CMU Quantum Information Programs in Mathematica

(Version 050, Date: 3 January 2006)

Robert B. Griffiths © 2006

Contents

1 Introduction

This is a collection of Mathematica functions and other objects, such as lists representing quantum gates, intended to be useful in quantum information studies of small quantum circuits. Some prior knowledge of Mathematica is needed in order to make use of them, and the user will probably want to combine them with his own definitions in order to achieve something useful. An advantage of employing Mathematica is that one can carry out either symbolic or numerical calculations. Many, though not all, of the functions in the collection can be used for either purpose.

All the objects are contained in a single master file qinfvvv.ma, where vvv is a version number, and are documented in several ways. First, they are given one-line definitions in a single alphabetical list, and these are repeated in various categories, in the header of the master file. Brief descriptions accessible when running Mathematica are given in standard fashion (name::usage=) in the master file just ahead of the actual definition. In some cases explanatory notes having to do with the particular way a function has been defined have been inserted as a comment `(*functionname:...*)` between the brief description and the actual definition. In a few cases there are duplicate definitions, in which case the later one is that actually employed; by removing it or commenting it out the user can employ the earlier one. Some functions defined in an earlier version of qinf are given in qupdatevvv.ma.

Section 2 describes the way we represent quantum kets (vectors) and operators (matrices). Section 3 takes up tensor products and the different ways in which kets and matrices are represented as tensors. The Pauli representation of operators for one or several qubits is discussed in Sec. 4, along with Bell states for two qubits. A summary of how to convert objects between different representations is given in Sec. 5. A small set of kets, projectors, quantum gates, and quantum codes are included in the collection, as described in Sec. 6. Section 7 describes functions for generating random kets and unitary operators, and Sec. 8 a few miscellaneous functions which we have found helpful. Section 9 takes up probabilities and conditional states. A simple three-qubit example that illustrates how many of the features work will be found in Sec. 10. Contributors to the project are acknowledged in Sec. 11.

2 Standard (Matrix) Representation

2.1 Kets, operators, dot product

We use a basic or *standard* or *matrix* representation in which a column vector or *ket* on an N-dimensional Hilbert space is represented by a list, of complex numbers or symbolic variables, of length N, thus

$$|\mathtt{ket}\rangle \to \mathtt{ket\ =\ \{e1,e2,...,eN\}}. \qquad (2.1)$$

A *bra* or row vector is represented in exactly the same way, and converting a ket to the corresponding bra, or vice versa, is a matter of taking the complex conjugate, thus

$$\mathtt{bra\ =\ Conjugate[ket]\ =\ adjoint[ket]}, \qquad (2.2)$$

where an initial lower case letter indicates a function in our collection:

`adjoint[]` applied to a (possibly rectangular) matrix `mt` yields the complex conjugate of its transpose.

An operator on the Hilbert space corresponds to an $N \times N$ matrix, represented by a list of N lists, each of the latter being one row of the matrix, as in the following 2×2 example:

$$\begin{pmatrix} \langle 1|M|1\rangle & \langle 1|M|2\rangle \\ \langle 2|M|1\rangle & \langle 2|M|2\rangle \end{pmatrix} \rightarrow \{\{\langle 1|M|1\rangle\,, \langle 1|M|2\rangle\}\,, \{\langle 2|M|1\rangle\,, \langle 2|M|2\rangle\}\,\}. \tag{2.3}$$

Note that in Mathematica lists are indexed in such a way that the first element is 1, not 0, which is why the kets in (2.3) are $|1\rangle$ and $|2\rangle$, rather than the $|0\rangle$ and $|1\rangle$ which are nowadays customary for a two-dimensional (single qubit) Hilbert space. Using these conventions allows one to employ the Mathematica dot operation, so that

$$\langle \texttt{kt1}|\,\texttt{kt2}\rangle \rightarrow \texttt{adjoint[kt1].kt2}, \tag{2.4}$$

$$\langle \texttt{kt1}|\texttt{matrix}|\texttt{kt2}\rangle \rightarrow \texttt{adjoint[kt1].matrix.kt2}. \tag{2.5}$$

These can also be written using the function

`ketinner[kt1,kt2]`, the inner product of `kt1` and `kt2`, defined by the right side of (2.4).

Products of operators can also be taken care of using a dot. For example, if `unitar` is an $N \times N$ unitary matrix corresponding to time development from t_0 to t_1, then

$$\texttt{kt1} = \texttt{unitar.kt0}, \quad \texttt{rho1} = \texttt{unitar.rho0.adjoint[unitar]}, \tag{2.6}$$

where `kt0` and `kt1` are lists of length N representing the initial and final kets, whereas `rho0` and `rho1` are $N \times N$ matrices representing the initial and final density operators.

One of the less attractive features of Mathematica is its awkward method for handling symbolic complex variables, in contrast to numerical constants. In particular, complex conjugation attaches `Conjugate[]` to all sorts of symbols, and the only way of getting rid of it for symbols denoting real quantities is to use `ComplexExpand[]`, which has its own peculiarities. This is why our collection provides two alternatives to `adjoint[]`, which are sometimes, but not always, helpful in reducing the mess:

`adjointr[mt]` is the transpose of `mt` and thus its adjoint if `mt` is real.

`adjointc[mt]` is `adjoint[mt]` followed immediately by `ComplexExpand[]`.

It is often convenient to use normalized kets, and

`ketnorm[kt]` normalizes `kt` (assumed not to be zero) by multiplying it with a positive constant. Similarly,

`ketnormr[kt]` will produce a normalized version of `kt` when it is real.

In a similar way

`pop2dop[mt]` multiplies the operator `mt` (assumed to be positive semi-definite) with a positive number to produce a trace 1 density matrix.

The function

`dyad[kt1,kt2]` produces the operator $|\texttt{kt1}\rangle\langle\texttt{kt2}|$ as a matrix. It is not necessary that `kt1` and `kt2` have the same dimension; if they do not, the result is an appropriate $\texttt{dim1}\times\texttt{dim2}$ rectangular matrix. The alternative

`dyadr[kt1,kt2]` is useful for real kets containing symbolic elements if one wants to avoid `Conjugate[]`, whereas

`dyadc[kt1,kt2]` includes the complex conjugate, but immediately applies `ComplexExpand[]` in hopes of reducing complications at an early stage.

In the case of a normalized ket `kt` the corresponding projector is produced by

`dyap[kt]=dyad[kt,kt]`.

The matrix (Hilbert-Schmidt) inner product can be written as

$$\langle A, B\rangle := \mathrm{Tr}[A^\dagger B] \to \texttt{Tr[adjoint[mta].mtb]} = \texttt{matinner[mta,mtb]}; \tag{2.7}$$

`matinner[mta,mtb]` is the sum of the products of the corresponding matrix elements of `mta` and `mtb` when the former have been complex conjugated. This is faster for large matrices than calculating the matrix product `(mta . mtb)` and then taking the trace. Note that `mta` and `mtb` can be rectangular, provided they both have the same shape: the same number of rows and the same number of columns.

2.2 Orthonormal collections and bases

We follow the convention that a collection of m kets, each of dimension n is to be thought of as a list of m lists, each of length n, which one can think of as an $m \times n$ matrix. This agrees with the Mathematica convention in the function `Eigenvectors[]`, but has the awkward feature that while one normally thinks of kets as column vectors, they are here represented as row vectors. `Transpose[]` will convert a collection of row vectors to a collection of column vectors; the only danger is that one may forget to do this. When $m = n$ and the rows are normalized and mutually orthogonal, the collection of rows, i.e., the list of lists, forms an *orthonormal basis*, and when we refer to such a basis we shall think of the rows (rather than the columns) as the elements of the basis.

`coeffs[kt,abasis]` returns as a list the expansion coefficients $\{c_j\}$ in $|\texttt{kt}\rangle = \sum_j c_j|a_j\rangle$, where the orthonormal basis $\{|a_j\rangle\}$ is represented by the matrix `abasis`, whose first row corresponds to $|a_1\rangle$, second row to $|a_2\rangle$, etc.

Mathematica has a Gram-Schmidt orthogonalization procedure in the package LinearAlgebra`Orthogonalization`, which we have not succeeded in getting to work properly for the case at hand with its complex inner product. In this collection the task is carried out by

`grschm[ls]` produces from a list `ls` of m linearly-independent kets, each an n-component vector, an orthonormal set as a list of m lists of length n.

`grschmr[ls]` does the same thing as `grschm[ls]` when all the kets are real.

3 Tensor Products

3.1 Kets

On a tensor product $\mathcal{H}_1 \otimes \mathcal{H}_2 \otimes \cdots \mathcal{H}_n$ of Hilbert spaces with dimensions d_1, $d_2 \dots d_n$, a ket can be represented as an n-component tensor

$$\langle j_1, j_2, \dots j_n|\texttt{kt}\rangle \to \texttt{kt[[j1,j2, .. ,jn]]}, \tag{3.1}$$

where $j_m \to$ `jm` lies between 1 and d_m, and labels an orthonormal basis. In Mathematica such a tensor is a list of lists of lists, etc., with `jn` labeling elements in the innermost lists, and one references a particular element using the `[[...]]` notation indicated in (3.1). In addition to this *tensor* or *k-tensor representation*, the same ket can be expressed in the *standard representation* (Sec. 2) as a single vector or list of $d = d_1 d_2 \cdots d_n$ complex numbers, assuming an ordering in which the last index varies most rapidly. Thus for $n = 2$, j_1, j_2 precedes k_1, k_2 if $j_1 < k_1$, or if $j_1 = k_1$ and $j_2 < k_2$. Our collection contains the following functions to map kets from one representation to the other:

`ket2kten[kt,dl]` assumes that `kt` is a single list, and returns the corresponding tensor form as a nested list as determined by

`dl ={d1,d2,...,dn}`, a *dimension list*, with `dj` the dimension the Hilbert space $\mathcal{H}_j$. If all dimensions are 2 (one is dealing with qubits), one can omit the dimension list and use

`ket2kten2[kt]` assumes a tensor product of two-dimensional spaces, and converts `kt` to tensor form. The final 2 in a function name indicates it is designed for use with qubits. The function assumes, without checking, that the length of `kt` is a power of 2.

`kten2ket[kten]` is the same as `Flatten[]`, so requires no dimension list, and in such cases we do not define a separate function for qubits.

In the case of product spaces involving only qubits, a quite common notation for kets is exemplified by $|010\rangle$, meaning that the first and third qubits are in the $|0\rangle$ state and the second is in the $|1\rangle$ state. Let us call this the *binary* representation. The convention given above makes $|010\rangle$ correspond to a third rank tensor `kt` in which `kt[[1,2,1]]=1` and all other components are zero. (Note the necessity of adding a 1 to each bit in $|\cdots\rangle$, because in a Mathematica list the first element is 1, not 0.) The function

`bket[...,n]`, as in $|0010\rangle \rightarrow$ `bket[0010,4]`, where the second argument is the number of qubits, yields the corresponding ket in standard form; e.g., a list of length 16 if `n` is 4. One can multiply such kets by scalars, and when `n` is the same add them, as in

$$1.5|0101\rangle + (1.2+3)i|1100\rangle \rightarrow \texttt{1.5*bket[0101,4] + (1.2+3 I)*bket[1100,4]}. \tag{3.2}$$

An alternative is provided by

`bin2ket[ls]`, which takes a list `ls` of n 0's and 1's, and converts it into the corresponding 2^n-component ket:

$$1.5|0101\rangle \rightarrow \texttt{1.5*bin2ket[\{0,1,0,1\}]}. \tag{3.3}$$

Its inverse is the function

`ket2bin[kt]`, which converts a vector ket `kt` to a list of terms in binary form. If, for example, `kt` is the vector produced by the right side of (3.2), the output of `ket2bin[kt]` is

```
{{1.5, |0101>}, {1.2 + 3 I, |1100>}}.        (3.4)
```

While less legible than the right side of (3.2), this may still be preferable to the corresponding vector

```
{0, 0, 0, 0, 0, 1.5, 0, 0, 0, 0, 0, 0, 1.2 + 3 I, 0, 0, 0}        (3.5)
```

3.2 Operators, n-tensors and o-tensors

We employ two different tensor representations for operators, corresponding to the following two ways of writing an operator W acting on the tensor product of two spaces in Dirac notation:

$$W = \sum \langle j,k|W|j',k'\rangle \Big(|j,k\rangle\langle j',k'|\Big) = \sum \langle j,k|W|j',k'\rangle \Big(|j\rangle\langle j'| \otimes |k\rangle\langle k'|\Big), \tag{3.6}$$

We shall refer to them as the *n form* or *n-tensor* or *normal* representation, and the *o form* or *o-tensor* or *dyad* representation, respectively. One can think of "n" as standing for "normal" and "o" as the first letter in "otimes", the symbol ⊗. Thus we have—note once again that the possible values for `j`, etc., start at 1, not at 0—

$$\texttt{nW[[j,k,j',k']]} = \langle j,k|W|j',k'\rangle = \texttt{oW[[j,j',k,k']]}, \tag{3.7}$$

with an obvious generalization to a tensor product of 3 or more spaces. Both tensor representations of W are distinct from the standard or matrix representation introduced in Sec. 2.1. When thinking of $\langle j,k|W|j',k'\rangle$ as a matrix, we regard (j,k) as a double label for rows using the same ordering convention discussed above following (3.1): the order of the rows from top to bottom is $(1,1)$, $(1,2)$, $\ldots (2,1)$, $(2,2)$, $\ldots$. The same ordering applies to the double label (j',k') for the columns.

If instead of an operator on $\mathcal{H}_1 \otimes \mathcal{H}_2$, W is thought of as a linear map from $\mathcal{H}_1' \otimes \mathcal{H}_2'$ to $\mathcal{H}_1 \otimes \mathcal{H}_2$, its matrix may be rectangular and, indeed, the dimensions of all four spaces may be different. We allow for this possibility in functions which convert from one representation to another by generalizing the notion of a dimension list, Sec. 3.1, to

`ddl = {{d1a,d1b},{d2a,d2b}...}`: a *double dimension list*, where `d1a` and `d1b` are the dimensions of $\mathcal{H}_1$ and $\mathcal{H}_1'$, i.e., the number of rows and columns in a matrix representing a map from $\mathcal{H}_1'$ to $\mathcal{H}_1$. Where a double dimension list is expected, our functions will accept a single list or a mixed list, and convert it to the required double list, e.g., they will convert

$$\texttt{\{3,2,\{5,3\}\}} \rightarrow \texttt{\{\{3,3\},\{2,2\},\{5,3\}\}}. \tag{3.8}$$

(Be careful: an extra {} as in `{3,{2},{5,3}}` will produce an error!)

Here are functions for converting an operator from one representation to another. Note that a double dimension list `ddl`, which could also be a single list `dl` or a mixture as on the left side of (3.8), is required when converting a matrix to an n-tensor or o-tensor, because a given matrix might correspond to various different tensors, but is not needed for the other conversions. It can be omitted in the case of qubits (all factor spaces are 2-dimensional) by using the functions with names ending in 2. These functions do not check that the matrix has dimensions consistent with `ddl`, or is a $2^n \times 2^n$ matrix in the case of qubits.

`mat2nten[mt,ddl]` converts a (standard) matrix `mt` to an n-tensor using the (double) dimension list `ddl`.

`mat2nten2[mt]` assumes that `mt` is a $2^n \times 2^n$ matrix, and returns the corresponding n-tensor.

`mat2oten[mt,ddl]` converts a (standard) matrix `mt` to an o-tensor using the (double) dimension list `ddl`.

`mat2oten2[mt]` assumes that `mt` is a $2^n \times 2^n$ matrix, and converts it to an o-tensor.

`nten2mat[ntn]` converts the n-tensor `ntn` to a (standard) matrix.

`nten2oten[ntn]` converts the n-tensor `ntn` to the corresponding o-tensor.

`oten2mat[otn]` converts an o-tensor `otn` to a (standard) matrix

`oten2nten[otn]` converts an o-tensor `otn` to an n-tensor

3.3 Building up kets and operators

One often faces the problem of starting with kets or operators which refer to particular factor spaces of a tensor product and constructing their counterparts on the full tensor product space. There are various functions which assist in carrying out this task. One of the most useful is

`outer[tn1,tn2,...]=Outer[Times,tn1,tn2,...]`, which gives the tensor product of an arbitrary number of tensors: if they are one-dimensional vectors or kets, the result is a k-tensor in the notation of Sec. 3.1; if they are (possibly rectangular) matrices, the result is an o-tensor in the notation of Sec. 3.2. Based on this are:

`ketprod[kt1,kt2,...]` produces the ket (as a vector, not a tensor!) $|\texttt{kt1}\rangle \otimes |\texttt{kt2}\rangle \otimes \cdots$ on the space $\mathcal{H}_1 \otimes \mathcal{H}_2 \cdots$. In the case of qubits it may sometimes be simpler to use the functions `bket[]` or `bin2ket[]` described above in Sec. 3.1.

`tenprod[mt1,mt2,...]` generates the (standard representation) matrix corresponding to $\texttt{mt1} \otimes \texttt{mt2} \otimes \cdots$, where the matrices `mt1`, `mt2`,... can be rectangular and of different sizes.

Another task is best illustrated by means of an example. Suppose one is constructing unitary operators for a collection of three qubits, and wants to apply a particular one-qubit gate W, represented by a 2×2 unitary matrix `ww`, to the third qubit. The 8×8 matrix representing $I \otimes I \otimes W$ is produced by the function

`expandout[ww,{3},{2,2,2}]`, where the first argument is the 2×2 matrix `ww` representing the one-qubit operator, the second—note curly brackets indicating a one-member list—indicates it is to be applied to the third factor in the tensor product, and `{2,2,2}` is the associated dimension list. Changing `{3}` to `{1}` would result in an 8×8 matrix representing $W \otimes I \otimes I$. As we are dealing with qubits, the same matrix is produced by

`expandout2[ww,{3},3]`, where the number of qubits 3 replaces the dimension list as the third argument.

Next, suppose we want the 8×8 matrix representing a controlled-not gate acting between the third qubit (as control) and the second qubit (as data). Included in our collection is `cnot`, a 4×4 matrix representing the controlled-not operation between two qubits, with the first the control and the second the data qubit. The desired 8×8 matrix is generated by either of the functions

`expandout[cnot,{3,2},{2,2,2}]`,

`expandout2[cnot,{3,2},3]`,

where the second argument, `{3,2}`, is a list that tells `expandout[]` or `expandout2[]` which qubits the gate is to act on, and which of these is the control. Replacing `{3,2}` with `{2,3}` would make qubit 2 the control and qubit 3 the data. The function `expandout[]`, but not `expandout2[]`, can be used for tensor products of spaces with dimensions larger than 2 by using a suitable dimension list `dl` as the final argument.

There are special functions for producing tensor products of Pauli matrices (including the identity):

`sigl[ls]` is the tensor product of the Pauli operators in the list `ls`; e.g., `sigl[{1,0,3}]` is $\sigma_x \otimes I \otimes \sigma_z$.

`sigprod[j,k,...]` yields $\sigma_j \otimes \sigma_k \otimes \cdots$.

Finally, the function

`copygate[W,n]`, returns, as a matrix in standard form, the tensor product of a matrix `W` with itself `n` times: $W \otimes W \otimes \cdots$.

3.4 Permuting the order of factors

Sometimes one has a tensor product $\mathcal{A} \otimes \mathcal{B} \otimes \mathcal{C}$ and wants to convert expressions to a different order of the factors, say $\mathcal{C} \otimes \mathcal{A} \otimes \mathcal{B}$. If one is using the tensor forms, this can be done using `Transpose[]`, with its second argument a suitable permutation. Our collection includes functions that carry out the appropriate transformations on kets and square matrices in the standard representation:

permket[kt,pm,dl] converts kt, defined on a tensor product with dimension list dl, to the form appropriate for a new ordering of the factors determined by the permutation pm. Suppose, for example, that kt is in standard form, a list of 24 complex numbers, for the tensor product $\mathcal{A}\otimes\mathcal{B}\otimes\mathcal{C}$, where $d_a = 2$, $d_b = 3$, and $d_c = 4$, corresponding to a dimension list dl = {2,3,4}. To convert this to the corresponding ket in standard form for $\mathcal{C}\otimes\mathcal{A}\otimes\mathcal{B}$, the appropriate permutation is pm = {2,3,1}, interpreted as $1 \to 2$, $2 \to 3$, $3 \to 1$. Think of it as "1 ($\mathcal{A}$) moves to (position) 2, 2 ($\mathcal{B}$) moves to (position) 3, and 3 ($\mathcal{C}$) moves to (position) 1." After the conversion the old dimension list dl is no longer valid, and for further operations it needs to be replaced by

dlp = permute[dl,pm]. (For the present example dlp = {4,2,3}.) In the same way

permmat[mt,pm,dl] converts the matrix mt of an operator on a tensor product with dimensions given by the list dl to that appropriate when the factors have been rearranged according to the permutation pm. Both mt and the new matrix represent an operator in standard form. See the example in the previous paragraph for the meaning of pm, and note that the rearrangement corresponds, in general, to a new dimension list dlp = permute[dl,pm].

3.5 Expanding a ket in an orthonormal basis

Let $A = \{|a_j\rangle\}$ be an orthonormal basis of an n-dimensional Hilbert space $\mathcal{A}$, which is the first factor in a tensor product $\mathcal{H} = \mathcal{A}\otimes\mathcal{B}$. Any ket $|\psi\rangle$ on $\mathcal{H}$ can be written in the form

$$|\psi\rangle = \sum_j |a_j\rangle \otimes |\beta_j\rangle, \tag{3.9}$$

where the expansion coefficients $\{|\beta_j\rangle\}$ are kets on $\mathcal{B}$ which, in general, are neither normalized nor mutually orthogonal. The function

ketcofs[psi,abasis,dl] returns the coefficients $\{|\beta_j\rangle\}$ as a list $\{|\beta_1\rangle, |\beta_2\rangle, \ldots\}$. Here abasis is the orthonormal basis for $\mathcal{A}$, a matrix (list of lists) in which the first row (list) represents the first basis ket, etc. (see Sec. 2.2), and each $|\beta_j\rangle$ is itself a list of d_b elements, where d_b is the dimension of $\mathcal{B}$, and dl = {da,db} is the dimension list for the spaces $\mathcal{A}$ and $\mathcal{B}$.

If $|\psi\rangle$ is a ket in a tensor product $\mathcal{H} = \mathcal{A}\otimes\mathcal{B}\otimes\mathcal{C}$ of three spaces, one can again employ ketcofs[psi,abasis,dl] with dimension list dl = {da,db,dc}, to provide the expansion coefficients, which are now kets (in standard form) on $\mathcal{B}\otimes\mathcal{C}$, for $|\psi\rangle$ in terms of the orthonormal basis abasis of $\mathcal{A}$. Since both $|\psi\rangle$ and the expansion coefficients are in standard form (single lists, not tensors), the tensor product structure of $\mathcal{B}\otimes\mathcal{C}$ plays no role. The same procedure works for a tensor product of $\mathcal{A}$ with three or more spaces.

There is no special function for giving the coefficients, as kets on $\mathcal{A}$, when $|\psi\rangle$ on $\mathcal{A}\otimes\mathcal{B}$ is expanded in the orthonormal basis $B = \{|b_j\rangle\}$ of $\mathcal{B}$. To do this it is necessary to first apply permket[], Sec. 3.4, to psi in order to interchange the order of $\mathcal{A}$ and $\mathcal{B}$, and then use ketcofs[] with the $\mathcal{B}$ basis and the dimension list {db,da}. Similar procedures can be used for tensor products of three or more spaces.

3.6 Schmidt representation; entanglement

The Schmidt representation

$$|\psi\rangle = \sum_j \lambda_j |a_j\rangle \otimes |b_j\rangle \tag{3.10}$$

is a special case of the expansion (3.9) in which both the $\{|a_j\rangle\}$ and the $\{|b_j\rangle\}$ are orthonormal bases, and the *Schmidt coefficients* $\{\lambda_j\}$ are nonnegative numbers. The function

`schmidt[kt,dl]` finds the Schmidt decomposition of $|\texttt{kt}\rangle$, and returns a list

$$\{\{\lambda_1, |a_1\rangle, |b_1\rangle\},\ \{\lambda_2, |a_2\rangle, |b_2\rangle\}, \ldots\} \tag{3.11}$$

in which the $|a_j\rangle$ and $|b_j\rangle$ are themselves lists. If Schmidt coefficients are smaller than some specified number (10^{-8} in the function as written; the user can, of course, change that) they are simply discarded along with the associated $|a_j\rangle$ and $|b_j\rangle$. Thus while $\{|a_j\rangle\}$ and $\{|b_j\rangle\}$, are orthonormal collections of the same length, they need not actually be bases of $\mathcal{A}$ and $\mathcal{B}$. The function `schmidt[kt,dl]` only works when `kt` is numerical, and will fail if it contains symbolic elements.

Other functions in this category include:

`schmidt2ket[ls]` returns the original `kt` when given a list `ls` produced by `schmidt[]`; i.e., it is the inverse of `schmidt[]`.

`schmidtprobs[kt,dl]` returns as a list the *Schmidt probabilities* $\{p_1 = \lambda_1^2, p_2 = \lambda_2^2, \ldots\}$, but does not supply the kets of the two orthonormal bases. For this reason it is faster than `schmidt[]`. If `kt` is not normalized the function will again return the $\{\lambda_j^2\}$, but they can no longer be interpreted as probabilities.

`schmidtproj[ls]` returns the projector $\sum_j |a_j\rangle\langle a_j| \otimes |b_j\rangle\langle b_j|$ when given a list `ls` of the form produced by `schmidt[]`.

The *entanglement* of a normalized ket $|\psi\rangle$ on $\mathcal{A}\otimes\mathcal{B}$ is defined as the Shannon entropy of the probability distribution $\{\lambda_1^2, \lambda_2^2, \ldots\}$ generated by the Schmidt coefficients in (3.10). The function

`entang[kt,dl]`, with `dl = {da,db}` the appropriate dimension list, will first normalize `kt` and then calculate the entanglement using logarithms to base 2. Again, this function as written will only work when `kt` is a list of numbers, not symbols.

A Renyí entanglement equal to minus the logarithm to base 2 of the trace of the square of the reduced density operator on one subsystem is computed by

`entsq[kt,dl]`, with `dl = {da,db}` the dimension list. This function first normalizes the ket, and will only work when the ket is numerical, not symbolic.

3.7 Partial trace and partial transpose

The Mathematica `Tr[]` yields the trace of a square matrix. When using a tensor product, one often wants to find the *partial trace* of an operator over one or more of the factor spaces. To this end we provide two functions:

`partrace[mt,q,dl]` takes a matrix `mt` on a tensor product $\mathcal{H}_1\otimes\mathcal{H}_2\cdots\mathcal{H}_n$, with dimensions `dl = {d1,d2,...dn}`, and returns the corresponding matrix with the space $\mathcal{H}_\texttt{q}$ traced out. For example, if `dl = {2,3}` and `q=2`, `mt` must be a 6×6 matrix, and the output will be a 2×2 matrix. (Note that both input and output are in standard form.)

`traceout[mt,ls,dl]` is similar to `partrace`, except that now `ls` is a *list* of spaces to be traced out. E.g., `traceout[mt,{2,4},{2,2,2,2}]` starts with a 16×16 matrix `mt` on a system of four qubits, traces out qubits 2 and 4, and returns a 4×4 matrix representing an operator on qubits 1 and 3. (While the same thing could be done by repeatedly applying `partrace`, the need to keep changing the dimension list would make it confusing.)

Sometimes it is useful to calculate the *partial transpose* of a matrix representing an operator. On $\mathcal{A}\otimes\mathcal{B}$ with orthonormal bases $\{|a_j\rangle\}$ and $\{|b_k\rangle\}$, the partial transpose P^{TA} relative to $\{|a_j\rangle\}$

of an operator P is defined by

$$\langle a_j, b_k|P^{TA}|a_{j'}, b_{k'}\rangle = \langle a_{j'}, b_k|P|a_j, b_{k'}\rangle. \tag{3.12}$$

Note that the partial transpose of an operator defined in this way depends upon the choice of basis $\{|a_j\rangle\}$ (but not $\{|b_k\rangle\}$) relative to which it is taken. To carry out such an operation, and its generalization to the tensor product of three or more spaces, the collection contains the function:

`partrans[mt,q,dl]` takes a matrix `mt` (in standard form, a list of lists) on the tensor product of spaces described by the dimension list `dl`, and returns its partial transpose (again in standard form), relative to the standard basis—i.e., the basis used to define the matrix—on factor space `q`. E.g., for the case in (3.12) one would use `partrans[mt,1,{da,db}]`.

4 Pauli Representation and Bell States

4.1 Pauli representation

In the case of qubits (2-dimensional spaces) and tensor products of qubits it is often convenient to use the *Pauli representation* for operators. Thus for an operator V on the space of one qubit one writes

$$V = \sum_{j=0}^{3} v_j\sigma_j \rightarrow \texttt{Sum[v[[j+1]]*sig[j],\{j,0,3\}]}, \tag{4.1}$$

where we use the convention that $\sigma_0 = I$ is the identity, and the σ_j for $j = 1, 2, 3$ are σ_x, σ_y and σ_z, respectively. In our collection the Pauli matrices the σ_j form an array:

`sig[j]`, with `j` taking on the values 0, 1, 2, and 3. E.g., `sig[3] = {{1,0},{0,-1}}`. Using an array rather than a list is convenient if one wants `j` to start with 0, but leads to the awkward appearance of both `j` and `j+1` in (4.1). On a tensor product of two qubits the corresponding expression is

$$W = \sum_{j=0}^{3}\sum_{k=0}^{3} w_{jk}\sigma_j \otimes \sigma_k \rightarrow$$
$$\texttt{Sum[w[[j+1,k+1]]*tenprod[sig[j],sig[k]],\{j,0,3\},\{k,0,3\}]}. \tag{4.2}$$

We use the term *Pauli coefficient tensor*, or simply *Pauli tensor*, abbreviated as `ptn`, when referring to `v[[ ]]`, `w[[ ]]`, and the like. The function

`mat2paul[mt]` generates the Pauli (coefficient) tensor from the matrix `mt` in the standard representation, assuming (without checking!) that `mt` is $2^n \times 2^n$ for some integer n, while

`paul2mat[ptn]` carries out the reverse process on the Pauli tensor `ptn`.

One can generate a Pauli tensor directly using

`paulten[j,k,...]`, where, for example, `paulten[1,0,3]` is the tensor corresponding to $\sigma_x \otimes I \otimes \sigma_z$. For the corresponding matrix, use `paul2mat[ paulten[...] ]`. One can multiply by scalars and add, thus

```
ptn = 4 paulten[0,0,3] + (5+2 I) mu paulten[2,1,3] - 6 nu paulten[2,3,1]    (4.3)
```

corresponds to

$$4\big(I \otimes I \otimes \sigma_z\big) + (5 + 2i)\mu\big(\sigma_y \otimes \sigma_x \otimes \sigma_z\big) - 6\nu\big(\sigma_y \otimes \sigma_z \otimes \sigma_x\big). \tag{4.4}$$

Interpreting the Pauli tensor written in the standard Mathematica format as a list of lists of lists, etc., can be rather daunting, so the following output function can be helpful:

`prtpaul[ptn]` uses `Print[]` to display the nonzero elements of the Pauli tensor `ptn` in the following form when `ptn` has been defined by (4.3):

$$\texttt{c[0,0,3]= 4 c[2,1,3]= (5 + 2 I) mu c[2,3,1]= -6 nu} \tag{4.5}$$

Thus `c[0,0,3]` is the coefficient of $I \otimes I \otimes \sigma_z$, and so forth. While not as legible as (4.4), it is at least a step in the right direction in comparison to examining `ptn` as a list of lists of lists.

Eliminating zero elements is a helpful feature of `prtpaul[]`, but if the tensor contains floating point numbers the "chop" version may be more useful:

`prtpaulch[ptn,ep]` uses `Chop[]` to eliminate numbers of absolute value less than `ep`, where if the second argument is not indicated, `prtpaulch[ptn]` will use a default value of 10^{-10}, which the user can alter by modifying the helper function `paulnzch[]`.

To print out the Pauli form of an operator on a set of n qubits in the form indicated in (4.5) it is first necessary to convert it to a Pauli tensor using `mat2paul[]` before applying `prtpaul[]`. The user can, of course, define a single function combining both, but we have found that when dealing with symbolic expressions it is often, though not always, useful to apply `Simplify[]` at the intermediate stage.

4.2 Bell states

For two qubits one sometimes makes use of a basis of fully-entangled *Bell states* on the four-dimensional product space. They can be defined and labeled in various different ways, and our collection includes three possibilities, in each case given as an array of four kets labeled `bell[j]` for $0 \leq$ `j` ≤ 3 (or `sbell[j]` for the special Bell basis). Conversions between the standard basis ($|00\rangle$, etc.) and the Bell basis can be carried out using two matrices, `bellbas` and its adjoint `basbell`, in the following manner,

$$\texttt{ktbell = basbell.kt}, \quad \texttt{kt = bellbas.ktbell}, \tag{4.6}$$

where `kt` and `ktbell` are the coefficient lists for the same ket in the standard and Bell representations, respectively. The conversion of matrices can be done using the dot product or by employing the corresponding functions,

$$\texttt{mtbell = mat2bell[mt] = basbell.mt.bellbas} \tag{4.7}$$

$$\texttt{mt = bell2mat[mtbell] = bellbas.mtbell.basbell} \tag{4.8}$$

Here `mt` and `mtbell` refer to matrices of the same operator in the standard and Bell bases, respectively.

In the function list two versions of the Bell basis are given. The second is the one which will be used if the file is read in as is, but the user can delete it or comment it out or replace it with his own definitions. In addition a "special Bell basis" (sometimes referred to as a "magic Bell basis") called `sbell[j]` is given, along with matrices `bassbell`, `sbellbas` and functions `mat2sbell[]`, `sbell2mat[]`, that are the obvious counterparts to those discussed above.

5 Transforming Between Representations

We have introduced four representations for operators: standard (matrix), n-tensor, o-tensor, and, in the case qubits, the Pauli representation. Kets have only a standard (vector) representation and a single tensor (k-tensor) representation. Here is a summary of the functions in our collection which transform operators and kets from one representation to the other.

5.1 Kets

To convert a ket `kt` to the tensor (k-tensor) form, which can be considered either an o-tensor or an n-tensor, use

`ket2kten[kt,dl]`, for the general case with a dimension list `dl`, or

`ket2kten2[kt]` for qubits.

The inverse transformation

`kten2ket[ktn]` is a fancy name for `Flatten[ktn]`, and also works for qubits.

In the case of qubits, note the additional functions

`bket[...]` creates kets (vectors or lists) corresponding to $|001\rangle$ =`bket[001,3]` and the like, while

`bin2ket[ls]` does the same thing for a list `ls` of 0's and 1's; $|001\rangle$ =`bin2ket[{0,0,1}]`. The inverse function

`ket2bin[kt]` takes a vector `kt` and produces a form involving coefficients of $|001\rangle$ and the like as in (3.4).

5.2 Operators: standard (matrix), o-tensor, and n-tensor forms

See Sec. 3.2 for definitions, and note that one can, in the "square" case, use a single dimension list `dl` in place of the double list `ddl`, or a mixed list as in (3.8).

Matrix to n-tensor or o-tensor:

`mat2nten[mt,ddl]` for (double) dimension list `ddl`.

`mat2nten2[mt]` for qubits; no dimension list needed.

`mat2oten[mt,ddl]` for (double) dimension list `ddl`.

`mat2oten2[mt]` for qubits; no dimension list needed.

Tensors to matrices:

`nten2mat[ntn]`

`oten2mat[otn]`

One kind of tensor to another:

`nten2oten[ntn]`

`oten2nten[otn]`

5.3 Pauli representation

The following functions only apply to tensor products of qubits; for more details, see Sec. 4.1.

`oten2paul[otn]` yields the Pauli tensor corresponding to the o-tensor `otn`.

`paul2oten[ptn]` converts the Pauli tensor `ptn` to an o-tensor.

These and the matrix-to-o-tensor conversions mentioned above are combined in the functions:

`mat2paul[mt]` converts the matrix `mt` to a Pauli tensor, and

`paul2mat[ptn]`, which carries out the inverse operation.

6 Kets, Projectors, Gates, Codes

6.1 Kets and projectors

The following kets for qubits are provided as two-element arrays, with `j = 0` corresponding to the positive and `j = 1` to the negative intersection of the axis with the Bloch sphere

`xket[j]` for `j=0,1`

`yket[j]` for `j=0,1`

`zket[j]` for `j=0,1`: `zket[0]={1,0}`, `zket[1]={0,1}`,

A normalized ket corresponding to the point `r={x,y,z}` on the Bloch sphere, $\texttt{x}^2\texttt{+y}^2\texttt{+z}^2\texttt{=1}$, is produced by

`blochket[{x,y,z}]`.

Users who dislike our choice of phases should substitute their own.

The projectors

`xprj[j]=dyap[ xket[j] ]` for `j=0` and `j=1`

`yprj[j]=dyap[ yket[j] ]` for `j=0` and `j=1`

`zprj[j]=dyap[ zket[j] ]` for `j=0` and `j=1`

are 2×2 matrices which are, of course, independent of the phase conventions used for the kets. For example, `zprj[1]={{0,0},{0,1}}`.

6.2 Gates

The following one qubit gates are 2×2 unitary matrices. To use them for circuits of several qubits, one must first apply `expandout[]` or `expandout2[]` as discussed in Sec. 3.3.

`hgate` is the Hadamard gate: `{{1,1},{1,-1}}/Sqrt[2]`

`xgate, ygate, zgate` are the same as σ_x, σ_y and σ_z, identical to `sig[1]`, `sig[2]`, `sig[3]`.

`rgate[j,th]` has arguments `j`=1, 2, or 3, for `x`, `y` and `z`, and `th` an angle in radians. For example, `rgate[2,Pi/2]` is the same as $\exp[-i(\pi/4)\sigma_y]$, a rotation of the Bloch sphere by an angle of $\pi/2$ about the y axis.

The following two-qubit gates are included:

`cnot` is a controlled-not gate, with the first qubit the control.

`cphase` is a controlled-phase gate for two qubits.

`exchg` exchanges two qubits,

For a space of arbitrary dimension:

`fourierg[n]` is an $n \times n$ matrix representing the quantum Fourier transform; to be specific, the j, k element is $e^{2\pi i(j-1)(k-1)/n}/\sqrt{n}$ if j and k take values between 1 and n. On the other hand

`fouriern[kt]`, the same as `Fourier[kt]`, can be used to evaluate the Fourier transform applied to the ket `kt` when it is a vector of complex numbers, and will be faster than first computing `qft=fourierg[n]` and then `qft . kt`.

`ident[n]`, the same as `IdentityMatrix[n]`, is the $n \times n$ identity matrix.

In addition, the following special constructions are provided.

`copygate[W,n]`, returns, as a matrix in standard form, the tensor product of a matrix `W` with itself `n` times: $W \otimes W \otimes \cdots$.

`cgate[W]` will produce a *controlled*-`W` gate: a matrix (in standard form) on $\mathcal{A} \otimes \mathcal{B}$, with $\mathcal{A}$ the controlling qubit, and $\mathcal{B}$ the d-dimensional space on which W acts. The resulting operator is

$$|0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes W. \tag{6.1}$$

For example, `cgate[xgate]` is the controlled-not gate, while `cgate[cnot]` is the 8×8 matrix for a Tofolli gate with the third qubit controlled by the first two.

6.3 Quantum codes

The simplest quantum codes are two-dimensional (one qubit) subspaces on tensor products of n qubits. We provide a pair of orthonormal basis vectors (as a list) for four well-known examples:

`threecode`, $n = 3$
`fivecode`, $n = 5$
`sevencode`, $n = 7$
`ninecode`, $n = 9$

The first is the three-qubit code that corrects bit-flip errors, while each of the others can correct any error on just one of the n qubits carrying the code. See Ch.10 of M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, (Cambridge University Press, 2000) for details.

7 Random Operations

The functions listed below were designed to provide random kets and orthonormal collections of kets. They employ `Random[]` and and `RandomArray[]` for a `NormalDistribution[]`. If one wants to get the same "random" sequence every time one runs a particular program, one should insert `SeedRandom[int]`, with a particular choice of the integer `int`, before the first call to one of the following functions. The distributions generated by the functions given below are invariant under unitary transformations of the Hilbert space.

`ranbas[n]` returns a random basis for a complex Hilbert space of dimension `n` as a list of basis vectors, each basis vector a list of `n` complex numbers.

`ranket[n]` returns a normalized random ket on a complex Hilbert space of dimension `n`, a single list of `n` complex numbers.

`ranorn[m,n]` produces a random orthonormal collection of `m` (complex) kets on an `n`-dimensional complex Hilbert space as a list of `m` elements, each a list of `n` complex numbers (thus as an `m`×`n` matrix). For `m=1` the result is the same as `ranket[n]`, and for `m=n` it is the same as `ranbas[n]`.

On a *real* Hilbert space of dimension `n` the counterparts of the functions just mentioned are

`ranbasr[n]`
`ranketr[n]`
`ranornr[m,n]`

For two qubits,

`ranbell` will generate a random orthonormal basis of fully-entangled states as a list of four kets.

8 Miscellaneous Functions

A few additional functions not discussed earlier, which have turned out to be of some use, are the following:

`diags[mat]` returns the diagonal elements of the square matrix `mat` as a list; it is the inverse of `DiagonalMatrix[]`.

`entropy[ls]` gives the Shannon entropy H (log base 2) of a list `ls` of probabilities.

`invperm[pm]` returns the inverse permutation to `pm`

`matinp[mta,mtb]` finds the trace of `mta.mtb` without evaluating the full matrix product, so is faster than `Tr[mta.mtb]`; it is similar to `matinner[]` defined in Sec. 2.1.

`matinq[mta,mtb]` is the sum over j and k of `mta[[j,k]]*mtb[[j,k]]` whenever this makes sense; in particular, it is $\mathrm{Tr}(A^{\mathrm{T}}B)$ for (possibly rectangular) matrices A and B. Can also be used when `mtb` is a tensor of rank greater than 2.

`matnorm[mt]` normalizes each row of the matrix `mt`.

`outer[tn1,tn2,...]`= `Outer[Times,tn1,...]` is the outer product of `tn1`, `tn2`. . . .

`permute[ls,pm]` applies permutation `pm` to list `ls`; thus
`permute[{a,b,c},{2,3,1}] = {c,a,b}`.

`permutmat[pm]` is the matrix for permutation `pm`; thus
`permutmat[{2,3,1}].{a,b,c} = {c,a,b}`.

`prodlist[ls]` is the product of the elements in the list `ls`.

`quadn[ob]` is the quadratic norm, the sum of the absolute squares of all of the elements, of `ob`, which can be a ket, a matrix, or a tensor of any rank,

`quadr[ob]` is the sum of the squares of the elements of `ob`, thus the same as `quadn[]` when these elements are real.

`sumlist[ls]` gives the sum of the elements in the list `ls`.

`transpose[]` is the same as `Transpose[]`, except that when given the (simple) list corresponding to a ket (vector), it returns the list rather than an error message.

The collection also includes functions which are helper functions called by other functions; these are listed in the header of the master file.

9 Probabilities and Conditional States

There are no functions in the collection designed specifically for calculating probabilities of measurement outcomes, or ("collapsed") quantum states conditional on measurement outcomes. These can be computed in the manner indicated below.

9.1 Born probabilities

The Born rule expresses the probability at time t_1 for a quantum property represented by a projector P=`proj` as

$$\Pr(P) = \langle\psi_1|P|\psi_1\rangle = \texttt{ketinner[kt1,pproj.kt1]}, \tag{9.1}$$

where the normalized $|\psi_1\rangle$ = `kt1` is obtained from an earlier initial state $|\psi_0\rangle$ = `kt0` by unitary time evolution, as in (2.6). In the case of a density operator the corresponding expression is

$$\Pr(P) = \mathrm{Tr}(\rho_1 P) = \texttt{Tr[rho1.pproj]} = \texttt{matinner[rho1,pproj]}, \tag{9.2}$$

where ρ_1 has evolved from from an initial state ρ_0, again see (2.6). (Using `matinner[]` rather than `Tr[]` speeds things up for large matrices.) If the projector $P = |\phi\rangle\langle\phi|$ corresponds to the normalized ket $|\phi\rangle$ =`ktp`, the probability $|\langle\phi|\psi_1\rangle|^2$ or $\langle\phi|\rho_1|\phi\rangle$ can be calculated using

$$\Pr(P) = \texttt{Abs[ketinner[ktp,kt1]]\^{}2} \quad\text{or}\quad \Pr(P) = \texttt{adjoint[ktp].rho1.ktp}. \tag{9.3}$$

The analogs of (9.1) or (9.2) can be used for POVM's by replacing P with the appropriate positive operator(s). We provide no special functions for constructing positive operators or projectors, aside from the small set for qubits described in Sec. 6.1. These can be extended to systems of multiple qubits using the functions as described in Sec. 3.3.

9.2 Conditional states

If the measurement of some qubit yields the result $|0\rangle$, what *conditional* quantum state is to be assigned to the remainder? If the system is described at the time t_1 of a measurement on the first qubit by a pure state $|\psi_1\rangle$, the answer is obtained by expanding this ket in the form (3.9) with $|a_0\rangle = |0\rangle$, $|a_1\rangle = |1\rangle$, using `ketcofs[]` (see the discussion in Sec. 3.5) and using the appropriate $|\beta_j\rangle$, which would be $|\beta_0\rangle$ in case the measurement outcome corresponds to $|0\rangle$. Note that $|\beta_j\rangle$ is (in general) not normalized; $\langle\beta_j|\beta_j\rangle =$ `ketinner[betaj,betaj]` is the probability for measurement outcome j, while the normalized state conditioned on this outcome will be `ketnorm[betaj]`. To apply this scheme to measurements on something other than the first factor in a tensor product, it is necessary to first carry out a permutation using `permket`, as explained in Sec. 3.4.

For a state described by a density operator ρ_1 at t_1 a different approach is required. Again suppose that the first qubit is measured in the standard basis. The (unnormalized) conditional state is obtained by first multiplying ρ_1 by the appropriate projector corresponding to the measurement outcome, and then taking a *partial trace* of this product over the first qubit using `partrace[]`. The positive operator resulting from this process when normalized by dividing it by its trace (which is what `pop2dop[]` does) is the desired conditional density operator. The same procedure works if one is interested in a measurement on the second or any other qubit: one only has to construct the appropriate projector P, and then take the appropriate partial trace of $P\rho_1$. See Sec. 10 for an example. And one can do the same thing for a pure state $|\psi_1\rangle$ by first forming $\rho_1 = |\psi_1\rangle\langle\psi_1|$ using `dyad` and then multiplying by P, etc. The end result will, of course, be a (pure state) density operator rather than a ket, as in the method described previously.

10 A Simple Example

The following example illustrates how the functions in this collection can be applied to a simple quantum circuit composed of three qubits, Fig 1, in which an H gate (Hadamard) is applied to the first qubit, an X gate (bit flip) to the second qubit, followed by a controlled-not from 1 (control) to 3, a second controlled-not from 3 (control) to 2, and finally a measurement on qubit 2 in the standard or computational basis.

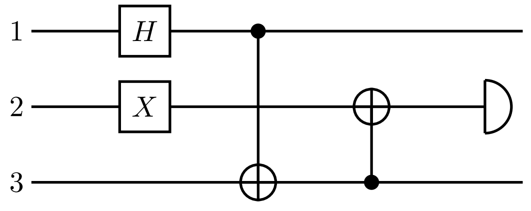

Figure 1: Quantum circuit with three qubits.

Let us construct the unitary transformation corresponding to the part of the circuit preceding the measurement. The Hilbert space is 8-dimensional and the dimension list is `dl = {2,2,2}`. The unitary transformation has the form

$$\texttt{unitar = cnot32 . cnot13 . xgt2 . had1,} \tag{10.1}$$

in a fairly obvious notation, with each of the three operators in the product on the right represented by an 8×8 unitary matrix. Note that the order of the operators in (10.1) follows the usual convention, with the right most operator applied first, and is thus written in the reverse order to the circuit in Fig. 1, where we follow the usual convention of time increasing from left to right. The X gate `xgt2` could be placed to the right of `had1` or to the left of `cnot13` in (10.1), but does not commute with `cnot32`.

The Hadamard and the X gate are constructed using either `expandout[]` or `expandout2[]`

$$\texttt{had1} = \texttt{expandout[hgate,\{1\},\{2,2,2\}]} = \texttt{expandout2[hgate,\{1\},3]}, \tag{10.2}$$

$$\texttt{xgt2} = \texttt{expandout[xgate,\{2\},\{2,2,2\}]} = \texttt{expandout2[xgate,\{2\},3]}. \tag{10.3}$$

For the controlled-not gates we give only the `expandout2` form:

$$\texttt{cnot13} = \texttt{expandout2[cnot,\{1,3\},3]}, \tag{10.4}$$

$$\texttt{cnot32} = \texttt{expandout2[cnot,\{3,2\},3]}. \tag{10.5}$$

Note that `cnot` is already defined as a 4×4 matrix, and the order of integers in the list forming the second argument of `expandout[]` is used to specify which qubit in the circuit corresponds to the control and which to the data.

Once the matrix `unitar`, (10.1), is available, it can be used for various calculations. For example, suppose that initially qubit 1 is in the state $(|0\rangle + 2|1\rangle)/\sqrt{5}$, and 2 and 3 are in $|0\rangle$ and $|1\rangle$, respectively, so the 8-dimensional ket for the total system is

$$\texttt{kt0 = (bket[001,3] + 2*bket[101,3])/Sqrt[5],} \tag{10.6}$$

which could also be written as `ketprod[ {1,2},bket[01,2] ]/Sqrt[5]`. Then

$$\texttt{kt1 = unitar . kt0} \tag{10.7}$$

is the state just before the measurement. To calculate the probability p_1 of a measurement outcome corresponding to the second qubit being in the state $|1\rangle$, first construct the projector

$$\texttt{proj = expandout2[zprj[1],\{2\},3]} \tag{10.8}$$

and then evaluate

$$p_1 = \texttt{ketinner[kt1 , proj.kt1]}. \tag{10.9}$$

Next consider a mixed initial state corresponding to $|1\rangle$ for qubit 1, $(|0\rangle + |1\rangle)/\sqrt{2}$ for qubit 2, and qubit 3 in the maximally-mixed state $I/2$. The initial density operator is

$$\texttt{rho0 = tenprod[zprj[1], xprj[0], sig[0]/2],} \tag{10.10}$$

and unitary time development leads to

$$\texttt{rho1 = unitar . rho0 . adjoint[unitar].} \tag{10.11}$$

In place of (10.9) the probability that qubit 2 is in $|1\rangle$ is now given by

$$q_1 = \texttt{Tr[rho1 . proj] = matinner[rho1,proj]}. \tag{10.12}$$

The conditional state of qubits 1 and 3 given this measurement outcome is the 4×4 matrix

$$\texttt{rhoc = pop2dop[partrace2[rho1.proj,2]]}, \tag{10.13}$$

where `pop2dop[]` normalizes the outcome of the partial trace in order to produce a density operator. The nonzero terms in its Pauli representation can be displayed using

$$\texttt{prtpaulch[mat2paul[rhoc]]}. \tag{10.14}$$

11 Acknowledgments

The following individuals have made significant contributions to producing the functions and other objects, writing the descriptions, and making the collection accessible on the Internet: Christopher Brust, Michael Crouch, Samuel Pinansky, Adam Steele, Yuqing Sun. Financial support for this effort has come from Carnegie-Mellon University and from the Physics Division of the National Science Foundation.