$ is the value of $\texttt{command}_n$ given that, if for some i we have that $\texttt{command}_i$ is a `new` declaration, then that newly created variable exists in all $\texttt{command}_j$ for $j > i$. That is to say, the store is expanded to carry the newly declared variable to all future commands in this `begin` block and, as a consequence, this new variable exists in all calls to `evaluate` made by a $\texttt{command}_j$ (a similar rule can be described between the relationship of a `new` command and a `while` block).

Example 2.1.

```
begin
  (print here-the-store-is-empty)
  (new integer i)
  (print now-the-store-has-i)
  (set i (integer 1))
  (get i)
```

The value of this program will be an integer 1, since the last command `get i` returns the value of `i` which has been set to 1.

2.2 Quantum Simulations

To work with a quantum system in `Q-Prog` (which is, of course, our primary goal), one first typically declares a new space. For instance, to work with a qubit, one would write `new space Qubit (integer 2)` which creates a new space called "Qubit" (the name need not be "Qubit") of dimension 2. `Q-Prog` however provides great flexibility in defining a quantum system. As mentioned before, one of the design goals was to permit a user to simulate a quantum system of any arbitrary (not necessarily power of two) finite dimension. This is achieved by running `new space NAME DIMENSION`. After this call, there will be $2 \cdot$`DIMENSION` new vectors in the store labeled `NAME.ket.i` and `NAME.bra.i` for $\texttt{i} = 0, 1, \cdots, \texttt{DIMENSION}-1$. These are the ket and bra orthonormal vectors which span the Hilbert space of the specified dimension.

Following this, one may create a new system, which is the product of one or more spaces. By explicitly defining systems in this manner (instead of simply specifying the total dimen-

sion), we allow `Q-Prog` to easily handle measurements and the application of operators. A system is defined via `new system SYSTEMNAME {SPACENAME COPIES}`$^+$ where `COPIES` is an integer specifying the number of times to copy `SPACENAME`.

Example 2.2. For example, to create a five qubit system (call it `H` for *Hilbert* space), one would run the program:

```
begin
  (new space Qubit (integer 2))
  (new system H Qubit (integer 5) )
```

Note, again, that the space-time complexity required to work with a quantum system goes up exponentially with the number of subspaces. In the above example, we require 2^5 complex numbers (each complex number requires two double precision floating point numbers). This is one of the reasons why it is very useful to support arbitrary dimensional spaces. If a user wishes to simulate a 17 dimensional space, using only qubits would require 32 dimensions (thereby requiring 240 additional bytes to store a single quantum state). Furthermore, operators (in the qubit case) would require $32^2 = 1024$ complex entries (thereby requiring 11760 extra bytes for just a single operator). Additionally the time complexity goes up exponentially also when working with larger systems (for instance, applying an operator is, essentially, a matrix multiplication).

After defining a system, you may define an actual quantum state. Since, as we mentioned before, a state is simply a vector, one first creates a new matrix variable and sets it to the appropriate ket vector (or tensor of ket vectors). If `psi` is the name of the matrix (previously created via a `new` command), and `H` is the name of the system (composed of spaces named `space-1`, `space-2`, $\cdots$, `space-n`), a quantum state may be defined by either manipulating the matrix entries of `psi` manually (via the commands `get_element` and `set_element`) or by using one of the following methods:

```
set psi (tensor
  (get space-1.ket.0)
  (tensor
    (get space-2.ket.0)
    (tensor
      ...)))
```

```
set psi (state (get H)
  (integer 0)
  (integer 0)
  ...
  )
```

These will both construct the state $|\psi\rangle = |0\rangle_1 \otimes |0\rangle_2 \otimes \cdots \otimes |0\rangle_n$. The first manually constructs a state by tensoring the kets of each space individually. This method, of course,

may also be used in a while loop. The second method calls the `state` command which will automatically tensor the appropriate kets given system `H`. This is useful if you know exactly the dimension and indices to use; whereas the first method may be extended in a while loop to support user modifiable sized spaces.

To create a superposition, again you may manipulate the state vector manually. Alternatively, one may use an operator (described next) or simply add previously created states together multiplying with the correct scaler. `Q-Prog` treats quantum states simply as vectors so one can manipulate them as such when needed.

Example 2.3. The following program will create a space of dimension 3. Then it will create a state `psi` in the form:

$$|\psi\rangle = \sqrt{\frac{2}{3}}\,|1\rangle + \frac{1}{\sqrt{3}}\,|2\rangle\,.$$

Recall from the previous section that this state, when measured, will "collapse" to the state $|1\rangle$ with probability $(\sqrt{2/3})^2 = 2/3$ and will collapse to $|2\rangle$ with probability $1/3$.

Notice below that, when we declare `psi`, we do not specify the exact size required (which is 3 rows by 1 column). `Q-Prog` automatically readjusts the size of a matrix when using the `set` command.

```
begin
  (new space Example (integer 3))
  (new matrix psi (integer 1) (integer 1))
  (set psi (add
    (mult
      (sqrt (div (double 2) (double 3)))
      (get Example.ket.1))
    (mult
      (div (double 1) (sqrt(double 3)))
      (get Example.ket.2))))
```

After the `set` command, the store will hold a vector $(0, \sqrt{2/3}, \sqrt{1/3})$ called `psi`.

As mentioned in the previous section, there are two ways to manipulate a quantum system: applying a unitary operator and measuring. `Q-Prog` is very flexible in the manner in which one applies an operator. For example, you can explicitly describe an operator on the entire system as a matrix (constructing it manually in its entirety or tensoring multiple smaller matrices together) then simply using the `mult` command to multiply the operator matrix with the quantum state vector. Alternatively, you can define an operator on a particular sub-space of a system and apply it only to that subspace. Thirdly, you can use one of the pre-built operators available including the quantum Fourier transform or, if you are working with qubits, the controlled NOT gate.

Measurements may be made on any sub-space of a defined system. Unlike several other quantum simulators, `Q-Prog` does not create a branch point for a measurement (i.e. it will

not simulate all possible outcomes of a measurement at once). Instead, when a measurement is made, it will calculate the probabilities of the state collapsing to a particular state and return a new vector representing this quantum state after measurement. This is useful for running real time quantum simulations where one is interested in the result after measurement more than the possible results over all measurement results. Also since `Q-Prog` was designed to handle sub-spaces with dimension greater than two, branching can quickly grow in complexity. That being said, it wouldn't be difficult to add a second measurement command to `Q-Prog` supporting this branching mechanism (we describe later how to extend the functionality of `Q-Prog`).

Given a state `psi` in some already defined system `H` consisting of n spaces, one may make a measurement by simply calling: `measure STATE SYSTEM SUBSPACE_INDEX RESULT_NAME`. Here `STATE` is the quantum state vector, `SYSTEM` is the system the quantum state lives in, `SUBSPACE_INDEX` is an integer specifying which of the n spaces to make a measurement on (if you wish to measure the entire system, simply repeat this call n times). Finally `RESULT_NAME` is a string specifying a variable name in the store to save the measurement result in (this variable should of course be an integer). The value of the `measure` command is a new vector describing the new quantum state after measurement.

Note that, even though the sub-space measured is now in a known state, the other $n-1$ sub spaces may be in a superposition still. Also note that, since this is a simulation, it is also possible to not destroy the original quantum state after measurement. Either copy the state vector to another variable name before measuring or ignore the returned vector of `measure`.

Example 2.4. The following defines $|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle|0\rangle + \frac{1}{\sqrt{2}}|1\rangle|2\rangle$ and performs a measurement of the first sub-space. With probability $1/2$ the measured value will be $|0\rangle$ and the entire state will collapse to $|0\rangle|0\rangle$. Otherwise, the measured value will be $|1\rangle$ and the entire state will be $|1\rangle|2\rangle$. This is an example of an entangled state.

```
begin
  (new space First (integer 2)
  (new space Second (integer 3))
  (new system H First (integer 1)
    Second (integer 1) )
  (new matrix psi (integer 1) (integer 1))
  (set psi (add
    (state (get H) (integer 0) (integer 0)))
    (state (get H) (integer 1) (integer 2)))
  (set psi (mult (div (double 1)
    (sqrt (integer 2))) (get psi)))
  (new integer result)
  (set psi (measure (get psi) (get H)
    (integer 0) result))
```

After running this program, `result` will hold either 0 or 1 (depending on the result of the measurement which is, of course, a probabilistic process). If the measurement result is

0, the matrix `psi` will represent the quantum state $|0\rangle|0\rangle$. Otherwise, `psi` will represent $|1\rangle|2\rangle$.

Since this is a simulation, we may also calculate the probability of a state collapsing to a certain outcome in a subspace of our system (this is a very useful operation, especially if one wants to know how a state will behave after applying an operator and before measurement). This may be done using the `measure_distribution` command which operates exactly as `measure` however it does not return a new state vector and `result` should be an array of appropriate size (specifically, the size should be the dimension of the subspace you're investigating). In particular, if running this command on an n dimensional subspace of a particular system, the i'th entry of the result array represents the probability that, upon measuring, the state will collapse to $|i\rangle$ in this subspace. One may then use the `print_file` command to print this distribution to the hard disk in a CSV style format.

One last language feature we'll mention is the ability to easily generate random unitary matrices (for use as random operators). Using the technique described in [8] and [16], one may construct a random unitary matrix (for an n dimensional system) simply by providing an array of n^2 random doubles with each entry in the range $[0, 2\pi)$. The matrix constructor can then be called and passed these arrays, upon which a random unitary matrix will be created. This will be very useful in a later application we'll describe involving the evolution of unitary operators.

2.3 Embedding

As mentioned, one of the design goals was to permit a user to easily embed `Q-Prog` in another C++ program. This may be done simply by including a single header file and compiling/linking with the `Q-Prog` object files (there are two of them - one for the algebra system, the other for the interpreter). The only dependencies are the standard C++ libraries (in particular, STL).

When `QProg.h` is included, one may create multiple stand-alone instances of the interpreter (each with its own store) by instantiating the class `QProg`. From this, you may alter the store directly if necessary (using `getGlobalStore()`). You may then use the:

```
evaluate(string line, store, Object*, bool globalMode))
```

command to evaluate the program specified by `line`. If `globalMode` is set to `true`, then any calls to `new` (in the program specified by `line`) alter the global store; as such any changes will remain between evaluations of different programs. If this is set to false, the store is reset to its original state when the call to `evaluate` completes (usually empty unless one first manually inserts data into it). Furthermore, at any point in your program, you may run `shell()` which will provide access to the `Q-Prog` interpreter via a command prompt (including the current global store). This permits a user to modify/view the store, and run new programs allowing for easier debugging of embedded applications.

Certain applications require more specific commands. A user may write their own application-specific commands as follows. For each command, the user will overwrite an abstract class `Command`. This class contains a virtual function `evaluate(textParser, QProg*,`

Object*, store*). Recall that Q-Prog (at the moment at least) parses programs on the fly. Hence textParser contains at least 1 token, namely the name of the command that's currently running. Any additional tokens are strings which must be evaluated (using the pointer to the QProg interpreter) before being used by the command. The command should evaluate any arguments it requires. This may be done simply by calling

qp->evaluate(textParser[i], store) for $i = 1, 2, \cdots$

It should then store its output in the provided Object pointer.

After defining the command by inheriting the Command class, one must register it with the interpreter by calling:

registerCommand(new NAME_OF_COMMAND).

When the interpreter is running and it is processing a command call, it will first look through the list of pre-defined commands. If no match is found, it will look through the registered commands. If a match is found there, it will run the appropriate Command::evaluate procedure.

We will discuss an example later which embeds Q-Prog in a graphical program. This example will also demonstrate the creation of new commands which, in this case, are used to permit a Q-Prog program to interface with the graphing application.

3 Applications

We now turn our attention to applications of Q-Prog . We'll first consider quantum random walks (described in [9], [11], [5]) and simulate a random walk on the integers modulo N (including a real time graph of the distribution of the particle's position evolving over time). This application will take advantage of Q-Prog's ability to handle non power of two dimensional spaces. Following this, we will consider a problem discussed in [8] where the goal is to find a unitary operator which satisfies certain properties. This will take advantage not only of Q-Prog's ability to handle arbitrary dimensional spaces, but also the ability to easily be embedded in a user specific C++ program.

Full source code for these applications (along with some other test programs such as the Deutsch-Jozsa algorithm and Shor's algorithm) may be found on the website:
http://www.walterkrawec.org/qprog.

3.1 Discrete Time Quantum Random Walks

Quantum random walks (see [9], [11], [5]) are an exciting area of research in quantum computing. For the purposes of this demonstration, we'll simulate the results of [9] involving a discrete time walk over the integers modulo N. This setup requires two Hilbert spaces: a "coin space" denoted $\mathcal{H}_C$ of dimension 2 and a "position space" denoted $\mathcal{H}_P$ of dimension N (here is where Q-Prog's ability to handle multiple dimensional spaces will be useful).

The walk begins with a quantum state $|\psi_0\rangle = |0\rangle|j\rangle$ for some fixed $j \in \{0, 1, \cdots N-1\}$. A single iteration of the walk involves applying a unitary operator U which acts only on the coin space. In this example we'll be using the Hadamard transform. Therefore, after one application we'll have $U|\psi_0\rangle = \frac{1}{\sqrt{2}}|0\rangle|j\rangle + \frac{1}{\sqrt{2}}|1\rangle|j\rangle$. The last step is to apply a "shift" operator. This will send the state $|0\rangle|p\rangle \rightarrow |0\rangle|p-1 \bmod N\rangle$ and $|1\rangle|p\rangle \rightarrow |1\rangle|p+1 \bmod N\rangle$. That is to say, if the coin space is a 0, it will shift the position of the particle left (subtract one); otherwise it will move right. Hence, one iteration of the quantum random walk will leave the state $|\psi_0\rangle$ in the state: $\frac{1}{\sqrt{2}}|0\rangle|j-1\rangle + \frac{1}{\sqrt{2}}|1\rangle|j+1\rangle$. This process is repeated T times and then a measurement on the position space is made (not the coin space). As it turns out, the distribution of the particle's position is very different from the classical case as we'll soon see.

We will now write a `Q-Prog` program to simulate this. You may find the entire program online at our website; here however we will show the important parts. Assuming that the store contains the values `N`, `T`, and `j` (set appropriately, possibly using the `user.input` command or by modifying the global store), we first create the appropriate spaces and system (the following code snippets are all contained within a `begin` block):

```
(new space Coin (integer 2)
(new space Position (get N))
(new system H Coin (integer 1)
  Position (integer 1))
```

Next we will define our shift operator. Mathematically, the operator is defined as follows:

$$\begin{aligned} S &= |0\rangle\langle 0| \otimes \left(\sum_{i=0}^{N-1} |i-1 \bmod N\rangle\langle i|\right) \\ &+ |1\rangle\langle 1| \otimes \left(\sum_{i=0}^{N-1} |i+1 \bmod N\rangle\langle i|\right) \end{aligned} \tag{1}$$

Which we may write in `Q-Prog` as:

```
(new matrix Shift (integer 1) (integer 1))
(new matrix temp-mat (integer 1)(integer 1))
(set temp-mat (matrix (get N) (get N) zero))
(new integer i) (set i (integer 0))
(while (comp.lt (get i) (get N))
  (set temp-mat (add (get temp-mat)
    (mult
      (get (index Position.ket
        (mod (subtract (get i) (integer 1))
          (get N))))
      (get (index Position.bra (get i))))))
  (set i (add (get i) (integer 1)))
)
(set Shift (tensor
  (mult (get Coin.ket.0) (get Coin.bra.0))
```

```
  (get temp-mat)))
```

Technically, the above defines "half" of the shift operator; however the $|1\rangle\langle 1| \otimes (\cdots)$ portion is symmetric and we omit it here for space reasons.

The operator U acting on the coin space we define next as simply the Hadamard transform. In `Q-Prog`, this operator may be created from a matrix constructor specifying `QFT` (Quantum Fourier Transform) as an argument. If the dimension of the matrix is 2, the quantum Fourier transform is exactly the Hadamard transform.

```
(new matrix U (integer 2) (integer 2))
(set U (matrix (integer 2) (integer 2) QFT))
```

We now are ready to run the random walk. On iteration i, let the current state be $|\psi_i\rangle$ (where $|\psi_0\rangle$ is the previously defined initial state). Then, after iteration i, the state will be simply $|\psi_{i+1}\rangle = \texttt{Shift} \cdot U |\psi_i\rangle$.

```
(set i (integer 0))
(while (comp.lt (get i) (get T))
  (set psi (apply_op (get H) (integer 0)
    (get U) (get psi)))
  (set psi (mult (get Shift) (get psi)))
  (set i (add (get i) (integer 1))))
```

After this loop is run, one may measure the distribution induced by the particle's position.

```
(new array P-dist double (get N) )
(measure_distribution (get psi) (get H)
  (integer 1) P-dist)
```

After evaluating the above program, the i'th entry of the array `P-dist` will specify the probability that, upon measuring, the particle will be found on position $|i\rangle$ (this array may then be saved to a file using the `print_file` command). Figure 1 shows a graph of this distribution for $N = 250$, $T = 100$, and $j = 125$ (you will see the quantum setting is very different from the classical walk). On the paper's website, you will find the full `Q-Prog` program for running the experiment. Finally, we also embedded `Q-Prog` in a custom-made graphing application. From this we were able to see the particle's distribution evolve with time. A video is available online along with the source code (which also demonstrates the ability to add new commands to `Q-Prog`, in particular we add commands to interface with the graphing application).

Besides walks on the number line, an area of active research involves quantum walks on graphs. For instance, d-regular graphs, which are graphs where each vertex has degree d, requires a "coin" space of dimension d and a "position" space with dimension equal to the number of vertices in the graph. Again, this is where `Q-Prog`'s ability to handle non power of two dimensional spaces would be very useful.

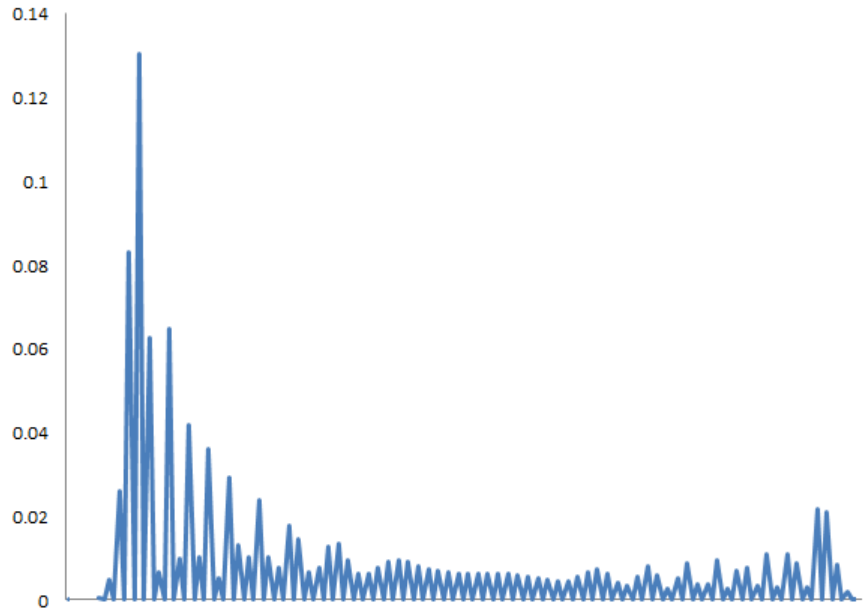

Figure 1: A graph of the distribution induced by the particle in a quantum random walk over the integers modulo 250 after 100 iterations starting with the particle at position $|0\rangle |125\rangle$, Notice that the distribution is not symmetric; this agrees with the graph in [9].

3.2 Evolving Quantum Operators

In this section we will discuss and simulate an idea presented in [8] which described a mechanism for evolving (via a genetic algorithm) a unitary operator so as to produce a desired distribution given a fixed quantum state as input. This will take advantage not only of `Q-Prog`'s ability to handle arbitrary dimensional systems, but also its ability to easily generate random unitary matrices and also its ability to be embedded in another application.

Formally, the problem is as follows. Let $\mathcal{H}_N$ be an N dimensional Hilbert space (with standard orthonormal basis $\{|i\rangle\}_{i=1}^N$). Given a quantum state $|\psi_0\rangle \in \mathcal{H}_N$ and a collection $\{c_i^*\}_{i=1}^N \subset \mathbb{C}$ where $\sum_i |c_i^*|^2 = 1$ as input, find a unitary operator U such that $|\langle i|U|\psi_0\rangle|^2 = |c_i^*|^2$ for all i. Recall, from section 2, that this notation is the dot product of $|i\rangle$ with $U|\psi_0\rangle$; since the basis vectors are orthogonal, this will be exactly the probability amplitude of $|i\rangle$ in $U|\psi_0\rangle$. That is to say, we wish to find an operator U such that after applying U and then measuring the resulting state, the probability that the state will collapse to $|i\rangle$ is exactly $|c_i^*|^2$. As an equation, after applying U to the initial state, we want:

$$U|\psi_0\rangle = \sum_{i=1}^{N} c_i |i\rangle = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{pmatrix}, \tag{2}$$

where $|c_i|^2 = |c_i^*|^2$ (the closer these two quantities, the more *fit* U is considered to be).

To generate a random unitary matrix, `Q-Prog` uses the mechanism described in [8] and [16]. With this system, one may describe a unitary matrix via N^2 doubles in the range $[0, 2\pi)$. More specifically, to construct a random unitary matrix, one first defines three arrays of doubles: `phi` and psi both of size $(N-1)N/2$, and also `chi` of size $N-1$. After setting the elements of these arrays to random values in the range $[0, 2\pi)$, one may call the matrix constructor as follows:

```
matrix (integer N) (integer N) u_random
  phi psi chi
```

Alternatively, if embedding `Q-Prog` in a C++ program, one may also create the same sized arrays as a C++ variable (actually, they should each be a `std::vector<double>`) and run the procedure `QProg::random_unitary(phi, psi, chi)`. This is the method we'll use here.

As in the previous example, we will not describe every line of code here but only mention some of the details. Full source code however is available online. We first define a C++ structure which will hold a single operator (we will then create a list of these for our genetic algorithm's population). Part of this structure is defined as follows (there are also several functions specific to genetic algorithms):

```
struct Operator {
  algebra::mat U;
  std::vector <double> phi, psi, chi;
  void apply();
  void refresh();
  void randomize();
}
```

Notice that we may define operators outside of a `Q-Prog` store simply by declaring an instance of the `algebra::mat` class. Alternatively, we may change the definition above to be `algebra::mat* U` which will then point to an operator we could define in the global store of `Q-Prog` (how to access the global store will become apparent shortly). The `randomize()` procedure will simply choose random values for the three double arrays and `refresh()` will simply call `U.random_unitary(phi, psi, chi)` to produce a unitary matrix.

Next we must instantiate the `Q-Prog` interpreter and set its global store appropriately. For the first, we simply create an instance of the `QProg` class (we call it `qp` in this example). For the second, we will run a setup script permitting a user to change the program's functionality without having to recompile the C++ program. The setup script (which we call "`setup.txt`") will initialize the value of N and setup the quantum system:

```
begin
  (new integer N)
  (set N (integer 3))
```

```
(new space Evolve (get N))
(new system H Evolve (integer 1))

(new matrix psi (get N) (integer 1))
(new matrix U (get N) (get N))

(new matrix c (get N) (integer 1))
(NOP ToDo: modify elements of c
  to be desired prob. ampl. after
  applying operator U)

(OK)
```

The vector `c` will store the values c_i^* mentioned earlier; that is to say, `c` is the desired state of the system after applying operator U. Notice also, that the value of the above program is the string "`OK`". Then we run this script in our C++ program as follows:

```
QProg::Object output;
qp.evaluate("file setup.txt",
  qp.getGlobalStore(), output, true);

if(QProg::ObjectHelper::getString(output)
    != "OK")
  return 0;
```

The first argument, `"file setup.txt"`, is the actual program to evaluate. In this case, the command `file` will tell `Q-Prog` to load and then evaluate the program specified in "setup.txt". The "`true`" in the above call to `evaluate` tells `Q-Prog` to run in "global mode". Hence any variable created via a call to `new` in `setup.txt` will be added to the global store (allowing future calls of `evaluate` access to them). The value stored in `output` is the value of the program. In this case, if all is well, it should be a string type set to the value "`OK`". The class `ObjectHelper` contains methods to extract values from an `Object` assuming you know the data type which you can learn simply by calling `output.getType()` (this returns a string specifying the data type, e.g. "integer", "string", "system", etc.).

Following this setup, our C++ program needs to know the value of N and also the vector `c`. It may access the store and retrieve this information using one of the following methods:

```
qp.evaluate("get N", qp.getGlobalStore(),
    output, false);
dim = ObjectHelper::getInteger(output);

qp.getGlobalStore().get("c", output);
c = ObjectHelper::getMatrix(output);
output.clear();
```

The first method may be used for reading values from a store, the second for both reading and writing. Evaluating a `get` command will store a copy of the value in `output`. Calling the `get()` function from the global store will retrieve the requested variable based on its name and also the pointer to its data in the provided `Object output` (hence, in this case, any changes affect the value `Q-Prog` stores also). With both methods, one may use the `ObjectHelper` to retrieve the information. Note that, alternatively, instead of returning "OK", the file `setup.txt` could have returned the matrix `c` (by having `(get c)` the last command of the `begin` block); then `output`, after the call to `evaluate`, would have a copy of `c` from which N may also be implied.

We next write a new script "prepare.txt" which will be run to initialize the state $|\psi_0\rangle$. The following script will apply the quantum Fourier transform (the `QFT` argument is passed to the matrix constructor creating the unitary operator shown in Equation 3) to $|1\rangle$, after which the state will be:

$$|\psi_0\rangle = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i j/N} |j\rangle .$$

```
begin
  (set psi (mult
    (matrix (get N) (get N) QFT)
    (get Evolve.ket.1)))
```

$$\frac{1}{\sqrt{N}} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & & & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{pmatrix}, \tag{3}$$

where $\omega = e^{2\pi i/N}$ (an N'th root of unity).

With this complete, our C++ program creates an initial population, each element of which is an instance of the `Operator` structure previously defined. To test any of the operators, we define the function `Operator::apply()` as follows (note this is not the only way to accomplish this functionality):

```
qp.evaluate("file prepare.txt",
    qp.getGlobalStore(), output, false);
qp.getGlobalStore().get("U", output);
*output.data = this->U;

qp.evaluate("apply_op (get H) (integer 0)\
    (get U) (get psi)", qp.getGlobalStore(),
    output, false);
```

```
algebra::mat psi = ObjectHelper::getMatrix(output);
```

The fitness may be computed from the C++ variables `psi` and `c` (for more information, see [8]). Then the genetic algorithm should manipulate the `phi`, `psi`, and `chi` arrays to find an appropriate operator. Note that, most of the work in this application so far is done in C++. However, while not mentioned in [8], we may use a similar technique to evolve an operator on a subspace of a larger quantum system. Here, the `apply_op` and also the `measure_distribution` `Q-Prog` commands should be used to easily retrieve the necessary information so as to compute the fitness of a particular operator. Finally, if one actually wishes to use the evolved operator in an application (requiring measurements and further manipulation of quantum states), it is beneficial to use `Q-Prog` as we have done here (as a mix of C++ and `Q-Prog` programs).

4 Related Work

There have been, of course, many quantum simulators written in the past. In this section we describe some of the systems which we feel are most similar to `Q-Prog` and how our system differs from them.

Perhaps the system closest to ours is QGame [13] (Quantum Gate and Measurement Emulator). This system (written originally in Lisp but ported later to C++) uses a similar syntax structure to our own and manipulates quantum systems in a similar fashion to our own. QGame however can only support experiments on qubits and systems of qubits (hence only powers of two); also QGame creates branch points for all measurements which might not be useful for certain applications. Finally, we remark that QGame was originally designed for experiments in genetic programming; due to the similar syntax structure, one should be able to adapt their work to use `Q-Prog` and thereby permit the evolution of quantum algorithms over non powers of two spaces.

LanQ [1] and QCL (Quantum Computation Language) [2] both permit users to write quantum programs (these use a syntax style similar to C). However they also only work with qubits. LanQ has the ability to make measurements in alternative bases a feature not yet supported by `Q-Prog` (though it is planned in the near future to support the more general POVM [10]; a feature we're not aware exists in other simulators). Finally, qLambda (see [3] and [14]) is a system with a Scheme style syntax. However, again, it only works with qubits.

There are also many code libraries supporting quantum computations (for instance libQuantum [4]). While these libraries are optimized for speed, they are not full interpreters.

5 Closing Remarks

There is still some work to be done. For the interpreter itself, it would be useful, for efficiency reasons, to first parse an entire program into an abstract syntax tree before evaluating. It would also be useful to support procedure calls and recursion. On the quantum simulation side, it would be very useful to support *Positive Operator-Valued Measures* (or POVM) [10].

Furthermore, `Q-Prog` only works currently with the standard basis for a quantum space; it would be useful to expand its functionality to support alternative bases. Despite this, however, we believe `Q-Prog` can be a valuable research (and educational) tool for multiple fields of quantum information theory.

Full source code for `Q-Prog` along with the applications discussed here (and other test scripts), may be found on the paper's website:
`http://www.walterkrawec.org/qprog`

References

[1] LanQ: `http://lanq.sourceforge.net`

[2] QCL: `http://tph.tuwien.ac.at/~oemer/qcl.html`

[3] qLambda: `http://www.het.brown.edu/people/andre/qlambda`

[4] libQuantum: `http://www.libquantum.de`

[5] D. Aharonov, A. Ambainis, J. Kempe, and U. Vazirani. Quantum walks on graphs *Proceedings of ACM Symposium on Theory of Computation (STOC '01)*, pages 50–59, July, 2001

[6] J. R. Busemeyer, Z. Wang, J.T. Townsend. Quantum Dynamics of Human Decision-Making. *Journal of Mathematical Pshychology*, vol 50 (3), pages 220–241, 2006.

[7] P. A. M. Dirac. A New Notation for Quantum Mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 35 (3), pages 416–418, 1939.

[8] S. R. Hutsell, and G.W. Greenwood Applying Evolutionary Techniques to Quantum Computing Problems. *IEEE Congress on Evolutionary Computation (CEC 2007)*, pages 4081–4085, September, 2007

[9] J. Kempe. Quantum random walks - an Introductory Overview *Contemporary Physics*, vol. 44 (4):307–327, 2003

[10] K. Kraus. States, Effects, and Operations: Fundamental Notions of Quantum Theory. *Lecture Notes in Physics* vol. 190, Berlin: Springer-Verlag, 1983

[11] H. Krovi, and T. A. Brun. Quantum walks on quotient graphs *Physical Review A*, 75, 062332, 2007

[12] M. A. Nielsen, and I. L. Chuang. Quantum Computation and Quantum Information. *Cambridge University Press*, Cambridge, MA. 2000.

[13] L. Spector. Automatic Quantum Computer Programming: A Genetic Programming Approach. *Kluwer Academic Publishers*, Boston, MA. 2004.

[14] A. van Tonder. A Lambda Calculus for Quantum Computation. *SIAM J. Comput.* 33, pages 1109–1135, 2004.

[15] V.I. Yukalov and D. Sornette Quantum Decision Theory as Quantum Theory of Measurement *Phys. Lett. A*, 372, 6867-6871, 2008.

[16] K. Zyczkowski and M. Kus. Random Unitary Matrices. *Phys. A: Math Gen.* vol. 27, pages 4235–4245, 1994