# User's Manual for *qinf* Quantum Information and Entanglement Package v. 0.10 For The *Maxima* Computer Algebra System

G. John Lapeyre, Jr.

September 1, 2008



## Contents

# 1 Introduction

This quantum information package for the *Maxima* computer algebra system allows the manipulation of instances of objects— operators, vectors, tensors, *etc.* that appear in the theory of quantum information and of quantum entanglement. More precisely these objects are typically represented in this package as row and column vectors and matrices, whose entries may be explicit numbers (of various classes) or algebraic expressions. This software occupies a niche distinct from high performance numerical linear algebra software as well as software such as the *Maxima* tensor packages that manipulate abstract mathematical objects. This document describes the functions and data in the package and how to use them with *Maxima* , assuming that you do not know much about *Maxima* , but do know quantum information theory. However, most of the examples are also found in introductory texts on quantum information. The package is intended for research and teaching in the theory of entanglement and quantum information and related fields.

Examples of the facilities of the package are

- Methods for constructing pure and mixed states and operators.
- Methods for executing standard operations found in computational linear algebra as well as the tensor product, partial trace, etc.
- Functions to compute commonly appearing quantities such as entropy and purity.

This document begins with a very brief introduction to *Maxima* emphasizing features that are important for *qinf*. The remainder of the manual is a series of small sections introducing functions with examples. The examples mostly consist of testing equations. First identities and textbook exercises are presented, in part because they are the natural calculations to include in the test suite from which they are taken. Then more complicated calculations are tested, as that is the initial application of the author— to check manual calculations and results found in other documents. In this restricted sense, the package can give results on simple abstract statements: If 1) generic instances of objects are generated, and 2) a representation-invariant statement is formulated, and 3) the subexpressions are successfully coerced into some canonical form, then defects in the statement can sometimes be detected if the statement is not true.

Some suggestions and things to be aware of in the following sections.

- You probably need to read at least a ten minute *Maxima* tutorial before or in conjunction with reading this document. There are several listed at the *Maxima* website, and others that a search engine can find. If you are too impatient there is a very brief introduction to *Maxima* below. The best source for most questions is the *Maxima* manual.
- There are several user interfaces to *Maxima* . All the examples here are generated using the *imaxima* package for the *emacs* editor/environment, but the results are similar to other graphical frontends to *Maxima* . Other popular graphical frontends are *wxmaxima*, and *texmacs*.
- Most functions currently work only with qubits, others for variable number of states.

### 1.1 Acknowledgments

Some of the ideas used in *qinf* are inspired by the package *qdensity* at `http://www.pitt.edu/~tabakin/QDENSITY/` written for a proprietary symbolic algebra system. None of the code in qdensity has been borrowed for this project, however. Advice on *Maxima* and *lisp* programming was provided by, among others, Robert Dodier, Stavros Macrakis, and Barton Willis.

## 2 Using *Maxima*. features important for *qinf* .

The package is loaded using the load† function like this

```
(%i1) load("qinf.mac");

(%o1)                                qinf.mac
```

There are several tutorials and manuals available for *Maxima*. Here is a very brief one focused on aiding the introduction to the *qinf* package. We will not give examples of matrices until later, but point out that the notation for matrix multiplication in *Maxima* is a dot , eg. `A . B`. If $A$ is a $m \times n$ matix and $B$ a $p \times q$ matrix, then `A . B` is a $n \times p$ matrix. The inner product of quantum state vectors, the outer product (dyad) of quantum state vectors, the composition of operators, and the mapping of one vector to another by an operator are all special cases of matrix multiplication and are all represented by the dot (along with conjugation in the case of the inner and outer products.) The remaining product, the tensor product, becomes the Kronecker product in the matrix representation of a finite dimensional Hilbert space. To agree with standard terminology, we introduce the infix operator **otimes** and the function **tensor_product** that eventually call the *Maxima* function **kronecker_product** . See the section on matrices in the *Maxima* manual.

*Maxima* can use exact real and complex numbers or the standard floating point approximations, or arbitrary precision floating point numbers. Numerical expressions are simplified upon entry. Each input line must be terminated by a semicolon (some interfaces do this automatically) or by a dollar sign, which suppresses the output.

```
(%i1) 1 + 1;
```

(%o1) 2

Assignment is denoted by a colon while function definitions are denoted by := For example, `a : b+c ;` evaluates `b+c` and assigns the result to `a`. On the other hand `a(x,y) := x^y ;` defines the function $a(x, y)$.

```
(%i2) a : 2 * 2;
```

(%o2) 4

```
(%i3) a;
```

(%o3) 4

```
(%i4) b : expand( (x+y)^4 );
```

(%o4) $y^4 + 4\,x\,y^3 + 6\,x^2\,y^2 + 4\,x^3\,y + x^4$

We suppress the output here with a dollar sign because it's big– 51 terms.

```
(%i5) b : expand( (x+y)^50 )$
(%i6) length(b);
```

(%o6) 51

Some exact numbers and floating point approximations.

```
(%i7)  1  + sqrt(2);
```

(%o7) $\sqrt{2}+1$

```
(%i8)  1  + sqrt(2), float;
```

(%o8) 2.4142135623730949

Defining and using a function.

```
(%i9) f(x) := 3 * cos(x);
```

(%o9) $f(x) := 3\cos x$

```
(%i10) f(a);
```

(%o10) $3\cos 4$

```
(%i11) f(0);
```

(%o11) 3

Complex numbers. %i is the identifier for $i = \sqrt{-1}$.

```
(%i12) expand ( (1 + 2 * %i)^2 );
```

(%o12) $4\,i - 3$

Some special numbers are defined, such as %pi and %e.

```
(%i13) cos(%pi/2);
```

(%o13) 0

```
(%i14) %e^(%i * %pi/2);
```

(%o14) $i$

# 3 Representation of states and operators

Kets are represented by $n \times 1$ matrices, bras by $1 \times n$ matrices. The objects are represented in the computational basis. Bras and kets representing the same states are related by *Maxima*'s conjugate transpose function **ctranspose** Density operators and other operators are represented by matrices. The tensor product is represented by the Kronecker product. There is no strong typing. You are responsible for knowing that a particular vector represents a state vector in a particular space. Below, we often substitute "the operator" for "the matrix representing the operator".

# 4 Creating instances of states

Here are some methods for creating instances of states, from scratch or from other states. Although all operators 'create' states in this sense, we omit most of them here, because they are better described as manipulating states.

## 4.1 ketz, braz,ketx, brax, kety, bray — multipartite eigenstate kets and bras of $\sigma_x, \sigma_y, \sigma_z$

create normalized $n$-partite product states in the computational basis. In all cases the indices are 0 or 1. The pair **ketz**$(i_1, \ldots, i_n)$ and **braz**$(i_1, \ldots, i_n)$ produce eigenstates of $\sigma_z^{(1)} \otimes \cdots \otimes \sigma_z^{(n)}$, with the index $i = 0$ selecting the state with eigenvalue 1 and $i = 1$ selecting the state with eigenvalue $-1$. In other words the ket produced represents $|i_1, i_2, \ldots, i_n\rangle$.

```
(%i3) ketz(1)
```

(%o3)
$$\begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

```
(%i4) braz(1)
```

(%o4)
$$\begin{pmatrix} 0 & 1 \end{pmatrix}$$

```
(%i5) braz(0)
```

(%o5)
$$\begin{pmatrix} 1 & 0 \end{pmatrix}$$

Here are the states $\langle 00|, \langle 11|$, and $\alpha_0\langle 00| + \alpha_1\langle 11|$. (we show bras rather than kets to conserve space)

```
(%i6) braz(0,0)
```

(%o6) $\begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}$

```
(%i7) braz(1,1)
```

(%o7) $\begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$

```
(%i8) alpha[1]*braz(1,1)+alpha[0]*braz(0,0)
```

(%o8) $\begin{pmatrix} \alpha_0 & 0 & 0 & \alpha_1 \end{pmatrix}$

The functions **ketx** , **brax** , **kety** , **bray** produce eigenstates of $\sigma_x^{(1)} \otimes \cdots \otimes \sigma_x^{(n)}$, or $\sigma_y^{(1)} \otimes \cdots \otimes \sigma_y^{(n)}$, with, as before, the index $i = 0$ selecting the state with eigenvalue 1 and $i = 1$ selecting the state with eigenvalue $-1$.

```
(%i9) brax(1)
```

(%o9) $\begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$

```
(%i10) bray(1,0,1)
```

(%o10) $\begin{pmatrix} \frac{1}{2\sqrt{2}} & \frac{i}{2\sqrt{2}} & -\frac{i}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{i}{2\sqrt{2}} & -\frac{1}{2\sqrt{2}} & \frac{1}{2\sqrt{2}} & \frac{i}{2\sqrt{2}} \end{pmatrix}$

## 4.2 ket_n, bra_n — alternate form of ketz, etc.

, **ket_n**$(j, i_1, \ldots, i_m)$ and **bra_n**$(j, i_1, \ldots, i_m)$ are an alternate way to call **ketx**, **kety**, etc. The index $j \in (1, 2, 3)$ is mapped to $(x, y, z)$ and the appropriate function, eg. **ketx** is called with the remaining arguments.

## 4.3 proj — Density matrix representation of a pure state (projection operator).

The projection operator (or equivalently, the density matrix) corresponding to a state vector is generated via the outer product, which is represented by the dot operator. A convenience function **proj**(*ket*) is also provided to form a projection operator. The argument *ket* can be either a bra or a ket (ie column or row vector). ( **proj** does not check that *ket* is normalized.) Below, we use the *Maxima* function **ctranspose** for the complex transpose. Here is the outer product, or dyad $|0\rangle\langle 0|$.

```
(%i19) ketx(1) . brax(1);
```

(%o19) $$\begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

Compare this to the inner product

```
(%i20) brax(1) . ketx(1);
```

(%o20) 1

Here are different ways to make $|000\rangle\langle 000|$ and $|101\rangle\langle 101|$.

```
(%i21) is ( ketz(0,0,0) . braz(0,0,0) =  ketz(0,0,0) . ctranspose(ketz(0,0,0)) );
```

(%o21) **true**

```
(%i22) is ( ketz(1,0,1) . braz(1,0,1) =  proj(ketz(1,0,1)) );
```

(%o22) **true**

This example used *Maxima*'s **is**(*expr*)† function which tries to determine whether the predicate *expr* is true or false.

There is also a function **tovect** that is the inverse (up to a phase) of **proj** — it returns the ket corresponding to a projection operator. If the input matrix is not a projection operator, the result is undefined.

```
(%i17) is ( tovect( proj(schmidt_ket(alpha))) = schmidt_ket(alpha));
```

(%o17) **true**

## 4.4 otimes, tensor_product, tensor_power — Creating state vectors with the tensor product.

The function **tensor_product**$(v_1, \ldots, v_n)$, returns $v_1 \otimes v_2 \cdots \otimes v_n$, where $v_i$ are vectors or matrices. The **otimes** operator is an 'infix' operator that is equivalent to the function **tensor_product**. The function **tensor_power**$(op, n)$, returns the $n$th tensor power of the operator `op`. Once again, we will employ the **is**† function. Keep in mind that, in this example, the expressions are not analyzed abstractly, but rather vectors with integer elements are generated and compared elementwise.

```
(%i12) is (ketz(0,1) = ketz(0) otimes ketz(1))

(%o12)                                          true

(%i13) is (ketz(0,1) = tensor_product(ketz(0),ketz(1)))

(%o13)                                          true

(%i14) is (ketx(0,1,0) otimes kety(1,0,1)
           = tensor_product(ketx(0),ketx(1),ketx(0),kety(1),kety(0),kety(1)))

(%o14)                                          true
```

### 4.5 schmidt_ket — bipartite ket in Schmidt basis.

**schmidt_ket**$(a)$creates a ket in the schmidt form. This is equivalent to `sqrt(a)*ket(0,0)+ sqrt(1-a)*ket(1,1)`. This only works for qubits ($d = 2$). Note that you may need to enter `assume(a>0,1-a>0)` when manipulating this state The **assume**$^\dagger$function is used to build a database of facts used, for instance, by the **is** function.

### 4.6 bell and belln — Bell state kets in computational basis.

create vector bell states. **bell[a,b]** creates the state

$$|\Psi_{a,b}\rangle = \frac{1}{\sqrt{2}}|0,b\rangle + (-1)^a|1,\bar{b}\rangle, \tag{1}$$

where $a, b \in \{0, 1\}$. The array **belln[i]** creates the same states where $i$ is the decimal representation of the binary numeration $(a, b)$. That is, $(0, 1, 2, 3)$ corresponds to $((0, 0), (0, 1), (1, 0), (1, 1))$. Note that **belln[i]** is a *Maxima* array as indicated by the square brackets.

As an exercise, we will check our definitions of the Bell states by testing for orthonormality. We first define an array function that returns the inner product of two Bell states. An array function **f[x,y]** is like an ordinary function **f(x,y)** except that it can be used where an array is expected.

```
(%i2) f[x,y] := belln[x] . belln[y];

(%o2)                                   f_{x,y} := belln_x · belln_y
```

Create a $4 \times 4$ matrix with **genmatrix**$^\dagger$ which maps the two dimension array **f** over the indices of the matrix with the given range.

```
(%i3) genmatrix( f , 3,3,0,0);
```

(%o3)
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

But instead of the named function **f** we could have used just a function body with the **lambda**[†] function, which returns a function that is not bound to a symbol.

```
(%i4) genmatrix( lambda( [x,y], belln[x] . belln[y]) , 3,3,0,0);
```

(%o4)
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

It is obviously the $4 \times 4$ identity matrix. The function **identitymatrixp**(*mat*) is a predicate defined in the quantum information package in analogy to the *Maxima* function **zeromatrixp** . It returns **true** only if its argument is an identity matrix. (The symbol % refers to the previous output.

```
(%i5) identitymatrixp(%);
```

(%o5) **true**

In the following sections, we often perform these comparisons in a single line. This is how the test appears in the regression test suite.

```
(%i6) identitymatrixp(genmatrix( lambda( [x,y], belln[x] . belln[y]) , 3,3,0,0));
```

(%o6) **true**

We see that these four vectors are orthonormal and thus form a basis in $\mathbb{C}^2 \otimes \mathbb{C}^2$. We can also check that

(2)
$$|\Psi_{00}\rangle\langle\Psi_{00}| + |\Psi_{01}\rangle\langle\Psi_{01}| + |\Psi_{10}\rangle\langle\Psi_{10}| + |\Psi_{11}\rangle\langle\Psi_{11}| \ = \mathbb{1}_4.$$

```
(%i2) identitymatrixp(apply("+",map(lambda([i],proj(belln[i])),[0,1,2,3])));
```

(%o2) **true**

### 4.7 ghz and ghzn — GHZ state kets.

The array **ghz**$[i, j, k]$ contains the GHZ kets defined by

$$|\Psi_{\mathrm{GHZ}}(a, b, c)\rangle \ = \frac{1}{\sqrt{2}}|0, b, c\rangle + (-1)^a|1, \bar{b}, \bar{c}\rangle, \tag{3}$$

where the bar denotes the logical not operation. The array **ghzn**$[n]$ is the same array indexed by a single decimal number equivalent to the binary numeration given by $a, b, c$.

### 4.8 werner — Werner state density matrix.

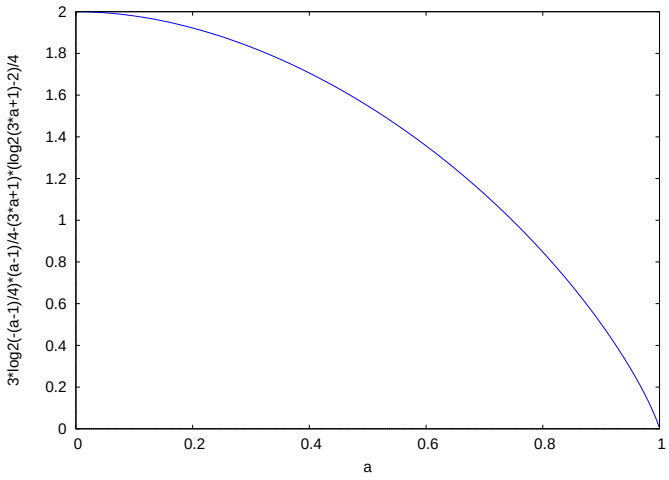
The Werner state is defined by

$$|\Psi(\lambda, i, j)\rangle_{\mathrm{Werner}} \ = \lambda|\Psi(i, j)\rangle_{\mathrm{Bell}}\langle\Psi(i, j)|_{\mathrm{Bell}} + (1 - \lambda)\rho_u,$$

where $\rho_u = \mathbb{1}_4/4$. The *qinf* function is `werner(a,i,j)`. The entropy depends on neither $i$ nor $j$ and varies with $\lambda$ between a value of 0 and 1 per bit, as the state is tuned from a pure state to the uniform mixed state.

```
(%i2) wxplot2d(entropy(werner(a,1,0)), [a,0,1]);
```

(%o2)




# 5 Creating and using operators

## 5.1 Pauli matrices

### 5.1.1 pauli — pauli matrices.

**pauli**$[i]$ creates the pauli matrices $(\sigma_0, \sigma_1, \sigma_2, \sigma_3) = (\mathbb{1}_2, \sigma_x, \sigma_y, \sigma_z)$.

```
(%i12) [ pauli[0], pauli[1], pauli[2], pauli[3] ];
```

(%o12) $$\left[\begin{pmatrix}1&0\\0&1\end{pmatrix},\begin{pmatrix}0&1\\1&0\end{pmatrix},\begin{pmatrix}0&-i\\i&0\end{pmatrix},\begin{pmatrix}1&0\\0&-1\end{pmatrix}\right]$$

Show that the ket $|1\rangle_x$ is an eigenvector of $\sigma_x$ with eigenvalue $-1$.

```
(%i8) is (  pauli[1] . ket_n(1,1) = -1 * ket_n(1,1) );
```

(%o8) **true**

Here are we check that all our definitions of the pauli matrices and kets are consistent in this sense.

```
(%i9) mapapply( lambda([i,j], is (pauli[i] . ket_n(i,j) = (-1)^j * ket_n(i,j))),
         [[1,0],[1,1],[2,0],[2,1],[3,0],[3,1]  ]);
```

(%o9) [**true**, **true**, **true**, **true**, **true**, **true**]

Here we use **anticommutator**$(op_1, op_2)$ to test the anticommutation relations among the pauli matrices: $\{\sigma_i, \sigma_j\} = 2\delta_{i,j}$ for $i, j \in \{1, 2, 3\}$.

```
(%i3) genmatrix(lambda([i,j],  anticommutator(pauli[i],pauli[j])/2 ), 3,3,1,1);
```

(%o3) $$\begin{pmatrix}\begin{pmatrix}1&0\\0&1\end{pmatrix}&\begin{pmatrix}0&0\\0&0\end{pmatrix}&\begin{pmatrix}0&0\\0&0\end{pmatrix}\\\begin{pmatrix}0&0\\0&0\end{pmatrix}&\begin{pmatrix}1&0\\0&1\end{pmatrix}&\begin{pmatrix}0&0\\0&0\end{pmatrix}\\\begin{pmatrix}0&0\\0&0\end{pmatrix}&\begin{pmatrix}0&0\\0&0\end{pmatrix}&\begin{pmatrix}1&0\\0&1\end{pmatrix}\end{pmatrix}$$

The *Maxima* function **mat_unblocker** , flattens the blocks in the above expression, so we can write

```
(%i4) identitymatrixp( mat_unblocker (genmatrix(lambda([i,j],
         anticommutator(pauli[i],pauli[j])/2 ), 3,3,1,1)));
```

(%o4) **true**

Now we load the **itensor** package, which provides the levi-civita tensor, and make use of the *Maxima* functions **permutations** (which returns the set of all permutations of a list) and **listify** (which converts the set data type into the (ordered) list data type). The *qinf* package provides **mapapply**$(func, [list1, list2, \ldots])$, which **apply** s *func* to each of the *list*s and returns a list of the results. (see the *Maxima* documentation for **apply**.) With all these, we can

test the commutation relations of the pauli matrices. (In reality, the matrix definitions are not complicated, we are actually testing the other functions.): $[\sigma_i, \sigma_j] = 2i\epsilon_{i,j,k}\sigma_k$.

```
(%i5) load("itensor");

(%o5)                    /usr/share/maxima/5.15.0/share/tensor/itensor.lisp

(%i6) mapapply(lambda([i,j,k],zeromatrixp(commutator(pauli[i],pauli[j])
     - 2*%i*levi_civita([i,j,k])*pauli[k])), listify(permutations([1,2,3])));

(%o6)                                [true, true, true, true, true, true]
```

#### 5.1.2 pauli_product — tensor product of pauli matrices.

**pauli_product**$(i_1, \ldots, i_n)$ returns the tensor product $\sigma_{i_1} \otimes \cdots \otimes \sigma_{i_n}$, where the indices $i_j$ are in $\{0, 1, 2, 3\}$. As elsewhere in this document, $\sigma_0$ is the $2 \times 2$ identity matrix.

The next three functions **pauliexp**$(\rho)$, **invpauliexp**$(c)$, and **correlation_tensor**$(c, i_1, \ldots, i_n)$ are related. An example using them follows their definitions.

#### 5.1.3 pauliexp — expansion of operator in terms of tensor products of pauli matrices.

**pauliexp**$(\rho)$ returns the correlation tensor, that is, the coefficients in the expansion of the matrix $\rho$ in tensor products of pauli matrices. Explicitly, **pauliexp** returns the coefficients $c_{i_1,\ldots,i_n}$ in

$$\rho = \sum_{i_1,\ldots,i_n=0}^{3} c_{i_1,\ldots,i_n} \; \sigma_{i_1} \otimes \cdots \otimes \sigma_{i_n}. \tag{4}$$

$\rho$ must be a $2^n \times 2^n$ matrix. The coefficients are returned as a list of $2^n \times 2^n$ elements. The place of $c_{i_1,\ldots,i_n}$ in the returned list is determined by taking $i_1, \ldots, i_n$ to be the binary representation of an integer. For convenience, the coefficient can be retrieved by index with the function **correlation_tensor**.

#### 5.1.4 invpauliexp — inverse of expansion in terms of tensor products of pauli matrices.

**invpauliexp**$(c)$ is the inverse of **pauiexp**. Given a list $c$ representing the correlation tensor (*i.e.* expansion coefficients), **invpauliexp** returns the matrix $\rho$ given by (4).

#### 5.1.5 correlation_tensor — retrieve component of correlation tensor by index.

**correlation_tensor**$(c, i_1, \ldots, i_n)$ returns the expansion coefficient for the term $\sigma_{i_1} \otimes \cdots \otimes \sigma_{i_n}$ in the expansion of $\rho$, where $c$ is the list of coefficients in the expansion of $\rho$ as given, for instance, by **pauliexp**.

#### 5.1.6 Using pauliexp and invpauliexp; an example.

Here is an example using the three functions defined above. First we create three generic $2 \times 2$ (complex) matrices.

```
(%i2) m1 : matrix([a1,b1],[c1,d1]);
```

(%o2)
$$\begin{pmatrix} \mathrm{a1} & \mathrm{b1} \\ \mathrm{c1} & \mathrm{d1} \end{pmatrix}$$

```
(%i3) m2 : matrix([a2,b2],[c2,d2]) $
(%i4) m3 : matrix([a3,b3],[c3,d3]) $
```

Here is the tensor product of the three matrices. This is *not* a generic element in the three qubit Hilbert space represented by $M(\mathbb{C}, 8)$. For instance, the three matrices have 12 complex parameters while a generic matrix in the tensor product space has 64 complex parameters.

```
(%i5) mp : m1 otimes m2 otimes  m3 ;
```

(%o5)
$$\begin{pmatrix}
\mathrm{a1\,a2\,a3} & \mathrm{a1\,a2\,b3} & \mathrm{a1\,a3\,b2} & \mathrm{a1\,b2\,b3} & \mathrm{a2\,a3\,b1} & \mathrm{a2\,b1\,b3} & \mathrm{a3\,b1\,b2} & \mathrm{b1\,b2\,b3} \\
\mathrm{a1\,a2\,c3} & \mathrm{a1\,a2\,d3} & \mathrm{a1\,b2\,c3} & \mathrm{a1\,b2\,d3} & \mathrm{a2\,b1\,c3} & \mathrm{a2\,b1\,d3} & \mathrm{b1\,b2\,c3} & \mathrm{b1\,b2\,d3} \\
\mathrm{a1\,a3\,c2} & \mathrm{a1\,b3\,c2} & \mathrm{a1\,a3\,d2} & \mathrm{a1\,b3\,d2} & \mathrm{a3\,b1\,c2} & \mathrm{b1\,b3\,c2} & \mathrm{a3\,b1\,d2} & \mathrm{b1\,b3\,d2} \\
\mathrm{a1\,c2\,c3} & \mathrm{a1\,c2\,d3} & \mathrm{a1\,c3\,d2} & \mathrm{a1\,d2\,d3} & \mathrm{b1\,c2\,c3} & \mathrm{b1\,c2\,d3} & \mathrm{b1\,c3\,d2} & \mathrm{b1\,d2\,d3} \\
\mathrm{a2\,a3\,c1} & \mathrm{a2\,b3\,c1} & \mathrm{a3\,b2\,c1} & \mathrm{b2\,b3\,c1} & \mathrm{a2\,a3\,d1} & \mathrm{a2\,b3\,d1} & \mathrm{a3\,b2\,d1} & \mathrm{b2\,b3\,d1} \\
\mathrm{a2\,c1\,c3} & \mathrm{a2\,c1\,d3} & \mathrm{b2\,c1\,c3} & \mathrm{b2\,c1\,d3} & \mathrm{a2\,c3\,d1} & \mathrm{a2\,d1\,d3} & \mathrm{b2\,c3\,d1} & \mathrm{b2\,d1\,d3} \\
\mathrm{a3\,c1\,c2} & \mathrm{b3\,c1\,c2} & \mathrm{a3\,c1\,d2} & \mathrm{b3\,c1\,d2} & \mathrm{a3\,c2\,d1} & \mathrm{b3\,c2\,d1} & \mathrm{a3\,d1\,d2} & \mathrm{b3\,d1\,d2} \\
\mathrm{c1\,c2\,c3} & \mathrm{c1\,c2\,d3} & \mathrm{c1\,c3\,d2} & \mathrm{c1\,d2\,d3} & \mathrm{c2\,c3\,d1} & \mathrm{c2\,d1\,d3} & \mathrm{c3\,d1\,d2} & \mathrm{d1\,d2\,d3}
\end{pmatrix}$$

We compute the correlation tensor of `mp`

```
(%i6) pe : pauliexp(mp) $
```

Check that the tensor has 64 elements and see what a coefficient looks like.

```
(%i7) length(pe);
```

(%o7) 64

```
(%i8) part(pe,10);
```

(%o8)
$$\frac{-i\,\mathrm{c1\,c2\,d3} - i\,\mathrm{b1\,c2\,d3} + i\,\mathrm{b2\,c1\,d3} + i\,\mathrm{b1\,b2\,d3} - i\,\mathrm{a3\,c1\,c2} - i\,\mathrm{a3\,b1\,c2} + i\,\mathrm{a3\,b2\,c1} + i\,\mathrm{a3\,b1\,b2}}{8}$$

Check that the inverse of the expansion gives the original matrix back

```
(%i9) is ( ratsimp( invpauliexp( pauliexp(mp) )) = mp);
```

(%o9) **true**

Here is the convenience function to return an element of the correlation tensor by index

```
(%i10) correlation_tensor(pe,1,2,3);
```

$$(\%o10) \qquad \frac{i\,\mathrm{c1}\,\mathrm{c2}\,\mathrm{d3}+i\,\mathrm{b1}\,\mathrm{c2}\,\mathrm{d3}-i\,\mathrm{b2}\,\mathrm{c1}\,\mathrm{d3}-i\,\mathrm{b1}\,\mathrm{b2}\,\mathrm{d3}-i\,\mathrm{a3}\,\mathrm{c1}\,\mathrm{c2}-i\,\mathrm{a3}\,\mathrm{b1}\,\mathrm{c2}+i\,\mathrm{a3}\,\mathrm{b2}\,\mathrm{c1}+i\,\mathrm{a3}\,\mathrm{b1}\,\mathrm{b2}}{8}$$

## 5.2 spinor_rotation, spinor_rotation_trig.

**spinor_rotation**$(phi, theta, gamma)$ returns the matrix that represents the operator that rotates a spinor through an angle *gamma* about the axis specified by *phi* (angle about the $z$-axis) and *theta* (inclination from the $z$-axis). The function **spinor_rotation_trig**$(phi, theta, gamma)$ returns the same matrix expressed only with cosines and sines. This is the standard axis-angle parameterization. Explicitly the matrices are

```
(%i2) spinor_rotation(phi,theta,gamma);
```

$$(\%o2) \qquad \begin{pmatrix} \cos\left(\frac{\gamma}{2}\right) - i\,\cos\vartheta\,\sin\left(\frac{\gamma}{2}\right) & -i\,e^{-i\,\varphi}\,\sin\vartheta\,\sin\left(\frac{\gamma}{2}\right) \\ -i\,e^{i\,\varphi}\,\sin\vartheta\,\sin\left(\frac{\gamma}{2}\right) & i\,\cos\vartheta\,\sin\left(\frac{\gamma}{2}\right)+\cos\left(\frac{\gamma}{2}\right) \end{pmatrix}$$

```
(%i3) spinor_rotation_trig(phi,theta,gamma);
```

$$(\%o3) \qquad \begin{pmatrix} \cos\left(\frac{\gamma}{2}\right) - i\,\cos\vartheta\,\sin\left(\frac{\gamma}{2}\right) & (-\sin\varphi\,\sin\vartheta - i\,\cos\varphi\,\sin\vartheta)\,\sin\left(\frac{\gamma}{2}\right) \\ (\sin\varphi\,\sin\vartheta - i\,\cos\varphi\,\sin\vartheta)\,\sin\left(\frac{\gamma}{2}\right) & i\,\cos\vartheta\,\sin\left(\frac{\gamma}{2}\right)+\cos\left(\frac{\gamma}{2}\right) \end{pmatrix}.$$

## 5.3 insert_operator — insert single qubit operators in $n$-qubit operator

**insert_operator**$(nbits, [op1, i1, i2, ..], [op2, j2, j2, ...], ...)$ returns the operator $\mathbb{1}_2^{\otimes \mathrm{nbits}}$, with some of the identity operators $\mathbb{1}_2$ substituted by the operators *op1,op2,...* at the indices specified by the indices $i1, i2, \ldots, j1, j2, \ldots$. Each replacement operator replaces a single qubit identity operator $\mathbb{1}_2$, even if the replacement operator has dimension other than 2. For example

```
insert_operator(8,[pauli[1],1,3],[pauli[2],2,5],[pauli[3],8])
```

returns

$$\sigma_x \otimes \sigma_y \otimes \sigma_x \otimes \mathbb{1}_2 \otimes \sigma_y \otimes \mathbb{1}_2 \otimes \mathbb{1}_2 \otimes \sigma_z.$$

**insert_operator** is used to build the operators and gates listed below.

## 5.4 Gates

### 5.4.1 hadamard operator.

*qinf* defines both a variable and a function named **hadamard**. The value of the variable **hadamard** is as follows,

```
(%i2) hadamard;
```

(%o2)
$$\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

The function **hadamard**$(nbits, i1, i2, \ldots)$ returns the tensor product of *nbits* one-qubit operators each of which is $\mathbb{1}_2$ except for the operators in positions *i1,i2,...* which are the hadamard operator given by the variable **hadamard**. The function is defined by `hadamard(nbits,[t]) := insert_operator(nbits,cons(hadamard,t))`, which is an idiom that will work for similar user defined functions as well. The dummy argument `[t]` causes all arguments after `nbits` to be collected in a single list. The *Maxima* function **cons**$(expr, list)$ returns the list given by prepending `expr` to the list `list`.

### 5.4.2 controlled_gate — gate with $n$ control qubits

A controlled gate applies an operator $m$ to a single qubit (the target) in a register only if each of a set of control qubits is set. Otherwise the operator is equivalent to the identity operator. In any case, the operator is the identity on every qubit other than the target. The function **controlled_gate**$(nbits, qop, t, clst)$ creates a controlled gate with one or more control qubits embedded in a `nbits`-qubit operator. The target qubit is at the index `t`, while the control qubits are at the indices listed in the list `clst`. In the returned operator, the $\mathbb{1}_2$ operator is at the remaining positions. The controlled gate operator applies `qop` at qubit `t` if all of the control qubits are set (ie are 1) and is equivalent to the identity operator otherwise. For instance a **cnot** operator on $H_2 \otimes H_2$ is given by `controlled_gate(2,pauli[1],2,[1])`. The controlled gate in an $n$-qubit space with $n-1$ control bits and the 1-qubit target operator $m$ is implemented in *qinf* as

$$\mathbb{1}_{2^n} + (|1\rangle\langle 1|)^{\otimes n-1} \otimes (m - \mathbb{1}_2).$$

In the case that this controlled gate operator is embedded in an operator in a larger space, the same formula is used, with additional factors of $\mathbb{1}_2$ inserted at the appropriate places. Also the target qubit may occupy any index. This is implemented via the **insert_operator** function described above.

### 5.4.3 cnot, cphase, crot.

**cnot**$(nbits, t, c1, c2, \ldots)$ returns a cnot gate on an `nbits`-qubit register with the target at index `t` and control qubits at indices `c1,c2,...`. The definition of this function is `cnot(nbits,t,[c]) := controlled_gate(nbits,pauli[1],t,c)`. The functions **cphase** and **crot** are defined in the same way except that operators `pauli[3]` and `%i*pauli[2]` respectively are substituted for `pauli[1]`. For example, the elementary cnot gate is given by `cnot(2,2,1)`.

### 5.4.4 swap.

**swap**$(nbits, t1, t2)$ returns the operator that swaps qubits `t1` and `t2` in an `nbits`-qubit register. It is defined by `swap(nbits,t1,t2) := cnot(nbits,t1,t2) . cnot(nbits,t2,t1) . cnot(nbits,t1,t2)`.

#### 5.4.5 toffoli.

This function is provided for convenience. It is defined by `toffoli(nbits,t,c1,c2) := cnot(nbits,t,c1,c2)`. Note that there are exactly two control qubits.

#### 5.4.6 fredkin.

**fredkin**$(nbits, t1, t2, c)$ is controlled swap operator. The qubits at indices `t1` and `t2` are swapped if the control qubit at index `c` is set.

# 6 Entanglement

Now we are ready to introduce features that are more specific to the study quantum entanglement. Included here are measures of the degree of purity and measures of the degree of entanglement. For a review of quantum entanglement, see Ref.[1].

## 6.1 ptrace, ptracen — partial trace.

compute the partial trace of the density operator $\rho$ over the component spaces given by the indices. For **ptrace**$(\rho, i_1, \ldots)$ the density matrix $\rho$ is assumed to represent an operator in $(H_2)^{\otimes m}$, with $n = 2$, that is a tensor product of qubit spaces. For **ptracen**$(n, \rho, i_1, \ldots)$ the component spaces are all $n$-state qudits.

In this example we create three arbitrary $3 \times 3$ matrices, and check that

$$\mathrm{Tr}_{1,2}\left(m_1 \otimes m_2 \otimes m_3\right) = \left(\mathrm{Tr}(m_1)\mathrm{Tr}(m_2)\right) m_3$$

and that

$$\mathrm{Tr}_3\left(m_1 \otimes m_2 \otimes m_3\right) = \mathrm{Tr}(m_3)(m_1 \otimes m_2).$$

Notice that we use the *Maxima* function **ratsimp** to put both sides of the equation in the same canonical form. We also make use of *Maxima*'s **mat_trace** function ( so named to avoid conflicting with the code-execution trace function) and *Maxima*'s **matrix** function, which creates a matrix from a list of rows.

```
(%i1) m1 : matrix([a1,b1,c1],[d1,e1,f1],[g1,h1,i1])$

(%i2) m2 : matrix([a2,b2,c2],[d2,e2,f2],[g2,h2,i2])$

(%i3) m3 : matrix([a3,b3,c3],[d3,e3,f3],[g3,h3,i3])$

(%i4) is ( ratsimp( ptracen(3, m1 otimes m2 otimes m3 , 1,2) =
            mat_trace(m1)*mat_trace(m2)*m3));

(%o4)                                          true

(%i5) is ( ratsimp( ptracen(3, m1 otimes m2 otimes m3 ,3) = mat_trace(m3)* m1 otimes m2));

(%o5)                                          true
```

Here we trace over one component repeatedly and check that the result is equal to the full trace. Note that, each time, we are tracing over the new first component.

```
(%i10) factor( ptracen(3,ptracen(3,ptracen(3,m1 otimes m2 otimes m3,1),1),1));
```

(%o10) $$( (\mathrm{i1} + \mathrm{e1} + \mathrm{a1})\ (\mathrm{i2} + \mathrm{e2} + \mathrm{a2})\ (\mathrm{i3} + \mathrm{e3} + \mathrm{a3}) )$$

## 6.2 entropy — von Neumann entropy.

**entropy**($\rho$) returns the von Neumann **entropy** of the density matrix $\rho$ defined by

$$S(\rho) \ = -\mathrm{Tr}\left(\rho \log_2 \rho\right). \tag{5}$$

**entropyf** is the floating point version of **entropy**.

## 6.3 renyi_entropy.

**renyi_entropy**($alpha, rho$) gives the Rényi entropy, defined by

$$S_{\mathrm{renyi}}(\alpha, \rho) \ = \frac{1}{1-\alpha} \log_2 \left(\mathrm{Tr}(\rho^{\alpha})\right). \tag{6}$$

**renyi_entropyf** is the floating point version of **renyi_entropy**.

## 6.4 tsallis_entropy.

**tsallis_entropy**($q, rho$) gives the Tsallis entropy, defined by

$$S_{\mathrm{tsallis}}(q, \rho) \ = \frac{1}{q-1} \left(1 - \mathrm{Tr}(\rho^{q})\right). \tag{7}$$

**tsallis_entropyf** is the floating point version of **tsallis_entropy**.

## 6.5 purity.

**purity**($\rho$) returns the purity of the density matrix $\rho$ defined by $\mathrm{Tr}(\rho^2)$. The purity is 1 for a pure state and is less than 1 for a mixed state.

## 6.6 fidelity.

**fidelity**($\rho_1, \rho_2$) returns the scalar valued fidelity of the density matrices $\rho_1$ and $\rho_2$ defined by

$$\mathrm{Tr}\left(\sqrt{\sqrt{\rho_2}\rho_1\sqrt{\rho_2}}\right).$$

## 6.7 concurrence — Wootter's concurrence.

The concurrence of a two-qubit state $\rho$ is defined by $\max(0, \sqrt{\lambda_1} - \sqrt{\lambda_2} - \sqrt{\lambda_3} - \sqrt{\lambda_4})$, where the $\lambda_i$ are the eigenvaluse of

$$\rho(\sigma_y \otimes \sigma_y)\rho^*(\sigma_y \otimes \sigma_y),$$

in decreasing order[2]

`concurrence(rho)` returns the concurrence of the state `rho`. `concurrence_vals(rho)` returns a list of the square roots of the eigenvalues in decreasing order, to the extent that *Maxima* can determine the order.

## 6.8 separable — test for separability.

This currently accepts only pure, bipartite states. **separable**($e$) attempts to determine if $e$ is a separable state. **separable** returns a scalar value $r$. If $r = 1$, then $e$ is separable. If $r < 1$, it is not. Explicitly, the value returned for the state with density operator $\rho_{AB}$ is $\mathrm{Tr}(\rho_A^2)$

When other methods are added, the organization and naming of the tests may change.

## 6.9 An entangled pure global state has mixed reduced states

We examine a textbook example of entanglement— the joint state of two qubits. The state of the whole system is pure, but the local states are mixed. We begin by creating a joint state of two qubits in Schmidt basis $|\alpha\rangle = \sqrt{\alpha}|00\rangle + \sqrt{1-\alpha}|11\rangle$. In order to see the mixed character of the local states, we need to express the full state as a density operator (or equivalently as a projection operator.) Let's try to make $|\alpha\rangle\langle\alpha|$.

```
(%i2)  pr : proj(schmidt_ket(alpha));
```

(%o2)
$$\begin{pmatrix} \sqrt{\alpha}^\star\sqrt{\alpha} & 0 & 0 & \sqrt{1-\alpha}^\star\sqrt{\alpha} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \sqrt{\alpha}^\star\sqrt{1-\alpha} & 0 & 0 & \sqrt{1-\alpha}^\star\sqrt{1-\alpha} \end{pmatrix}$$

We see that *Maxima* is allowing that the quantities under the radicals may be negative. So we set some rules, and try again.

```
(%i3) assume(alpha>0, 1-alpha>0);
```

(%o4)
$$[\alpha > 0, \alpha < 1]$$

```
(%i5)  pr : proj(schmidt_ket(alpha));
```

(%o5)
$$\begin{pmatrix} \alpha & 0 & 0 & \sqrt{1-\alpha}\sqrt{\alpha} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \sqrt{1-\alpha}\sqrt{\alpha} & 0 & 0 & 1-\alpha \end{pmatrix}$$

The entropy vanishes for a pure state, so that $S(|\alpha\rangle\langle\alpha|)$ is

```
(%i6)  entropy(pr);
```

(%o7) $0$

The **purity** is equal to 1 if and only if $\rho$ is a pure state.

```
(%i8) purity(pr);
```

(%o8) $$\alpha^2 + 2\,(1-\alpha)\,\alpha + (1-\alpha)^2$$

The above line should be simplified by writing it as a canonical rational expression (CRE)

```
(%i9) ratsimp(%);
```

(%o9) $1$

Now we compute the reduced density matrix of the second qubit by tracing over the first— $\rho_2 = \mathrm{Tr}_1|\alpha\rangle\langle\alpha|$.

```
(%i10)  pr2 : ptrace(pr,1);
```

(%o10) $$\begin{pmatrix} \alpha & 0 \\ 0 & 1-\alpha \end{pmatrix}$$

Tracing over the second qubit instead gives the same result

```
(%i11) ptrace(pr,2);
```

(%o11) $$\begin{pmatrix} \alpha & 0 \\ 0 & 1-\alpha \end{pmatrix}$$

Computing the entropy of a local state shows that this state is, in general, mixed

```
(%i12) entropy(pr2);
```

(%o12) $$-\alpha\log 2\,(\alpha) - \log 2\,(1-\alpha)\,(1-\alpha)$$

Each eigenvalue $\lambda$ satisfies $0 \leq \lambda < 1$, so that the sum of their squares is less than one

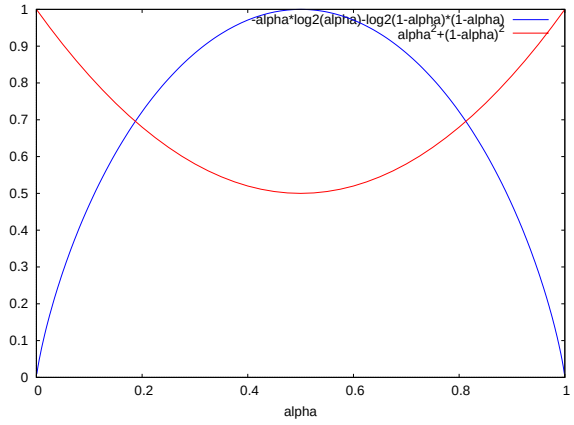
```
(%i13) purity(pr2);
```

(%o13) $$\alpha^2 + (1-\alpha)^2$$

We can plot the results (the plot function **plot2d** is more common, depending on your user interface. **wxplot2d** has the same calling syntax, but inlines the resulting plot.) We see that the maximum entanglement occurs at $\alpha = 1/2$ and decreases monotonically from there in both directions, with $\alpha = 0, 1$ giving pure joint states.

```
(%i14)  wxplot2d([entropy(pr2), purity(pr2)],[alpha,0,1]);
```

(%o14)




# 7 More operators and functions

Unless noted, all functions return either true or false.

## 7.1 Predicate functions — testing for properties

### 7.1.1 identitymatrixp

**identitymatrixp**($e$) returns true if $e$ is the $n \times n$ identity matrix for any $n$.

### 7.1.2 ketp, brap, braketp

**ketp**($e$) returns true if the expression $e$ is a ket (column vector). **brap**($e$) returns true if the expression $e$ is a bra (row vector). **braketp**($e$) returns `(ketp(e) or brap(e))`, ie returns true if $e$ is either a ket or a bra.

### 7.1.3 square_matp

**square_matp**($e$) returns true if $e$ is a square matrix.

# 8 Entanglement. longer examples

## 8.1 Entanglement swapping

In this subsection, we check a calculation that would be relatively difficult to check by hand. Consider a pair of entangled qubits $A$ and $B$, and another entangled pair $C$ and $D$. By performing a joint measurement on, say $B$ and $C$, we can put $A$ and $B$ in an entangled state although they may be widely separated. We begin by considering the most general projective measure on $B$ and $C$, and calculate the reduced density matrix for a single qubit and the probability of outcome. In this example we calculate these quantities two ways— one, directly from the density matrix formalism, and two, via formulas taking advantage of the particulars of this problem. To do the first calculation by hand would be extremely unpleasant, as it involves multiplying $16 \times 16$ matrices with several factors in a single element. Carrying it out below with *Maxima* is a concise exercise. At present this example does not continue by discussing the measurements that maximize the resulting entanglement of $A$ and $D$.

Qubits $A$ and $B$ are in the state

$$|\alpha\rangle \ = \sqrt{\alpha}|00\rangle + \sqrt{1-\alpha}|11\rangle,$$

$C$ and $D$ are in the state

$$|\beta\rangle \ = \sqrt{\beta}|00\rangle + \sqrt{1-\beta}|11\rangle,$$

with the Schmidt coefficients satisfying $\alpha, \beta > 1/2$. For now, we only want to tell *Maxima* that the coefficients of the kets are real.

```
(%i2) assume(alpha>0,1-alpha>0,beta>0,1-beta>0);
```

(%o2) $$[\alpha > 0, \alpha < 1, \beta > 0, \beta < 1]$$

```
(%i3) a : schmidt_ket(alpha);
```

(%o3) $$\begin{pmatrix} \sqrt{\alpha} \\ 0 \\ 0 \\ \sqrt{1-\alpha} \end{pmatrix}$$

```
(%i4) b : schmidt_ket(beta);
```

(%o4) $$\begin{pmatrix} \sqrt{\beta} \\ 0 \\ 0 \\ \sqrt{1-\beta} \end{pmatrix}$$

We consider the projective measurement $\{E_m\}$, that is $E_m = |u_m\rangle\langle u_m|$ and $\sum_m E_m = \mathbb{1}_4$. We consider only a single basis vector here, so we don't use the subscript $m$ for *Maxima* vector name. We need to use *Maxima*'s **declare** to declare that the components are complex. The state $|u_m\rangle$ is normalized, but we don't need to impose that condition in *Maxima* at this point.

```
(%i5)  declare([u00,u01,u10,u11], complex);
```

(%o5) **done**

```
(%i6) u : ket(u00,u01,u10,u11);
```

(%o6) $$\begin{pmatrix} \text{u00} \\ \text{u01} \\ \text{u10} \\ \text{u11} \end{pmatrix}$$

The initial joint state $|\alpha\beta\rangle\langle\alpha\beta|$ is pure and remains so after the measurement applying $|u_m\rangle\langle u_m|$ to qubits $B$ and $C$. But we write the density operator because we will examine the reduced states, which are mixed. In the case that $B$ and $C$ are projected onto $|u_m\rangle$, the state of the entire system of four qubits after the measurement is given by

$$\rho = ((\mathbb{1}_2 \otimes |u_m\rangle\langle u_m| \otimes \mathbb{1}_2)|\alpha\beta\rangle\langle\alpha\beta|)\,, \tag{8}$$

with

```
(%i7) rho : conjsimp((ident(2) otimes proj(u) otimes ident(2)) . proj(a otimes b))$,
```

where **conjsimp** (supplied via the *Maxima* listserv by Barton Willis) replaces $xx^*$ with $|x|^2$, and **ident**(n) is the $n \times n$ identity matrix. The output was suppressed with the trailing dollar sign because the $\rho$ is a $16 \times 16$ matrix with large expressions for entries. The reduced state of qubits $A$ and $D$ is obtained by tracing out components 2 and 3 corresponding to qubits $B$ and $C$, ie $\rho_{AD} = \mathrm{Tr}_{BC}\rho$.

```
(%i8) rho_14 : ptrace(rho,2,3);
```

(%o8)

$$\begin{pmatrix} \alpha\,\beta\,|\mathrm{u00}|^2 & \alpha\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u00}^\star\,\mathrm{u01} & \sqrt{1-\alpha}\sqrt{\alpha}\,\beta\,\mathrm{u00}^\star\,\mathrm{u10} & \sqrt{1-\alpha}\sqrt{\alpha}\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u00}^\star\,\mathrm{u11} \\ \alpha\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u00}\,\mathrm{u01}^\star & (\alpha-\alpha\,\beta)\,|\mathrm{u01}|^2 & \sqrt{1-\alpha}\sqrt{\alpha}\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u01}^\star\,\mathrm{u10} & \left(\sqrt{1-\alpha}\sqrt{\alpha}-\sqrt{1-\alpha}\sqrt{\alpha}\,\beta\right)\mathrm{u01}^\star\,\mathrm{u11} \\ \sqrt{1-\alpha}\sqrt{\alpha}\,\beta\,\mathrm{u00}\,\mathrm{u10}^\star & \sqrt{1-\alpha}\sqrt{\alpha}\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u01}\,\mathrm{u10}^\star & (1-\alpha)\,\beta\,|\mathrm{u10}|^2 & (1-\alpha)\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u10}^\star\,\mathrm{u11} \\ \sqrt{1-\alpha}\sqrt{\alpha}\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u00}\,\mathrm{u11}^\star & \left(\sqrt{1-\alpha}\sqrt{\alpha}-\sqrt{1-\alpha}\sqrt{\alpha}\,\beta\right)\mathrm{u01}\,\mathrm{u11}^\star & (1-\alpha)\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u10}\,\mathrm{u11}^\star & ((\alpha-1)\,\beta-\alpha+1)\,|\mathrm{u11}|^2 \end{pmatrix}$$

Likewise, the reduced state of just qubit $D$ is $\rho_D = \mathrm{Tr}_{ABC}\rho$.

```
(%i9) rho_4 : ptrace(rho,1,2,3);
```

(%o9)

$$\begin{pmatrix} (1-\alpha)\,\beta\,|\mathrm{u10}|^2+\alpha\,\beta\,|\mathrm{u00}|^2 & (1-\alpha)\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u10}^\star\,\mathrm{u11}+\alpha\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u00}^\star\,\mathrm{u01} \\ (1-\alpha)\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u10}\,\mathrm{u11}^\star+\alpha\sqrt{1-\beta}\sqrt{\beta}\,\mathrm{u00}\,\mathrm{u01}^\star & ((\alpha-1)\,\beta-\alpha+1)\,|\mathrm{u11}|^2+(\alpha-\alpha\,\beta)\,|\mathrm{u01}|^2 \end{pmatrix}$$

The second method of calculating $\rho_D$ is as follows. Considering the following map from $\mathbb{C}^2 \otimes \mathbb{C}^2$ to $M(\mathbb{C}, 2)$:

$$|a\rangle = \sum_{i,j=0}^{1} a_{ij}|ij\rangle \quad \mapsto \quad \widehat{a} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}, \tag{9}$$

one can show that $\rho_D$ is equal to $X_m^\dagger X_m$, with $X_m = \widehat{\alpha}\,\widehat{u}_m\,\widehat{\beta}$. The *Maxima* function implementing the mapping (9) is

```
(%i10) ket_to_mat(iket) := matrix([iket[1,1],iket[2,1]],[iket[3,1],iket[4,1]])$
```

Then the second calculation of $\rho_D$, which we call `rho_4a` is given by the following lines.

```
(%i11) mu : ket_to_mat(u)$
(%i12) X :  ket_to_mat(a) . mu . ket_to_mat(b);
```

(%o12)
$$\begin{pmatrix} \sqrt{\alpha}\,\sqrt{\beta}\,\mathrm{u00} & \sqrt{\alpha}\,\sqrt{1-\beta}\,\mathrm{u01} \\ \sqrt{1-\alpha}\,\sqrt{\beta}\,\mathrm{u10} & \sqrt{1-\alpha}\,\sqrt{1-\beta}\,\mathrm{u11} \end{pmatrix}$$

```
(%i13) rho_4a : conjsimp( ctranspose(X) . X );
```

(%o13)
$$\begin{pmatrix} (1-\alpha)\ \beta\ |\mathrm{u10}|^2 + \alpha\,\beta\ |\mathrm{u00}|^2 & (1-\alpha)\ \sqrt{1-\beta}\,\sqrt{\beta}\,\mathrm{u10}^\star\,\mathrm{u11} + \alpha\,\sqrt{1-\beta}\,\sqrt{\beta}\,\mathrm{u00}^\star\,\mathrm{u01} \\ (1-\alpha)\ \sqrt{1-\beta}\,\sqrt{\beta}\,\mathrm{u10}\,\mathrm{u11}^\star + \alpha\,\sqrt{1-\beta}\,\sqrt{\beta}\,\mathrm{u00}\,\mathrm{u01}^\star & ((\alpha-1)\ \beta-\alpha+1)\ |\mathrm{u11}|^2 + (\alpha-\alpha\,\beta)\ |\mathrm{u01}|^2 \end{pmatrix}$$

We compare `(%o9)` and `(%o13)`, to see that the two methods of calculating the reduced state for qubit $D$ after the measurement give the same result

```
(%i14) is ( ratsimp(rho_4a) = ratsimp(rho_4) );
```

(%o14) **true**

Now we compute the probability $p_m = \mathrm{Tr}(\rho) = \mathrm{Tr}(\rho_D)$ that the state is in fact projected onto $|u_m\rangle$.

```
(%i15) P1 : conjsimp( mat_trace(rho));
```

(%o15) $$((\alpha-1)\ \beta-\alpha+1)\ |\mathrm{u11}|^2 + (1-\alpha)\ \beta\ |\mathrm{u10}|^2 + (\alpha-\alpha\,\beta)\ |\mathrm{u01}|^2 + \alpha\,\beta\ |\mathrm{u00}|^2$$

Finally, we compare this to the trace computed by hand from the expression following (9), which is given by

$$p_m \ = \sum_{i,j=0}^{1} \alpha_i \beta_j |\widehat{u}_{m,ij}|^2, \tag{10}$$

where $\alpha_0 = \alpha,\ \alpha_1 = 1-\alpha,\ \beta_0 = \beta,\ \beta_1 = 1-\beta,$

```
(%i16) av : [alpha,1-alpha]$
(%i17) bv : [beta,1-beta] $
(%i18) P2 : apply( "+", create_list( av[i] * abs(mu[i,j])^2 * bv[j], i,[1,2],j,[1,2]));
```

(%o18) $$(1-\alpha)\ (1-\beta)\ |\mathrm{u11}|^2 + (1-\alpha)\ \beta\ |\mathrm{u10}|^2 + \alpha\ (1-\beta)\ |\mathrm{u01}|^2 + \alpha\,\beta\ |\mathrm{u00}|^2 .$$

Here we have used *Maxima*'s **apply** and **create_list** functions. Once again we compare the probabilities computed via the two methods

```
(%i19) is (ratsimp(P1) = ratsimp(P2));

(%o19)                                    true
```

and see that they give the same result.

# References


[1] R. Horodecki, P. Horodecki, M. Horodecki, and K. Horodecki. *Rev. Mod. Phys.*, 2008. to appear. `arXiv:arXiv:quant-ph/0702225v2`.

[2] William K. Wootters. Entanglement of formation of an arbitrary state of two qubits. *Phys. Rev. Lett.*, 80:2245–2248, 1998. `arXiv:arXiv:quant-ph/9709029v2`.

# Index