1.1 Introduction to package qm

The `qm` package was written by Eric Majzoub, University of Missouri. Email: majzoube-at-umsystem.edu

The package is loaded with: `load(qm);`

The `qm` package provides functions and standard definitions to solve quantum mechanics problems in a finite dimensional Hilbert space. For example, one can calculate the outcome of Stern-Gerlach experiments using built-in definitions of the Sx, Sy, and Sz operators for arbitrary spin, e.g. `s={1/2, 1, 3/2, ...}`. For spin-1/2 the standard basis kets in the `x`, `y`, and `z`-basis are available as `{xp,xm}`, `{yp,ym}`, and `{zp,zm}`, respectively. One can create general ket vectors with arbitrary but finite dimension and perform standard computations such as expectation value, variance, etc. The angular momentum |*j,m*> representation of kets is also available. Tensor product states for multiparticle systems can be created to perform calculations on those systems.

Let us consider a simple example involving spin-1/2 particles. A bra vector in the `z`-basis may be written as

```
<psi| = a <z+| + b <z-|.
```

The bra `<psi|` will be represented in Maxima by the row vector `[a b]`, where the basis vectors are

```
<z+| = [1 0]
```

and

```
<z-| = [0 1].
```

In a Maxima session this looks like the following. The basis kets `{zp,zm}` are transformed into bras using the `dagger` function.

```
(%i1) psi_bra:a*dagger(zp)+b*dagger(zm);
(%o1)                              [ a  b ]
```

1.1.1 Types of kets and bras

There are two types of kets and bras available in the `qm` package, the first type is given by a *matrix representation*, as returned by the functions `mbra` and `mket`. `mket`s are column vectors and `mbra`s are row vectors, and their components are entered as Maxima *lists* in the `mbra` and `mket` functions. The second type of bra or ket is *abstract*; there is no matrix representation. Abstract bras and kets are entered using the `bra` and `ket` functions, while also using Maxima lists for the elements. These general kets are displayed in Dirac notation. Abstract bras and kets are used for both the `(j,m)` representation of states and also for tensor products. For example, a tensor product of two ket vectors `|a>` and `|b>` is input as `ket([a,b])` and displayed as

`|[a,b]>`   (general ket)

Note that abstract kets and bras are *assumed to be orthonormal*. These general bras and kets may be used to build arbitrarily large tensor product states.

The following examples illustrate some of the basic capabilities of the `qm` package. Here both abstract, and concrete (matrix representation) kets are shown. The last example shows how to contstruct an entangled Bell pair.

```
(%i1) ket([a,b])+ket([c,d]);
(%o1)                        |[c, d]> + |[a, b]>
(%i2) mket([a,b]);
                                   [ a ]
(%o2)                              [   ]
                                   [ b ]
(%i3) mbra([a,b]);
(%o3)                            [ a  b ]
(%i4) bell:(1/sqrt(2))*(ket([u,d])-ket([d,u]));
                             |[u, d]> - |[d, u]>
(%o4)                        -------------------
                                   sqrt(2)
(%i5) dagger(bell);
                             <[u, d]| - <[d, u]|
(%o5)                        -------------------
                                   sqrt(2)
```

Note that `ket([a,b])` is treated as tensor product of states `a` and `b` as shown below.

```
(%i1) braket(bra([a1,b1]),ket([a2,b2]));
(%o1)                kron_delta(a1, a2) kron_delta(b1, b2)
```

Constants that multiply kets and bras must be declared complex by the user in order for the dagger function to properly conjugate such constants. The example below illustrates this behavior.

```
(%i1) declare([a,b],complex);
(%o1)                               done
(%i2) psi:a*ket([1])+b*ket([2]);
(%o2)                         |[2]> b + |[1]> a
(%i3) psidag:dagger(psi);
(%o3)               <[2]| conjugate(b) + <[1]| conjugate(a)
(%i4) psidag . psi;
(%o4)                   b conjugate(b) + a conjugate(a)
```

The following shows how to declare a ket with both real and complex components in the matrix representation.

```
(%i1) declare([c1,c2],complex,r,real);
(%o1)                               done
(%i2) k:mket([c1,c2,r]);
                                   [ c1 ]
                                   [    ]
(%o2)                              [ c2 ]
                                   [    ]
                                   [ r  ]
(%i3) b:dagger(k);
(%o3)                [ conjugate(c1)  conjugate(c2)  r ]
(%i4) b . k;
                    2
(%o4)              r  + c2 conjugate(c2) + c1 conjugate(c1)
```

1.2 Functions and Variables for qm

hbar [Variable]

Planck's constant divided by `2*%pi`. `hbar` is not given a floating point value, but is declared to be a real number greater than zero.

ket ([k_1,k_2,...]) [Function]

`ket` creates a general state ket, or tensor product, with symbols k_i representing the states. The state kets k_i are assumed to be orthonormal.

```
(%i1) k:ket([u,d]);
(%o1)                           |[u, d]>
(%i2) b:bra([u,d]);
(%o2)                           <[u, d]|
(%i3) b . k;
(%o3)                               1
```

ketp (*abstract ket*) [Function]

`ketp` is a predicate function for abstract kets. It returns `true` for abstract `kets` and `false` for anything else.

bra ([b_1,b_2,...]) [Function]

`bra` creates a general state bra, or tensor product, with symbols b_i representing the states. The state bras b_i are assumed to be orthonormal.

```
(%i1) k:ket([u,d]);
(%o1)                           |[u, d]>
(%i2) b:bra([u,d]);
(%o2)                           <[u, d]|
(%i3) b . k;
(%o3)                               1
```

brap (*abstract bra*) [Function]

`brap` is a predicate function for abstract bras. It returns `true` for abstract `bras` and `false` for anything else.

mket ([c_1,c_2,...]) [Function]

`mket` creates a *column* vector of arbitrary finite dimension. The entries c_i can be any Maxima expression. The user must `declare` any relevant constants to be complex. For a matrix representation the elements must be entered as a list in `[...]` square brackets.

```
(%i1) declare([c1,c2],complex);
(%o1)                              done
(%i2) mket([c1,c2]);
                                   [ c1 ]
(%o2)                              [    ]
                                   [ c2 ]
(%i3) facts();
(%o3) [kind(hbar, real), hbar > 0, kind(c1, complex), kind(c2, complex)]
```

`mketp` (*ket*) [Function]

`mketp` is a predicate function that checks if its input is an mket, in which case it returns `true`, else it returns `false`. `mketp` only returns `true` for the matrix representation of a ket.

```
(%i1) k:ket([a,b]);
(%o1)                        |[a, b]>
(%i2) mketp(k);
(%o2)                          false
(%i3) k:mket([a,b]);
                               [ a ]
(%o3)                          [   ]
                               [ b ]
(%i4) mketp(k);
(%o4)                          true
```

`mbra` ($[c_1, c_2, \ldots]$) [Function]

`mbra` creates a *row* vector of arbitrary finite dimension. The entries c_i can be any Maxima expression. The user must `declare` any relevant constants to be complex. For a matrix representation the elements must be entered as a list in `[...]` square brackets.

```
(%i1) kill(c1,c2);
(%o1)                          done
(%i2) mbra([c1,c2]);
(%o2)                        [ c1  c2 ]
(%i3) facts();
(%o3)                 [kind(hbar, real), hbar > 0]
```

`mbrap` (*bra*) [Function]

`mbrap` is a predicate function that checks if its input is an mbra, in which case it returns `true`, else it returns `false`. `mbrap` only returns `true` for the matrix representation of a bra.

```
(%i1) b:mbra([a,b]);
(%o1)                         [ a  b ]
(%i2) mbrap(b);
(%o2)                          true
```

Two additional functions are provided to create kets and bras in the matrix representation. These functions conveniently attempt to automatically `declare` constants as complex. For example, if a list entry is `a*sin(x)+b*cos(x)` then only `a` and `b` will be `declare`-d complex and not `x`.

`autoket` ($[a_1, a_2, \ldots]$) [Function]

`autoket` takes a list $[a_1, a_2, \ldots]$ and returns a ket with the coefficients a_i `declare`-d complex. Simple expressions such as `a*sin(x)+b*cos(x)` are allowed and will `declare` only the coefficients as complex.

```
(%i1) autoket([a,b]);
                                       [ a ]
(%o1)                                  [   ]
                                       [ b ]
(%i2) facts();
(%o2)  [kind(hbar, real), hbar > 0, kind(a, complex), kind(b, complex)]
(%i1) autoket([a*sin(x),b*sin(x)]);
                                   [ a sin(x) ]
(%o1)                              [          ]
                                   [ b sin(x) ]
(%i2) facts();
(%o2)  [kind(hbar, real), hbar > 0, kind(a, complex), kind(b, complex)]
```

autobra ([a_1,a_2,...]) [Function]

autobra takes a list [a_1,a_2,...] and returns a bra with the coefficients a_i declare-d complex. Simple expressions such as a*sin(x)+b*cos(x) are allowed and will declare only the coefficients as complex.

```
(%i1) autobra([a,b]);
(%o1)                                [ a  b ]
(%i2) facts();
(%o2)  [kind(hbar, real), hbar > 0, kind(a, complex), kind(b, complex)]
(%i1) autobra([a*sin(x),b]);
(%o1)                             [ a sin(x)  b ]
(%i2) facts();
(%o2)  [kind(hbar, real), hbar > 0, kind(a, complex), kind(b, complex)]
```

dagger (*vector*) [Function]

dagger is the quantum mechanical *dagger* function and returns the conjugate transpose of its input. Arbitrary constants must be declare-d complex for dagger to produce the conjugate.

```
(%i1) dagger(mbra([%i,2]));
                                     [ - %i ]
(%o1)                                [      ]
                                     [  2   ]
```

braket (psi,phi) [Function]

Given two kets psi and phi, braket returns the quantum mechanical bracket <psi|phi>. The vector psi may be input as either a ket or bra. If it is a ket it will be turned into a bra with the dagger function before the inner product is taken. The vector phi must always be a ket.

```
(%i1) declare([a,b,c],complex);
(%o1)                                  done
(%i2) braket(mket([a,b,c]),mket([a,b,c]));
(%o2)          c conjugate(c) + b conjugate(b) + a conjugate(a)
(%i3) braket(ket([a1,b1,c1]),ket([a2,b2,c2]));
(%o3)     kron_delta(a1, a2) kron_delta(b1, b2) kron_delta(c1, c2)
```

norm (psi) [Function]

Given a `ket` or `bra` `psi`, `norm` returns the square root of the quantum mechanical bracket `<psi|psi>`. The vector `psi` must always be a `ket`, otherwise the function will return `false`.

```
(%i1) declare([a,b,c],complex);
(%o1)                          done
(%i2) norm(mket([a,b,c]));
(%o2)      sqrt(c conjugate(c) + b conjugate(b) + a conjugate(a))
```

magsqr (c) [Function]

`magsqr` returns `conjugate(c)*c`, the magnitude squared of a complex number.

```
(%i1) declare([a,b,c,d],complex);
(%o1)                          done
(%i2) A:braket(mket([a,b]),mket([c,d]));
(%o2)               conjugate(b) d + conjugate(a) c
(%i3) P:magsqr(A);
(%o3) (conjugate(b) d + conjugate(a) c) (b conjugate(d) + a conjugate(c))
```

1.2.1 Spin-1/2 state kets and associated operators

Spin-1/2 particles are characterized by a simple 2-dimensional Hilbert space of states. It is spanned by two vectors. In the z-basis these vectors are `{zp,zm}`, and the basis kets in the z-basis are `{xp,xm}` and `{yp,ym}` respectively.

zp [Function]

Return the |z+> ket in the z-basis.

zm [Function]

Return the |z-> ket in the z-basis.

xp [Function]

Return the |x+> ket in the z-basis.

xm [Function]

Return the |x-> ket in the z-basis.

yp [Function]

Return the |y+> ket in the z-basis.

ym [Function]

Return the |y-> ket in the z-basis.

```
(%i1) zp;
                                [ 1 ]
(%o1)                           [   ]
                                [ 0 ]
(%i2) zm;
                                [ 0 ]
(%o2)                           [   ]
                                [ 1 ]
```

```
(%i1) yp;
                                        [    1    ]
                                        [ ------- ]
                                        [ sqrt(2) ]
(%o1)                                   [         ]
                                        [   %i    ]
                                        [ ------- ]
                                        [ sqrt(2) ]
(%i2) ym;
                                       [     1     ]
                                       [  -------  ]
                                       [  sqrt(2)  ]
(%o2)                                  [           ]
                                       [     %i    ]
                                       [ - ------- ]
                                       [   sqrt(2) ]
(%i1) braket(xp,zp);
                                            1
(%o1)                                    -------
                                         sqrt(2)
```

Switching bases is done in the following example where a z-basis ket is constructed and the x-basis ket is computed.

```
(%i1) declare([a,b],complex);
(%o1)                                  done
(%i2) psi:mket([a,b]);
                                        [ a ]
(%o2)                                   [   ]
                                        [ b ]
(%i3) psi_x:'xp*braket(xp,psi)+'xm*braket(xm,psi);
                    b         a                 a         b
(%o3)           (------- + -------) xp + (------- - -------) xm
                 sqrt(2)   sqrt(2)        sqrt(2)   sqrt(2)
```

1.2.2 Pauli matrices and Sz, Sx, Sy operators

`sigmax` [Function]

Returns the Pauli x matrix.

`sigmay` [Function]

Returns the Pauli y matrix.

`sigmaz` [Function]

Returns the Pauli z matrix.

`Sx` [Function]

Returns the spin-1/2 Sx matrix.

`Sy` [Function]

Returns the spin-1/2 Sy matrix.

`Sz` [Function]

Returns the spin-1/2 *Sz* matrix.

```
(%i1) sigmay;
                                  [  0    - %i ]
(%o1)                             [            ]
                                  [ %i     0   ]
(%i2) Sy;
                             [              %i hbar ]
                             [    0       - ------- ]
                             [                 2    ]
(%o2)                        [                      ]
                             [ %i hbar              ]
                             [ -------        0     ]
                             [    2                 ]
```

`commutator (X,Y)` [Function]

Given two operators `X` and `Y`, return the commutator `X . Y - Y . X`.

```
(%i1) commutator(Sx,Sy);
                            [       2               ]
                            [ %i hbar               ]
                            [ --------      0       ]
                            [    2                  ]
(%o1)                       [                       ]
                            [                     2 ]
                            [             %i hbar   ]
                            [    0      - -------- ]
                            [                 2     ]
```

`anticommutator (X,Y)` [Function]

Given two operators `X` and `Y`, return the commutator `X . Y + Y . X`.

```
(%i1) (1/2)*anticommutator(sigmax,sigmax);
                                    [ 1  0 ]
(%o1)                               [      ]
                                    [ 0  1 ]
```

1.2.3 SX, SY, SZ operators for any spin

`SX (s)` [Function]

`SX(s)` for spin `s` returns the matrix representation of the spin operator `Sx`. Shortcuts for spin-1/2 are `Sx,Sy,Sz`, and for spin-1 are `Sx1,Sy1,Sz1`.

`SY (s)` [Function]

`SY(s)` for spin `s` returns the matrix representation of the spin operator `Sy`. Shortcuts for spin-1/2 are `Sx,Sy,Sz`, and for spin-1 are `Sx1,Sy1,Sz1`.

`SZ (s)` [Function]

`SZ(s)` for spin `s` returns the matrix representation of the spin operator `Sz`. Shortcuts for spin-1/2 are `Sx,Sy,Sz`, and for spin-1 are `Sx1,Sy1,Sz1`.

Example:

```
(%i1) SY(1/2);
                             [            %i hbar ]
                             [    0     - ------- ]
                             [               2    ]
(%o1)                        [                    ]
                             [ %i hbar            ]
                             [ -------      0     ]
                             [    2               ]
(%i2) SX(1);
                          [           hbar            ]
                          [    0     -------     0    ]
                          [          sqrt(2)          ]
                          [                           ]
                          [  hbar                hbar ]
(%o2)                     [ -------     0     ------- ]
                          [ sqrt(2)           sqrt(2) ]
                          [                           ]
                          [           hbar            ]
                          [    0     -------     0    ]
                          [          sqrt(2)          ]
```

1.2.4 Expectation value and variance

`expect (O,psi)` [Function]

Computes the quantum mechanical expectation value of the operator `O` in state `psi`, `<psi|O|psi>`.

```
(%i1) ev(expect(Sy,xp+ym),ratsimp);
(%o1)                              - hbar
```

`qm_variance (O,psi)` [Function]

Computes the quantum mechanical variance of the operator `O` in state `psi`, `sqrt(<psi|O`^{2}`|psi> - <psi|O|psi>`^{2}`)`.

```
(%i1) ev(qm_variance(Sy,xp+ym),ratsimp);
                                   %i hbar
(%o1)                              -------
                                      2
```

1.2.5 Angular momentum representation of kets and bras

1.2.5.1 Matrix representation of (j,m)-kets and bras

The matrix representation of kets and bras in the `qm` package are represented in the `z`-basis. To create a matrix representation of of a ket or bra in the (j,m)-basis one uses the `spin_mket` and `spin_mbra` functions.

`spin_mket` $(s,m_s,[1,2])$ [Function]

`spin_mket` returns a ket in the `z`-basis for spin `s` and z-projection m_s, for axis 1=X or 2=Y.

`spin_mbra` (s,m_s,[*1*,*2*]) [Function]

`spin_mbra` returns a bra in the `z`-basis for spin `s` and z-projection `m_s`, for axis 1=X or 2=Y.

```
(%i1) spin_mket(3/2,1/2,2);
                                   [  sqrt(3)   ]
                                   [  -------   ]
                                   [    3/2     ]
                                   [   2        ]
                                   [            ]
                                   [     %i     ]
                                   [    ----    ]
                                   [     3/2    ]
                                   [    2       ]
(%o1)                              [            ]
                                   [     1      ]
                                   [    ----    ]
                                   [     3/2    ]
                                   [    2       ]
                                   [            ]
                                   [ sqrt(3) %i ]
                                   [ ---------- ]
                                   [     3/2    ]
                                   [    2       ]
(%i2) spin_mbra(1,1,1);
                                  [ 1     1     1 ]
(%o2)                             [ -  -------  - ]
                                  [ 2  sqrt(2)  2 ]
```

1.2.5.2 Abstract (j,m)-kets and bras

To create kets and bras in the |*j*,*m*> representation you use the abstract `ket` and `bra` functions with `j,m` as arguments, as in `ket([j,m])` and `bra([j,m])`.

```
(%i1) bra([3/2,1/2]);
                                       3  1
(%o1)                                <[-, -]|
                                       2  2
(%i2) ket([3/2,1/2]);
                                       3  1
(%o2)                                |[-, -]>
                                       2  2
```

`jmketp` (*jmket*) [Function]

`jmketp` checks to see that the ket has an `m`-value that is in the set `{-j,-j+1,...,+j}`.

```
(%i1) jmketp(ket([j,m]));
(%o1)                                 false
(%i2) jmketp(ket([3/2,1/2]));
(%o2)                                 true
```

`jmbrap` (*jmbra*) [Function]

`jmbrap` checks to see that the bra has an `m`-value that is in the set `{-j,-j+1,...,+j}`.

`jmcheck (j,m)` [Function]

`jmcheck` checks to see that m is one of {-j, . . ., +j}.

```
(%i1) jmcheck(3/2,1/2);
(%o1)                               true
```

`JP` (*jmket*) [Function]

`JP` is the J_+ operator. It takes a `jmket` `jmket(j,m)` and returns `sqrt(j*(j+1)-m*(m+1))*hbar*jmket(j,m+1)`.

`JM` (*jmket*) [Function]

`JM` is the J_- operator. It takes a `jmket` `jmket(j,m)` and returns `sqrt(j*(j+1)-m*(m-1))*hbar*jmket(j,m-1)`.

`Jsqr` (*jmket*) [Function]

`Jsqr` is the J^2 operator. It takes a `jmket` `jmket(j,m)` and returns `(j*(j+1)*hbar`^{2}`*jmket(j,m)`.

`Jz` (*jmket*) [Function]

`Jz` is the J_z operator. It takes a `jmket` `jmket(j,m)` and returns `m*hbar*jmket(j,m)`.

These functions are illustrated below.

```
(%i1) k:ket([j,m]);
(%o1)                              |[j, m]>
(%i2) JP(k);
(%o2)            hbar |[j, m + 1]> sqrt(j (j + 1) - m (m + 1))
(%i3) JM(k);
(%o3)            hbar |[j, m - 1]> sqrt(j (j + 1) - (m - 1) m)
(%i4) Jsqr(k);
                                 2
(%o4)                      hbar  j (j + 1) |[j, m]>
(%i5) Jz(k);
(%o5)                            hbar |[j, m]> m
```

1.2.6 Angular momentum and ladder operators

`SP (s)` [Function]

`SP` is the raising ladder operator S_+ for spin `s`.

`SM (s)` [Function]

`SM` is the raising ladder operator S_- for spin `s`.

Examples of the ladder operators:

```
(%i1) SP(1);
                              [ 0  sqrt(2) hbar       0       ]
                              [                               ]
(%o1)                         [ 0       0       sqrt(2) hbar ]
                              [                               ]
                              [ 0       0             0       ]
(%i2) SM(1);
                              [      0            0         0 ]
                              [                               ]
(%o2)                         [ sqrt(2) hbar      0         0 ]
                              [                               ]
                              [      0       sqrt(2) hbar   0 ]
```

1.3 Rotation operators

`RX (s,t)` [Function]

`RX(s)` for spin `s` returns the matrix representation of the rotation operator `Rx` for rotation through angle `t`.

`RY (s,t)` [Function]

`RY(s)` for spin `s` returns the matrix representation of the rotation operator `Ry` for rotation through angle `t`.

`RZ (s,t)` [Function]

`RZ(s)` for spin `s` returns the matrix representation of the rotation operator `Rz` for rotation through angle `t`.

```
(%i1) RY(1,t);
Proviso: assuming 4*t # 0
                       [ cos(t) + 1    sin(t)   1 - cos(t) ]
                       [ ----------  - -------  ---------- ]
                       [     2         sqrt(2)      2      ]
                       [                                   ]
                       [  sin(t)                  sin(t)   ]
(%o1)                  [  -------     cos(t)    - -------  ]
                       [  sqrt(2)                 sqrt(2)  ]
                       [                                   ]
                       [ 1 - cos(t)   sin(t)    cos(t) + 1 ]
                       [ ----------   -------   ---------- ]
                       [     2        sqrt(2)       2      ]
```

1.4 Time-evolution operator

`UU (H,t)` [Function]

`UU(H,t)` is the time evolution operator for Hamiltonian `H`. It is defined as the matrix exponential `matrixexp(-%i*H*t/hbar)`.

```
(%i1) UU(w*Sy,t);
Proviso: assuming 64*t*w # 0
                              [     t w         t w  ]
                              [ cos(---)  - sin(---) ]
                              [      2           2   ]
(%o1)                         [                      ]
                              [     t w        t w   ]
                              [ sin(---)   cos(---)  ]
                              [      2          2    ]
```

1.5 Tensor products

Tensor products are represented as lists in the `qm` package. The ket tensor product `|z+,z+>` can be represented as `ket([u,d])`, for example, and the bra tensor product `<a,b|` is represented as `bra([a,b])` for states `a` and `b`. For a tensor product where the identity is one of the elements of the product, substitute the string `Id` in the ket or bra at the desired location. See the examples below for the use of the identity in tensor products.

Examples below show how to create abstract tensor products that contain the identity element `Id` and how to take the bracket of these tensor products.

```
(%i1) K:ket([a1,b1]);
(%o1)                           |[a1, b1]>
(%i2) B:bra([a2,b2]);
(%o2)                           <[a2, b2]|
(%i3) braket(B,K);
(%o3)                 kron_delta(a1, a2) kron_delta(b1, b2)
(%i1) bra([a1,Id,c1]) . ket([a2,b2,c2]);
(%o1)         |[-, b2, -]> kron_delta(a1, a2) kron_delta(c1, c2)
(%i2) bra([a1,b1,c1]) . ket([Id,b2,c2]);
(%o2)         <[a1, -, -]| kron_delta(b1, b2) kron_delta(c1, c2)
```

In the next example we construct the state function for an entangled Bell pair, construct the density matrix, and then trace over the first particle to obtain the density submatrix for particle 2.

```
(%i1) bell:(1/sqrt(2))*(ket([u,d])-ket([d,u]));
                               |[u, d]> - |[d, u]>
(%o1)                          -------------------
                                     sqrt(2)
(%i2) rho:bell . dagger(bell);
(%o2) (|[u, d]> . <[u, d]| - |[u, d]> . <[d, u]| - |[d, u]> . <[u, d]|
                                                   + |[d, u]> . <[d, u]|)/2
(%i3) assume(not equal(u,d));
(%o3)                           [notequal(u, d)]
(%i4) trace1:bra([u,Id]) . rho . ket([u,Id])+bra([d,Id]) . rho . ket([d,Id]);
                  |[-, u]> . <[-, u]|   |[-, d]> . <[-, d]|
(%o4)             ------------------- + -------------------
                           2                     2
```

Appendix A Function and Variable index