Quantum Information Package For The Maxima Computer Algebra System

G. John Lapeyre, Jr.

August 15, 2008

1 Introduction

This quantum information package for the Maxima computer algebra system allows the manipulation of instances of objects, operators,vectors,tensors, *etc.* appearing in quantum information theory. More precisely these objects are typically represented in a particular basis as row and column vectors and matrices, whose entries may be explicit numbers (of various classes) or algebraic expressions. 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.

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.

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

- You probably need to read at least a ten minute 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.
- Functions and features that are part of the standard Maxima distribution, rather than part of the quantum information package are marked, where not obvious, with the dagger superscript— †.
- There are several user interfaces to the 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.

- Most functions currently work only with qubits, others for variable number of states.

The package is intended to be used for research in the theory of entanglement and quantum information and related fields.

1.1 Acknowledgments

Some of the ideas used in this package are inspired by the package qdens written for a proprietary symbolic algebra system. (put reference here)

2 Tutorial introduction

The majority of the examples below are taken from the regression tests for the qinf package or from the author's research.

The package is loaded by entering the command `load("qinf.mac");`

```
(%i2) load("qinf.mac")
```

(%o4) qinf.mac

2.1 Using Maxima

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 $m \times n$ and B a $p \times q$ matrix, then the result is a $n \times p$ matrix. The inner product of quantum state vectors, the outer product 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.

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

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

Suppress the output here, because it's big– 51 terms.

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

(%o6) 51

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) $$f\left(x\right) := 3\,\cos x$$

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

(%o10) $$3\,\cos 4$$

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

(%o11) 3

Complex numbers.

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

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

Special numbers.

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

(%o13) 0

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

(%o14) i

2.2 Creating and manipulating states and operators

2.2.1 Representation of states and operators

Kets are represented by $n \times 1$ matrices, bras by $1 \times n$ matrices. These vectors are in the z basis. Bras and kets representing the same states are related by the conjugate transpose function. 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 the appropriate space. That is, there is no facility to distinguish between a vector in the product space of two qubits $\mathcal{H}_1 \otimes \mathcal{H}_2$ and the space of a single four state qudit.

2.3 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.

2.3.1 **ketz**$(i_1, \ldots, i_n)$**, braz**$(i_1, \ldots, i_n)$**, ketx**$(i_1, \ldots, i_n)$**, brax**$(i_1, \ldots, i_n)$**, kety**$(i_1, \ldots, i_n)$**, bray**$(i_1, \ldots, i_n)$

create normalized n-partite statesin the z-basis. In all cases i_n are 0 or 1. The pair **ketz** and **braz** 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)
$$\begin{pmatrix} 1 & 0 \end{pmatrix}$$

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

2.3.2 ket_n($j, i_1, \ldots, i_m$),bra_n($j, i_1, \ldots, i_m$)

These 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.

2.3.3 Density matrix representation of a pure state (projection operator)

The projection operator for corresponding to a state vector is generated via the outer product, which is represented by the dot operator. A convenience function **toproj**(*ket*) is also provided to form a projection operator (**toproj** does not check that *ket* is normalized.) Below, we use the Maxima function **ctranspose**.

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

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

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

(%o20) 1

```
(%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) =  toproj(ketz(1,0,1)) );
```

(%o22) **true**

There is also a function **tostate** (needs a better name) that is the inverse of **toproj**— it returns the ket corresponding to a projection operator. If the input matrix is not a projection operator, the result is undefined.

```
(%i17) is ( tostate( toproj(schmidt_ket(alpha))) = schmidt_ket(alpha));
```

(%o17) **true**

2.3.4 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 following uses Maxima's **is**$(expr)$ function which tries to determine if the predicate *expr* is true. 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) otimes 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**

2.3.5 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.

2.3.6 bell[a,b] and belln[i]

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 $((0, 0), (0, 1), (1, 0), (1, 1))$.

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) $$f_{x,y} := \mathrm{belln}_x \cdot \mathrm{belln}_y$$

Create a 4×4 matrix with Maxima's **genmatrix** 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 Maxima's **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 4×4 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],toproj(belln[i])),[0,1,2,3])));
```

(%o2) **true**

2.4 Creating and using operators

2.4.1 pauli[i]

creates the pauli matrices.

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

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

The ket $|1\rangle_x$ is an eigenvector of σ_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.

```
(%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** and **listify** (which turns a set into an ordered list). 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. 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.)

```
(%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**]

2.5 Entanglement– partial trace, entropy, purity

Now we are ready to introduce features that are more specific to the study quantum entanglement. We examine the textbook example of entanglement– the joint state of two qubits. We begin by creating a joint state of two qubits in Schmidt bases $\sqrt{\alpha}|00\rangle + \sqrt{1-\alpha}|11\rangle$. In order to see the mixed character of the local states, we need to express the full state as a density operator (or equivelantly as a projection operator.)

```
(%i2)  pr : toproj(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 : toproj(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 von Neumann entropy defined as

(3)
$$S(\rho) = -\mathrm{Tr}\rho\log_2\rho,$$

vanishes for a pure state

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

(%o7)
$$0$$

The purity $\mathrm{Tr}(\rho^2)$ is equal to 1 for 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 rations expression (CRE)

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

(%o9) 1

Now we compute the reduced density matrix of the second qubit by tracing over the first

```
(%i10)  pr1 : 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(pr1);
```

(%o12) $$-\alpha \log2 (\alpha) - \log2 (1-\alpha)\ (1-\alpha)$$

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

```
(%i13) purity(pr1);
```

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

We can plot the results (the plot command **plot2d** is more common, depending on your user interface)

```
(%i14)  wxplot2d([entropy(pr1), purity(pr1)],[alpha,0,1]);
```

(%o14)