QuIDDPro User's Guide
version 3.8

George F. Viamontes
Héctor J. García
Igor L. Markov
John P. Hayes

updated on June 21, 2011

The University of Michigan

QuIDDPro is a fast, scalable, and easy-to-use computational interface for generic quantum circuit simulation. It supports state vectors, density matrices, and related operations using the Quantum Information Decision Diagram (QuIDD) data structure [1, 2, 4]. Software packages including Matlab, Octave, QCSim [5], and libquantum [6], have also been used to simulate quantum circuits. However, unlike these packages, QuIDDPro does not always suffer from the exponential blow-up in size of the matrices required to simulate quantum circuits. As a result, we have found that QuIDDPro is much faster and uses much less memory as compared to other generic simulation methods for some useful circuits with much more than 10 qubits [1, 2, 4].

VERSION HISTORY (SEE APPENDICES A AND B FOR DETAILS)

- VERSION 1.0 - JANUARY 27, 2004
- VERSION 1.1 - MARCH 29, 2004
- VERSION 1.2 - DECEMBER 12, 2004
- VERSION 2.0 - JULY 27, 2005
- VERSION 2.1 - AUGUST 17, 2005
- VERSION 2.1.1 - OCTOBER 25, 2005
- VERSION 2.1.2 - DECEMBER 2, 2005
- VERSION 2.1.3 - FEBRUARY 24, 2006
- VERSION 2.1.4 - FEBRUARY 27, 2006
- VERSION 2.1.5 - SEPTEMBER 15, 2006
- VERSION 3.0β - OCTOBER 16, 2006
- VERSION 3.1β - OCTOBER 31, 2006
- VERSION 3.1 - MARCH 8, 2007
- VERSION 3.5β - JULY 10, 2010
- VERSION 3.8 - JUNE 21, 2011

GEORGE F. VIAMONTES, HÉCTOR J. GARCÍA, IGOR L. MARKOV, JOHN P. HAYES
QUANTUM CIRCUITS GROUP
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF MICHIGAN
{gviamont, hjgarcia, imarkov, jhayes}@eecs.umich.edu

Contents

1 Overview

The goal of the QuIDDPro simulator is to provide a fast, scalable, and easy-to-use quantum-mechanical simulator for applications such as quantum computation and communication. This documentation and examples distributed with the simulator provide enough information for any user to get started. QuIDDPro uses an input language similar to Matlab. It supports density matrices, state vectors, and related operations using a recently developed data structure called the Quantum Information Decision Diagram (QuIDD) [1, 2, 4]. Version 3.1 introduces a number of new features including functions which efficiently compute the equivalence of states and operators up to global and relative phase, string data types, Dirac notation, and bit operations (see Sections 4 and 5 for a more detailed discussion of the new features). Versions 1.2 and later are up to 60 times faster than the older version 1.1 (see Appendix B for performance results).

Software packages, including Matlab, Octave (a free GNU package that is very similar to Matlab), and others [5, 6, 7], have been used to simulate quantum circuits and communication. Unlike existing packages, QuIDDPro does not always suffer from the exponential blow-up in size of the matrices required to simulate quantum circuits. As a result, we have found that QuIDDPro is much faster and uses much less memory as compared to other generic simulators [1, 2, 4] for many useful circuits with 10 qubits or more. For example, QuIDDPro can simulate a randomly generated 100-qubit stabilizer circuit with 10000 gates in approximately nine minutes and has a peak memory usage of 7.4MB. However, other circuits, including instances of Shor's number factoring algorithm, still require a substantial amount of computational resources.

The remainder of this documentation is organized as follows. Section 2 explains how to run the simulator and provides examples. Section 3 describes how to create user-defined functions and import code from multiple files. Section 4 shows how to use the optional Dirac notation support and string manipulation functionality using examples of several well-known quantum states to illustrate the relevant QuIDDPro features. Section 5 demonstrates efficient functions which can determine the equivalence of states and operators up to global and relative phase and compute the phases. Section 8 contains a language reference, and lastly, Section 9 outlines the ongoing work in developing the QuIDDPro simulator and related features. Appendix A lists the major features which have been added to the latest version of QuIDDPro. Appendix B describes major performance improvements which make versions 1.2 and later significantly faster than version 1.1. Appendix C summarizes important guidelines for evaluating the runtime and memory performance of the QuIDDPro simulator. Since QuIDDPro is under active development, we welcome user feedback and contributed examples/circuits in the form of QuIDDPro scripts. Scripts contributed by users will drive future performance improvements in the computational back-end of QuIDDPro. Although the current scope of QuIDDPro is simulation, the development of other tools to work in concert with QuIDDPro, such as a quantum circuit synthesizer, is also under consideration.

2 Running the Simulator

The QuIDDPro simulator can be run in two modes, namely batch mode and interactive mode. In batch mode, the user supplies the simulator with an ASCII text file containing the script code to be executed. The text file can be provided as an argument in the command line to the simulator executable or redirected to standard input as in the following examples:

File "my_code.qpro" passed as an argument:

```
% ./qp my_code.qpro
```

File “my_code.qpro” redirected to standard input:

```
% ./qp < my_code.qpro
```

Note that although the examples use a “.qpro” extension in the filenames, any valid filename will do.

Interactive mode is triggered when the simulator executable is given no arguments at the command line. In this mode, the simulator will be started and produce a prompt to await input from the user as shown in the next example:

```
% ./qp
QuIDDPro>
```

Similar to Matlab, valid lines of code may be typed at the prompt and executed when the return or enter key is pressed (i.e. when a newline is given as input). The command “quit” can be issued to exit the simulator. Also, multiple expressions may be placed in a single line by separating each expression by one or more semicolons. An example of this method of input is as follows:

```
QuIDDPro> a = pi/3; r_op = [cos(a/2) -i*sin(a/2); -i*sin(a/2) cos(a/2)]
r_op =
0.866025 0-0.5i
0-0.5i 0.866025
```

In this example, a 1-qubit rotational X operator matrix is created with the θ parameter $\pi/3$. Notice that only the value of the variable “r_op” is printed out. In general, the value of the last expression is printed out for an input line containing multiple expressions separated by semicolons. However, the other expressions are still computed. In this example, for instance, the variable “a” will contain the value $pi/3$, even though this result is not printed out. This is clearly true since the definition of “r_op” depends on the value of “a.” In addition to providing the means to place multiple expressions on the same line, semicolons can be used more generally to suppress output to the screen. If screen output for any particular expression is not desired, simply place a semicolon at the end of the expression to compute it silently. Matlab behaves in the same fashion.

QuIDDPro contains a number of built-in functions and predefined variables. A listing of such functions and variables can be found in Section 8. Notice that in the last example, the predefined variables “pi” and “i” are used. “pi” contains the value π (to a large number of digits), while “i” contains the value $0+i$. Predefined variables can be overwritten by the user. In addition to the predefined variables just mentioned, the built-in functions “cos” and “sin” were also used in the last example. To demonstrate the use of built-in functions further, consider the next example:

```
QuIDDPro> r_op = rx(pi/3, 1)
r_op =
0.866025 0-0.5i
0-0.5i 0.866025
```

In this example, the built-in function “rx” is used to create the same matrix that was created in the previous example, namely the 1-qubit rotational X operator. QuIDDPro provides a number of such

functions to create commonly used operators. See Section 8 for more details.

Although interactive mode is useful for quick calculations, it may not be preferable for non-trivial pieces of code that are reused many times. Thus, batch mode is highly recommended for most contexts. In the next example, we demonstrate how to use QuIDDPro to simulate a quantum circuit in batch mode. The code shown here can be placed into a file for execution at any time. In fact, this particular example and others can be found in the examples/ directory.

Consider the canonical decomposition of a two-qubit unitary operator U described in [10]. U can be expressed as:

$$U = (A_1 \otimes B_1) e^{i(\theta_x X \otimes X + \theta_y Y \otimes Y + \theta_z Z \otimes Z)} (A_2 \otimes B_2)$$

subject to the constraint that $\frac{\pi}{4} \geq \theta_x \geq \theta_y \geq |\theta_z|$ and A_1, A_2, B_1, and B_2 are one-qubit unitary operators.

Suppose we wish to simulate a quantum circuit in which some two-qubit unitary operator U is to be applied to two qubits in the density matrix state $|10\rangle\langle 10|$. Further suppose that U must be computed given the canonical decomposition parameters $\theta_x = 0.702$, $\theta_y = 0.54$, and $\theta_z = 0.2346$. Additionally, we are given that A_1 is a one-qubit Hadamard operator, A_2 is X, B_1 is I, and B_2 is Y. This can be implemented with the following code (from examples/misc/two_q_canonical.qpro):

```
theta_x = 0.702;
theta_y = 0.54;
theta_z = 0.2346;
A1 = hadamard(1);
A2 = sigma_x(1);
B1 = identity(1);
B2 = sigma_y(1);
```

Next, U can be computed with the code:

```
Xpart = theta_x*kron(sigma_x(1), sigma_x(1));
Ypart = theta_y*kron(sigma_y(1), sigma_y(1));
Zpart = theta_z*kron(sigma_z(1), sigma_z(1));
U = kron(A1, B1)*expm(i*(Xpart + Ypart + Zpart))*kron(A2, B2)
```

U is then applied to the density matrix state $|10\rangle\langle 10|$ with the code:

```
state = cb(''10'');
final_state = U*(state*state')*U'
```

Deterministic measurement can be performed to eliminate the correlations associated with each qubit:

```
q_index = 1;
while (q_index < 3)
   final_state = measure(q_index, final_state);
   q_index = q_index + 1;
end
```

```
measured_state = final_state
```

U can also be applied very easily to the state vector representation of the state if it is preferred to the density matrix representation. In addition, the probability of measuring a 1 or 0 for any qubit in the state vector can be computed using other measurement functions:

```
final_state_v = U*state
p0_qubit1 = measure_sv0(1, final_state_v)
p1_qubit1 = measure_sv1(1, final_state_v)
p0_qubit2 = measure_sv0(2, final_state_v)
p1_qubit2 = measure_sv1(2, final_state_v)
```

Probabilistic measurement can also be performed on both density matrices and state vectors. See `pmeasure` and `pmeasure_sv` in Section 8 for more details.

Upon execution of the above script, the output is:

```
U =
-0.110927-0.0265116i  -0.0530448-0.222078i -0.650863+0.15556i 0.162218-0.678733i
-0.162218+0.678733i   0.650863-0.15556i   0.0530448+0.222078i 0.110927+0.0265116i
-0.110927-0.0265116i  0.0530448+0.222078i 0.650863-0.15556i   0.162218-0.678733i
0.162218-0.678733i    0.650863-0.15556i   0.0530448+0.222078i -0.110927-0.0265116i

final_state =
0.447822   2.15483e-05+0.152794i -0.447822    2.15483e-05+0.152794i
2.15483e-05-0.152794i 0.0521324  -2.15483e-05+0.152794i   0.0521324
-0.447822 -2.15483e-05-0.152794i  0.447822   -2.15483e-05-0.152794i
2.15483e-05-0.152794i 0.0521324  -2.15483e-05+0.152794i   0.0521324

measured_state =
 0.447822      0          0          0
     0     0.0521324      0          0
     0          0      0.447822      0
     0          0          0      0.0521324

final_state_v =
-0.650863+0.15556i
0.0530448+0.222078i
0.650863-0.15556i
0.0530448+0.222078i

p0_qubit1 =
0.499955

p1_qubit1 =
0.499955
```

```
p0_qubit2 =
0.895644

p1_qubit2 =
0.104265
```

Although the examples in this section demonstrate scripts that use small numbers of qubits, the real power of QuIDDPro lies in simulating quantum-mechanical systems with many quantum states (usually 10 or more). See steaneX.qpro, steaneZ.qpro, and large_h.qpro in the examples/ directory for examples of such systems. large_h.qpro, for instance, applies a 50 qubit Hadamard operator to a density matrix of 50 qubits. steaneX.qpro and steaneZ.qpro demonstrate error correction in quantum circuits of 12 and 13 qubits, respectively. On a single processor of one of our workstations, each of these scripts requires less than 5 seconds to run and less than 0.5 MB of peak memory usage.

3 Functions and Code in Multiple Files

QuIDDPro supports user-defined functions via the "m-file" model commonly used in Matlab. Specifically, a function call to a user-defined function may appear anywhere as long as the function body is contained in a separate file in the working directory. The name of the file containing the function body must be the same as the function name with ".qpro" or ".qp" appended. To illustrate, consider the following script which uses an oracle function to implement a simple instance of Grover's algorithm shown on page 256 of [9].

(*examples/functions/simple_grover.qpro*)
```
state = cb(''001'');
state = hadamard(3)*state;
state = oracle(state);
state =* cu_gate(hadamard(1), ''xxi'');
# Note:  The =* operation is shorthand for
# state = cu_gate(hadamard(1), ''xxi'')*state;
state =* cu_gate(sigma_x(1), ''xxi'');
state =* cu_gate(hadamard(1), ''ixi'');
state =* cu_gate(sigma_x(1), ''cxi'');
state =* cu_gate(hadamard(1), ''ixi'');
state =* cu_gate(sigma_x(1), ''xxi'');
state =* hadamard(3)
```

(*examples/functions/oracle.qpro*)
```
function new_state = oracle(curr_state)
new_state = cu_gate(sigma_x(1), ''ccx'')*curr_state;
```

The user-defined function is "oracle" with its function body defined in the file "oracle.qpro." The other functions used are part of the QuIDDPro language (see Section 8 for more details). Notice that in this particular example, the QuIDD (matrix) "state" is passed as a function argument. In QuIDDPro, a QuIDD function argument only requires $O(1)$ memory usage because a pointer to the head of the

QuIDD is passed to a function rather than the entire QuIDD. The same holds true for returning QuIDDs from a function. Thus, passing QuIDD arguments and return values is extremely efficient. In general, a user-defined function can contain any number of parameters which can be any combination of QuIDDs or complex numbers. Arguments passed as parameters to functions are not modified by the function (i.e. pass-by-value is always used).

Unlike Matlab, QuIDDPro functions must have only one return variable (a function that returns nothing is also not allowed). If the function is intended to return no values, such as a diagnostic printing function, then a dummy variable can be used for the return variable. The return variable need not be used in the function body, and when this occurs, it is automatically assigned a value of 0. A semicolon can be appended to the function call to suppress the output of the 0 value. When multiple return values are desired, they can be stored together in a matrix. Thus, requiring a single return variable does not actually restrict the number of values that can be returned.

Like Matlab and other languages, variables declared locally in a function body exist in their own scope. In other words, variables declared in a function body are undefined upon leaving the function body. By the same token, such variables do not overwrite the values of variables with the same name declared outside the function body.

In addition to functions, QuIDDPro supports the *run* command. Like its Matlab counterpart, this command runs script code contained in another file. In the following example, the same circuit as before is simulated, but this time the run command is used instead of a user-defined function.

(*examples/run/simple_grover.qpro*)
```
run ‘‘oracle_def.qpro’’
state = cb(‘‘001’’);
state =* hadamard(3);
state =* oracle;
state =* cu_gate(hadamard(1), ‘‘xxi’’);
state =* cu_gate(sigma_x(1), ‘‘xxi’’);
state =* cu_gate(hadamard(1), ‘‘ixi’’);
state =* cu_gate(sigma_x(1), ‘‘cxi’’);
state =* cu_gate(hadamard(1), ‘‘ixi’’);
state =* cu_gate(sigma_x(1), ‘‘xxi’’);
state =* hadamard(3)
```

(*examples/run/oracle_def.qpro*)
```
oracle = cu_gate(sigma_x(1), ‘‘ccx’’);
```

Notice that the run command does not introduce a new scope. All variables declared in a run file exist in the current scope. As a result, the run command is ideal for declaring variables which can be re-used in multiple projects. Also, there is no constraint on where a run command may appear other than that it may not be placed within an explicit matrix.

4 Dirac Notation and String Manipulation

As of version 3.1, QuIDDPro now supports a Dirac-style syntax as well as string data types that can be stored and manipulated. This section demonstrates how to use this functionality by presenting simple examples of well-known quantum states implemented with these features. The states include the cat

(GHZ) state, the W state, and the equal superposition state.

The cat state is an n-qubit generalization of the EPR pair and is defined as $|\psi_{\text{cat}}\rangle = (|00\ldots0\rangle + |11\ldots1\rangle)/\sqrt{2}$. A QuIDDPro function which creates this state given the number of qubits n is listed below (from examples/states/create_cat_state.qpro).

```
function |cs:> = create_cat_state(n)
   |cs:> = (|0:>_n + |2^n - 1:>)/sqrt(2);
```

There are two important points to note in this example. First, the QuIDDPro form of a ket utilizes the following syntax, `|x:>`, where `x` can be an integer expression, a state vector variable, or a string expression of the form used in the `cb(·)` function (see Section 8 for a description of this function). In the case of a state vector variable, the ket form is merely an alias for the Matlab-style variable name and can be used to store expressions as illustrated by `|cs:>` above. In the case of an integer, the state becomes the binary representation in qubits of the integer, where the left-most qubit in the state is the most-significant bit of the integer. The number of qubits in the resulting state vector QuIDD is the minimum number of qubits required to represent the integer in binary. QuIDDPro uses similar syntax for a bra, `<:x|`, and support for writing inner and outer products using bras and kets is discussed later in the section. The only difference is that kets may be assigned to, but bras may not.

The second point to note is that an optional subscript can be appended to the ket as follows, `|x:>_y`, where `y` can be any integer expression. The subscript adds leading $|0\rangle$ qubits to the resulting QuIDD state vector. This functionality is useful to make states with integer expressions that need fewer qubits have the same number of qubits as states with larger integer expressions. It is good to get into the habit of putting parentheses around subscript expressions, since only integer literals and variables may be subscript expressions without parentheses.

The next state created using these features is the W state, which is defined as $|\psi_{\text{W}}\rangle = (|10\ldots0\rangle + |01\ldots0\rangle + |00\ldots1\rangle)/\sqrt{n}$. A QuIDDPro function which creates this state given the number of qubits n is given below (from examples/states/create_w_state.qpro).

```
function |ws:> = create_w_state(n)
   |ws:> = |1:>_n;
   j = 1;
   while (j < n)
      |ws:> += |2^j:>;
      j++;
   end
   |ws:> /= sqrt(n);
```

Another interesting feature used in this example is the `+=` and `/=` operators. As in other languages like C and C++, `x op= y` is merely convenient shorthand for `x = x op y`. All the basic arithmetic operations support this notation, including `+`, `-`, `*`, `/`, `<<`, and `>>`, where the last two operators are left and right bit shift, respectively. Also, the `++` and `--` operators are supported, which are equivalent to `+= 1` and `-= 1`, respectively.

The next example uses the equal superposition state to demonstrate the string manipulation functionality. An n-qubit equal superposition state represents all possible 2^n measurement outcomes with equal probability. It is defined as $\frac{1}{\sqrt{2^n}}\sum_{j=0}^{2^n-1}|j\rangle$ and can be created with Hadamard gates. To this state, controlled-Z gates are applied such that each odd numbered qubit i is a control and each even numbered

qubit $i+1$ is the corresponding target. A QuIDDPro function which generates this state given the number of qubits n is provided below (from examples/states/create_equal_even_odd.qpro). For odd values of n, the last qubit is skipped.

```
function |eeo:> = create_equal_even_odd(n)
   |eeo:> = H(n)*|0:>_n;
   count = 1;
   while (count < n)
      gate_spec = ''c'' + count + ''x'' + (count + 1);
      |eeo:> =* cu_gate(sigma_z(1), gate_spec, n);
      count += 2;
   end
```

Notice that the string `gate_spec` can be constructed from pieces of strings and numbers via the `+` operation. In QuIDDPro, this operation simply appends to a string and automatically converts numeric values to strings. It is important to note the difference between appending `(count + 1)` and `count + 1`. In the first case, if `count` contains the value 3, then the string representation of 4 will be appended. However, in the second case the string 31 will be appended since the `+` operator will be treated as another string append rather than as numerical addition.

The final example in this section demonstrates other types of supported Dirac notation for the inner and outer products. It is often convenient, for instance, to apply a projector to a state when writing an expression for the probability of a measurement outcome such as $\langle\psi|0\rangle\langle 0|\psi\rangle$, where $|\psi\rangle$ is some 1-qubit state. Although QuIDDPro supports several optimized measurement functions, support exists to write such expressions directly as shown in this example (from examples/states/projector.qpro)

```
|psi:> = [0.8; 0.6];
p_0 = <:psi|0:><:0|psi:>
p_1 = <:psi|1:><:1|psi:>
```

As illustrated in the above code, inner products are only separated by a single `|` just as in the Dirac notation. There is no limit to the number of inner and outer products that can be concatenated in this way. Implicit multiplication of a ket by an operator, as in `Op|psi:>`, is not yet supported and requires an explicit multiplication sign, `Op*|psi:>`.

5 Checking Equivalence up to Global and Relative Phase

In addition to simulation, a number of research efforts are focused on classical synthesis of quantum circuits [11, 12, 13]. Checking the equivalence of digital circuits is a major part of classical synthesis and verification and is likely to continue to play a role in the quantum case. Equivalence checking of quantum states, operators, and circuits is more challenging since they can differ by global and relative phase yet be equivalent upon measurement. As a result, QuIDDPro provides several QuIDD-based functions which efficiently compute equivalence up to global or relative phase for both states and operators.

One such function is `gp_equals` which returns a 2-element row vector containing a 1 (0) if the two states or operators passed as arguments are (not) equal up to global phase and, if so, the global phase factor. The following example illustrates the use of this function to compare a state generated in an instance of Grover's algorithm to the same state differing by a global phase factor.

(from examples/phase_checks/gp_grover_state.qpro).

```
ckt_size = 500;

# Create an oracle that marks the last element in the ‘‘database.’’
# The oracle uses one ancillary qubit that is flipped to mark
# the last element.
count = 0;
spec = ‘‘’’;
while (count < ckt_size - 1)
   spec += ‘‘c’’;
   count++;
end
spec += ‘‘x’’;
oracle = cu_gate(sigma_x(1), spec);

# Construct the gate operators used in a Grover iteration.
hn = H(ckt_size);
hni = H(ckt_size - 1) (X) identity(1);
cps_op = cps(ckt_size - 1) (X) identity(1);

# Create the state
state = |0:>;
count = 0;
while (count < ckt_size - 2)
   state = state (X) |0:>;
   count++;
end
state = state (X) |1:>;

# Apply one Grover iteration to an equal superposition.
state =* hn;
state =* oracle;
state =* hni;
state =* cps_op;
state =* hni;
state =* (identity(ckt_size - 1) (X) H(1));

# Create a state that differs by a global phase.
gp_state = exp(i*0.784)*state;

# Compute equality up to global phase using gp_equals.
gp_info = gp_equals(gp_state, state)
```

There are several points to note in the above example. First, note the alternate syntax for constructing an n-qubit Hadamard gate using `H(n)`. Similarly, note the operator form of the `kron` function, where `a (X) b == kron(a, b)`. This alternate syntax makes QuIDDPro development seem more natural and closer to Dirac expressions. Second, although `gp_equals` is used on state vectors in this example, matrices representing operators may also be passed to the function. Third, the order in which arguments are passed to the function can affect the return value of the phase, but not the value which represents if the two arguments are equal up to global phase. Specifically, if $|\varphi\rangle = e^{i\theta}|\psi\rangle$, where θ is some real number (i.e. both states are equal up to global phase), then `gp_equals` will return a phase factor of $e^{i\theta}$ if $|\varphi\rangle$ is the first argument and will return $\frac{1}{e^{i\theta}}$ if $|\psi\rangle$ is the first argument. In either case the result of equivalence up to global phase will be a 1 (or true). Lastly, although the above example contains 500 qubits, it runs in just over one second on an Intel Xeon workstation due to the efficiency of the QuIDD datastructure and equivalence-checking algorithm.

Several QuIDDPro functions exist to efficiently compute equivalence up to relative phase. In the case of relative phase, operators or state vectors will differ by a matrix of phase factors along the diagonal, and this matrix may appear on the left, right, or on both sides. Two QuIDDPro functions that can compute these relative phases given two operators or state vectors are `rp_equals_op` and `rp_equals_sv`, respectively. Like `gp_equals`, these functions take two arguments to be compared for equivalence up to relative phase. If the two arguments are indeed equivalent up to relative phase, a vector containing the diagonal phases is returned. If they are not, then a vector of 0's is returned. A sample program is given below which compares a Hamiltonian consisting of Pauli operators against another such Hamiltonian at a different time step. Such Hamiltonians are equivalent up to relative phase (from examples/phase_checks/hamiltonian.qpro).

```
ckt_size = 70; qp_epsilon = 1e-7;
# Construct the gate representing the Hamiltonian.
cnot_diag = cnot(''cx'');
if (ckt_size > 2)
   cnot_diag = identity(ckt_size - 2) (X) cnot_diag;
end
count = 0;
while (count < ckt_size - 2)
   c_part = proj0(1) (X) identity(count + 2);
   t_part = proj1(1) (X) identity(count + 1) (X) sigma_x(1);
   curr_cnot = c_part + t_part;
   if (count < ckt_size - 3)
      curr_cnot = identity(ckt_size - count - 3) (X) curr_cnot;
   end
   cnot_diag *= curr_cnot;
   count++;
end
op = cnot_diag*(identity(ckt_size - 1) (X) expm(-i*0.3*sigma_z(1)))*
↪ cnot_diag';

# Create a version of the gate that differs by relative phase.
op_rp = cnot_diag*(identity(ckt_size - 1) (X) expm(-i*0.72*sigma_z(1)))*
↪ cnot_diag';
```

```
# Compute equality up to relative phase using rp_equals_op.
rps = rp_equals_op(op, op_rp);
```

Another useful function that can compute a necessary but not sufficient for two state vectors or operators to be equal up to relative phase is `one_merge`. It creates a new QuIDD matrix or vector in which all the non-zero values of the given argument are set to 1. It is proven in [14] that two QuIDD operators or state vectors that are equal up to relative phase are exactly equal when their non-zero terminals are merged into a single terminal with the value 1. However if the transformed operators or state vectors are not equal, then this does not necessarily mean that they are not equal up to relative phase. It is also shown in [14] how to use the complex modulus operation and inner (matrix) product on state vectors (operators) to compute global and relative phase equivalences. Thus, QuIDDPro also supports the complex modulus operation in the form of the `abs` function.

6 Compiling Quantum Circuits from QuIDDPro Scripts

As of version 3.5, QuIDDPro features a *compile* batch-mode option that maps a high-level specification of a quantum algorithm (in the QuIDDPro language) into an intermediate representation (IR) based on the quantum circuit model. The compiled circuit is specified in the *UMICH QuCirc* file format. QuCirc is a compact data structure for quantum IRs available in the UMICH Quantum Circuit Toolbox package (for details contact Héctor J. García–hjgarcia@umich.edu). If you received QuIDDPro via the UM Quantum Circuit Toolbox tarball, the open-source code for using QuCirc is available to you in the `qc-0.9-i386` directory.

To compile a QuCirc file from a QuIDDPro script simply add the `-c` option when running QuIDDPro in batch mode as follows.

```
% ./qp -c myscript.qpro
```

The quantum compiler or *q-compiler* will generate a QuCirc file for each state vector that is created/manipulated in the QuIDDPro script. The current version of the q-compiler does not support density matrices. Any QuIDDPro commands or functions that do not involve a state vector will be ignored by the q-compiler. User-defined functions are supported as long as the function returns a state vector. This is because the scope of user-defined functions is local and therefore any circuit-related changes will not propagate to the calling script unless a state vector is returned.

7 Improved Simulation of Quantum Circuits

The current version of QuIDDPro (3.8) features an *improved simulator for quantum circuits called QuIDDProLite*. If you intend to simulate a stand-alone quantum circuit (rather than a full-fledge quantum algorithm with interacting quantum and classical components), you should first compile the circuit into a QuCirc file (Section 6) and then simulate the generated circuit using QuIDDProLite by adding the `-s` batch mode option. For example,

```
% ./qp -c myscript.qpro (generates myscript.qct file)
% ./qp -s myscript.qct (calls QuIDDProLite)
```

Alternatively, you can perform the above actions with a single call to QuIDDPro as follows.

```
% ./qp -cs myscript.qpro
```

QuIDDProLite is not asymptotically faster than the native QuIDDPro simulator, but runs much faster and consumes less memory when simulating many practical quantum circuits such as stabilizer circuits [16]. Figure 1 compares the performance of both simulators on random n-qubit stabilizer circuits with $n \lg n$ stabilizer gates and n measurements. QuIDDProLite runs an average of four times faster than the native QuIDDPro simulator.

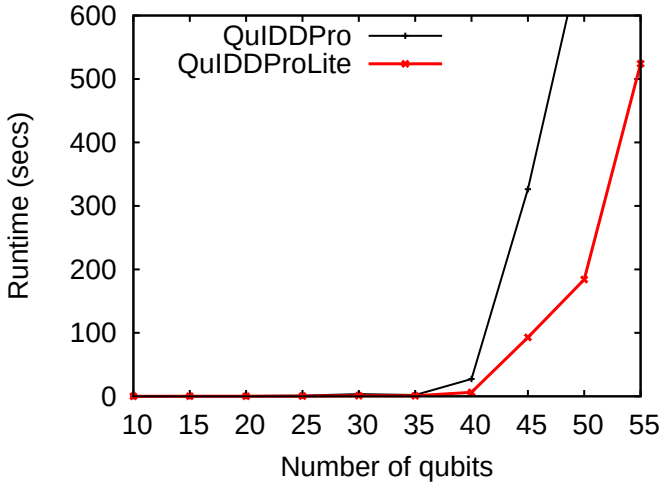

Figure 1: Average time needed by QuIDDPro and QuIDDProLite (precompiled circuit simulator) to simulate and n-qubit stabilizer circuit with $n \lg n$ unitary gates and n measurements.

Two additional options are available when using QuIDDProLite. The first is the `-sN` option, where `N` is an integer that designates the initial basis state. If the circuit to be simulated acts on n qubits, the range of `N` is $[0, 2^n - 1]$. If `N` is not specified (the `-s` option shown previously), the simulator assumes `N = 0`, i.e., it assumes the all-zeros initial basis state $|00\ldots0\rangle$. The second option is `-q`, which outputs the resulting state vector to the terminal at the end of the simulation.

8 Language Reference

This section provides a reference for the QuIDDPro input language. Although the language is similar to Matlab, there are many functions in QuIDDPro specific to quantum mechanics which do not exist in Matlab. There are also a large number of functions in Matlab which are not supported by QuIDDPro. Additionally, some of the functions that have the same names as those in Matlab have slightly different functionality from their Matlab counterparts. New language features will be added in future versions of the QuIDDPro simulator, and we welcome user suggestions. The new features as of version 3.8 are highlighted below in **bold** text.

==	~=, !=	<	<=
>	>=	&&	||
+	+=	++	-
-=	- -	*	*=
/	/=	=	=*
(...)	∧	<<	<<=
>>	>>=	(X)	′

Operations

cutoff_val	i
output_prec	pi
qp_epsilon	r2
r3	

Predefined variables

[...]	;	a(n,k)
|a:>_n	<:a|_n	a{n}
a($n_1,n_2,n_3,\ldots$)	bn	else
elseif	function	if
run	tic	toc
while	end	

Language features

abs	atan	cb	cnot
conj	cos	cps	cu_gate
dump_dot	echo	exp	expm
eye	fredkin	gen_amp_damp	get_bit
gp_equals	hadamard, H	identity	kron
norm	measure	measure_sv	measure_sv0
measure_sv1	one_merge	pmeasure	pmeasure_norm_sv
pmeasure_sv	proj0	proj1	projplus
ptrace	px, Px	py, Py	pz, Pz
quidd_info	rand	round	rp_equals_op
rp_equals_sv	rx, Rx	ry, Ry	rz, Rz
set_bit	sigma_x	sigma_y	sigma_z
sin	sqrt	swap	toffoli
zeros	**phase**		

Built-in Functions

- [...] defines a matrix explicitly. Expressions are placed between the brackets. Elements in the same row are separated by whitespace (including newlines) or commas, while rows are separated by one or more semicolons. The brackets can be nested within other brackets (matrices within matrices).
- # starts a one-line comment. Everything from the # symbol to the first newline is ignored. An alternative comment symbol is %.
- % starts a one-line comment. Everything from the % symbol to the first newline is ignored. An alternative comment symbol is #.
- ′ returns the complex-conjugate transpose of a matrix. For example, $[1\ 2;\ 3+2i\ 4]' \rightarrow [1\ 3-2i;\ 2\ 4]$
- == equality operation that returns 1 if the two expressions compared are equal; otherwise it returns 0. Comparison between matrices is supported. A complex number and a matrix are considered not equal unless the matrix has dimensions 1×1 and contains a number equal to the one being compared to.
- ~= inequality operation that performs the complement function of ==.
- != an alternative symbol for ~=.
- < less than operation. It returns 1 if the left-hand expression is less than the right-hand express; otherwise it returns 0. It can only be used to compare numbers. For numbers with nonzero imaginary components, only the real parts are compared.

- $<=$ less than or equal operation. It returns 1 if the left-hand expression is less than or equal to the right-hand express; otherwise it returns 0. It can only be used to compare numbers. For numbers with nonzero imaginary components, only the real parts are compared.
- $>$ greater than operation. It returns 1 if the left-hand expression is greater than the right-hand express; otherwise it returns 0. It can only be used to compare numbers. For numbers with nonzero imaginary components, only the real parts are compared.
- $>=$ greater than or equal operation. It returns 1 if the left-hand expression is greater than or equal to the right-hand express; otherwise it returns 0. It can only be used to compare numbers. For numbers with nonzero imaginary components, only the real parts are compared.
- && logical AND connective. It returns 1 if both sides of the operator evaluate to 1; otherwise it returns 0. It can only be used to compare numbers with nonzero imaginary components.
- || logical OR connective. It returns 1 if either side of the operator evaluates to 1; otherwise it returns 0. It can only be used to compare numbers with nonzero imaginary components.
- `+`, `+=`, `++` addition operation. For complex numbers, it returns the sum of the numbers. For matrices, it returns the element-wise addition of both matrices (both matrices must have the same number of rows and columns). When a matrix is added to a complex number, the complex number is added to each element of the matrix as a scalar. The `+=` form adds the right-hand expression to the left-hand variable and saves the result in that variable. `++` is equivalent to `+= 1`.
- `-`, `-=`, `- -` subtraction operation. For complex numbers, it returns the difference of the numbers. For matrices, it returns the element-wise difference of both matrices (both matrices must have the same number of rows and columns). When a matrix is subtracted from a complex number or vice-versa, scalar subtraction is performed element-by-element. When there is no left-hand expression, it is treated as a unary minus applied to the right-hand side expression. Within a matrix definition, for example $[1-2]$, the minus sign is treated as a unary minus. However, in $[1-2]$ and $[1-2]$, the minus sign is treated as the binary minus expression. Parenthesis can be used to force the minus sign to be treated one way or the other. The `-=` form subtracts the right-hand expression from the left-hand variable and saves the result in that variable. `--` is equivalent to `-= 1`.
- `*`, `*=` , `=*` multiplication operation. For complex numbers, it returns the product of the numbers. For matrices, matrix multiplication is performed (as opposed to element-wise multiplication). Scalar multiplication is performed when a matrix and a complex number are multiplied together. The `*=` form multiplies the right-hand expression with the left-hand variable and saves the result in that variable. The `=*` form performs left-hand multiplication and saves the result in that variable.
- `/`, `/=` division operation. For complex numbers, it returns the division of the numbers. Unlike the C language, integer division is *not* performed if the operands are both integer values. Double floating point division is always performed. For matrices, element-wise division is performed (both matrices must have the same number of rows and columns). When a matrix is divided by a complex number, scalar division is performed. However, a complex number may not be divided by a matrix. The `/=` form divides the left-hand variable by the right-hand expression and saves the result in that variable.
- $=$ assignment operation. Assigns the value of an expression (right-hand side) to a variable (left-hand side). The expression may result in either a complex number or a matrix. The left-hand side expression must be a variable name (it must start with a letter and contain only alpha-numeric characters and optionally underscores). Variables can be assigned "on-the-fly." In other words, unlike languages like C/C++, variables are not declared nor typed in any way prior to their first

assignment. However, a variable must be assigned a value before it can be used in an expression. Similar to languages such as C/C++, an assignment expression returns a value just like any other expression, namely the value that was assigned to the variable on the left-hand side. Therefore, statements such as $x = y = 3 + 4i$ are valid. In statements like these, if output is not suppressed, the value of the leftmost variable will be output to the screen. Although the other variables assigned values will not be output to the screen, they are still assigned their values. Another important note is that even though string literals appear as arguments in some functions, including *cu_gate* and *echo*, assignment of a string literal to a variable is not yet supported.

- $\wedge$ exponentiation operation for complex numbers. It returns the expression on the left-hand side of the $\wedge$ raised to the power of the expression on the right-hand side. For matrix exponentiation, see the `expm` function.
- $<<$, $<<=$ performs a left bit shift on the left-hand expression or variable. The right-hand expression must be a non-negative integer which specifies the number of times to shift the bits to the left. The left-hand expression or variable must be an integer. The $<<=$ form shifts the left-hand variable right-hand-many times and saves the result in that variable.
- $>>$, $>>=$ performs a right bit shift on the left-hand expression or variable. The right-hand expression must be a non-negative integer which specifies the number of times to shift the bits to the right. The left-hand expression or variable must be an integer. The $>>=$ form shifts the left-hand variable right-hand-many times and saves the result in that variable.
- `(X)` an operator form of the `kron` function which implements the tensor or Kronecker product. The left-hand side is tensored with the right-hand side.
- $(\ldots)$ forces precedence for an expression as in any other programming language. An expression within the parentheses is evaluated before evaluating expressions outside of the parentheses.
- `;` the semicolon suppresses output of an expression. For example, $x = 1$ would store the value of 1 in the variable x and output $x = 1$ to standard output, whereas $x = 1;$ would also store the value of 1 in the variable x but would not output anything to standard output. When a semicolon appears in a matrix definition, it has a different meaning entirely. Within a matrix definition, a semicolon denotes the end of a row.
- `a(`*n*, *k*`)` if `a` is a variable containing a matrix, then this expression returns the element indexed by the row index n and the column index k. Numbering of indices starts at 1. Unlike languages such as Matlab, this expression may not be used to assign values to elements of a matrix. It may only be used to read a particular element from a matrix (e.g. `x = a(`*1, 2*`) + 2` is valid, but `a(`*1, 2*`) = 3+2` is not). Future versions may support this, however, if there is demand for such functionality. n and k must be complex numbers with no imaginary components. n and k must also each be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, n and k must each be at least 1 after rounding.
- `|a:>_n, <:a|_n` Dirac-style syntax for the ket and bra respectively. `a` may be any state vector or integer expression, and the complex-conjugate transpose is automatically applied where necessary when expressing it as a bra or a ket. If `a` is a state vector variable, it may be assigned to using this syntax. The `_n` portion is an optional integer expression, and it prepends n $|0\rangle$ or $\langle 0|$ state vectors to the resulting state. This feature is useful when combining bras and kets whose bit representations of `a` have different numbers of bits and therefore ensures the dimensions are the same. The bras and kets may be concatenated as in standard Dirac notation to compute inner and outer products. Examples of this feature are presented in Section 4.

- `a{n}` indexes the *n*th character or bit of `a` if `a` is a string variable or integer variable, respectively. `n` must be an integer expression. This feature can be used to read or set the *n*th character or bit. For integers, `a{1}` is the least-significant bit.
- `a(`n_1, n_2, n_3, ...`)` if `a` is not a variable containing a matrix, it is considered to be user-defined function call. n_1, n_2, and n_3 are function arguments that can be expressions or variables of any type. There is no constraint on the number of arguments. Also note that passing QuIDD arguments and QuIDD return values only requires $O(1)$ memory since only a single pointer to the head of a QuIDD needs to be passed. Arguments passed as parameters to functions are not modified by the function (i.e. pass-by-value is always used). See Section 3 for more details.
- `abs(n)` computes the complex modulus of the numerical expression, QuIDD matrix, or QuIDD vector `n`. When `n` is a QuIDD matrix or vector, the complex modulus is computed on each element. In the case of a real-valued number `n`, this operation reduces to the standard absolute value.
- `atan(`*n*`)` returns the arc tangent of the expression *n* passed as an argument. If *n* is a matrix, it returns a matrix containing the element-wise arc tangent of *n*.
- `bn` evaluates the binary expression `n` as a decimal integer. For example, `foo = b100` sets the variable `foo` to the integer value 4. Notice that the left-most bit is the most-significant bit. As a result, in this example, `foo{1}` would evaluate to 0, while `foo{3}` would evaluate to 1. Since this is an ordinary integer expression, it may be used in conjunction with the Dirac notation. `|b100:>` would create the QuIDD state vector representation for the state $|100\rangle$ for instance.
- `cb(`"..."`)` returns a computational basis state vector. The string literal argument consists of a sequence of any number and combination of '0' and '1' characters. The string is parsed from left to right. Each '0' causes a $|0\rangle$ to be tensored into the vector, and each '1' causes a $|1\rangle$ to be tensored into the vector. `cb` can easily be used to create density matrices by using it in conjunction with the complex-conjugate transpose operation (`'`), matrix multiplication, and scalar operations.
- `cnot(`"..."`)` returns a 2-qubit controlled-NOT (CNOT) gate matrix. This is a faster, specialized version of `cu_gate`. If a controlled gate matrix with different numbers of controls/targets and/or a different action (*U* operator) is desired, then use the more general `cu_gate` function. The argument to `cnot` is a string literal using the same gate specification syntax as `cu_gate`. However, the only valid parameters accepted by `cnot` are *'cx'* and *'xc'*, since these string specifications are the only possible strings that produce a valid 2-qubit CNOT gate matrix. For example, `cnot(`*'cx'*`)` produces a CNOT gate matrix with the control on the "top" wire and the action (*X* operator) on the "bottom" wire. For a discussion of how the concept of wires relates to creating controlled gate matrices, see `cu_gate`.
- `conj(`*n*`)` returns the complex-conjugate of the expression *n* passed as an argument. *n* can be a complex number or a matrix.
- `cos(`*n*`)` returns the cosine of the expression *n* passed as an argument. If *n* is a matrix, it returns a matrix containing the element-wise cosine of *n*.
- `cps(`*n*`)` returns an *n*-qubit conditional phase shift (CPS) gate matrix. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own CPS matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly. The conditional phase shift gate is particularly useful in Grover's quantum search algorithm [8].

- `cu_gate(`*a*, "…"`)` is a generalized controlled-*U* gate matrix creation function. It returns a controlled or uncontrolled gate matrix given an action matrix (*a*) and a string literal with the gate specification (the second argument contained in "s). The string literal consists of a sequence of characters. The idea is for the string literal to specify what the gate should do to each "wire" in a quantum circuit. When conceptualizing a quantum circuit graphically and reading top-down, the first character corresponds to the first qubit wire, the second character corresponds to the second qubit wire, etc. Each character can take one of four possible values. 'i' denotes the identity, which means that the gate does nothing to the wire at that location. 'x' denotes an action, which means that the matrix specified by the argument *a* is applied to the wire at that location. 'c' denotes a control, which means that the wire at that location is used as a control on any 'x' wire (a $|1\rangle$ state forces *a* to operate on any 'x' wire, whereas a $|0\rangle$ causes nothing to happen on any 'x' wire). 'n' is a negated control, which is the opposite of 'c' (a $|0\rangle$ state forces *a* to operate on any 'x' wire, whereas a $|1\rangle$ causes nothing to happen on any 'x' wire). Any sequence of these characters may be used. Although there is no "actual" circuit, the string characters allow a user to conceptualize a circuit and construct a matrix which operates on the wires in that conceptualized circuit. *a* may be a matrix that operates on more than one qubit as long as one or more blocks of contiguous 'x' characters appear such that the size of each block is equal to the number of qubits operated on by *a*. For examples, see steaneX.qpro and steaneZ.qpro under the examples/nist/ subdirectory. Always use this function instead of defining your own gates explicitly, since it is asymptotically faster and uses asymptotically less memory. Since `cu_gate` must parse the input specification string, other functions such as `hadamard` and `cps` should be used instead of `cu_gate` for specific gates because they do not perform any parsing and are therefore a bit more efficient. An alternative function name for `cu_gate` is `lambda`. Also see the alternative, condensed version of `cu_gate` discussed next. The alternative version may be preferable for circuits with many qubits.
- `cu_gate(`*a, "…", n*`)` An alternative syntax for `cu_gate` which takes a condensed string literal. This condensed string literal specifies only the actions and controls along with the qubit wires they are applied to. For example, a Toffoli gate in a 5-qubit circuit, with controls on the second and fourth wires and the action on the fifth wire, can be created with the call `cu_gate(sigma_x(1),` *"c2c4x5", 5*`)`. As implied by this example, *n* is the total number of qubits in the circuit that the gate is applied to. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. More examples can be found in the examples/ directory and include hadder_bf1.qpro and rc_adder1.qpro, among others.
- `cutoff_val` If the cutoff value is set, any portion of all QuIDD element values that is less than the cutoff value will be rounded. For example, `cutoff_val =` $1e-15$ will cause all subsequently created QuIDD element values to be rounded at the 15th decimal place. By default, the cutoff value is not set and no rounding occurs. If the cutoff value is set by the user, it can be reset to the default (i.e. no rounding) by assigning 0 to `cutoff_val`.
- `dump_dot(`"…", "…", *a*`)` outputs the *dot* form of the graphical QuIDD representation of the matrix/vector *a* to a file specified by the second argument. The first argument is the name that will appear at the top of the QuIDD image. dot is a simple scripting language supported in the Graphviz package[1] Once the dot file is generated, dot can be run from the command line to produce a PostScript image of the QuIDD representation as such:

[1] Graphviz can be obtained at `http://www.graphviz.org/`.

`dot -Tps filename.dot -o filename.ps`
dot can generate other graphical file formats as well. Consult Graphviz for more details. A simple example is contained in the examples/dot subdirectory.

- `echo("...")` prints the string literal passed as an argument to standard output. Putting one or more semicolons after `echo` does not suppress its output. `echo` has no return value, so it cannot be used in expressions.
- `else` program flow control construct that is part of an "if-elseif-else" control block sequence. Its meaning is the same as in just about any other language. Only one `else` may optionally appear in an "if-elseif-else" block, and it must appear only at the end of the block. If an `else` block is used, its body (a sequence of zero or more expressions and/or control blocks to be executed) must be terminated by an `end` even if the body is empty. The body following `else` is executed when the preceding `if` and `elseif` conditions evaluate to "false" (i.e. a complex numbered value of zero).
- `elseif` program flow control construct that is part of an "if-elseif-else" control block sequence. Its meaning is the same as in just about any other language. It contains a condition which is an expression enclosed in parentheses. Zero or more `elseif`'s may appear in an "if-elseif-else" block, but the first `elseif` must appear after an `if`, and the last `elseif` must appear before an optional `else`. If no `else` appears after an `elseif`, the body of the `elseif` (a sequence of zero or more expressions and/or control blocks to be executed) must be terminated by an `end` even if the body is empty. The condition determines whether or not the statements in the body are executed. The body of the `elseif` is executed when the following two conditions are met: 1.) the preceding `if` and `elseif` conditions evaluate to "false" (i.e. a complex numbered value of zero), and 2.) the `elseif` condition evaluates to "true" (i.e. any non-zero complex numbered value).
- `end` keyword that signifies the end of a program flow control construct. In other words, `end` should be used to denote the end of "if-elseif-else" and "while" blocks.
- `exp(`n`)` returns e^n. If `n` is a matrix, then it returns a matrix containing the element-wise computation of e^k where k is an element from n.
- `expm(`n`)` returns e^n, where `n` is a matrix. This is standard matrix exponentiation and is approximated by a finitely bounded Taylor series. In the current version of the QuIDDPro simulator, you may only apply `expm` to a matrix n whose dimensions do not exceed 8×8 for efficiency reasons. Future versions may support larger dimensional arguments, but it is unlikely that larger dimensional arguments will be needed for most quantum-mechanics applications. If n is a complex number, then it returns e^n.
- `eye(`n`)` returns an $n \times n$ identity matrix. If you only need an identity matrix whose dimensions are a power of 2 in size (e.g. for k-qubit identity gate matrices) then use $identity(k)$ instead (see below), which runs slightly faster. n must be a complex number with no imaginary component. n must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, n must be at least 1 after rounding. Always use `eye` or `identity` instead of defining identity matrices explicitly because they are asymptotically faster and use asymptotically less memory.
- `fredkin()` returns a Fredkin gate matrix.
- `function` *var_name* `=` *func_name*`(`n_1, n_2, n_3, ...`)` defines a function body. This definition should exist in a file by itself with a filename that matches *func_name* appended by the ".qpro"

or ".qp" extensions. *var_name* is the name of the variable that contains the return value. n_1, n_2, and n_3 are function parameters that can be of any type. There is no constraint on the number of parameters. Also note that passing QuIDD arguments and QuIDD return values only requires $O(1)$ memory since only a single pointer to the head of a QuIDD needs to be passed. Arguments passed as parameters to functions are not modified by the function (i.e. pass-by-value is always used). Following the return value/function name line, the script code comprising the function body should appear. See Section 3 for more details.

- `gen_amp_damp(`*d, p, n, a*`)` performs generalized amplitude dampening (see [9, p. 382] for a description of generalized amplitude dampening). *a* is a density matrix (it must be square and have dimensions that are a power of 2 in size) on which dampening is to be performed. *a* is not modified, but the result of dampening applied to *a* is returned. *d* is the dampening parameter and must be a complex number with no imaginary component. *p* is the probability parameter and must also be a complex number with no imaginary component. *d* and *p* must each be in the range $[0,1]$. *n* is the qubit wire number that dampening is to be applied to. This wire number is only conceptual and can alternatively be thought of as the *n*th quantum state in the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.
- `get_bit(n, a)` returns the value of the *n*th bit of the integer `a`. $n=1$ is the least-significant bit.
- `gp_equals(a, b)` returns a 2-element row vector, where the the first element is a 1 (0) if `a` and `b` are (not) equal up to global phase, and the second element is the global phase factor if the first element is a 1. *a* and *b* must be QuIDD matrices or vectors. Examples of this function are presented in Section 5.
- `hadamard(`*n*`),H(n)` returns an *n*-qubit Hadamard gate matrix. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own Hadamard matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.
- `i` is a variable that is preset to the value $0+1i$. It can be overwritten at runtime by the user.
- `identity(`*n*`)` returns an *n*-qubit identity gate matrix. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own identity matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly. Also see the *eye* function.
- `if` program flow control construct that is part of an "if-elseif-else" control block sequence. Its meaning is the same as in just about any other language. It contains a condition which is an expression enclosed in parentheses. An "if-elseif-else" block must be started by a single `if`, but "if-elseif-else" blocks can be nested within other "if-elseif-else" blocks (nesting with "while" blocks is also allowed). An `if` must be followed by a body of zero or more expressions and/or

control blocks, and this body must be terminated by either an `elseif`, an `else`, or an `end`, even if the body is empty. The condition determines whether or not the statements in the body are executed. The body is executed once if the condition evaluates to "true" (i.e. any non-zero complex numbered value). Otherwise if the condition evaluates to "false" (i.e. a complex numbered value of zero), the body is not executed.

- `kron(`*n, k*`)` returns the tensor (Kronecker) product of the matrix expressions *n* and *k*. If *n* and *k* are complex numbers, then they are multiplied together.
- `lambda(`*a*, "*...*"`)` an alternative name for the function `cu_gate`.
- `measure(`*n, a*`)` performs deterministic measurement on the *n*th qubit in the density matrix *a*. In other words, all off-diagonal correlations corresponding to the qubit being measured are zeroed out, and the resultant density matrix is returned (for probabilistic measurement of a qubit in a density matrix that returns a 1 or 0, see `pmeasure`). *a* must be square and have dimensions that are a power of 2 in size. *a* is not modified, but the result of measurement applied to *a* is returned. *n* is the qubit wire number that measurement is to be applied to. This wire number is only conceptual and can alternatively be thought of as the *n*th quantum state in the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.
- `measure_sv(`*n, a*`)` probabilistic measurement is performed on qubit *n*. A state vector is returned which represents the state vector *a* as modified by the measurement result and its associated norm. If the measurement result and the associated norm have already been computed with a previous call to `pmeasure_norm_sv`, then `measure_sv` can be called with the alternative syntax `measure_sv(`*n, a, res, norm*`)`. *res* and *norm* denote the precomputed measurement result and associated norm, respectively. Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. *res* must have the value 0 or 1 to within the rounding threshold. *norm* should be a valid norm of a state vector.
- `measure_sv0(`*n, a*`)` returns the probability of measuring qubit *n* as a 0 in state vector *a* (for probabilistic measurement of a qubit in a state vector that returns a 1 or 0, see `pmeasure_sv`). Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.
- `measure_sv1(`*n, a*`)` returns the probability of measuring qubit *n* as a 1 in state vector *a* (for probabilistic measurement of a qubit in a state vector that returns a 1 or 0, see `pmeasure_sv`). Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are

valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `norm(a)` returns the norm of a state vector or complex number *a*. Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2.
- `one_merge(a)` returns the result of changing every non-zero element of `a` into a 1. `a` must be a QuIDD matrix, QuIDD vector, or a number. As discussed in Section 5, this function can be used to determine a necessary condition for two operators or states to be equal up to relative phase.
- `output_prec` denotes the output precision. When assigned a non-negative integer value, it specifies how many digits should be output to the screen. Any digits which exceed this number are rounded. For example, `output_prec = 3` will cause $1/3$ to output 0.333 to the screen. Note that the internal precision of any numbers and variables are unaffected. `output_prec` only affects the screen output precision. By default, the variable `output_prec` is not set, but the output precision is initially 6. Assigning a negative value to `output_prec` restores the default output precision. However, assigning a matrix to `output_prec` leaves the precision unchanged from its previous value.
- **phase(*n*)** returns an *n*-qubit Phase gate matrix (aka. *S* gate [9]). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own Phase matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.
- `pi` is a variable that is preset to the value of π to a large number of decimal places. It can be overwritten at runtime by the user.
- `pmeasure(n, a)` performs probabilistic measurement on the *n*th qubit in the density matrix *a*. The result returned is a 1 or 0 (for deterministic measurement of a qubit in a density matrix, see `measure`). *a* must be square and have dimensions that are a power of 2 in size. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.
- `pmeasure_norm_sv(n, a)` performs probabilistic measurement on the *n*th qubit in the state vector *a*. A 1×2 vector is returned containing a 1 or 0 for the measurement result (the first element) and the norm associated with the measurement result (the second element). Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.
- `pmeasure_sv(n, a)` performs probabilistic measurement on the *n*th qubit in the state vector *a*. The result returned is a 1 or 0 (for deterministic measurement of a qubit in a state vector see `measure_sv0` and `measure_sv1`. Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are

rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `proj0(`*n*`)` returns an *n*-qubit $|0\rangle$ projector gate matrix (i.e. $|0\ldots0\rangle\langle0\ldots0|$, for *n* 0's). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own $|0\rangle$ projector matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.
- `proj1(`*n*`)` returns an *n*-qubit $|1\rangle$ projector gate matrix (i.e. $|1\ldots1\rangle\langle1\ldots1|$, for *n* 1's). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own $|1\rangle$ projector matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.
- `projplus(`*n*`)` returns an *n*-qubit $|+\rangle$ projector gate matrix (i.e. $|+\ldots+\rangle\langle+\ldots+|$, for *n* +'s). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own $|+\rangle$ projector matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.
- `ptrace(`*n, a*`)` performs the partial trace over the *n*th qubit in the density matrix *a*. *a* must be square and have dimensions that are a power of 2 in size. *a* is not modified, but the result of the partial trace applied to *a* is returned. *n* is the qubit wire number that is traced over. This wire number is only conceptual and can alternatively be thought of as the *n*th quantum state in the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.
- `px(`*p, n, a*`)` applies a probabilistic Pauli *X* gate matrix to the *n*th qubit in the density matrix *a*. *a* must be square and have dimensions that are a power of 2 in size. *a* is not modified, but the result of dampening applied to *a* is returned. *p* is the probability parameter and must be a complex number with no imaginary component. *p* must be in the range $[0,1]$. *n* is the qubit wire number that the probabilistic *X* gate matrix is to be applied to. This wire number is only conceptual and can alternatively be thought of as the *n*th quantum state in the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.
- `Px(`*p, n, a*`)` an alternative name for the function `px`.

- `py(`*p, n, a*`)` applies a probabilistic Pauli *Y* gate matrix to the *n*th qubit in the density matrix *a*. *a* must be square and have dimensions that are a power of 2 in size. *a* is not modified, but the result of dampening applied to *a* is returned. *p* is the probability parameter and must be a complex number with no imaginary component. *p* must be in the range $[0,1]$. *n* is the qubit wire number that the probabilistic *Y* gate matrix is to be applied to. This wire number is only conceptual and can alternatively be thought of as the *n*th quantum state in the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.
- `Py(`*p, n, a*`)` an alternative name for the function `py`.
- `pz(`*p, n, a*`)` applies a probabilistic Pauli *Z* gate matrix to the *n*th qubit in the density matrix *a*. *a* must be square and have dimensions that are a power of 2 in size. *a* is not modified, but the result of dampening applied to *a* is returned. *p* is the probability parameter and must be a complex number with no imaginary component. *p* must be in the range $[0,1]$. *n* is the qubit wire number that the probabilistic *Z* gate matrix is to be applied to. This wire number is only conceptual and can alternatively be thought of as the *n*th quantum state in the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.
- `Pz(`*p, n, a*`)` an alternative name for the function `pz`.
- `qp_epsilon` When creating new QuIDD element values, a cache is checked internally to see if those values have already been created. The more repeated values there are in a matrix, the more the matrix is compressed by its QuIDD representation. When checking the cache, QuIDDPro compares the equality of a new value to other values already in the cache to using an epsilon. Specifically, *a* and *b* are considered equal if $abs(a-b) < epsilon * a$ and $abs(a-b) < epsilon * b$. Epsilon can be changed by assigning values to `qp_epsilon`. By default, the epsilon value is $1e-8$. Currently, the epsilon value is not always used when creating new QuIDD element values, but in future versions of QuIDDPro, the epsilon value will play a much greater role.
- `quidd_info(`*a*`)` prints information about an operator or state to standard output. This information includes the number of qubits represented (or acted upon), the dimensions of the explicit representation of the matrix, and the number of nodes in the QuIDD representation of the matrix. Note that the explicit matrix representation is not actually stored anywhere. *a* must be a valid operator, state vector, or density matrix.
- `r2` is a variable that is preset to the value of $\sqrt{2}$ to a large number of decimal places. It can be overwritten at runtime by the user.
- `r3` is a variable that is preset to the value of $\sqrt{3}$ to a large number of decimal places. It can be overwritten at runtime by the user.
- `rand(`*n*`)` returns a pseudo-random value between 0 and *n*. *n* can be any real value, including negative values.
- `round(`*n*`)` returns *n* with its real and imaginary parts rounded to the nearest integer. "Halfway" cases are rounded away from 0. Since there is no native integer type supported in QuIDDPro,

round can be extremely helpful in ensuring that values which are supposed to be integer values are indeed integer values.

- `rp_equals_op(a, b)` returns a vector containing relative phase factors if `a` and `b` are equal up to relative phase, otherwise it returns a vector of 0's. `a` and `b` must be QuIDD matrices. Examples of this function are presented in Section 5.
- `rp_equals_sv(a, b)` returns a vector containing relative phase factors if `a` and `b` are equal up to relative phase, otherwise it returns a vector of 0's. `a` and `b` must be QuIDD state vectors. Examples of this function are presented in Section 5.
- `run "..."` executes all script code contained in the file specified by the argument. The run command may appear anywhere in a script except inside an explicit matrix. This command is ideal for declaring variables that may be re-used in multiple projects.
- `rx(`n, k`)` returns a k-qubit rotational Pauli X gate matrix given a real valued angle parameter n. n must be a complex number with no imaginary component. n must be in the range $[0,1]$. k must be a complex number with no imaginary component. k must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, k must be at least 1 after rounding.
- `Rx(`n, k`)` an alternative name for the function `rx`.
- `ry(`n, k`)` returns a k-qubit rotational Pauli Y gate matrix given a real valued angle parameter n. n must be a complex number with no imaginary component. n must be in the range $[0,1]$. k must be a complex number with no imaginary component. k must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, k must be at least 1 after rounding.
- `Ry(`n, k`)` an alternative name for the function `ry`.
- `rz(`n, k`)` returns a k-qubit rotational Pauli Z gate matrix given a real valued angle parameter n. n must be a complex number with no imaginary component. n must be in the range $[0,1]$. k must be a complex number with no imaginary component. k must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, k must be at least 1 after rounding.
- `Rz(`n, k`)` an alternative name for the function `rz`.
- `set_bit(n, a)` sets the value of the nth bit of the integer `a`. $n=1$ is the least-significant bit.
- `sigma_x(`n`)` returns an n-qubit Pauli X gate matrix. n must be a complex number with no imaginary component. n must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, n must be at least 1 after rounding. Always use this function instead of explicitly defining your own X matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.
- `sigma_y(`n`)` returns an n-qubit Pauli Y gate matrix. n must be a complex number with no imaginary component. n must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, n must be at least 1 after rounding. Always

use this function instead of explicitly defining your own X matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.

- `sigma_z(n)` returns an n-qubit Pauli Z gate matrix. n must be a complex number with no imaginary component. n must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, n must be at least 1 after rounding. Always use this function instead of explicitly defining your own X matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.
- `sin(n)` returns sine of the expression n passed as an argument. If n is a matrix, it returns a matrix containing the element-wise sine of n.
- `tan(n)` returns the tangent of the expression n passed as an argument. If n is a matrix, it returns a matrix containing the element-wise sine of n.
- `sqrt(n)` returns the square root of the expression n passed as an argument. If n is a matrix, it returns a matrix containing the element-wise square root of n.
- `swap(n, k, a)` returns the vector resulting from swapping qubits n and k in the state vector a. This function swaps qubits *much more quickly* than swapping using *CNOT* and Hadamard gates. Since a must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. a is not modified by this function. n and k must be complex numbers with no imaginary components. n and k must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). n and k must also be at least 1 after rounding.
- `tic` starts a timer and also starts to record the peak memory usage from the point `tic` is called. `tic` has no return value, so it cannot be used in expressions. The timer only records time spent and memory used while running code. Thus, in the case of interactive mode, the timer will not be recording time spent nor memory used while at an idle prompt.
- `toc` stops a timer started by a previous `tic` or `toc` command. It outputs to standard output the time that has elapsed (i.e. time spent running code), the number of gates applied, the average runtime per gate, and memory that was used (peak memory) since the last `tic` or `toc` command. It also outputs the base memory which is the memory used in initializing the simulator and reading the input code. Base memory should be interpreted as a one-time initialization cost of the simulator and should not be considered when measuring performance. Please see Appendix B for more details. Operations that are recorded as applied gates include matrix multiplication, `gen_amp_damp`, `measure`, `measure_sv`, `Px`, `Py`, and `Pz`.
- `toffoli("...")` returns a 3-qubit Toffoli gate matrix. This is a faster, specialized version of `cu_gate`. If a controlled gate matrix with different numbers of controls/targets and/or a different action (U operator) is desired, then use the more general `cu_gate` function. The string argument uses the same syntax as that of `cu_gate`. However, `toffoli` only accepts the strings *'ccx'*, *'cxc'*, and *'xcc'*, since these are the only valid Toffoli specifications. For example, `toffoli('ccx')` produces a Toffoli gate matrix with the controls on the "top" two wires and the action (X operator) on the "bottom" wire. For a discussion of how the concept of wires relates to creating controlled gate matrices, see `cu_gate`.
- `while` program flow control construct that allows multiple iterations of a body of code ("looping"). Its meaning is the same as in just about any other language. It contains a condition which is an expression enclosed in parentheses. A "while" block must be started by a single `while`, but

"while" blocks can be nested within other "while" blocks (nesting with "if-elseif-else" blocks is also allowed). A `while` must be followed by a body of zero or more expressions and/or control blocks, and this body must be terminated by an `end`, even if the body is empty. The condition determines whether or not the statements in the body are executed. As long as the condition evaluates to "true" (i.e. any non-zero complex numbered value), the body is iteratively executed. The iterations stop when the condition becomes "false" (i.e. a complex numbered value of zero). The condition is checked once prior to executing each iteration of the body.

- `zeros(`*n*, *k*`)` returns an $n \times k$ matrix of all 0's. *x* and *y* must be complex numbers with no imaginary components. *n* and *k* must be complex numbers with no imaginary component. *n* and *k* must also each be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* and *k* must each be at least 1 after rounding. Always use *zeros* instead of defining zero matrices explicitly because it is asymptotically faster and uses asymptotically less memory.

9 Ongoing Work

We are currently implementing and considering several new features, mentioned below. Feedback on the relevance and utility of these features and requests for other features are greatly appreciated.

- We are looking to extend the interface between the QuCirc data structure and the QuIDDPro simulator. In particular, we are developing a feature that allows the user to load pre-compiled circuits into memory and built-in functions to apply such circuits during interactive mode.
- Equivalence checking can be improved for circuits with only classical gates (NOT, CNOT, Toffoli, SWAP and Fredkin). When checking two such circuits, one can convert each gate to AND and XOR gates, and use fast verification tools such as ABC/CEC [17]. We are developing QuIDDPro functions to detect such cases and make external calls to ABC automatically.
- For matrices with too little structure, using an array-based matrix representation instead of a QuIDD may improve runtime performance. Thus, we are developing heuristics for switching internally between explicit matrices and QuIDDs "on the fly." We also hope to decrease memory consumption and runtime of QuIDDPro by enhancing the QuIDD datastructure.
- Other extensions to the input language may also be useful. We are looking at alternative input languages, including [15], to explore such extensions.
- More examples will be added to the QuIDDPro package. For example, we are planning to use QuIDDPro to simulate quantum adiabatic computation, among other applications. However, since it is difficult to envision all potential quantum-mechanical contexts in which QuIDDPro can be applied, **user feedback can be particularly helpful in this regard**. We will incorporate QuIDDPro code submitted by users into the package, with proper credits.

Appendix A: New Features in version 3.8

- New features
  - Support for construction of Phase matrices directly using a built-in function. This is particularly useful for simulating stabilizer circuits.
  - Batch-mode options for simulating a UMICH 1.0 quantum circuit description file using an optimized version of QuIDDPro. See Sections 6 and 7 for details.
- Bug fixes
  - Fixed a bug that crashed the application when a user typed tab or backspace during interactive mode.

Appendix B: New Features in version 3.5

- New features
  - Support for left-multiplication via =*. This is useful for applying operators to state vectors. The script examples in this document illustrate how to use this feature.
  - Compilation mode for generating a UMICH 1.0 quantum circuit description file from a QuIDDPro script. See Section 6 for details.
- New example scripts
  - `qft/` directory in `examples/` contains functions for calculating the QFT and inverse QFT.

Appendix C: New Features in version 3.1

- New features
  - Efficient functions for checking equivalence up to global and relative phase of operators and states. The phase factors are also computed by most of these functions. See Section 5 for details. The algorithms that these functions are based on are described in [14].
  - Support for Dirac-style notation. See Section 4 for details.
  - Support for string data types, including variable storage and string manipulation, has been added. Section 4 has examples of this feature. This feature makes the `cu_gate` function much easier to use.
  - New operators including `+=`, `-=`, `*=`, and `/=`.
  - Bit manipulation operators including `<<` and `>>`. Also functions to get and set individual bits of an integer variable have been added.
  - Binary integer expressions are now supported using the `bn` syntax.
  - The `abs` function has been added, which implements the complex modulus operation.
  - Alternate syntax for the `kron` function in the form of the `(X)` operator, which makes writing Dirac-style expressions more natural.
  - Alternate syntax for the `hadamard` function. `H` is now syntactic short-hand for it.
- New example scripts
  - New example scripts have been added which demonstrate all of the new features.

- C++ compatibility
  - A C++ library and API are available to use QuIDDPro in C++ programs.
- Bug fixes
  - A bug which degraded runtime and memory performance when using loops and function calls within tight loops has been fixed.
  - In some cases when multiplying a call to `cu_gate` with a state vector, the state vector was modified incorrectly. This bug has been fixed.
  - Fixed the `cnot` and `toffoli` functions, which previously would incorrectly modify state vectors in certain cases.

Appendix C: Notes on Performance Analysis

The QuIDDPro simulator uses the QuIDDPro library developed in C++ by George Viamontes at the University of Michigan. This library is integrated as a back-end. The Bison-generated front-end parser accepts a "QuIDDPro input language" similar to Matlab (see Section 8).

The commands *tic* and *toc* report runtime, base, and peak memory (see Section 8). The base memory refers to the initialization of the simulator and input, rather than the simulation itself. The peak memory refers to the usage by the simulation back-end between the `tic` and `toc` commands. The sum of these two readings gives the memory required to run the overall simulation. The base memory due to the simulator initialization is a constant of about 10.5MB on Linux, and may be larger on Solaris. Most of this base memory is due to initialization of the CUDD manager. For large quantum circuits, this overhead is often dwarfed by asymptotic improvements of QuIDDs over array-based representations of states and operators.

References

[1] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Graph-based simulation of quantum computation in the density matrix representation," *Quantum Information and Computation*, **5** (2), pp. 113-130, 2005.

[2] G. F. Viamontes, I. L. Markov, J. P. Hayes, "Improving Gate-Level Simulation of Quantum Circuits," *Quantum Information Processing*, **2** (5), 347-380, October 2003. `http://www.arxiv.org/abs/quant-ph/0309060`

[3] G. F. Viamontes, I. L. Markov, J. P. Hayes, "Is Quantum Search Practical?" *Computing in Science and Engineering*, **7** (4), pp. 22-30, May/June 2005. `http://www.arxiv.org/abs/quant-ph/0405001`

[4] G. F. Viamontes, M. Rajagopolan, I. L. Markov, and J. P. Hayes, "Gate-level simulation of quantum circuits," *Proc. of ACM/IEEE Asia and South-Pacific Design Automation Conf.* (ASPDAC), pp. 295-301, Kitakyushu, Japan, January 2003.

[5] P. E. Black et al., "Quantum compiling and simulation," `http://hissa.nist.gov/~black/Quantum/`

[6] libquantum, `http://www.enyo.de/libquantum/`

[7] "QHDL: A Design Language for Quantum Computing," `http://www.atcorp.com/Projects/Quantum%20computing/quantum.htm`

[8] L. Grover, “Quantum mechanics helps in searching for a needle in a haystack,” *Phys. Rev. Lett.* (79), pp. 325-8, 1997.

[9] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge Univ. Press, 2000.

[10] A. M. Childs, H. L. Haselgrove, and M. A. Nielsen, “Lower bounds on the complexity of simulating quantum gates,” *Phys. Rev. A* (68), 052311, 2003.

[11] A. Barenco et al., “Elementary gates for quantum computation,” *Phys. Rev. A*, **52**, pp. 3457-3467, 1995.

[12] V. V. Shende, S. S. Bullock, I. L. Markov, “Synthesis of quantum logic circuits,” *IEEE Trans. on Computer-Aided Design*, **25**, pp. 1000-1010, 2006.

[13] G. Song and A. Klappenecker, “Optimal realizations of simplified Toffoli gates,” *Quantum Information and Computation*, **4**, pp. 361-372, 2004.

[14] G. F. Viamontes, “Efficient Quantum Circuit Simulation,” Ph.D. Dissertation at the University of Michigan, 2007.

[15] Andrew Petersen and Mark Oskin, “A new algebraic foundation for quantum programming languages,” In the 2nd workshop on Non-Silicon Computing (NSC) at ISCA, June 2003.

[16] S. Aaronson and D. Gottesman, “Improved simulation of stabilizer circuits,” *Phys. Rev. A*, **70**, 052328, 2004, `http://www.scottaaronson.com/chp/`

[17] R. Brayton and A. Mishchenko, “ABC: An Academic Industrial-Strength Verification Tool,” *Computer-aided Verification (CAV)*, **6174**, pp. 24-40, 2010, `http://www.eecs.berkeley.edu/ alanmi/abc/`