## 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 the built-in definition 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 states 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. It is also possible to create tensor product states for multiparticle systems and 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 will be represented in Maxima by the row vector `[a b]`, where the basis vectors are

`<z+| = [1 0]`

and

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

There are two types of kets and bras available in this package, the first type is given by a *matrix representation*, as in the above example. `ket`s are column vectors and `bra`s are row vectors, and their components are entered as Maxima *lists* in the `bra` and `ket` functions. The second type of bra or ket is *abstract*; there is no matrix representation. Abstract bras and kets are entered *without* using lists for the elements. Thus, if one wishes to do purely symbolic calculations, then input of abstract kets, (j,m)-kets, and so forth should be done without lists. If one wishes to do numerical or component computations using the kets then enter the arguments as a list. Note that abstract kets and bras are *assumed to be orthonormal.*

The following examples illustrate some of the basic capabilities of the `qm` package. Here both abstract, and concrete kets are shown.

```
(%i1) ket(a,b)+ket(c,d);
(%o1)                          |c, d> + |a, b>
(%i2) ket([a,b,c])+ket([d,e,f]);
                                  [ d + a ]
                                  [       ]
(%o2)                             [ e + b ]
                                  [       ]
                                  [ f + c ]
```

Next, tensor products of the spin-1/2 basis states `{zp,zm}` are shown in abstract and matrix representations.

```
(%i1) ketprod('zp,'zm)+ketprod('zm,'zp);
(%o1)                  ketprod(zp, zm) + ketprod(zm, zp)
(%i2) ketprod([zp,zm]);
                                       [ 1 ]  [ 0 ]
(%o2)                        ketprod([[     ], [     ]])
                                       [ 0 ]  [ 1 ]
```

Abstract kets and bras are assumed to be orthonormal as shown in the following examples.

```
(%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)
(%i1) declare([c1,c2],complex,r,real);
(%o1)                                done
(%i2) k:ket([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` ([$c_1$,$c_2$,...]) [Function]

`ket` 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. If no list is entered the ket is represented as a general ket, `ket(a)` will return `|a>`.

```
(%i1) kill(a);
(%o1)                                done
(%i2) ket(a);
(%o2)                                 |a>
(%i3) declare([c1,c2],complex);
(%o3)                                done
(%i4) ket([c1,c2]);
                                    [ c1 ]
(%o4)                               [    ]
                                    [ c2 ]
(%i5) facts();
(%o5) [kind(hbar, real), hbar > 0, kind(c1, complex), kind(c2, complex)]
```

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

`bra` 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 bracbras. If no list is entered the bra is represented as a general bra, `bra(a)` will return `<a|`.

```
(%i1) kill(c1,c2);
(%o1)                                done
(%i2) bra(c1,c2);
(%o2)                              <c1, c2|
(%i3) bra([c1,c2]);
(%o3)                             [ c1  c2 ]
(%i4) facts();
(%o4)                      [kind(hbar, real), hbar > 0]
```

`ketp` (*vector*) [Function]

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

```
(%i1) kill(a,b,k);
(%o1)                                done
(%i2) k:ket(a,b);
(%o2)                               |a, b>
(%i3) ketp(k);
(%o3)                                false
(%i4) k:ket([a,b]);
                                     [ a ]
(%o4)                                [   ]
                                     [ b ]
(%i5) ketp(k);
(%o5)                                true
```

`brap` (*vector*) [Function]

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

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

Two additional functions are provided to create kets and bras in the matrix representation. Additionally these functions 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.

```
(%i1) dagger(bra([%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(ket([a,b,c]),ket([a,b,c]));
(%o2)          c conjugate(c) + b conjugate(b) + a conjugate(a)
```

`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(ket([a,b,c]));
(%o2)       sqrt(c conjugate(c) + b conjugate(b) + a conjugate(a))
(%i3) norm(ket(a,b,c));
(%o3)                           norm(|a, b, c>)
```

`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(ket([a,b]),ket([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 Handling general kets and bras

General kets and bras are, as discussed, created without using a list when giving the arguments. The following examples show how general kets and bras can be manipulated.

```
(%i1) ket(a)+ket(b);
(%o1)                              |b> + |a>
(%i2) braket(bra(a),ket(b));
(%o2)                          kron_delta(a, b)
(%i3) braket(bra(a)+bra(c),ket(b));
(%o3)                 kron_delta(b, c) + kron_delta(a, b)
```

### 1.2.2 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:ket([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)
```

### 1.2.3 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   ]
```

### 1.2.4 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)          ]
```

### 1.2.5 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
```

### 1.2.6 Angular momentum representation of kets and bras

To create kets and bras in the |*j*,*m*> representation you can use the following functions.

`jmket (j,m)` [Function]

`jmket` creates the ket |*j*,*m*> for total spin *j* and *z*-component *m*.

`jmbra (j,m)` [Function]

`jmbra` creates the bra <*j*,*m*| for total spin *j* and *z*-component *m*.

```
(%i1) jmbra(3/2,1/2);
                                         3  1
(%o1)                              jmbra(-, -)
                                         2  2
(%i2) jmbra([3/2,1/2]);
                                        [ 3  1 ]
(%o2)                           [jmbra, [ -  - ]]
                                        [ 2  2 ]
```

`jmketp (`*jmket*`)` [Function]

`jmketp` checks to see that the ket has the 'jmket' marker.

```
(%i1) jmketp(jmket(j,m));
(%o1)                                  false
(%i2) jmketp(jmket([j,m]));
(%o2)                                  true
```

`jmbrap (`*jmbra*`)` [Function]

`jmbrap` checks to see that the bra has the 'jmbra' marker.

`jmcheck (j,m)` [Function]

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

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

`jmbraket` (*jmbra,jmket*) [Function]

`jmbraket` takes the inner product of the jm-kets.

```
(%i1) K:jmket(j1,m1);
(%o1)                              jmket(j1, m1)
(%i2) B:jmbra(j2,m2);
(%o2)                              jmbra(j2, m2)
(%i3) jmbraket(B,K);
(%o3)                kron_delta(j1, j2) kron_delta(m1, m2)
(%i4) B:jmbra(j1,m1);
(%o4)                              jmbra(j1, m1)
(%i5) jmbraket(B,K);
(%o5)                                    1
(%i6) K:jmket([j1,m1]);
(%o6)                           [jmket, [ j1  m1 ]]
(%i7) B:jmbra([j2,m2]);
(%o7)                           [jmbra, [ j2  m2 ]]
(%i8) jmbraket(B,K);
(%o8)                                    0
(%i9) jmbraket(jmbra(j1,m1)+jmbra(j3,m3),jmket(j2,m2));
(%o9) kron_delta(j2, j3) kron_delta(m2, m3)
                                   + kron_delta(j1, j2) kron_delta(m1, m2)
```

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

### 1.2.7 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) RZ(1/2,t);
Proviso: assuming 64*t # 0
                                   [     %i t         ]
                                   [   - ----         ]
                                   [      2           ]
                                   [ %e          0    ]
(%o1)                              [                  ]
                                   [            %i t  ]
                                   [            ----  ]
                                   [             2    ]
                                   [    0     %e      ]
```

## 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 Maxima. The ket tensor product `|z+,z+>` is represented as `[tpket,zp,zp]`, and the bra tensor product `<a,b|` is represented as `[tpbra,a,b]` for kets `a` and `b`. The list labels `tpket` and `tpbra` ensure calculations are performed with the correct kind of objects.

`tpket` ($k_1$, $k_2$, ...) [Function]

`tpket` produces a tensor product of kets $k_i$. All of the elements must pass the `ketp` predicate test to be accepted.

`tpbra` ($b_1$, $b_2$, ...) [Function]

`tpbra` produces a tensor product of bras $b_i$. All of the elements must pass the `brap` predicate test to be accepted.

`tpketp` (*tpket*) [Function]

`tpketp` checks to see that the ket has the 'tpket' marker.

`tpbrap` (*tpbra*) [Function]

`tpbrap` checks to see that the bra has the 'tpbra' marker.

`tpbraket (B,K)` [Function]

`tpbraket` takes the inner product of the tensor products `B` and `K`. The tensor products must be of the same length (number of kets must equal the number of bras).

Examples below show how to create tensor products and take the bracket of tensor products.

```
(%i1) tpket(zp,zm);
                                     [ 1 ]  [ 0 ]
(%o1)                          tpket([   ], [   ])
                                     [ 0 ]  [ 1 ]
(%i2) tpket('zp,'zm);
(%o2)                            tpket(zp, zm)
(%i3) tpket([zp,zm]);
                                      [ 1 ]  [ 0 ]
(%o3)                        [tpket, [[   ], [   ]]]
                                      [ 0 ]  [ 1 ]
(%i1) kill(a,b,c,d);
(%o1)                                done
(%i2) declare([a,b,c,d],complex);
(%o2)                                done
(%i3) tpbra([bra([a,b]),bra([c,d])]);
(%o3)                     [tpbra, [[ a  b ], [ c  d ]]]
(%i4) tpbra([dagger(zp),bra([c,d])]);
(%o4)                     [tpbra, [[ 1  0 ], [ c  d ]]]
(%i1) K:tpket([zp,zm]);
                                      [ 1 ]  [ 0 ]
(%o1)                        [tpket, [[   ], [   ]]]
                                      [ 0 ]  [ 1 ]
(%i2) zpb:dagger(zp);
(%o2)                              [ 1  0 ]
(%i3) zmb:dagger(zm);
(%o3)                              [ 0  1 ]
(%i4) B:tpbra([zpb,zmb]);
(%o4)                     [tpbra, [[ 1  0 ], [ 0  1 ]]]
(%i5) tpbraket(K,B);
(%o5)                                false
(%i6) tpbraket(B,K);
(%o6)                                  1
```

# Appendix A Function and Variable index