QUANTUM ARCHITECTURES AND COMPUTATION

Language-Integrated Quantum Operations (LIQ$Ui|\rangle$) Simulator
User's Manual

LIQ$Ui|\rangle$ User's Manual

QUANTUM ARCHITECTURES AND COMPUTATION

LIQ$Ui|\rangle$ Users Manual

V3.6 20160517

BY DAVE WECKER

Table of Contents

List of Figures

List of Examples

Equations

Chapter
1

Introduction

What is LIQ$Ui|\rangle$?

LIQ$Ui|\rangle$ is a simulation platform to aid in the exploration of quantum computation. LIQ$Ui|\rangle$ stands for "Language-Integrated Quantum Operations". A quantum operation is usually referred to as a unitary operator (U) applied to a column state vector (also known as a ket: $|\cdot\rangle$). The "i" is just a constant scaling factor, hence the acronym.

Currently, there are three classes of simulators built into the system representing different levels of abstraction:

1. **Physical Modeling:** This is the Hamiltonian simulator which attempts to model some of the actual physics in a quantum system. It differs from the other simulators in that it has the concept of the time it takes for an operation to be performed (since it is numerically solving a differential equation) and can only operate on a small number of qubits (around 30). It is also (by its very nature) slow.

2. **Universal Modeling:** This is the most flexible of the simulators. It allows a wide range of operations to be performed (including ones defined by the user). It can handle millions of operations (gates) to be executed, is highly optimized for parallel execution and is highly efficient in memory usage. Its main limitation is the number of qubits (~30) that can be entangled at one time.

3. **Stabilizer Modeling:** This simulator has the virtue of allowing large circuits (millions of operations) on massive numbers of qubits (tens of thousands). The main limitation is the types of gates which may be included in the circuit. They are fixed in the system and come from the "stabilizer" class (e.g., Clifford group). This limits the usefulness of the types of algorithms that can be implemented and tested. However, it does allow the design and test of Quantum Error Correction Codes (QECC) which requires large numbers of qubits per logical qubits.

Simulations can be accomplished in several ways:

1. **Test mode**: Several built-in tests of the system can be invoked from the command line and are useful demonstrations.

2. **Script mode:** The system can be run directly from an F# text script (.fsx file). This allows the simulator to be operated by simply running the executable (no separate language compilation required). The entire simulator is available from this mode, but interactive debugging is difficult. Script mode allows users to experiment (with fast turn-around time) as well as being able to "kick the tires" without having to install a complete development environment.

3. **Function mode:** This is the normal development mode. It requires a compilation environment (e.g., Visual Studio) and the use of a .Net language (typically F#). The user has the full range of APIs at their disposal and can extend the environment in many ways as well as building their own complete applications.

4. **Circuit mode:** Function mode can be compiled into a circuit data structure that is extremely general. This data structure can be manipulated by the user, run through built-in optimizers, have quantum error correction added, rendered as drawings, exported for use in other environments and may be run directly by all the simulation engines.

The entire architecture is summarized in Figure 1 . Here are each of the major sections:

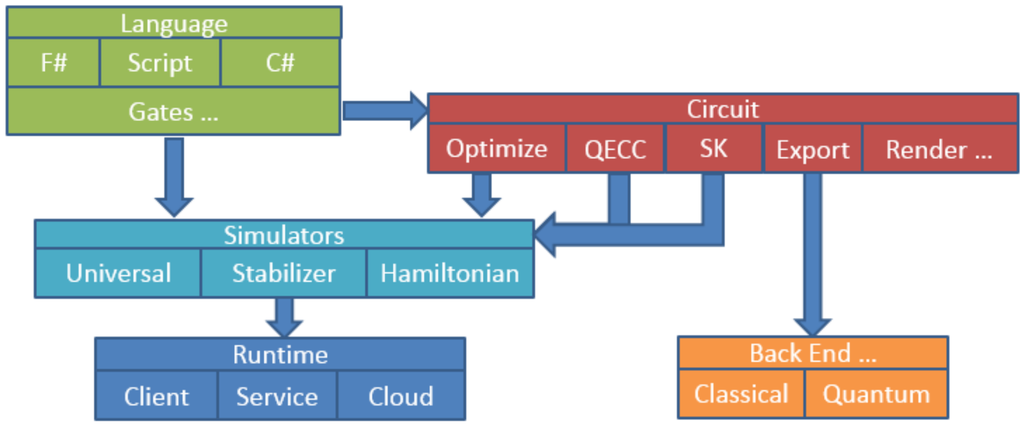


Figure 1: The LIQU$i|\rangle$ Platform Architecture

Suggested References

References cited throughout this document and generally useful to have around (in any case).

This manual will not be providing a background in either quantum computation or functional programming. The author suggests the following as good sources of information:

1. **Quantum Computation and Quantum Information:** This book by Michael Nielsen and Isaac Chuang is an invaluable reference source and I encourage you to obtain a copy. Most of the subjects discussed in the rest of this manual are fully covered in this reference.

2. **Programming F#: A comprehensive guide for writing simple code to solve complex problems:** This book by Chris Smith is an excellent introductory text into functional programming and F# in particular. If you're serious about developing your own simulations with LIQ$Ui|\rangle$ I would pick up a copy of either this book, or the following one.

3. **Expert F# 2.0:** This book by Don Syme, Adam Granicz and Antonio Cisternino is the book on F# that I use more than any other. It is available as an eBook as well.

4. The F# language reference can be found on MSDN (the Microsoft Developer Network) web site at:

   http://msdn.microsoft.com/en-us/library/dd233181.aspx

   While the full language reference is maintained at:

   http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.html

5. **Simulation of Electronic Structure Hamiltonians Using Quantum Computers:** This paper by Whitfield, Biamonte and Aspuru-Guzik gives a good background for the fermionic section of the Hamiltonian simulator (http://arxiv.org/abs/1001.3855).

6. More recent quantum chemistry papers (utilizing LIQ$Ui|\rangle$) include:

   a. **Gate count estimates for performing quantum chemistry on small quantum computers** http://arxiv.org/abs/1312.1695

   b. **Improving Quantum Algorithms for Quantum Chemistry** http://arxiv.org/abs/1403.1539

   c. **The Trotter Step Size Required for Accurate Quantum Simulation of Quantum Chemistry** http://arxiv.org/abs/1406.4920

   d. **Chemical Basis of Trotter-Suzuki Errors in Quantum Chemistry Simulation** http://arxiv.org/abs/1410.8159

7. **LIQUi|>: A Software Design Architecture and Domain-Specific Language for Quantum Computing** http://arxiv.org/abs/1402.4467

8. Additional information about the Microsoft Quantum Architectures and Computation group may be found at: http://research.microsoft.com/QuArC along with the LIQUi|> project page at: http://research.microsoft.com/en-us/projects/liquid/

Obtaining the Software

How to kick the tires

The first place to visit is:

http://github.com/msr-quarc/Liquid

This site explains how to obtain LIQ$Ui|\rangle$. The software may be used to model quantum systems and algorithms as described above in any of the three supported modes. In addition, the system can be extended in many ways, including adding user defined gates (unitary operators) and custom quantum error correcting codes.

Updates, news and discussions may also be found at the same location. News will also be published to the liquid-news email list; you can sign up for the list by sending an email to LISTSERV@lists.research.microsoft.com with a one-line body reading:

```
SUB Liquid-news FirstName LastName
```

replacing FirstName and LastName with your first and last names.

If you prefer to remain anonymous, you may instead send an email containing:

```
SUB Liquid-news anonymous
```

Chapter

2

Concepts and Data Types

The fundamental pieces of LIQ$Ui|\rangle$

This user's manual will not attempt to teach either Quantum Mechanics or Quantum Computation (there are a plethora of sources available). However, we need to have some fundamental agreement on terms used in the sections that follow and a basic understanding of the actual definitions shared inside of the simulator. We will start at the bottom and work our way up.

Bit binary values used inside the simulator.

The basic data type is the `Bit`. Closely related to the classical bit, our data type contains the usual states of `One` and `Zero` but adds a new state `Unknown`. At most times, quantum systems do not admit to an exact value until we measure them. For this reason, a bit may return the value `Unknown` while it is inside of a quantum computation and has not been viewed externally as of yet.

Qubit quantum value that represents an entity that may be measured as a **Bit**.

Quantum values are represented as `Qubits`. The qubit is defined as a pair of complex vectors pointing to a spot on a unit sphere (see Figure 2). Traditionally, a qubit pointing directly up (positive on the σ_z axis) is denoted as the column vector $|0\rangle$ and the vector pointing down is known as $|1\rangle$. When measured, these become the `Bits` `Zero` and `One` respectively. Another way to think of this is directly with matrices where:

$$|0\rangle = \begin{bmatrix} 1+0i \\ 0+0i \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad and \quad |1\rangle = \begin{bmatrix} 0+0i \\ 1+0i \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Equation 1: State Definition

Any qubit may be viewed as a linear combination of these two vectors, so typically we will refer to the state of a single qubit as determined by two complex values **a** and **b** where:

$$|\Psi\rangle = a|0\rangle + b|1\rangle = \begin{bmatrix} a \\ b \end{bmatrix}$$

Equation 2: Qubit State

`Qubits` will auto-normalize any states that are handed to them, so that $|a|^2 + |b|^2 = 1$. Another way to view all of this is that **a** squared is the probability of measuring the qubit as `Zero` and **b** squared is the probability of measuring the qubit as `One`.

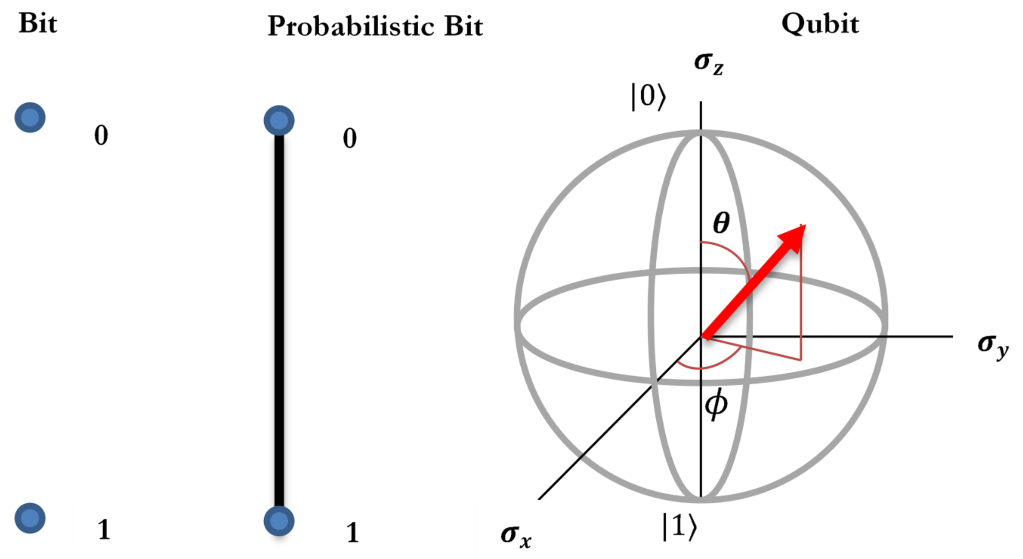


Figure 2: Bloch Sphere representation of a Qubit

One other useful way of interpreting the state of a qubit is by the angles of the vectors, in this case:

$$\mathbf{a} = \cos\left(\frac{\theta}{2}\right), \boldsymbol{b} = e^{i\phi}\sin\left(\frac{\theta}{2}\right)$$
$$(x, y, z) = (sin\theta cos\phi, sin\theta sin\phi, cos\theta)$$

Equation 3: Qubit as angles on the Bloch sphere

Ket Complete state of a quantum system

So far, we've seen what a single `Qubit` looks like, now we wish to combine them. In LIQ$Ui|\rangle$, this is known as a `Ket` (reference to a column vector in Dirac notation) and will always be 2^n in length, where n is the number of qubits in the state. Of course, this number grows very rapidly with the number of qubits and the simulator does everything it can to keep from actually materializing the entire vector unless it needs to (e.g., when the qubits are fully entangled). This limits most of the simulation to ~30 qubits (1 billion states = 1 billion double precision floating point complex numbers = 17GB) which is the most we can fit on a 32GB machine and still perform simulations. The exception to this rule is the Stabilizer simulation engine (which will be discussed in its own chapter).

In LIQ$Ui|\rangle$, every `Qubit` must belong to a single `Ket` and any `Qubit` can be queried to ask what `Ket` it belongs to (hence, one only needs to pass qubits around).

Gate Represents an operator

To perform operations on `Kets` we define `Gates`. A `Gate` may simply represent a unitary matrix that defines the operation (e.g., Hadamard, X, CNOT…), a non-unitary operation (e.g., Measurement and Reanimation) or a higher level concept (composite gates, binary controlled gates, extension gates (e.g., adjoint operator), Native and Label (UI support)). In addition to the actual operation, gates may define Names, Arity, Help and Rendering information for Circuit drawing). All of this will be described in detail in the section on user defined `Gates`.

Operations Represents the operation of a gate on a state

`Gates` are merely data structures. When they're wrapped in F# functions, they become operators that can apply the `Gate` to a set of `Qubits`. Inside of LIQ$Ui|\rangle$, all `Gates` are exposed as Operations and user defined gates are wrapped in the same way. However, any F# function may be viewed as an Operation, so this is not really a new data type, it's just the signature of any function that takes in `Qubits` and doesn't return a value (since the `Ket` is altered by the operation performed. Logically, you can view this as:

$$|\Psi_2\rangle = U|\Psi_1\rangle$$

Equation 4: Operations

Where U is the operation being performed on the state at time 1 resulting in a new state at time 2. Details will follow in the programming section, but if you wanted to apply a Hadamard gate on the first `Qubit` in a list of qubits, it would look like this:

```
H qs
```

This is why the system is called "Language Integrated". Once we reach the level of operations, everything is completely embedded in the host language (in this case F#). The other benefit of this type of integration is that depending on the `Ket` that the `qs` parameter belongs to this one line will do any of three things:

1. Apply the `Gate` and update the `Ket` containing the `Qubits`
2. Return the `Gate` structure that `H` refers to (for use in higher level functions)
3. Build a `Circuit` that contains the `Gate` (for optimizations and re-writing)

Circuit Represents a list of operations on gates.

One of the goals of LIQ$Ui|\rangle$ is to provide post-processing of quantum algorithms for various reasons: drawing, parallelizing, substitution (some gates will not be available in target physical systems), optimization, export and re-execution to name but a few. The `Circuit` data structure achieves this goal. Instead of running the Operations defining the quantum algorithm, the same calls can be used to build a `Circuit` that can be manipulated by various tools.

The tools that operate on `Circuits` make up the majority of the simulation system and will be described in detail in later chapters.

Chapter

3

Basic Operation

Getting up and running quickly

The two ways to interaction with the system are via a full compilation environment in Visual Studio linked to the LIQ$Ui|\rangle$ library (dll), or via an F# script hosted by the LIQ$Ui|\rangle$ application (exe). Both provide advantages. Compilation provides IntelliSense editing and a full debugging environment, while, scripting provides a quick and easy way to prototype and extend LIQ$Ui|\rangle$ while being able to quickly turn around simulations with varying parameters. There are also several test functions described in the rest of this manual that are built-in as well as provided in scripts (in the `samples` directory). All examples shown in this manual may be run in either way.

Execution Starting the simulator

Running LIQ$Ui|\rangle$ simply entails starting the LIQUiD.exe file which should reside in the same directory as the LIQUiD1.dll file. If started without arguments (or with illegal arguments), the program will give command line help:

```
==================================================
!!! ERROR: Need to provide at least one argument
==================================================
Liquid Usage:  Liquid [/switch...] function
    Enclose multi-word arguments in double quotes

Arguments (precede with / or -):

   Switch     Default              Purpose
   ------     -------------------- ------------------------
    /log      Liquid.log           Output log file name path
    /la       Unset                Append to old log files (otherwise erase)

    /s        ""                   Compile and load script file
    /l        ""                   Load compiled script file

 Final arg is the function to call:
   function(pars,...)

============================================

TESTS (all start with two underscores):
   __Big()            Try to run large entanglement tests (16 through 33 qubits)
   __Chem(m)          Solve Ground State for molecule m (e.g., H2O)
   __ChemFull(...)    See QChem docs for all the arguments
   __Correct()        Use 15 qubits+random circs to test teleport
   __Entangle1(cnt)   Run n qubit entanglement circuit (for timing purposes)
   __Entangle2(cnt)   Entangle1 with compiled and optimized circuits
   __Entangles()      Draw and run 100 instances of 16 qubit entanglement test
   __EntEnt()         Entanglement entropy test
   __EIGS()           Check eigenvalues using ARPACK
   __EPR()            Draw EPR circuit (.htm and .tex files)
   __Ferro(false,true) Test ferro magnetic coupling with true=full, true=runonce
   __JointCNOT()       Run CNOTs defined by Joint measurements
   __Kraus(cnt,AD,DP,v)Teleport w/noise for iters,pAmpDamp,pDePolar,verbose"
   __Noise1(d,i,p)     d=# of idle gates, i=iters, p=probOfNoise
   __NoiseAmp()        Amplitude damping (non-unitary) noise
   __QECC()            Test teleport with errors and Steane7 code (gen drawing)
   __QFTbench()        Benchmark QFT used in Shor (func,circ,optimized)
   __QLSA()            Test of HHL linear equation solver
   __QuAM()            Quantum Associative Memory
   __QWalk(typ)        Walk tiny,tree,graph or RMat file with graph information
   __Ramsey33()        Try to find a Ramsey(3,3) solution
   __SG()              Test spin glass model
   __Shor(N,true)      Factor N using Shor's algo false=direct true=optimized
   __show("str")       Test routine to echo str and then exit
   __Steane7()         Test basic error injection in Steane7 code
   __Teleport()        Draw and run original, circuit and grown versions
   __TSP(5)            Try to find a Traveling Salesman solution for 5 to 8 cities
   __UserSample()      Stub for placing user code (in Main.fs)
```

Example 1: Command Line Syntax

Any function in the system that has the `[<LQD>]` attribute may be entered on the command line with:

```
LIQUiD <function>(<arg>,…)
```

Arguments may be ints, floats, strings (with or without double quotes) and Booleans (true/false). All of the listed tests are defined in this way.

If there are spaces required in a string argument, place the entire string in double quotes ("). If you need to pass a comma, use "`\,`" since it will otherwise be used to separate arguments to the function. For example, to print out a string that contains commas and spaces, you could type:

```
> Liquid __show("This\, is a \"function call\"\, with commas")

0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.0/This, is a "function call", with commas
0:0000.0/=============== Logging to: Liquid.log closed ================
```

Example 2: Command line function execution

Output from LIQ$Ui|\rangle$ is typically generated by the `show` command which takes the same arguments as `printfn` (implied newline at the end of a line). This routine provides several benefits over and above the standard `printf` family:

1. Output is thread safe and guaranteed to output the complete line without being interrupted by output from other threads.

2. All output is duplicated in the log file (or may be sent to the log file without being sent to the console).

3. Each line is prepended with a thread ID to identify the source (`0:` above) and elapsed time (in minutes) since the start of this run (`0000.0`).

To create a new executable, the easiest approach is to place your code in the provided `Main.fs` file in the `Liquid` sub-directory and then build the entire solution (`liquid.sln`) with Visual Studio. This will create a new `Liquid` executable that has all of the capabilities of the original (argument parsing, scripts, ensemble execution…) as well as being able to call any of your functions from the command line that have the `[<LQD>]` attribute defined. The provided sample is very simple:

```
[<LQD>]
let __UserSample() =
    show "This module is a good place to put compiled user code"
```

Example 3: UserSample - Extending the simulator

Any routine that has the attribute `[<LQD>]` is callable from the command line. User routines do not need to begin with underscores (these are only used to delineate built-in sample routines). A full description of how to compile code will be given in a later chapter. For now, we'll focus on extending the simulator via scripts (`'liquid /s <script>.fsx'`).

The `TESTS` will be described in later sections of the manual (with sample code that generates them).

Creating a script

Scripts are F# source code files (ending in `.fsx`) and are executed against the LIQ$Ui|\rangle$ library. We'll work through a complete example to show how one might write a script to perform a computation. Any function delimited with the `[<LQD>]` attribute (described later) may be called from the command line (`__show` used above has this attribute).

Script files are very flexible and several examples are provided in the `samples` directory. We'll work through the `Entangle1.fsx` file now (details on the actual quantum calls will be filled in later). The first thing we'll need is a common header:

```
#if INTERACTIVE
#r @"..\bin\Liquid1.dll"
#else
namespace Microsoft.Research.Liquid // Tell the compiler our namespace
#endif

open System                         // Open any support libraries

open Microsoft.Research.Liquid      // Get necessary Liquid libraries
open Util                           // General utilities
open Operations                     // Basic gates and operations
```

Example 4: Script Header

Scripts may be run in several ways:

1. From within LIQ$Ui|\rangle$ via the `/s` command. This is covered by the `namespace` line (when running non-interactively).
2. After loading and compiling the script (in the previous step), you are left with a new `.DLL` that has your compiled code. You can efficiently execute this with the `/l` command (load a dll).
3. From `fsi` (the F# interpreter) as a complete command (e.g., `fsi -exec Entangle1.fsx`). The file will load into the interpreter, execute and then exit (this is what the `#if INTERACTIVE` is for).
4. From `fsi` interactively (e.g., `fsi --use:Entangle1.fsx`). The file will load and execute but the user is left inside the F# interpreter where all of the loaded functions (as well as all of LIQ$Ui|\rangle$ are available.

To use `fsi`, you will need to be in the `samples` directory and it will need to be in your path. A typical location to find it would be:

```
"%ProgramFiles(x86)%\Microsoft SDKs\F#\4.0\Framework\v4.0\fsi.exe"
```

All necessary System and LIQ$Ui|\rangle$ modules are opened in the header. The header is followed by the code we wish to define and execute. Usually, we put this in a module called `Script` where we define any number of routines, flagging any that we wish to call from the command line with the `[<LQD>]` attribute:

```
[<LQD>]
let Entangle1(entSiz:int) =
    logOpen "Liquid.log" false

    let qt      = QubitTimer()

    let ket     = Ket(entSiz)     // Start with a full sized state vector
    let _       = ket.Single()
    qt.Show "Created single state vector"
```

Example 5: User script module

Here we opened a log file and defined a timing function that will let us print out statistics as we run (the `QubitTimer` definition may be found in the script file). We then make a state vector (`Ket`) that will represent all of our qubits and force it to full size (`ket.Single()`) converting the efficient (unentangled) representation of the state vector that LIQ$Ui|\rangle$ normally uses to a fully realized state vector (2^N in size). This is being used to show what are timings are like with fully entangled state vectors. If you comment out this line, everything will run much faster.

Now let's do the rest of the entanglement timing test:

```
    let qs      = ket.Qubits
    H qs                                  // Hadamard the first qubit
    qt.Show "Did Hadamard"
    let q0  = qs.Head
    for i in 1..qs.Length-1 do
        let q   = qs.[i]
        CNOT[q0;q]                        // Entangle all the other qubits
        let str = sprintf "  Did CNOT: %2d" i
        qt.Show(str,i,(i=qs.Length-1))

    M >< qs                               // Measure all the qubits
    qt.Show("Did Measure",qs.Length)
    show ""
```

Example 6: Entanglement circuit

We first ask the state vector for the qubits that it represents (`qs:Qubits`). The qubits are always represented as an F# list and all qubit operations (gates) always have a qubit list as their last argument. In addition, these functions never return a value because the qubits are maintained in their own state (the `Ket` vector). Now we apply a Hadamard gate (`H`) to the first qubit in the list. By convention, all gate operations will apply to the head of the list, using as many qubits as they need (e.g., `CNOT` will apply to the first two). This allows lists of any length to be used (useful for register operations) as well as allowing functions that operate on variable numbers of qubits (e.g., Quantum Fourier Transform (`QFT`)).

So far, we've called one quantum function (Hadamard) on the first qubit. Now we perform a loop to entangle (`CNOT`) the first qubit with the remaining qubits. This is an expensive operation and so we print out the timing statistics as we do each qubit. Finally, we measure all the qubits (`M >< qs`) using a built-in LIQ$Ui|\rangle$ function, the "bowtie" that applies a gate (`M`) to all the qubits in a list.

We have several ways to run this sample. The easiest is to use the built-in version that's already in LIQ$Ui|\rangle$ (assuming that we're in the samples directory):

```
> ..\bin\Liquid __Entangle1(22)

0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.0/
0:0000.0/ Secs/Op  S/Qubit  Mem(GB) Operation
0:0000.0/ -------  -------  ------- ---------
0:0000.0/   0.788    0.788    0.365 Created single state vector
0:0000.0/   0.522    0.522    0.365 Did Hadamard
0:0000.0/   0.485    0.485    0.366   Did CNOT:  1
0:0000.0/   0.995    0.498    0.367   Did CNOT:  2
0:0000.1/   1.491    0.497    0.368   Did CNOT:  3
0:0000.1/   1.949    0.487    0.369   Did CNOT:  4
0:0000.1/   2.433    0.487    0.370   Did CNOT:  5
0:0000.1/   2.906    0.484    0.370   Did CNOT:  6
0:0000.1/   3.378    0.483    0.370   Did CNOT:  7
0:0000.1/   3.835    0.479    0.370   Did CNOT:  8
0:0000.1/   4.301    0.478    0.371   Did CNOT:  9
0:0000.1/   4.766    0.477    0.372   Did CNOT: 10
0:0000.1/   5.230    0.475    0.373   Did CNOT: 11
0:0000.1/   5.697    0.475    0.374   Did CNOT: 12
0:0000.1/   6.165    0.474    0.375   Did CNOT: 13
0:0000.1/   6.624    0.473    0.375   Did CNOT: 14
0:0000.1/   7.089    0.473    0.376   Did CNOT: 15
0:0000.2/   7.559    0.472    0.377   Did CNOT: 16
0:0000.2/   8.020    0.472    0.378   Did CNOT: 17
0:0000.2/   8.488    0.472    0.379   Did CNOT: 18
0:0000.2/   8.948    0.471    0.380   Did CNOT: 19
```

```
0:0000.2/   9.419    0.471    0.381   Did CNOT: 20
0:0000.2/   9.873    0.470    0.382   Did CNOT: 21
0:0000.2/   0.700    0.032    0.431 Did Measure
0:0000.2/
0:0000.2/=============== Logging to: Liquid.log closed ================
```

Example 7: Running Entangle1

The numbers in front of the slash (`0:0000.2/`) are output on every `show` command (superset of `printfn`). The number in front of the colon is a thread ID (useful when using multiple threads to output… also `show` is fully thread safe and won't create partial outputs). The number after the colon is the number of minutes (.tenths) since the app started.

In this case, we ran for .2 minutes or approximately 12 seconds total. If we wanted to run our script (Entangle1.fsx) from the `samples` directory, we would enter:

```
> ..\bin\Liquid /s Entangle1.fsx Entangle1(22)
0:0000.0/=============== Logging to: Liquid.log opened ================
Microsoft (R) F# Compiler version 14.0.23020.0
Copyright (c) Microsoft Corporation. All Rights Reserved.
0:0000.0/
0:0000.0/ Secs/Op  S/Qubit  Mem(GB) Operation
0:0000.0/ -------  -------  ------- ---------
0:0000.0/   0.803    0.803    0.356 Created single state vector
0:0000.0/   0.601    0.601    0.356 Did Hadamard
0:0000.0/   0.495    0.495    0.357   Did CNOT:  1
0:0000.1/   0.967    0.484    0.358   Did CNOT:  2
0:0000.1/   1.433    0.478    0.358   Did CNOT:  3
...
0:0000.2/   9.170    0.483    0.373   Did CNOT: 19
0:0000.2/   9.645    0.482    0.374   Did CNOT: 20
0:0000.2/  10.107    0.481    0.375   Did CNOT: 21
0:0000.2/   0.723    0.033    0.431 Did Measure
0:0000.2/
0:0000.2/=============== Logging to: Liquid.log closed ================
```

Example 8: Entangle1 running from a script

The main difference is that now we had to call the F# Compiler to build a DLL that we loaded back into our image for execution. This is why it took slightly longer to run. However, now that we have the DLL, we can just do:

```
> ..\bin\Liquid /l Entangle1.dll Entangle1(22)
```

Example 9: Entangle1 from a pre-built DLL

This will run the same without requiring a compilation step. One other option we have is to run using the F# interpreter. Provided that fsi.exe is in our path, we can just do: **`fsi --exec Entangle1.fsx`** and the script will run using the "`INTERACIVE`" defaults that are in the file.

Another option is to take the circuit we've defined and compile the gates into a data structure called a circuit. This is demonstrated in the `Entangle2` routine. Here, we define all the same quantum operations in a function:

```
        let ops (qs:Qubits) =
H qs
let q0  = qs.Head
for i in 1..qs.Length-1 do CNOT !!(qs,0,i)
M >< qs                                          // Measure all the qubits
```

Example 10: Single Entanglement circuit

We've made use of another built-in helper function (`!!`) which will pull out the indexed qubits from a list and create a new list. If we call `ops` directly, we will get the exact results we saw in `Entangle1`. However, we can turn this into a circuit data structure and then execute the data structure:

```
let circ    = Circuit.Compile ops qs     // Compile and run circuit
circ.Run qs
let circ2   = circ.GrowGates(ket)        // Optimize and run circuit
circ2.Run qs
```

Example 11: Circuit Compilation and running

The circuit is actually slightly slower than the direction function calls. However, circuits are useful for many purposes. Since they are a data structure, the can be analyzed, modified, optimized, replaced, rendered as drawings, exported to other systems and simply run as if they were the original circuit. The second two lines in the example create an optimized version of the circuit. Here's what happens when we run all three variants:

```
0:0000.0/ Secs/Op  S/Qubit  Mem(GB) Operation
0:0000.0/ -------  -------  ------- ---------
0:0000.2/  11.474   11.474    0.419 Straight function calls
0:0000.2/   0.952    0.952    0.366 Compile cost
0:0000.4/  12.746   12.746    0.419 Compiled circuit run time
0:0000.5/   0.878    0.878    0.544 Optimization cost
0:0000.5/   2.330    2.330    0.362 Optimized circuit run time
```

Example 12: Entangle1 with optimization run times

Here you can see that the compiled circuit is slightly slower, but the optimized version is ~5 times faster. LIQ$Ui|\rangle$ contains a large number of user accessible optimizations to make quantum simulations as efficient as possible.

To see the generated circuit, we could call the Dump command:

```
circ.Dump(showLogInd)
```

Example 13: Dump of Teleport Circuit

The `Dump` command is defined throughout the system on most LIQ$Ui|\rangle$ data types and may be nested. For this reason it takes up to two arguments: a dumping function (`showLogInd`) which writes the output to the log in an indented manner and a starting indentation level (`showInd` will output to the log and the console). The result for Entangle2(2) is:

```
Circuit dump (in Liquid.log):
SEQ
```

```
  APPLY
    GATE H is a  (Normal)
      0.7071 0.7071
      0.7071 -0.7071
    WIRE(Id:0)
    WIRE(Id:1)
  APPLY
    GATE CNOT is a Controlled NOT (Normal)
      1 0 0 0
      0 1 0 0
      0 0 0 1
      0 0 1 0
    WIRE(Id:0)
    WIRE(Id:1)
  APPLY
    GATE Meas is a Collapse State (Measure(,Joint=))
      1 0
      0 1
    WIRE(Id:0)
  APPLY
    GATE Meas is a Collapse State (Measure(,Joint=))
      1 0
      0 1
    WIRE(Id:1)
```

Example 14: Dump of Entangle2(2) Circuit

Another manipulation would be to take the circuit and combine multiple gates together into larger unitary representations. This allows for more efficient simulation and is known in LIQ$Ui|\rangle$ as "growing gates". If we dump out `Entangle2(4)`, and then look at the result, we see that the Hadamard gate and all the CNOT gates have been reduced to a single Unitary:

```
 APPLY
    GATE B00ECF3 is a B00ECF3 (Normal)
      0.7071 0 0 0 0 0 0 0 0.7071 0 0 0 0 0 0 0
      0 0.7071 0 0 0 0 0 0 0 0.7071 0 0 0 0 0 0
      0 0 0.7071 0 0 0 0 0 0 0 0.7071 0 0 0 0 0
      0 0 0 0.7071 0 0 0 0 0 0 0 0.7071 0 0 0 0
      0 0 0 0 0.7071 0 0 0 0 0 0 0 0.7071 0 0 0
      0 0 0 0 0 0.7071 0 0 0 0 0 0 0 0.7071 0 0
      0 0 0 0 0 0 0.7071 0 0 0 0 0 0 0 0.7071 0
      0 0 0 0 0 0 0 0.7071 0 0 0 0 0 0 0 0.7071
      0 0 0 0 0 0 0 0.7071 0 0 0 0 0 0 0 -0.7071
      0 0 0 0 0 0 0.7071 0 0 0 0 0 0 0 -0.7071 0
      0 0 0 0 0 0.7071 0 0 0 0 0 0 0 -0.7071 0 0
      0 0 0 0 0.7071 0 0 0 0 0 0 0 -0.7071 0 0 0
      0 0 0 0.7071 0 0 0 0 0 0 0 -0.7071 0 0 0 0
      0 0 0.7071 0 0 0 0 0 0 0 -0.7071 0 0 0 0 0
      0 0.7071 0 0 0 0 0 0 0 -0.7071 0 0 0 0 0 0
      0.7071 0 0 0 0 0 0 0 -0.7071 0 0 0 0 0 0 0
    WIRE(Id:0)
    WIRE(Id:1)
    WIRE(Id:2)
    WIRE(Id:3)
```

Example 15: Optimized Entanglement circuit

One other example shown in `Entangle1.fsx` is drawing the circuits we've created. This can be easily accomplished by calling the rendering package:

```
circ.Fold().RenderHT("Entangle2raw")
circ2.Fold().RenderHT("Entangle2opt")
```

Example 16: Rendering a circuit

`RenderHT` creates both HTML (SVG) and LaTeX (TikZ) drawings. The `Fold` call shifts everything to the left for nicer output. Here's what the created drawing looks like for `Entangle2(10)` for the raw and optimized circuits:

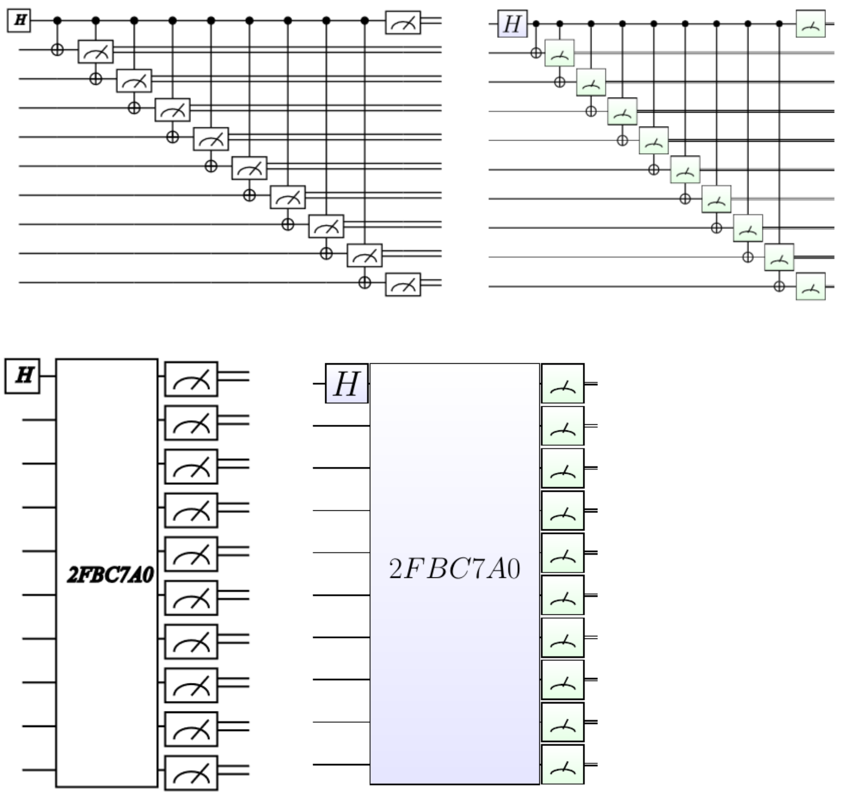


Figure 3: Entangle2(10) circuit drawing (HTML and LaTeX)

An example of a very sophisticated built-in test is Shor's algorithm. It is called with two parameters: The number to be factored and whether to optimize (grow) the circuit. If we hand it an illegal number to factor, we're given a table of sample legal numbers to try:

```
> Liquid.exe __Shor(1,true)
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.0/Legal numbers include:
0:0000.0/ 4 bits:     15
0:0000.0/ 5 bits:     21
0:0000.0/ 6 bits:     63    57    55    51    45    39    35    33
0:0000.0/ 7 bits:    123   119   117   115   111   105    99    95    93    91
0:0000.0/ 8 bits:    255   253   249   247   245   237   235   231   225   221
0:0000.0/ 9 bits:    511   507   505   501   497   495   493   489   485   483
0:0000.0/10 bits:   1023  1017  1015  1011  1007  1005  1003  1001   999   995
0:0000.0/11 bits:   2047  2045  2043  2041  2037  2035  2033  2031  2025  2023
0:0000.0/12 bits:   4095  4089  4087  4085  4083  4081  4077  4075  4071  4069
0:0000.0/13 bits:   8189  8187  8185  8183  8181  8177  8175  8173  8169  8165
0:0000.0/14 bits:  16383 16379 16377 16375 16373 16371 16367 16365 16359 16357
0:0000.0/15 bits:  32767 32765 32763 32759 32757 32755 32753 32751 32747 32745
```

```
0:0000.0/16 bits:  65535 65533 65531 65529 65527 65525 65523 65517 65515 65513
```

Example 17: Sample numbers to factor

Factoring 55 yields:

```
> Liquid.exe __Shor(55,true)
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.0/======== Doing Shor Round =========
0:0000.0/      55 = N = Number to factor
0:0000.0/       2 = a = coPrime of N
0:0000.0/       6 = n = number of bits for N
0:0000.0/      64 = 2^n
0:0000.0/      15 = total qubits
0:0000.0/      29 = starting memory (MB)
0:0000.0/  30.66% = prob of random result (1256/4096)
0:0000.0/  38.69% = prob of Shor (worst case)
0:0000.0/         - Compiling circuit
0:0000.0/0.000945 = mins for compile
0:0000.0/   30540 = cnt of gates
0:0000.0/    7351 = cache hits
0:0000.0/     143 = cache misses
0:0000.0/      36 = compiled memory (MB)
0:0000.0/         - Wrapping circuit pieces
0:0000.0/       8 = wires have possibles:158 (prv=  0GB did=     0 big=     0)
0:0000.0/       9 = wires have possibles:153 (prv=  0GB did=     5 big=    60)
0:0000.0/      10 = wires have possibles:136 (prv=  0GB did=    22 big=   124)
0:0000.0/      11 = wires have possibles:111 (prv=  0GB did=    47 big=   217)
0:0000.0/      12 = wires have possibles:109 (prv=  0GB did=    49 big=   318)
0:0000.0/      13 = wires have possibles:105 (prv=  0GB did=    53 big=   416)
0:0000.0/      14 = wires have possibles:103 (prv=  0GB did=    55 big=   514)
0:0000.0/      15 = wires have possibles:103 (prv=  0GB did=    55 big=   612)
0:0000.0/      16 = Ran out of wires
0:0000.0/           MM: g:    55 b:   714  13=2 12=4 11=2 10=25 9=17
0:0000.0/0.008407 = mins for growing gates
0:0000.0/    1236 = cnt of gates
0:0000.0/     165 = grown memory (MB)
0:0000.0/        Bit:  11 [MB:  177 m=1]
0:0000.0/        Bit:  10 [MB:  236 m=1]
0:0000.0/        Bit:   9 [MB:  259 m=0]
0:0000.1/        Bit:   8 [MB:  272 m=0]
0:0000.1/        Bit:   7 [MB:  295 m=1]
0:0000.1/        Bit:   6 [MB:  313 m=1]
0:0000.1/        Bit:   5 [MB:  352 m=0]
0:0000.1/        Bit:   4 [MB:  329 m=0]
0:0000.1/        Bit:   3 [MB:  328 m=1]
0:0000.1/        Bit:   2 [MB:  337 m=1]
0:0000.1/        Bit:   1 [MB:  355 m=0]
0:0000.2/        Bit:   0 [MB:  345 m=1]
0:0000.2/0.070623 = mins for running
0:0000.2/ 8.75308 = Total Elapsed time (seconds)
0:0000.2/      15 = Max Entangled
0:0000.2/       0 = Gates Permuted
0:0000.2/    1191 = State Permuted
0:0000.2/      80 = None  Permuted
0:0000.2/    2867 = m = quantum result
0:0000.2/0.699951 = c = 2867/4096 =~ 7/10
0:0000.2/       5 = 10/2 = exponent
0:0000.2/      33 = 2^5 + 1 mod 55
0:0000.2/      31 = 2^5 - 1 mod 55
0:0000.2/      11 = factor = max(11,1)
0:0000.2/CSV N a m den f1 f2 good,55,2,2867,10,11,5,1
0:0000.2/GOT:   55=  11x   5 co=    2 n,q= 6,15 mins=0.15 SUCCESS!!
```

Example 18: Factoring 65 with Shor's algorithm

Some items to note from this run:

1. The circuit had 30,540 basic gates that were reduced to 1,236 grown gates

2. The entire run took 8.75 seconds on a laptop. If we had run this without optimization (`__Shor(55,false)`) it would have taken 1.6 minutes (11x slower!)
3. We succeeded: 55 =11 x 5 even though the random probability of success was only 31%.

Advanced Topics

Quantum Error Correction

Several advanced techniques are built into the simulator beyond simple gates and circuits. In the following chapters we will delve in to them in more detail. For now, let's just show how they may be accessed directly.

Let's use a circuit for `teleport` as our basic circuit (Figure 4). This circuit is the "hello world" of quantum computing and is fully described in Nielsen and Chuang. It will take a *Src* qubit in any state and teleport it to the *Dest* qubit (no matter how physically far apart they are). We can run this example directly from LIQ$Ui|\rangle$ as `__Teleport()` or as the script in the samples directory as `Teleport.fsx`. In either case, the example shows teleportation of several values via function calls, a compiled circuit and an optimized circuit). It also generates several renderings (in HTML/SVG and LaTeX/TikZ). The one in the figure shown here is `Teleport_CF.tex` (CF = Circuit Folded).

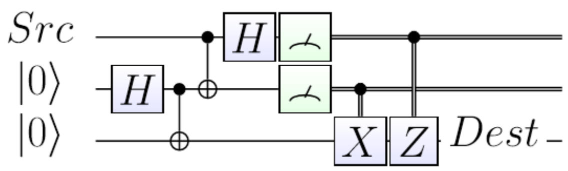


Figure 4: Basic Teleport Circuit

Now, we'd like to add error correction to our operation. The first thing we're going to do is define two functions and their circuits that teleport the values $|0\rangle$ and $|1\rangle$ respectively (the complete example is in `QECC.fsx`):

```
// Teleport for Stabilizers and QECC
let tele0 (qs:Qubits)   = teleport qs; M [qs.[2]]
let tele1 (qs:Qubits)   = X qs; teleport qs; M [qs.[2]]
let k                   = Ket(3)
let tgtC0               = Circuit.Compile tele0 k.Qubits
let tgtC1               = Circuit.Compile tele1 k.Qubits
```

Example 19: Teleport circuit definitions

Quantum Error Correction Codes (or QECC) are defined as a user extensible class (see the Extensions chapter for detail). One of the built-in codes is called Steane7 which is a Caldebrank-Shor-Steane 7 qubit code (CSS Steane [[7,1,3]]). In this code, each logical qubit gets expanded into 7 physical qubits which can be used to detect errors and correct them.

Following `tgtC0`, we'll convert from a logical circuit into a physical circuit with:

```
let s7      = Steane7(tgtC0)
let s7C     = s7.Circuit
s7C.Fold().RenderHT("QECC_min",0,100.0,33.0)
s7C.Fold().RenderHT("QECC_all",1,50.,20.)
```

Example 20: Mapping a Logical to a Physical circuit

The third line create a drawing of a high level view of the circuit (100% in on figure at a scale of 33%):

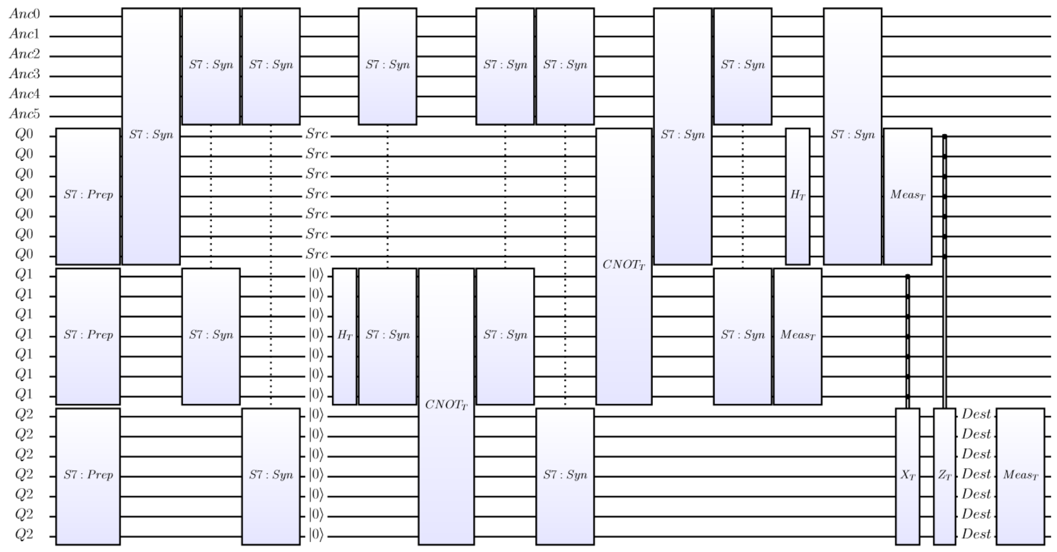


Figure 5: High level Teleport QECC circuit (LaTeX version)

The last line generates a low level view (Level=1, 50% of the circuit is in each figure, scale to 20% of the default size):

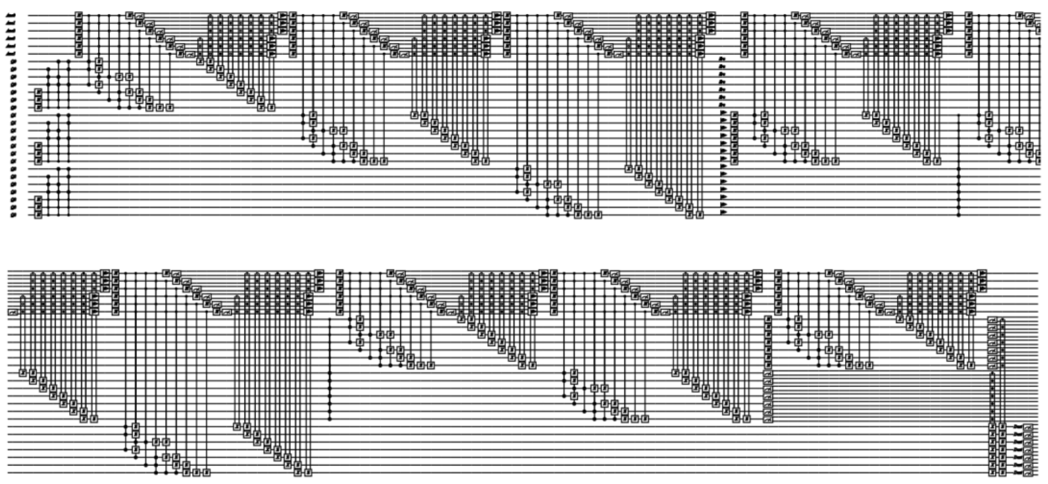


Figure 6: Low level Teleport QECC circuit (HTML version)

We started with 3 qubits and a small circuit… To do error correction we had to encode each of our 3 logical qubits into 7 physical qubits (21 qubits) + 6 qubits for computation (Ancilla). 27 qubits is almost at our limit… so we can't go much beyond this before we run out of memory... this will lead us to the next simulator type – Stabilizers.

The circuit also got a lot more complicated because we had to provide:

1. State Preparation circuits that convert a logical qubit into 7 encoded physical qubits
2. New gates that replace the old ones, operating on 7 encoded qubits for every input/output logical qubit, returning an encoded result.
3. Syndrome measurement circuitry that determines which if any errors have occurred.
4. Error correction circuitry that fixes bad codes.

If we run this circuit, it will work and provide the correct answers… but there are two other things we can do with it that are more interesting. First, we'd like to inject errors. This can be done with:

```
let errC,stats  = s7.Inject 0.01
```

Example 21: Injecting errors for QECC

The probability of an error is specified `0.01` for any wire between gates and errors are injected (a randomly chosen X, Y or Z gate) throughout the circuit. This generates a new circuit (`errC`) as well as a list of statistics (`stats`) that tell us the number of `X`, `Y` and `Z` gates inserted into the circuit. Of course, this new circuit can be run and analyzed as well as manipulated further.

This output states that the new circuit contains an X error and a Y error. If we draw this circuit (at level 0) we can easily see where the errors were inserted. Here's the section that contains the errors:

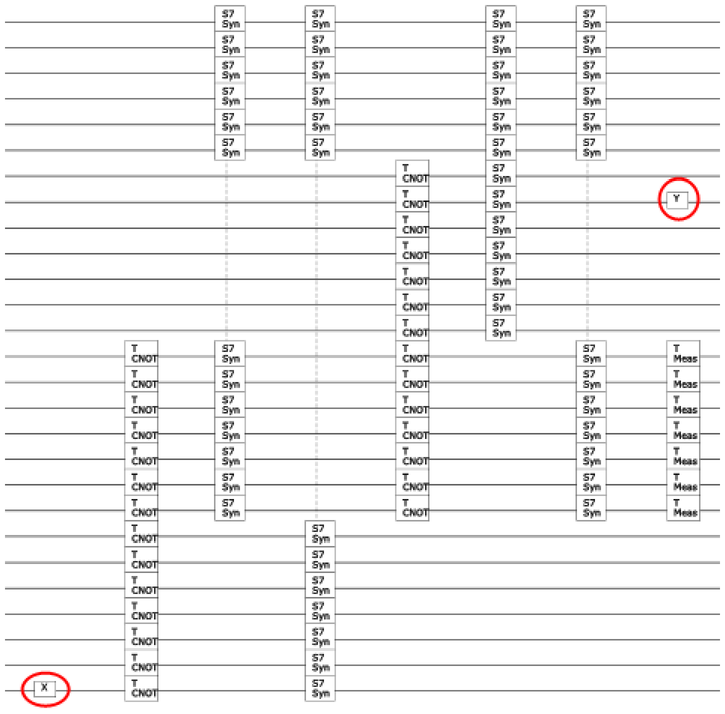


Figure 7: Error Insertion

We can run the circuit directly at this point, but there's a better alternative. We can switch to a Stabilizer simulation engine (since the circuit only contains legal gates for that simulator (by design)) by issuing the command:

```
let stab            = Stabilizer(errC,ket)
stab.Run()
```

Example 22: Running a Stabilizer simulation

This will run very quickly (it can handle 10s of thousands of qubits, but only a limited gate set). However, we now have the problem of interpreting the results. We need to convert physical qubits back to logical qubits. This can be done with statements like:

```
let bit0,dist0  = s7.Log2Phys 0 |> s7.Decode
let bit1,dist1  = s7.Log2Phys 1 |> s7.Decode
let bit2,dist2  = s7.Log2Phys 2 |> s7.Decode
```

```
show "InjectedXYZ(%d,%d,%d) Fixes=%d (%4s,%4s,%4s)
                                  dist=(%d,%d,%d)%s"
    stats.[0] stats.[1] stats.[2]
    s7.NumFixed
    (bit0.ToString()) (bit1.ToString()) (bit2.ToString())
    dist0 dist1 dist2
    (if bit2 <> inp then " <====== BAD" else "")
```

Example 23: Obtaining QECC results

The first three statements pick out the physical qubits that represent a logical qubit (`Log2Phys`) and then converts them to the closest code value (`Decode`) returning the bit that best represents the code and the classical distance from the code. We then **`show`** for output the injected errors (`stats`) the number of times we needed to apply a fix in the quantum circuit (`NumFixed`), the actual bits we decoded and their distance from the correct code value. Typical output looks like this:

```
InjectedXYZ(0,1,1) Fixes=3 (Zero,Zero,Zero) dist=(0,0,0)
```

Example 24: Output of Stabilizer run

Here we injected a Y and a Z error, fixed three errors that propagated through the circuit, measured the three logical qubits as al `Zero` and had no classical decoding errors (`0,0,0`)

The third simulation engine (Hamiltonian) will be described in its own chapter, however there is a sample script provided (`h2.fsx`) which will allow you to solve the ground state energy for an H_2 molecule (2nd Quantized Hamiltonian) and `Ferro.fsx` which demonstrates various ferro-magnetic chain examples (1st Quantized Hamiltonian).

Chapter 4

Serious Coding

Compilation

Compiliation via Visual Studio provides the most immersive environment for creating sophisticated circuits and simulations. It also provides a full interactive debugging environment. It is no more difficult to use than scripting mode and a sample project is provided to help getting started. This approach to a data specific language stems from the Language INtegrated Query (LINQ) model used for many other application areas in Microsoft .Net languages. LIQ$Ui|\rangle$ is <u>not</u> a LINQ language but shares many of the same goals. This section will heavily rely on the F# programming language. However, it will be presented in such a way that knowledge of any other high level language should be sufficient to understand the examples and to get started on your own quantum algorithm implementations.

Let's return to the teleport example from the previous chapter and re-create it directly in a Visual Studio project. If you open the provided solution (`Liquid.sln`), you'll find the top level application project (`Liquid`):

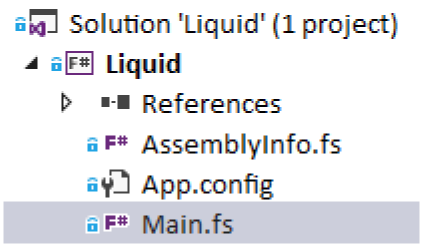


Figure 8: Liquid project

A sample file (`Main.fs`) is provided that you can edit yourself, or supplement with one or more files that will be called by `Main.fs` when you run LIQ$Ui|\rangle$. `Main.fs` contains a module named `UserSample` for this purpose; its contents are straight-forward:

```
module UserSample =
    open Util
    open Operations
    // Optional extras:
    //open Native              // Support for Native Interop
    //open HamiltonianGates    // Extra gates for doing Hamiltonian
simulations
    //open Tests               // All the built-in tests

    [<LQD>]
    let __UserSample() =
        show "This module is a good place to put compiled user code"
```

Figure 9: Main.fs

The header looks just like the script (`.fsx`) examples and we have one routine marked with the `[<LQD>]` attribute which will be callable from the command line. If we compile and run the system, we get:

```
> liquid UserSample()
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.0/This module is a good place to put compiled user code
0:0000.0/=============== Logging to: Liquid.log closed ================
```

Example 25: UserSample execution

One thing to note is that white-space indentation is significant in F#, this allows you to define blocks without the need for curly braces (or equivalent). Normally, statements are separated by newlines, but you are allowed to use semi-colon as well (to put multiple statements on the same line).

Data Types

We're now going to re-visit the basic data types of the system that we first summarized in Chapter 2 and go into more detail. Let's return to the `Teleport.fsx` sample script to show some of the data types. We're going to move to fully interpreted mode by executing:

> fsi --use:Teleport.fsx

Example 26: Starting Teleport in fully interactive mode

This assumes that fsi.exe is in your `PATH` and your current directory is the `Samples` directory. After the Teleport example runs, we are left in the F# interpreter where we now have a full LIQ$Ui|\rangle$ environment. A couple of basic notes about `fsi` are in order:

1. Comands are only executed after two semi-colons are entered (`;;`)
2. `#quit;;` will exit the interpreter

The first step is to open the Script module, defined in Teleport.fsx, so your code can access the teleport function without prefixing it with "Script.":

```
> open Script;;
```

Kets and Qubits
States and the parts

Now we'd like to create our own `Ket` vector of 3 `Qubits`:

```
> let ket = Ket(3);;

val ket : Ket =
  Ket of 3 qubits:
=== KetPart[ 0]: Qubits (High to Low): 2
          1
          0
=== KetPart[ 1]: Qubits (High to Low): 1
          1
          0
=== KetPart[ 2]: Qubits (High to Low): 0
          1
          0
```

Example 27: Creation of a Ket from fsi

Since fsi prints out the value of the last statement executed, we see that the `Ket` is made up of 3 `KetParts` each containing 1 qubit of state 0: $\begin{pmatrix}1\\0\end{pmatrix}$. `KetParts` represent entangled qubits, so since we started with 3 unentangled qubits, we have 3 `KetParts`. As we run circuits, qubits will be become more and more entangled and each individual `KetPart` will grow in size, while the number of them will decrease (there will be only one if all the `Qubits` are fully entangled). We can now set a variable to the list of qubits:

```
> let qs  = ket.Qubits;;

val qs : Qubit list =
  [    1|0>+      0|1>;   1|0>+      0|1>;  1|0>+      0|1>]
```

Example 28: Obtain Qubits from state

…and run teleport on them:

```
> teleport qs;;

val it : unit = ()
```

Example 29: Run teleport in fsi

Operations (gates and circuits) don't return a value, so all we see is `unit` (which means "nothing"). If however, we take a look at the individual qubits at this point, we see:

```
> for q in qs do show "q[%d]=%s" q.Id (q.ToString());;
0:0001.5/q[0]=      0|0>+      1|1>
0:0001.5/q[1]=      1|0>+      0|1>
0:0001.5/q[2]=      1|0>+      0|1>
```

Example 30: Printing out Qubit values

The output shows that `q0` got measured as a $|1\rangle$, `q1` got measured as a $|0\rangle$ and the message was returned as a $|0\rangle$. Not very impressive, but it is what we started with.

Now let's do the same circuit, but start from a less trivial state:

```
let ket = Ket(3)
let qs  = ket.Qubits
let a = sqrt(0.25)
let b = sqrt(0.75)
qs.[0].StateSet(a,0.0,b,0.0)
show "Input  message: %s" (qs.[0].ToString());;
teleport qs
show "Output message: %s [Measured: %d%d]"
      (qs.[2].ToString()) qs.[0].Bit.v qs.[1].Bit.v;;
```

Example 31: Teleport with an interesting initial state

This is the same as before, except we're starting from a new state (`StateSet`) and printing out the starting and ending states. The results are:

```
0:0014.1/Input  message:     0.5|0>+  0.866|1>
0:0014.2/Output message:     0.5|0>+  0.866|1> [Measured: 10]
```

Example 32: Results from non-trivial teleport

Now we've sent a non-trivial message and it arrived in good shape. Also, we randomly measured `q0` as a **One** which means we had to apply the controlled `Z` gate to recover the input message.

Circuits Compiling functions to circuits

Continuing our example, we'd like to compile our teleport function into a circuit for further manipulation (e.g., drawing, printing, parallelizing, optimizing…). To obtain a **Circuit** all we need is a state vector that defines our **Qubit**s and the function we wish to compile:

```
> let circ = Circuit.Compile teleport qs;;

val circ : Circuit =
  Seq
    [Apply (GATE Src is a Label qubit: Src (String), Mat(2),[0]);
     Apply (GATE |0> is a Label qubit: |0> (String), Mat(2),[1]);
     Apply (GATE |0> is a Label qubit: |0> (String), Mat(2),[2]);
     Apply (GATE H is a  (Normal), Mat(2),[1; 2]);
     Apply (GATE CNOT is a Controlled NOT (Normal), Mat(4),[1; 2]);
     Apply (GATE CNOT is a Controlled NOT (Normal), Mat(4),[0; 1; 2]);
     Apply (GATE H is a  (Normal), Mat(2),[0; 1; 2]);
     Apply (GATE Meas is a Collapse State (Measure), Mat(2),[1]);
     BitCon
       (GATE BitContol is a Bit control Qubit operator (BitControl(1)), Mat(0),
        [1; 2],<fun:op@183>,
         Apply (GATE X is a Pauli X flip (Normal), Mat(2),[2]));
     Apply (GATE Meas is a Collapse State (Measure), Mat(2),[0]);
     BitCon
       (GATE BitContol is a Bit control Qubit operator (BitControl(1)), Mat(0),
        [0; 2],<fun:op@183>,
         Apply (GATE Z is a Pauli Z flip (Normal), Mat(2),[2]));
     Apply (GATE Dest is a Label qubit: Dest (String), Mat(2),[2])]
```

Example 33: Circuit data structure

Here we see the data structure of the circuit we've created. `Circuits` may contain the following elements:

1. `Seq`: List of `Circuits` to execute in sequential order
2. `Par`: List of `Circuits` that may be executed in parallel
3. `Apply`: Application of a `Gate` to a set of wires (`Qubits`)
4. `Ext`: A `Gate` that extends the meaning of another `Gate`
5. `BitCon`: List of binary wires and a function on them that determine if a sub-`Circuit` is applied (Binary Control)
6. `Wrap`: Meta `Gate` that contains a `Circuit` of other `Gates`
7. `Empty`: Denotes an empty `Circuit`

This is fairly simple structure for an abstract syntax tree (AST) but is both sophisticated enough to hold all of our desired circuits and is still simple enough for the user to parse and manipulate easily (more on this in the Circuit Manipulation chapter)

Bits Measured values

Mentioned several times previously are items that have the `Bit` data type. This is not a standard binary value or replaceable by a Boolean. It specifically refers to a quantum value (`Qubit`) that has been measured, or a true unmeasured `Qubit` if the value is `Unknown`. Qubits maintain knowledge that they've been measured and can no longer be used in unitary operations. They may be brought "back to life" by non-unitary gates (described later in this chapter) and are also used for initializing unentangled qubits by referring directly to `Zero` (initializes $|0\rangle$) or `One` (initializes $|1\rangle$).

Gates Fundamental elements

Gates themselves have a specific structure that allows for re-use in many ways. We'll go into all the options in the chapter on extending the simulator, but here's an example of taking our teleport function and turning into a first-class gate:

```
> let teleGate    =
        let gate (qs:Qubits) =
            new Gate(
                Qubits  = 3,
                Name    = "teleGate",
                Op      = WrapOp teleport
            )
        fun (qs:Qubits) -> (gate qs).Run qs;;
```

Example 34: Creating teleport as a new Gate

Now we can run this new gate like any other:

```
> ket.Reset 3 |> teleGate;;
val it : unit = ()
```

Example 35: Creating a simple gate

A few non-obvious things were done in this one line. Since the Ket vector we used last time was left in an unusable state (entangled, measured…) we performed a `Reset` on the state re-initializing it to a known number of `Qubits` (3) in a known state (by default $|000\rangle$). The `Reset` returned a list of qubits, so we passed it to our new `Gate` for execution. Now that our teleport function is a `Gate`, we can define drawing instructions for it, it can be built into larger gates and it is manipulable inside of compiled circuits.

Built-in Gates

What's available?

Gates can range from a simple unitary matrix definition to large complex pieces of code that dynamically decide what to do at runtime. In this section we'll explore the various kinds of gates already available in the system and in the following chapter we'll describe how to define new ones.

The most straight-forward gates are those that represent standard unitary operations. These include:

- Hadamard (`H`) which takes an input qubit and rotates the basis: $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$
- Pauli NOT gate (`X`) which performs a bit-flip: $\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$
- Pauli Y gate (`Y`): $\begin{bmatrix}0 & -i\\i & 0\end{bmatrix}$
- Pauli phase flip gate (`Z`): which changes the sign of $|1\rangle$: $\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$
- Pauli identity gate (`I`): which does nothing: $\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$
- Phase gate (`S`) which flips the phase of a qubit: $\begin{bmatrix}1 & 0\\0 & i\end{bmatrix}$

- $\frac{\pi}{8}$ phase gate (`T`): $\begin{bmatrix}1 & 0\\0 & e^{i2\pi/8}\end{bmatrix}$
- General rotation gate (`R k`): $\begin{bmatrix}1 & 0\\0 & e^{i2\pi/2^k}\end{bmatrix}$
- Eigenvalue measuring gate (`U k`). K = fraction of 2π
- Controlled-Not gate (`CNOT`): First qubit is control, second qubit is flipped if first qubit is a $|1\rangle$: $\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$
- Swap two qubits (`SWAP`): $\begin{bmatrix}1 & 0 & 0 & 0\\0 & 0 & 1 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\end{bmatrix}$
- Toffoli gate (`CCNOT`): Controlled-Controlled-Not. Operates on 3 qubits

In addition, several non-unitary and parameterized gates are provided:

- Measurement (`M`): Measures the first qubit in the list and collapses its state to one of $|0\rangle$ or $|1\rangle$ (bit values **Zero** or **One**) with probabilities based on the state before measurement.
- Reset (`Reset Bit):` Take a measured **`Qubit`** and re-animate it as a **`Qubit`** with state of $|0\rangle$ or $|1\rangle$ based on `Bit` value provided.
- Restore (`Restore`): Take the first qubit in the list (which must be measured), get its **Bit** value and uses that to do a `Reset`.
- Label (`Label string`) will place a label in the circuit on the head qubit at this point in time. There are several variants of this gate, The variants are of the form: Label<typ>, where <typ> is one of:
  - U - float the label upward
  - D - float the label downward
  - L - float the label left (used at beginning of circuits)
  - R - float the label right (used at end of circuits).
  - C - float the label to the center of the wire
  - CD - float the label to the center and down
  - Raw - outputs raw LaTeX at this point
- Native (`Native (Qubits->unit)`): Apply the native function at this point in the circuit. This allows for native code to be applied at any desired point even after functions have been compiled to circuits. This is especially useful for saving intermediate values and debugging.
- Native Debug (`NativeDbg (Qubits->unit)`): Same as `Native` but doesn't show up in Circuit diagrams.
- Bit Control (`BC gate`). This gate reads the first qubit in the list (which must be measured). If it's a **One** then apply `gate` to the remaining qubits in the list.
- Arbitrary Bit Control (`BCany gate`). This is the same as the `BC` gate except that it takes a count of how many classical bits to use (measured qubits) and

a function that will receive the qubits and return a Boolean on whether or not to execute the quantum gate.

- Adjoint (`Adj gate`): Take the unitary matrix in `gate` and performs an adjoint operation on it before applying the matrix to the qubit list. This only works if the provided `gate` is defined by a unitary matrix (of course).
- Control gate (`Cgate gate`). This may be thought of as the quantum equivalent of `BC`. Take the matrix in `gate` and expand it to include a control line. Then apply the resulting gate to the qubit list. For example: `Cgate X` is the same thing as `CNOT`.
- Control Control gate (`CCgate gate`). Take the matrix in `gate` and expand it to include two control lines. Then apply the resulting gate to the qubit list. For example: `CCgate X` is the same thing as `CCNOT`.
- Transverse Gate (`Transverse`). Take any other gate and convert it into its transverse equivalent (for QECC).
- Transverse Binary Control (`T_BC`). Convert a binary control gate into its transverse equivalent (for QECC).

There are also sets of specialized gates. The largest set is for Hamiltonian operations (these are in the model `HamiltonianGates`):

- Couple two σ_z operations (`ZZ`).
- Rotated Pauli (`Rpauli`). Takes a Pauli gate (`X,Y,Z`) and rotates it to an arbitrary angle (R_x, R_y, R_z).
- Rotate a Pauli Z and ZZ (`ZR, ZZR`). This is just a short-hand version.
- Rotate phase by arbitrary angle (`Ttheta`).
- Rotate global phase by an arbitrary angle (`Gtheta`).
- Flip the current qubit basis from Z to Y or back (`Ybasis, YbasisAdj`).
- Rotate around Z, Y or X in the natural units (`Rz, Ry, Rx`).
- Controlled versions of above (`CRz, Cry, CRx, CTtheta, CGtheta`).
- Use `CNOT`s to entangle/unentangle across any number of qubits (`Entangle, UnEntangle`). This is for implementing Jordan-Wigner strings.

There are also a few specialized gates for Joint Measurement operations (used in braiding circuits for Majorana Fermions). See the sample in `Joint.fsx` for details:

- Joint measure in the Z basis (`JMz`). This gate take a symbol name to store the result in (since it isn't local to any one qubit) and a list of qubits to perform the joint measurement on.
- Joint measure in the X basis (`JMx`).
- Joint measure in multiple basis (`JM`). This op takes a string of basis values "`xyz`" that match up 1-to-1 with the qubit list you provide.

- Parity Control (`PC`). This gate takes the results of previous joint measurements and decides whether to apply a gate (like `BC`). It takes a label (to put on drawings to show the formula used), a function to compute the desired Boolean operation in addition to the gate to control and the qubit list. If we wanted to check if two previous measurements were not equal, and if so, apply an `X` gate, we could do (from `Joint.fsx`):

```
PC "p1<>p3" (fun qs -> k.Symbol "p1" <> k.Symbol "p3") X [t]
```

Gate and Qubit Operators

Shortcuts that help

Some manipulations of **Gate**s and **Qubit**s are done on a regular basis and therefore LIQ$Ui|\rangle$ provides some short-hand F# operators to make these various function easier:

>< Apply operator to qubits

The map operator (><) takes a `Gate` function and a list of `Qubits` and applies the `Gate` to each `Qubit` in turn. For example: `H >< qs` will perform a Hadamard operation on each of the qubits in qs

>!< Apply operator to qubits with argument

The argument map operator (`>!<`) takes a `Gate` function and a tuple of arguments and `Qubits` and applies the `Gate` (with the provided argument) to each `Qubit` in turn. For example, the following two syntaxes are available:

```
    Label >!< (["q0";"q1";"q2"],qs)
    Label >!< ("|0>",qs)
```

Example 36: Map with arguments operator

The first will place a label on each of three qubits. The second will put the same label on all the qubits in the list.

!! Build a qubit list

The build operator (`!!`) will take various arguments and turn them into a legal `Qubit` list. Here are some examples:

```
    !!k                   // When k is a Ket
    !!q                   // When q is a Qubit
    !!(q,q)               // Two qubits
    !!(q,q,q)             // Three qubits
    !!qs                  // Qubit list
    !!(qs,qs)             // Two qubit lists
    !!(qs,qs,qs)          // Three qubit lists
    !!(qs list)           // List of qubit lists
```

```
!!(qs,q)              // Qubit list and a qubit
!!(q,qs)              // Qubit and a qubit list
!!(qs,i)              // Take qubit i from qubit list
!!(qs,i,j)            // Take qubits i,j from qubit list
!!(qs,i,j,k)          // Take qubits i,j,k from qubit list
!!(qs,is)             // Take qubits in is list from qubit list
!!(qs,i)              // Pull out qubit i and turn into a list
!!(qs,i,j)            // Make a list from qubits i and j
!!(qs,i,j,k)          // Make a list from qubits i,j and k
!!(qs,idxs)           // Make a list from qubits with idxs list
```

Example 37: Build a Qubit list operator

!< Get Gate The extract gate operator (`!<`) will call a Gate function, ask it for the underlying `Gate` and return the data structure. This is used when defining parent gates in custom gate functions and during gate mapping (e.g., implementing Quantum Error Correction Codes).

Chapter 5

Extending the Simulator

LIQ$Ui|\rangle$ has been architected to allow extensions in several different directions. At the bottom Gates may be defined (or re-defined), Circuits may be re-written, modules may be added including new simulation, rendering, optimization and export engines. We'll start with gate definitions and work our way up.

Custom Gates

What can I create?

Custom `Gates` are defined by the user in the identical way that built-in gates were defined by the system. Indeed, no gates are actually "built-in", they are just defined as a convenience to the user. We will walk through several of the built-in gates as an example of how to define your own gates.

A distinction should be made between the `Gate` data structure and gate functions (also called Operations). The latter is an F# function that will carry out the definition of the `Gate` that it contains. The reason for this separation is that it allows us to call Operations in multiple modes (Run, Circuit and Gate) that perform different desired behaviors. All the `Gates` we define will always be wrapped in Operations so that they may be used in any mode desired. The `Gate` structure itself looks like this:

```
type Gate(  ?Name:string,
            ?Qubits:int,
            ?Mat:CSMat,
            ?Draw:string,
            ?Help:string,
            ?Op:GateOp,
            ?Parent:Gate option)
```

Example 38: Gate constructor

Many fields are optional depending on the type of gate being implemented. Here is what each one means:

- `Name`: Name of the gate (used in output functions for printing and drawing). Typically, this is always specified
- `Qubits`: Arity of the gate. This must be specified if there is no associated unitary matrix (i.e., if it can't be determined from the other parameters).
- `Mat`: unitary matrix. Not needed if the type of `Op` doesn't require it.
- `Draw`: Drawing instructions for circuits (discussed in detail later)
- `Help`: Information for a user of this Gate to understand it
- `Parent`: What gate we're based on (if any)
- `Op`: `GateOp` for this gate (default s to `Normal)` and must be one of:
    - `Normal`: Standard unitary operator
    - `Measure`: Measurement on first qubit
    - `Reset(Bit)`: Re-create a `Qubit` from a `Bit`. If the provided bit = `Unknown` then use the current digital value of the qubit as the value to re-constitute.
    - `String`: Used by `Label` gates (just for drawing)
    - `Modify(n)`: Create a new gate with an additional n qubits at the front. Used for Adjoint and Control gates. Requires `Parent` field to be specified
    - `BCOp(n,op)`: Binary control on n qubits (at front of list). Hand `op` the qubits and let it classically decide if we should execute the `Parent` gate on the remaining qubits (if it returns true).
    - `WrapOp(op)`: Wrap other gates into one meta-gate
    - `WrapHam(pqrs,op)`: Represents a second quantized Hamiltonian term with PQRS spin orbitals.

We'll now walk through each of the `GateOp` types and show how they are defined.

Normal Basic kind of Gate

This is the gate type that defines normal unitary operations. We'll show the `CNOT` gate as a first example (because it requires more than one `Qubit`) and then give a few examples of parameterized gates. The Qcircuit like drawing instructions will be explained in the section on Rendering.

The full code for CNOT follows:

```
let CNOT (qs:Qubits) =
    let gate =
        Gate.Build("CNOT",fun () ->
            new Gate(
                Name    = "CNOT",
                Help    = "Controlled NOT",
                Mat     = (CSMat(4,[(0,0,1.,0.); (1,1,1.,0.);
                                 (2,3,1.,0.); (3,2,1.,0.)])),
                Draw    = "\\ctrl{#1}\\go[#1]\\targ"
        ))
    gate.Run qs
```

Example 39: CNOT Gate implementation

At the inside of the function is the actual `Gate` structure (`new Gate(…)`). You'll notice that it has the fields we already described. We need to explain the `Mat` entry further (`Draw` will await a separate section on rendering circuits). Matrices in LIQ$Ui|\rangle$ are defined to be Square, Complex and Sparse (hence `CSMat`). The constructor used is fairly straight forward. The first argument is the size of the matrix N ($N = 2^k$ where k is the number of qubits), defining an NxN matrix. The second argument is a list of non-zero entries in the form: (row, col, real, imaginary). In this case, we've defined CNOT as: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

Next out, is a call to `Gate.Build(…)`. This call is optional, but should be used whenever possible. A global cache of gates is maintained for efficient memory usage and quick initialization. By calling `Gate.Build(…)` with a unique key (usually the name of the gate with any unique parameters), the gate creation function is only called once and from then on is looked up in a dictionary. The function cannot be used if the gate created could be different on each instantiation (for instance, changing binary values). It is most useful for fixed unitary matrices.

The outer most section of the function calls the `gate` we just created with `gate.Run qs`. This allows the runtime to choose the mode to call the gate in (Run, Circuit or Gate) dynamically and completes the definition of the gate function (or Operation).

Measure Measurement of a qubit

The measure operation takes two parameters (`name,joint`). If `joint` is the empty string, then this is a normal (destructive measurement). Otherwise, `name` is where we will put the result in the state symbol table and `joint` is the string of basis measurements ("`xyz`") that we wish to jointly perform.

The `Draw` parameter typically uses a built-in shortcut to generate the picture of a meter. The actual code used for measure is:

```
let M (qs:Qubits) =
    let gate =
        Gate.Build("M",fun () ->
            new Gate(
                Name    = "Meas",
                Help    = "Collapse State",
                Mat     = CSMat(2),
                Draw    = "\\meter",
                Op      = Measure("","")
        ))
    gate.Run qs
```

Example 40: Measurement Gate definition

Reset Turn a bit into a qubit

All variations of this operation are covered by the `Reset` and the `Restore Gate` functions. However, it might be instructive to show the `Reset` definition since it's our first example of a parameterized gate function:

```
let Reset (b:Bit) (qs:Qubits) =
    let gate (b:Bit) =
        let key = "Reset" + b.v.ToString()
        Gate.Build(key,fun () ->
            new Gate(
                Name        = key,
                Help        = "Re-create qubits with " + key,
                Mat         = CSMat(2),
                Draw        = "\\gate{\\ket{"+b.v.ToString()+"}}",
                Op          = Reset b
        ))
    (gate b).Run qs
```

Example 41: Reset Gate definition

Note that we now have a `Bit` value (b) that has to be passed down through the definition (along with the input qubits (`qs`) of course). You can also see that the `key, Name, Help` and `Draw` fields are all parameterized by the bit value.

Restore is simpler, because it requires no parameters:

```
let Restore (qs:Qubits) =
    let gate =
        Gate.Build("Restore",fun () ->
            new Gate(
                Name        = "Restore",
                Help        = "Restore qubit to measured value",
                Mat         = CSMat(2),
                Draw        = "\\gate{\\ket{M}}",
                Op          = GateOp.Reset Unknown
        ))
    gate.Run qs
```

Example 42: Restore Gate definition

Modify change a parent gate

Modify is worth a couple of examples. In the first, we just want to change the definition of a parent gate. In this example, we'll take the adjoint of the gate we're given as a parent:

```
let Adj (f:Qubits->unit) (qs:Qubits) =
    let gate (f:Qubits->unit) (qs:Qubits) =
        let parent      = !< f qs
        let pMat        = parent.Mat
        if pMat.Length = 0 then
            failwithf "Adj can't control parent %s, no matrix "
                                parent.Name
        Gate.Build("Adj_" + parent.Name,fun () ->
            new Gate(
                Name    = sprintf "%s'" parent.Name,
                Help    = sprintf "Adjoint of %s" parent.Help,
                Draw    = DrwAST.Morph(parent.Draw,
                   fun (str:string) -> str + "^\\dagger"),
                Op      = Modify 0,
                Mat     = pMat.Adj(),
                Parent  = Some parent
            ))
    (gate f qs).Run qs
```

Example 43: Adjoint Gate defintion

There are a few new items to note. The parameter to this gate function is another gate function (`f`). From the gate function (Operation), we need to discover the actual `Gate`. This is done with the extract gate operator (`!<`). Everything else is pretty straight forward (including creating unique keys and names). The new matrix is created from the adjoint of the parent's matrix (`pMat.Adj()`). We also need to store away the parent we discovered in the Parent field (`Some parent`). For this gate, our operation is `Modify 0` which means that we are not adding any additional wires to the parent, just transforming it.

A Control `Gate` is an example where we want to operate on more qubits than the Parent needs. This is a more complicated gate, but still fairly straight forward. The main difference is the need to actually build an entirely new matrix based on the parent matrix. To make this easy, LIQ$Ui|$ adds a function to build it for you:

```
let Cgate (f:Qubits->unit) (qs:Qubits) =
    let gate (f:Qubits->unit) (qs:Qubits) =
        let parent      = !< f qs
        parent.AddControl()
    (gate f qs).Run qs
```

Example 44: Control Gate definition

`AddControl` creates a matrix twice the dimension of the parent matrix, fills in the diagonal of the new entries with 1s and copies the values from the old matrix into the lower right of the new one. This gives us a generalized gate that can take any unitary gate of any arity and create a controlled version (very powerful). There is also a version (`CgateNC`) that requests the gate not be cached (useful for Hamiltonians with many different angles).

BCOp Bit Controlled operation

The bit controlled gate is a special case because classical values need to be manipulated at simulation-time and then quantum operations need to be performed based on these bits. There are two main uses for this `GateOp`. The first is to perform classical functions within a circuit (possibly for debugging or saving of information)… but without the application of a quantum gate as a result. An example of this is the `Native` gate function:

```
let Native (f:Qubits->unit) (qs:Qubits) =
    let gate (f:Qubits->unit) =
        new Gate(
            Name        = "Native",
            Help        = "Run native code in the circuit",
            Mat         = CSMat(2),
            Draw        = "\\gate{Native}",
            Op          = BCOp(0,(fun (qs:Qubits) -> f qs;true))
        )
    (gate f).Run qs
```

Example 45: Native Gate definition

The looks pretty much like the previous gates with a couple of exceptions:

1. The operator says that it needs 0 binary `Bits` and just calls the provided function (`f`) with all of the qubits available (`qs`). This allows classical code to look at the `Ket`, do operations outside of the quantum simulation (like print out status) and then return.
2. The matrix is specified as `CSMat(2)` which is a dummy because it never gets used.
3. The `Parent` entry is never specified. This guarantees that we won't call another gate.

The other main use is as an actual binary controlled gate:

```
let BC (f:Qubits->unit) (qs:Qubits) =
    let gate (f:Qubits->unit) (qs:Qubits) =
        let parent          = !< f qs
        let op (qs:Qubits)  = qs.Head.Bit = One
        new Gate(
            Name    = "BitContol",
            Help    = "Bit control Qubit operator",
            Qubits  = 1 + parent.Arity,
            Draw    = "\\control\\cwx[#1]",
            Op      = BCOp(1,op),
            Parent  = Some parent
        )
    (gate f qs).Run qs
```

Example 46: Binary Control Gate definition

Here we actually eat up one qubit (`BCOp(1,op)`) and the `op` just looks to see if the first qubit is a `One` (`qs.Head.Bit=One`). Now the parent gate is called if this function returns true. You'll also note that neither of these gates use `Gate.Build(…)`. The reason is that they aren't cacheable because they can do unpredictable things at run time (classical operations that are not under the control of the simulation system).

WrapOp create a meta-gate

The last operation allows for wrapping of lower level gates into higher (more abstract) ones. There are two reasons to do this. The first is to create a "macro" that allows a single gate to be called (logically) from many places in your circuit. The second is to provide an abstraction for circuit drawing. One of the options on drawing is how far to "unwrap" the circuit. This allows for high level views of your circuit without seeing the possibly thousands of gates underneath.

Here's an example of the adjoint of a controlled rotation gate as a `WrapOp Gate`:

```
let CRAdj (k:int) (qs:Qubits) =
    let gate (k:int) (qs:Qubits)    =
        Gate.Build("CR'_" + k.ToString() ,fun () ->
            new Gate(
                Qubits  = qs.Length,
                Name    = "CR'",
                Help    = "Controled R' gate",
                Draw    = sprint "\\ctrl{#1}\\go[#1]\\gate{R%d^\\dagger}"
                              k,
                Op      = WrapOp (fun (qs:Qubits) ->
                              Cgate (Adj (R k)) qs)
        ))
    (gate k qs).Run qs
```

Example 47: Wrap Gate Implementation

The operations done (`WrapOp`) could be as complicated as desired with hundreds of gates involved. The only restriction is that the whole gate may only operate on the qubits that it's provided (of course).

Rendering

Drawing pretty circuits

Gates contain a field that directs how a circuit containing the gate will represent it (`Draw`). In this section we'll give a short overview of the rendering instructions that are available to the user when building custom gates.

The `Draw` field itself is just a list of drawing instructions to use patterned after Qcircuit (see http://www.cquic.org/Qcircuit/ for more information) which is a LaTeX wrapper for the XY-pic package. The current implementation is a complete re-write based on the TikZ drawing package but retains some of the "feel" of Qcircuit.

The file `LiquidTikZ.tex` in the `samples` directory contains the full quantum drawing package and should be placed at `c:\Liquid\LiquidTikZ.tex` so that it can be found by the `.tex` files generated from LIQ$Ui|\rangle$. If you'd like to see the capabilities of the drawing package, go into the `LiquidTikZ.tex` file and change the line:

```
\iffalse      % Sample Drawings
```

to `\iftrue` and compile. The result (provided in `samples` as `LiquidTikZ.pdf`) shows several circuit drawings. Here's one of the examples showing a QECC circuit with bending vertical wires to show where they go:

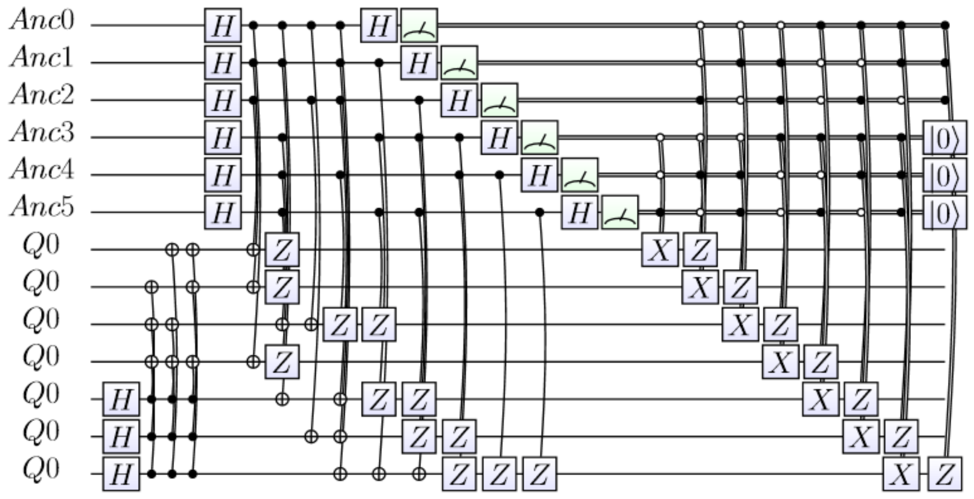


Figure 10: Sample drawing from LiquidTikX.tex

When called in a gate, the drawing cursor is placed on the wire representing the first qubit in the list at the current time step in the circuit. Each of the legal elements for this list will be described in detail. The output of a drawing request is either an SVG file (if the file extension is `.htm`) a LaTeX file (if the file extension is `.tex`) or both (when calling the `RenderHT` function) containing the drawing instructions. SVG files can be opened with a number of applications (including your Internet Browser) and TeX files may be used in LaTeX documents (provided you include the `LiquidTikZ.tex` file).

gate draw a gate on the circuit

The `\gate{string}` command takes a string and draws it on the circuit diagram with a box around it. The string must be a legal TeX text string and may contain TeX math symbols (e.g., `^\dagger` which becomes † as a superscript) and commands (e.g., `\ket{0}` which becomes $|0\rangle$

lstick draw text on the circuit

The `\lstick{string}` command draws the string on the circuit at the left of the current column. Variations include:

`rstick`, `ustick, dstick, cstick, cdstick` and `raw`. See the description of `Label` in Built-in Gates

multigate draw a multiline gate

The `\multigate{#1}{string}` command draws a gate that spans multiple wires. The numeric argument should be specified as the wire number where the bottom of the gate resides. For example if the gate is three wires long, then referencing `\multigate{#2}{U_a}` will create a gate 3 wires high (wires 0, 1 and 2) with U_a as its name.

control draw a circle

The `\control` command draws a closed circle on the circuit. For an open circle, use `\controlo.`

qwx draw a vertical line

The `\qwx[#1]` command draws a vertical line to the numbered wire (always specify the #). For a classical (double) wire, use `\cwx[#1]`.LIQ$Ui|\rangle$ also provides a `\dwx[#1]` to draw dotted lines. In general, there is no need to draw horizontal lines (`\qw`, `\cw` or `\dw`) since they will be inserted automatically.

ctrl draw a circle and a vertical line

The `\ctrl{#1}` command draws a closed circle on the circuit and then a vertical line to the numbered wire (always specify the #). For an open circle, use `\ctrlo.`

targ draw a mod 2 addition symbol on a wire

The `\targ` command puts a $\oplus$ on the current wire (target of a ctrl).

qswap draw cross on a wire

The `\qswap` command puts an `X` on the current wire (target of a quantum swap).

meter draw an meter

The `\meter` command puts a measurement meter on the current wire and converts the wire to digital (double line).

go positions the drawing cursor

The `\go[#1]` command will move the drawing cursor to a specific wire in the list of qubits the gate operates on (starting with 0). Subsequent commands will refer to that wire until another `\go` command is given. For example, to go to the second wire of a CNOT, you would specify: `\go[#1]`. Before the first drawing command, an implicit `\go[#0]` is performed.

Here's what the drawing instructions for CNOT look like:

```
Draw    = "\\ctrl{#1}\\go[#1]\\targ"
```

Example 48: CNOT Render instructions

Implicitly move to wire 0, draw a filled circle and a vertical line from wire 0 to wire 1. Then move to wire 1 and draw a $\oplus$.

Several rendering helper functions have also been defined to make common operations easy and efficient:

Morph re-write instructions

The `DrawAST.Morph(string,strFunc)` command takes two arguments: a string of drawing instructions (typically from a parent `Gate`) and a function that maps one string into another. In the `Adj Gate` shown above changes all of the string parameters to commands in the parent gate with the same name followed by a superscripted dagger (†).

`Draw` instructions `Wrap` gates are handled a little differently. If we're at the lowest level that we're going to draw, then the drawing instructions for this gate are executed. If we are going to go further (inside the wrapper), then we ignore these drawing instructions and instead proceed to the drawing instructions of the inner gates. This allows us to render circuits at various levels of abstraction.

Here's an example of a set of drawing instructions for the `Hpq` Hamiltonian gate:

```
Draw     = "\\ctrl{#1}\\go[#1]\\multgate{#2}{Hpq}",
```

Example 49: Rendering `Wrap` gates

Here we'll draw a closed circle on the wire 0 and then a vertical line to wire 1 (control from a phase estimation qubit). Then we move to wire 1 and draw a multiple qubit gate around qubits #1 to #2. Note that in an actual circuit, these qubits could be far apart and the box might be many qubits high.

Asking for a high level view, here's what the circuit for H_2 looks like (which utilizes some of the `Hp__` variants in `HamiltonianGates`:

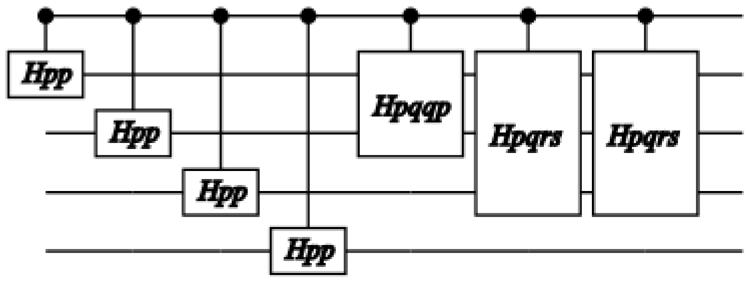


Figure 11: Rendering of a complex circuit (high level)

Here's the detailed rendering:

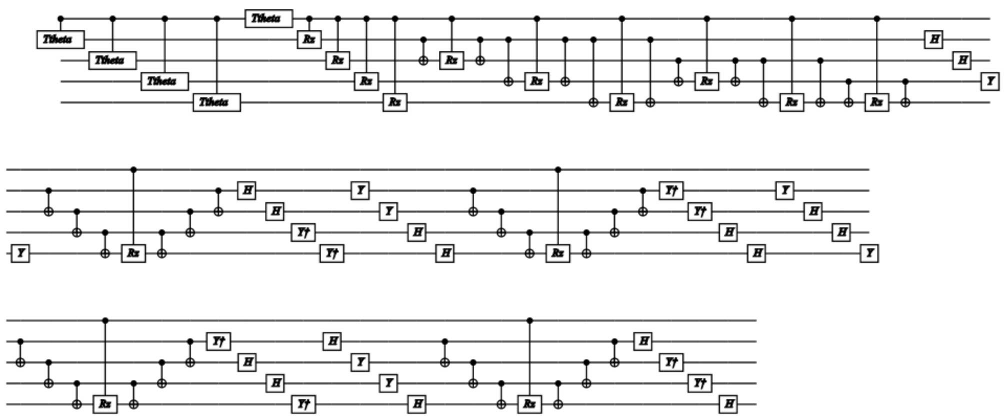


Figure 12: Rendering of a complex circuit (low level)

Chapter 6

Circuit Manipulation

Optimization

Circuit mode has been touched on a few times earlier in this document. We will now go over the various options in detail and then move into optimizations and alternative simulation engines that can be used with circuits.

Once an algorithm has been defined as a function, it may be converted to a **Circuit** by simply asking for a circuit compilation:

```
    let ket     = Ket(3)
    let circ    = Circuit.Compile teleport ket.Qubits
```

Example 50: Compiling a Circuit

A set of `Qubits` from a `Ket` vector must be provided so the `Circuit` has context of what qubits are needed and how they are used by the `Gate` functions (Operations). In this case we've taken the `teleport` function and converted it to a `Circuit` data structure. Elements of the `Circuit` data type include:

Seq Sequence of Circuits

`Seq(Circuit list)` represents an ordered list of `Circuits` to execute one after the other.

Par Parallel set of Circuits

`Par(Circuit list)` represents a parallel set of `Circuits` to execute at the same time. Optimizers generate this from `Seq` when there are no over-lapping qubits touched between multiple operations (e.g., the `Fold()` command).

Apply do a standard gate operation

`Apply(Gate,Wires)` instructs the system to apply a `Gate` to a set of wires (wires = `Qubit` ids). Wires are mapped back to actual `Qubits` before the `Gate` is called.

Ext extends the function of a Gate

`Ext(Gate,Wires,Circuit)` applies a new version of a `Gate` that was derived from another gate (the sub-`Circuit` in the argument list). A typical example would be taking the adjoint of another gate.

BitCon represents binary control gates

`BitCon(Gate,Wires,Func,Circuit)` runs the `Func` (that must return a Boolean value). If the value is true, then call the sub-`Circuit`.

Wrap meta gate that wraps a list of other gates

`Wrap(Gate,Wires,Circuit)` allows meta-gates to be represented. This is especially useful for drawing circuits with varying levels of resolution. The sub-`Circuit` is usually a `Seq` or `Par` circuit representing multiple `Gates`.

Empty dummy Circuit

Empty allows for a representation of a `Circuit` that has no elements. It is usually a place holder that is removed when an actual circuit starts being built. It may be thought of as a noop.

Once we have a `Circuit` several useful function are immediately available:

1. `Dump`: Dump allows a circuit to be printed to the console and/or the log. This is useful for both debugging and export to other simulation environments (see Example 13: Dump of Teleport Circuit).
2. `FindIDs(detail)`: Given a detail level (0=least), get a set of `Qubit` ids that are used by the circuit and total time steps necessary to execute the `Circuit`.
3. `Render(file:string,?typ:string,?detail:int,?split:float, ?scale:float)`: Draw a diagram for the `Circuit` into a file.
4. `RenderHT(file:string,?detail:int,?split:float,?scale:float)`: Call render for both svg and tikz output files (leave off extension).
5. `Fold(?aggressive)`: Convert `Seq` entries to `Par` entries where possible. This has the effect of sliding the `Circuit` elements to the left (useful to call before `Render()`). If you set aggressive=true then the entire circuit is flattened before parallelizing. This won't be as pretty for drawing, but will give much better estimates of the actually depth count of the circuit.
6. `GateCount(?doParallel)`: How many low level gates are there in the `Circuit`? If you want to overlay parallel circuits (`doParallel`), you should call `Fold(true)` first to get the most accurate count.
7. `Run(Qubits)`: Run the `Circuit` in the same way the original function (that it was compiled from) would have run. This becomes powerful after optimizations to the `Circuit` have been performed, or `Gates` have been substituted based on architectural or error considerations.
8. `Grow(…)`: The most sophisticated of the built-in `Circuit` functions. We will describe this function in detail next.

The family of `Circuit` functions that perform optimizations are all wrapped together under the `Grow(Ket,GrowPars)` function. This operation will take a

Circuit and re-write it for optimal execution. On a large circuit (hundreds of thousands of gates), reductions in gate count of 100:1 are easily achievable (with massive execution speed-ups). The `Ket` argument is only used for bookkeeping.

The `GrowPars` argument is usually created in one of two modes:

1. Gates: `GrowPars(?maxWires,?verbose,?allowDense)` will create an optimized `Circuit` from any other `Circuit`:
    a. `maxWires`: How many wires can be used in a single grown gate, the default of 11 is a fairly optimal value.
    b. `verbose`: Whether to report on what was done.
    c. `allowDense`: In most cases we don't want big dense matrices, but for some simulations we may want to force this to happen.
2. Single Unitary: `GrowPars(half,?eCnt,?oCnt,?skip,?diff,?verbose,?parity, ?redund,?colaesce)` will generate highly optimized circuits for Hamiltonian circuits (only):
    a. `half`: For the qubits representing electrons, are spin-up the first half of the qubits vs. interleaved (up,down).
    b. `eCnt`: How many electrons are there?
    c. `oCnt`: How many orbitals are there?
    d. `skip`: How many of the initial qubits are <u>not</u> electrons (ancilla, e.g., phase estimation qubits).
    e. `diff`: What is the required difference between up and down spin counts ([]=no restriction, otherwise a list of allowable differences).
    f. `verbose`: Report on what was done?
    g. `parity`: Force row and column parity to match? This forces conservation of angular momentum
    h. `redund`: Force removal of redundant gates (like an `X` following an `X` or two sequential `CNOT`s on the same wires).
    i. `coalesce`: Force coalescing of small angles across Trotterization steps. This is a tuple of (`size,keep`) where `size` is the limit of small angles and `keep` is whether to keep them.

Each of the restrictions specified in `GrowPars` affects how optimal the final output is. In the case of Hamiltonian `Circuits`, this can a massively complex circuit and turn it into one that is easily simulable.

QECC: Quantum Error Correction Codes

Adding faults and fault tolerance

Now that we have `Circuits` to manipulate, one the most useful things we can do is apply a transformation to the circuit that makes it fault-tolerant, add faults (via injected `Gates`) and then test the result (to see how fault tolerant the circuit really is). The problem we need to solve is allowing the user to create their own fault tolerant circuitry (extending LIQ$Ui|\rangle$) in such a way that all the other tools in the system are still available. That is what the `QECC` class is all about.

We refer the user to other sources on Quantum Error Correction Codes and their theory, but suffice it to say that there is a basic strategy we will follow for all implemented QECC inside of LIQ$Ui|\rangle$:

1. A `Circuit` to test will need to be re-written where each original `Qubit` (called a logical qubit) will need to be replaced with a set of `Qubits` (called the physical qubits).

2. These physical qubits will need to be "prepared" in a logical $|0\rangle$ state (given the code under investigation) via a provided `Gate`.

3. Each of the original `Gates` will need to be replaced with `Gates` that operate on a set of physical qubits which represent the logical qubits.

4. The QECC of a logical qubit will need to be measured, analyzed and fixed with a circuit known as the "syndrome" `Gate`.

5. After measurement, the QECC will need to be decoded from a set of physical qubits back to a logical qubit by doing classical error correction. This will be known as the `Decode` function.

6. Errors can be introduced in many ways. LIQ$Ui|\rangle$ provides a simple model of a de-polarizing channel that will insert X, Y or Z gates on any physical qubit with a user provided probability.

Besides the new class, we also provide two new gates that are useful in a number of codes:

Transverse implements a default QECC version of another gate

The Transverse gate will take an input gate and create a new version (via the Wrap operation) that performs the same operations on each of the physical qubits that make up a logical qubit. For example, `Transverse 7 X` will create a version of the `X` gate on 7 physical qubits.

T_BC Transverse version of Binary Control

The `T_BC` gate is a pre-built version of the `BC` gate that allows a logical binary control of other logical qubits (using transverse encoding). The reason that this gate is special (and can't be implemented with `Transverse` is that the binary control needs to be `Decoded` with a QECC specific function (turning the set of physical qubits back into a logical qubit that can be checked against `Zero` and `One`).

`QECC` itself is an abstract class that needs specific elements (added by the user) to turn it into a code that may be simulated. We'll walk through the definition of a specific CSS code (Steane [[7,1,3]]) which is provided with LIQ$Ui|\rangle$.

First, we need to derive our code implementation from `QECC`. The base class requires the number of "scratch" qubits we want (called Ancilla, which are typically used for syndrome measurement and control) as well as the number of physical qubits that make up a logical qubit. In addition, we need to provide the `Circuit` that we want to convert from standard to fault tolerant:

```
type Steane7(tgt:Circuit) =
    inherit QECC(6,7,tgt)
    let aCnt    = 6
    let cCnt    = 7
```

Example 51: Steane7 constructor definition

When we decode measured values, we'll need to know what the legal representations of a logical 0 and logical 1 are:

```
// Here are the logical 0 and 1 codes (for decoding)
let logical0 = [0x00;0x55;0x33;0x66;0x0F;0x5A;0x3C;0x69]
let logical1 = List.map (fun c -> c ^^^ 0x7F) logical0
```

Example 52: Logical values for Steane7

The first `Operation` we need is the one that prepares a set of physical qubits into a logical $|0\rangle$ qubit:

```
/// Prep gate for Steane7
let prep (qs:Qubits) =
    let nam = "S7_Prep"
    let nam2= "S7\nPrep"
    let gate (qs:Qubits)    =

        // Create logical |0> prep circuit
        let op (qs:Qubits)  =
            let xH i     = H [qs.[i]]
            let xC i j   = CNOT [qs.[i];qs.[j]]

            xH 6;   xC 6 3; xH 5;   xC 5 2; xH 4
            xC 4 1; xC 5 3; xC 4 2; xC 6 0; xC 6 1
            xC 5 0; xC 4 3

        Gate.Build(nam,fun () ->
            new Gate(
                Qubits   = qs.Length,
                Name     = nam,
                Help     = "Prepare logical 0 state",
                Draw     = sprintf Error! Hyperlink reference not valid.
                     (qs.Length-1) nam
                Op       = WrapOp op
        ))
    (gate qs).Run qs
```

Example 53: Steane7 preparation Gate

This looks just like the `Gates` we've seen before. It's made up of a bunch of Hadamard and CNOT gates. The `Circuit` for it looks like this:

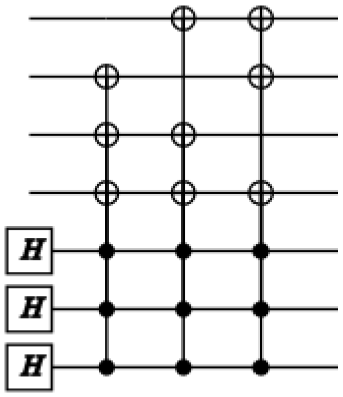


Figure 13: Steane7 Prep Circuit

What this `Circuit` does is create a superposition of all the legal codes for a logical $|0\rangle$ state.

Next we need to provide the syndrome measurement, decode and fix `Gate` this is a very complicated `Gate` and much of the detail will not be discussed here (see the references from the Introduction for more details). However, let's get a flavor for the "fix" section:

```
// Fix up a syndrome measurement
let fix (syn:int) (f:Qubits->unit) (qs:Qubits) =
    let nam              = "S7_Fix"
    let gate (syn:int) (f:Qubits->unit) (qs:Qubits) =
        let parent          = !< f qs
        let op (qs:Qubits)  =
            let b0  = qs.[0].Bit.v
            let b1  = qs.[1].Bit.v
            let b2  = qs.[2].Bit.v
            let bs  = (b0 <<< 2) + (b1 <<< 1) + b2

            // If we match the syndrome, then do the parent
            bs = syn

            else false
        new Gate(
            Name    = nam,
            Help    = "Fix syndrome measurements",
            Draw    = sprintf "\\cwx[#3]\\control%s\\go[#1]
                         \\control%s\\go[#2]\\control%s"
                        (if syn &&& 4 <> 0 then "" else "o")
                        (if syn &&& 2 <> 0 then "" else "o")
                        (if syn &&& 1 <> 0 then "" else "o")
                        ,

            Op      = BCOp(3,op),
            Parent  = Some parent
        )
    (gate syn f qs).Run qs
```

Example 54: Fix a detected error in Steane7

This `Gate` is a binary controlled operator (`BCOp`) that reads 3 Ancilla, decodes them, sees if they represent the specific syndrome we're looking for… and if so, applies the `Parent Gate` we were handed (`f`). It is called 7 times for possible `X` flips (Ancilla representing errors 1-7) and 7 more times for possible `Z` flips. The parent `Gate` that does this is:

```
    // Syndrome measurement
    let synd (qs:Qubits) =
        let nam = "S7_Syn"
        let nam2= "S7\nSyn"
        let gate (qs:Qubits)    =

            // Syndrome ops (assume first 6 qubits are ancilla)
            let op (qs:Qubits)  =
                let xH   i          = H [qs.[i]]
                let xXs i js        =
                    for j in js do
                        CNOT [qs.[i];qs.[j]]
                let xZs i js        =
                    for j in js do
                        Cgate Z [qs.[i];qs.[j]]
                let xM   i          = M [qs.[i]]
                let xFX syn i       = fix syn X !?(qs,[3;4;5;i])
                let xFZ syn i       = fix syn Z !?(qs,[0;1;2;i])
                let xR   i          = Reset Zero [qs.[i]]

                // Measure syndrome
                for i in 0..5 do xH i
                xXs 0 [9;10;11;12]
                xXs 1 [7;8;11;12]
                xXs 2 [6;8;10;12]
                xZs 3 [9;10;11;12]
                xZs 4 [7;8;11;12]
                xZs 5 [6;8;10;12]

                for i in 0..5 do xH i
                for i in 0..5 do xM i

                // Error correct
                for syn in 1..7 do
                    xFX syn (5+syn)
                    xFZ syn (5+syn)

                // Reset ancilla back to zero
                for a in 0..5 do xR a

            Gate.Build(nam,fun () ->
                new Gate(
                    Qubits  = qs.Length,
                    Name    = nam,
                    Help    = "Measure/Fix Syndrome",
                    Draw    = sprintf "\\multigate{#%d}{%s}"
                                 (qs.Length-1) nam,
                    Op      = WrapOp op
            ))
        (gate qs).Run qs
```

Example 55: Full syndrome Gate for Steane7

We will not go through the details here (left for an exercise to the reader ☺). However, the `Circuit` generated looks like this:

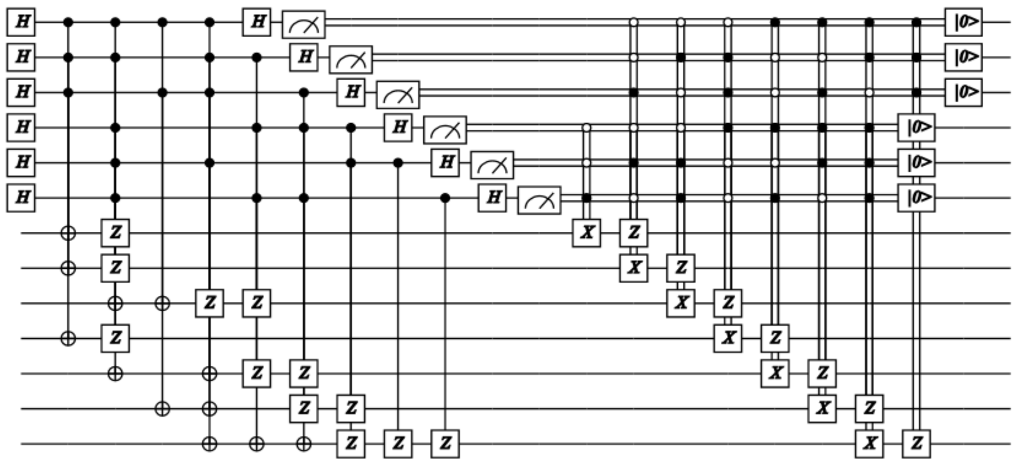


Figure 14: Steane7 Syndrome Circuit

The left half (through the Measurement boxes) is the syndrome measurement while the right half is the application of the "fix" gate 14 times (count the controlled `X` and `Z` gates). At the very end, the Ancilla are reset from `Bits` back to |0⟩ `Qubits` for use the next time.

In the actual `Steane7` class, we now can start overriding the abstract members:

```
override s.Prep qs = prep qs
override s.Syndrome qs = synd qs
override s.Replace(g:Gate) = base.Replace g
```

Example 56: Steane7 override definitions

Strictly speaking, we did not have to override `Replace` since the default `QECC` definition builds a set of Transverse `Gates` (which is what we need for Steane). The actual definitions of the gates are held in a dictionary:

```
// Default gate dictionary
let dic =
    let dic = Dictionary<Gate,Qubits->unit>()
    let q   = [ket.Qubits.[0]]
    dic.Add(!< CNOT ket.Qubits,Transverse cCnt CNOT)
    dic.Add(!< H q,Transverse cCnt H)
    dic.Add(!< S q,Transverse cCnt S)
    dic.Add(!< X q,Transverse cCnt X)
    dic.Add(!< Y q,Transverse cCnt Y)
    dic.Add(!< Z q,Transverse cCnt Z)
    dic.Add(!< I q,Transverse cCnt I)
    dic.Add(!< M q,Transverse cCnt M)
    dic
```

Example 57: Transverse gate dictionary

The default `Replace` function is implemented as:

```
default q.Replace(g:Gate) =
    if dic.ContainsKey g then Some dic.[g]
    else
        match g.Op with
        | BCOp(1,_)->   // Only single binary supported for now
            let gParent  =
                match g.Parent with
                | None      ->
                    failwith "QECC: BitCon needs a Parent gate"
                | Some g    -> g
            let T_Parent    =
                match q.Replace gParent with
                | None      ->
                     failwithf "QECC; BitCon missing gate: %s"
                            gParent.Name
                | Some f    -> f
            T_BC q.Decode T_Parent |> Some
        | _     -> None
```

Example 58: Default Gate replacement function

`QECC` also provides two useful functions that will help us write `Decode`:

1. `Log2Phy` which will take a logical wire number and return the physical qubits associated with it.

2. `GetMeasured` which will take a set of measured physical qubits and return a single hex value the represents the code measured $(0 - 2^{n-1})$

We can now write our `Decode` function:

```
// Compute bit best distance between codes
let bestCode measured =
    let best logical =
        let rec dist a b v =
            if a = 0 && b = 0 then v
            elif (a &&& 1) ^^^ (b &&& 1) <> 0 then
                 dist (a>>>1) (b>>>1) (v+1)
            else dist (a>>>1) (b>>>1) v
        List.mapi (fun i c -> i,(dist c measured 0)) logical
        |> List.minBy (fun (i,d) -> d)

    // Find min distance to a logical 0 code
    let best0,dist0 = best logical0
    let best1,dist1 = best logical1
    if dist0 <= dist1 then Zero,dist0 else One,dist1


override s.Decode (qs:Qubits) =
    let measured    = base.GetMeasured qs
    bestCode measured
```

Example 59: Decode implementation for Steane7

What this does is see if the number we measured is closer to a logical 0 or a logical 1 and then return a `Zero` or `One` accordingly (along with the Hamming distance to that code).

We now have a complete implementation of a `QECC`. The one function not mentioned that is provided in the `QECC` class is: `Inject(prob)`. This will inject random `X`, `Y` and `Z` gates into the `Circuit` with the given **prob**ability on each wire in the circuit. The current version only injects before a Wrapped gate (e.g., all the Transverse versions of a gate). This means that errors can be inserted before measurement… but they will be fixed classically via the `Decode` function (hence the need to return the Hamming distance).

The `Circuit` returned by `Steane7` can be simulated directly (like any other circuit)… however, we quickly reach the limits of LIQ$Ui|\rangle$. Just for teleport, which is only 3 qubits in size… the `Steane7` version is now 27 qubits! (3*7 + 6 Ancilla). A better way is clearly needed and that brings us to the next section.

Stabilizers

Simulating large numbers of qubits

QECC circuits are a good example of where our Functional Simulator is severely limited. As the number of qubits (n) grows, the memory required to store the state of the system grows as 2^n. We quickly run out of memory to hold the simulation. There is a way around this if we're willing to limit the types of operations that we'll allow in a circuit.

If we restrict ourselves to operations from the Clifford-Stabilizer framework (see references from the Introduction) then the state we need to maintain will grow linearly with n instead of exponentially. The `Gates` that are allowed for use in LIQ$Ui|\rangle$ Stabilizers are:

```
H,CNOT,S,X,Y,Z,I,M,Reset,Restore,CGate X,CGate Z,BC,Wrap
```

This is by no means a universal set of gates. However, it does include all the gates necessary to investigate QECC, so it's very useful for that purpose.

Another restriction is that `Qubit` states must be $|0\rangle$ or any state reachable by applying the `Gates` listed above (no teleport with "random" states). In fact, let's do a teleport of the value $|1\rangle$ using Stabilizers. First, we start with the definition of teleport we've used in all the other examples, but we'll add a state-flip at the beginning and a measurement at the end:

```
let tele1 (qs:Qubits) = X qs; teleport qs; M [qs.[2]]
```

Example 60: Teleport of $|1\rangle$ with final measurement

This gives us a $|1\rangle$ for the input message (by flipping the 0 `Qubit` with an `X`) and then measuring the result on `Qubit` 2 so we can see what happened. Let's just run the function normally:

```
let k       = Ket(3)
let qs      = k.Qubits
tele1 qs
show "tele1 returns: [%d%d] => %d"
  qs.[0].Bit.v qs.[1].Bit.v qs.[2].Bit.v
```

Example 61: Running the tele1 function

A typical output would be:

```
tele1 returns: [00] => 1
```

Example 62: tele1 result

Our message of 1 was teleported from the beginning to the end (as expected). Now, let's compile it into a **Circuit** and create a **Stabilizer** instance for it and run it:

```
let tgtC1   = Circuit.Compile tele1 qs
let stab    = Stabilizer(tgtC1,k)
stab.Run()
let _,b0    = stab.[0]
let _,b1    = stab.[1]
let _,b2    = stab.[2]
show "tele1 stabilizer: [%d%d] => %d" b0.v b1.v b2.v
```

Example 63: Stabilizer simulation of tele1

We get the same result as before, but now we've used a simulator that can handle thousands of qubits at one time. One difference is that we have to ask the

Stabilizer for the values of the Qubits (stab.[0]) because they are maintained in a different simulator with different statistics. For example, the first returned value (which we ignored using "_") is a Boolean that lets us know if the result was random or deterministic.

We can also query the **Stabilizer** simulator for its state at the end of the simulation:

```
    show "=== Final State: "
    stab.ShowState showInd 0
```

```
      0:0000.1/=== Final State:
      0:0000.1/
      0:0000.1/-X..
      0:0000.1/-.XX
      0:0000.1/-..X
      0:0000.1/----
      0:0000.1/-Z..
      0:0000.1/+.Z.
      0:0000.1/-.ZZ
```

Example 64: Final Stabilizer tableau

This shows the status of the various generators. We can also ask for a Gaussian reduction of the state:

```
    stab.Gaussian()
    show "=== After Gaussian: "
    stab.ShowState showInd 0
```

```
      0:0000.1/=== After Gaussian:
      0:0000.1/
      0:0000.1/-X..
      0:0000.1/+.X.
      0:0000.1/-..X
      0:0000.1/----
      0:0000.1/-Z..
      0:0000.1/+.Z.
      0:0000.1/-..Z
```

Example 65: Gaussian Stabilizer tableau

Of course, teleport isn't very interesting (in terms of size)… but if do a Steane7 code on the teleport circuit and then run it (see the QECC.fsx script for an example), we can look at a much more complex final tableau:

```
      0:0000.1/=== Final State:
      0:0000.1/
      0:0000.1/+.........ZZZZ...XXXX.......
      0:0000.1/+..Z...ZZ..ZZ.ZXZ.Y.Y..X.XX.
      0:0000.1/+..Z..........Y.Z.ZXY..X.XX.
      0:0000.1/+...X.......................
      0:0000.1/+....X......................
      0:0000.1/+.....X.....................
      0:0000.1/-......X....XX..............
      0:0000.1/-.......X..X.X..............
      0:0000.1/-........X.XX...............
      0:0000.1/+.........XXXX..............
```

```
0:0000.1/+.Z............ZZ..ZZ.XX..XX
0:0000.1/+..Z..........Z.Z.Z.ZX.X.X.X
0:0000.1/+Z...............ZZZZ...XXXX
0:0000.1/+..Z...Z.Z.Z.ZZ.Y.YXZ..X.XX.
0:0000.1/-..........X.X.Z............
0:0000.1/-..........XX...Z...........
0:0000.1/+..........XXX...Z..........
0:0000.1/-X.X..............X......X..
0:0000.1/-XX................X......X.
0:0000.1/-XXX................X......X
0:0000.1/-..Z..........Z.Z.Z.Z..X.X.X
0:0000.1/-.Z............ZZ..ZZ..X..XX
0:0000.1/-......................X....
0:0000.1/-Z...............ZZZZ....XXX
0:0000.1/-X.XX.X....X......Z......Z..
0:0000.1/-XX.XX......X......Z......Z.
0:0000.1/-XXXXXX......X......Z......Z
0:0000.1/----------------------------
0:0000.1/-................Z..........
0:0000.1/+..............Z............
0:0000.1/-.............Z.............
0:0000.1/+Z..Z............ZZZZ.......
0:0000.1/+.Z..Z.........ZZ..ZZ.......
0:0000.1/+..Z..Z.......Z.Z.Z.Z.......
0:0000.1/+......Z....................
0:0000.1/-.......Z...................
0:0000.1/+........Z..................
0:0000.1/-.........Z.................
0:0000.1/-.....................Z.....
0:0000.1/-....................Z......
0:0000.1/-.......................Z...
0:0000.1/+...............Z...........
0:0000.1/+.ZZ...ZZ..ZZ.ZZ..ZZ........
0:0000.1/+..Z...Z.Z.Z.ZZ.Z.Z.Z.......
0:0000.1/+Z........ZZZZ...ZZZZ.......
0:0000.1/+..............ZZZZ.........
0:0000.1/+.............Z.ZZ.Z........
0:0000.1/+.............ZZ.Z..Z.......
0:0000.1/+.............Z....ZZZ....ZZ
0:0000.1/+..............Z..Z.Z.Z..Z.Z
0:0000.1/+...............Z.ZZ...Z.ZZ.
0:0000.1/+................ZZZZ...ZZZZ
0:0000.1/+ZZ............ZZZZ.........
0:0000.1/+Z.Z..........Z.ZZ.Z........
0:0000.1/+ZZZ..........ZZ.Z..Z.......
```

Example 66: QECC teleport Stabilizer tableau

The code we need to change is fairly simple:

```
let s7          = Steane7(tgtC1)
let s7C         = s7.Circuit
let stab        = Stabilizer(s7c,s7.Ket)
stab.Run()
```

Example 67: Running teleport with QECC under a Stabilzer simulation

All we did was take the original `Circuit`, create an instance of `Steane7` and then use the circuit from the `Steane7` instance (and the `Ket` vector with all the new physical `Qubits`) to create the instance of the `Stabilizer` simulator.

To look at the values, we need to decode the final measurements from physical qubits back to logical qubits:

```
    let bit0,dist0  = s7.Log2Phys 0 |> s7.Decode
    let bit1,dist1  = s7.Log2Phys 1 |> s7.Decode
    let bit2,dist2  = s7.Log2Phys 2 |> s7.Decode
```

Example 68: Decoding QECC output

Once more, we have `Bits` returned as well as their Hamming distance to a legal code.

The Advanced Topics section (in Basic Operations) has a good example of what the circuit looks like with error injection. A typical run of the QECC test may be invoked with the following command. Typical output from running teleport of |0⟩ and |1⟩ looks like this:

```
       > Liquid __QECC()

0:0000.1/LOOP[Zer0]: InjectedXYZ(0,0,0) Fixes=0 (Zero, One,Zero) dist=(0,0,0)
0:0000.1/LOOP[Zer1]: InjectedXYZ(0,1,0) Fixes=2 ( One,Zero,Zero) dist=(0,0,0)
0:0000.1/LOOP[Zer2]: InjectedXYZ(0,1,1) Fixes=2 (Zero,Zero,Zero) dist=(0,0,0)
0:0000.1/LOOP[Zer3]: InjectedXYZ(0,1,1) Fixes=0 (Zero,Zero,Zero) dist=(0,0,1)
0:0000.1/LOOP[Zer4]: InjectedXYZ(1,1,0) Fixes=3 (Zero, One,Zero) dist=(0,0,0)
0:0000.1/LOOP[Zer5]: InjectedXYZ(1,0,0) Fixes=2 ( One,Zero,Zero) dist=(0,0,0)
0:0000.1/LOOP[Zer6]: InjectedXYZ(0,1,0) Fixes=2 (Zero, One,Zero) dist=(0,0,0)
0:0000.1/LOOP[Zer7]: InjectedXYZ(1,0,0) Fixes=1 ( One,Zero,Zero) dist=(0,0,0)
0:0000.1/LOOP[Zer8]: InjectedXYZ(1,1,0) Fixes=3 ( One,Zero,Zero) dist=(0,0,0)
0:0000.1/LOOP[Zer9]: InjectedXYZ(0,1,0) Fixes=1 ( One, One,Zero) dist=(0,0,0)

0:0000.1/LOOP[One0]: InjectedXYZ(0,0,0) Fixes=0 (Zero, One, One) dist=(0,0,0)
0:0000.1/LOOP[One1]: InjectedXYZ(1,0,0) Fixes=1 (Zero, One, One) dist=(0,0,0)
0:0000.1/LOOP[One2]: InjectedXYZ(0,0,1) Fixes=1 ( One, One, One) dist=(0,0,0)
0:0000.1/LOOP[One3]: InjectedXYZ(0,1,1) Fixes=3 (Zero,Zero, One) dist=(0,0,0)
0:0000.1/LOOP[One4]: InjectedXYZ(0,0,1) Fixes=1 ( One, One, One) dist=(0,0,0)
0:0000.1/LOOP[One5]: InjectedXYZ(0,0,1) Fixes=1 ( One,Zero, One) dist=(0,0,0)
0:0000.1/LOOP[One6]: InjectedXYZ(1,0,0) Fixes=1 (Zero,Zero, One) dist=(0,0,0)
0:0000.1/LOOP[One7]: InjectedXYZ(0,1,0) Fixes=1 (Zero, One, One) dist=(0,0,0)
0:0000.1/LOOP[One8]: InjectedXYZ(1,0,1) Fixes=0 (Zero,Zero, One) dist=(0,0,1)
0:0000.1/LOOP[One9]: InjectedXYZ(1,1,0) Fixes=3 (Zero, One, One) dist=(0,0,0)
```

Example 69: Command line QECC test with Stabilizers

This shows that `Qubit` 2 always got the right answer even though we injected errors in the form of random `X`, `Y` and `Z` gates. The `Fixes=` column shows the number of errors detected by the error syndrome and were fixed (X or Z flips). The `dist=` column shows the Hamming distance for each measured bit from a good code. The two that aren't 0 are because the errors were injected just before measurement and therefore were classical and could not be fixed by QECC (but were fixed correctly by the `Decode` function).

Chapter

7

Advanced Noise Models

Simulating the real world

Real world noise is much richer than the de-polarizing channel discussed in previous chapters. Here we'll apply a much more sophisticated model to represent both unitary and non-unitary noise sources.

Noise class The Noise class provides a harness for circuits that will run them, inject noise and gather statistics. Noise maybe Unitary (e.g., polarizing) or non-Unitary (e.g., a decoherence event).

NoiseEvents class The NoiseEvents class is used to keep track of summary noise statistics for a circuit being analyzed. It stores the number of times a gate was executed, how many times noise was applied to a gate and the total number of noise events (may be multiple per application).

NoiseModel class NoiseModel holds all the information necessary to model noise for a gate time. Gate names may have a trailing '*' to allow for wild-carding. In addition to execution time for the this particular gate there are separate statistics kept for gates that are part of the normal computation and gates that are part of the error correction circuitry (e.g., syndrome circuits). Each NoiseModel may have its own custom Noise Function which allows the user full flexibility in defining the specific characteristics of their noise model.

NoiseStat class Detailed statistics stored for each noise event that occurs.

Full Example

The best way to show how to implement a noise model is to work through a full example. We will use the one that appears in `samples` as `NoiseAmp.fsx` (this is also the `__NoiseAmp()` test built in to LIQ$Ui|\rangle$). Our goal will be to simulate a simple circuit over time and watch the effects of various types of noise:

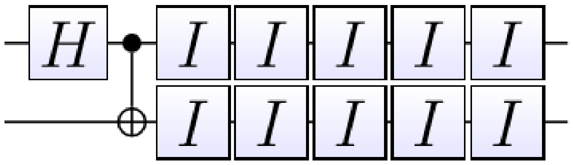


Figure 15: Circuit for Noise Analysis

We will prepare the circuit with a two qubit Ket vector and then apply a Hadamard and CNOT in the normal way so that we're ready to run the idle gates with a noise model. The actual noise model setup, looks like this:

```
// Create Idle circuit
let circ    = Circuit.Compile (fun (qs:Qubits) -> I >< qs) ket.Qubits

// Create noise model
let mkM (p:float) (g:string) (mx:int) =
                    {Noise.DefaultNoise p with gate=g;maxQs=mx}
let models      = [
    mkM 0.0             "H"         1
    mkM 0.0             "CNOT"      2
    mkM probPolar       "I"         1
]
let noise           = Noise(circ,ket,models)
```

Example 70: Create a noise model

This will give us a circuit with one Idle gate (`I`) in each qubit. We also define noise models for each gate type we might use. In this case, we're only going to run an `Idle` gate, but we can list the probability of noise per unit time for each gate type we're going to use. Gate names may have an asterisk (*) as a wild-card. In fact the entire name can be "*" which will apply that noise probability to all gates with the supplied qubit count.

For each of the gates, we chose to use `Noise.DefaultNoise` which creates a `NoiseModel` that implements depolarizing noise. The function can be replaced by the user with a different noise model if desired. The function itself is handed three arguments:

- Time: Amount of time spent in this gate (which may actually cover several sequential executions).
- Duration: Time to run one instance of the gate (Time/Duration is (approximately) the count of gate executions for this call
- Qubits: Qubits to apply noise to

If you define your own function (`f`), the easiest thing to do is to call "`NoiseModel.Default f`" which will initialize a `NoiseModel` that you can then override (as shown in Example 70: Create a noise model.

Now that we have a `Noise` class, we need to set desired options:

```
noise.LogGates     <- false     // Show each gate execute?
noise.TraceWrap    <- false
noise.TraceNoise   <- false
noise.DampProb(0)  <- probDamp
noise.DampProb(1)  <- probDamp
```

Example 71: Noise options

The first three are logging options (you can make things very verbose if desired). The last two are where we get to define which qubits we wish to have an Amplitude Damping model defined. This will give us both Unitary and non-Unitary effects. `DampProb()` for a given `Qubit` ID defines the probability of an amplitude damping decoherence event happening. If you are modeling a system that has a fastest gate time of T_g and has a decoherence time of T_1, then the probability that you want to use is $\exp(-T_g/T_1)$ provided you define the time for all other gates in terms of T_g. The next section gives full detail on the specific Amplitude Damping model used in LIQ$Ui|\rangle$.

In the sample file, we do one more preparatory step. We collapse the `Ket` state vector into a single dense vector and remember the vector (so that we can dump out running statistics as we go). We're now ready to run:

```
// Get a handle to the state vector for output
let v           = ket.Single()

dump 0 v
for iter in 1..500 do
    if iter = 1 then noise.Run ket else noise.Run()
    dump iter v

noise.Dump(showInd,0,true)
```

Example 72: Running the noise model

The actual run with noise is very simple. The first time we initialize by handing in a Ket vector (`noise.Run ket`) and for continued runs, we just drop the parameter (`noise.Run()`). This allows us to gather statistics across as many runs as we'd like (in this case after each pair of Idle gates).

The `dump` call (implemented in the sample file) will give us statistics at each of the 500 time steps we ran for:

```
0:0000.0/Iter,qs=00,qs=01,qs=10,qs=11
0:0000.0/   0,0.50000,0.00000,0.00000,0.50000
0:0000.0/   1,0.50100,0.00000,0.00000,0.49900
0:0000.0/   2,0.50200,0.00000,0.00000,0.49800
0:0000.0/   3,0.50300,0.00000,0.00000,0.49700
0:0000.0/   4,0.50400,0.00000,0.00000,0.49600
0:0000.0/   5,0.50500,0.00000,0.00000,0.49500
0:0000.0/   6,0.50601,0.00000,0.00000,0.49399
0:0000.0/   7,0.50701,0.00000,0.00000,0.49299
. . .
```

Example 73: Output from noise run

This is more interesting if plot the results (the output is already in CSV format:

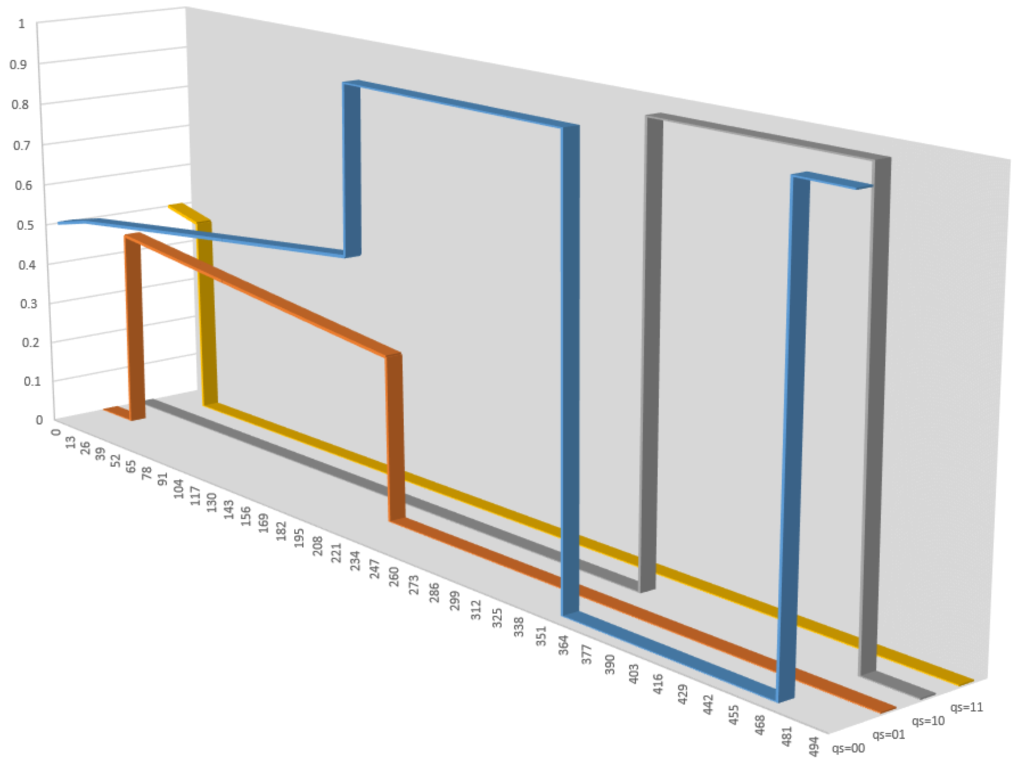


Figure 16: Output from Noise run

We can see the Unitary amplitude damping (sloped lines), the depolarization (where states are flipping between |0⟩ and |1⟩ and decoherence (where states are collapsing to |0⟩). We also get a summary table at the end of the run:

```
========== Noise models =========
   Gate Pattern   Dur G_count G_Apply G_event ECcount ECapply ECevent
```

```
        ------------   --- ------- ------- ------- ------- ------- -------

                   H   1.0       0       0       0       0       0       0
                CNOT   1.0       0       0       0       0       0       0
                   I   1.0    1000       3       3       0       0       0
             AmpDamp   1.0       2       2       2       0       0       0

     ========== Detailed statistics =========
           Time    Gate Pattern     Dur Wires        Type Detail
           ----    ------------     --- -----        ---- ------
          24.00         AmpDamp     1.0 0            norm <hard>
         127.00               I     1.0 0            norm depol_Z
         222.00         AmpDamp     1.0 1            norm <hard>
         355.00               I     1.0 0            norm depol_Y
         470.00               I     1.0 0            norm depol_X
```

Example 74: Noise final summary

In the first part we see that there were 1000 executions of the Idle gate (two each time) and 2 amplitude damping (decoherence) events. It also shows that there were 3 applications of de-polarizing noise. We can see the detail in the second table showing what time each event happened (and see the events in the graph above).

Amplitude Damping

The specific Amplitude Damping channel used ($\mathcal{E}_{AD}$) on a single qubit ρ is a CPTP map defined as[1]:

$$\mathcal{E}_{AD}(\rho) = K_1 \rho K_1^\dagger + K_2 \rho K_2^\dagger$$

Equation 5: Amplitude Damping Channel

where

$$K_1 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\rho_{AD}} \end{pmatrix}, \; K_2 = \begin{pmatrix} 0 & \sqrt{\rho_{AD}} \\ 0 & 0 \end{pmatrix}$$

Equation 6: Krauss operators

are Krauss operators (satisfying $K_1^\dagger K_1 + K_2^\dagger K_2 = \mathbb{I}$). The amplitude damping channel describes a process in which the state $|1\rangle$ can relax to the state $|0\rangle$ with some probability ρ_{AD}. If the input state is $|\psi_{in}\rangle = a|0\rangle + b|1\rangle$, where $|a|^2 + |b|^2 = 1$, then we will observe the output state to be

[1] Many thanks to Aleksander Kubica for this write-up of the LIQ$Ui|\rangle$ amplitude damping model.

$$|\psi_{out}\rangle = \begin{cases} \dfrac{a|0\rangle + b\sqrt{1-\rho_{AD}}|1\rangle}{\sqrt{1-|b^2|\rho_{AD}}}, with\ probability\ 1-|b|^2\rho_{AD} \\ |0\rangle, \qquad with\ probability\ |b|^2\rho_{AD} \end{cases}$$

Equation 7: Output state

Let us analyze what happens if we apply the amplitude damping channel $\mathcal{E}_{AD}\otimes\mathcal{E}_{AD}$ to an entangled two-qubit state, for example a Bell pair $|\psi_{in}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. One can check that the output state is

$$|\psi_{out}\rangle = \begin{cases} \dfrac{|00\rangle + (1-\rho_{AD})|11\rangle}{\sqrt{1+(1-\rho_{AD})^2}}, with\ probability\ 1-\rho_{AD}+\dfrac{1}{2}\rho_{AD}^2 \\ |10\rangle, with\ probability \dfrac{1}{2}\rho_{AD}(1-\rho_{AD}) \\ |01\rangle, with\ probability \dfrac{1}{2}\rho_{AD}(1-\rho_{AD}) \\ |00\rangle, with\ probability \dfrac{1}{2}\rho_{AD}^2 \end{cases}$$

Equation 8; Two Qubit Amplitude Damping

Note that we observed the output state $|\psi_{out}\rangle = |01\rangle$, then only the first qubit relaxed; similarly for the $|10\rangle$ state. If $|\psi_{out}\rangle = |00\rangle$, then both qubits relaxed. One can notice that the entanglement survives (i.e. the output state $|\psi_{out}\rangle$ is not a product state) only if no qubit undergoes a relaxation process (and $\rho_{AD} \neq 1$).

Usually, to simulate the effects of the amplitude damping noise on a circuit, we imagine performing ideal gates of the circuit followed by the amplitude damping channel. If the relaxation time of a qubit is T_1 and the execution time of a gate is t, then the probability ρ_{AD} for the qubit to relax while the gate is being applied is $\rho_{AD} = 1 - \exp(-\frac{t}{T_1})$.

Noise + QECC

One of the more useful applications of the advanced noise modeling capability is to apply it to quantum error correction. The system lets you analyze how errors in the data sections and syndrome sections of the circuit affect correctness.

The `Noise1.fsx` file in `samples` shows how to apply complex noise models to QECC (this can also be run directly from LIQ$Ui|\rangle$ as `__Noise1(depth,iters,prob)`). This circuit only has one logical qubit and a specified number of `Idle` gates. The parameters to the function are:

- **Depth**: How many `Idle` gates to use in the circuit.
- **Iters**: How many runs of the circuit to use for gathering statistics
- **Prob**: Probability of an error occurring

A typical call would be:

```
> Liquid.exe __Noise1(1,500,1.0e-2)
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.3/     , 1,1.00e-002, 107, 125,0.86
0:0000.5/     , 1,1.00e-002, 207, 250,0.83
0:0000.8/     , 1,1.00e-002, 307, 370,0.83
0:0001.0/     , 1,1.00e-002, 399, 488,0.82
0:0001.0/FINAL, 1,1.00e-002, 407, 500,0.81
0:0001.0/HIST, #,     prob,gate,  ec,good, all,frac
0:0001.0/HIST, 1,1.00e-002,   0,   0,  85,  85,1.00
0:0001.0/HIST, 1,1.00e-002,   0,   1, 122, 136,0.90
0:0001.0/HIST, 1,1.00e-002,   0,   2,  99, 131,0.76
0:0001.0/HIST, 1,1.00e-002,   0,   3,  41,  63,0.65
0:0001.0/HIST, 1,1.00e-002,   0,   4,  17,  24,0.71
0:0001.0/HIST, 1,1.00e-002,   0,   5,   8,  17,0.47
0:0001.0/HIST, 1,1.00e-002,   0,   6,   4,   7,0.57
0:0001.0/HIST, 1,1.00e-002,   1,   0,   6,   6,1.00
0:0001.0/HIST, 1,1.00e-002,   1,   1,   9,  11,0.82
0:0001.0/HIST, 1,1.00e-002,   1,   2,   7,  10,0.70
0:0001.0/HIST, 1,1.00e-002,   1,   3,   6,   7,0.86
0:0001.0/HIST, 1,1.00e-002,   1,   4,   2,   2,1.00
0:0001.0/HIST, 1,1.00e-002,   2,   6,   1,   1,1.00
0:0001.0/=============== Logging to: Liquid.log closed ================
```

Example 75: Advanced Noise plus QECC

The statistics at the end (`HIST`) are:

- **Gate**: Number of errors injected in the actual circuit (the 7 physical copies of the `Idle` gate that represents the one logical gate)
- **ec**: The number of errors injected into the error correction circuitry. Since these are the majority of the gates, this is where most of the errors occur
- **good**: Number of time the circuit generated the right answer
- **all**: Total number of runs for the circuit
- **frac**: Fraction of the runs that were correct

The details of this test can be found in the sample file. However, there are a few new techniques that we haven't seen before that are worth mentioning:

```
let prep'           = prep.Reverse()
noise.NoNoise      <- ["S7:Prep"]
noise.ECgates      <- ["S7:Prep";"S7:Syn"]
```

Example 76; New noise techniques

Once the circuit completes running, we execute a reversed version of the `prep` circuit which will convert the 7 physical qubits back to a logical qubit. If we then measure that qubit, we can see if it's in the correct state. The `NoNoise` property states which circuits will not have noise added. We've placed the name of the prep circuit here so that we can prepare everything perfectly (without nose injection). Likewise, the `ECgates` property is used to tell the `Noise` class which circuits are the error correction ones (this is how it knows the difference between data gates and syndrome gates).

Channels and POVMs

The most general and sophisticated noise model available in the system is implemented via a `Channel` gate type. Channels are implemented using a list of Kraus operators on one or more qubits. In addition, the system will report which operator was applied (its index), yielding a generalized measurement also known as a positive-operator value measurement (POVM). This section will describe a complete example (that is built into LIQ$Ui|\rangle$) showing how to define `Channel` gates, utilize them in `Circuits`, gather statistics and analyze the results.

LIQ$Ui|\rangle$ comes with two examples of `Channel` gates: AD, an Amplitude Damping channel:

```
let AD (prob:float) (qs:Qubits) =
    let gate (prob:float) =
        Gate.Build("AmpDamp_" + prob.ToString("E2"),fun () ->
            new Gate(
                Name    = sprintf "AD%.2g" prob,
                Help    = sprintf "Amplitude Damping channel: %.2g" prob,
                Kraus   = [
                     {tag=0;mat=CSMat(2,[(0,0,1.,0.);(1,1,sqrt(1.-prob),0.0)])}
                     {tag=1;mat=CSMat(2,[(0,1,sqrt(prob),0.0)])}
                    ],
                Draw    = "\\gate{" + ("AD" + (prob.ToString("E2"))) + "}",
                Op      = Channel("POVM")
               ))
    (gate prob).Run qs
```

Example 77: AD (Amplitude Damping) Gate

and `DP`, which is a Depolarizing channel:

```
let DP (prob:float) (qs:Qubits) =
    let gate (prob:float) =
        let e0  = sqrt(1.-3.*prob/4.)
        let e1  = sqrt(prob/4.)
        Gate.Build("DePol_" + prob.ToString("E2"),fun () ->
            new Gate(
                Name    = sprintf "DP%.2g" prob,
                Help    = sprintf "Depolarizing channel: %.2g" prob,
                Kraus   =
                    [
                     {tag=0;mat=CSMat(2,[(0,0,e0, 0.);(1,1, e0,0.)])}   // I
```

```
                      {tag=1;mat=CSMat(2,[(0,1,e1, 0.);(1,0, e1,0.)])}   // X
                      {tag=2;mat=CSMat(2,[(0,1,0.,-e1);(1,0, 0.,e1)])}   // Y
                      {tag=3;mat=CSMat(2,[(0,0,e1, 0.);(1,1,-e1,0.)])}   // Z
                     ],
                Draw    = "\\gate{" + ("DP" + (prob.ToString("E2"))) + "}",
                Op      = Channel("POVM")
               ))
    (gate prob).Run qs
```

Example 78: DP (Depolarizing Noise) Gate

Each of the channel examples takes a probability of noise occurring and then creates a set of Kraus matrices based on the supplied value. The two changes from making a standard gate are the addition of a `Kraus` entry that contains the list of operators and a new gate type: `Channel(symbol)`.

Each Kraus operator consists of a record that contains a `tag` and a matrix (`mat`). The tag is a numeric Id that will be returned during any generalized measurement (POVM). The matrix may be any legal Kraus operator. The only requirement is that if you take the list of matrices and combine them ($\sum M^{\dagger}M$) the result must be the identity matrix.

A channel defined by Kraus operators may apply to multiple qubits, but it is required that all the matrices in the same channel have the same dimension (define different channels for operators that work on different numbers of qubits or have different noise models).

The `Channel(symbol)` entry tells the system to treat the gate as a channel and that whenever it is called in a circuit, store the `tag` of the result (which matrix was actually applied) in the state vector symbol table (more on this later). The symbol name that it is stored under is whatever `symbol` you provide. Typically, we will read back this information right after the application of the channel, so using one name (e.g., "`POVM`") for all your channels isn't a problem. Make them unique if you have different requirements.

We're now ready to build an example (complete source code is in `Samples\Kraus.fsx`). The first thing we need to do is to define our noise channel. In this case, we're going to combine amplitude damping with depolarization:

```
let gateAD  = !< (AD probAD) qs
let gateDP  = !< (DP probDP) qs
let chan    = gateAD.OptimizeKraus(gateDP,"K")
let nams    = [" I";" X";" Y";" Z";"DI";"DX";"DY";"DZ"]
let dic     = SortedDictionary<int*string,int>()
```

Example 79: Build a noise channel

We take both the `AD` and `DP` gate functions, hand them a noise probability and then ask the system to return the actual gates that were built. The next call (`OptimizeKraus`) should always be called when you create channels. It has several important functions and its output is the gate function you should actually apply in your circuit. The first thing it does is analyze your matrices to make sure that they are correct. Secondly, it re-orders your matrices to increase the probability that a matrix earlier in the list will get executed before a later one (maximize trace of $M^{\dagger}M$), making your circuit run faster. This is what will happen if you call `OptimizeKraus` without any parameters. It also lets you perform useful manipulations of `Channel` gates:

1. You can provide a second channel gate that will be appended to the calling one. This lets you build up channels from multiple sets of Kraus operators. This is how we're combining Amplitude Damping with Depolarizing noise.

2. You can give the new gate a specific name (instead of the default of K#### where #### is a random 4-digit hex value).

3. You can provide an Id multiplier which will multiply the Ids of the first gate before adding the Id of the second gate (when it combines matrices). The default is to use a multiplier of the count of the number of gates in the second Channel. For example, if you have Amplitude Damping (2 matrices) and Depolarizing Noise (4 matrices), by default you will get Ids 0-7 assigned to the output matrices (2x4 total matrices). If you gave a multiplier of 10, you would get 0-3 and 10-13. This would allow you to use each decimal digit to identify which parent matrix the Id came from.

4. You can create a different POVM symbol (`Channel(symbol)`) for the output gate.

The last two lines of Example 79 gives labels to each of the 8 generated matrices and creates a dictionary to store statistics in.

The next two functions in the file (`saveStat` and `makeNoisy`) are helpers. The first allows us to save POVM measurement statistics in our dictionary and the second creates a "noisy" version of standard gates. The heart of the routine is the `Gate.Build` call (Example 80). It creates a `Wrap` gate that performs 3 operations:

1. Execute the original gate (e.g., `H`, `CNOT`, `X` …)

2. On each qubit, run the Channel gate we created (`chan`).

3. On each qubit, use embedded F# code to ask for the POVM index and save it in a dictionary. This is accomplished via the `Native` gate which lets us put classical code in the midst of executing circuits.

```
f       qs                   // Run the input function
for i in 0..g.Arity-1 do
    let qs' = !!(qs,i)
    chan qs'                 // Run the noise channel
```

```
Native (fun (qs:Qubits) ->  // Get the POVM result
    let q       = qs.Head
    let povm    = q.Ket.Symbol "POVM"
    saveStat q.Id povm) qs')
```

Example 80: Application of the noise channel

We can now use our helper function to build noisy versions of the gates we wish to call in our circuit:

```
let Hn (qs:Qubits)      = makeNoisy H qs
let Xn (qs:Qubits)      = makeNoisy X qs
let Zn (qs:Qubits)      = makeNoisy Z qs
let CNOTn (qs:Qubits)   = makeNoisy CNOT qs
```

Example 81: Noisy standard gates

This allows us to create a "noisy" Teleport that looks (at the source level) just like the original:

```
// Teleport circuit with noise channel added
let teleport (qs:Qubits) =
    Hn qs.Tail                      // EPR 2nd two qubits
    CNOTn qs.Tail
    CNOTn qs                        // Entangle first two qubits
    Hn qs
    M !!(qs,1)
    BC Xn qs.Tail                   // Conditionally apply X
    M !!(qs,0)
    BC Zn !!(qs,0,2)                // Conditionally apply Z
```

Example 82: Noisy Teleport

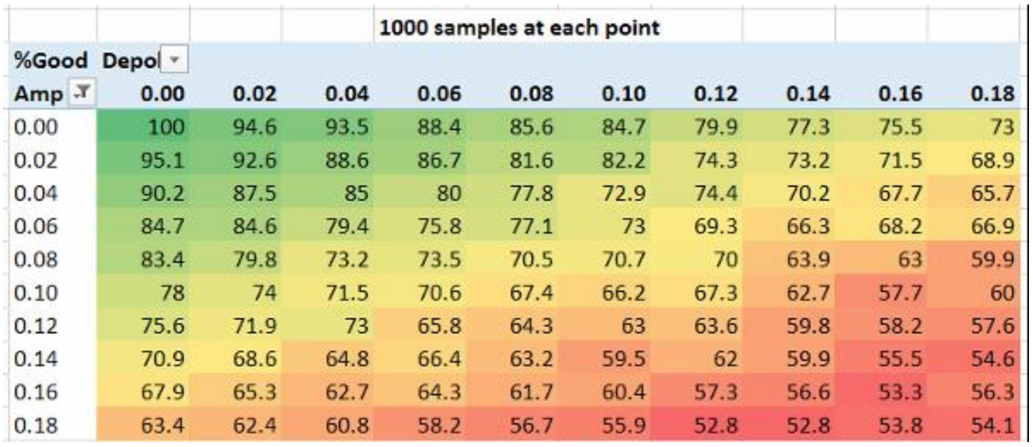
At this point, we have everything in place to execute our circuit. We'll do that a large number of times and then read the statistics generated in our dictionary. Here's a sample output from 1000 runs with error probabilities of (AmpDamp, Depol) = (.03,.03):

```
0:0000.0/CSV,qId,POVM,Count
0:0000.0/CSV, 0, I,  1916
0:0000.0/CSV, 0, X,    13
0:0000.0/CSV, 0, Y,    19
0:0000.0/CSV, 0, Z,    12
0:0000.0/CSV, 0,DI,    39
0:0000.0/CSV, 0,DZ,     1
0:0000.0/CSV, 1, I,  2914
0:0000.0/CSV, 1, X,    24
0:0000.0/CSV, 1, Y,    19
0:0000.0/CSV, 1, Z,    16
0:0000.0/CSV, 1,DI,    27
0:0000.0/CSV, 2, I,  1914
0:0000.0/CSV, 2, X,    13
0:0000.0/CSV, 2, Y,    14
0:0000.0/CSV, 2, Z,     8
0:0000.0/CSV, 2,DI,    43
0:0000.0/CSV, 2,DX,     1
```

```
0:0000.0/CSV, 2,DZ,     1
0:0000.0/CSV,rslt,0.0300,0.0300,   884,  1000,88.4
```

Example 83: Output from 1000 runs

After the "`CSV,`" you can see the qubit Id (0,1 or 2) and then the type of measurement (I,X,Y,Z for depolarizing noise and DI,DX,DY,DZ for Amplitude Damping + Depolarizing noise). At the end of the line is the count of occurrences of that type of noise. The last line shows the input parameters, the count of good instances (where we successfully teleported a $|1\rangle$). In the samples directory, there is also a small script (`Kraus.cmd`) that shows how to iterate over many sets of parameters to obtain a heat map. If you plot the generated "`CSV,rslt`" lines, you get:



1000 samples at each point

%Good	Depol									
Amp	0.00	0.02	0.04	0.06	0.08	0.10	0.12	0.14	0.16	0.18
0.00	100	94.6	93.5	88.4	85.6	84.7	79.9	77.3	75.5	73
0.02	95.1	92.6	88.6	86.7	81.6	82.2	74.3	73.2	71.5	68.9
0.04	90.2	87.5	85	80	77.8	72.9	74.4	70.2	67.7	65.7
0.06	84.7	84.6	79.4	75.8	77.1	73	69.3	66.3	68.2	66.9
0.08	83.4	79.8	73.2	73.5	70.5	70.7	70	63.9	63	59.9
0.10	78	74	71.5	70.6	67.4	66.2	67.3	62.7	57.7	60
0.12	75.6	71.9	73	65.8	64.3	63	63.6	59.8	58.2	57.6
0.14	70.9	68.6	64.8	66.4	63.2	59.5	62	59.9	55.5	54.6
0.16	67.9	65.3	62.7	64.3	61.7	60.4	57.3	56.6	53.3	56.3
0.18	63.4	62.4	60.8	58.2	56.7	55.9	52.8	52.8	53.8	54.1

Figure 17: Noisy Teleport sensitivity to errors

Hamiltonian Mode

Simulating the physics

The third simulator in LIQ$Ui|\rangle$ (after Universal and Stabilizer) is the Hamiltonian simulator. This environment allows you to define circuits that represents various types of Hamiltonians and to efficiently simulate them.

We've previously mentioned the gates that support Hamiltonians in the section on Built-In gates. All of these gates appear in module `HamiltonianGates`.

Hamiltonian class

The Hamiltonian class is at the base of the simulator, but is never used directly. Instead, one of the two derived classes that follow are instantiated by the user.

Spin-Glass simulation

Spin class The `Spin` class is used to define Spin-Glass problems to the system. The Hamiltonian being simulated is:

$$H = \Gamma(\mathrm{t})\sum_i \Delta_i \sigma_i^x + \Lambda(\mathrm{t})\left(\sum_i h_i\sigma_i^z + \sum_{i<j} J_{ij}\sigma_i^z\sigma_j^z\right)$$

Equation 9: Adiabatic Hamiltonian

We are starting in a ground state that we know in the σ_x direction ($\Gamma = 1, \Lambda = 0$) and ending in a target state in the σ_z direction (when $\Gamma = 0, \Lambda = 1$) which we'd like to discover. This is referred to as an adiabatic evolution since it is expected that if we move slowly enough (changing Γ, Λ) we can stay in the ground state of the entire system and smoothly move to the solution. The changing of strength over time is called the annealing schedule and typically looks like this:

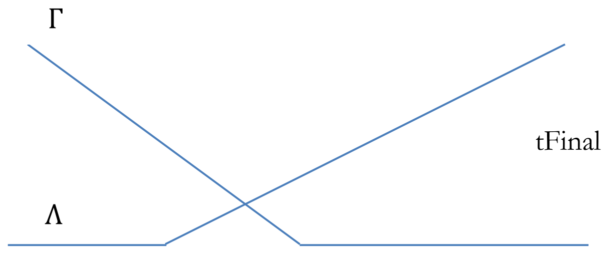


Figure 18: Typical Annealing Schedule

There are two main ways to instantiate the class. The first is the "bottom level" version that lets you specify everything:

```
type Spin(
    spinTerms : SpinTerm list,
    numSpins  : int,
    runMode   : RunMode)
```

Example 84: Spin constructor (1)

The constructor arguments are:

1. `spinTerms` which are a list of elements that contain:
    a. `schedule`: 0 based index of an annealing schedule that will be used
    b. `op`: Operatrion (`Gate`) to apply. Typically `ZR` or `ZZR`
    c. `ampl`:Amplitude (strength) of this term
2. `numSpins`: How many spins are there (qubits)
3. `runMode`: Trotterization to use:
    a. `Trotter1`: First order Trotter term
    b. `Trotter1x`: Split the transvers (`X`) field terms to the start and end of the `Circuit`
    c. `Trotter2`: Second order Trotterization

The second form of the constructor takes care of many of the details for you:

```
type Spin(
    hs      : Dictionary<int,float>,
    Js      : Dictionary<int*int,float>)
```

Example 85: Spin Constructor (2)

The constructor arguments are:

- `hs` – Create a dictionary for each `Qubit` that you want to provide a strength to in the range (typically) of -1.0 to +1.0. These are the `ZR` terms.
- `Js` – Coupling strength between two `Qubits`. This is a dictionary of `Qubit` Id pairs and strength. Only Ids where the first is less than the second is searched for (i<j) and typical values are -1.0 which is ferromagnetic coupling and +1.0 which is anti-ferromagnetic coupling (0.0 = no coupling). These are the `ZZR` terms.

Note that there's no `XR` term, since it's implied automatically and is on annealing schedule 0. The `ZR` and `ZZR` are terms on automatically placed on schedule 1.

Two built-in static members are available to aid in setting up spin-glass systems. Let's go through the example provided in the samples directory (`Ferro.fsx`) which simulates a ferromagnetic chain. The file shows both static members, but let's just go over the simpler one here:

```
let tests    = 50                 // Tests to run
let qCnt     = 12                 // Qubit count
let h0       = 1.0                // h0: Left most qubit Z strength
let hn       = -1.0               // hn: Right most qubit z strength
let coupling= 1.0                 // 1=ferro -1=anti-ferro 0=none
let sched    = [(100,0.0,1.0)]    // Annealing schedule
let runonce = true                // Runonce: Virtual measurements
let decohere= []                  // No decoherence model
Spin.Ferro(tests,qCnt,h0,hn,coupling,sched,runonce,decohere)
```

Example 86: Ferromagnetic script

The arguments are:

- `tests`: How many instances to run.
- `qCnt`: Total number of qubits to use as spin terms
- `h0`: Strength of the h term on `Qubit` 0. +1 = Force spin up
- `hn`: Strength of the h term on the last `Qubit`. -1 = Force spin down
- `coupling`: Strength of the between qubit terms (build a ferromagnetic chain by specifying +1.0)
- `sched`: At time 0, schedule 0 is always 1.0 (the σ_x term) and all the other schedules are at 0.0 . For this reason, we only need to specify the ending point for the schedules. Here we've specified a final time of 100 where the σ_x term (schedule 0) becomes 0.0 and the σ_z terms (schedule 1) become 1.0.
- `runonce`: This is a simulation optimization that lets us run a test once and then look directly into the state vector (since we're a simulator) and obtain all the probabilities instead of running 100s or 1000s of times and measuring to get the same result (which we'd have to actually do on a quantum computer).
- `decohere`: This is an advanced option that allows dechoherence models to be plugged in. For this test, we're using perfect qubits.

What we've done is created a twisted chain (one end up; one end down) so when we simulate, we get both details for 1 run and a histogram across all runs:

```
0:0000.0/  1%: ............ [<H>=-6.000 Stdev 0.000] [S_mid=0.000]
0:0000.0/ 10%: -..........+ [<H>=-5.525 Stdev 0.043] [S_mid=0.009]
0:0000.0/ 18%: 0..........1 [<H>=-5.242 Stdev 0.049] [S_mid=0.036]
0:0000.0/ 20%: 0-........+1 [<H>=-5.197 Stdev 0.049] [S_mid=0.048]
0:0000.0/ 25%: 0--......++1 [<H>=-5.142 Stdev 0.058] [S_mid=0.095]
0:0000.0/ 26%: 00-......+11 [<H>=-5.142 Stdev 0.061] [S_mid=0.108]
0:0000.0/ 28%: 00--....++11 [<H>=-5.154 Stdev 0.065] [S_mid=0.139]
0:0000.0/ 30%: 000-....+111 [<H>=-5.185 Stdev 0.069] [S_mid=0.180]
0:0000.0/ 32%: 000--..++111 [<H>=-5.234 Stdev 0.070] [S_mid=0.228]
0:0000.0/ 34%: 0000-..+1111 [<H>=-5.303 Stdev 0.067] [S_mid=0.267]
0:0000.0/ 81%: 00000..11111 [<H>=-9.031 Stdev 0.060] [S_mid=0.631]
0:0000.1/!Histogram:
0:0000.1/!Ferro  6.0% 000000000001 (E=-11.0000) [ones= 1]
0:0000.1/!Ferro 10.0% 000000000011 (E=-11.0000) [ones= 2]
0:0000.1/!Ferro  8.0% 000000000111 (E=-11.0000) [ones= 3]
0:0000.1/!Ferro 12.0% 000000001111 (E=-11.0000) [ones= 4]
0:0000.1/!Ferro  4.0% 000000011111 (E=-11.0000) [ones= 5]
0:0000.1/!Ferro 12.0% 000000111111 (E=-11.0000) [ones= 6]
0:0000.1/!Ferro 30.0% 000001111111 (E=-11.0000) [ones= 7]
0:0000.1/!Ferro  6.0% 000011111111 (E=-11.0000) [ones= 8]
0:0000.1/!Ferro  2.0% 000111111111 (E=-11.0000) [ones= 9]
0:0000.1/!Ferro 10.0% 001111111111 (E=-11.0000) [ones=10]
```

Example 87: Output from Ferromagnetic run

The detailed output shows the probability of each qubit between 0 and 1 (- = tending to 0, +=tending to 1 and .= no tendency). The histogram shows each case seen, what percentage of the runs fell into that category and the final energy (showing we reached the ground state). The test also generated two diagrams. The first was `Ferro.htm` which shows all the pieces (as well as the fact that we used `RunMode Trotter1X` visible from the fact that the `RpX` gates are both at the beginning and end of the `Circuit`):

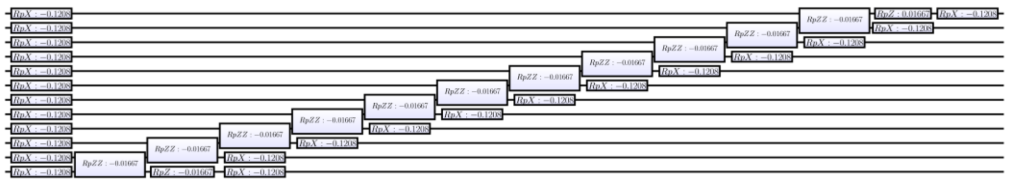


Figure 19: 12 Qubit ferromagnetic chain

The second diagram shows the Circuit that was actually run after "Gate Growing" was performed:

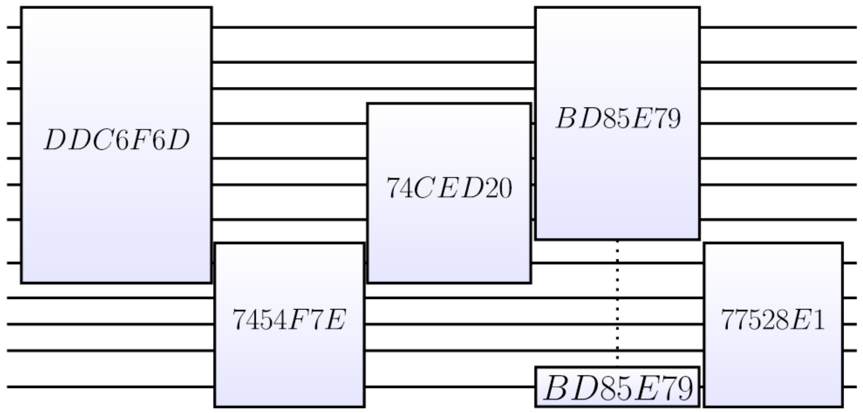


Figure 20: 12 Qubit ferromagnetic chain (grown)

Here you can see that we only had to do 5 matrix multiplies to perform the entire circuit (major speed-up). One of the reasons not to grow even further is that the circuit changes at every time step (due to the annealing schedule), so spending time optimizing beyond a certain point simply doesn't pay.

The `Spin.Test(...)` static routine allows arbitrary connections (not just chains) and much finer control (including Trotter number). An couple of examples are also provided in the `Ferro.fsx` script.

Fermionic simulation

Fermion class The `Fermion` class is used to define Fermionic problems to the system. The Hamiltonian being simulated is (in 2nd quantized form):

$$H = \sum_{p<q} h_{pq} a_p^\dagger a_q + \frac{1}{2} \sum_{p<q<r<s} h_{pqrs}\, a_p^\dagger a_q^\dagger a_r a_s$$

Equation 10: Fermionic Hamiltonian

We are simulating the Bohr model of a molecule where we will ignore the motion of the nuclei since they are massive and can be viewed as fixed in space in comparison to the electrons. The first summation describes single electrons as they move (annihilation: a_q and creation: $a_p^\dagger$). The second summation describes the interactions between pairs of electrons. These terms lead to the basic gates (`Hpp`, `Hpq`, `Hpqqp`, `Hpqqr` and `Hpqrs` used internally). For more details, see the reference to the various quantum chemistry papers in the introduction.

One item to note is that is that in the two body terms the outer (ps) and inner (qr) values must match in parity (spin-up or spin-down)

`Fermion` is normally accessed via it's static method: `Fermion.Run(dic,data)`. The first parameter is a dictionary of possible arguments (all have defaults) and the second is an array of data containing strengths for the h_{pq} and h_{pqrs} terms (obtained from any conventional molecular chemistry simulator). We'll describe each in detail using the `H2.fsx` sample script as an example. The first thing we need to do is to create a dictionary for all our parameters:

```
let dic = Dictionary<string,string>()   // Parameters to Fermion

dic.["Test"]   <- "26"    // Test to process in data[]
dic.["Bits"]   <- "18"    // Bit accuracy
dic.["Trotter"]<- "32"    // Trotter number
dic.["Thresh"] <- "-0.6"  // Max threshold to accept as an
                          // energy answer (with nuclear repulsion)
dic.["Emin"]   <- "-2.5"  // Min possible energy (w/o nuc repul)
dic.["Emax"]   <- " 1.5"  // Max possible energy (w/o nuc repul)
dic.["Ecnt"]   <- "2"     // Electron count
dic.["SOs"]    <- "4"     // Spin orbitals
dic.["Preps"]  <- "[1;2]" // Prepared start states (list of lists)
```

Example 88: H2 Fermoinic dictionary definition

Many of these are obvious (if you work in Quantum Chemistry ☺) but a few are non-standard:

- `Thresh` is just a cut-off to say that we don't want any answers with an energy higher than this (keep running until we get a good answer).
- `Emin, Emax` define the range of our phase estimation (0.0 to 1.0)
- `Ecnt` is our electron count (H_2 only has 2 electrons).
- `SOs` refer to the number of spin orbitals, this is twice the number of orbitals (up and down). In the case of H_2 we only have 4 SOs.
- `Preps` are the states to start from the full set for H_2 would be `[1;2];[1;3];[1;4];[3;4]` if we wanted to start with 2 electrons in each of the legal spin orbital configurations (SOs start at 1).

The `data` array consists of strings in a specific format that represent a test to run. The elements in each string are separated by whitespace and consist of:

- `tst=#` The test number of this line (does not have to be the same as the index in the array).
- `info=str` Any identifying information that you'd like the simulator to output for this test (in this case the separation of the nuclei).
- `nuc=#` Nuclear repulsion term. This is used to calculate the total energy values as opposed to the raw numbers without nuclear repulsion.
- `Ehf=#` Energy of the Hartree Fock calculation. This comes for free from your Quantum Chemistry package and can be used to sanity check the simulator results. Note that this term is with nuclear repulsion.
- `i,j=#` These are the single electron terms. The i and j refer to orbital IDs (starting at 0).
  `i,j,k,l=#` These are the double electron terms. The i, j, k and l refer to orbital IDs (starting at 0).

We give two examples in the `samples` directory of how to use classical quantum chemistry packages to generate the integrals:

PyQuante	See http://pyquante.sourceforge.net/ for details on obtaining, installing and running the package. A sample input file for generating `H2O.dat` if given in `H2O.py`. PyQuante is available on Windows.
Psi4	See http://www.psicode.org/ for details on obtaining, installing and running the package. A sample input file for generating `H2O.dat` is given in `H2O.inp`. In addition, the sub-directory `mointegrals` contains sample code for generating the molecular integrals that are placed in a `.dat` file. Psi4 does not currently run on Windows.

At the bottom of `h2.fsx` are two examples of running tests (by test number and by Trotter number). The chapter on Quantum Chemistry will give many more options on types of tests that could be run.

To run a single test, we could just type: "`fsi h2.fsx`" and get back a large set of information. The final, most detailed output is on the lines labeled `!CSV`:

```
0:0000.0/!CSV,1.40,32,1,
        -1.851551055908200,-1.137265055908200,
        "P.110101100111111111101",98.7
```

Example 89: Solution for H2 molecule

We had 18 terms that become 96 gates (see details in the log). We then reduced all the gates to a single unitary and then solved 18 bits of phase estimation yielding a total energy of -1.137265055908200 which was found by measuring a phase of: 110101100111111111101 using single qubit phase estimation.

If we run a full ensemble (with preps of electrons in different orbitals) and plot the results, we get the blue dots in the following figure (energy based on spacing of the nuclei):

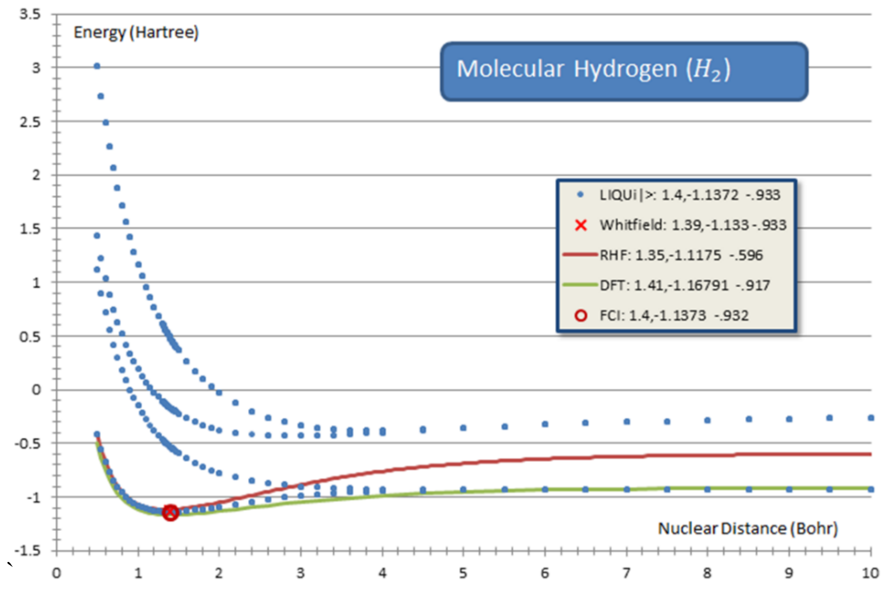


Figure 21: Molecular Hydrogen Energy Spectra

We can also do the equivalent for H_2O:

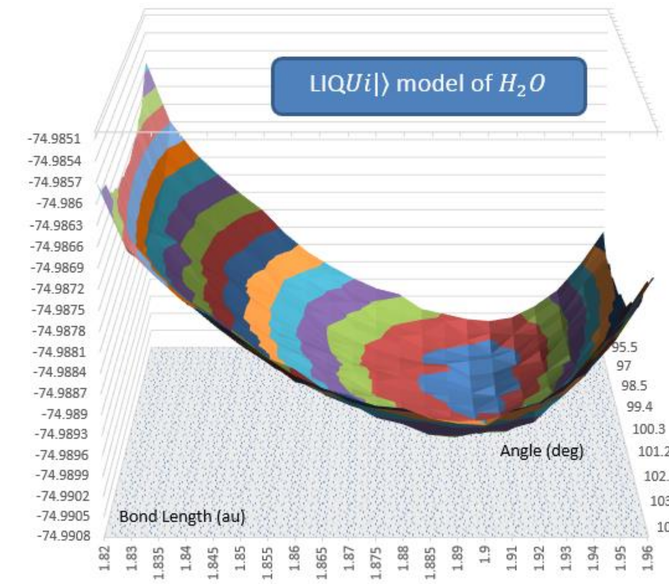


Figure 22: Water Energy vs. Bond Length and Angle

Details of more sophisticated use of LIQ$Ui|\rangle$ for Quantum Chemistry may be found in the next section.

Quantum Chemistry

In the previous section, we presented a short introduction to running quantum chemistry models. Now we would like to give in-depth details of how to utilize the built-in quantum chemistry system that is based on the Fermionic Hamiltonian simulator.

__Chem function The simplest way to invoke the system is to invoke LIQ$Ui|\rangle$ from the command line with: `__Chem("molecule")`. Using an illegal argument will provide help:

```
> liquid __Chem("")
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.0/Built-in tests:
0:0000.0/       tag                 data eCnt sorbs   eMin       eMax
0:0000.0/        H2       h2_sto3g_4.dat   2   4     -3.142      3.142
0:0000.0/      HeH+     HeH+_3-21g_8.dat   2   8     -7.142     -0.858
0:0000.0/        Be      Be_sto6g_10.dat   4  10    -17.142    -10.858
0:0000.0/       LiH     LiH_sto6g_10.dat   4  10    -12.142     -5.858
0:0000.0/        HF      hf_sto6g_12.dat  10  12   -108.142   -101.858
0:0000.0/      BeH2    BeH2_sto6g_14.dat   6  14    -22.142    -15.858
0:0000.0/       H2O     h2o_sto6g_14.dat  10  14    -88.142    -81.858
0:0000.0/       NH3     nh3_sto6g_16.dat  10  16    -71.142    -64.858
0:0000.0/       CH4     ch4_sto6g_18.dat  10  18    -57.142    -50.858
0:0000.0/       Li2     Li2_sto6g_20.dat   6  20    -20.142    -13.858
0:0000.0/       HCl     hcl_sto6g_20.dat  18  20   -468.142   -461.858
0:0000.0/        F2      f2_sto6g_20.dat  18  20   -231.142   -224.858
0:0000.0/       H2S     h2s_sto6g_22.dat  18  22   -413.142   -406.858
```

```
0:0000.0/
0:0000.0/Provide your own .dat file in the samples directory with a tag of:
0:0000.0/   "fileName:eCnt:sOrbs:eMin:eMax"
0:0000.0/        fileName - file name in samples directory
0:0000.0/        eCnt     - electron count
0:0000.0/        sOrbs    - Spin Orbitals (2x number of orbitals)
0:0000.0/        eMin     - energy min (in Hartree) for Phase Estimation
0:0000.0/        eMax     - energy max (in Hartree) for Phase Estimation
```

Example 90: Calling the __Chem function

This shows you all of the built-in molecules that are available. The `.dat` all live in the `samples` directory. Each molecule has an associated electron count (`eCnt`), spin orbital count (`sorbs`) and energy window (`eMin`, `eMax`).

If we wished to solve for the ground state of water, we could just type: `liquid __Chem(H2O)` (the double quotes are optional here). A large amount of information is generated (both at the console and even more in the log file). We will describe the output shortly.

You can also prepare your own molecules (details on the `.dat` file format were given in the Fermionic simulation section). In that case, you need to provide the ancillary information found at the end of the help output above. For example, to run H2O as if you created it yourself, you could type:

```
liquid __Chem("h2o_sto6g_14.dat:10:14:-88:-81")
```

Example 91: Running your own molecule

This would do the same as the built-in version, but gives the complete syntax for substituting your own `.dat` file.

__ChemFull function When you wish to control many more features of the quantum chemistry package use the `__ChemFull` command line function. The arguments to the function are:

- **mol**: Same argument as for the `__Chem` function described above.
- **test**: The test number in the `.dat` file that you wish to run.
- **opts**: String of all the single character options that you wish to set. There are many of these detailed in the following section.
- **trot**: The Trotter number that you wish to use. Typically this is around 32.
- **bits**: How many bits of accuracy you want in the phase estimation
- **order**: This is the Trotter order (currently only 1 or 2). For quantum chemistry, there's really no need to use 2nd order but it's there for comparison purposes.

A typical call would look like: `liquid __ChemFull(H2O,0,"",32,28,1)` which would perform the same function as `__Chem(H2O)`.

Quantum Chemistry Options

The flexibility of the quantum chemistry package lies in the options that are available. All of the options are single characters and will be described in the way that they are logically grouped.

TermOrder ?IJLMP When simulating a Hamiltonian the order of the terms may greatly influence both the accuracy of the result and the size of the executed circuit (whether terms may be nested or redundant gates removed). The term orders supported by LIQ$Ui|\rangle$ includes:

?	Random term ordering while maintain interleaved `PQ` and `PRRQ` terms
I	Interleaved `PQ` and `PRRQ` terms with lexicographically ordered terms. This is the default if not specified (optimal)
J	Jumbled (fully randomized)
L	Lexicographic order without interleaving
M	Sorted by Magnitude of the terms
P	Partial lexicographic ordering (for prettier display)

TermType ACOW There are multiple variants of the `PP, PQ, PQQP, PQQR` and `PQRS` circuits implemented. Details on the circuits may be found in the quantum chemistry papers referenced at the top of this manual. These options let you choose among them:

A	Ancilla based. Entangle the Jordan-Wigner strings onto one or more ancilla qubits (controlled by other options) instead of with each other directly. This allows for nesting of Hamiltonian terms. Typically not used for simulation.
C	CNOT based **O**ptimized circuits. These circuits replace the controlled routines (used for phase estimation) with standard rotations bracketed by `CNOT` gates. This is to simulate quantum hardware where we don't have controlled rotations available.
N	Nest terms if using the **A** option.
O	Optimized circuits. These are the term variants that have their Jordan-Wigner strings moved to the outside so that they can be collapsed out when terms are in lexicographic order.
W	Whitfield circuits. These are the original quantum chemistry circuits as described in the Whitfield, Biamonte and Aspuru-Guzik paper.

DropTerms QZpqusz

In some experiments, it's useful to be able to drop terms to see how this affects the result. There are several ways to do this:

Q	Drop the PQQP terms completely
Z	Drop a random 20% of all PQRS terms
p	Drop the PP terms completely
q	Drop the PQ terms completely
u	Drop the PRRQ terms completely
s	Drop the PQRS terms completely
z	Scale all the PQRS terms to 80% of their initial value

GrowGates 0DRgh A large part of the efficiency of the system to do quantum chemistry comes from how it reduces circuits to more efficient (large) unitary matrices. In the case of quantum chemistry we can be extremely efficient by only allowing physically realizable states. For example, even though the Hamiltonian for H_2O takes 15 qubits and would a 32768×32768 matrix to represent it, after removing non-physically realizable states, this can be reduced to just 441×441. All of this depends on the parameters we supply to `GrowGates`:

0	Turn off parity conservation (same number of up and down electrons enter and exit the operator (conservation of angular momentum).
D	Turn off enforcing a difference of 0 between the number of up and down electrons. This needs to be turned off if we have an odd number of electrons.
R	Turn off greedy decimation of the array while terms are being built because we have redundant gates being removed and may not be unitary until the end.
g	Turn off the entire growth into a single matrix and use the full circuit instead (very, Very, VERY slow). This sets `single` to `false`.
h	Turn off "half-up" ordering of the qubits. The default is to have all the spin-up qubits followed by all the spin-down qubits. This has been found to be a more optimal ordering. By turning this off, you return to a natural ordering where the qubits (in-order) represent: inner-orbital-up, inner-orbital-down, next-orbital-up, next-orbital-down… (fully interleaved).

`Parity` and `Diff` refer to optimizations we do if `Single` is true. When we grow a single unitary for the entire circuit we have the option to require parity between the rows and columns (we conserve angular momentum) and guarantee that the number of electrons we start with are the number we end with. `Diff` refers to the difference we expect between up and down spin counts (don't specify if you don't want this). Since we are looking for ground states with an even number of electrons we expect the number of UP spins – the number of DOWN spins to equal 0.

Accuracy FUclm~ There are times when you may want to increase (or decrease the accuracy of a computation (independent of the number of bits of the phase estimation). These options let you make several choices:

F Integrals coming from classical quantum chemistry packages are optimized for a Hartree Fock solution. We can use perturbation theory to make the off-diagonal elements more accurate for an FCI solution (which is what the quantum chemistry package is doing. This flag tells the system to do a "diagonal fix-up" of the integrals.

U Use the Hartree-Fock energy (in the .dat file) as well as the nuclear repulsion value to compute an energy range for the phase estimate (overriding the range provided by the user).

c The default accuracy for comparison of complex numbers is 1.0×10^{-18}. This is overkill for many situations (esp. long running computations). The first application of the "`c`" option reduces the accuracy to 1.0×10^{-11}. Each "`c`" after the first one will reduce the accuracy by a further $\sqrt{10}$.

l There are times when a few runs of the computation may not be enough (usually the system does from 10 to 20 depending on the repeatability of the results). By specifying this option, the system will run the experiment 200 times (no matter what the results). Since compiling the circuits is much more expensive than running, this is sometimes a good choice.

m Multiply the PQRS values by a random perturbation. The random value will be between 0.0 and 1.0. It will always use the same seed, so perturbations will be the same from run to run.

~ This is a more sophisticated random perturbation applied to all `Rz` gates in the circuit. This will actually happen as the Trotterization occurs and can be used to simulate jitter in actual (physical) rotation angles. The random values range within $\pm 1.0 \times 10^{-4}$.

Output GHSTarw Many types of output may be generated by the system. The section following this one will give detailed examples of what may be computed. Here are the options that control them:

G This option will dump the entire circuit for one pass of the molecule into the `Liquid.log` file.

H This makes it easy to get data for the created Hamiltonian into other applications (like Matlab). This file creates (`Terms_<datFileName>.txt`) which is suitable for loading directly into Matlab with arrays for `PP, PQ, PQQP, PQQR` and `PQRS` matrices.

S Normally, the log file only contains information on the first 100 terms in the molecule. This option places <u>all</u> terms in the log file which can be used for analysis outside of LIQ$Ui|\rangle$.

T It is sometimes useful to see what the Term Expectations are, give a state vector. This option shows the expectations of each of the term types from the prep vector as well as after solving the ground state. Here's an example from H_2O:

```
Initial expectation:
      pp: exp=0.730176 sum=0.590136
    pqqp: exp=0.227273 sum=0.269279
      pq: exp=0.014817 sum=0.046850
    pqqr: exp=0.015315 sum=0.054574
    pqrs: exp=0.012419 sum=0.039161

Final   expectation:
      pp: exp=0.723948 sum=0.590136
    pqqp: exp=0.224795 sum=0.269279
```

```
  pq: exp=0.019911 sum=0.046850
pqqr: exp=0.019558 sum=0.054574
pqrs: exp=0.011787 sum=0.039161
```

Example 92: H2O Term Expectations

a When dumping a binary `Ket` vector (see below) it might be interesting to see a human readable version. This option will create `Liquid.ket_txt` in the current directory (when dumping a binary version is requested).

r Read in a previously dumped `Ket` vector (see `w`) from the `Liquid.ket` file and use it for the initial (prep) state.

w Write the resulting `Ket` vector out to a `Liquid.ket` file.

Run BXf There are only a few options that globally affect how the system runs:

B Even if a molecule looks too big to run, try to run it anyway.

X No matter what, exit the system after statistics have been outputted but before actually running the molecule.

f Use temporary files to store the phase estimation matrices. This allows much larger molecules to run that could normally fit in memory..

Quantum Chemistry Output

When running the system, there is a large amount of output generated (esp. in the `Liquid.log` file). Some of which isn't immediately obvious. Let's go through running water and seeing what a typical log output looks like. We'll invoke the system with the command: `__Chem(H2O)`.

The first thing you'll see in the log is a list of all the settings being used to run your molecule:

```
0:0000.0/Test: __Chem("H2O",0,"",32,28,1): with 14 SOs
0:0000.0/        Parity = 1
0:0000.0/          Diff = 0
0:0000.0/        HalfUp = true
0:0000.0/        Single = true
0:0000.0/         Preps = [1;8;2;9;3;10;4;11;5;12]
0:0000.0/    AlterNoise = 0.0
0:0000.0/        Redund = false
0:0000.0/     TermOrder = Interleave
0:0000.0/      TermType = Optimized
0:0000.0/        PEtype = default
0:0000.0/            PP = true
0:0000.0/            PQ = true
0:0000.0/          PQQP = true
0:0000.0/          PQQR = true
0:0000.0/          PQRS = true
0:0000.0/      TolLevel = 1E-18
0:0000.0/          Test = 0
0:0000.0/       Trotter = 32
0:0000.0/          Bits = 28
```

```
0:0000.0/        Order = 1
0:0000.0/     Coalesce = 0
0:0000.0/      CplxTol = 1.00E-018
0:0000.0/         File = h2o_sto6g_14.dat
0:0000.0/          SOs = 14
0:0000.0/         Ecnt = 10
0:0000.0/         Emin = -88.141592650
0:0000.0/         Emax = -81.858407350
0:0000.0/       Thresh = -79.327433385
```

Example 93: H2O log, parameters

Most of these are settable by the options listed in the previous section as well as parameters to the `__ChemFull` function. One thing that's interesting to note is that since `HalfUp` is `true`, you can see that the `Preps` (where the electrons are initially) fills spin-orbtials 1-5 (spin-up) and spin-orbitals 9-12 (spin-down) instead of the interleaved version which would have filled spin-orbitals 1-10.

Next, the system will read in the `.dat` file and give summary statistics:

```
0:0000.0/  pp Spin orbital circuits:     14 (      14 terms,  145.219 summags)
0:0000.0/  pq Spin orbital circuits:     14 (      14 terms,   11.529 summags)
0:0000.0/pqqp Spin orbital circuits:     91 (     133 terms,   66.263 summags)
0:0000.0/pqqr Spin orbital circuits:    168 (     238 terms,   13.429 summags)
0:0000.0/pqrs Spin orbital circuits:    147 (     350 terms,   11.406 summags)
```

Example 94 H2O log, loaded terms

We can see the various term types that were loaded, the initial term count (in the parenthesis) and how many actual circuits were generated. The number of circuits will always be less than or equal to the number of terms, since multiple terms may be collapsed on top of each other (since they describe equivalent things). These circuits are then dumped to the log file (truncated at 100 unless you use the option to ask for all of them). Here is a sampling from the water log:

```
0:0000.0/  pp: -33.023097 H01,01 * A01,01 [00,00]
0:0000.0/  pp:  -7.719358 H02,02 * A02,02 [01,01]
0:0000.0/  pp:  -6.383992 H03,03 * A03,03 [02,02]
0:0000.0/  pp:  -7.012261 H04,04 * A04,04 [03,03]
0:0000.0/pqqp:   0.957020 H01,02,01,02 * A01,02,02,01 [00,01,00,01] (-pqpq,2)
0:0000.0/pqqp:   0.791225 H01,03,03,01 * A01,03,03,01 [00,02,02,00] (-pqpq,2)
0:0000.0/  pq:  -0.567081 H01,02 * A01,02 [00,01]
0:0000.0/pqqr:   0.021881 H01,03,03,02 * A01,03,03,02 [00,02,02,01] (-pqrq,2)
0:0000.0/pqqr:   0.032091 H01,04,02,04 * A01,04,04,02 [00,03,01,03] (-pqrq,2)
0:0000.0/pqrs:  -0.059498 H01,03,07,02 * A01,02,03,07 [00,02,06,01]
                                         -0.009143   -0.021201    0.030344
0:0000.0/pqrs:   0.076696 H01,02,04,06 * A01,02,04,06 [00,01,03,05]
                                          0.001723    0.020607   -0.022331
```

Example 95: H2O log, sample term dump

Looking at the first line for `pp`, we see the following information:

`-33.023097`:	This is the strength of the term after LIQ$Ui|\rangle$ has collapsed any equivalent terms together
`H01,01`:	This is the natural ordering that the term was read in (spin-orbital numbers starting at 1

`A01,01`:	This is the lexicographic ordering that the term became (spin-orbital numbers starting at 1. Notice the second `pqqr` term where they're different.
`[00,00]`:	Original orbital numbers (zero based) from the `.dat` file
`(-pqpq,2)`:	In addition, a line may be followed by information on other terms that were collapsed together. In this case (the first `pqqp` entry shown), there were two other terms with order `pqpq` that were subtracted off (since flipping the last `pq` causes a sign to flip).
`h1,h2,h3`:	In the case of `pqrs`, there are three extra value that represent the h1, h2 and h3 values described in the Whitfield, Biamonte and Aspuru-Guzik paper. These are used to build the actual circuits.

The next thing that in the output are summary gate statistics for the circuit built:

```
0:0000.0/Counts: Rot=1.42e+003 Seq=1.54e+004 Par=1.54e+004
                        Nest=1.54e+004 RedundBest=1.54e+004 (qubits: 15)
0:0000.0/Combining 15362 gates...
0:0000.0/           Hpp,   CTtheta:      14
0:0000.0/           Hpq,      CNOT:      88
0:0000.0/           Hpq,       CRz:      28
0:0000.0/           Hpq,        CZ:      24
0:0000.0/           Hpq,         H:      56
0:0000.0/           Hpq,    Ybasis:      28
0:0000.0/           Hpq,   Ybasis':      28
0:0000.0/         Hpqqp,      CNOT:     182
0:0000.0/         Hpqqp,       CRz:     105
0:0000.0/         Hpqqp,    Ttheta:       1
0:0000.0/         Hpqqr,      CNOT:    2576
0:0000.0/         Hpqqr,       CRz:     672
0:0000.0/         Hpqqr,         H:     672
0:0000.0/         Hpqqr,    Ybasis:     336
0:0000.0/         Hpqqr,   Ybasis':     336
0:0000.0/         Hpqrs,      CNOT:    4414
0:0000.0/         Hpqrs,       CRz:     616
0:0000.0/         Hpqrs,        CZ:     258
0:0000.0/         Hpqrs,         H:    2464
0:0000.0/         Hpqrs,    Ybasis:    1232
0:0000.0/         Hpqrs,   Ybasis':    1232
```

Example 96: H2O log, gate statistics

The `Counts:` line summarizes total numbers of rotations and sequential gates. The parallel, nesting and redundant gate removal statistics are the same as the sequential ones because we didn't do any of those operations on the circuit. The system then gives a complete dump of where all the 15,362 sequential gates are used.

At this point, the molecule is grown into a single unitary matrix (which can take a long time and then is exponentiated for Trotterization and Phase Estimation. At this point, we have a complete optimized version of the circuit that we can execute any number of times to obtain the ground state energy. Typically we'll see from 10 to 20 runs depending on how numerically stable the result is.

As we phase estimate, we see the results for every bit computed:

```
0:0000.3/  >> 27: 0=  0,  4 c2=  8.0 100.0%
0:0000.3/  >> 27: 1=  4,  0 c2=  8.0 100.0% -1.5708
0:0000.3/  >> 27: 001 = 0.125000
0:0000.3/  >> 26: 0=  0,  4 c2=  8.0 100.0%
0:0000.3/  >> 25: 0=  0,  4 c2=  8.0 100.0%
```

```
0:0000.3/  >> 24: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >> 23: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >> 22: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >> 21: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >> 20: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >> 19: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >> 18: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >> 17: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >> 16: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >> 15: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >> 14: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >> 13: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >> 12: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >> 11: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >> 10: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >>  9: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >>  8: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >>  7: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >>  6: 0=  0,  1 c2=  0.5 100.0%
0:0000.3/  >>  5: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >>  4: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >>  3: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >>  2: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >>  1: 1=  1,  0 c2=  0.5 100.0%
0:0000.3/  >>  0: 0=  0,  1 c2=  0.5 100.0%
```

Example 97: H2O log, Phase Estimation

The first 3 lines show the computation for the lowest bit. The reason there are 3 lines is that we're doing an arc-tangent calculation to obtain an extra 2 bits of accuracy (see: http://arxiv.org/abs/1304.074 on Faster Phase Estimation). After this, the lines show the bit number, whether we determined it as a 0 or a 1, the number of times we sampled it as a 1 and the number of times we sampled it as a 0 and finally c2 is the chi squared value with the percentage of time we chose the final answer.

We now are ready to output the final answer for this run:

```
0:0000.3/Result: tst=  0 info=0.9573,104.5000 Trot=  32 Order=1
        Egs=< -84.922750, -75.728698> Phi=3.0643425 prep=1,8,2,9,3,10,4,11,5,12
        bits=0.0111110011011010010000000100001
0:0000.3/!CSV,0.9573,104.5000,32,1,
        -84.922749879453500,-75.728698349136500,
        "P.0111110011011010010000000100001",97.3
0:0000.3/OCC*1.000*0.996*0.987*0.991*0.999 0.013 0.013*1.000*
        0.996*0.987*0.991*0.999 0.013 0.013
0:0000.3/State   expectation: prep[00003e7c]=0.973195 big[00003e7c]=0.973195
```

Example 98: H2O log, final result

The first line (`Result`) shows the parameters to the run. Next we have the computed ground state (`Egs`) both without and with nuclear repulsion as well as the Phase Estimated (`Phi`) and a listing of our initial orbitals (`prep`). Finally, we have the actual `bits` computed during phase estimation.

The second line is the most useful. It's in `CSV` format and by pulling all of these out of the log file, you get a nice summary of all the runs done. The fields in order are:

`0.9573`:	This is from the `.dat` file part of the `info` field showing the hydrogen bond length.
`104.5000`:	This is also from the `info` field showing the bond angle.

`32`:	Trotter Number used
`1`:	Trotter Order
`-84.9227`:	Non-nuclear ground state energy
`-75.7287`:	Ground state energy with nuclear repulsion
“`P.011`:	Phase estimation bits
`97.3`:	Overlap of the prep state with the final ground state found

The next line shows occupancies in the final ground state of the initial prep state and the last line shows the prep state expectation (with the hex bits representing the filled orbitals). The `big` value is which state entry has the biggest overlap with the ground state. When `prep` and `big` aren't the same bits, it says you're prep state isn't the best place to get to the ground state from.

Chapter

9

Built-in Samples

Playing with the executable

LIQ$Ui|\rangle$ contains a number of built-in samples to allow the user to play with the system and to see application areas that LIQ$Ui|\rangle$ has been applied to (this is only a small sampling). Several of these have been described in earlier sections where their source code is available in the `samples` directory. This chapter documents the complete list as well as any options that are available.

__**Big** ent This sample takes no parameters and starts an entanglement test with 16 qubits and grows as far as it's allowed (up to 33 qubits).

__**Chem** chemistry See the sections relating to this function in the Quantum Chemistry sections. The simplest example would be: `__Chem(H2)`. For a list of available molecules, type: `__Chem("")`.

__**ChemFull** chemistry See the sections relating to this function in the Quantum Chemistry sections. This is the detailed call that allows for setting of many parameters: `__ChemFull("molecule",test,"opts",trot,bits,order)`. There is an entire section of this manual just devoted to the `opts` (Quantum Chemistry Options)
.

__**Correct** test Correct tests if the simulator is working correctly. It takes no arguments and simply tests Teleport on different qubits from a larger state vector and will flag if anything unexpected happens. It runs the tests in Code, Circuit and Grown modes to try all variations.

__**Entangle1** ent Call __Entangle1 with the number of qubits you want to entangle (typically 10 through 22). This system will give you detailed statistics on how long it takes to do the various operations.

__**Entangle2** ent

This does the same test as `__Entangle1`, but compiles the circuit to show the difference in timings when using optimized code.
See the Creating a script section for more details.

__Entangles ent This sample takes no parameters but runs 100 entanglement tests on 16 qubits to show the statistics on the bits measured. The should be all 1 or all 0 (each time) and in the end, approximately half should be 1s and half 0s.

__EntEnt entropy Shows the system computing entanglement entropy in two sample different cases. The circuits are output as `.htm` files and the entropy for each qubit at the end is given.

__EIGS math Validates that LAPACK is installed correctly and uses zgeev to compute a Wilkinson test. There are a few cases where LIQ$Ui|\rangle$ may use LAPACK for ancillary statistics and this test validates that the functions are available.

__EPR circ Simply creates drawings of an Einstein-Podolsky-Rosen circuit (a `H`adamard and a `CNOT`) to show simple a circuit drawing.

__Ferro hamiltonian Performs the simulation of a ferromagnetic chain using a first quantized Hamiltonian. There are two arguments. Set the first to `true` if you want to see all the variations of the chain (Isolated, Ferro, Anti-Ferro, Freeze Up, Freeze Down, Freeze Up/Down). If the first argument is `false`, then only the last example will run (the most interesting one). The second argument is `true` if you only want to run each test once, otherwise set it to false.

__JointCNOT braid There are many ways to implement a joint `CNOT` gate for braiding operations with joint measurement and parity control gates. This sample tests several implementation with various input combinations and shows the results. This sample is only provided to show some of the types of research that we do with LIQ$Ui|\rangle$.

__Kraus qecc Documented in the section Channels and POVMs. This is an example of the most sophisticated noise model in the system

__Noise1 qecc Documented in the section Noise + QECC For technical details on amplitude damping, refer to the Amplitude Damping section.

__NoiseAmp qecc See the Full Example in Advanced Noise Models.

__QECC qecc Full details may be found in the section on Stabilizers.

__QFTbench math Benchmarks the `QFT` algorithm used inside of `Shor`. The `CSV` lines are the benchmark results (the first one explains what all the fields are). The

Quantum Fourier Transform is run in each of Code, Circuit and Optimized modes to show the difference in performance. All three show time in seconds and memory used in megabytes.

__**QLSA** linalg This is an example of the Quantum Linear Algebra algorithm by Harrow, Hassidim and Lloyd (http://arxiv.org/abs/0811.3171). The output includes the circuits used (created in the files `QLSA0.htm` (`.tex`) and `QLSA9.htm` (`.tex`)). If you take the `CSV` entries in the output and plot them, you will see a graph like the following:

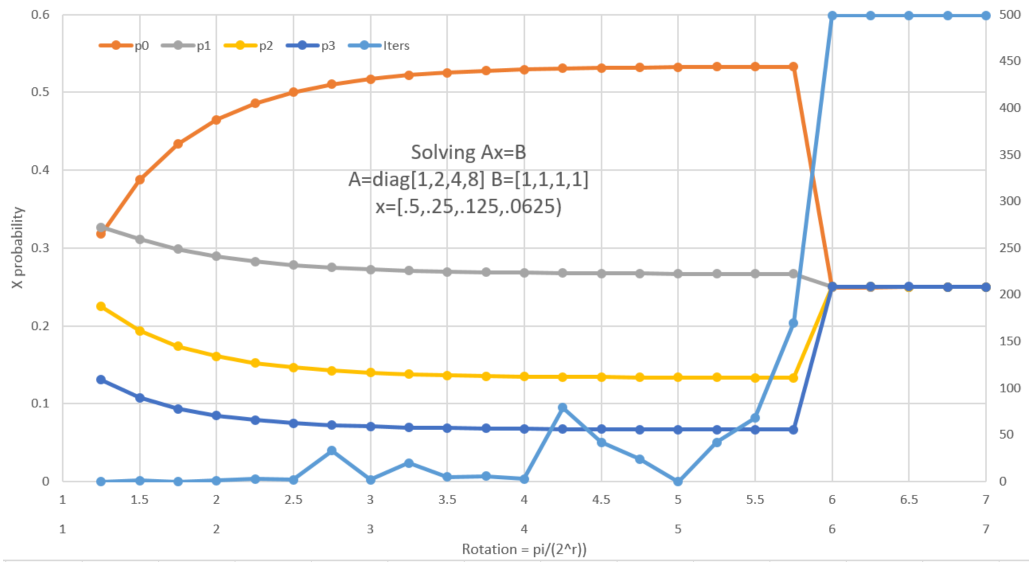


Figure 23: Output from QLSA sample

__**QuAM** memory This is an example of the Quantum Associative memory algorithm by Ventura and Martinez (http://arxiv.org/abs/quant-ph/9807053). The output shows storing a number of key/value pairs (each is a 4 bit hex single digit):

```
0x04:  6.3%
0x1c:  6.3%
0x27:  6.3%
0x39:  6.3%
0x47:  6.3%
0x5e:  6.3%
0x68:  6.3%
0x7c:  6.3%
0x8b:  6.3%
0x9b:  6.3%
0xab:  6.3%
0xbd:  6.3%
0xcd:  6.3%
0xd4:  6.3%
0xec:  6.3%
0xfb:  6.3%
```

Example 99: QuAM: Storing key value pairs

Looking inside the state vector, we can see that the circuit was able to store all the items with equal probability. Now we do a Grover search for the item with key 6:

```
Grover[ 0]: 0x68:  7.7%
Grover[ 1]: 0x68: 12.4%
Grover[ 2]: 0x68: 16.9%
Grover[ 3]: 0x68: 19.2%
Grover[ 4]: 0x68: 18.2%
Grover[ 5]: 0x68: 14.5%
Grover[ 6]: 0x68:  9.6%
Grover[ 7]: 0x68:  5.5%
Grover[ 8]: 0x68:  2.9%
Grover[ 9]: 0xec:  3.7%
Grover[10]: 0xec:  3.8%
Grover[11]: 0x68:  3.2%
Grover[12]: 0x68:  6.0%
Grover[13]: 0x68: 10.3%
Grover[14]: 0x68: 15.1%
Grover[15]: 0x68: 18.6%
Grover[16]: 0x68: 19.1%
```

Example 100: QuAM: Searching for a key

We can see how Grover cycles through the optimal probability of finding the desired key/value pair.

__**QWalk** page rank This is an example of the Quantum PageRank algorithm by Paparo and Martin-Delgado (http://arxiv.org/abs/1112.2079). The argument provided to the function must be one of:

`tiny`:	2 Node graph
`tree`:	7 Node tree graph
`graph`:	7 Node web graph (from the paper)
`<path>`:	Path to file that contains a specified web graph

If you're running from the `samples` directory, a command line using `<path>` would look like:

```
..\bin\Liquid.exe __QWalk(Web_4.graph)
```

The samples directory contains web graphs at sizes 4-9 (which are powers of 2, representing web graphs with from 16-512 vertices). You can compare the classical page range (`PRank`) and the quantum page rank (`QRank`) at the end of the output.

__**Ramsey33** Ham This is an example of solving Ramsey numbers on the D-Wave machine (http://arxiv.org/abs/1201.1842). See the paper for details. The example also outputs the circuits in `Ramsey33_*.htm` (`.tex`) files.

__**SG** Ham This is an example of solving a spin-glass problem with a first quantized Hamiltonian. Circuits are drawn in `SG*.htm` (`.tex`). The couplings used are (1 based):

```
1,5,1; 1,6,1; 1,7,-1; 1,8,1; 2,5,1; 2,6,-1; 2,7,1; 2,8,-1;
3,5,-1; 3,6,1; 3,7,-1; 3,8,1; 4,5,1; 4,6,1; 4,7,-1; 4,8,-1;
9,13,-1; 9,14,-1; 9,15,-1; 9,16,-1; 10,13,1; 10,14,1; 10,15,1;
10,16,1; 11,13,1; 11,14,1; 11,15,-1; 11,16,-1; 12,13,-1;
12,14,1; 12,15,-1; 12,16,-1; 1,9,-1; 2,10,1; 3,11,1; 4,12,-1;
```

Example 101: Spin Glass couplings

__Shor factor The Shor algorithm example takes two parameters. The first is the number to factor and the second is whether to optimize the circuit (`true`=optimize). See the discussion in Creating a script.

__show demo This is just a call to the LIQ$Ui|\rangle$ `printf` equivalent. Pass it a string with surrounding double quotes and it will echo back. This is a simple way to test that the runtime is working correctly.

__Steane7 QECC This a validation that the Steane7 CSS code has been implemented correctly. It will run through one single qubit error on each of the data qubits using `X`, `Y` and `Z` errors to validate that they will all be fixed.

__Teleport basic Implementation of the classic Teleport algorithm. See the discussion in Creating a script.

__TSP Hamiltonian Solves the traveling salesman problem. The argument is the number of cities (5 to 8). A typical final result (for 5 cities) is:

```
Histogram:
TSP 20.0% 1001100101 (E=-10.0379) [ones= 5]
TSP 60.0% 1010001110 (E=-10.0618) [ones= 5]
TSP 10.0% 1010100011 (E=-10.0092) [ones= 5]
TSP 10.0% 1100001101 (E=-10.0573) [ones= 5]
```

Example 102: TSP optimizatoin result

If we plug in the edges for `1010001110` that are dumped (0 is the leftmost bit), we get:

```
Edge[0] = sea to lax is  961 miles
Edge[2] = sea to ord is 1725 miles
Edge[6] = lax to dfw is 1239 miles
Edge[7] = jfk to ord is  742 miles
Edge[8] = jfk to dfw is 1396 miles
```

Example 103: TSP Final Route

So the discovered route is: SEA -> LAX -> DFW -> JFK -> ORD -> SEA.

__UserSample basic This is a placeholder to show how to add user functions to the simulator. See the chapter on Serious Coding for complete information.

Index