Notes for Space-Efficient Quantum Computer Simulator

Michael P. Frank (mpf@eng.fsu.edu)

Started Thursday, December 20, 2007
Version 2.1.
Last modified Sunday, April 20, 2008

1. Text for Proposal

Simulating quantum computers on ordinary classical digital hardware is useful for many purposes, particularly for testing and debugging of new quantum algorithms.

Unfortunately, nearly all of the published systems and algorithms for quantum-computer simulation suffer from a severe limitation of the size (in qubits) of quantum computers they can simulate [give citations], due to exponential growth (as a function of the number n of qubits) in the memory required for most of the commonly-used simulation techniques. This is due to the fact that traditional techniques work by storing the entire quantum state vector (wavefunction) at each point in time, and stepping it forwards in time via a direct time-domain emulation of the quantum algorithm.

The reason for the large space requirement in this approach is that the state vector itself is exponentially large in the number n of qubits, requiring storage of 2^n complex numbers. In many cases it can be compressed substantially, but in a worst-case analysis of useful quantum algorithms, even the compressed form of the state vector in the known useful encodings is exponentially large.

However, an alternative exists to this approach. For many purposes, including validation of quantum algorithms through direct testing on sample inputs, the entire state vector is never really required, since in a real quantum computer, one never sees it anyway—all that one sees is a random statistical sampling of the measurement results on the final quantum state. If one can generate outputs with the same probability distribution as the real quantum computer would, that is sufficient for purposes of

emulating the real quantum computer's observable input-output relationship. This can be done without ever storing the state vector, using a technique inspired by Feynman's path-integral formulation of quantum mechanics [1,2], in which one obtains final-state amplitudes by summing over possible sequences of concisely representable (e.g., computational basis) configurations proceeding from the initial state to a candidate final state. Expressing the difference between two successive configurations in such a sequence requires only a few bits of storage, so as long as the quantum algorithm being simulated is a time-efficient one, requiring $t << 2^n$ steps, the order-t memory required to temporarily store the trajectory currently being explored can be far smaller than what would be required to store even a single general state vector.

It has been known that this general class of simulation techniques was possible since at least 1993, when the fact of its existence was used [3] to first prove the important result **BQP** $\subseteq$ **PSPACE** which is now a well-known fact of quantum complexity theory [4]. However, this approach has rarely been implemented in practice, due most likely to the fact that the traditional algorithm is somewhat easier to understand and explain, and also because the path-integral approach appears more inefficient when it is not well optimized. But we have realized [5] that the path-integral approach turns out to be actually fairly simple to implement as well, even when a number of optimizations are applied to maximize its performance. Moreover, due to its modest memory requirement, it can easily be implemented in relatively memory-limited environments such as FPGA-based SoC (System-on-Chip) designs, which are also well suited to high performance, since execution can take place directly in (reconfigurable) hardware without suffering the code execution overhead and off-chip memory latencies that are inherent to a traditional microprocessor-based system.

The algorithm can be briefly summarized as follows: Starting from an initial state $| x \rangle$, we compute a random real marker value m in [0,1), initialize the accumulated probability of final states to $p = 0$, and then iterate through all possible final-state configurations y. For each y, we compute the quantum probability-amplitude $a = \langle x | y \rangle$ for the system to pass from the initial state to that final state, based recursively on the amplitudes $a_{y'} = \langle x | y' \rangle$ for the possible predecessors y' of y, and we accumulate the $|a|^2$ of the y's into p until $p > m$. The final configuration y at this time is our randomly-sampled final output state. The run time for this particular algorithm is of order 2^{nt}, but with further optimization, it can be reduced to order 2^t.

In this proposal, we aim to implement an optimized version of this algorithm in VHDL, test it on an FPGA development board, and demonstrate that it can replicate the correct statistical distribution of outputs for one or more standard quantum algorithms being simulated.

[1] Laurie M. Brown ed., *Feynman's Thesis: A New Approach to Quantum Theory* , World Scientific, 2005. (Original copyright date: 1942.)

[2] R. P. Feynman, "Space-time approach to non-relativistic quantum mechanics," *Rev. Mod. Phys.* **20** (1948), pp. 367-387.

[3] Bernstein, E. and U. Vazirani, Quantum complexity theory. In Proceedings of the 25th ACM Symposium on the Theory of Computation. New York: ACM Press, pp. 11-20, (1993).

[4] Michael A. Nielsen and Isaac L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2000, pp. 201-202.

[5] Michael Frank and DoRon Motter, "Quantum Computer Architectures for Physical Simulations," invited talk presented by Frank at the *Quantum Computation for Physical Modeling* workshop sponsored by the Air Force research labs, held at Martha's Vineyard, Wed., May 8, 2002. Slides in PowerPoint at http://www.cise.ufl.edu/research/revcomp/talks/QCPM-talk.ppt

2. Overall Strategy

We will first implement and test the simulator in C in a conventional OS environment, then port the code over to VHDL, possibly via a Matlab/Simulink block from which we can automatically generate VHDL code from the MDL file. This VHDL code can then be tested, first in a simulation-based testbed, and then in an embedded development-board based testbed.

3. Testbed Environment

Our overall picture of the testbed scenario is as follows.

First, a standard quantum algorithm and problem size are chosen, sample inputs are chosen, and the quantum algorithm is precompiled (possibly with help from some existing quantum computing toolkits / programming language environments that are out there) to a fairly low-level form, e.g., ANDX/ORX/NOTX plus H-gates and other required 2-bit gates. For measurements, we can have explicit measurement operations on specific qubits at different times, or else implicitly just measure the entire state at some predetermined final time point).

Next, an output modality is selected: Options for this include: (1) outputting a sequence of randomly-selected trajectories (generated in Bohm-like fashion using a tracer configuration), each of a predetermined length, (2) output a sequence of randomly-selected final output states, (3) like 2, but binning the outputs and collecting statistics on them which are periodically output, (4) similar again, but this time calculating all of the exact probabilities for all possible configurations for a subset of qubits which are to be measured.

A particularly cool option: For any selected 20-qubit subspace, the accumulated probabilities for that subspace can be projected down to a 1024x1024 pixel color display, which can be directly output to a monitor using a SVGA driver. We can watch the algorithm walk through the subspace calculating all the probabilities. When the frame is finished, we can step the display forwards in time by 1 step, and then watch the probability masses move around. (This gets exponentially slower as the simulation progresses.)

Another, super-cool animated output option: Render on the VGA display a representation of the entire quantum configuration trajectory that is currently being explored. For example, one could display the (classical) qubit values for *n* qubits (up to the number of scan lines on the display) in a single vertical column of B&W pixels. A different time-

step of the algorithm can be displayed in each column across the screen (up to the horizontal resolution of the display). Below each column of pixels could be a pixel-wide vertical line of color, intended as an annotation giving the total complex amplitude accumulated so far for that particular configuration at that point in time. At the top of the screen can be a numerical scale showing which is step 0, 10, 20, etc. of the quantum program being executed. To support longer-running quantum algorithms, the display can even be scrolled horizontally if necessary to follow the progress of the "cursor," the current point being explored along the current trajectory.

Third, the testbed is configured, to have it input the desired quantum algorithm, and produce the desired output modality.

Fourth, the testbed is run, for a predetermined period of time.

Fifth, if statistics were generated, in a data file, we can post-process this file to compare it to theoretically-derived predictions (or output from other quantum simulators) to check for correctness and accuracy.

4. Input Files

These are described as they would be set up for a C-language prototyping environment. For the VHDL implementation, the same basic information would likely be precompiled into binary-format data structures residing within ROM blocks in the design.

qoperators.txt – This text file, in a predefined file format, defines all of the quantum "gates" (which really should be called "operators") to be used in the given quantum algorithms. These can include any classical reversible gates up to say 3 qubits wide (out of the 8!=40,320 possible such gates), together with any generalized unitary gate up to say 2 qubits wide (requires specifying a matrix of 16 complex numbers, or 4 numbers for the 1-qubit gates). Other special operators, such as state-preparation and state-measurement operators, may also implicitly exist in the system and do not need to be defined here. Other interesting possibilities include "magic window" operators, that are really just flags telling the simulator to output information about the probability distribution over a particular bit or set of bits at a given point in time, but without actually simulating a collapsing measurement at that point.

qconfig.txt – This text file specifies the configuration of the quantum computer to be simulated. This includes specifying its size, in terms of its total number of qubits. Labels for the qubits can also be defined here, if desired. (Otherwise qubits can just be referenced by their numerical index, like a single flat bit-addressed memory.)

qopseq.txt – This text file, in a predetermined file format, defines the quantum circuit to be simulated, i.e., the sequence of quantum gate operations to be applied. Each operation in the sequence applies an operator to a selected set of qubits at a particular stage (time point) in the simulated sequence. This file should usually be generated automatically by some sort of compiler starting from a higher-level quantum programming language. However, for simple circuits, it can just be typed in by hand.

outmod.txt – This text file, in a predefined file format, configures the simulator's output modality. This may include specifying which output files (and/or displays) to generate, as well as specifying things such as how many sampled trajectories or measurement results to produce.

prefs.txt – This text file, in a predetermined file format, configures things like: Whether to use forward-branching or backward-branching for enumerating trajectories for accumulating the propagators. Forward-branching means we do a depth-first forward search of all trajectories starting from the known initial state. Backward-branching means a depth-first backwards search of all trajectories ending up at a given final state. Another thing to specify here would be things like the precision for representing amplitudes; fixed-point vs. floating-point, and how many bits per real value?

input.txt – The input state of the quantum algorithm. A single classical configuration of all n bits.

Additional config files may be needed to specify a classical algorithm to be wrapped around the quantum algorithm (as is done in Shor's algorithm, for example).

5. Output Files

These are described as they would be set up for a C-language prototyping environment. For the VHDL implementation, the same information would likely be output into binary-format data structures residing within RAM blocks within the design, or perhaps streamed out along some serial I/O interface.

Depending on the output modality selected, not all output files may be produced in a given run.

windows.txt – For each magic window in the program, this file shows the complete probability distribution over its possible configurations (2^s real numbers for an s-bit window).

traces.txt – This file contains a sequence of Bohmish traces, random trajectories generated consistently with the flow of probability mass, without imposing collapsing measurements in the middle of the system.

samples.txt – Contains a set of random samples for all the measurement operators in the circuit.

6. Core Data Structures

The core internal engine of the simulation algorithm requires the following data structures, in addition to others specifying miscellaneous configuration information. Parameters here are n, number of qubits; t, number of time-steps (ops); g, number of distinct gate types.

1. n bit ROM specifying the initial state
2. (lg t)-bit writable register specifying the current cursor position in time
3. n bit RAM specifying the current (cursor) state
4. For generating Bohmish traces, we also need another (lg t)-bit wide counter register specifying the location of the current endpoint in time, and another n-bit RAM specifying the configuration at the current endpoint, and a 2-word fractional register (with whatever precision we're using for amplitudes) for specifying the current amplitude.
5. $[(\lg g) + 3(\lg n)] \times t$ bit ROM specifies the sequence of operations to be applied (the quantum circuit, i.e. the dynamic execution trace of the quantum algorithm),
6. $2t$-bit writable array specifying the branching decision made for each 2-bit general-unitary operation applied at each point in time along the present trajectory. For the VGA "trajectory strip" display, it might be possible to generate the configurations themselves dynamically during the span of a dot-clock, in which case an explicit sequence of configurations need not be stored.
7. $2t$-word writable array for the amplitudes accumulated so far for the propagators to the configurations in the present trajectory. The word precision affects the size of this.
8. 1-word fractional register (with whatever precision we're using for probabilities) specifying a random coin value in [0,1) for purposes of selecting random full trajectories for purposes of output or binning of statistics. After each sampling run is completed, this should be recalculated to a new random value.
9. Additional RAMs if statistics are being binned here. If an output file or stream is generated, the binning can also be done offline.

If we want to use fixed-point arithmetic for amplitude/probability values, one wrinkle to all this is that some quantum algorithms may require $O(t)$ bits of precision in the probabilities, if the probability mass gets spread out among say 2^t intermediate states at some point midway through the algorithm. This implies that item 7 above actually needs to be order t^2 bits big, in which case it by far dominates the space requirements of the algorithm. However, this problem can be avoided by simply using floating-point instead, so this might be advisable.

7. Outline of Core Algorithm

Here is the outline of the core simulation algorithm, in a simple case with a prespecified number of steps, no Bohm-trace, and no measurement or state-preparation operators or magic windows being done in the middle of the quantum algorithm to be simulated. Also no binning – we're just sampling random outputs. This can be elaborated upon as needed to add more features.

1. Given: Initial state configuration x;
2. Generate a random "marker" value m in [0,1) – this will be used in a roulette-wheel algorithm for picking a random final state with the correct distribution;
3. Initialize accumulated total probability mass $p := 0$. This is the current "position" of the roulette wheel;
4. Enumerate all possible final-state configurations y, and for each:

a. Compute its amplitude $a = \langle x|y \rangle$ (the propagator from the initial state to the final state), by the following recursive procedure:
    i. (Base case:) If t=0, then a=1 if x=y, and a=0 otherwise, and go to step 4b; else do the following:
    ii. Generate all of y's possible immediate predecessors. There is exactly 1 of these if the immediately preceding op was classical (specified by a 0-1 matrix), and it is computed by applying that op to y in reverse; there are up to 2 (or 4) of these if the preceding op is a 1-bit (or 2-bit) general unitary op, in which case these are generated by enumerating the possible values of the bit-set to which the op is applied;
    iii. For each possible predecessor configuration having a nonzero matrix element in the operation currently being applied, recursively compute its amplitude by the same procedure,
        1. and add the predecessor's amplitude (multiplied by the appropriate matrix element, in the case of general unitary ops) into a;
b. Add $|a|^2$ into p;
c. If $p > m$, output y as the randomly-sampled final state.

The kernel here is the recursion in step 4a, which of course can be implemented iteratively by a process that steps backwards and forwards through the quantum circuit (and the arrays) as necessary.

The implementation of this algorithm should be really straightforward.

7.1. Bohmian variation

The worst-case run time of the algorithm described above is (unfortunately) $\boldsymbol{\Theta}(2^{nt})$. This is because there are 2^n possible final states, and then for each one we examine a tree of predecessor states with up to 2^t leaf nodes (given a branching factor of at most 2 at each node, if we restrict ourselves to 1-bit general unitary gates and cNOT).

However, it turns out that there is actually a simple way to reduce the run time to just $\boldsymbol{O}(2^t)$, although it takes longer to describe. This is the "Bohmian" method referred to previously. Basically, it is a kind of Monte Carlo approach, inspired by Bohm's interpretation of quantum theory. In Bohm's model, the system is always in a unique "actual" basis state at any given time, but this state evolves nondeterministically, following the flow of probability mass through configuration state, as specified by the evolution of the wavefunction.

The way this works is as follows. Suppose we know $x(t)$, the "actual" basis state at current time t. Initially, for t=0 this is just $x(0)$=x. And suppose we also have the current amplitude (propagator) $a(t) = \langle x(0) \mid x(t) \rangle$.

Then to step this information forward in time, and compute $x(t+1)$ and $a(t+1)$, we do the

following. For a classical reversible operator, we just apply it to x and leave a unchanged. For a general 1-bit unitary operator U, there are two possible next states, call them s and s', where s is identical to $x(t)$ and s' (call it the "neighbor" state) differs from it by 1 bit (the bit to which U is being applied).

What we do next is use the recursive algorithm to compute $a' = \langle x(0) \mid s'(t) \rangle$, the amplitude to go all the way from the initial state $x(0)$ to the *neighbor* state s' at time t. Given both a and a', we can now step both these amplitudes forward in time by 1 step, by just applying the 2x2 U matrix to the column vector of a and a' (appropriately ordered). Knowing these, we now generate a normalized probability distribution (p, p') over the two states s and s' at time t+1: let $z = |a|^2 + |a'|^2$; then $p = |a|^2/z$, and $p' = |a'|^2/z$. Then we nondeterministically set $x(t+1)$ to s with probability p, and to s' with probability p'. We also select either a or a' as the new value of a correspondingly.

A simple inductive proof (which we can provide in reports) demonstrates that this algorithm arrives at any given final state $y=x(t)$ at any given time t with the correct overall probability. (Proof outline: Basically, for any given 1-bit unitary gate, for any given configuration of the bits not involved in that gate, if the probability of arriving at each of the two possible input configurations is correct, and the computed amplitudes of the two possible input configurations are correct, then the computed output amplitudes will be correct, and the probability of the simulation arriving at each output configuration (at a given time) is correct.

As for its time complexity, the run time to get to step t is $O(2^t)$, but the total time for all steps is also only $O(2^t)$, due to the fact that the sum for i from 0 to t-1 of 2^i is (2^t)-1. This algorithm also has the advantage that it generates not just the final state, but an entire configuration trajectory. This can be useful for understanding the flow of the algorithm.

7.2. Pruning of the Trajectory Tree

Another optimization that will help a little bit (in the recursive calculations of amplitude values) is if we prune the backwards search tree at nodes that cannot possibly receive any amplitude from the given initial state x. This can be done by comparing the Hamming distance between x and the current node. We can prune if the Hamming distance is greater than the current node's time index t in the gate sequence. This is because each gate can at most modify 1 bit (if we restrict ourselves to 1-bit unitary gates and 2-bit cNOT), so if the Hamming distance is larger, there would not be enough gates to get to the current node from the initial state.

8. File Formats

qoperators.txt (Quantum Logic Operator Definitions)

This is a plain 7-bit US ASCII text file, with lines terminated by LF or CR/LF.

Line 1: Magic cookie string "`qoperators.txt format version 1`"

Line 2: The string "`operators: g`", where *g* is a scanf()-readable non-negative integer that fits in an unsigned short (16 bits), giving the number of distinct quantum operator types to be used in the quantum algorithm.
Line 3: The string "`operator #: i`", where *i* is formatted like *g* and is the index $0 \le i < g$ of the gate to be specified.
Line 4: The string "`name: s`", where *s* is a string terminated by end-of-line.
Line 5: The string "`size: b` bits", where *b* is 1, 2, or 3.
Line 6: The string "`matrix:`"
Lines 6 through $(5+2^b)$:
Each is a sequence of 2^b strings formatted like: "`(R + i*I) `", where *R* and *I* are scanf()-readable double-precision floating-point values.
…

and similarly for the remaining operators in the sequence. The operators need not be specified in numerical order, but each of the *g* operators (indexed 0 through $g-1$) must be specified exactly once. The rows and columns of the matrix are implicitly indexed using big-endian bit ordering; that is, for a three-bit operator, the ordering is $b_2b_1b_0$; thus, the second row and second column, counting from the upper-left corner of the matrix, correspond to the bit-value assignment $b_2=0$, $b_1=0$, $b_0=1$. Throughout the file, any optional extra lines starting with "`comment:`" are ignored.

Example qoperators.txt

```
qoperators.txt format version 1
operators: 2
comment: ----------------------------------------------------
operator #: 0
name: X
size: 1 bits
comment: In-place unary NOT, or Pauli x-axis spin operator.
matrix:
(0 + i*0) (1 + i*0)
(1 + i*0) (0 + i*0)
comment: ----------------------------------------------------
operator #: 1
name: cNOT
size: 2 bits
comment: Controlled-NOT; XOR the 1st bit into the 2nd.
matrix:
(1 + i*0) (0 + i*0) (0 + i*0) (0 + i*0)
(0 + i*0) (1 + i*0) (0 + i*0) (0 + i*0)
(0 + i*0) (0 + i*0) (0 + i*0) (1 + i*0)
(0 + i*0) (0 + i*0) (1 + i*0) (0 + i*0)
comment: ----------------------------------------------------
```

qconfig.txt (Quantum Computer Configuration)

This is a plain 7-bit US ASCII text file, with lines terminated by LF or CR/LF.

Line 1: Magic cookie string "`qconfig.txt format version 1`"

Line 2: The string "`bits:` *n*", where *n* is a scanf()-readable non-negative integer that fits in an unsigned short (16 bits), giving the number of distinct quantum bits (qubits) making up the quantum computer.

Lines 3+: Each line contains the string "`named bit:` *s* `@` *i*" where *s* is an alphanumeric identifier starting with a letter or _ and consisting of letters, digits, and _'s, and *i* is a non-negative integer less than *n* identifying the bit-index in memory of the named bit; or else it contains the string "`named bitarray:` *s*`[`*k*`]` `@` *i*" where *s* is the array name, *k* is a non-negative integer giving the number of bits in the array, *i* is a non-negative integer less than *n* identifying the bit-index in memory of the first array element, *s*[0].

There may be any number of names. A single location may have multiple names. If a name is redefined, the last definition in the file holds. The same identifier may not be used simultaneously for both a bit and a bit-array. If an array overflows memory, it wraps around.

Example qconfig.txt

```
qconfig.txt format version 1
bits: 10
named bit: a @ 0
named bit: b @ 1
named bit: c @ 2
named bit: d @ 3
named bit: e @ 4
named bit: f @ 5
named bit: g @ 6
named bit: h @ 7
named bit: i @ 8
named bit: j @ 9
named bitarray: arr[5] @ 5
comment: arr[0]-arr[4] are aliases for bits f-j
```

qopseq.txt – Quantum Operation Sequence

This is a plain 7-bit US ASCII text file, with lines terminated by LF or CR/LF.

Line 1: Magic cookie string "`qopseq.txt format version 1`"

Line 2: The string "operations: *t*," where *t* is a scanf()-readable non-negative integer that fits in an unsigned short (16 bits), giving the number of quantum operations to be dynamically executed in the given operation sequence (the length of the sequence).

Lines 3 through 2+*t*: Each contains one of the following:

1. The string "`operation #`*j*`: apply unary operator` *o* `to bit` *b*", where *j* is a nonnegative integer less than *t*, *o* is the name or index number of a 3-bit operator (as defined in qoperators.txt), and *b* is the name or index number (as defined in qconfig.txt) of a particular qubit.
2. The string "`operation #`*j*`: apply binary operator` *o* `to bits` b_1, b_0", where *o* is the name or index number of a 3-bit operator (as defined in qoperators.txt), and each b_i is the name or index number (from qconfig.txt) of a particular qubit.
3. The string "`operation #`*j*`: apply ternary operator` *o* `to bits` b_2, b_1, b_0", where *o* is the name or index number of a 3-bit operator (as defined in qoperators.txt), and each b_i is the name or index number (from qconfig.txt) of a particular qubit.
4. The string "`operation #`*j*`: measure bit` *b*" where *b* is the name or index number (as defined in qconfig.txt) of a particular qubit.
5. The string "`operation #`*j*`: measure bits` b_s-b_f" where *b* is the name or index number (as defined in qconfig.txt) of a particular qubit. All qubits in the range of memory starting at bit b_s and continuing upwards (wrapping around, if necessary) to bit b_f will be measured.

Note that the ordering of the bits within the operator is the same as that used in the matrix ordering in qoperators.txt. In each line *j* is a sequence number and the values 0 through $t-1$ should occur in the correct order.

If the name of a bit or operator is not found, or it lies outside the defined range of the quantum computer's memory, this should be reported as an error. The number of operations listed must be exactly *t*. If no measurement operations are specified, the intended semantics is that all qubits are implicitly measured simultaneously at the end of the algorithm.

Example qopseq.txt

```
qopseq.txt format version 1
operations: 4
operation #0: apply binary operator cNOT to bits a, b
operation #1: apply binary operator cNOT to bits b, c
operation #2: apply binary operator cNOT to bits c, a
operation #3: apply unary operator X to bit b
```

qinput.txt – Input configuration

This text file should be a single line, a simple sequence of 0's and 1's (in 7-bit ASCII) exactly *n* characters long, giving the initial values of bits $n-1$ down to 0 (big-endian). It may be terminated with a LF or CR/LF sequence.

Slightly more sophisticated:

Line 1: Magic cookie string "qinput.txt format version 1"

Lines 2+: Each contains one of the following:

1. The string "*b* = *v*", where *b* is the name (from qconfig.txt) or index number of a qubit or bitarray member, and *v* is a bit value, specified by one of the strings:
   a. "`0`", "`zero`", "`Zero`", "`ZERO`," "`f`", "`F`", "`false`", "`False`", or "`FALSE`"
   b. "`1`", "`one`", "`One`", "`ONE`", "`t`", "`T`", "`true`", "`True`", or "`TRUE`"
2. The string "*a* = *v*", where *a* is the name (from qconfig.txt) of a bitarray, *l* bits long, and *v* is a bit-array value, specified by one of the following means:
   a. A sequence of exactly *l* characters that are each "`0`" or "`1`".
   b. A non-negative decimal integer less than 2^l. Any commas appearing in the digit sequence are ignored.

Bits that are unset by a given input file are assumed to be initialized to (prepared as) 0. If a given bit is set multiple times, the last assignment given is the effective one, and a warning may be reported.

Example qinput.txt

```
qinput.txt format version 1
a = 1
arr = 31
```

The above input files are sufficient to define and run simple quantum algorithms.

9. C++ Data Types

Here are some C++ data types suitable for storing the information contained in the input data files (formats summarized in sec. 8) and executing the core simulation algorithm (outlined in section 7).

9.2. Operator class

```
public class Operator {
   public:                // For simplicity, let all fields be public.
      unsigned short id;  // Numerical unique ID for this operator.
      char name[];        // Zero-terminated alphanumeric identifier; the operator name.
      unsigned char arity; // The arity (number of bits) of this operator.
      Complex matrix[][];  // Two-dim. array (indexed by row, column) of cpx no's.
                           //   Note there are 2^arity rows and columns.
}
```