Documentation – jaQuzzi Symbolic

Author: Jan Limbeck

Copyright (c) 2006 Jan Limbeck
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation.
The full version of the license can be obtained here:
http://www.gnu.org/licenses/fdl.txt

Introduction:

This extension to jaQuzzi was developed during my programming practical in summer term 2006 at the University of Passau. The idea for the modifications, which will be described later on, came during the lecture Quantum Computing held by PD Dr. Thomas Sturm. During my practical Dr. Thomas Sturm was also my tutor, who helped me wherever he could.

Overview:

The starting point for the development was the latest version of the source code provided by Felix Schuermann under the GPL license. The source code was (mostly) ported from Java 1.3 to Java 1.5. The coding was done using Eclipse. As a consequence the source code, provided by me, is an Eclipse project. The original functionality of the program was not removed and coexists with the new parts, which interact with Maple™. At the moment both calculations are performed at the same time to be able to compare the results of both exact and numeric calculation. Note that this parallel computation will be deactivated at a later time to improve overall performance.
A new package named maple was created, that contains almost all functionality that is needed to access Maple™. The classes in the package will be described later.

Platform independence:

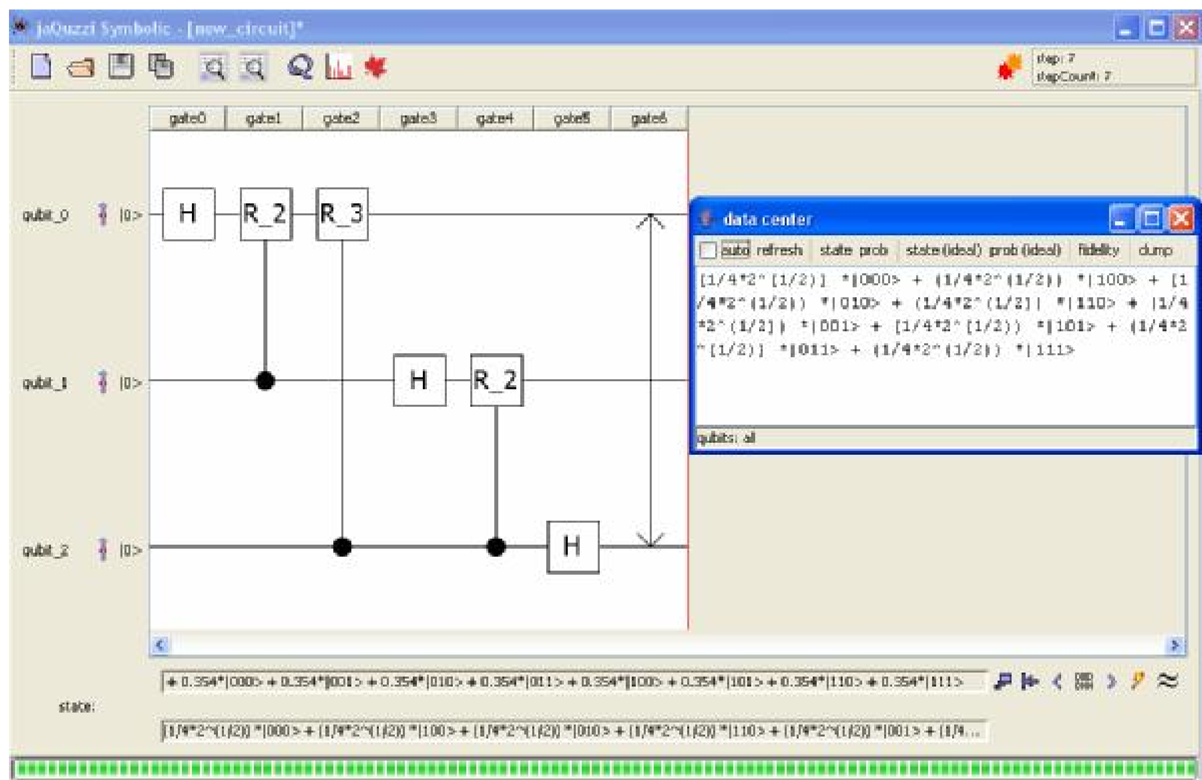
Though this Java application can theoretically run on any platform that has a Java 1.5 virtual machine, there are some restrictions that are due to the way JOpenMaple™ works. It should be clear that you need a working Maple 9.5 or greater Installation on your OS. But as JOpenMaple™ uses the platforms native libraries, path variables to these libraries have to be set. Unfortunately this is not possible at the runtime of the Java application, nor do I know the correct path at compile time. To workaround these problems, a shell script for the underlying OS is created when the path to the bin dir is set in the application. Now you have to close jaQuzzi Symbolic and start it with the shell script, which was created. If you are aware of any better solution feel free to contact me at jan_limbeck@hotmail.com.

Changes to the GUI:

Assuming that you are already familiar with jaQuzzi, there are only two visible changes to the GUI:

- There is now a second textbar that contains the current quantum circuitry state, as it was processed by Maple.
- The new Maple state bar can be toggled by the approximate ≈ button between an exact mode and an approximate mode. The accuracy of the approximation can be set using the math engine console. Set the variable DIGITS as needed. Default value is 3.

Picture 1 shows the GUI of jaQuzzi Symbolic with the minor extensions.



Picture 1: jaQuzzi Symbolic GUI

Performance of the implementation:

As using unitary matrices is computationally unfeasible, a similar approach as used by Felix Schuermann was chosen. In particular this means that the simulation's states are encoded in Bra-Ket Notation. These Kets are stored in a LinkedHashSet to allow both fast access and iteration over the elements. I.e. as long as there are "few" Kets in a state we see a tremendous advantage in speed over using unitary matrices that are exponential in the number of qubits.

Implementation Details:

All matrices that were previously available in jaQuzzi are now available with full symbolic precision. In the following section you will find an overview, describing how elementary operations are implemented

$$Not := \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix} \qquad C-Not := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Picture 2: Not and Controlled-Not martrices

Not and C-Not:
Though possible Not and C-Not gates are not implemented as matrices, for performance reasons. This step requires no interaction with Maple, and is handled entirely in jaQuzzi. In pseudo notation we get the following code snippet:

```
for each Ket a in State do
        if (controllBitsSet(a))
                negateAffectedQubit(a)
        end if
loop
```

Table 1: Pseudo Code for (C)-Not operations

The routine's runtime depends on the number of kets in a state and may therefore be exponential in the number of qubits. But as we can assume that in a general stetting $|kets| = 2^{|qubits|}$ we have a significant speedup here.

Unitary matrices and C-Unitary matrices:

$$Hadamard := \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$
$$Phase(j) := \begin{pmatrix} e^{pij} & 0 \\ 0 & e^{pij} \end{pmatrix}$$

Picture 3: Examples of common unitary matrices

If a controlled action takes place we first check whether all necessary control qubits are unequal zero. If that is the case we take the affectedQubits and calculate their Kronecker Prodcut. Theses operations are all still performed within jaQuzzi still not using Maple. But now the multiplication vector times matrix takes place within Maple. Then again using Maple the result is simplified. We now decode the result Vector into the corresponding Kets. These Kets now have to be filled up with the qubits that were not used in the previous calculations. We also have to be careful that it may be necessary to merge several states into one.
Pseudocode:

```
for each Ket a in State do
        if (controllBitsSet(a))
                temp:= KronProd(affectedQubits(a))
                tempResult:= simplify(temp * Matrix);
                tempKet: = toKet(tempResult);
                result:= upSize(Ket);
        end if
loop
```

Table 2: Pseudo code for generic (C)-matrices

Again the runtime of the algorithm is in $O\left(2^{|Qubits|}\right)$, but in practical situations it should be a lot better.
As you should know these operations form a basis for all operations in qubit circuits. That is why the implementation of these operations was discussed in more detail. The original jaQuzzi has more built-in matrices. These were also reproduced in a similar fashion. If you are interested in details for other operations please have a look at the source code.

Classes in package maple:

Here I will give a short textual description of all the classes and methods in the jaQuzzi/maple package. If you want even further details you are encouraged to have a look at program's source code.

DynamicClassLoader:

This class is needed to workaround another Java problem. Before starting the application, the problem is we do not know where some of our resources are located, as the Maple directory is different from platform to platform. But setting a different classpath does not work for jar files, what is a real limitation in this scenario. Using the reflection API it is possible to bypass this limitation and load the resources dynamically.

InitMaple:

This class provides functionality to find out whether Maple is available and all required path variables are set correctly.

public static void loadMapleClasses(String mapleBinPath)
Loads the required Maple™ libraries jopenmaple.jar and externalcall.jar at runtime. The path mapleBinPath is used as a reference to determine the correct location of the files. If something goes wrong Maple™ is again not available.

public static boolean checkMaplePathSet()
This method tries to find out whether all necessary variables are set. There is a platform dependent distinction between the necessary path variables. For example on a Mac Os X™ system "MAPLE" and "DYLD_LIBRARY_PATH" have to be set to the correct locations.

public static String createBatchFiles(String mapleBinPath)
This method creates platform dependent shell scripts from the path specified in mapleBinPath. On windows this would be a batch file, on UNIX like system you have to mark the created scripts as executable. If you changed the name of jaQuzzi.jar to something else you should not forget to make the necessary changes to the shell scripts.

public static boolean writeMaplePath(String path)
Writes the path of Maple to the file settings.txt

public static String getMaplePath()
Reads the maplePath from settings.txt if present.

Ket:

This class holds a single ket consisting of a front factor encoded as a String and the qubit sequence stored in an integer array. The class provides additional functionality to create a *copy* of a state and a *toString()* method for an appropriate output. The toString() method is used for the formatting of the output to the screen.

KetUtilities:

This is the central class that provides most of the functionality needed for the manipulation of the kets. Therefore most member methods will be presented in detail.

public static LinkedHashSet algebraicToKet (Algebraic vector)
This method is used to transform a Maple Algebraic vector object into a LinkedHastSet. So this method creates a ket for every entry in the vector that is unequal 0. This is done by converting the position number of the entry in binary representation. Leading zeros have to be added until the length of the binary matches the number of qubits. As a result every one and zero can be identified with a qubit. All Kets are inserted into a LinkedHashSet and then returned.

public static String[][] applyErrorMatrix(String[][] matrix, Matrix errorMatrix)
This method applies an error matrix to a given 2x2 dimensional Matrix. Note that only 2x2 dimensional matrices are supported at this time. This is no newly introduced limitation, but was also present in the non symbolic version of jaQuzzi.

public static Algebraic applyMatrix(Algebraic alg, String[][] matrix)
This method applies a matrix on a Maple vector. The input matrix is converted in Maple matrix format first and then applied on the vector. This is done using a Maple. The result is of course a vector in Maple Algebraic format.

public static LinkedHashSet<Ket> applyMatrix(LinkedList<Integer> controllQubits, LinkedList<Integer> effectedQubits, LinkedHashSet<Ket> kets, String[][] matrix)
This method can be used in a general setting to apply a matrix on a given state in the form of a LinkedHashSet (kets). Additionally it is possible to specify the control qubits and the effected qubits. We iterate over all kets in our LinkedHashSet. First we check whether all control qubits are unequal zero. If that is the case we proceed by converting the effected qubits in vector representation. Now we use the applyMatrix(Algebraic alg, String[][] matrix) method on our vector. We convert the result back into kets and upsize them with the missing qubits that were not effected by the matrix operation. It is now possible that different kets are mapped onto the same ket by a matrix operation, therefore we have to check and in case merge the kets with the same qubit sequence. Finally we return the result in the form of a LinkedHastSet.

public static Ket mergeKets(Ket a, Ket b)
Merges two kets, that have the same qubitSequence by simply adding their front factors.

private static String makeComparable(int[] key)
Transforms an int[] in a String by concatenating all entries. This is necessary as int[] can't be compared via equals and Strings can.

public static LinkedHashSet<Ket> upSizeKet(LinkedList effectedQubits, Ket originalKet, LinkedHashSet ketsToUpsize)
This method is used to add the missing non affected qubits back into the ket. First, new kets with the original size are created. Then, iterating over the number of qubits we know from effectedQubits which elements we have to copy from the originalKet and which from ketsToUpsize.

public static String getMatrix(String[][] matrix, String frontFactor)
Constructs a Maple matrix String from a matrix in String[][] representation. To make things easier a common front factor may be specified.

public static Algebraic ketToStringArray(LinkedList effectedQubits, Ket ket)
This method is used to convert the effected qubits in a Maple algebraic vector. We iterate over all effectedQubits and construct an array using the Kronecker product of all effected qubits. This array is converted in correct Maple syntax and multiplied with the ket's front factor. Afterwards Maple is used to convert the result in a vector, which we return as the Operations result.

public static String toMapleString(String[] toConv)
Converts a String[] containing a vector into a correct Maple vector String.

public static String[][] kronProd(String[][] m1 , String[][] m2)
Implements the Kronecker product on two matrices m1 and m2. For further details about the Kronecker product have a look here.

public static String printState(LinkedHashSet<Ket> lhs)
Constructs a String representation of the current state. This is used in the GUI.

public static void getOperation(Gate container)
This is used for debugging only and prints some information about the current gate, state, matrix, etc. to the console.

public static String[][] getBuiltInMatrix(String identifier)
All built in matrices that can be used in jaQuzzi are stored here. Identifier is used to retrieve the correspond matrix. For example H is the Hadamard matrix.

public static LinkedHashSet<Ket> exchangeQubits(int a, int b, LinkedHashSet<Ket> state)
This method implements jaQuzzi's exchange Qubit operation. Again we iterate over every ket in the current state and swap qubits a and b.

public static boolean checkBitsSet(LinkedList<Integer> controllQubits, Ket ket)
This method is used to find wether all qubits that are used for a controlled operation are set. If one of the qubits specified in controllQubits is equal zero, the method returns false, true otherwise.

public static LinkedHashSet<Ket> cNot(LinkedList<Integer> controllQubits, int effectedQubit, LinkedHashSet<Ket> state)
This method implements a controlled not operation. For performance reasons no matrices are used. First it is cheched whether the controllQubits are set. If this is the case we simply negate the qubit specified by effectedQubit.

public static LinkedHashSet<Ket> cPhase(LinkedList<Integer> controllQubits, int effectedQubit, LinkedHashSet<Ket> state, String phase)
Performs a controlled phase shift. Again for performance reasons the operation was not implemented as a matrix. The concrete implementation is analogue to cNot.

public static OperationContainer parseOperation(Gate gate, Matrix errorMatrix)
This method is used to parse a gate and set all necessary paramters for the invocation of the corresponding functions. E.g. it sets the controllQubits, the effectedQubit and the state for cNot. The other operations are analogue.

public static LinkedHashSet<Ket> invokeOperation(OperationContainer con, LinkedHashSet<Ket> state)
Uses the OperationContainer returned by parseOperation to apply an operation i.e. matrix, cNot, cPhase, etc. on a state.

public static LinkedHashSet<Ket> fullMeasurement (LinkedHashSet<Ket> state)
This method performs a full measurement of a state. That means all entangled states will collapse. First we choose a random number between zero and one. If the sum of the probabilities for qubit i to be one is below the random number. The qubit i is set to 0 in all kets and vice versa. At the end of the process all probabilities have to be normalized again.

public static LinkedHashSet<Ket> partialMeasurement(int effectedQubit, LinkedHashSet<Ket> state)
This method is used to measure one qubit. Entanglement of the qubit will be destroyed if present. A partial measurement is implemented as a full measurement with only one qubit effected.

public static String[][] toStringMatrix(Matrix m)
Convertes a jaQuzzi matrix m in a String[][] matrix. This is necessary for interoperation with Maple.

Log class:

This class is used for debugging and contains some logging functionality. For example the current state can be written to a file. The default filename is state.txt. The current state is always appended at the end of the file.

MapleConnectorClass:

This class provides access to Maple™ and is used to initialize the connection to Maple™. Additionally it stores all Maple™ states.

public static void initializeMaple()
This method is used to setup the initial connection. If something goes wrong here all Maple functionality is deactivated and we are in the standard jaQuzzi mode prior to jaQuzzi symbolic.

OperationContainer:

This class is used as a wrapper for a gate operation. It stores the matrix representation of the operation, the affected qubits as well as the control qubits.