A Quantum Optics Toolbox for Matlab 5

Sze Meng Tan
The University of Auckland
Private Bag 92019, Auckland
New Zealand

Table of Contents

Introduction

In quantum optics, it is often necessary to simulate the equations of motion of a system coupled to a reservoir. Using a Schrödinger picture approach, this can be done either by integrating the master equation for the density matrix[1] or by using some state-vector based approach such as the quantum Monte Carlo technique[2][3]. Starting from the Hamiltonian of the system and the coupling to the baths, it is in principle a simple process to derive either the master equation or the stochastic Schrödinger equation. In practice, however for all but the simplest systems, this process is tedious and can be error-prone. For a system which is described by an n dimensional Hilbert space, there are n simultaneous complex-valued Schrödinger equations of motion and n^2 simultaneous real-valued equations of motion for the components of the Hermitian density matrix. Although the matrix of coefficients is the square of the number of simultaneous equations (i.e. n^2 for the Schrödinger equation and n^4 for the density matrix equations), many of these coefficients are zero and it is feasible to integrate systems of 10^2 to 10^3 equations numerically.

The recent interest in quantum systems involving only a few photons and atoms in cavity QED systems and in quantum logic devices makes it often possible to employ a truncated Fock space basis for the light field modes, yielding essentially exact numerical simulations. This document addresses the problem of semi-automatically generating the equations of motion for a wide variety of quantum optical systems working directly from the Hamiltonian and couplings to the bath. This approach substantially reduces the algebraic manipulations required, allowing a variety of configurations to be investigated rapidly.

The Matlab programming language[4] is used to set up the equations of motion. Matlab supports the manipulation of complex valued matrices as primitive data objects. This is particularly convenient for representing quantum mechanical operators taken with respect to some basis. We shall see that the sparse matrix facilities built into the Matlab language allow efficient computation with these quantities. Once the equations of motion have been derived, a variety of solution techniques may be employed. For small problems with constant Liouvillians for which the eigenvectors and eigenvalues may be computed explicitly, the steady-state and time-dependent solutions of the master equation can be found in forms which may be evaluated at arbitrary times without step-by-step numerical integration. For Liouvillians with special time-dependences, techniques such as matrix continued fractions may be used to find periodic solutions of the master equation which are useful in problems such as calculating the forces on atoms in standing light fields. When direct numerical integration cannot be avoided due to the size of the state space or non-trivial time-dependence in the Hamiltonian, a selection of numerical differential equation solvers written in C may be used to carry out the solution after the problem has been formulated in Matlab.

In the next section, the overall philosophy of the toolbox is described and the basic "quantum object" data structures are introduced. Quantum objects are generic containers for scalars, states, operators and superoperators and thus possess a fairly rich structure. This will be elaborated upon and illustrated through a series of problems which demonstrate how the toolbox may be used.

Installation

To install the toolbox on an IBM compatible PC running Windows, unpack the files into some convenient directory, such as `c:\qotoolbox`. The executable files `_solvemc.exe`, `_stochsim.exe` and `_solvesde.exe` and the associated batch files `solvemc.bat`, `stochsim.bat` and `solvesde.bat` which are in the `bin` subdirectory must be copied to a directory which is on the *system* path (set up in the `autoexec.bat` file, and which is usually distinct from the Matlab path). Unless this is done, the numerical integration routines will not operate correctly.

On a machine running Unix or other operating system, it is necessary to make the executable files `solvemc`, `stochsim` and `solvesde` from the files contained within subdirectories of the `unixsrc` directory. Simply compile and link all the C files contained in each of the subdirectories to produce the appropriate executable files and ensure that these executables are placed in a directory on the path. For example, if `/home/mydir/bin` is on the path, in order to compile `solvemc` using the Gnu C compiler, one should change to the subdirectory `unixsrc/solvemc` and issue a command such as

```
gcc -o /home/mydir/bin/solvemc *.c -lm
```

After starting Matlab, add the directory containing the toolbox to the path by entering a command such as `addpath('c:/qotoolbox')`. It should now be possible to access the toolbox files. In order to try out the examples, one should `addpath('c:/qotoolbox/examples')` as well. If you wish these directories to be added to your Matlab path automatically, it will be necessary to use a `startup.m` file, or to edit the system-wide default path in the `matlab/toolbox/local` directory.

Run the script `qdemos` to view a series of demonstrations contained within the examples directory. The buttons with an asterisk in the names indicate those demonstrations which require the numerical integration routines to be correctly installed. These demonstrations are discussed in more detail in this document, where they are referred to in terms of the script files which run them. In order to show these script file names rather than the descriptions of the routines in the buttons of the demonstration, turn on the checkbox labelled "Show script names". When running these demonstrations, prompts in the command window will lead the user through each example. The menu window will be hidden until while each demonstration is in progress.

Note that the files in the `examples` directory beginning with the letter `x` are script files which may also be run manually from the command line. Usually the file `xprob`*yyy* calls a function file named `prob`*yyy* which is discussed in the notes.

This article is presented as a tutorial and the reader is encouraged to try out the examples if a computer is available. A reference manual with more complete details of the implementation is currently under preparation. Lines which are in `typewriter` font and preceeded by a `>>` may be typed in at the Matlab prompt.

Quantum Objects

Within the toolbox, the basic data type is a "quantum array" object which, as its name suggests, is a collection of one or more simple "quantum objects". Each quantum object may represent a vector, operator or super-operator over some Hilbert space representing the state space of the problem. In the computer, the members of a quantum array object are represented as complex-valued vectors or matrices. We shall use the terms "element" and "matrix" to refer to the representation of the individual quantum objects and the terms "member" and "array" to refer to the organization of these objects within a quantum array object.

The usual arithmetic operations are overloaded so that quantum array objects may be combined whenever this combination makes sense. Furthermore, many of the toolbox functions are polymorphic, so that, for example, if a state is required, this state may be specified either as a ket vector or as a density matrix. In the following sections, we shall introduce the structure of a quantum object and show how several of these may be collected into an array.

1. The representation of pure states

Given a single quantum system such as a mode of a quantized light field or the internal dynamics of an atom, a state ket can be represented by a column vector whose components give the expansion coefficients of the state with respect to some basis. In Matlab, a column vector with n components is written as `[c1;c2;...;cn]` where the semicolons separate the rows. In order to create a state within the toolbox and assign it to a variable `psi`, we would enter a command such as

```
>> psi = qo([0.8;0;0;0.6]);
```

The function `qo` packages the column vector into as a quantum object. It is technically called a **constructor** for the class `qo`. As a result of the assignment, `psi` is now a simple quantum object (i.e., a quantum array object with a single member). If one now types `psi` at the prompt, the response is

```
>> psi
psi = Quantum object
Hilbert space dimensions [ 4 ] by [ 1 ]
    0.8000
         0
         0
    0.6000
```

Within the quantum object, additional information is stored together with the elements of the vector. In the example, this additional information was deduced from the size of the input argument to `qo`. More precisely, an object of type `qo` contains the following fields:

`dims`	Hilbert space dimensions of each object in the array
`size`	Size of the array, specifying the number of members
`shape`	Shape of each object in the array as a two-dimensional matrix
`data`	Data for the quantum object stored as a "flattened" two-dimensional matrix

For the most part, the user need not be too concerned about manipulating these fields, as they are handled by the toolbox routines. In order to examine these fields, one may enter

```
>> psi.dims
    [4]    [1]
>> psi.size
     1     1
>> psi.shape
     4     1
>> psi.data
   (1,1)       0.8000
   (4,1)       0.6000
```

The user can examine, but not modify the contents of these fields. The `dims` field is the cell array `{[4],[1]}`, which means that the objects are matrices of size 4×1, which are column vectors. The `size` field is `[1,1]` since there is only one object in the array. The `shape` field indicates that each object is of size 4×1, which at this stage may appear to duplicate the information in the `dims` field, but the distinction will become clearer as we proceed. Finally the data themselves, i.e., the numbers 0.8, 0.0, 0.0 and 0.6, are stored as a single column in the `data` field as a sparse matrix (i.e., only the non-zero elements are stored explicitly). Note that it is also possible to use `dims(psi)`, `size(psi)` and `shape(psi)` to access the above information.

In order to produce a unit ket in an N dimensional Hilbert space, the toolbox function `basis(N,indx)` creates a quantum object with a single one in the component specified by `indx`. Thus we have, for example,

```
>> basis(4,2)
ans = Quantum object
Hilbert space dimensions [ 4 ] by [ 1 ]
     0
     1
     0
     0
```

2. Tensor product of spaces

Often, we are concerned with systems which are composed of two or more subsystems. Suppose there are two subsystems for which the dimension of the state space for the first alone is m and that of the second alone is n. The dimension of the Hilbert space for the composite system is mn. If the first system is prepared in state `c` = [`c1`; ...; `cm`] and the second system is independently prepared in the state `d` = [`d1`; ...; `dn`], the joint state is given by the tensor product of the states which has components

```
[c1*d1; ... ; c1*dn; c2*d1; ... ; c2*dn; ... ; cm*d1; ... ; cm*dn]
```

Within the toobox, the construction of the tensor product is carried out as shown in the following example:

```
>> psi1 = qo([0.6; 0.8]);
>> psi2 = qo([0.8; 0.4; 0.2; 0.4]);
>> psi = tensor(psi1,psi2)
psi = Quantum object
Hilbert space dimensions [ 2 4 ] by [ 1 1 ]
     0.4800
     0.2400
     0.1200
     0.2400
     0.6400
     0.3200
     0.1600
     0.3200
```

Notice that the `dims` field of the composite object has been set to `{[2;4],[1;1]}` since this is the tensor product of an object of dimensions `{[2],[1]}` and one of dimensions `{[4],[1]}`. By keeping a record of the component spaces which are combined together, it becomes possible to carry out calculations such as partial traces (as described later) as well as to check whether operations are being carried out on compatible objects. The `shape` field of the composite object is `[8,1]` which is the shape of the resulting ket vector.

The `tensor` function may be used with more than two input arguments if desired in order to form objects for systems with more components.

3. The representation of operators

Quantum mechanical operators are represented with respect to a basis by matrices in the usual way. It is possible to construct operators using the `qo` constructor directly. For example

```
>> A = qo([1,2,3;2,5,6;3,6,9])
A = Quantum object
Hilbert space dimensions [ 3 ] by [ 3 ]
     1     2     3
     2     5     6
     3     6     9
```

The dimensions of the Hilbert space are obtained from the size of the matrix specified. If desired, one can specify the Hilbert space dimensions explicitly using a second argument to the constructor. For example

```
>> A = qo(randn(6,6),{[3;2],[3;2]});
```

generates a random 6×6 matrix, but specifies that the dimensions are to be taken as `[3;2]` by `[3;2]` rather than as `6` by `6`, which would have been assigned to the matrix by default.

Since several operators are used extensively in quantum optics, functions which generate them are built into the toolbox. For example, the annihilation operator for a single bosonic mode has the Fock-space representation

$$\langle m|a|n\rangle = \sqrt{n}\delta_{m,n-1}$$

which can be represented by a sparse matrix with entries $\sqrt{n}$ on the first subdiagonal. The toolbox function `destroy(N)` produces a quantum object whose data are a sparse $N \times N$ matrix representing this operator trucated to the Fock space consisting of states with zero to $N-1$ bosons. In order to produce the creation operator, it is only necessary to calculate the conjugate transpose, denoted in Matlab by the apostrophe. Thus `destroy(N)'` is the creation operator in the same space, which may also be produced by using `create(N)`.

Operators for the angular momentum algebra may also be generated using the toolbox function `jmat(j,type)`. The `type` argument may be one of the strings `'x'`, `'y'`, `'z'`, `'+'` or `'-'` while the argument `j` is an integer or half-integer. These satisfy the following relations

$$[J_x, J_y] = iJ_z \text{ et cyc.}, \; J_\pm = J_x \pm iJ_y$$

The resulting matrix is of size $(2j+1) \times (2j+1)$ and the matrix elements are given in units of $\hbar$, so that for example, we have

```
>> jmat(1,'x')
ans = Quantum object
Hilbert space dimensions [ 3 ] by [ 3 ]
         0    0.7071         0
    0.7071         0    0.7071
         0    0.7071         0
```

Note that the matrix is stored internally in sparse format. In order to extract the data portion of the quantum object as an ordinary double matrix, the function `double` may be used. This returns a sparse matrix which can be converted into a full matrix using the `full` function. Thus we have:

```
>> full(double(jmat(1,'x')))
ans =
         0    0.7071         0
    0.7071         0    0.7071
         0    0.7071         0
```

It is also possible to obtain the operator for $\hat{\mathbf{n}} \cdot \mathbf{J}$ where $\mathbf{n}$ is a three component vector specifying a direction by specifying this vector in place of the `type` string as the second argument of `jmat`. The vector $\mathbf{n}$ is normalized to unit length, and the operator returned is $\hat{n}_x J_x + \hat{n}_y J_y + \hat{n}_z J_z$.

The Pauli spin operators are of special significance since they may be used to represent a two-level system. The convention we use is to define

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = 2J_x^{(1/2)},\ \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = 2J_y^{(1/2)},\ \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = 2J_z^{(1/2)}$$

which are obtained by using `sigmax`, `sigmay` and `sigmaz` while

$$\sigma_- = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \text{ and } \sigma_+ = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

are generated using `sigmam` and `sigmap`. Note that $\sigma_\pm = \frac{1}{2}(\sigma_x \pm i\sigma_y)$, which is somewhat inconsistent with the definition $J_\pm = J_x \pm iJ_y$ used for the general angular momentum operators.

The function `tensor` described above is useful for constructing operators in a joint space as it is for constructing states. Consider a specific example of a cavity supporting mode a in which a single two-level atom is placed. If we truncate the space of the light field to be N dimensional, the following lines of code define operators which act on the space of the joint system:

```
ida = identity(N); idat = identity(2);
a = tensor(destroy(N),idat);
sm = tensor(ida,sigmam);
```

In the first line, identity operators are defined for the space of the light field and the space of the atom. The annihilation operator for the light field does not affect the space of the atom, and so `a` is defined with the identity operator in the atomic slot. Similarly the atomic lowering operator `sm` is defined with the identity in the light field slot.

Let us consider constructing the Hamiltonian operator for the above system driven by an external classical driving field.

$$H = \omega_0 \sigma_+ \sigma_- + \omega_c a^\dagger a + ig\left(a^\dagger \sigma_- - \sigma_+ a\right) + \mathcal{E}\left(e^{-i\omega_L t} a^\dagger + e^{i\omega_L t} a\right)$$

where ω_0 is the atomic transition angular frequency, ω_c is the cavity resonant angular frequency and ω_L is the angular frequency of the classical driving field, and we have taken $\hbar = 1$. Moving to an interaction picture rotating at the driving field frequency yields

$$H = (\omega_0 - \omega_L)\sigma_+ \sigma_- + (\omega_c - \omega_L) a^\dagger a + ig\left(a^\dagger \sigma_- - \sigma_+ a\right) + \mathcal{E}\left(a^\dagger + a\right)$$

with the Matlab definitions given above, the operator H is simply given by

```
H=(w0-wL)*sm'*sm + (wc-wL)*a'*a + i*g*(a'*sm-sm'*a) + E*(a'+a);
```

This can be written down by inspection and will automatically generate the operator for H in the chosen representation. Notice that by defining the operators `a` and `sm` as above, we can generate the representation for operators such as $a^\dagger \sigma_-$ simply by writing `a'*sm`. This could alternatively have been formed using `tensor(destroy(N)',sigmam)` but the advantage of the former construction is its similarity to the analytic expression. Since the operators are stored as sparse matrices, the fact that `a` and `sm` have dimensions larger than `destroy(N)'` and `sigmam` is not an excessive overhead for the notational convenience.

4. Superoperators and density matrix equations

The generic form of a master equation is

$$\frac{d\rho}{dt} = \mathcal{L}\rho$$

where ρ is the density matrix and the Liouvillian $\mathcal{L}$ is a superoperator which may involve both premultiplication and postmultiplication by other operators. For example, a typical Liouvillian (for spontaneous emission from a two-level atom) is

$$\mathcal{L}\rho = \gamma\sigma_-\rho\sigma_+ - \frac{\gamma}{2}\sigma_+\sigma_-\rho - \frac{\gamma}{2}\rho\sigma_+\sigma_-$$

This is a linear, operator-valued transformation acting on ρ. When a super-operator such as $\mathcal{L}$ acts on a matrix such as ρ, this is actually done by regarding the elements of the matrix ρ as being strung out as a column vector. In Matlab, given a matrix such as `A=[1,2,3;4,5,6;7,8,9]`, we can construct the vector denoted by `A(:)` which simply consists of the elements of `A` written column-wise, so that in this example, `A(:)=[1;4;7;2;5;8;3;6;9]`. We shall follow the convention of column-wise ordering when converting from matrices to vectors and call this process “flattening” the matrix. Corresponding to a matrix ρ, we shall denote the flattened vector by $\widetilde{\rho}$.

Suppose that we have the operator a and the density operator ρ with 2×2 matrix representations

$$a = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \text{ and } \rho = \begin{pmatrix} \rho_{11} & \rho_{12} \\ \rho_{21} & \rho_{22} \end{pmatrix}.$$

If we premultiply the matrix ρ by the matrix a, we obtain

$$a\rho = \begin{pmatrix} a_{11}\rho_{11} + a_{12}\rho_{21} & a_{11}\rho_{12} + a_{12}\rho_{22} \\ a_{21}\rho_{11} + a_{22}\rho_{21} & a_{21}\rho_{12} + a_{22}\rho_{22} \end{pmatrix}$$

The column vector associated with $a\rho$ is related to that associated with ρ via the following linear transformation

$$\widetilde{a\rho} = \begin{pmatrix} a_{11}\rho_{11} + a_{12}\rho_{21} \\ a_{21}\rho_{11} + a_{22}\rho_{21} \\ a_{11}\rho_{12} + a_{12}\rho_{22} \\ a_{21}\rho_{12} + a_{22}\rho_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{11} & a_{12} \\ 0 & 0 & a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} \rho_{11} \\ \rho_{21} \\ \rho_{12} \\ \rho_{22} \end{pmatrix} = \mathrm{spre}\,(a)\,\tilde{\rho}$$

The 4×4 matrix represents the superoperator associated with premultiplication by a. We see that it may be formed simply from the matrix a. This is true in general, and the function `spre` in the toolbox is provided in order to compute the superoperator associated with premultiplication by a matrix. Thus if `A` and `B` are square matrices of the same size, the flattened version of `A*B` is equal to `spre(A)*B(:)`.

In the same way, we may represent postmultiplication by a matrix by a superoperator acting on the vector representation. In the toolbox, the function `spost` performs this conversion so that the flattened version of `A*B` can alternatively be found as `spost(B)*A(:)`. We regard `spost(B)` as a superoperator acting on the flattened matrix `A(:)` which is a column vector. Notice that since flattening always produces a column vector, superoperators always act on the left of such vectors, i.e.,

$$\widetilde{ab} = \mathrm{spost}\,(b)\,\tilde{a}.$$

The advantage of considering superoperators rather than operators is that the actions of premultiplication and postmultiplication by operators are both converted into premultiplication by a superoperator. For example, the Liouvillian $\mathcal{L}$ given above

$$\mathcal{L}\rho = \gamma\sigma_-\rho\sigma_+ - \frac{\gamma}{2}\sigma_+\sigma_-\rho - \frac{\gamma}{2}\rho\sigma_+\sigma_-,$$

may be written as a single superoperator once we have the matrices `sm` representing σ_- and `sm'` representing σ_+. It is simply

```
gamma*(spre(sm)*spost(sm')-0.5*spre(sm'*sm)-0.5*spost(sm'*sm))
```

Note that the calculation of $\sigma_-\rho\sigma_+$ involves postmultiplication by σ_+ and premultiplication by σ_-. These operations correspond to the multiplication of the superoperators `spre(sm)*spost(sm')` which happens to be commutative in this case. In the other two terms, we compute the superoperators corresponding to premultiplication and postmultiplication by $\sigma_+\sigma_-$. It is also possible to consider a term such as $\sigma_+\sigma_-\rho$ as a premultiplication by σ_- followed by a premultiplication by σ_+ and so this could be expressed as `spre(sm')*spre(sm)*rho(:)`. It is easy to check however that `spre(sm'*sm) = spre(sm')*spre(sm)`, and that it is more efficient to compute this by multiplying the operators together first.

If there is a Hamiltonian component to the evolution as well, represented by the matrix `H`, we simply add in the commutator to the Liouvillian

$$\mathcal{L}_H\rho = -i\,[H,\rho] = -i\,(H\rho - \rho H)$$

This is written using the toolbox as

```
-i*(spre(H)-spost(H))
```

From these examples, it should be clear that obtaining the sparse matrix representation of the Liouvillian starting from the Hamiltonian and the collapse operators in the Linblad form of the master equation is largely automatic, with the bookkeeping done using the sparse matrix structures.

In the toolbox, an additional refinement has been included. When the functions `spre` and `spost` are applied to operators, they return structures which are more complex than just matrices and so it is possible to identify the resulting objects as superoperators. For example, if we enter

```
>> a = tensor(destroy(5),identity(2))
a = Quantum object
Hilbert space dimensions [ 5 2 ] by [ 5 2 ]
...
>> L = spre(a)
L = Quantum object
Hilbert space dimensions ([ 5 2 ] by [ 5 2 ]) by ([ 5 2 ] by [ 5 2 ])
...
```

we find here that `a` is an operator with Hilbert space dimensions `[5 2]` by `[5 2]` since it acts on ten-dimensional kets to produce ten-dimensional kets. Once we use the function `spre`, the dimensions become `([ 5 2 ] by [ 5 2 ]) by ([ 5 2 ] by [ 5 2 ])` which indicates that `L` is now a super-operator which acts on matrices of dimensions `[5 2]` by `[5 2]` to produce a matrix of the same dimensions. Thus we may write `L*rho` to compute the product of an super-operator and a density matrix (operator), returning a matrix (operator) result. Internally, the toolbox calculates the product by using

```
reshape(L*rho(:),N,N)
```

where the colon operator and the reshape command are used to flatten the density matrix and "unflatten" the resulting vector. It should be emphasized that one should **not** use the reshape command explicitly with toolbox objects, as this is done automatically. This is an example of operator overloading since the `*` operator carries out multiplication of super-operators and operators or of two operators depending on what makes sense.

5. Calculation of operator expectation values

Given the density matrix ρ,finding the expectation value of some operator a involves calculation of

$$\langle a\rangle = \mathrm{Tr}\,(a\rho)\,,$$

which is a linear functional acting on ρ to produce a number. Similarly, given a state ket $|\psi\rangle$, the expectation value of a is given by

$$\langle a\rangle = \mathrm{Tr}\,(a\,|\psi\rangle\,\langle\psi|) = \langle\psi|\,a\,|\psi\rangle$$

In the toolbox, the function `expect(op,state)` is used to compute the expectation value of an operator for the specified state. The state may be specified either as a density matrix or as a state vector. Note that the trace of ρ may be found by setting a to the identity.Steady state solution of a Master Equation

This is a simple, yet complete example of a problem which may be solved using the toolbox. We consider a cavity with resonant frequency ω_c and leakage rate κ containing a two-level atom with transition frequency ω_0, field coupling strength g and spontaneous emission rate γ. The cavity is driven by a coherent (classical) field $\mathcal{E}$ which is such that the maximum photon number in the cavity is small so that it is adequate to represent it by a truncated Fock state basis. The Hamiltonian of the system is as given above,

$$H = (\omega_0 - \omega_L)\,\sigma_+\sigma_- + (\omega_c - \omega_L)\,a^\dagger a + ig\left(a^\dagger\sigma_- - \sigma_+ a\right) + \mathcal{E}\left(a^\dagger + a\right)$$

and there are two collapse operators

$$C_1 = \sqrt{2\kappa}a$$
$$C_2 = \sqrt{\gamma}\sigma_-$$

corresponding to leakage from the cavity and spontaneous emission from the atom respectively. The Liouvillian has the standard Linblad form

$$\mathcal{L} = \frac{1}{i}\left(H\rho - \rho H\right) + \sum_{k=1}^{2} C_k \rho C_k^\dagger - \frac{1}{2}\left(C_k^\dagger C_k \rho + \rho C_k^\dagger C_k\right).$$

Having found the Liouvillian, we seek a steady-state solution for ρ, i.e., we wish to find ρ such that $\mathcal{L}\rho = 0$. The toolbox function `steady(L)` returns the density matrix ρ representing the steady-state density matrix for an arbitrary Liouvillian $\mathcal{L}$. This is done using the inverse power method for obtaining the eigenvector belonging to eigenvalue zero. The solution is normalized so that $\mathrm{Tr}\left(\rho\right) = 1$.

As an illustration, the function below returns the steady-state photocounting rates $\left\langle C_1^\dagger C_1 \right\rangle$ and $\left\langle C_2^\dagger C_2 \right\rangle$ for photodetectors monitoring the output field of the cavity and the spontaneous emission of the atom, as well as $\langle a \rangle$ which is proportional to the intracavity field. The intracavity photon number can be found from $\left\langle a^\dagger a \right\rangle = \left\langle C_1^\dagger C_1 \right\rangle / \left(2\kappa\right)$ and it is also easy to add to the programme to find expectation values of other quantities such as σ_z or higher moments moments of the intracavity field a.

```
function [count1, count2, infield] = probss(E,kappa,gamma,g,wc,w0,wl,N)
%
% [count1, count2, infield] = probss(E,kappa,gamma,g,wc,w0,wl)
%  solves the problem of a coherently driven cavity with a two-level atom
%
%  E = amplitude of driving field, kappa = mirror coupling,
%  gamma = spontaneous emission rate, g = atom-field coupling,
%  wc = cavity frequency, w0 = atomic frequency, wl = driving field frequency,
%  N = size of Hilbert space for intracavity field (zero to N-1 photons)
%
% count1 = photocount rate of light leaking out of cavity
% count2 = spontaneous emission rate
% infield  = intracavity field

ida = identity(N); idatom = identity(2);
% Define cavity field and atomic operators
a  = tensor(destroy(N),idatom);
sm = tensor(ida,sigmam);
% Hamiltonian
H = (w0-wl)*sm'*sm + (wc-wl)*a'*a + i*g*(a'*sm - sm'*a) + E*(a'+a);
% Collapse operators
C1    = sqrt(2*kappa)*a;
C2    = sqrt(gamma)*sm;
C1dC1 = C1'*C1;
C2dC2 = C2'*C2;
% Calculate the Liouvillian
LH = -i * (spre(H) - spost(H));
L1 = spre(C1)*spost(C1')-0.5*spre(C1dC1)-0.5*spost(C1dC1);
L2 = spre(C2)*spost(C2')-0.5*spre(C2dC2)-0.5*spost(C2dC2);
L  = LH+L1+L2;
% Find steady state
rhoss = steady(L);
% Calculate expectation values
count1  = expect(C1dC1,rhoss);
count2  = expect(C2dC2,rhoss);
```

```
infield = expect(a,rhoss);
```

A driver routine called `xprobss` demonstrates how we can obtain the system response as the frequency of the driving field is swept across the common resonant frequencies of the atom and cavity. Figures 1 and 2 show the results of this calculation. In Figure 2 it is evident that the effect of the atom on the cavity response is reduced for the stronger driving field.

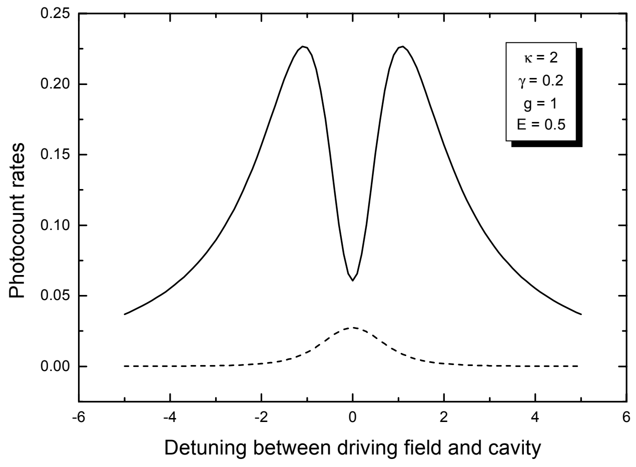

Figure 1: Photocounting rates for cavity output light (solid line) and for atomic spontaneous emission (dashed line)

Although the separation of the problem into a function which computes the response for a given set of parameters and a driver routine which loops over the parameter values is convenient, it leads to several redundant re-evaluations of quantities such as `L1` and `L2` which are not changed as the driving frequency is swept. The example file `xprobss2.m` illustrates a more efficient way of carrying out the above calculation.

Extracting the underlying matrix from a quantum object

Given a quantum object, the underlying matrix can be extracted by using subscript notation. For example,

```
>> A = qo([1,2,0;2,0,6;3,0,9])
A = Quantum object
Hilbert space dimensions [ 3 ] by [ 3 ]
     1     2     0
     2     0     6
     3     0     9
>> A(:,:)
ans =
   (1,1)        1
   (2,1)        2
   (3,1)        3
   (1,2)        2
   (2,3)        6
   (3,3)        9
```

Notice that only the non-zero elements of `A` are stored as a sparse matrix. This may be converted into a full matrix using the notation `full(A(:,:))`.

Instead of using the colon notation which extracts all the rows and/or columns of the matrix, one can provide an integer vector of indices using the standard Matlab rules. Thus for example, with the above

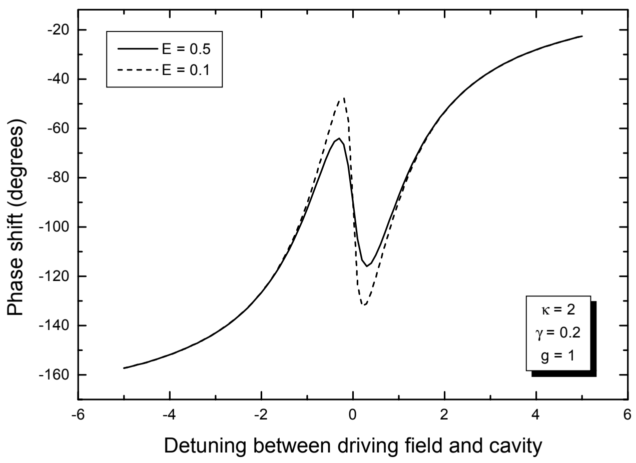

Figure 2: Phase of intracavity light field for two values of external driving field amplitude

definition of A,

```
>> full(A(2:3,[1,3]))
ans =
     2     6
     3     9
```

Note that the result of a subscripting operation applied to a quantum object is to produce an ordinary sparse matrix, *not* a quantum object.

Arrays of quantum objects

So far, we have created single quantum objects which may be thought of as quantum array objects with a single member. It is sometimes convenient to construct an array of several quantum objects, all with the same dimensions. For example, the state vector of an electron confined in one dimension is a spinor-valued function, so that at each point in space, there is a 2×1 ket. If space is discretized, we have a ket at each of a set of N points. This can be conveniently represented by a $N \times 1$ member array of quantum objects each of dimensions 2×1. As another example, we may wish to consider the three operators J_x, J_y and J_z together as a 3×1 array of operators which we may denote by $\mathbf{J}$.

Consider entering the array of angular momentum operators associated with $j = 2$. This may be done as follows:

```
>> J = [jmat(2,'x'); jmat(2,'y'); jmat(2,'z')]
J = 3 x 1 array of quantum objects
Hilbert space dimensions [ 5 ] by [ 5 ]
Member (1,1)
  ...
Member (2,1)
  ...
Member (3,1)
  ...
```

The size field of this object is [3,1] indicating the number of members in the array. It is possible to access the members by using subscipts within **braces**. Thus for example, J{2} or J{2,1} evaluates to the second member $J_y^{(2)}$. One can also assign to a member of a quantum object array by placing a subscripted expression on the left of the equals sign, so that we could alternatively have written

```
>> J = jmat(2,'x'); % J is a quantum array object with a single member
>> J{2} = jmat(2,'y'); % Define second member
>> J{3} = jmat(2,'z'); % Define third member
```

This makes `J` into a 1×3 quantum array object. Note that it is important that `J` is a quantum array object before additional members are assigned to it. If necessary, a null `qo` object may be generated first as illustrated below

```
>> J = qo; % Define a null object
>> J{1} = jmat(2,'x'); % Define first member
>> J{2} = jmat(2,'y'); % Define second member
>> J{3} = jmat(2,'z'); % Define third member
```

If the initial call to the constructor is omitted, `J` becomes a cell array of simple quantum objects, rather than a single quantum array object with three members.

The arrays of quantum objects can have as many dimensions as desired. For example, one can make an assignment to `J{2,2}`, which would automatically enlarge the array to the smallest size which includes all the assigned members, leaving the unassigned members equal to zero. Subscripting with index vectors and the colon operator are also supported. Note however that the special index `end` may not be used.

In the above, the construction `[A;B;C]` was used to form a column of objects. It is similarly possible to constuct a row of objects by using `[A,B,C]` or alternatively `[A B C]`. The rules for constructing arrays are identical to those for making Matlab matrices and so arrays may also be concatenated horizontally or vertically in the usual way. Arrays with more than two dimensions can also be constructed by assigning to an element with more than two subscripts, but support for such constructs is still rudimentary. In particular, the `cat` function has not been implemented.

The function `jmat` may also be called with only one argument `j`, in which case it returns the 3×1 array, formed manually above. Similarly, `basis` described previously may also be called with a single argument, e.g., `basis(N)` in which case it returns an $1 \times N$ array of quantum objects, the k'th member being a unit ket with a one in the k'th element of the matrix.

6. Operations involving arrays of objects

The basic operations which are defined on individual quantum objects may also be applied to arrays of objects. Unary operations, such as the transpose (`.'`) and conjugate transpose (`'`) operations when applied to an array of quantum objects cause the operation to be applied to each member of the array in turn, **without** changing the shape of the array. Thus for example, if `psi` is a 3×2 array of kets with Hilbert space dimensions [5] by [1], `psi'` is a 3×2 array of bras with Hilbert space dimensions [1] by [5].

Binary operations can be performed on compatible arrays. Two arrays of quantum objects are compatible if they are of the same size, or if one of the arrays has only one member. When the arrays are of the same size, the binary operations are applied to the corresponding members. For example if we have a 2×2 array of operators `[A B;C D]` and a 2×2 array of kets `[va vb;vc vd]` which have the same Hilbert space dimensions as the operators, the result of multiplying these two arrays is to form the array `[A*va B*vb;C*vc D*vd]`. If one of the arrays has only one member, that member is applied (using the binary operation) to each of the elements of the other array and the resulting array has the size of the bigger array. Thus for example, `v + [u1;u2;u3]` is equal to `[v+u1;v+u2;v+u3]`.

Binary operations can often also be performed between quantum object arrays and ordinary double (possibly complex-valued) arrays. If the arrays are of the same size, operations take place between corresponding members. For example if `[A B;C D]` is an array of quantum objects, we can compute `[A B;C D]^[3 4;2 5]` which evaluates to `[A^3 B^4;C^2 D^5]`. If one of the arrays has only one member, that member is applied to each of the elements of the other array, and the resulting array has the size of the bigger array. Thus for example, `2*[A B;C D]=[2*A 2*B;2*C 2*D]`.

Quantum array objects whose elements are from a one dimensional Hilbert space are regarded as being equivalent to ordinary double arrays for the purpose of the above compatibility rules.

Table 1 is a list of the operations available for quantum object arrays. Capital letters denote quantum object (arrays) and lower case letters indicate ordinary double matrices.

For all of the above, the array structure of the quantum object is largely ignored as operations take place between corresponding members or between a single object and each of an array of objects.

The two functions for which the array structure of the objects is important are `combine` and `sum`:

`A+B`	Addition
`A-B`	Subtraction
`A*B`	Multiplication
`A.*B`	Elementwise multiplication
`A^d`	Matrix power
`d^A`	Matrix exponentiation
`A.^B`	Elementwise power
`A'`	Conjugate transpose
`A.'`	Transpose
`A.\B`	Elementwise left division
`A/B`	Matrix right division
`A./B`	Elementwise right division
`+A`	Unary plus
`-A`	Unary minus
`diag(Op)`	Diagonal, returned as a double array
`expect(Op,State)`	Expectation value
`trace(Op)`	Trace of operator
`spre(Op)`, `spost(Op)`	Convert operator to superoperator
`tensor(Op1,Op2,...)`	Tensor product of states or operators

Table 1: Operations on Arrays of Quantum Objects

- The function `combine(A,B)` is used when an array multiplication is required. It is only valid when `A` and `B` are arrays with two or fewer dimensions, and the number of rows of `A` is equal to the number of columns of `B`. If `C` = `combine(A,B)`, $C_{ik} = \sum_j A_{ij} B_{jk}$ where the multiplication between A_{ij} and B_{jk} is of the appropriate type for the objects A_{ij} and B_{jk}. In particular, it is valid for A or B to be a matrix of doubles, so that we can write, for instance, `combine([A,B,C],[1;2;3])` to calculate `A+2*B+3*C`.
- The function `sum(A)`computes the sum of the members of the array `A` along the first non-singleton dimension. Thus if `A` is a row or a column array, `sum(A)` adds up all the members of `A`. On the other hand, if `A` is a rectangular array, `sum(A)` adds up the columns of `A`. More generally, one can use `sum(A,nd)` to add up along the `nd`'th dimension of the array. Thus if `C=[C1,C2,C3]`, we can use `sum(C'*C)` to calculate `C1'*C1 + C2'*C2 + C3'*C3`.

7. Orbital Angular Momentum States

The ket space for a system with angular momentum quantum number j is $2j+1$ dimensional, where j can be an integer or half-integer. For systems with a central potential, the angular part of the wave function may be decomposed into a sum over spherical harmonics $\sum_{l,m} c_{lm} Y_l^m(\theta,\phi)$, where each spherical harmonic $Y_l^m(\theta,\phi)$ is an eigenfunction of the orbital angular momentum operators L^2 and L_z with eigenvalues $l(l+1)^2$ and m respectively. The spherical wave function $\psi(\theta,\phi) = \sum_{l,m} c_{lm} Y_l^m(\theta,\phi)$ provides a convenient way of visualizing a ket when the angular momentum quantum number is an integer. The space in which the ket resides is the direct sum of the spaces $\mathcal{S}^{(0)} \oplus \mathcal{S}^{(1)} \oplus \mathcal{S}^{(2)} \oplus ...$ where $\mathcal{S}^{(l)}$ denotes the $2l+1$ dimensional space associated with quantum number l. The coefficients c_{lm} may be divided into $[c_{0,0}]$, $[c_{1,1}; c_{1,0}; c_{1,-1}]$, $[c_{2,2}; c_{2,1}; c_{2,0}; c_{2,-1}; c_{2,-2}]$, ... each of which specify a vector in the component spaces. The toolbox function `orbital` is used to help us visualize a ket in this space by allowing us to evaluate $\psi(\theta,\phi)$ on a grid of θ and ϕ and for any set of coefficients.

First let us consider the situation of a ket in a space of fixed l, so that $\psi(\theta,\phi) = \sum_m c_m Y_l^m(\theta,\phi)$ is a sum over m alone. The coefficients c_m form a column vector of $2l+1$ numbers. For example if we wish to plot a representation of $|l=2, m=1\rangle$, the following may be used:

```
>> theta = linspace(0,pi,45); phi = linspace(0,2*pi,90);
>> c2 = qo([0;1;0;0;0]); % Components are for m = 2,1,0,-1 and -2 respectively
>> psi = orbital(theta,phi,c2);
>> sphereplot(theta,phi,psi);
```

Enter the above code, or use the file `xorbital` to observe the result. Note that the resulting plot shows both $|\psi|$ as the distance of the surface from the origin and $\arg(\psi)$ as the colour of the surface. The file `xorbital` also shows how a rotation of the spatial axes may be applied to the ket `c2` to give the state in a different orientation.

If we have a ket with components c_{lm} for several different values of l, the function `orbital` may be called with additional arguments. For example, it is known that the direction eignket $|\hat{\mathbf{z}}\rangle$ may be written as

$$|\hat{\mathbf{z}}\rangle = \sum_{l=0}^{\infty} \sqrt{\frac{2l+1}{4\pi}}\, |l, m=0\rangle$$

If we wish to compute this angular wave function for $l = 0$ to $l = 4$, we can write

```
>> c0 = qo([1]); c1 = sqrt(3)*qo([0;1;0]); c2 = sqrt(5)*qo([0;0;1;0;0]);
>> c3 = sqrt(7)*qo([0;0;0;1;0;0;0]); c4 = sqrt(9)*qo([0;0;0;0;1;0;0;0;0]);
>> psi = 1/sqrt(4*pi)*orbital(theta,phi,c0,c1,c2,c3,c4);
```

and use `sphereplot` again to display the result. The file `xdirection` illustrates a way of calculating this for an arbitrary maximum l value by using cell arrays of quantum objects.

8. Simultaneous Diagonalization of Operators

In this section, we introduce the function `simdiag` which is used to find the eigenkets of an operator or the common eigenkets of a set of mutually commuting Hermitian operators. We can illustrate the calculation of Clebsch-Gordon coefficients by explicit diagonalization of the angular momentum matrices. This will provide an example of using arrays of quantum objects and performing operations on such arrays.

Refer to the following listing and consider two objects with angular momentum quantum numbers j_1 and j_2 respectively. The Hilbert space dimensions for the two objects are then $2j_1+1$ and $2j_2+1$ so `id1` and `id2` represent identity operators for these spaces. Next we form an arrays for the angular momentum operators: `J1` is the array $[J_x \otimes \mathbf{1};\ J_y \otimes \mathbf{1};\ J_z \otimes \mathbf{1}]$ for the first object in the two-particle space, and `J2` is the corresponding array $[\mathbf{1} \otimes J_x;\ \mathbf{1} \otimes J_y;\ \mathbf{1} \otimes J_z]$ for the second object. The quatity `Jtot` is then the 3×1 array of operators for the total angular momenta J_x, J_y and J_z.

Recalling that operators act on arrays component-wise, we see that `J1^2` is the array $[(J_{1x})^2;\ (J_{1y})^2;\ (J_{1z})^2]$. The `sum` function finds the sum of the members of a quantum array object, so `J1sq=sum(J1^2)` is the operator for $J_1^2 = J_{1x}^2 + J_{1y}^2 + J_{1z}^2$. Alternatively, the `combine` function may be used to take a matrix product of `[1,1,1]` and the array `J1^2,` also giving the operator `J1sq`. Similarly we form the operators J_2^2 and $(J_1+J_2)^2$. In order to find the Clebsch-Gordon coefficients, we need to find the simultaneous eigenstates of $(J_1+J_2)^2$ and $(J_1+J_2)_z$. This is done using `simdiag` which gives the state vectors in the original basis (i.e., in terms of eigenstates of J_{1z} and J_{2z}.)

Enter the commands shown or use the file `xclebsch`. The array `evalues` is of size 4×6 which has a column for each of the eigenvectors. The first row gives the eigenvalues of `Jsqtot` which we see are $\left(\frac{3}{2}\right)\left(\frac{3}{2}+1\right)$ and $\left(\frac{1}{2}\right)\left(\frac{1}{2}+1\right)$, while the second row gives the eigenvalues of `Jztot` which are $\pm\frac{3}{2}, \pm\frac{1}{2}$ for $j=\frac{3}{2}$ and $\pm\frac{1}{2}$ for $j=\frac{1}{2}$. The third and fourth rows give the eigenvalues of J_1^2 and J_2^2 which are obviously $\left(\frac{1}{2}\right)\left(\frac{1}{2}+1\right)$ and $(1)(1+1)$. The output variable `states` is an array of the six eigenvectors arranged as a quantum object. The basis kets in the original $|m_1, m_2\rangle$ basis are $\left|\frac{1}{2}, 1\right\rangle$, $\left|\frac{1}{2}, 0\right\rangle$, $\left|\frac{1}{2}, -1\right\rangle$, $\left|-\frac{1}{2}, 1\right\rangle$, $\left|-\frac{1}{2}, 0\right\rangle$, $\left|-\frac{1}{2}, -1\right\rangle$ while those in the new $|j, m\rangle$ basis are $\left|\frac{3}{2}, \frac{3}{2}\right\rangle$, $\left|\frac{3}{2}, \frac{1}{2}\right\rangle$, $\left|\frac{3}{2}, -\frac{1}{2}\right\rangle$, $\left|\frac{3}{2}, -\frac{3}{2}\right\rangle$, $\left|\frac{1}{2}, \frac{1}{2}\right\rangle$, $\left|\frac{1}{2}, -\frac{1}{2}\right\rangle$. The function `ket2xfm` expresses the states as the transformation matrix $\langle j, m | m_1, m_2\rangle$ or as the matrix $\langle m_1, m_2 | j, m\rangle$ depending on whether its second argument is `'fwd'` or `'inv'`.

```
j1 = 1/2;
j2 = 1;
id1 = identity(2*j1+1);
id2 = identity(2*j2+1);

J1 = tensor(jmat(j1),id2);
J2 = tensor(id1,jmat(j2));
Jtot = J1 + J2;
```

```
J1sq = sum(J1^2); % or use combine([1,1,1],J1^2);
J2sq = sum(J2^2);
Jsqtot = sum(Jtot^2);
Jztot = Jtot{3};

[states,evalues] = simdiag([Jsqtot,Jztot,J1sq,J2sq]);
xform = ket2xfm(states,'inv')
```

For example, we find $\left|j=\frac{3}{2},m=-\frac{1}{2}\right\rangle = 0.5774\left|m_1=\frac{1}{2},m_2=-1\right\rangle + 0.8165\left|m_1=-\frac{1}{2},m_2=0\right\rangle.$

As a second example of the use of `simdiag`, we demonstrate the GHZ paradox which strikingly illustrates the differences in the predictions of quantum mechanics and those of locally realistic theories. Consider three spin $\frac{1}{2}$ particles, on each of which we may choose to measure either the spin in the x direction or in the y direction. Associated with the operator $\sigma_x^1\sigma_y^2\sigma_y^3$, for example, is a measurement of the x component of the spin of the first particle and the y component of the spin of the second and third particle. Working in units of $\hbar/2$, each measurement yields either 1 or -1, and so the product of the three is also either 1 or -1.

The GHZ argument considers the following four operators which all commute with each other: $A_1 = \sigma_x^1\sigma_y^2\sigma_y^3$, $A_2 = \sigma_y^1\sigma_x^2\sigma_y^3$, $A_3 = \sigma_y^1\sigma_y^2\sigma_x^3$ and $A_4 = \sigma_x^1\sigma_x^2\sigma_x^3$. Under the assumption of local reality, each of the spin operators $\sigma_{x,y}^{1,2,3}$ will in principle have well-defined values, although they may not be measured simultaneously. Classically, the product $A_1A_2A_3$ is equal to A_4 since each $\sigma_y^{1,2,3}$ appears twice in the product and $(\pm1)^2 = 1$. On the other hand, the code in the listing shown below (which is in the file `xghz.m`) demonstrates that when $A_1, ..., A_4$ are diagonalized simultaneously, there is an eigenvector for which the eigenvalues for $A_1, ..., A_4$ are $1, 1, 1, -1$. This shows that there is a state for which measurements of A_1, A_2 and A_3 are certain to yield 1 but for which a measurement of A_4 would yield -1, which is opposite in sign to the product of the results for the first three observables. This program is contained in the file `xghz`.

```
sx1 = tensor(sigmax,identity(2),identity(2));
sy1 = tensor(sigmay,identity(2),identity(2));

sx2 = tensor(identity(2),sigmax,identity(2));
sy2 = tensor(identity(2),sigmay,identity(2));

sx3 = tensor(identity(2),identity(2),sigmax);
sy3 = tensor(identity(2),identity(2),sigmay);

op1 = sx1*sy2*sy3;
op2 = sy1*sx2*sy3;
op3 = sy1*sy2*sx3;
op4 = sx1*sx2*sx3;

[states,evalues] = simdiag([op1 op2 op3 op4]);
evalues(:,1)
```

9. Operator Exponentiation

The toolbox overloads the function `expm` to allow the exponentiation of operators and superoperators. The following examples demonstrate the use of this function together with several of the functions available for visualizing states.

Consider the displacement and squeezing operators for a single mode of the electromagnetic field defined by

$$D(\alpha) = \exp\left(\alpha a^\dagger - \alpha^* a\right)$$
$$S(\varepsilon) = \exp\left(\frac{1}{2}\varepsilon^* a^2 - \frac{1}{2}\varepsilon a^{\dagger 2}\right)$$

The following program (contained in `xsqueeze.m`) draws the Wigner and Q functions of a squeezed state. The values of α and ε should be chosen to be small enough that the state can be represented adequately in a 20 dimensional Fock space. The Wigner and Q functions are computed using the functions `wfunc` and `qfunc` at a set of points defined by the vectors `xvec` and `yvec` which may be chosen arbitrarily.

Note that the relationships between x, y and a, $a^\dagger$ are assumed to be given by

$$a = \frac{g}{2}(x + iy), \; a^\dagger = \frac{g}{2}(x - iy)$$

where g is a constant which may be used to account for various definitions in use. If the parameter `g` is omitted in the call to `wfunc` or to `qfunc`, the value $g = \sqrt{2}$ is assumed.

```
N = 20;
alpha = input('alpha = ');
epsilon = input('epsilon = ');
a = destroy(N);
D = expm(alpha*a'-alpha'*a);
S = expm(0.5*epsilon'*a^2-0.5*epsilon*(a')^2);
psi = D*S*basis(N,1);
g = 2;
xvec = [-40:40]*5/40; yvec = xvec;
W = wfunc(psi,xvec,yvec,g);
figure(1); pcolor(xvec,yvec,real(W));
shading interp; title('Wigner function of squeezed state');
Q = qfunc(psi,xvec,yvec,g);
figure(2); pcolor(xvec,yvec,real(Q));
shading interp; title('Q function of squeezed state');
```

In the above, the state for `wfunc` and for `qfunc` is specified as a ket `psi`. It is also valid to use a density matrix to specify the state. The file `xschcat` similarly illustrates the calculation of the Wigner and Q functions of a Schrödinger cat state.

The operator exponential is also useful for generating matrices for rotations through finite angles since a rotation through ϕ about the $\mathbf{n}$ axis may be written as $\exp(-i\mathbf{n}\cdot\mathbf{J}\phi)$. This is illustrated in the file `xorbital`. The file `rotation` is also available for converting between descriptions of a rotation specified in various formats (as Euler angles, as an axis and an angle, as a 3×3 orthogonal matrix in $SO(3)$ or as a unitary matrix in $SO(2)$). As an example, try entering `rotateworld([pi/3,pi/4,pi/2],'euler')`. The file `xrotateworld` shows a spinning globe by making successive calls to `rotateworld`.

10. Superoperator Adjoints

Given a superoperator $\mathcal{L}$ which acts on a density matrix $\boldsymbol{\rho}$ via the operation

$$\mathcal{L}\boldsymbol{\rho} = \mathbf{A}\boldsymbol{\rho}\mathbf{B}$$

where $\mathbf{A}$ and $\mathbf{B}$ are operators, we often wish to compute the "superoperator adjoint" $\widetilde{\mathcal{L}}$ which is defined so that its action on $\boldsymbol{\rho}$ is

$$\widetilde{\mathcal{L}}\boldsymbol{\rho} = \mathbf{B}^\dagger\boldsymbol{\rho}\mathbf{A}^\dagger$$

where the dagger denotes the adjoints of the operators. In the toolbox, the function `sadjoint(L)` may be used to find the superoperator adjoint of a quantum object.

Hilbert Space Permutation and Calculation of Partial Traces

It is sometimes desirable to re-order the spaces which are multiplied together in a tensor product. For example, if we take the product of three operators such as

```
>> A = tensor(destroy(5),jmat(1/2,'x'),destroy(4));
```

the result of this is to produce a matrix with Hilbert space dimensions [5 2 4] by [5 2 4]. The toolbox function `permute` is provided to change the order of the spaces. For example if we enter

```
>> B = permute(A,[3,1,2])
```

the result is to re-order the spaces in `A` so that `B = tensor(destroy(4),destroy(5),jmat(1/2,'x'))`. The function `permute` may be applied to vectors, operators and super-operators.

Another operation which is often useful is the calculation of partial traces, usually of density matrices or more generally of any operator. Given a density matrix `rho` with Hilbert space dimensions $[d_1\ d_2\ ...\ d_m]$ by $[d_1\ d_2\ ...\ d_m]$, it is possible to trace over any collection of indicies by specifying which indicies are to remain after the trace is computed. For example, if we want to trace over spaces 2, 3, ..., $m-1$ so that only spaces 1 and m remain, the required command is

```
>> ptrace(rho,[1,m]);
```

As an example, consider the density matrix of one of the particles of a Bell pair which may be found by tracing over the second particle:

```
>> up = basis(2,1); dn = basis(2,2);
>> bell = (tensor(up,up) + tensor(dn,dn))/sqrt(2);
>> ptrace(bell*bell',1)
ans = Quantum object
Hilbert space dimensions [ 2 ] by [ 2 ]
   (1,1)       0.5000
   (2,2)       0.5000
```

The result is indistinguishable from a statistical mixture of up and down with equal probability. Note that the first argument of `ptrace` which specifies the state can be either a density matrix or a ket, so that it would also be valid to write `ptrace(bell,1)`. In either case, the result of `ptrace` is to produce a (reduced) density matrix.

On the other hand, suppose that we measure the second particle and find it to be in a state which is $|\psi_r\rangle = (|\mathrm{up}\rangle + |\mathrm{dn}\rangle)/\sqrt{2}$, we can calculate the probability that this occurs and the conditional state of the first particle given this measurement result.

```
>> left = (up + dn)/sqrt(2);
>> Omega_left = tensor(identity(2),left*left');
>> rho_new = ptrace(Omega_left*bell,1);
>> Prob = trace(rho_new);
>> rho = rho_new / Prob
rho = Quantum object
Hilbert space dimensions [ 2 ] by [ 2 ]
   (1,1)       0.5000
   (2,1)       0.5000
   (1,2)       0.5000
   (2,2)       0.5000
```

We find the value of `Prob` is $\frac{1}{2}$ and that the conditional density matrix is just `left*left'`, indicating that the first particle is projected into the `left` state. Computation of `trace(rho^2)` confirms that the system is left in a pure state. The script `xptrace` carries out the calculations described above.

Truncation of Hilbert space

Given an n dimensional Hilbert space, it is sometimes necessary to extract the components of a quantum object which lie in some m dimensional subspace of the full space. The toolbox function `truncate` is used to achieve this. First consider an example in which the Hilbert space which is not the tensor product of other spaces:

```
>> A = qo([ 1, 2, 3, 4;
            5, 6, 7, 8;
            9,10,11,12;
           13,14,15,16]);
>> B = truncate(A,{[1,2,4]})
ans = Quantum object
Hilbert space dimensions [ 3 ] by [ 3 ]
     1     2     4
     5     6     8
    13    14    16
```

The second argument to `truncate` is a Matlab cell array object (enclosed in braces). This cell array has only one element, the selection vector `[1,2,4]` which specifies the dimensions along which the truncated object is to be computed. The result of the function is another quantum object which is defined over a three dimensional Hilbert space. The order of the elements in the selection vector need not be increasing. This allows one to permute the order of the basis elements in the Hilbert space.

Next consider a Hilbert space which is the tensor product of several spaces. In this case, the second argument to `truncate` is a cell array with as many selection vectors as there are subspaces in the tensor product. For example,

```
>> A = tensor(qo(randn(4,4)),qo(randn(3,3)),qo(randn(4,4)))
A = Quantum object
Hilbert space dimensions [ 4 3 4 ] by [ 4 3 4 ]
...
>> B = truncate(A,{1:3,[1,3],3:4})
B = Quantum object
Hilbert space dimensions [ 3 2 2 ] by [ 3 2 2 ]
...
```

The truncated operator is defined in the space formed by taking the first three dimensions of the first four-dimensional subspace, the first and third dimensions of the second three-dimensional subspace, and the last two dimensions of the third four-dimensional subspace. The resulting Hilbert space is twelve dimensional and is the product of subspaces of dimensions 3, 2 and 2 as indicated.

If the truncated Hilbert space cannot be expressed as a tensor product of truncated subspaces, it is possible to specify an arbitrary collection of basis vectors along which the result is to be computed. For example, suppose that out of the 48 dimensional Hilbert space over which `A` operates, we wish to select a three-dimensional subspace. The basis vectors of the full Hilbert space are outer products of three basis vectors, one in the first four-dimensional subspace spanned by $\{\mathbf{u}_i\}_{i=1}^4$, say; one in the second three-dimensional subspace spanned by $\{\mathbf{v}_j\}_{j=1}^3$, say; and one in the third four-dimensional subspace spanned by $\{\mathbf{w}_k\}_{k=1}^4$. Thus for example, we may specify a basis vector of the Hilbert space by the index vector `[1,3,2]` which represents $\mathbf{u}_1 \otimes \mathbf{v}_3 \otimes \mathbf{w}_2$. Similarly, the index vectors `[1,1,2]` represents $\mathbf{u}_1 \otimes \mathbf{v}_1 \otimes \mathbf{w}_2$ and `[1,2,4]` represents $\mathbf{u}_1 \otimes \mathbf{v}_2 \otimes \mathbf{w}_4$ If we wish to truncate `A` to the space spanned by these three vectors, we use

```
>> B = truncate(A,[1,3,2; 1,1,2; 1,2,4])
B = Quantum object
Hilbert space dimensions [ 3 ] by [ 3 ]
...
```

The second argument to `truncate` in this case is an ordinary matrix, and *not* a cell array. The rows of the matrix specify the indices of the basis vectors in each of the subspaces. The number of rows in the matrix gives the dimensionality m of the truncated space.

Note: The truncate function may be applied to bras, kets, operators or superoperators. It also operates element by element on arrays of quantum objects.

Exponential Series

In this section we consider obtaining the time-dependent solution of the problem involving an atom in an optical cavity. In this case, the Liouvillian is *time-independent* and is small enough to be diagonalized numerically. Under such circumstances, provided that the eigenvalues of $\mathcal{L}$ are not degenerate, it is possible to write down the time-dependent solution $\rho(t)$ of the master equation as a sum of complex exponentials $\exp(s_j t)$ where each s_j is an eigenvalue of $\mathcal{L}$. Obtaining the solution in this form allows it to be evaluated readily at any time in the future. By contrast, carrying out a numerical integration of the master equation can be computationally intensive for large times. In terms of the framework used in the toolbox, we wish to express the operator $\rho(t)$ as a linear combination of the $\exp(s_j t)$, where the amplitudes may be quantum objects. The general form of this expansion is

$$\tilde{\rho}_i(t) = \sum_j a_{ij} \exp s_j t$$

where $\tilde{\rho}$ represents the density operator "flattened" into a vector. In order to find the coefficient matrix

a_{ij}, we diagonalize $\mathcal{L}$ writing it as

$$\mathcal{L}_{ik} = \sum_j \mathbf{V}_{ij} s_j \left(\mathbf{V}^{-1}\right)_{jk}$$

Thus

$$\left(e^{\mathcal{L}t}\right)_{ik} = \sum_j \mathbf{V}_{ij} \exp\left(s_j t\right) \left(\mathbf{V}^{-1}\right)_{jk}$$

and so

$$\begin{aligned}\tilde{\rho}_i(t) &= \sum_k \left(e^{\mathcal{L}t}\right)_{ik} \tilde{\rho}_k(0) = \sum_j \left(\mathbf{V}_{ij} \sum_k \left(\mathbf{V}^{-1}\right)_{jk} \tilde{\rho}_k(0)\right) \exp\left(s_j t\right) \\ &= \sum_j \mathbf{V}_{ij} r_j \exp\left(s_j t\right)\end{aligned}$$

where r_j are the components of the vector $\mathbf{r} = \mathbf{V}^{-1}\tilde{\rho}(0)$. This shows that the matrix $\mathbf{A} = (a_{ij})$can be computed by multiplying $\mathbf{V}$ by the diagonal matrix with r_j on the diagonal. The conversion from $\mathcal{L}$ and $\rho(0)$ to $\mathbf{A}$ and $\mathbf{s}$ (the vector of eigenvalues s_j of $\mathcal{L}$) is carried out by the toolbox function

```
ES = ode2es(L,r0)
```

which is so-named as it solves a system of ordinary differential equations with constant coefficients as an *exponential series.* In this context, an exponential series is a sum of the form

$$\mathbf{a}(t) = \sum_j \mathbf{a}_j \exp\left(s_j t\right)$$

where each $\mathbf{a}_j$ is a simple quantum object (i.e., a scalar, vector, operator or superoperator). Once the solution in the form of an exponential series, the toolbox may be used to evaluate and manipulate such exponential series, as will be described in more detail in the next section.

11. Manipulating Exponential Series

In the toolbox, an exponential series is an instance of the class `eseries` which internally consists of a one-dimensional quantum array object containing the amplitudes and a vector of rates associated with the series. If we wish to define the scalar exponential series

$$f(t) = 1 + 4\cos t - 6\sin 2t = 1 + 2\exp(it) + 2\exp(-it) + 3i\exp(2it) - 3i\exp(-2it),$$

which consists of four **amplitudes** (1, 2, 2, $3i$ and $-3i$) and four **rates (** 0, i, $-i$, $2i$ and $-2i$**)**, the constructor function for the `eseries` class may be used as

```
>> f = eseries(1)+eseries(2,i)+eseries(2,-i)+eseries(3i,2i)+eseries(-3i,-2i);
```

Note that the first argument to `eseries` is converted to a quantum object and specifies an amplitude, while the second is a rate. If the rate is omitted, a rate of zero is assumed. If we wanted to specify an operator-valued exponential series such as

$$b(t) = \sigma_x \exp(-2t) + \sigma_y \exp(-3t)$$

we would have used

```
>> b = eseries(sigmax,-2)+eseries(sigmay,-3);
```

An alternative way of specifying the series is give all the amplitudes as a quantum array object followed by the rates, i.e.,

```
>> b = eseries([sigmax,sigmay],[-2,-3]);
```

but of course, with this method it is not possible to replace the quantum array object by a double array. When the exponential series is displayed, the result is

```
b =
Exponential series: 2 terms.
Hilbert space dimensions [ 2 ] by [ 2 ].
Exponent{1} = (-2)
```

```
     0     1
     1     0

Exponent{2} = (-3)
        0                  0 - 1.0000i
        0 + 1.0000i        0
```

Once an exponential series has been formed, the field name `ampl` may be used to extract the quantum array object containing the amplitudes and `rates` may be used to extract the array of rates. Thus,

```
>> b.ampl
ans = 2 x 1 array of quantum objects
Hilbert space dimensions [ 2 ] by [ 2 ]
Member (1,1)
     0     1
     1     0

Member (2,1)
        0                  0 - 1.0000i
        0 + 1.0000i        0
>> b.rates
ans=
        -2
        -3
```

In the exponential series for $b(t)$, the first term is $\sigma_x \exp(-2t)$ while the second is $\sigma_y \exp(-3t)$. One can place an index in braces to select out some of the terms of the series. Thus we find

```
>> b{2}
b{2}
ans =
Exponential series: 1 terms.
Hilbert space dimensions [ 2 ] by [ 2 ].
Exponent{1} = (-3)
        0                  0 - 1.0000i
        0 + 1.0000i        0
```

Similarly, `b{[2,1]}` is the original series with the terms reversed. It is also possible to extract out terms from an exponential series on the basis of the rates. In this case, parentheses are used to carry out the indexing. Since the rate associated with $\sigma_x \exp(-2t)$ in the exponential series $b(t)$ is -2, we find that

```
>> b(-2)
ans =
Exponential series: 1 terms.
Hilbert space dimensions [ 2 ] by [ 2 ].
Exponent{1} = (-2)
     0     1
     1     0
```

If the specified rate(s) do not appear within the series, the result is zero. The above constructs may be combined to extract the amplitude(s) or rate(s) associated with term(s) in the exponential series. For example, in order to get the amplitude of the term with rate -3, we write

```
>> b(-3).ampl
ans = Quantum object
Hilbert space dimensions [ 2 ] by [ 2 ]
        0                  0 - 1.0000i
        0 + 1.0000i        0
```

The operators listed in Table 2 have been overloaded for exponential series:

When using `expect` for exponential series, either the operator or the state (or both) can be exponential series. An operand which is not an exponential series is first converted into one (with rate zero) before

`+`	Addition, unary plus
`-`	Subtraction, unary minus
`*`	Multiplication
`'`	Conjugate transpose
`.'`	Transpose
`spre(Op)`	Convert operator to superoperator for premultiplication
`spost(Op)`	Convert operator to superoperator for postmultiplication
`expect(Op,State)`	Calculate expectation value of an operator for a given state

Table 2: Operations on exponential series

it is used. It is thus invalid to pass a quantum array object with more than one member as a parameter to `expect` when the other parameter is an exponential series.

Two special functions associated with exponential series are

- `estidy(ES)` which "tidies up" an exponential series by removing terms with very small amplitudes and merging terms with rates which are very close to each other. Control over the tolerances used to discard and merge terms is available by using additional parameters to `estidy`. Many of the functions which manipulate exponential series automatically call `estidy(ES)` during their operation, but the function is also available to the user.
- `esval(ES,tarray)`which evealuates the exponential series `ES` at the times in `tarray`. The function returns an array of quantum objects of the same size as `tarray.` In the special case of a scalar exponential series, the result of `esval` is an array of doubles, rather than an array of quantum objects. For example, for the exponential series `f` defined above, we may plot `f` by using

```
>> t = linspace(0,10,101);
>> y = esval(f,t);
>> plot(t,y);
```

12. Time Evolution of a Density Matrix

Using these techniques, it easy to alter the code for the problem of an atom in a cavity to give the time-varying expectation values $\left\langle C_1^\dagger(t)\, C_1(t)\right\rangle$, $\left\langle C_2^\dagger(t)\, C_2(t)\right\rangle$ and $\langle a(t)\rangle$ for a given initial state. If for example the atom is initially in the ground state and the intracavity light field is in the vacuum state, the initial state vector `psi0` is the tensor product of `basis(N,1)` for the light and `basis(2,2)` for the atom. The initial density matrix is the outer product of the state vector and its conjugate. This file and its driver are called `probevolve` and `xprobevolve`.

```
function [count1, count2, infield] = probevolve(E,kappa,gamma,g,wc,w0,wl,N,tlist)
%
% [count1, count2, infield] = probevolve(E,kappa,gamma,g,wc,w0,wl,tlist)
%  solves the problem of a coherently driven cavity with a two-level atom
%  with the atom initially in the ground state and no photons in the cavity
%
%  E = amplitude of driving field, kappa = mirror coupling,
%  gamma = spontaneous emission rate, g = atom-field coupling,
%  wc = cavity frequency, w0 = atomic frequency, wl = driving field frequency,
%  N = size of Hilbert space for intracavity field (zero to N-1 photons)
%  tlist = times at which solution is required
%
% count1 = photocount rate of light leaking out of cavity
% count2 = spontaneous emission rate
% infield  = intracavity field

%%% Same code as in probss.m up to line:
```

```
L = LH+L1+L2;

% Initial state
psi0 = tensor(basis(N,1),basis(2,2));
rho0 = psi0 * psi0';
% Calculate solution as an exponential series
rhoES = ode2es(L,rho0);
% Calculate expectation values
count1 = esval(expect(C1dC1,rhoES),tlist);
count2 = esval(expect(C2dC2,rhoES),tlist);
infield = esval(expect(a,rhoES),tlist);
```

Figure 3 shows the photocount outputs from the cavity and the spontaneous emission as functions of time when the cavity is driven on-resonance. We see that at early times before the atom has had time to respond to the input field, the intracavity field is high and the spontaneous emission rate is low. As the atom comes to steady-state however, the intracavity field strength falls.

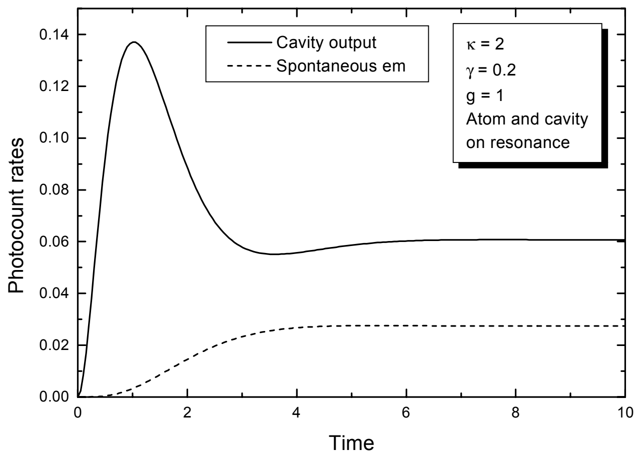

Figure 3: Time-dependent photocount rates for driven cavity with atom.

Correlations and Spectra, the Quantum Regression Theorem

13. Two time correlation and covariance functions

Continuing with the example of the atom in the cavity with coherent driving, we now turn to the computation of the two-time covariance of the intracavity field. If we work temporarily in the joint space of the system and the environment so that the evolution is specified by a total (time-independent) Hamiltonian $\hat{H}_{\text{tot}}$, the total density matrix evolves according to

$$\rho_{\text{tot}}(\tau) = \exp\left(-i\hat{H}_{\text{tot}}\tau\right)\rho_{\text{tot}}(0)\exp\left(i\hat{H}_{\text{tot}}\tau\right).$$

The time-evolution of the expectation value of an operator such as $\hat{a}^\dagger$ is then

$$\left\langle a^\dagger(\tau)\right\rangle = \text{trace}\left\{a^\dagger \exp\left(-i\hat{H}_{\text{tot}}\tau\right)\rho_{\text{tot}}(0)\exp\left(i\hat{H}_{\text{tot}}\tau\right)\right\}$$

where the operator $a^\dagger(\tau)$ on the left hand side is in the Heisenberg picture while those on the right are in the Schrödinger picture. If we now wish to find the two-time correlation function of the Heisenberg operator $a(t)$, this may be written in the Schrödinger picture by using the identity

$$\begin{aligned}\left\langle a^{\dagger}(t+\tau)\,a(t)\right\rangle &= \operatorname{trace}\left(a^{\dagger}(t+\tau)\,a(t)\,\rho_{\text{tot}}(0)\right)\\ &= \operatorname{trace}\left(\exp\left(i\hat{H}_{\text{tot}}(t+\tau)\right)a^{\dagger}\exp\left(-i\hat{H}_{\text{tot}}\tau\right)a\exp\left(-i\hat{H}_{\text{tot}}t\right)\rho_{\text{tot}}(0)\right)\\ &= \operatorname{trace}\left\{a^{\dagger}\exp\left(-i\hat{H}_{\text{tot}}\tau\right)a\rho_{\text{tot}}(t)\exp\left(i\hat{H}_{\text{tot}}\tau\right)\right\}.\end{aligned}$$

Formally, the right-hand side is identical to that involved in the calculation of $\left\langle a^{\dagger}(\tau)\right\rangle$, except that the initial condition $\rho_{\text{tot}}(0)$ is replaced by $a\rho_{\text{tot}}(t)$. If the operators whose correlation is required are in the space of the system alone, the integration of the differential equation for time-evolution may be carried out in the system space alone by utilizing the Liouvillian in place of the total Hamiltonian.

This last expression may be interpreted as follows. Starting with initial condition $a\rho_{\text{tot}}(t)$, this is propagated forward in time using the evolution operator for duration τ. After this process, the result is pre-multiplied by $a^{\dagger}$ and the trace is computed. If we consider taking the partial trace over the reservoir and re-interpret ρ as the *system* density operator rather than the total density operator, the evolution in time through τ is performed using the exponential of the Liouvillian, i.e., $\exp(\mathcal{L}\tau)$. This is the essence of the quantum regression theorem [5]. In the steady-state, as $t\to\infty$, the correlation function depends only on τ. We can thus compute the stationary two-time correlation by

1. Setting the initial condition for the differential equation solver to $a\rho_{\text{ss}}$ where ρ_{ss} is the steady state density matrix which is the solution of $\mathcal{L}\rho=0$,
2. Propagating the initial condition through time τ, representing the answer as an exponential series, using the function `ode2es`.
3. Finding the trace of $a^{\dagger}$ multiplied by the solution of the differential equation. This can be thought of as an expectation value with a “state” given by the solution of the differential equation.

In view of these considerations, we see that the following code calculates $\left\langle a^{\dagger}(t+\tau)\,a(t)\right\rangle$ and $\left\langle a^{\dagger}(t+\tau),a(t)\right\rangle$ as exponential series in τ.

```
function [corrES,covES] = probcorr(E,kappa,gamma,g,wc,w0,wl,N)
%
% [corrES,covES] = probcorr(E,kappa,gamma,g,wc,w0,wl,N)
%  returns the two-time correlation and covariance of the intracavity
%  field as exponential series for the problem of a coherently driven
%  cavity with a two-level atom
%
%  E = amplitude of driving field, kappa = mirror coupling,
%  gamma = spontaneous emission rate, g = atom-field coupling,
%  wc = cavity frequency, w0 = atomic frequency, wl = driving field frequency,
%  N = size of Hilbert space for intracavity field (zero to N-1 photons)
%
% Same code as in probss up to line:
L = LH+L1+L2;
% Find steady state density matrix and field
rhoss = steady(L);
ass = expect(a,rhoss);
% Initial condition for regression theorem
arho = a*rhoss;
% Solve differential equation with this initial condition
solES = ode2es(L,arho);
% Find trace(a' * solution)
corrES = expect(a',solES);
% Calculate the covariance by subtracting product of means
covES = corrES - ass'*ass;
```

In the last line of this example, the correlation is converted to a covariance by subtracting the constant $\left\langle a^{\dagger}\right\rangle\langle a\rangle$. This is a scalar which is automatically converted into an exponential series.

Figure 4 shows the two-time covariance function for the system for on-resonance driving for two values of atom-field coupling g. For small values of g, the covariance falls monotonically whereas for larger values of g, there are oscillations.

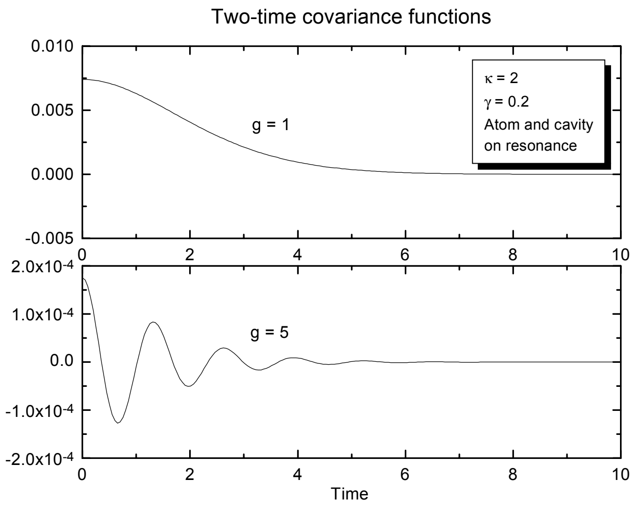

Figure 4: Two-time covariance function for different atom-field interaction strengths

The method described above may similarly used to compute other correlation functions involving two different operators, such as $\langle a(t_1)\, b(t_2)\rangle$.

14. Calculation of the power spectrum

Given a stationary two-time covariance function $\phi(\tau) = \left\langle a^\dagger(t+\tau), a(t)\right\rangle$ expressed as an exponential series

$$\phi(\tau) = \begin{cases} \sum_j c_j \exp(s_j \tau) & \text{for } \tau \geq 0 \\ \phi^*(-\tau) & \text{for } \tau < 0 \end{cases}$$

the power spectrum is simply given by

$$\Phi(\omega) = 2\,\mathrm{Re} \sum_j \frac{c_j}{i\omega - s_j}$$

The toolbox function `esspec(es,wlist)` evaluates the power spectrum from an exponential series at the frequencies specified in `wlist`. The result is an array of quantum objects of the same size as `wlist.` In the special case when all the elements of `es` are scalars, the result is an array of doubles.

Figure 5 shows the power spectrum calculated for the system for the parameters shown in 4. Note that the interaction picture is based on a frame rotating at the frequency of the driving field, so that the frequency axis is relative to the driving frequency. The example files `xprobcorr` and `probcorr` illustrate the above calculation.

15. Force and momentum diffusion of a stationary two-level atom in a standing-wave light field

We now consider the problem of a stationary two-level atom in a standing wave light field which is treated classically [6]. Due to the fluctuations in the dipole force, there will be momentum diffusion causing an increase in the mean-square momentum with time. Consider a standing wave in the z direction with the effective Rabi frequency being given by

$$g(z) = 2g_0 \cos k_L z$$

The Hamiltonian for this problem in an interaction picture rotating at the frequency of the light field is

$$H = -\Delta\sigma_+\sigma_- - i(g^*\sigma_- - g\sigma_+)$$

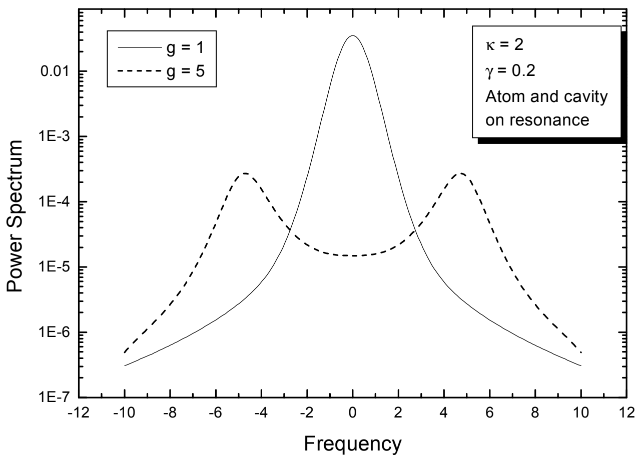

Figure 5: Power spectra calculated from two-time covariance functions

where Δ is the detuning of the laser relative to the atomic transition. The only dissipative is spontaneous emission with associated collapse operator $C_1 = \sqrt{\gamma}\sigma_-$.

At any time, if the dipole moment of the atom is $\mathbf{d}$ and the electric field at the location of the atom is $\mathbf{E}$, the force experienced by the atom in the light field is given by

$$f_i = \frac{\partial}{\partial x_i}\left(d_x E_x + d_y E_y + d_z E_z\right).$$

In the case of the one-dimensional standing wave, we find that

$$\begin{aligned} f = f_z &= i\left(\left(\nabla g^*\right)\sigma_- - \left(\nabla g\right)\sigma_+\right) \\ &= -2ik_L g_0\left(\sigma_- - \sigma_+\right)\sin k_L z \end{aligned}$$

In order to find the spatial variation of the force when the atom is stationary at z_0, we simply need to find the steady-state values of $\langle\sigma_+\rangle$ and $\langle\sigma_-\rangle$. The force on the atom at that position is then

$$\langle f\rangle = \langle f_z\rangle = i\left(\left(\nabla g^*\right)\langle\sigma_-\rangle - \left(\nabla g\right)\langle\sigma_+\rangle\right).$$

The momentum diffusion constant is defined as half the rate of increase of the momentum variance with time, i.e.,

$$D_p = \frac{1}{2}\frac{d}{dt}\left\langle\left(p - \langle p\rangle\right)^2\right\rangle.$$

This may simply be related to the force acting on the atom,

$$\begin{aligned} D_p &= \frac{1}{2}\left\langle\left(p - \langle p\rangle\right)\cdot\frac{d}{dt}\left(p - \langle p\rangle\right) + \frac{d}{dt}\left(p - \langle p\rangle\right)\cdot\left(p - \langle p\rangle\right)\right\rangle. \\ &= \frac{1}{2}\langle f\cdot p + p\cdot f\rangle - \langle f\rangle\cdot\langle p\rangle \end{aligned}$$

At a specific time t, assuming that $p(-\infty) = 0$,

$$\begin{aligned} D_p(t) &= \frac{1}{2}\left\langle f(t) \cdot \int_0^\infty f(t-\tau)\, d\tau + \int_0^\infty f(t-\tau)\, d\tau \cdot f(t) \right\rangle \\ &- \langle f(t) \rangle \cdot \left\langle \int_0^\infty f(t-\tau)\, d\tau \right\rangle \\ &= \mathrm{Re} \int_0^\infty \left(\langle f(t) \cdot f(t-\tau) \rangle - \langle f(t) \rangle \cdot \langle f(t-\tau) \rangle \right) d\tau \\ &= \mathrm{Re} \int_0^\infty \langle f(t), f(t-\tau) \rangle \, d\tau \end{aligned}$$

where the real part arises because the two-time force correlation function is hermitian symmetric in the time difference τ. Thus D_p is given by the real part of the time-integral of the force covariance.

The methods developed in the previous examples can be applied directly to this problem. Our strategy is to find the steady-state density matrix for the atom at the position z_0 and to use this to compute the two-time force covariance $\phi(\tau)$ as an exponential series. Once in the form of an exponential series

$$\phi(\tau) = \sum_k \phi_k \exp(s_k t)$$

we see that if $s_k < 0$ for all k, the integral is simply given by

$$\int_0^\infty \phi(\tau)\, d\tau = -\sum_k \frac{\phi_k}{s_k}$$

The toolbox function `esint(ES)` integrates an exponential series from zero to infinity. This function may also be used with extra arguments, `esint(ES, t1)` integrates a series from t_1 to infinity and `esint(ES, t1, t2)` integrates a series from t_1 to t_2.

The file `probstatic.m` listed below calculates the force, momentum diffusion constant and spontaneous emission rate for a stationary atom in a standing wave. The corresponding driver file is called `xprobstatic.m`. Figure 6 shows these quantities for a standing wave with $g_0 = 1$, $\Delta = 2$, $\gamma = 1$. Note that this momentum diffusion does *not* include the contribution due to random recoil during spontaneous emission. This additional term is proportional to the spontaneous emission rate and depends on the polar pattern of spontaneous emission.

```
function [force,diffuse,spont] = probstatic(Delta,gamma,g0,kL,z)
% [force,diffuse,spont] = probstatic(Delta,gamma,g0,kL,z) calculates the force,
%  momentum diffusion and spontaneous emission rate for a stationary
%  two-level atom in a standing wave light field.
%
%  Delta = laser freq - atomic freq
%  gamma = atomic decay
%  g0    = coupling between atom and light, g(z) = 2*g0*cos(kL*z)
%  kL    = wavenumber of light
%  z     = position in standing wave
g = 2*g0*cos(kL*z);
gg = -2*g0*kL*sin(kL*z);
%
sm = sigmam;
%
H = -Delta*sm'*sm - i*(g'*sm - sm'*g);
F = i*(gg'*sm - gg*sm');
C1 = sqrt(gamma)*sm;
%
C1dC1 = C1'*C1;
LH = -i*(spre(H)-spost(H));
L1 = spre(C1)*spost(C1')-0.5*spre(C1dC1)-0.5*spost(C1dC1);
L = LH + L1;
```

```
% Find steady state density matrix and force
rhoss = steady(L);
force = expect(F,rhoss);
spont = expect(C1'*C1,rhoss);
% Initial condition for regression theorem
Frho = spre(F)*rhoss;
% Solve differential equation with this initial condition
solES = ode2es(L,Frho);
% Find trace(F * solES)
corrES = expect(F,solES);
% Calculate the covariance by subtracting product of means
covES = corrES-force^2;
% Integrate the force covariance and return real part
diffuse = real(esint(covES));
```

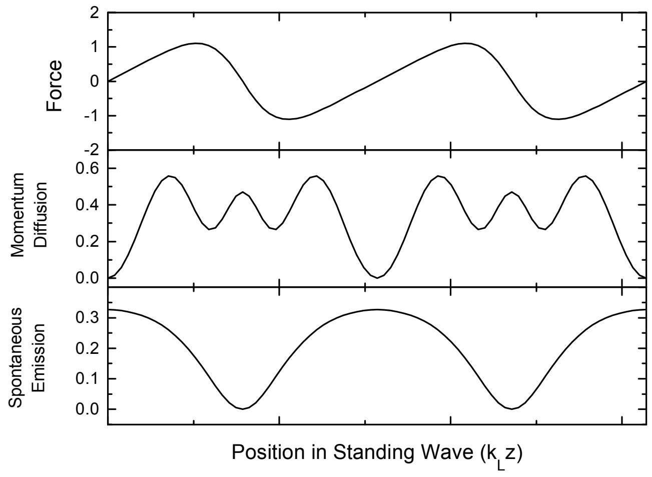

Figure 6: Force, momentum diffusion and spontaneous emission from a stationary two-level atom in a standing-wave light field.

Force on a moving two-level atom

16. Introduction to Matrix Continued Fractions

In this example, we introduce a problem in which the Liouvillian matrix is time-dependent, so that it is no longer possible to write the answer as a finite exponential series. If the time-dependence is simply the sum of a constant and a sinusoidally varying term, it is sometimes possible to use the method of matrix continued fractions (see e.g., Risken [7]) in order to obtain special solutions of the equation. This method is especially useful for finding the force on an atom moving in a light field with an intensity or polarization gradient. We consider a master equation of the form

$$\frac{d\rho}{dt} = \left(\mathcal{L}_0 + \mathcal{L}_1 \exp\left(i\omega t\right) + \mathcal{L}_{-1} \exp\left(-i\omega t\right)\right)\rho$$

and ask for solutions $\rho\left(t\right)$ which may be expressed as an exponential series of the form

$$\rho\left(t\right) = \sum_{n=-\infty}^{\infty} \rho_n \exp\left(in\omega t\right).$$

In the application of finding the force on a moving atom, the atom is taken to be infinitely massive and is dragged at constant velocity v through a sinusoidally varying light field. Since the Hamiltonian (and the force) depends on the position in the atom, the Liouvillian $\mathcal{L}$ contains terms which vary sinusoidally in space and hence in time with $\omega = k_L v$ where k_L is the wave number of the light field. In general, for certain special values of ω_0, there are solutions to the differential equation of the form

$$\rho(t) = \exp(i\omega_0 t) \sum_{n=-\infty}^{\infty} \rho_n(\omega_0) \exp(in\omega t)$$

We wish to compute the exponential series for the case $\omega_0 = 0$, so that $\rho(t)$ is given by a Fourier series. Substituting the supposed solution into the master equation yields

$$\begin{aligned}\sum_{n=-\infty}^{\infty} in\omega\rho_n(\omega_0)\exp(in\omega t) = \sum_{n=-\infty}^{\infty} &(\mathcal{L}_0 + \mathcal{L}_1 \exp(i\omega t) + \mathcal{L}_{-1}\exp(-i\omega t)) \\ &\times \rho_n(\omega_0)\exp(in\omega t)\end{aligned}$$

Equating the coefficients of the series yields the tridiagonal recursion relation

$$0 = \mathcal{L}_1 \rho_{n-1} + (\mathcal{L}_0 - in\omega\mathbf{I})\rho_n + \mathcal{L}_{-1}\rho_{n+1}$$

Such a recursion may often be solved by the method of continued fractions. Suppose that for $n \geq 1$, it is possible to find a sequence of matrices $\mathbf{S}_n$ with the property that $\rho_n = \mathbf{S}_n\rho_{n-1}$. If these matrices exist, then by the recursion relationship, we require

$$[\mathcal{L}_1 + (\mathcal{L}_0 - in\omega\mathbf{I})\mathbf{S}_n + \mathcal{L}_{-1}\mathbf{S}_{n+1}\mathbf{S}_n]\rho_{n-1} = 0.$$

This can be done simply by setting

$$\mathcal{L}_1 + (\mathcal{L}_0 - in\omega\mathbf{I})\mathbf{S}_n + \mathcal{L}_{-1}\mathbf{S}_{n+1}\mathbf{S}_n = 0$$

or

$$\mathcal{L}_1 + [(\mathcal{L}_0 - in\omega\mathbf{I}) + \mathcal{L}_{-1}\mathbf{S}_{n+1}]\mathbf{S}_n = 0$$

$$\mathbf{S}_n = -[(\mathcal{L}_0 - in\omega\mathbf{I}) + \mathcal{L}_{-1}\mathbf{S}_{n+1}]^{-1}\mathcal{L}_1.$$

This expresses $\mathbf{S}_n$ in terms of $\mathbf{S}_{n+1}$. We start with some large N and set $\mathbf{S}_N = 0$. Using this recursion we obtain the matrix $\mathbf{S}_1$. This is called a continued fraction because if we consider the case when $\mathbf{S}$ and $\mathcal{L}$ are in fact a scalars, and if we write $a_n = -\mathcal{L}_1$, $b_n = \mathcal{L}_0 - in\omega\mathbf{I}$ and $c_n = \mathcal{L}_{-1}$, the recursion may be written as

$$\mathbf{S}_n = \frac{a_n}{b_n + c_n\mathbf{S}_{n+1}}$$

and so

$$\mathbf{S}_1 = \cfrac{a_1}{b_1 + \cfrac{c_1 a_2}{b_2 + \cfrac{c_2 a_3}{b_3 + \ldots}}}$$

which is a classical continued fraction.

Similarly, if for $n \leq -1$, we can find a sequence of matrices $\mathbf{T}_n$ with the property that $\rho_n = \mathbf{T}_n\rho_{n+1}$, we require that

$$[\mathcal{L}_1\mathbf{T}_{n-1}\mathbf{T}_n + (\mathcal{L}_0 - in\omega\mathbf{I})\mathbf{T}_n + \mathcal{L}_{-1}]\rho_{n+1} = 0$$

which can be done by setting

$$\mathcal{L}_1\mathbf{T}_{n-1}\mathbf{T}_n + (\mathcal{L}_0 - in\omega\mathbf{I})\mathbf{T}_n + \mathcal{L}_{-1} = 0$$

from which we find

$$\mathbf{T}_n = -[(\mathcal{L}_0 - in\omega\mathbf{I}) + \mathcal{L}_1\mathbf{T}_{n-1}]^{-1}\mathcal{L}_{-1}.$$

This expresses $\mathbf{T}_n$ in terms of $\mathbf{T}_{n-1}$. We start with some large N and set $\mathbf{T}_{-N} = 0$. Using this recursion we obtain the matrix $\mathbf{T}_{-1}$.

Having obtained $\mathbf{S}_1$ and $\mathbf{T}_{-1}$, we return to the tridiagonal relation for $n = 0$, namely

$$0 = \mathcal{L}_1\rho_{-1} + \mathcal{L}_0\rho_0 + \mathcal{L}_{-1}\rho_1$$

and substitute for ρ_1 and ρ_{-1} in terms of ρ_0, obtaining

$$0 = (\mathcal{L}_1\mathbf{T}_{-1} + \mathcal{L}_0 + \mathcal{L}_{-1}\mathbf{S}_1)\,\rho_0$$

If the matrix $\mathcal{L}_1\mathbf{T}_{-1} + \mathcal{L}_0 + \mathcal{L}_{-1}\mathbf{S}_1$ is singular, which indicates that the choice $\omega_0 = 0$ does in fact give a solution to the original equation, this enables us to calculate ρ_0 and subsequently ρ_n for all other values of n, provided that we have stored the appropriate $\mathbf{S}_n$ and $\mathbf{T}_n$.

The toolbox function `mcfrac` uses the matrix continued fraction algorithm to find the exponential series for the density matrix when given the exponential series for the Liouvillian. This function also takes two additional input parameters, one to indicate the value of N at which $\mathbf{S}_N$ and $\mathbf{T}_{-N}$ are set to zero in order to start the recursion, and the other to indicate how many terms of the exponential series for ρ are required, since in principle, this is an infinite series.

17. Setting up the problem

We consider a two level atom which is being dragged at constant velocity v through a light field which is oriented along the z direction. Due to the motion, the position of the atom at time t is $z = vt$. The electric field experienced by the atom is $g(z)$, normalized such that the interaction energy of the atom within the field is

$$V(z) = -\left(g^*(z)\,\sigma_- + g(z)\,\sigma_+\right)$$

The force exerted on the atom due to the light field is

$$F(z) = \left(\nabla g^*(z)\,\sigma_- + \nabla g(z)\,\sigma_+\right).$$

Since $z = vt$, each of these becomes a function of time. For a running wave, $g(z)$ is a complex exponential $g(z) = g_0 \exp(ik_L z)$, whereas for a standing wave, $g(z) = 2g_0 \cos k_L z$. When using the toolbox, functions which are the sum of terms with exponential time-dependence are represented by exponential series. So for example, if due to the motion, the atom sees $g(t) = 2g_0 \cos k_L vt$, this is an exponential series with two terms $g(t) = g_0 \exp(ik_L vt) + g_0 \exp(-ik_L vt)$. In order to specify this using the toolbox, we use

```
g = eseries(g0,i*kL*v) + eseries(g0,-i*kL*v);
```

With this definition, the contribution to the Hamiltonian due to the optical potential may be calculated as an exponential series:

```
A = sigmam;
H = -(g'*A + A'*g) + H0;
```

Similarly, the two lines given below compute the exponential series for the force operator:

```
gg = eseries(i*kL*g0,i*kL*v) + eseries(-i*kL*g0,-i*kL*v);
F = gg'*A + gg*A';
```

The Hamiltonian and Liouvillian are also required in the form of exponential series. In the code below, we see that `spre(H)` and `spost(H)` convert the operator exponential series for `H` into superoperator exponential series required for the computation of `L`. Although it is straightforward to manipulate exponential series using the toolbox, it is usually best to do as much of the calculation as possible with constant matrices, converting these to exponential series only when absolutely necessary, since calculations involving series are somewhat slower than those involving constants. Thus for example, in the typical Liouvillian with a time-dependent Hamiltonian,

$$\mathcal{L}\rho = -i\,[H(t), \rho] + \sum_n C_n\rho C_n^\dagger - \frac{1}{2}C_n^\dagger C_n\rho - \frac{1}{2}\rho C_n^\dagger C_n,$$

the loss terms involving the collapse operators are not time-dependent. We would then calculate these collapse operators using ordinary matrices and form the entire superoperator

$$\sum_n C_n\rho C_n^\dagger - \frac{1}{2}C_n^\dagger C_n\rho - \frac{1}{2}\rho C_n^\dagger C_n$$

as an ordinary matrix (using `spre` and `spost`, just as before). Finally, this superoperator would be converted into an exponential series and combined with the terms involving the Hamiltonian. In the line `L=LH+L1` below, `LH` is an exponential series while `L1` is a quantum object. When they are added together, the result is automatically converted to an exponential series.

When the Liouvillian can be expressed as an exponential series of the form

$$\mathcal{L} = \mathcal{L}_0 + \mathcal{L}_1 \exp\left(i\omega t\right) + \mathcal{L}_{-1} \exp\left(-i\omega t\right),$$

(where for this example $\omega = k_L v$), there is a special toolbox routine

`rhoES= mcfrac(LES,Ncf,Nes)`

which uses the matrix continued fraction method outlined above to find the central terms of the exponential series for ρ, namely if

$$\rho = \sum_{k=-\infty}^{\infty} \rho_k \exp\left(ik\omega t\right)$$

the routine finds ρ_k for $k = -$`Nes`$,...,$`Nes` by using `Ncf` terms of a matrix continued fraction expansion. Since the force operator and the density matrix are exponential series, the expected value of the force is also an exponential series. In order to get the correct value for the time-averaged force (i.e., averaged over travel through one cycle of the light), we need the terms ρ_1 and ρ_{-1} for the density matrix, since the force operator contains time dependencies of the form $\exp\left(\pm i\omega t\right)$. More precisely, since

$$\begin{aligned} F\left(z = vt\right) &= i\left(\nabla g^*\left(vt\right)\sigma_- - \nabla g\left(vt\right)\sigma_+\right) \\ &= F_1 \exp\left(i\omega t\right) + F_{-1}\exp\left(-i\omega t\right), \end{aligned}$$

the expectation value of F is

$$\begin{aligned} \mathrm{Tr}\left(\rho F\right) &= \mathrm{Tr}\left(\sum_{k=-\infty}^{\infty} \rho_k \exp\left(ik\omega t\right)\left(F_1 \exp\left(i\omega t\right) + F_{-1}\exp\left(-i\omega t\right)\right)\right) \\ &= \sum_{k=-\infty}^{\infty} \left[\mathrm{Tr}\left(\rho_{k-1}F_1\right) + \mathrm{Tr}\left(\rho_{k+1}F_{-1}\right)\right]\exp\left(ik\omega t\right) \end{aligned}$$

and the time-averaged force is the term with $k = 0$, namely

$$\mathrm{Tr}\left(\rho_{-1}F_1\right) + \mathrm{Tr}\left(\rho_1 F_{-1}\right).$$

The toolbox function `expect` has been overloaded to calculate the expectation values when both the operator and the state are exponential series. The result is also an exponential series. The file `prob2lev.m` calculates the cycle-averaged force on a two-level atom and this is listed below. The Hamiltonian used is

$$H_{int} = -\Delta\sigma_+\sigma_- - \left(g^*\left(z\right)\sigma_- + g\left(z\right)\sigma_+\right)$$

where, as usual Δ is the laser frequency minus the atomic transition frequency. Notice how the term in the Hamiltonian $-\Delta\sigma_+\sigma_-$ is synthesized using the projection operators onto the excited state, since $\sigma_+\sigma_- = \left|e\right\rangle\left\langle e\right|$. The result of the calculation is shown in Figure 7 for parameter values corresponding to Figure 1a of Minogin and Serimaa [8]. Taking into account the difference in the notation, the results are in agreement. Note that the result `force` is an exponential series, and that we require the cycle-averaged term associated with $s = 0$. This is obtained using the notation `force(0).ampl`.

```
function force = prob2lev(v,kL,g0,Gamma,Delta)
% force = prob2lev(v,kL,g0,Gamma,Delta)
%
% Calculates the force on a moving two-level atom travelling at
%  speed v in a standing wave with wavenumber kL. The coupling
%  strength is g0, the atomic spontaneous emission rate is
%  Gamma and the detuning is Delta
%
Ne = 1; Ng = 1; Nat = Ne+Ng;
% Form the lowering operator
```

```
A = sigmam;
% Time-dependent couplings for atom and field
g   = eseries(g0,i*kL*v) + eseries(g0,-i*kL*v);
gg  = eseries(i*kL*g0,i*kL*v) + eseries(-i*kL*g0,-i*kL*v);
% Form interaction picture Hamiltonian
% Excited state terms
excited = basis(Nat,1:Ne);
H0 = -Delta*sum(excited*excited');
% Interaction with light
H = -(g'*A + A'*g) + H0;
% Force operator
F = gg'*A + gg*A';
% Collapse operators
C1 = sqrt(Gamma)*A; C1dC1 = C1'*C1;
% Liouvillians
L1 = spre(C1)*spost(C1')-0.5*spre(C1dC1)-0.5*spost(C1dC1);
% Calculate total Liouvillian as an exponential series
LH = -i*(spre(H)-spost(H));
L = LH + L1;
% Matrix continued fraction calculation
rho = mcfrac(L,20,1);
force = expect(F,rho);
```

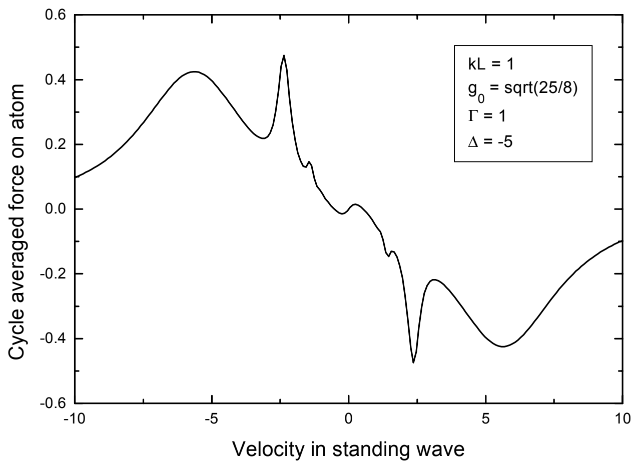

Figure 7: Cycle-averaged force on a moving two-level atom in a standing wave evaluated using the method of matrix continued fractions.

Multilevel atoms

In this example, we show how the previous calculation for two-level atoms may be straightforwardly extended to multiple-level atoms in light fields with polarization gradients, rederiving the results found by Dalibard and Cohen-Tannoudji [9]. We consider two cases, an atom with a $J_g = 1$ to $J_e = 2$ transition in counter-propagating (σ_+, σ_-) light and an atom with a $J_g = \frac{1}{2}$ to $J_e = \frac{3}{2}$ transition in a light field with counter-propagating lin$\perp$lin polarization.

With a multilevel atom, the Hilbert space of the atom is of dimension $N_a = N_e + N_g$ where $N_e = 2J_e+1$ and $N_g = 2J_g+1$. We represent this by a vector whose first N_e components are the excited state amplitudes followed by the N_g ground state amplitudes. Since the light fields are treated classically, this is the full

Hilbert space. Instead of a single lowering operator σ_- for the two-level atom, there are three lowering operators which we shall denote by A_{-1}, A_0, and A_1 which respectively couple to light of σ_-, π and σ_+ polarizations. Each of these involve linear combinations of transitions between excited and ground levels, with weighting factors given by the appropriate Clebsch-Gordon coefficients. The matrix representation of one of these matrices of size $(N_e + N_g) \times (N_e + N_g)$ may be compared with the two-level lowering operator σ_- of size 2×2 as follows

$$\sigma_- = \begin{array}{cc} \begin{array}{cc} \mathrm{e} & \mathrm{g} \end{array} & \\ \left(\begin{array}{cc} 0 & 0 \\ 1 & 0 \end{array} \right) & \begin{array}{c} \mathrm{e} \\ \mathrm{g} \end{array} \end{array} \qquad A_p = \begin{array}{cc} \begin{array}{cc} \mathrm{e} & \mathrm{g} \end{array} & \\ \left(\begin{array}{cc} \mathbf{0} & \mathbf{0} \\ \mathbf{a}_p & \mathbf{0} \end{array} \right) & \begin{array}{c} \mathrm{e} \\ \mathrm{g} \end{array} \end{array}$$

where $\mathbf{a}_p$ is a matrix of size $N_g \times N_e$ containing the Clebsch-Gordon transitions ($p = 0, \pm 1$) and the bold face zeros are block matrices of the appropriate sizes. The toolbox routine `murelj` calculates the submatrices $\mathbf{a}_-, \mathbf{a}_0$ and $\mathbf{a}_+$ for an arbitrary J_g to J_e transition and the routine `murelf` calculates the corresponding matrices for an (F_g, J_g) to (F_e, J_e) transition when the hyperfine structure induced by nuclear spin I is important.

Associated with light fields travelling in the z direction, there are functions $\Omega_+(z)$ and $\Omega_-(z)$ representing the Rabi frequencies for σ_+ and σ_- polarizations respectively. The light-induced portion of the interaction Hamiltonian is given by

$$H_{\text{light-atom}} = -\frac{1}{2}\left(\Omega_+(z) A_+ + \Omega_-(z) A_-\right) + \text{h.c.}$$

and the corresponding force operator is

$$F = \frac{1}{2}\left(\nabla\Omega_+(z) A_+ + \nabla\Omega_-(z) A_-\right) + \text{h.c.}$$

The collapse operators representing spontaneous emission into each of the possible polarizations are $\sqrt{\gamma}A_-$, $\sqrt{\gamma}A_0$ and $\sqrt{\gamma}A_+$.

Let us consider first the case of counter-propagating (σ_+, σ_-)light. The listing below (`probmulti1.m`) shows how the exponential series for the force may be found. Notice how similar this program is to `prob2lev.m` for the two-level atom. The noteworthy features are the calculation of the three lowering operators A_p and the formation of the exponential series for the atom-light interaction and the force operator. In place of `g` (and its gradient `gg`) for the single polarization which interacts with the two level atom, there are `gp` and `gm` (and their gradients `ggp` and `ggm`) specifying the amplitudes of the σ_+ and σ_- circular polarizations. For each of the polarizations, there is a travelling wave, the σ_+ light propagating towards $+z$ and the σ_- light propagating towards $-z$. Another noteworthy point is the formation of the loss Liouvillian for all three collapse channels together by using a quantum object array. In Figure 8 the cycle averaged force is shown as a function of the velocity. This is the same as shown in Figure 8 of the paper by Dalibard and Cohen-Tannoudji [?], taking into account the factor of two which they include in their units of the force.

```
function force = probmulti1(v,kL,Jg,Je,Omega,Gamma,Delta)
% Calculate the force on an atom with light fields in a
%  sigma+ sigma- configuration.
Ne = 2*Je+1; Ng = 2*Jg+1; Nat = Ne+Ng;
% Form the lowering operators for various polarizations of light
[am,a0,ap] = murelj(Jg,Je);
Am = sparse(Nat,Nat); Am(Ne+1:Ne+Ng,1:Ne)=am; Am = qo(Am);
A0 = sparse(Nat,Nat); A0(Ne+1:Ne+Ng,1:Ne)=a0; A0 = qo(A0);
Ap = sparse(Nat,Nat); Ap(Ne+1:Ne+Ng,1:Ne)=ap; Ap = qo(Ap);
% gp = 0.5*Omega*exp(i*kL*z);
% gm = 0.5*Omega*exp(-i*kL*z);
% ggp =  0.5*i*kL*Omega*exp(i*kL*z);
% ggm = -0.5*i*kL*Omega*exp(-i*kL*z);
% Due to motion, these become exponential series in time (z=vt)
gp  = eseries(0.5*Omega,i*kL*v);
gm  = eseries(0.5*Omega,-i*kL*v);
ggp = eseries(0.5*i*kL*Omega,i*kL*v);
```

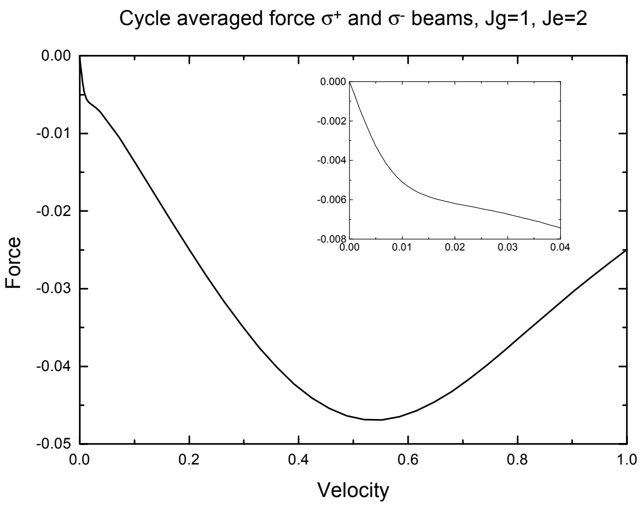

Figure 8: Cycle averaged force on an atom with a $J_g = 1$ to $J_e = 2$ transition in counterpropagating (σ_+, σ_-) light fields. Parameters used are $\gamma = 1$, $\Omega = 0.25$, $\Delta = -0.5$ and $k_L = 1$.

```
ggm = eseries(-0.5*i*kL*Omega,-i*kL*v);
% Form interaction picture Hamiltonian
% Excited state terms
excited = basis(Nat,1:Ne);
H0 = -Delta*sum(excited*excited');
% Interaction with sigma+ light
Hp = -(gp'*Ap + gp*Ap');
% Interaction with sigma- light
Hm = -(gm'*Am + gm*Am');
%
H  = H0 + Hp + Hm;
% Force operator
Fp = (ggp'*Ap + ggp*Ap');
Fm = (ggm'*Am + ggm*Am');
F = Fp + Fm;
% Collapse operators as an array
C = sqrt(Gamma)*[Am,A0,Ap];
% Loss Liouvillians
Lloss = spre(C)*spost(C')-0.5*spre(C'*C)-0.5*spost(C'*C);
% Calculate total Liouvillian
L = -i*(spre(H)-spost(H)) + sum(Lloss);
% Matrix continued fraction calculation
rho = mcfrac(L,20,1);
force = expect(F,rho);
```

The corresponding file for treating lin⊥lin polarization are shown in the following listing (`probmulti2.m`). Note that the changes are limited to the definitions of `gp` and `gm` (and their gradients `ggp` and `ggm`) of the light fields. In Figure 9 the results are shown for this case, using parameters which are the same as in Figure 7 of Dalibard and Cohen-Tannoudji [?].

```
function force = probmulti2(v,kL,Jg,Je,Omega,Gamma,Delta)
% Calculate the force on an atom with light fields in a
%  lin-perp-lin configuration.
Ne = 2*Je+1; Ng = 2*Jg+1; Nat = Ne+Ng;
% Form the lowering operators for various polarizations of light
```

```
[am,a0,ap] = murelj(Jg,Je);
Am = sparse(Nat,Nat); Am(Ne+1:Ne+Ng,1:Ne)=am; Am = qo(Am);
A0 = sparse(Nat,Nat); A0(Ne+1:Ne+Ng,1:Ne)=a0; A0 = qo(A0);
Ap = sparse(Nat,Nat); Ap(Ne+1:Ne+Ng,1:Ne)=ap; Ap = qo(Ap);
% gp = sqrt(0.5)*Omega*sin(kL*z);
% gm = sqrt(0.5)*Omega*cos(kL*z);
% ggp = sqrt(0.5)*kL*Omega*cos(kL*z);
% ggm = -sqrt(0.5)*kL*Omega*sin(kL*z);
% Due to motion, these become exponential series in time (z=vt)
gp  = 0.5*(eseries(sqrt(0.5)*i*Omega,i*kL*v)    - eseries(sqrt(0.5)*i*Omega,-i*kL*v));
gm  = 0.5*(eseries(sqrt(0.5)*Omega,i*kL*v)      + eseries(sqrt(0.5)*Omega,-i*kL*v));
ggp = 0.5*(-eseries(sqrt(0.5)*kL*Omega,i*kL*v)  - eseries(sqrt(0.5)*kL*Omega,-i*kL*v));
ggm = 0.5*(eseries(sqrt(0.5)*i*kL*Omega,i*kL*v) - eseries(sqrt(0.5)*i*kL*Omega,-i*kL*v));
%
% rest of code is identical to probmulti1
%
```

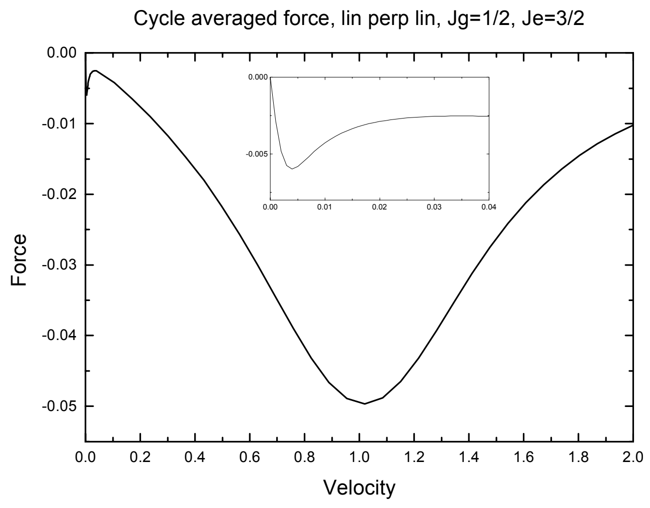

Figure 9: Cycle-averaged force for an atom with a $J_g = \frac{1}{2}$ to $J_e = \frac{3}{2}$ transition in a lin⊥lin light field configuration. Parameters are $\gamma = 1$, $\Omega = 0.3$, $\Delta = -1$, $k_L = 1$.

Larger problems, numerical integration

So far, the problems we have considered have been small enough that it has been possible to carry out operations such as the diagonalization or the inversion of the Liouvillian matrix. Since the size of the Liouvillian matrix becomes very large as the number of states is increased, these techniques are less useful for larger problems. The toolbox thus provides for the numerical integration of the equations of motion for a variety of problems.

1. Even when the Liouvillian matrix $\mathcal{L}$ is too large to be inverted or diagonalized (operations whose computational load scale with N^3 or N^4 when $\mathcal{L}$ is an $N \times N$ matrix), it is often possible to integrate the master equation numerically

$$\frac{d\rho}{dt} = \mathcal{L}\rho$$

starting with some initial condition $\rho(0) = \rho_0$. This is especially the case when $\mathcal{L}$ is a sparse matrix, so that the load associated with finding $\mathcal{L}\rho$ may be quite small. The strategy we adopt is to exploit the techniques used in the toolbox and the sparse matrix algorithms of Matlab to form the matrix $\mathcal{L}$ and subsequently to carry out the numerical integration of the equations of motion using a compiled routine written in C. We use Matlab to write out a file containing $\mathcal{L}$ and ρ_0 which can be read in by the differential equation solver. The solution is written to another file which may then be read back into Matlab where expectation values and other statistics may be computed. It is also relatively easy to treat the case in which the Liouvillian matrix $\mathcal{L}$ is explicitly time-dependent, and this is also provided for in the toolbox.

2. If there are too many components in the density matrix, an alternative approach is to consider the state vector ψ and the Schrödinger equations of motion, which in the case of no dissipation may be written in the form

$$i\frac{d\psi}{dt} = H\left(t\right)\psi$$

   This is again an initial value problem, with $\psi\left(0\right) = \psi_0$ being given. The routine will compute ψ at a specified list of times.

3. When dissipation is present, but a wave function approach is still required, it is possible to use a quantum trajectory (or quantum Monte Carlo) approach. In this approach, each trajectory consists of continuous evolution according to a modified Scrödinger equation of the form

$$i\frac{d\psi}{dt} = H_{\text{eff}}\left(t\right)\psi,$$

   which is punctuated by "quantum jumps" at which the state vector changes discontinuously according to

$$\psi\left(t_+\right) = \frac{C_k\left(t\right)\psi\left(t_-\right)}{\left\|C_k\left(t\right)\psi\left(t_-\right)\right\|},$$

   where $\{C_k\left(t\right)\}$ are a family of collapse operators, the probability of the kth collapse in an interval dt being given by

$$\left\|C_k\left(t\right)\psi\left(t\right)\right\|^2 dt.$$

   The effective non-Hermitian Hamiltonian is given by

$$H_{\text{eff}}\left(t\right) = H\left(t\right) - \frac{i}{2}\sum_k C_k^\dagger\left(t\right)C_k\left(t\right).$$

   The above procedure is repeated for a large number of trajectories. For *each* trajectory, the initial condition is $\psi\left(0\right) = \psi_0$ and the solution takes the form of the stochastic wave function $\psi\left(t\right)$ recorded at a pre-specified list of times together with a *classical record* which is a list of times at which a collapse occured together with the index of the collapse.

4. Since the use of a quantum trajectory approach often requires operator expectation values to be averaged over many trajectories in order to obtain ensemble predictions, recording the evolution of the stochastic wave function for each of these trajectories can use a large amount of disc space. In order to avoid this problem, the quantum trajectories simulation can also be run while specifying a list of operators represented by matrices $A_k\left(t\right)$. For each trajectory, the expectation of these operators are computed using

$$\left\langle A_k\left(t\right)\right\rangle = \frac{\left\langle\psi\left(t\right)\right|A_k\left(t\right)\left|\psi\left(t\right)\right\rangle}{\left\langle\psi\left(t\right)\right|\left.\psi\left(t\right)\right\rangle}$$

   These are averaged over trajectories, and the averaged results are written out after each of a specified number of trajectories.

All of these problems may be solved using a routine called `solvemc` which is an external program spawned as a subprocess from within Matlab. The programme `solvemc` is written in C using routines from RANLIB[10] and CVODE[11] both available from the Netlib repository. In the PC distribution, the files `_solvemc.exe` and `solvemc.bat` are included, and must be copied to a directory which is on the *system* path. In the Unix distribution, it is necessary to compile and link all the C files in the subdirectory `unixsrc/solvemc` and to place the result in a directory on the path.

In the above we have to deal with time-varying operators such as $\mathcal{L}(t)$, $H(t)$, $C_k(t)$ and $A_k(t)$. Each of these may be considered as a sparse matrix which depends on time. Typical time-variations may include exponential series, which may be of interest when considering atoms moving at constant velocity in light fields or for situations with light fields of several colours. In other cases, we may wish to consider other time-variations, such as a Gaussian to represent the effects of time-varying light pulses or the passage of atoms at constant velocity through fields with spatially varying profiles. In the toolbox, the standard differential equation solver is designed to represent an operator $\mathcal{A}(t)$ as a *function series* which takes the form

$$\mathcal{A}(t) = \mathcal{A}_1 f_1(t) + ... + \mathcal{A}_n f_n(t)$$

where each of $\mathcal{A}_1$ through $\mathcal{A}_n$ is a sparse matrix, and f_1 through f_n are chosen from a library of predefined functions, each of which may be controlled by several parameters. If $\mathcal{A}$ is in fact a constant, only a single term in this series is needed and the function f_1 is in fact a constant. In the following examples we shall illustrate the use of the differential equation solver in a variety of problems, and introduce the notation for forming function series as they become necessary.

18. Integration of a master equation with a constant Liouvillian

The solution of ordinary differential equations is handled using the toolbox functions `ode2file` and `odesolve`. The former writes out a data file which contains the definition of the problem and the latter is used to spawn the C routine `solvemc` which reads in the data file, carries out the numerical integration and writes the output file. For a master equation, the output file will consist of the density matrix evaluated at a list of times. The toolbox function `qoread` is then used to read in the data as a quantum array object.

A portion of the function `probintmaster` is shown below in order to illustrate how these routines are called in a typical problem.

```
% Calculate the Liouvillian L
% Initial state
psi0 = tensor(basis(N,1),basis(2,2));
rho0 = psi0 * psi0';
% Set up options, if required
options.lmm = 'ADAMS';
options.iter = 'FUNCTIONAL';
options.reltol = 1e-6;
options.abstol = 1e-6;
% Write out the data file
ode2file('file1.dat',L,rho0,tlist,options);
% Call the equation solver
odesolve('file1.dat','file2.dat');
% Read in the output data file
fid = fopen('file2.dat','rb');
rho = qoread(fid,dims(rho0),size(tlist));
fclose(fid);
```

The problem is set up in the usual way and `ode2file` is called. The first argument to `ode2file` is the name of the data file which is to be generated, here called `file1.dat`. The second argument is the operator or super-operator on the right-hand side of the system of differential equations, which in this case is just the Liouvillian `L`. The third argument is the initial condition, which in this case is the density matrix `rho0` formed by taking the outer product of the initial state `psi0`. The fourth argument `tlist` is a vector of times at which the solution is required.

The fifth argument to `ode2file` is optional and is used to specify additional options to the differential equation integration routines. If present, it is a structure containing one or more of the following fields:

- `lmm`: A string specifying the linear multistep method used. It may be `'ADAMS'` for the Adams-Bashford method or `'BDF'` for the backwards difference formula. The Adams-Bashford method (which is the default) is preferable for smooth problems, while the backwards difference formula method is preferable for stiff problems.

- `iter`: A string specifying how the implicit problem is solved within each step. It can be `'FUNCTIONAL'` for functional iteration (the default), or `'NEWTON'` for a Newton method with Jacobi preconditioning. The Newton method is preferable for stiff problems.
- `reltol`: A number specifying the relative tolerance, which defaults to 10^{-6}.
- `abstol`: Either a number or a vector specifying the absolute tolerance, which also defaults to 10^{-6}.
- `maxord`: Maximum linear multistep model order to be used by solver. Default is 12 for `ADAMS` and 5 for `BDF.`
- `mxstep`: Maximum number of internal steps to be taken by the solver in its attempt to reach the next timestep (default 500).
- `mxhnil`: Maximum number of warning messages issued by solver that $t+h=t$ on the next internal step (default 10).
- `h0`: initial step size.
- `hmax`: maximum absolute value of step size allowed.
- `hmin`: minimum absolute value of step size allowed.

Following the call to `ode2file`, the routine `odesolve` is called in order to carry out the integration. The two arguments to this routine are the names of the input file (which should be the same as that passed to `ode2file`) and the name of the output file, which will be overwritten if it already exisits. Matlab will spawn the external C programme to carry out the integration and waits until the programme terminates with either a success or failure. This is indicated by the external programme producing one of the files `success.qo` or `failure.qo` in the working directory. If the paths have not been set up correctly, the external program will not start and neither of these files will be produced. In this case, the Matlab programme will hang until a break command (usually Control-C) is entered.

While the external routine is carrying out the integration, a single-line progress bar indicates how far the calculation has proceeded. This may either appear in a separate window or in the Matlab command window, depending upon the operating system. On successful completion, the output file, in this case `file2.dat`, will contain the results of the integration. The `fopen` statement opens the output file and the file descriptor `fid` is passed to the function `qoread` to allow the data to be read into a Matlab quantum array object. A file descriptor rather than the file name is used since it is often necessary to make multiple calls to `qoread` to read in the data in sections, rather than all at once. The second argument to `qoread` specifies the Hilbert space dimensions of the quantum array object, which should be the same as those of the initial condition and the third argument specifies the number of records that are to be read. In the example, `size(tlist)` is used, which specifies that all the data are to be read. The size of the resulting quantum array object is given by this argument. Alternatively, a smaller number may be used to read in fewer records at a time.

Running the file `xprobintmaster` should give the same result as `xprobevolve` previously solved using exponential series. However, if the value of `N` is increased, the time required to complete `xprobintmaster` rises more slowly than for `xprobevolve.` Note that it is important to close the data files after use. If an error occurs, files may be left open which may cause the C routine to fail. If this happens, the Matlab command `fclose all` is useful for closing all files opened by Matlab.

19. Integration of a master equation with a time-varying Liouvillian

As an example of a time-dependent master equation, let us consider dragging a two-level atom through a standing light field at constant velocity, as considered above. Instead of calculating the cycle-averaged force in the steady-state, let us consider finding the time-dependnt force when the atom is initially prepared in some known state. The Liouvillian has the form

$$\mathcal{L} = \mathcal{L}_0 + \mathcal{L}_1 \exp(i\omega t) + \mathcal{L}_{-1} \exp(-i\omega t),$$

which is the sum of three terms. Using the code as for the problem of the force on a moving two-level atom, we may find the Liouvillian as an exponential series, and then extract the sparse matrices $\mathcal{L}_0$, $\mathcal{L}_1$ and $\mathcal{L}_{-1}$ corresponding to the rates 0, $i\omega$ and $-i\omega$. These amplitudes are then used to form a function series which appear on the right-hand side of the system of equations.

The following extract from `probtimedep.m` illustrates the formation of the function series

```
% Find Liouvillian L as an exponential series
% Atom initially is in ground state
psi0 = basis(2,2); rho0 = psi0 * psi0';
% Extract amplitudes of various terms in the exponential series
L0 = L(0).ampl;
Lp = L(i*kL*v).ampl;
Lm = L(-i*kL*v).ampl;
% Make the function series for the right hand side
rhs = L0 + Lp*fn('cexp',i*kL*v) + Lm*fn('cexp',-i*kL*v);
ode2file('file1.dat',rhs,rho0,tlist);
% Call the equation solver
odesolve('file1.dat','file2.dat');
% Read in the output data file
fid = fopen('file2.dat','rb');
rho = qoread(fid,dims(rho0),size(tlist));
fclose(fid);
% Calculate expectation values
Flist = esval(F,tlist);
force = expect(Flist,rho);
```

In this example the amplitudes `L0`, `Lp` and `Lm` are associated with rates `0`, `i*kL*v` and `-i*kL*v` respectively, so the subscripts in parentheses are used to extract the appropriate terms from the exponential series. These amplitudes are used to form the function series for the Liouvillian. Only a few functional dependences have been coded up so far, but they should cover many of the situations which arise in practice. Adding extra functions is relatively straightforward, although modifications need to be made both to the C routines and the Matlab routines in the `@fseries` directory.

The toolbox routine `fn` is used to generate a scalar-valued function for incorporation into a function series. The available functions are shown in the table

Syntax	Functional form
`fn('gauss',mu,sigma)`	$\exp\left[-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2\right]$
`fn('gauss',mu,sigma,omega)`	$\exp\left[-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2 - i\omega t\right]$
`fn('cexp',s)`	$\exp(st)$
`fn('pulse',s,t1,tr,t2,tf)`	$A(t)\exp(st)$

where for the pulse, the amplitude $A(t) = A_1(t)A_2(t)$ specifies a pulse from t_1 to t_2 with rise time t_r and fall time t_f. More precisely,

$$A_1(t) = \begin{cases} 0 & \text{for } t < t_1 - \frac{1}{2}t_r \\ \frac{1}{2}\left\{1 + \sin\left[\pi(t-t_1)/t_r\right]\right\} & \text{for } |t-t_1| \le \frac{1}{2}t_r \\ 1 & \text{for } t > t_1 + \frac{1}{2}t_r \end{cases}$$

and

$$A_2(t) = \begin{cases} 1 & \text{for } t < t_2 - \frac{1}{2}t_f \\ \frac{1}{2}\left\{1 - \sin\left[\pi(t-t_2)/t_r\right]\right\} & \text{for } |t-t_2| \le \frac{1}{2}t_f \\ 0 & \text{for } t > t_2 + \frac{1}{2}t_f \end{cases}.$$

In the table, the quantity `s` may be complex, but all the other numeric parameters to `fn` must be real. The functions generated by `fn` may be combined into a function series by premultiplying by quantum objects and by addition and subtraction. Thus in the example, the line

```
rhs = L0 + Lp*fn('cexp',i*kL*v) + Lm*fn('cexp',-i*kL*v);
```

creates the function series specifying the time-dependent Liouvillian. The second argument to `ode2file` specifying the operator(s) on the right-hand side of the differential equation can be a function series.

For checking whether a function series is correct, it is often useful to be able to evaluate one at a list of times so that the result can be plotted. The toolbox function `fsval` is useful for this purpose. The following shows how a function consisting of the sum of two gaussians may be plotted.

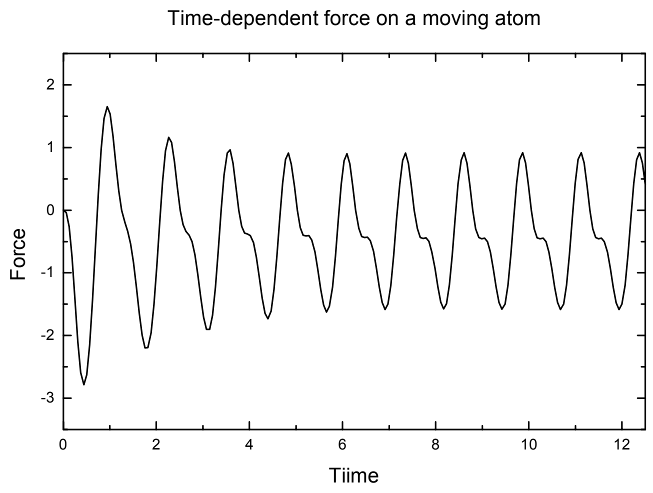

Figure 10: Time-dependent force on an atom dragged at constant velocity through a standing wave light field.

```
tlist = linspace(-5,5,201);
fs = fn('gauss',-2,1)+fn('gauss',3,0.5);
plot(tlist,fsval(fs,tlist));
```

After the data is read back from the equation solver, the force is calculated. A slightly unusual aspect is that the force operator is also time-depenedent, making it necessary to use `esval` in order to evaluate it at the times in `tlist` before the expectation value may be computed. In the last line, both `Flist` and `rho` are arrays of quantum array objects with the same number of elements, and so the force is evaluated element-by-element at each of the times in `tlist`.

Running the file `xprobtimedep` shows how the force on the atom gradually becomes periodic after the initial transient. The cycle average can be found and the value compares favourably with that found for the earlier problem. The result is shown in Figure 10.

Quantum Monte Carlo simulation

20. Individual trajectories

As an example of a quantum Monte Carlo simulation in which we write out the results of individual trajectories so that the evolution of the wave function may be considered in detail, let us consider a single cavity mode in a two-sided cavity with different mirror reflectivities. The code which performs this simulation is given in `xprobqmc1.m`.

```
Da = 0; Na = 10; Ka = 0.1;
a = destroy(Na); H0 = Da*a'*a;
C1 = sqrt(3*Ka/2)*a; C2 = sqrt(1*Ka/2)*a;
C1dC1 = C1'*C1; C2dC2 = C2'*C2;
Heff = H0 - 0.5*i*(C1dC1 + C2dC2);
psi0 = basis(10,10);
% Quantum Monte Carlo simulation
dt = 0.1;
tlist = (0:100)*dt;
ntraj = 100;
mc2file('test.dat',-i*Heff,{C1,C2},{},psi0,tlist,ntraj);
mcsolve('test.dat','out.dat','clix.dat');
% Read in wave functions, trajectory by trajectory
fid = fopen('out.dat','rb');
```

```
photnum = zeros(1,length(tlist));
for l = 1:ntraj
   if gettraj(fid) ~= l, error('Unexpected data in file'); end
   psi = qoread(fid,dims(psi0),size(tlist));
   photnum = photnum + expect(a'*a,psi)./norm(psi).^2;
end
% Find average photon number
photnum = photnum/ntraj;
fclose(fid);
plot(tlist,photnum,tlist,(Na-1)*exp(-2*Ka*tlist));
xlabel('Time'); ylabel('Number of photons');
% Read in classical record
fid = fopen('clix.dat','rb');
clix = clread(fid);
fclose(fid);
```

In the first part of this program, the parameters are set up. There is only a single mode of the light field which is expanded in a Fock space of zero to nine photons. The Hamiltonian is

$$H = \Delta_a a^\dagger a$$

and the collapse operators corresponding to photons leaking out of the two cavity mirrors are

$$C_1 = \sqrt{\frac{3\kappa_a}{2}}a \text{ and } C_2 = \sqrt{\frac{\kappa_a}{2}}a.$$

In the quantum Monte Carlo algorithm, it is necessary to calculate the effective non-Hermitian Hamiltonian which gives the evolution between quantum jumps. This is

$$H_{\text{eff}} = H - \frac{i}{2}\sum_k C_k^\dagger C_k.$$

The initial wave function is taken to be a nine photon Fock state, represented by the column vector `psi0`.

In order to carry out the Monte Carlo simulation, the problem description needs to be written out to a file, which is carried out by the routine `mc2file`. The first argument is the name of the data file to be written and the second defines the right-hand side of the system of equations, which in this case is $-iH_{\text{eff}}$. In general, this can be a function series, but in this example, we only have a constant quantum object. The third argument is a cell array of function series describing the collapse operators for the problem. Note that this is **not** a quantum array object as defined previously, which accounts for the use of the notation `{C1,C2}` rather than `[C1,C2]`. The reason why a quantum array object is not appropriate is that in general we want to specify a collection of function series (since the collapse operators may be time-dependent) and not a collection of quantum objects. In this example, the two function series happen to be the constant operators `C1` and `C2`.

The fourth argument is an empty cell array to indicate that we want the programme to generate and write out the stochastic wave functions (cf. the next section). The fifth argument is the initial condition, which is a state vector in this case. The sixth argument is the list of times at which the solution is required and the seventh argument is the number of trajectories to compute. Although not used here, an optional eighth argument may be included to pass options to the differential equation solver. The option structure is described above in connection with the routine `ode2file`. In addition to those mentioned above, there is the field `seed` which may be used to change the seed of the random number generator. If the seed is not set, a random value is taken based on the Matlab random number generator.

The routine `mcsolve` calls the external C programme and specifies the names of the data file (`test.dat`), the output file (`out.dat`) containing the state vectors and the classical record file (`clix.dat`). Following this, the data in the file `out.dat` are read back into Matlab for further processing. The data consist of `ntraj` blocks, each block containing the trajectory number followed by the state vector evaluated at the times specified in `tlist`. The toolbox function `gettraj` reads the next integer in the file and returns its value. This is compared to the expected trajectory number, and an error is signalled if this is incorrect. The function `qoread` is then called to read in the wave function which has Hilbert space dimensions `dims(psi0)` and is a quantum array object with `size(tlist)` members.

Note that in the quantum Monte Carlo algorithm, the wave function is **not normalized**, since the norm is used to calculate when the next collapse is to occur. When calculating expectation values, the toolbox routine `expect(C1,psi)` evaluates $\langle\psi|C_1|\psi\rangle$ which is correct if ψ is normalized. Since the wave funtion is not normalized, we need to divide this (element-by-element) by the square of the norm of ψ, thus evaluating $\langle\psi|C_1|\psi\rangle \, / \, \langle\psi|\psi\rangle$. Figure 11 shows the result of averaging over the five trajectories together with the analytic solution.

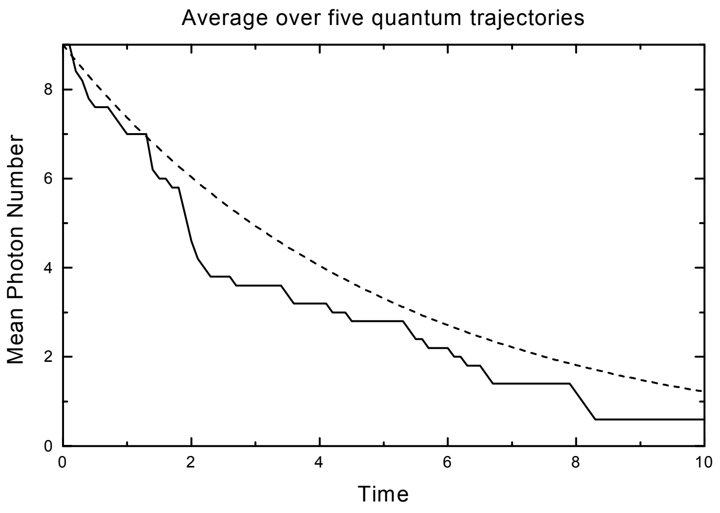

Figure 11: Mean photon number in cavity averaged over five quantum trajectories

Finally the classical record in the file `clix.dat` is processed. The routine `clread` is used to read in the data and places them in a structure array named `clix` with two fields `times` and `channels`. The list of times at which photons are detected leaving the two mirrors on trajectory `k` is given by `clix(k).times` while `clix(k).channels` contains a vector of ones and twos indicating the mirror at which each photon was detected. Note that the entire classical record must be read in at once using the routine `clread`. In the example, collapses occur more frequently due to C_1 than due to C_2 since the mirror transmissivity is higher.

21. Computing averages

Often when using the quantum Monte Carlo algorithm, we only wish to compute the expectation value of some collection of operators over a number of trajectories. The state vectors associated with the individual trajectories are not required and sometimes would require too much disk space to store. The toolbox provides a means by which the user can specify a list of operators for which these expectation values are required. As an example, let us redo the problem of the two-level atom in the driven cavity, this time using the Monte Carlo algorithm. Recall that the Hamiltonian is

$$H = (\omega_0 - \omega_L)\,\sigma_+\sigma_- + (\omega_c - \omega_L)\,a^\dagger a + ig\left(a^\dagger\sigma_- - \sigma_+ a\right) + \mathcal{E}\left(a^\dagger + a\right)$$

and that there are two collapse operators

$$\begin{aligned} C_1 &= \sqrt{2\kappa}a \\ C_2 &= \sqrt{\gamma}\sigma_-. \end{aligned}$$

We shall suppose that we wish to find the expectation values of $C_1^\dagger C_1$, $C_2^\dagger C_2$ and of the intracavity field a. The code for this problem is found in the file `probqmc2.m` which is listed below

```
function [count1, count2, infield] = probqmc2(E,kappa,gamma,g,wc,w0,wl,N,tlist,ntraj)
%
% [count1, count2, infield] = probqmc2(E,kappa,gamma,g,wc,w0,wl,N,tlist,ntraj)
%  illustrates solving the master equation by quantum Monte Carlo
%  integration of the equation. This is the same problem as solved by probevolve.m.
%
ida = identity(N); idatom = identity(2);
% Define cavity field and atomic operators
a  = tensor(destroy(N),idatom);
sm = tensor(ida,sigmam);
% Hamiltonian
H = (w0-wl)*sm'*sm + (wc-wl)*a'*a + i*g*(a'*sm - sm'*a) + E*(a'+a);
% Collapse operators
C1  = sqrt(2*kappa)*a;
C2  = sqrt(gamma)*sm;
C1dC1 = C1'*C1;
C2dC2 = C2'*C2;
% Calculate Heff
Heff = H - 0.5*i*(C1dC1+C2dC2);
% Initial state
psi0 = tensor(basis(N,1),basis(2,2));
% Quantum Monte Carlo simulation
nexpect = mc2file('test.dat',-i*Heff,{C1,C2},{C1dC1,C2dC2,a},psi0,tlist,ntraj);
mcsolve('test.dat','out.dat');
fid = fopen('out.dat','rb');
[iter,count1,count2,infield] = expread(fid,nexpect,tlist);
fclose(fid);
```

Notice that the call to the function `mc2file` now has a non-empty fourth argument, which is a cell array containing the function series describing the operators whose expectation values are required. In this example, the operators happen to be independent of time, but in general, they may be function series. If the fourth argument to `mc2file` is non-empty, the external integration routine does not write out the state vector for each trajectory but rather computes the averages of the expectation values of the specified operators, and writes them out at the end of the simulation, i.e., after the completion of `ntraj` trajectories. When the output file `out.dat` is read back into Matlab, the toolbox function `expread` is used (rather than `qoread`) since we are reading expectation values and not quantum objects. In order to read these data successfully, `expread` needs to know the number of expectation values which were computed as well as the list of times at which these were found. It is therefore important that the same parameter `tlist` sent to `mc2file` be also sent to `expread`. For convenience, the number of expectation values required (three in the example, corresponding to the number of function series in the cell array passed as the fourth argument to `mc2file`) is returned by `mc2file` so that it can be sent as the second argument to `expread`. The first output argument of `expread` is always the iteration number at which the expectation values are computed followed by the various expectation values (in this case `count1`, `count2` and `infield`) requested.

As usual, the file `xprobqmc2` calls `probqmc2` with some parameters and the results of averaging over five hundred trajectories are shown in Figure. Notice that this is converging to the result shown in Figure 3.

Sometimes it is desirable to carry out averages, not over all the `ntraj` trajectories but after a smaller number of trajectories. For example, instead of finding the average over all 500 trajectories, we may find averages over 5 groups of 100 trajectories each. This can be done by passing a two element vector `[500,100]` as the seventh argument to `mc2file`. The script `xprobqmc3` and function `probqmc3` show how this may be done for the above problem.

The script `xprobtimedep1` shows how the quantum Monte Carlo algorithm may be used to solve the problem of the moving atom in the standing wave previously treated using the master equation in `xprobtimedep`. For this example, the effective Hamiltonian and the force operator are explicitly time-dependent and are expressed as function series before being sent to `mc2file`.

Quantum State Diffusion

Instead of monitoring the cavity output using direct photodetection, it is possible to carry out homodyne

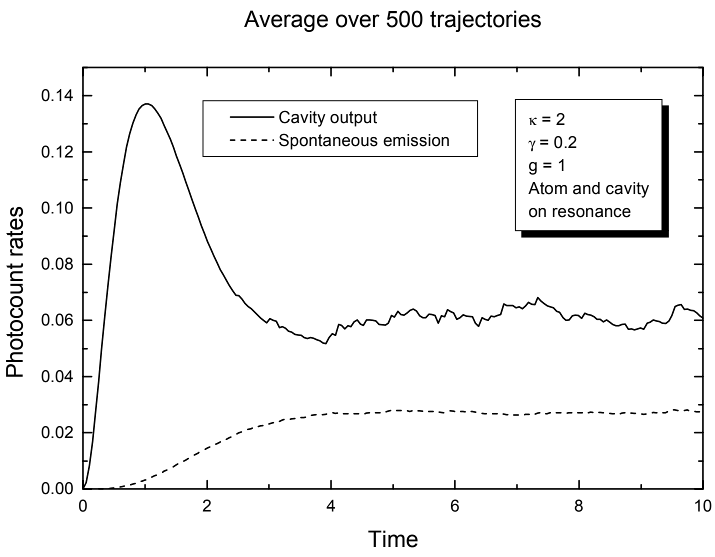

Figure 12: Monte Carlo simulation, average over 500 trajectories

and/or heterodyne detection. Two numerical algorithms are included to solve these problems, one based on the first-order Euler algorithm and the other based on converting the Itô form of the stochastic differential equation into Stratonovich form and integrating the resulting equation using the CVODE differential equation package. Simulations using these algorithms are often very much slower than for the quantum Monte Carlo algorithm, and care is necessary to use a sufficiently small timestep for stability and the required accuracy, especially with the first-order Euler equation.

Suppose that C_1 is the (only) collapse operator appearing in the Linblad form of the master equation. If the light associated with this output mode is measured with a homodyne detector when the (unnormalized) conditional state vector is $|\psi\rangle$, the stochastic Schrödinger equation for $|\psi\rangle$ is given by

$$d|\psi\rangle = -iH_{\text{eff}}\ |\psi\rangle\ dt + C_1|\psi\rangle\ dQ$$
$$dQ = \frac{\left\langle \psi|C_1 + C_1^\dagger|\psi\right\rangle}{\langle\psi|\psi\rangle}\,dt + dW$$

where $\dot{Q}$ represents the unfiltered homodyne photocurrent difference, $H_{\text{eff}} = H - \frac{i}{2}C_1^\dagger C_1$ and dW is a Wiener increment. The phase of the local oscillator in the homodyne detector may be set by considering $C_1 \exp(i\phi)$ as the collapse operator instead of C_1. In these units, we see that

$$\left\langle \frac{dQ}{dt} \right\rangle = \frac{\left\langle \psi|C_1 + C_1^\dagger|\psi\right\rangle}{\langle\psi|\psi\rangle}.$$

If on the other hand, we carry out heterodyne detection of this light field, this is equivalent to rotating the phase of the local oscillator rapidly and collecting a complex quantity Z by synchronous demodulation. The stochastic Schrödinger equation in this case is given by

$$d|\psi\rangle = -iH_{\text{eff}}\ |\psi\rangle\ dt + C_1|\psi\rangle\ dZ$$
$$dZ = \frac{\left\langle \psi|C_1^\dagger|\psi\right\rangle}{\langle\psi|\psi\rangle}\,dt + \frac{(dW_1 + i\,dW_2)}{\sqrt{2}}$$

where H_{eff} is as given above and dW_1 and dW_2 are independent Wiener increments.

In the toolbox, the C programs `stochsim` and `solvesde` are used instead of `solvemc` to carry out stochastic simulations in which homodyne, and heterodyne may be present. As before, the basic

Schrodinger equation is

$$i\frac{d\psi}{dt} = H_{\text{eff}}\left(t\right)\psi$$

where H_{eff} is a function series. The collapse operators $C_i\left(t\right)$ are now divided into two classes, the first are associated with homodyne measurements and the second with heterodyne measurements. There can be as many collapse operators as desired in each class, and each of these collapse operators is (in general) a function series. When a "classical record" is required in this case, this is a record of the photocurrents for all of the homodyne and heterodyne detectors.

The file `xprobdiffuse1` illustrates the problem of the two-sided leaky cavity, where homodyne detection takes place out of one mirror and heterodyne detection occurs out of the other. The setting up of the problem is identical with that discussed above for `xprobqmc1` until the line which calls `mc2file`. In order to carry out a state diffusion simulation, the routine `sde2file` is called instead as illustrated in the listing below

```
%
[nexpect,nphoto] = sde2file('test.dat',-i*Heff,{C1},{C2},{NN},psi0,tlist,ntraj);
sdesolve('test.dat','out.dat','photo.dat');
fid = fopen('out.dat','rb');
[iter,photnum] = expread(fid,nexpect,tlist);
fclose(fid);
f1 = figure(1);
plot(tlist,real(photnum),tlist,(Na-2)*exp(-2*Ka*tlist));
xlabel('Time'); ylabel('Number of photons in cavity');
fid = fopen('photo.dat','rb');
for k = 1:ntraj
   % Read homodyne and heterodyne records for successive trajectories
   [iter,homo,hetero] = phread(fid,nphoto,tlist);
end
```

The arguments for `sde2file` are very similar to those for `mc2file`. The first two arguments are the name of the data file and the function series for the right-hand side of the differential equation (i.e. $-iH_{\text{eff}}$). The third argument is a cell array of function series giving the operators for homodyne detection while the fourth is a cell array of function series giving the operators for heterodyne detection. Note that in this example there is only one operator in each cell array, but the braces are still required. The fifth argument is a cell array of function series specifying the operators whose expectation values are required, the sixth is the initial condition, the seventh is the list of times at which the solution is required, and the eighth is the number of trajectories or a vector with the total number of trajectories and the frequency at which averages are to be computed. An optional ninth parameter specifies options to the differential equation solver. The option structure is described above in connection with the routine `ode2file`. For stochastic differential equations, the option `hmax` is ignored. Note that the list of times must be evenly spaced and start at zero for this algorithm to work correctly.

The output arguments from `sde2file` are the number of expectation values `nexpect` and a two element vector `nphoto` which contains the number of homodyne operators and the number of heterodyne operators. After writing the data file, the routine `eulersolve` or `sdesolve` is used to call the external program which integrates the equations, using either the first-order Euler method or the CVODE library. The arguments are the data file name, the name of the output file for either the stochastic state vectors or the expectation values, and the name of the file to contain the photocurrent record. The routine `expread` is used in the usual way to read in the expectation values while `phread` is used to read in the photocurrent records for all the homodyne and heterodyne detectors. Each call to `phread` returns the record for one trajectory. The homodyne records are real while the heterodyne records are complex.

As another example, the files `xprobdiffuse2` and `probdiffuse2` illustrate the solution of the problem considered in `probqmc2` using state diffusion rather than a quantum jump simulation. The results after averaging 100 trajectories are similar to those using the jump simulations.

Quantum-state mapping between atoms and fields

As a final set of examples, the problem considered by Parkins, Marte, Zoller, Carnal and Kimble [12] is examined using the toolbox. The problem involves the passage of a multilevel atom through two light fields with different polarizations, one being a classical laser field and the other being the field within a high Q cavity. Each of these light fields has a Gaussian intensity profile, which turns into a time-varying coupling as the atom transits through the fields. By starting with the atom in a particular initial state, one can generate interesting states of the cavity field after the passage. Since the atom is assumed to have sixteen internal levels and the cavity has up to nine photons, the state space is rather large, and a numerical solution of the equations of motion is needed.

The file `xadiab1` solves for the coherent dynamics in the absence of spontaneous emission and cavity losses. This involves solving the ordinary time-dependent Schrödinger equation with a time-dependent Hamiltonian. The file `xadiab2` solves the problem using direct integration of the master equation in the situation where losses are important. Due to the adiabatic nature of the process, atomic spontaneous emission is relatively unimportant, but cavity decay affects the ability to leave the cavity in a number state at the end of the passage. In `xadiab3`, the quantum Monte Carlo algorithm is applied while in `xadiab4`, the state diffusion algorithm is used. The reader is encouraged to examine these files in conjunction with the original paper and to run them to obtain the results. Typically, a master equation problem requires a few minutes of computer time for integration on a 266 MHz Pentium II processor.

Quantum Computation Functions

Preliminary support for simulating a quantum computer is provided in the toolbox. A quantum register is a collection of m 2 state quantum systems, so the ket space is $\mathbb{C}^{2^m}$. In the toolbox, the two states are labelled 'u' and 'd' for up and down, usually corresponding to the bit values 1 and 0 respectively. In order to generate kets in this space, the function `qstate` has been written. It takes a string of m characters consisting of the letters 'u' and 'd' only (no spaces) and returns the ket in the appropriate Hilbert space. Thus for example, we may enter

```
>> qstate('u')
ans = Quantum object
Hilbert space dimensions [ 2 ] by [ 1 ]
     0
     1
>> qstate('ud')
ans = Quantum object
Hilbert space dimensions [ 2 2 ] by [ 1 1 ]
     0
     0
     1
     0
```

It is straightforward to take linear combinations of these kets. For instance, one of the Bell states, may be formed as:

```
>> uu = qstate('uu'); dd = qstate('dd'); bell=(uu+dd)/sqrt(2);
```

It is often more convenient to display a state in terms of strings rather than as a list of 2^m complex numbers. The function `qdisp` is available for this purpose. Having defined the Bell state above, we can display it using

```
>> qdisp(bell)
uu  0.70711
dd  0.70711
```

The output of `qdisp` consists of a line for each register content with non-zero amplitude, followed by the (complex) amplitude of that vector. If desired, the outputs of `qdisp` may be stored, rather than displayed. This is illustrated by

```
>>  [s,a]=qdisp(bell)
s =
uu
dd
a =
    0.7071
    0.7071
```

We next consider defining operators for quantum gates. A number of gates have been pre-defined, including the square root of not, the controlled not, Fredkin and Toffoli gates. As an example, the code which generates the Controlled-Not gate is shown

```
function Q = cnot
% CNOT computes the operator for a controlled-NOT gate
uu = qstate('uu'); ud = qstate('ud'); du = qstate('du'); dd = qstate('dd');
Q = dd*dd' + du*du' + uu*ud' + ud*uu';
```

The result is a 4×4 unitary matrix operating on the space of two quantum bits. The first bit is the control, while the second bit is the target. Thus we find

```
>> cnot
ans = Quantum object
Hilbert space dimensions [ 2 2 ] by [ 2 2 ]
     0     1     0     0
     1     0     0     0
     0     0     1     0
     0     0     0     1
```

We may operate on the Bell state, and display the result using

```
>> qdisp(cnot*bell)
ud  0.70711
dd  0.70711
```

Similarly, the functions `fredkin`, `snot` and `toffoli` have been defined. It is usually necessary to apply quantum gates to specific bits within a quantum register. For example, if we have a register of five bits, we may wish to apply the controlled not operation using bit 3 as the control bit and bit 1 as the target bit. The resulting operator is a matrix of size 32×32. In the toolbox, the function `qgate` forms the operator on the large space starting from a "template" operator which defines the gate. For example, using the operator `cnot`, we can form

```
>> Q = qgate(5,cnot,[3,1]);
>> qdisp(Q*qstate('uuuuu'))
duuuu  1
```

In the same way, any other template gate can be defined and connected up to the selected bits of a quantum register. Multiplying together the operators for a succession of operations will give a single matrix which performs the entire sequence, but it should be noted that the result is often (indeed, in all useful cases) not sparse, which leads to inefficient use of memory. It is thus often preferable to store the sequence of matrices, rather than their product.

Disclaimer

The software described in this document is still in its testing phase, is provided as-is, and no representation is made as to its correctness, accuracy or fitness for any purpose. Please send any comments or corrections to Sze Tan, Physics Department, University of Auckland, Private Bag 92019, New Zealand. My email address is `s.tan@auckland.ac.nz`.

References

[1] P. Meystre and M. Sargent III, *Elements of Quantum Optics*, Springer-Verlag, Berlin, 1990.

[2] C. Gardiner, A. Parkins, and P. Zoller, Physical Review A **46**, 4363 (1992).

[3] H. Carmichael, *An Open Systems Approach to Quantum Optics*, Springer-Verlag, Berlin, 1993.

[4] *http://www.mathworks.com*, The Mathworks, Mass.

[5] S. Swain, Journal of Physics A **14**, 2577 (1981).

[6] J. Gordon and A. Ashkin, Physical Review A **21**, 1606 (1980).

[7] H. Risken, *The Fokker-Planck Equation*, Springer-Verlag, Berlin, 1989.

[8] V. Minogin and T. Serimaa, Optics Communications **30**, 373 (1979).

[9] J. Dalibard and C. Cohen-Tannoudji, Journal of the Optical Society of America B **6**, 2023 (1989).

[10] B. W. Brown and J. Lovato, *RANLIB Library of Routines for Random Number Generation*, Department of Biomathematics, The University of Texas, Houston, 1994.

[11] S. D. Cohen and A. C. Hindmarsh, *CVODE User Guide*, Lawrence Livermore National Laboratory, 1994.

[12] A. Parkins, P. Marte, P. Zoller, O. Carnal, and H. Kimble, Physical Review A **51**, 1578 (1995).