1.1 Introduction to package qm

The `qm` package was written by Eric Majzoub, University of Missouri (email: majzoube-at-umsystem.edu), with help from Robert Dodier and Barton Willis.

This purpose of this package is to provide computational tools for solving quantum mechanics problems in a finite-dimensional Hilbert space. It was written with students in mind and is appropriate for upper-level undergraduate quantum mechanics at the level of Townsend's *A Modern Introduction to Quantum Mechanics*. Please report any errors or unexpected behavior by submitting an issue on the Github page for this project.

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 such as computing the Clebsh-Gordon coefficients.

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].
```

This bra vector can be created with the `mbra` command

```
mbra([a,b])
```

or taking the quantum mechanical dagger of the corresponding ket. 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)
```

1.1.2 Special ket types

Some kets are difficult to work with using either the matrix representation or the general ket representation. These include tensor products of (j,m) kets used in the addition of angular momentum computations. For this reason there are a set of `tpkets` and associated `tpXX` functions defined in section `(j,m)-kets and bras`.

1.1.3 Types of spin operators

When working with kets and bras in the matrix representation, use the spin operators `Sxx`. When working with abstract kets and bras in the (j,m) representation use the operators `Jxx`. The family of `Sxx` operators are represented as matrices in Maxima, while the family of `Jxx` operators are rule based or function based.

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,...]) [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,...]) [Function]

`autoket` takes a list [a_1,a_2,...] 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 a bra `psi` and ket `phi`, `braket` returns the quantum mechanical bracket `<psi|phi>`.

```
(%i1) declare([a,b,c],complex);
(%o1)                               done
(%i2) braket(mbra([a,b,c]),mket([a,b,c]));
                                 2    2    2
(%o2)                           c  + b  + a
(%i3) braket(dagger(mket([a,b,c])),mket([a,b,c]));
(%o3)          c conjugate(c) + b conjugate(b) + a conjugate(a)
(%i4) braket(bra([a1,b1,c1]),ket([a2,b2,c2]));
(%o4)     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(mbra([a,b]),mket([c,d]));
(%o2)                             b d + a c
(%i3) P:magsqr(A);
(%o3) (b d + a c) (conjugate(b) conjugate(d) + conjugate(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(dagger(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(dagger(xp),psi)+'xm*braket(dagger(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 and ladder operators in the matrix representation

`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 ]
```

1.2.6 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      ]
```

1.2.7 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    ]
```

1.3 Angular momentum representation of kets and bras

1.3.1 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 $\mathtt{m_s}$, 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 $\mathtt{m_s}$, 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 ]
```

1.3.2 Angular momentum (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
```

Some convenience functions for making the kets are the following:

`jmtop (j)` [Function]

`jmtop` creates a (j,m)-ket with `m=j`.

```
(%i1) jmtop(3/2);
                                        3  3
(%o1)                                  |-, ->
                                        2  2
```

`jmbot (j)` [Function]

`jmbot` creates a (j,m)-ket with `m=-j`.

```
(%i1) jmbot(3/2);
                                      3   3
(%o1)                                |-, - ->
                                      2   2
```

jmket (j,m) [Function]

jmket creates a (j,m)-ket.

```
(%i1) jmket(3/2,1/2);
                                       3  1
(%o1)                                 |-, ->
                                       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
```

1.3.3 Addition of angular momentum in the (j,m)-representation

Addition of angular momentum calculations can be performed in the (j,m)-representation using the function definitions below. The internal representation of kets and bras for this purpose is the following. Given kets `|j1,m1>` and `|j2,m2>` a tensor product of (j,m)-kets is instantiated as:

```
[tpket,1,|j1,m1>,|j2,m2>]
```

and the corresponding bra is instantiated as:

```
[tpbra,1,<j1,m1|,<j2,m2|]
```

where the factor of 1 is the multiplicative factor of the tensor product. We call this the *common factor* (cf) of the tensor product. The general form of a tensor product in the (j,m) representation is:

`[tpket, cf, |j1,m1>, |j2,m2>]`.

Using the function definitions below one must be careful to avoid errors produced by Maxima's automatic list arithmetic. For example, do not use `(J1z+J2z)`, and instead use the defined function `Jtz`. Similarly for any of the operators that are added together, one should always use the total `Jtxx` defined function.

tpket (*jmket1,jmket2*) [Function]

`tpket` instantiates a tensor product of two (j,m)-kets.

```
(%i1) tpket(ket([3/2,1/2]),ket([1/2,1/2]));
                                        3  1     1  1
(%o1)                         [tpket, 1, |-, ->, |-, ->]
                                        2  2     2  2
```

tpbra (*jmbra1,jmbra2*) [Function]

`tpbra` instantiates a tensor product of two (j,m)-bras.

```
(%i1) tpbra(bra([3/2,1/2]),bra([1/2,1/2]));
                                        3  1     1  1
(%o1)                         [tpbra, 1, <-, -|, <-, -|]
                                        2  2     2  2
```

tpbraket (*tpbra,tpket*) [Function]

`tpbraket` returns the bracket of a `tpbra` and a `tpket`.

```
(%i1) k:tpket(jmtop(1),jmbot(1));
(%o1)                        [tpket, 1, |1, 1>, |1, - 1>]
(%i2) K:Jtsqr(k);
                    2                                            2
(%o2) [tpket, 2 hbar , |1, 1>, |1, - 1>] + [tpket, 2 hbar , |1, 0>, |1, 0>]
(%i3) B:tpdagger(k);
(%o3)                        [tpbra, 1, <1, 1|, <1, - 1|]
(%i4) tpbraket(B,K);
                                                 2
(%o4)                                      2 hbar
```

tpcfset (cf,*tpket*) [Function]

tpcfset manually sets the *common factor* cf of a tpket.

tpscmult (a,*tpket*) [Function]

tpscmult multiplies the tensor product's common factor by a.

```
(%i1) k1:tpket(ket([1/2,1/2]),ket([1/2,-1/2]));
                                          1  1    1    1
(%o1)                        [tpket, 1, |-, ->, |-, - ->]
                                          2  2    2    2
(%i2) tpscmult(c,k1);
                                          1  1    1    1
(%o2)                        [tpket, c, |-, ->, |-, - ->]
                                          2  2    2    2
```

tpadd (*tpket,tpket*) [Function]

tpadd adds two tpkets. This function is necessary to avoid trouble with Maxima's automatic list arithmetic.

```
(%i1) k1:tpket(ket([1/2,1/2]),ket([1/2,-1/2]));
                                          1  1    1    1
(%o1)                        [tpket, 1, |-, ->, |-, - ->]
                                          2  2    2    2
(%i2) k2:tpket(ket([1/2,-1/2]),ket([1/2,1/2]));
                                          1    1    1  1
(%o2)                        [tpket, 1, |-, - ->, |-, ->]
                                          2    2    2  2
(%i3) tpadd(k1,k2);
                         1  1    1    1                 1    1    1  1
(%o3)     [tpket, 1, |-, ->, |-, - ->] + [tpket, 1, |-, - ->, |-, ->]
                         2  2    2    2                 2    2    2  2
```

tpdagger (*tpket or tpbra*) [Function]

tpdagger takes the quantum mechanical dagger of a tpket or tpbra.

```
(%i1) k1:tpket(ket([1/2,1/2]),ket([1/2,-1/2]));
                                   1  1     1     1
(%o1)                   [tpket, 1, |-, ->, |-, - ->]
                                   2  2     2     2
(%i2) tpdagger(k1);
                                   1  1     1     1
(%o2)                   [tpbra, 1, <-, -|, <-, - -|]
                                   2  2     2     2
```

J1z (*tpket*) [Function]

J1z returns the tensor product of a tpket with Jz acting on the first ket.

J2z (*tpket*) [Function]

J2z returns the tensor product of a tpket with Jz acting on the second ket.

```
(%i1) k:tpket(ket([3/2,3/2]),ket([1/2,1/2]));
                                    3  3     1  1
(%o1)                    [tpket, 1, |-, ->, |-, ->]
                                    2  2     2  2
(%i2) J1z(k);
                               3 hbar   3  3     1  1
(%o2)                  [tpket, ------, |-, ->, |-, ->]
                                 2      2  2     2  2
(%i3) J2z(k);
                                hbar   3  3     1  1
(%o3)                   [tpket, ----, |-, ->, |-, ->]
                                 2     2  2     2  2
```

Jtz (*tpket*) [Function]

Jtz is the total z-projection of spin operator acting on a tpket and returning (J_{1z}+J_{2z}).

```
(%i1) k:tpket(ket([3/2,3/2]),ket([1/2,1/2]));
                                    3  3     1  1
(%o1)                    [tpket, 1, |-, ->, |-, ->]
                                    2  2     2  2
(%i2) Jtz(k);
                                       3  3     1  1
(%o2)                  [tpket, 2 hbar, |-, ->, |-, ->]
                                       2  2     2  2
```

J1sqr (*tpket*) [Function]

J1sqr returns Jsqr for the first ket of a tpket.

J2sqr (*tpket*) [Function]

J2sqr returns Jsqr for the second ket of a tpket.

J1p (*tpket*) [Function]

J1p returns J_+ for the first ket of a tpket.

`J2p` (*tpket*) [Function]

`J2p` returns J_+ for the second ket of a tpket.

`Jtp` (*tpket*) [Function]

`Jtp` returns $(J_{1+}+J_{2+})$ for the tpket.

`J1m` (*tpket*) [Function]

`J1m` returns J_- for the first ket of a tpket.

`J2m` (*tpket*) [Function]

`J2m` returns $J_=$ for the second ket of a tpket.

`Jtm` (*tpket*) [Function]

`Jtm` returns $(J_{1-}+J_{2-})$ for the tpket.

`J1p2m` (*tpket*) [Function]

`J1p2m` returns $(J_{1+}J_{2-})$ for the tpket.

```
(%i1) k:tpket(ket([3/2,1/2]),ket([1/2,1/2]));
                                          3   1     1   1
(%o1)                        [tpket, 1, |-, ->, |-, ->]
                                          2   2     2   2
(%i2) b:tpdagger(k);
                                          3   1     1   1
(%o2)                        [tpbra, 1, <-, -|, <-, -|]
                                          2   2     2   2
(%i3) J1p2m(k);
                                           2    3   3     1    1
(%o3)                 [tpket, sqrt(3) hbar , |-, ->, |-, - ->]
                                                2   2     2    2
(%i4) J1m2p(k);
(%o4)                                      0
```

`J1m2p` (*tpket*) [Function]

`J1m2p` returns $(J_{1-}J_{2+})$ for the tpket.

`J1zJ2z` (*tpket*) [Function]

`J1zJ2z` returns $(J_{1z}J_{2z})$ for the tpket.

`Jtsqr` (*tpket*) [Function]

`Jtsqr` returns $(J_1{}^2+J_2{}^2+ J_{1+}J_{2-}+J_{1-}J_{2+}+J_{1z}J_{2z})$ for the tpket.

```
(%i1) k:tpket(ket([3/2,-1/2]),ket([1/2,1/2]));
                               3    1    1   1
(%o1)                [tpket, 1, |-, - ->, |-, ->]
                               2    2    2   2
(%i2) B:tpdagger(k);
                               3    1    1   1
(%o2)                [tpbra, 1, <-, - -|, <-, -|]
                               2    2    2   2
(%i3) K2:Jtsqr(k);
                   2   3    1    1  1                   2   3  1    1    1
(%o3) [tpket, 4 hbar , |-, - ->, |-, ->] + [tpket, 2 hbar , |-, ->, |-, - ->]
                       2    2    2  2                       2  2    2    2
(%i4) tpbraket(B,K2);
                                          2
(%o4)                               4 hbar
```

get_j (q) [Function]

get_j is a convenience function that computes j from j(j+1)=q where q is a rational number. This function is useful after using the function Jtsqr.

```
(%i1) get_j(15/4);
                                        3
(%o1)                               j = -
                                        2
```

1.3.3.1 Example computations

For the first example, let us see how to determine the total spin state |j,m> of the two-particle state |1/2,1/2;1,1>.

```
(%i1) k:tpket(jmtop(1/2),jmtop(1));
                                1  1
(%o1)                 [tpket, 1, |-, ->, |1, 1>]
                                2  2
(%i2) Jtsqr(k);
                                2
                         15 hbar    1  1
(%o2)             [tpket, --------, |-, ->, |1, 1>]
                            4       2  2
(%i3) get_j(15/4);
                                        3
(%o3)                               j = -
                                        2
```

This is an eigenket of Jtsqr, thus |3/2,3/2> = |1/2,1/2;1,1>, and it is also the top state. One can now apply the lowering operator to find the other states: |3/2,1/2>, |3/2,-1/2>, and |3/2,-3/2>.

```
(%i1) k:tpket(jmtop(1/2),jmtop(1));
                                            1   1
(%o1)                      [tpket, 1, |-, ->, |1, 1>]
                                            2   2
(%i2) k2:Jtm(k);
                                    1   1                              1     1
(%o2) [tpket, sqrt(2) hbar, |-, ->, |1, 0>] + [tpket, hbar, |-, - ->, |1, 1>]
                                    2   2                              2     2
(%i3) k3:Jtm(k2);
                3/2      2    1     1
(%o3) [tpket, 2    hbar , |-, - ->, |1, 0>]
                              2     2
                                                             2    1   1
                                       + [tpket, 2 hbar , |-, ->, |1, - 1>]
                                                              2   2
(%i4) k4:Jtm(k3);
                                      3    1     1
(%o4)                  [tpket, 6 hbar , |-, - ->, |1, - 1>]
                                           2     2
```

Let us see how to compute the matrix elements of the operator `(J1z-J1z)` in the z-basis for two spin-1/2 particles. Note that we use the `tpadd` and `tpscmult` functions to add the two operators. First, we form the four basis kets {phi_1,phi_2,phi_3,phi_4} of the form $|j_1,m_1;j_2,m_2>$. The next four entries are for the operator acting on the basis kets. We skip taking the braket below; the common factor is the resulting matrix element.

```
(%i1) phi1:tpket(ket([1/2,1/2]),ket([1/2,1/2]));
                                        1  1    1  1
(%o1)                    [tpket, 1, |-, ->, |-, ->]
                                        2  2    2  2
(%i2) phi2:tpket(ket([1/2,1/2]),ket([1/2,-1/2]));
                                       1  1    1    1
(%o2)                   [tpket, 1, |-, ->, |-, - ->]
                                       2  2    2    2
(%i3) phi3:tpket(ket([1/2,-1/2]),ket([1/2,1/2]));
                                       1    1    1  1
(%o3)                   [tpket, 1, |-, - ->, |-, ->]
                                       2    2    2  2
(%i4) phi4:tpket(ket([1/2,-1/2]),ket([1/2,-1/2]));
                                      1    1    1    1
(%o4)                  [tpket, 1, |-, - ->, |-, - ->]
                                      2    2    2    2
(%i5) tpadd(J1z(phi1),tpscmult(-1,J2z(phi1)));
(%o5)                                    0
(%i6) tpadd(J1z(phi2),tpscmult(-1,J2z(phi2)));
                                         1  1    1    1
(%o6)                  [tpket, hbar, |-, ->, |-, - ->]
                                         2  2    2    2
(%i7) tpadd(J1z(phi3),tpscmult(-1,J2z(phi3)));
                                          1    1    1  1
(%o7)                 [tpket, - hbar, |-, - ->, |-, ->]
                                          2    2    2  2
(%i8) tpadd(J1z(phi4),tpscmult(-1,J2z(phi4)));
(%o8)                                    0
```

In the example below we calculate the Clebsh-Gordon coefficients of the two-particle state with two spin-1/2 particles. We begin by defining the top rung of the ladder and stepping down.

```
(%i1) top:tpket(jmtop(1/2),jmtop(1/2));
                                      1  1    1  1
(%o1)                     [tpket, 1, |-, ->, |-, ->]
                                      2  2    2  2
(%i2) Jtsqr(top);
                                     2   1  1    1  1
(%o2)                  [tpket, 2 hbar , |-, ->, |-, ->]
                                         2  2    2  2
(%i3) get_j(2);
(%o3)                                j = 1
(%i4) Jtz(top);
                                        1  1    1  1
(%o4)                    [tpket, hbar, |-, ->, |-, ->]
                                        2  2    2  2
(%i5) JMtop:ket([1,1]);
(%o5)                               |1, 1>
(%i6) mid:Jtm(top);
                      1  1    1    1                    1    1    1  1
(%o6)  [tpket, hbar, |-, ->, |-, - ->] + [tpket, hbar, |-, - ->, |-, ->]
                      2  2    2    2                    2    2    2  2
(%i7) Jm(JMtop);
(%o7)                         sqrt(2) |1, 0> hbar
(%i8) mid:tpscmult(1/(sqrt(2)*hbar),mid);
                 1      1  1    1    1                1      1    1    1  1
(%o8) [tpket, -------, |-, ->, |-, - ->] + [tpket, -------, |-, - ->, |-, ->]
              sqrt(2)   2  2    2    2             sqrt(2)   2    2    2  2
(%i9) bot:Jtm(mid);
                                          1    1    1    1
(%o9)              [tpket, sqrt(2) hbar, |-, - ->, |-, - ->]
                                          2    2    2    2
(%i10) Jm(ket([1,0]));
(%o10)                       sqrt(2) |1, - 1> hbar
(%i11) bot:tpscmult(1/(sqrt(2)*hbar),bot);
                                    1    1    1    1
(%o11)                  [tpket, 1, |-, - ->, |-, - ->]
                                    2    2    2    2
```

1.4 General 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);
      |u, d> . <u, d| - |u, d> . <d, u| - |d, u> . <u, d| + |d, u> . <d, u|
(%o2) ---------------------------------------------------------------------
                                        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
```

1.5 Quantum harmonic oscillator

The `qm` package can perform simple quantum harmonic oscillator calculations involving the ladder operators `a+` and `a-`. These are referred to in the package as `ap` and `am` respectively. For computations with arbitrary states to work you must `declare` the harmonic oscillator state, say `n`, to be both `scalar` and `integer`, as shown in the examples below.

ap [Function]

`ap` is the raising operator `a+` for quantum harmonic oscillator states.

am [Function]

`a` is the lowering operator `a-` for quantum harmonic oscillator states.

A common problem is to compute the 1st order change in energy of a state due to a perturbation of the harmonic potential, say an additional factor `V(x) = x^2 + g*x^4` for small `g`. This example is performed below, ignoring any physical constants in the problem.

```
(%i1) declare(n,integer,n,scalar);
(%o1)                                done
(%i2) ap . ket([n]);
(%o2)                         sqrt(n + 1) |n + 1>
(%i3) am . ket([n]);
(%o3)                           |n - 1> sqrt(n)
(%i4) bra([n]) . (ap+am)^^4 . ket([n]);
                                    2
(%o4)                           6 n  + 6 n + 3
```

Another package that handles quantum mechanical operators is `operator_algebra` written by Barton Willis.

Appendix A Function and Variable index

T

U

X

Y

Z