Quantum System Simulator
by
Scott Schneider
Submitted to the Department of Physics
and
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Physics
and
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2000
© Scott Schneider, MM. All rights reserved.
The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis document
in whole or in part.

Author: Scott Schneider
Department of Physics
and
Department of Electrical Engineering and Computer Science
May 19, 2000
Certified by: Edward Farhi
Professor
Thesis Supervisor
Accepted by: Arthur C. Smith
Chairman, Department Committee on Graduate Students
Abstract

The Quantum System Simulator is an application that allows a user to specify and simulate quantum mechanical systems with time-dependent Hamiltonians. It allows significant flexibility in the types of Hamiltonians that can be specified. Although its main application will be simulating algorithms in quantum computation, it can handle Hamiltonians with any finite dimension and many types of time dependence. It provides a simple and convenient graphical user interface that enables users to specify complex Hamiltonians by building them out of smaller ones. The simulator engine is designed as a self-contained module, so that it is independent of the user interface, and can be easily enhanced.

Thesis Supervisor: Edward Farhi
Title: Professor
Contents

1 Overview 7

2 Related Research 9

3 QSS Architecture and Implementation 11
   3.1 Platform .................................. 12
   3.2 UI Data Representation ......................... 13
      3.2.1 Dimensions ................................ 13
      3.2.2 Hamiltonians ................................ 13
      3.2.3 Expressions ................................ 17
      3.2.4 Data Representation Classes ................ 17
   3.3 The User Interface .................................. 22
      3.3.1 CQSSView ................................ 23
      3.3.2 CTreeItemID and subclasses .................. 23
      3.3.3 CHPrimaryView ............................... 25
      3.3.4 CParser ................................... 25
   3.4 Simulator Engine .................................. 25
      3.4.1 Distributed Simulation ......................... 25
      3.4.2 Data Representation ............................ 26
      3.4.3 Simulation Algorithm ........................... 26
      3.4.4 Optimizations ................................ 28
   3.5 Testing ............................................ 28
# 4 Examples and Results

4.1 A Simple Example ........................................ 29
4.2 The Satisfiability Algorithm ............................ 29

# 5 Roadmap

5.1 UI Features .................................................. 33
  5.1.1 Wavefunctions and Hamiltonians .................. 33
  5.1.2 More Accessibility ................................. 34
  5.1.3 Global Environments .............................. 34
  5.1.4 Simulation in a Different Thread ................. 35
  5.1.5 Add Expression Features .......................... 35
  5.1.6 Sanity Checks ......................................... 35
5.2 Simulator Features ......................................... 36
  5.2.1 Simulator as Servers ............................... 36
  5.2.2 Speed Up $\delta |\Psi\rangle = -i\Delta t H |\Psi\rangle$ ....... 37
List of Figures

3-1  Hamiltonians of various sizes ................................. 14
3-2  A One-Qubit Primary Hamiltonian ............................ 15
3-3  Composite Hamiltonians ...................................... 16
3-4  Variable Definitions .......................................... 19
3-5  Example Folders ............................................... 20
4-1  Tunneling Hamiltonian ......................................... 30
List of Tables

4.1 Three Qubit Simulation Accuracy #1 ........................................ 31
4.2 Three Qubit Simulation Accuracy #2 ........................................ 31
4.3 Four Qubit Simulation Accuracy .............................................. 32
4.4 Five Qubit Simulation Accuracy .............................................. 32
Chapter 1

Overview

Quantum computation is a relatively young field, which attracted very little attention before 1994. In that year, Peter Shor created a quantum factoring algorithm that is asymptotically superior to its classical analogs. Since then, a few sophisticated quantum algorithms have been devised, but real quantum computers are still in their infancy.

Simulating a quantum algorithm on a classical computer is much slower – exponentially slower – than running the algorithm on a quantum computer would be. For the present, however, simulation is the only way to examine the application of quantum algorithms to specific problems. Several applications have been developed to simulate quantum computers, but they all deal with specific types of quantum algorithms or quantum computers.

The Quantum System Simulator (QSS), by contrast, is designed to simulate a very wide range of quantum mechanical systems. It can simulate Hamiltonians of any finite dimension and many types of time dependence. It is also designed to have a convenient and powerful graphical user interface (GUI). The wavefunctions and Hamiltonians that it stores are displayed in a hierarchical tree. The QSS simplifies the task of describing complex Hamiltonians by allowing the user to specify large Hamiltonians as sums of smaller ones. The QSS’s simulator engine is designed to be independent of the user interface (UI) and to be easily enhanced.

The main impetus for the QSS is the simulation of a quantum algorithm for solving
the satisfiability problem (which is \(NP\)-complete) through adiabatic evolution.[1] (I will refer to this algorithm as the satisfiability algorithm.) The running time of this algorithm is unknown, and if it is found to run in polynomial time, the implications for quantum computation would be tremendous.

The QSS is presently the only application capable of simulating systems with complicated, time-dependent Hamiltonians. It is also the only application that makes it easy to specify the complicated Hamiltonians that correspond to large satisfiability problems.

This paper assumes that the reader has a background in quantum mechanics. Some understanding of quantum computation (and familiarity with the satisfiability algorithm) will be extremely helpful. The descriptions of C++ classes assume an understanding of object-oriented programming.

In chapter two, I discuss existing applications for simulating quantum computers.

In chapter three, I discuss the architecture of the QSS, including the simulator engine.

In chapter four, I discuss a number of features that can be added to the QSS.

The appendix includes user documentation and source code.
Chapter 2

Related Research

At present, there are few software packages that simulate quantum computers. The task of simulating a quantum computer is quite daunting, since a system containing $\ell$ qubits must be represented by $2^\ell$ complex numbers. Most simulators target a specific problem type to get a significant speed-up. For example, one quantum simulator[3] represents a wavefunction with a hashtable, and only stores eigenstates with non-zero amplitudes. This implementation will have a large speed-up when the wavefunction is composed of relatively few eigenstates, but in the general case (where most components exist), the QSS would be significantly slowed by building on this framework.

The Open Qubit project[4] and Kevin Obenland’s simulator[2] both model quantum computation as a series of discrete steps that change the state of one qubit at a time. Two supported types of unitary operators (not Hamiltonians) – single-qubit and controlled operators – alter the wavefunction. Both are described by a $2 \times 2$ matrix, and controlled operators also store a bitmask that limits when the operator is applied (e.g. $U' = (1 - |\text{mask}\rangle \langle \text{mask}|)\hat{I} + |\text{mask}\rangle \langle \text{mask}|U |\text{mask}\rangle$). Obenland’s simulator speeds up quantum simulation by distributing components of the wavefunction across several processors (and occasionally redistributing data). The model used here is not general enough for the QSS – operators should be time-dependent Hamiltonians that can affect all qubits.

The Quantum Computer Emulator[5] comes closest to the flexibility outlined for
the QSS. It simulates quantum systems with time-dependent Hamiltonians that can affect multiple qubits. Hamiltonians can be a sum of arbitrary single-qubit Hamiltonians and $\sigma_x^{(i)} \times \sigma_x^{(j)}$, $\sigma_y^{(i)} \times \sigma_y^{(j)}$, and $\sigma_z^{(i)} \times \sigma_z^{(j)}$ two-qubit interactions. The magnitude of the single-qubit Hamiltonians may vary in time as a constant plus a sinusoidal term. This is too restricted for the QSS.

The QCE's GUI is similarly the closest match to the QSS, but not close enough. Complicated Hamiltonians must be created by specifying the value of every component, rather than building them out of smaller, simpler Hamiltonians.
Chapter 3

QSS Architecture and Implementation

The primary considerations in the design of the QSS were

- flexibility in the types of Hamiltonians allowed,
- a convenient UI for specifying complex Hamiltonians, and
- a simulator engine that is independent of the UI and easily enhanced.

In all cases, I designed the QSS to have a framework that would support the eventual completion (and ongoing improvement) of these goals, and still yield a useful application when only some of the features were implemented. In the first two cases, the QSS is essentially complete. There are a number of features that could be added, though none are necessary. In the last case, the simulator engine is functional, but lacks some powerful features that were originally proposed.

The major issues in designing the QSS were

- the software language, the UI software package, the target operating system, and development environment,
- the representation of wavefunctions and Hamiltonians within the UI (which determines the flexibility of allowed Hamiltonians),

11
• the UI, and

• the simulator engine.

Data representation and user interaction have been divided into distinct tasks, with distinct sets of classes, following an object-oriented approach (and following MFC's document/view architecture). This makes the QSS software simpler by drawing a line (in the form of an abstraction barrier) between the two tasks, so that a given piece of code either manipulates data or handles input from the user.

Each of the major decisions listed above has its own section in this chapter. There is also a section at the end that discusses the testing of the QSS.

3.1 Platform

The QSS was implemented in C/C++, using Microsoft Foundation Classes (MFC), for Windows, in Visual C++ 6.0. I made these decisions because they provide a satisfactory platform for the QSS, and also because they are the tools with which I am most familiar.

C/C++'s greater speed makes it a superior language for the simulator engine – it is a compiled language, and therefore faster than interpreted languages (such as matlab or Java). C/C++ is also a superior language for the UI component because of the UI software packages available (discussed below). Finally, using different languages for different components would have slowed software development significantly and made the QSS more difficult to maintain.

There were many available options for developing the UI. Platform-independence – the ability to run the QSS on any operating system – would be extremely valuable. Unfortunately, no platform-independent UI package is based on C/C++. I chose to use MFC because it provides adequate support for developing a useful GUI, and because I have significant experience programming with it.

Developing for Windows using Visual C++ 6.0 followed from my decision to use MFC. For the QSS, there is no clearly best operating system to target. The Center for
Theoretical Physics uses primarily Apple computers. The athena clusters make UNIX a good choice. Windows can also be found in various places at MIT. Ultimately, the fact that I have experience with MFC, and that MFC is Windows-specific, dictated the choice of Windows.

3.2 UI Data Representation

As previously stated, the main goals for data representation are flexibility and usability. These goals are achieved, for the most part, through three integrated features—flexibility in Hamiltonians’ dimensions, the ability to build large Hamiltonians out of smaller ones, and the ability to specify coefficients and Hamiltonian matrix elements as mathematical expressions.

The first three subsections discuss these features, and the last subsection discusses the framework C++ classes that actually store the data.

3.2.1 Dimensions

A wavefunction or Hamiltonian may have any finite size (dimension). This size consists of zero or more labels. A label is analogous to a qubit, but its dimension may be any integer greater than one. The labels do not all need to have the same dimension, and the order of the labels matters. The total size is the product of the labels’ dimensions. (See Figure 3-1.)

3.2.2 Hamiltonians

Hamiltonians may be either primary or composite Hamiltonians. Primary Hamiltonians are typically small, and have a dense representation. (They store all matrix elements.) Composite Hamiltonians are usually larger, are created as a sum of other Hamiltonians, and have sparse representation (only stores some matrix elements).

A primary Hamiltonian consists of a matrix (a two-dimensional array) of expressions (described below). The dimensions of the matrix are equal to the size of the
Figure 3-1: Hamiltonians of various sizes

- Null Hamiltonian
  - Size: 1
  - No Definitions

- 3-Qubit Hamiltonian
  - Size: 8; $2^3$
  - No Definitions

- Large Hamiltonian
  - Size: 7; 7
  - No Definitions

- Strange Hamiltonian
  - Size: 24; 2, 4, 3
  - No Definitions
A composite Hamiltonian is a weighted sum of other component Hamiltonians (which may be primary or composite). The components must be the same size as or smaller than the composite Hamiltonian. In composite Hamiltonians, each component has an associated coefficient and label mapping. The label mapping specifies how labels in the component Hamiltonian correspond to labels in the composite Hamiltonian. (These two labels must have the same dimensions.) For example, if a single-qubit Hamiltonian is a component of a multi-qubit composite Hamiltonian, the label mapping will specify the qubit of the composite Hamiltonian to which the composite applies.

Building large Hamiltonians out of smaller ones greatly simplifies the task of specifying complicated Hamiltonians. Aside from easing the mechanics of specifying Hamiltonians, it allows a user to think about a complex Hamiltonian as having a conceptual hierarchy, and to only have to specify the relationships between two adjacent levels in this hierarchy. Figure 3-3 shows the Hamiltonians that correspond
Figure 3-3: Composite Hamiltonians

![Composite Hamiltonians Diagram]

- **3-bit Base Hamiltonian**
  - Size: 8; $2^3$
  - Components
    - Standard: 2-bit $\psi_0 \Rightarrow \psi_1$
      - Coefficients: 1
    - Label Assignment
      - 0
      - 1
      - No Definitions

- **3-bit Problem Hamiltonian**
  - No Definitions

- **3-bit Total Hamiltonian**
  - Size: 8; $2^3$
  - Components
    - 3-bit Base Hamiltonian
      - Coefficients: 1 - (t / period)
    - Label Assignment
      - No Definitions
    - 3-bit Problem Hamiltonian
      - Coefficients: t / period
to the solution of a particular 3-qubit instance of the satisfiability algorithm. The components of the "problem Hamiltonian" are 2-qubit primary Hamiltonians that represent specific types of constraints. The problem Hamiltonian is composed of three such constraints, and specifies the qubits to which these constraints apply. The total Hamiltonian ties the base and problem Hamiltonians together by introducing time-dependence.

3.2.3 Expressions

Expressions can consist of complex numbers, variables, mathematical operators, and functions.

Certain variables, such as $t$ and label, have a specific meaning to the QSS. Other variables may be defined by the user. This is helpful in using fundamental constants, and also in making Hamiltonians that are a function of one or more variables. For example, the tunneling Hamiltonian:

$$
\begin{pmatrix}
1 & \delta \\
\delta^* & -1 \\
\end{pmatrix}
$$

Presently, the following mathematical operations and functions are supported: addition, subtraction, multiplication, division, exponentiation, complex conjugates, sine, cosine, and tangent.

3.2.4 Data Representation Classes

The data in the QSS – the wavefunctions and Hamiltonians – are organized into folders, which are represented by CQSSDoc objects. Wavefunctions and Hamiltonians are represented by the CGUIWavefunction and CGUIHamiltonian classes, which are derived from the CGUIHWOBJ and CGUIOBJStub classes. These classes make use of the CDimensions, CExpression, and CEnvironment classes.
CDimensions

This class stores the size of a wavefunction or Hamiltonian. If the labels have the same dimension, it stores this dimension and the number of labels. Otherwise, it stores an array of label dimensions. It also stores the total size.

CExpression and subclasses

CExpression is an abstract base class that represents an expression. It has a number of virtual member functions, the most important of which are Evaluate(), ToString(), ParseToken(), and Serialize().

Evaluate() takes a CEnvWrapper object (discussed below) as input, and returns a pointer to a newly allocated CExpression object. If the input defines all variables used in this CExpression object, the function returns a CExprComplex object. Otherwise, it returns a simplified (or identical) copy of this.

ToString() formats this into a string.

ParseToken() parses a single token into a CExpression object, and returns NULL if the token does not correspond to any CExpression object. This function is static, but subclasses should still provide their own implementations. CExpression::ParseToken() calls the subclasses’ implementations in the correct order.

Serialize() writes this object into or reads it out of a data stream (e.g. a file or network connection).

The subclasses of CExpression are CExprComplex, CExprVariable, and CExprOp.

CExprComplex stores a complex number as two double-precision (64 bit) floating-point numbers.

CExprVariable stores a variable by its name. See the description of CEnvWrapper for a description of how CExprVariable::Evaluate() works.

CExprOp represents an operator and one or more operands. It stores the operation type and an array of operands (CExpression pointers).
CEnvironment and CEnvWrapper

CEnvironment stores an array of variable names and an array of corresponding variable values. The values are CExpression pointers. (See Figure 3-4.)

CEnvWrapper is a linked list of CEnvironment objects. CEnvWrapper implements the idea of nested environments. When a variable is defined in several environments in a linked list, the value closest to the beginning is the appropriate one. Once a variable’s value is found, the value should be evaluated in the linked list of environments that begins after the variable definition’s environment.

CEnvWrapper::Lookup() iterates down its linked list, looking up a variable name in each CEnvironment object, and returns the first variable value found and the CEnvWrapper object where it was found (or two NULL pointers).
CExprVariable::Evaluate() first calls CEnvWrapper::Lookup() to find its variable’s value. If none is found, it returns a copy of this. Otherwise, it returns the result of evaluating the value in the environment after the one where the variable was found.

**CQSSDoc**

This class implements a folder: a set of wavefunctions and Hamiltonians that are saved in a single file. It has a name, which must be unique – no two currently loaded folders may have the same name. It stores an array of wavefunctions and an array of Hamiltonians. (See Figure 3-5.)

**CGUIObjStub and CGUIHWObject**

CGUIObjStub is the base class for any data type that will ever be stored in a folder. It stores a pointer to its folder, as well as the name of the object, which must be unique within that folder and type of object. (A Hamiltonian and a wavefunction may have the same name).

CGUIHWOObject is a subclass of CGUIObjStub, and is a superclass of CGUIWavefunction and CGUIHamiltonian. It stores the object’s size in a CDimensions
object. It also stores variable definitions for that object in a CEnvironment object.

CGUIWavefunction and subclasses

CGUIWavefunction is a subclass of CGUIHWOBJECT that represents a wavefunction. Its subclasses are CGUWPrimary and CGUIWAsFunction. The stored wavefunction need not be normalized. (It will be normalized before simulation.) CGUIWavefunction::Compile() creates a CSimWavefunction object (discussed below) for simulation.

CGUWPrimary stores the eigenstate coefficients of each eigenstates as an array of CExpression pointers.

CGUIWAsFunction stores a single CExpression object that can be evaluated to yield the coefficient for each state. The special variable, label, indicates the zero-based index of the eigenstate.

CGUHHamiltonian and subclasses

CGUHHamiltonian is a subclass of CGUIHWOBJECT that represents a Hamiltonian. Its subclasses are CGUIHPrimary and CGUIHComposite. CGUHHamiltonian::Compile() creates a CSimHamiltonian object (discussed below) in a straightforward way.

CGUIHPrimary stores the upper right portion of the Hamiltonian matrix in a two-dimensional vector of CExpression pointers. It also stores a pointer to a CHPrimaryView object (discussed below). (See Figure 3-2.)

CGUIHComposite stores an array of CGUIHComposite::CComponent objects. Each CComponent object stores a pointer to the component Hamiltonian, an array of integers that represents the label mapping, a CExpression pointer that represents the component coefficient, and a CEnvironment object that represents parameter definitions for that component. (See Figure 3-3.)

The component Hamiltonian must not introduce cyclic dependence among Hamiltonians. (In other words, the composite Hamiltonian may not be a component of the component Hamiltonian.) Furthermore, the component Hamiltonian must not introduce cyclic dependence among folders. (Composite Hamiltonians from one folder may
contain component Hamiltonians from another folder, but that second folder’s com-
posite Hamiltonians may not contain component Hamiltonians from the first folder.)

This first restriction is obviously necessary. The second restriction (regarding
folders) creates several necessary restrictions for the user, which could have been
avoided with a different design. Since the CComponent object contains a pointer
to the component Hamiltonian, when the folders are loaded from files, the compo-
nent Hamiltonian’s folder must be loaded before the composite Hamiltonian’s folder.
Therefore, there can be no cyclic folder dependence.

Alternative schemes that allow one folder to be loaded while components’ folders
are unloaded require significant additional work. A CComponent object must store
the component Hamiltonian’s name and folder’s name (or other identifying informa-
tion) rather than a pointer. These names must be matched against actual names
when the component’s folder is loaded. When a component Hamiltonian’s name is
changed, all composite Hamiltonians in all folders must be updated, and so on. Ulti-
mately, the minor benefits of a more complicated design did not justify the additional
implementation effort.

The length of the array in CComponent that stores label mappings is equal to
the number of the component’s labels. Each element in the array is a zero-based
number that indicates one of the composite Hamiltonian’s labels. (Therefore, it must
be non-negative and less than the number of the composite Hamiltonian’s labels.)
No two component labels may be mapped to the same composite Hamiltonian label.
Every component label must be mapped to a composite Hamiltonian label with the
same dimension.

3.3 The User Interface

Data in the QSS is presented, for the most part, in a single tree control. The hierarchy
of this tree is exactly parallel to the data hierarchy. The folders are at the top
level. Below this are the wavefunctions and Hamiltonians, and below this are the
properties of the wavefunctions and Hamiltonians (including components of composite
Hamiltonians).

  CHPrimaryView displays and allows editing of a primary Hamiltonian’s matrix elements.

  CParser translates text that the user has entered into CExpression objects.

### 3.3.1 CQSSView

This class implements the tree control. Only one instance of this class exists. It receives all user input to the tree control, and handles all of it (except for item editing, as discussed under CTreeItemID). Some of the tasks it handles include creating and destroying folders, wavefunctions, and Hamiltonians, and adding and removing variable definitions and components.

### 3.3.2 CTreeItemID and subclasses

One instance of a subclass of CTreeItemID exists for every item in the tree control. Every item contains information about exactly where it is in the tree control. Each CTreeItemID object has a pointer to the folder to which it belongs. If it is not a top-level item naming a folder, it has a pointer to a wavefunction or Hamiltonian. And so on.

CTreeItemID’s three important virtual functions are GetText(), OnBeginlabeledit(), and OnEndlabeledit(). GetText() determines what text should be displayed in that tree control item. OnBeginlabeledit() determines whether editing of a tree control item is possible, and prepares the item for editing. OnEndlabeledit() validates text that the user has typed in, and displays an error message or applies the input to data objects.

The subclasses are

**CTreeItemIDDoc** displays a document’s name.

**CTreeItemIDH** is a base class for all classes that display data about a wavefunction or Hamiltonian.
CTreeItemIDHHeader displays a wavefunction or Hamiltonian’s name.

CTreeItemIDHSize displays a wavefunction or Hamiltonian’s size.

CTreeItemIDHDefHeader displays the header for a list of variable definitions.

CTreeItemIDHDefItem displays an individual variable definition.

CTreeItemIDHCentHeader displays the header for a composite Hamiltonian’s list of components.

CTreeItemIDHCentItem is a base class for all classes that display data about a component Hamiltonian.

CTreeItemIDHCentItemHeader displays a component’s name.

CTreeItemIDHCentCoeff displays a component’s coefficient.

CTreeItemIDHCentLabelHeader displays the header for a component’s label mapping.

CTreeItemIDHCentLabelItem displays a single label mapping for a component.

CTreeItemIDHCentDefHeader displays the header for variable definitions for a component.

CTreeItemIDHCentDefItem displays a single variable definition for a component.

CTreeItemIDWCoeffHeader displays the header for a primary wavefunction’s list of coefficients.

CTreeItemIDWCoeffItem displays a single coefficient of a primary wavefunction.

CTreeItemIDWCoeffFunc displays the coefficient expression for a function-form wavefunction.
3.3.3 CHPrimaryView

The matrix elements of a primary Hamiltonian cannot be well accommodated by a tree view. Instead, they are displayed and edited in a separate child window, represented by the CHPrimaryView class. The user can double-click on any element and edit it. If the user changes an element in the lower-left section of the matrix, CHPrimaryView stores the complex conjugate of the entered expression into the upper-right of the primary Hamiltonian's matrix. (See Figure 3-2.)

3.3.4 CParser

This class contains several static functions that parse text that the user has entered into CExpression objects. CParser::ParseToExpression() is the main function. It first calls CParser::Tokenize(), which turns the text into a string of tokens. CParser::ParseTokens() then turns this list of tokens into a CExpression object.

3.4 Simulator Engine

One measures a quantum computation simulator's usefulness by how large a system it can simulate in a reasonable amount of time. The main difficulty in simulating a large quantum system is the size of the wavefunction. In a system with \( \ell \) qubits, the wavefunction is represented by \( 2^\ell \) complex numbers. (20 qubits take up several megabytes of memory.) The simulation consists in iteratively updating the wavefunction, and every coefficient in the wavefunction will probably be modified in every iteration.

3.4.1 Distributed Simulation

If the simulated Hamiltonian is complex enough that every coefficient of the wavefunction affects every other coefficient within a few iterations (which it will be if it is meant to solve an \( NP \)-complete problem), the simulation cannot be broken down into chunks that spend a significant amount of time separately simulating small parts
of the wavefunction. This means that a single quantum simulation cannot be distributed.

The original proposal for the QSS separated the UI and simulator engine into separate client and server applications, so that several different simulations could run simultaneously on several servers. This was not implemented due to time constraints, but if it is implemented later, the simulator engine will not need to be re-implemented. (This is discussed in detail in section 5.2.1.)

3.4.2 Data Representation

CSimWavefunction

The wavefunction is represented by a CSimWavefunction object. CSimWavefunction consists of a CDimensions object and an array of complex numbers. (The array’s length is equal to the wavefunction’s size.)

CSimHamiltonian and CSimSubHam

The Hamiltonian has a sparse representation. (Otherwise, its size would be the square of the wavefunction’s size, and a 20 qubit Hamiltonian would take up several terabytes of memory.) A CSimHamiltonian object consists of a CDimensions object and an array of CSimSubHam objects.

A CSimSubHam object stores the upper right portion of the Hamiltonian matrix as a two-dimensional array of CExpression pointers. (The elements must be CExpression pointers rather than complex numbers because they have time dependence. The next chapter discusses some alternatives.) It also stores label mappings – an array of integers that lists the labels of the CSimHamiltonian object that this CSimSubHam object affects.

3.4.3 Simulation Algorithm

The algorithm used is a straightforward implementation of the Runge-Kutta method with variable step sizes. (Runge-Kutta is a fourth-order method – its error is $O(\Delta t^4)$)
for step size $\Delta t$. It takes start and end times and minimum and maximum error tolerances as input. It begins by setting the time to the start time, and updating the wavefunction as it steps forward. Given a wavefunction $|\Psi(t)\rangle$, a Hamiltonian $H(t)$, and a step size $\Delta t$,

$$
\delta |\Psi_1\rangle = -i\Delta t H(t) |\Psi(t)\rangle \\
\delta |\Psi_2\rangle = -i\Delta t H(t + \frac{\Delta t}{2})[|\Psi(t)\rangle + \frac{1}{2} \delta |\Psi_1\rangle] \\
\delta |\Psi_3\rangle = -i\Delta t H(t + \frac{\Delta t}{2})[|\Psi(t)\rangle + \frac{1}{2} \delta |\Psi_2\rangle] \\
\delta |\Psi_4\rangle = -i\Delta t H(t + \Delta t)[|\Psi(t)\rangle + \delta |\Psi_3\rangle] \\
|\Psi(t + \Delta t)\rangle = |\Psi(t)\rangle + \frac{1}{6} [\delta |\Psi_1\rangle + 2\delta |\Psi_2\rangle + 2\delta |\Psi_3\rangle + \delta |\Psi_4\rangle] 
$$

To determine $\delta |\Psi\rangle = -i\Delta t H |\Psi\rangle$, the simulator iterates through the sub-Hamiltonians. For each sub-Hamiltonian, it iterates through $a$ where $0 \leq a < \text{dimension}(\text{Hamiltonian})$. For each sub-Hamiltonian and coefficient (indexed by $a$), it iterates through $b$ where $a \leq b < \text{dimension}(\text{SubHam})$, adding $-i\Delta t \text{SubHam}_{ab} \langle a |\Psi\rangle$ to $\delta \langle b |\Psi\rangle$. (If $a \neq b$, it also adds $-i\Delta t \text{SubHam}^*_{ab} \langle b |\Psi\rangle$ to $\delta \langle a |\Psi\rangle$.)

The sub-Hamiltonian will probably have a smaller dimension than the total Hamiltonian. Therefore, $a$ is taken to have a different value as an index of the sub-Hamiltonian than as an index of the total Hamiltonian. (The details of this conversion are not interesting.)

In pseudo-code,

for $\text{SubHam} \in \text{Hamiltonian}$

for $a = 0$ to $\text{dimension}(\text{Hamiltonian}) - 1$

for $b = a$ to $\text{dimension}(\text{SubHam}) - 1$

$$
\delta \langle b |\Psi\rangle += -i\Delta t \text{SubHam}_{ab} \langle a |\Psi\rangle \\
\text{if } a \neq b \text{ then} \\
\delta \langle a |\Psi\rangle += -i\Delta t \text{SubHam}^*_{ab} \langle b |\Psi\rangle 
$$

At the beginning of the simulation, and every 100 (or so) iterations, the simulation will consider whether to change $\Delta t$. It first finds $|\Psi(t + \Delta t)\rangle$ in two ways – by one
Runge-Kutta step of size $\Delta t$ and by two Runge-Kutta steps of size $\frac{1}{2}\Delta t$ – and takes the difference between the two results (as a sum of squares of coefficient differences). If this difference is more than the maximum error tolerance, it reduces $\Delta t$ and checks again. If this is less than the minimum error tolerance, it increases $\Delta t$ and checks again. If the error is within tolerance levels, it continues the simulation using $\Delta t$.

### 3.4.4 Optimizations

The QSS does a small amount of pre-processing to reduce the amount of time needed in iterating over the sub-Hamiltonians. When several CSimSubHam objects map to the same labels, or one CSimSubHam object maps to a superset of the labels mapped to by another CSimSubHam, the CSimSubHam objects can be added together.

### 3.5 Testing

No part of the QSS has been tested thoroughly. Most of the testing has focused on configuring and simulating the satisfiability algorithm.

If the user is only dealing with qubits – wavefunctions and Hamiltonians with any number of labels, all of dimension two – and linear time-dependence, the QSS will probably not have any major bugs. The UI has a number of rough edges (for example, uninformative error messages). Serious bugs in the UI will probably have work-arounds. The simulator engine has been tested somewhat, and probably does not have any major bugs in simulating qubits and linear time-dependence.

If the user is dealing with labels of varying sizes and/or non-linear time-dependence, the QSS will probably crash and burn in a variety of ways.
Chapter 4

Examples and Results

4.1 A Simple Example

This example applies the Hadamard operator to the zeroeth bit of a two-qubit wavefunction. The initial wavefunction is \( \frac{1}{2} (|00\rangle + |01\rangle + |10\rangle + |11\rangle) \). The operator is applied for \( \frac{\pi}{2} \) time units to transform it into approximately \( \frac{1}{\sqrt{2}} (|00\rangle + |10\rangle) \). (See Figure 4-1.)

4.2 The Satisfiability Algorithm

A satisfiability problem consists of a list of constraints, where each constraint is a logical function of a few bits. A solution is a list of bit values that satisfies all of the constraints. In the input wavefunction, all coefficients are \( 2^{-\ell/2} \) (where \( \ell \) is the number of qubits). The Hamiltonian for a satisfiability problem has two main parts - the base Hamiltonian and the problem Hamiltonian.

The base Hamiltonian consists of the single-qubit Hamiltonian, \( \frac{1}{2} (1 - \sigma_x) \), applied to each qubit with (constant) magnitude equal to the number of satisfiability clauses that contain that qubit.

The problem Hamiltonian is a sum of constraint Hamiltonians applied to the appropriate qubits (with magnitude one). A constraint Hamiltonian is a two- or three-qubit Hamiltonian with no off-diagonal elements. Diagonal elements are zero if
Figure 4-1: Tunneling Hamiltonian
Table 4.1: Three Qubit Simulation Accuracy #1

| period | $|\langle \Psi |110 \rangle|^2$ |
|--------|------------------|
| 5      | 0.45             |
| 10     | 0.68             |
| 15     | 0.84             |
| 20     | 0.93             |
| 25     | 0.970            |
| 30     | 0.985            |
| 35     | 0.990            |
| 40     | 0.993            |
| 50     | 0.9981           |
| 75     | 0.9995           |
| 100    | 0.9998           |

Table 4.2: Three Qubit Simulation Accuracy #2

| period | $|\langle \Psi |010 \rangle|^2$ |
|--------|------------------|
| 10     | 0.39             |
| 20     | 0.62             |
| 40     | 0.87             |
| 60     | 0.95             |
| 80     | 0.983            |
| 100    | 0.993            |
| 150    | 0.9992           |

the corresponding bit values satisfy the constraint, and one otherwise.

The total Hamiltonian is the sum of the base Hamiltonian, with magnitude $1 - \frac{t}{\text{period}}$, and the problem Hamiltonian, with magnitude $\frac{t}{\text{period}}$. period is a variable that must be defined before the simulation is run.

Figure 3-3 shows the Hamiltonian that corresponds to bit 0 implies bit 1, bits 0 and 2 disagree, and bits 1 and 2 agree – $(\Psi_0 \Rightarrow \Psi_1) \land \neg (\Psi_0 \leftrightarrow \Psi_2) \land (\Psi_1 \leftrightarrow \Psi_2)$. The correct answer is bits 1 and 2 equal one, and bit 0 equals zero – 110. Table 4.1 shows results of simulating this Hamiltonian with different values of period.

Another 3-bit example solves $- (\Psi_0 \oplus \Psi_1 \oplus \Psi_2) \land ((\Psi_1 \Rightarrow \Psi_2) \Rightarrow \Psi_0) \land \neg (\Psi_2 \leftrightarrow \Psi_1)$. ($\oplus$ is exclusive-or.) The solution is 010. (See Table 4.2.)

A 4-bit example solves $(\Psi_0 + \Psi_1 + \Psi_2 = 2) \land ((\Psi_1 \land \Psi_2) \lor \Psi_3) \land (\Psi_3 \leftrightarrow \Psi_0) \land (\Psi_1 \leftrightarrow \Psi_2)$.
Table 4.3: Four Qubit Simulation Accuracy

| period | $|\langle \Psi|0110\rangle|^2$ |
|--------|-------------------------------|
| 10     | 0.30                          |
| 20     | 0.41                          |
| 40     | 0.62                          |
| 80     | 0.82                          |
| 120    | 0.94                          |
| 160    | 0.975                         |
| 200    | 0.990                         |

Table 4.4: Five Qubit Simulation Accuracy

| period | $|\langle \Psi|10010\rangle|^2$ |
|--------|-------------------------------|
| 20     | 0.42                          |
| 40     | 0.67                          |
| 80     | 0.89                          |
| 120    | 0.967                         |
| 160    | 0.989                         |
| 200    | 0.997                         |

The solution is 0110. (See Table 4.3.)

A 5-bit example solves $(\Psi_0 + \Psi_1 + \Psi_2 = 1) \land ((\Psi_4 \Rightarrow \Psi_3) \Rightarrow \Psi_2) \land (\Psi_0 \Leftrightarrow \Psi_2)$. The solution is 10010. (See Table 4.4.)
Chapter 5

Roadmap

This chapter contains a list of potential features that would be useful for the QSS. The suggested UI features would slightly enhance the usability of the QSS. Most of the major improvements to the QSS will center on accelerating the simulator engine.

5.1 UI Features

5.1.1 Wavefunctions and Hamiltonians

Create classes that represent a composite wavefunction - a wavefunction that is a sum of other component wavefunctions. Unlike composite Hamiltonians, component wavefunctions must have the same size as the composite wavefunction.

Create classes that represent a product wavefunction - a wavefunction that is the product of other wavefunctions. (Two one-qubit wavefunctions would create one two-qubit wavefunction.) Every component wavefunction would have a label mapping that, for each component label, specifies the product label generates. No two component labels (from the same or different components) could map to the same product label.

Create classes that represent a function-form Hamiltonian - a single CExpression is specified that is a function of two special variables, row and column, which vary over $0 \leq row \leq column < \text{dimension}(Hamiltonian)$. (The expression is only ever
evaluated with \( row \leq column \).

Create classes that represent a product Hamiltonian — a Hamiltonian that is the product of other Hamiltonians. Every component Hamiltonian would have a label mapping, just like a composite Hamiltonian. Unlike a product wavefunction, several component labels (from different components) can specify the same product label. One can think of a component Hamiltonian as being padded out to the product Hamiltonian’s full size by taking a tensor product of the component with an identity matrix of the appropriate size. The product Hamiltonian is simply equal to the product of all its padded components. (Obviously, the order of component Hamiltonians matters.)

5.1.2 More Accessibility

Create accelerators. These allow the user to perform actions (e.g. deleting a Hamiltonian) with the keyboard.

Create a right-click menu in the tree control. The contents of the menu need to be generated based on the tree item that is being right-clicked, but CQSSView (rather than CTreeItemID subclasses) should determine the right-click menu content, because the right-click menu content is a subset of the main menu, and CQSSView already has all the necessary OnUpdate() functions.

Enable drag-and-drop. This would be useful for adding components to Hamiltonians, rearranging wavefunctions and Hamiltonians, and beginning a simulation.

5.1.3 Global Environments

Allow the creation of global environments (at the folder level) where global variables (such as \( \hbar \), or error tolerances) can be defined. Rules will have to be made for whether several global environments can exist in one folder, and how to order multiple global environments (from the same folder or different folders).
5.1.4 Simulation in a Different Thread

Running the simulation from a different thread allows many useful features to be added. A progress bar could indicate a simulation's progress. The simulation could be cancellable. (Both of these require slight modifications to the simulator engine also.) The simulator engine could return intermediate wavefunctions. The user could continue examining and modifying data in the QSS while the simulation runs.

Dealing with thread creation, cleanup, cancellation, etc, should be handled by the CSimHamiltonian class. The API that it would have to provide – StartSimulation(), CancelSimulation(), and a callback mechanism for simulation completion – would be essentially identical to the API required for running simulations on separate server applications.

5.1.5 Add Expression Features

Add new functions and special variables to CExpression.

Adding new functions to CExprOp is a straightforward task. One new function that would be extremely useful is the unit step. This function would allow parts of a Hamiltonian to take effect for only part of a simulation.

Many special variables could be added to the QSS. Adding a special variable does not involve modifying CExpression or its subclasses, although it does require modifying some other part of the QSS (most likely, CGUIHamiltonian::Compile()), depending on the variable. Some useful special variables would be t_start, t_end, min_error_tolerance, and max_error_tolerance. If all four of these are defined, the QSS will not need to prompt a user for these values when beginning a simulation.

5.1.6 Sanity Checks

Add features that allow the user to check whether he correctly specified a Hamiltonian. One way to do this is to allow the user to view the matrix elements of a part of a Hamiltonian. (It may be so large that it is impractical to view the whole matrix.)
Features of this type would clearly be useful, but I cannot think of more than one, and I am not even convinced of that feature’s usefulness.

5.2 Simulator Features

5.2.1 Simulator as Servers

If the simulator is separated into a server application, a user could run several servers and perform several simulations simultaneously. There are, of course, many issues involved in such a separation.

Platform

The first issue is choosing the platform on which the simulator would run. Choosing Windows would minimize the difficulty of separating the two parts. There are, however, few places at MIT with a large number of easily accessible Windows machines. Athena clusters provide a strong argument for UNIX.

The simulator engine is, for the most part, platform-independent. Setting up the two parts of the QSS to run on different operating systems introduces significant additional complexity, but not necessarily a prohibitive amount. It may even be worthwhile to create versions of the server application for Windows and UNIX (and Apple computers).

Additional Work

The code used by the simulator engine should be separated from the code used only by the UI, and made into a dynamically-linked library (for Windows) and a shared object file (for UNIX, if it is being used). The simulator code needs to be made entirely platform-independent. Code will have to be added to the UI client and simulator server to transfer data over the network. The UI will have to be enhanced so that the user can specify the location of servers, and so that several similar simulations can be initiated at once. (For example, running the satisfiability algorithm on the same
problem with several values of period.) Code will also have to be written that runs
the server as an independent application (handles application startup and incoming
simulation requests).

The simulator engine is presently Windows-specific because it uses the CExpression
classes, which use MFC's object serialization and run-time class identification
functionality. This is functionality that CExpression needs, and so this functionality
will need to be re-implemented. This should not be terribly difficult. A replacement
class for COobject (CExpression's current base class), with virtual functions analogous
to Serialize() and IsKindOf(), must be implemented.

The client's and server's network communication should be handled within CSimHamiltonian. This class should have public API, used by the client UI, including functions
such as StartSimulation(), CancelSimulation(), and a callback mechanism for simu-
lation completion. It should also have a private API, used by the server, including
functions such as OnStartSimulation() and OnCancelSimulation(). Functions in the
public API should initiate or use an existing network connection to a server, and the
private API should initiate or alter a simulation.

One good way for the user to configure Hamiltonians to vary across servers is
to create new special variables, SimNumber and SimTotal, that indicate the num-
ero of a particular simulation, and the total number of simulations being started,
respectively.

5.2.2 Speed Up
\[ \delta \ket{\Psi} = -i \Delta t \ H \ket{\Psi} \]

CSimSubHam Speed-ups

CSimSubHam objects contain many CExpression pointers. At each value of \( t \), these
expression objects should be evaluated only once. This can easily be done immediately
within the loop that iterates through sub-Hamiltonians by creating a two-dimensional
array of complex numbers, and referring to that object in the rest of the loop.

An alternative is to replace CSimSubHam's two-dimensional array of CExpression
pointers with a two-dimensional array of complex numbers and a single CExpression
sion pointer, which contains the time dependence as a coefficient. This means that CGUIHPrimary::Compile() may create a CSimHamiltonian object with more than one CSimSubHam. It also makes it more difficult to add two CSimSubHam objects.

One could go further than this, and shorten the two-dimensional array to discard matrix elements that are zero. This could provide a valuable speedup in simulating Hamiltonians that implement the satisfiability algorithm, as constraint Hamiltonians have no off-diagonal terms. This speed-up will be lessened by the already implemented optimization that adds sub-Hamiltonians together.

Memory Access Time

Memory access time is a critical factor in the simulator’s performance. In large simulations, most of the memory access time will be spent pulling wavefunction coefficients into the cache from RAM. The best way to improve memory access time is to increase locality of reference, which decreases the frequency of cache misses. This is rather difficult to do for a satisfiability Hamiltonian because it has a $\sigma_z$ operating on each qubit.

One way to increase locality of reference is to reverse the order of the loop that iterates on sub-Hamiltonians and the loop that iterates on coefficients of the wavefunction. This will reduce the number of passes through the whole wavefunction. It will also get a larger speed-up from the locality of reference that exists near the end of an iteration on the wavefunction coefficients. (Every time an off-diagonal Hamiltonian term from the upper right is applied, the corresponding term from the lower left is also applied, and so later in the iteration on wavefunction coefficients there are fewer off-diagonal terms left to be applied.) A minor drawback is that the previous suggestion of evaluating sub-Hamiltonians only once per time slice now requires evaluating and storing all sub-Hamiltonians at the beginning of each time slice.

A radical way to increase locality of reference is to restructure the wavefunction. One can re-order the wavefunction so that coefficients with fewer labels equal to one are stored earlier. (This will only help with the satisfiability algorithm and similar Hamiltonians, where the only off-diagonal terms come from $\sigma_x$’s.) In this scheme,
the wavefunction is divided up into groups, where in each group, the sum of the
label values has a constant value. Therefore, applying a series of $\sigma_z$'s to a coefficient
in a certain group only requires accessing coefficients in the two adjacent groups.
Unfortunately, the size of the largest group (where the number of zero labels equals
the number of one labels) is $\Theta(\frac{1}{\sqrt{t}}2^t)$. For six qubits, the middle group contains about
a third of the coefficients. For 20 qubits, the middle group contains about a sixth of
the coefficients.

Another way to reduce memory access time is to reduce memory usage. Complex
numbers could be represented by two 32-bit integers rather than two 64-bit floating
point numbers. This would also speed up simulation considerably by replacing floating
point multiplications with integer multiplications. If integers have enough precision,
this is definitely a good idea.

Optimize the Inner Loop

The best way to find performance-critical operations in the inner loop is to profile
the code. It is entirely reasonable to expect to improve performance by a factor of
ten by tweaking the code.

The following scheme reduces the amount of time spent iterating through sub-
Hamiltonians. As the algorithm iterates through wavefunction coefficients, it keeps
a list of all of the coefficients with which the current coefficient interacts (and the
corresponding Hamiltonian elements' values) - all the sub-Hamiltonian's information
about the current coefficient. (For the satisfiability algorithm, this list will contain
all coefficients whose labels differ in only one bit from the current coefficient.) At
the end of each iteration, it finds the highest-order label that changed. For each sub-
Hamiltonian that depends on this label or a lower-order label (each sub-Hamiltonians
that depends on a label that changed), it subtracts the sub-Hamiltonian's old contri-
butions (for the old coefficient) to the interaction list and adds its new contributions.

Although the algorithm does twice as much work when it checks a sub-Hamiltonian,
it does this less frequently for sub-Hamiltonians that do not depend on the few lowest-
order labels. One could augment this algorithm by re-ordering the labels in the wave-
function and sub-Hamiltonians so that fewer sub-Hamiltonians use the lowest-order labels. It is probably a good idea to only apply this algorithm to sub-Hamiltonians that do not affect the lowest-order labels, and to apply those sub-Hamiltonians by simply iterating through them.
Bibliography


