A C++ library using quantum trajectories to solve quantum master equations

Rüdiger Schack [a,b,1], Todd A. Brun [b,2]

[a] *Department of Mathematics, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK*
[b] *Department of Physics, Queen Mary and Westfield College, University of London, London E1 4NS, UK*

Abstract

Quantum trajectory methods can be used for a wide range of open quantum systems to solve the master equation by unravelling the density operator evolution into individual stochastic trajectories in Hilbert space. This C++ class library offers a choice of integration algorithms for three important unravellings of the master equation. Different physical systems are modelled by different Hamiltonians and environment operators. The program achieves flexibility and user friendliness, without sacrificing execution speed, through the way it represents operators and states in Hilbert space. Primary operators, implemented in the form of simple routines acting on single degrees of freedom, can be used to build up arbitrarily complex operators in product Hilbert spaces with arbitrary numbers of components. Standard algebraic notation is used to build operators and to perform arithmetic operations on operators and states. States can be represented in a local moving basis, often leading to dramatic savings of computing resources. The state and operator classes are very general and can be used independently of the quantum trajectory algorithms. Only a rudimentary knowledge of C++ is required to use this package. The package illustrates how computational physics can profit from object-oriented programming concepts like inheritance.

Keywords: Open quantum system; Master equation; Hilbert space; Quantum trajectories; Unravelling; Stochastic simulation; Quantum computation; Quantum optics; Quantum state diffusion; Quantum jumps; Monte Carlo wavefunction
PACS: 42.50.-p

PROGRAM SUMMARY

Title of program: Quantum trajectory class library

Catalogue identifier: ADFQ

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland, and also from: http://galisteo.ma.rhbnc.ac.uk/applied/QSD.html and the authors.

Licensing provisions: none

Operating systems under which the program has been tested: UNIX (Gnu g++ Version 2.7.2), VMS (DEC C++ Version 5.2)

Programming language used: C++

Memory required to execute with typical data: 1 MByte

[1] Corresponding author; e-mail: r.schack@rhbnc.ac.uk.
[2] E-mail: brun@itp.ucsb.edu.

Nature of physical problem
Open quantum systems, i.e., systems whose interaction with the environment cannot be neglected, occur in a variety of contexts. Examples are quantum optics, atomic and molecular physics, and quantum computers. If the time evolution of the system is approximately Markovian, it can be described by a master equation of Lindblad form [1], a first-order differential equation for the density operator. Solving the master equation is the principal purpose of the program. Since the state and operator classes are very general, they can be used in any physical problem involving Hilbert spaces with several degrees of freedom.

Method of solution
By analogy with the solution of a Fokker–Planck equation by numerical simulation of the corresponding stochastic differential equation, a master equation can be solved by simulating the stochastic evolution of a vector in Hilbert space. The correspondence between master equation and stochastic equation is not unique: there are many ways to *unravel* the master equation into stochastic quantum trajectories. The program implements three such unravellings, known as the "quantum state diffusion method (QSD)" [2], the "quantum jump method" [3–5], and the "orthogonal jump method"[6]. The phenomenon of phase-space localization [7,8] is exploited numerically by representing quantum states in a local moving basis obtained by applying the coherent-state displacement operator to the usual harmonic-oscillator basis, often leading to dramatic savings of computing resources.

Unusual features of the program
It is worth emphasizing the effortless way in which operators and states in product Hilbert spaces are represented. Primary operators implemented in the form of simple routines acting on single degrees of freedom can be used to build up arbitrarily complex operators in product Hilbert spaces with arbitrary numbers of components. Building operators, performing arithmetic operations on operators and states, and applying operators to states is done using standard algebraic notation. This program structure has been made possible by systematically implementing object-oriented programming concepts such as inheritance, concepts which are not (yet) widely used in computational physics. Encapsulation of program modules makes it easy to add new basic operators, alternative unravellings of the master equation, or different integration algorithms.

Typical running time
The running time depends on the complexity of the problem, the integration time, and the number of trajectories required. A typical running time for a simple problem is a few minutes. There is no upper limit. The running time for the example program in Section 6 is about 2 minutes on a DEC alpha 200 under Digital UNIX.

References
[1] G. Lindblad, Commun. Math. Phys. 48 (1976) 119.
[2] N. Gisin and I.C. Percival, J. Phys. A 25 (1992) 5677.
[3] H.J. Carmichael, An Open Systems Approach to Quantum Optics (Springer, Berlin, 1993).
[4] J. Dalibard, Y. Castin and K. Mølmer, Phys. Rev. Lett. 68 (1992) 580.
[5] C.W. Gardiner, A.S. Parkins and P. Zoller, Phys. Rev. A 46 (1992) 4363.
[6] L. Diósi, Phys. Lett. A 114 (1986) 451.
[7] T. Steimle, G. Alber and I.C. Percival, J. Phys. A 28 (1995) L491.
[8] R. Schack, T.A. Brun and I.C. Percival, J. Phys. A 28 (1995) 5401.

LONG WRITE-UP

1. Introduction

For many quantum systems of current interest it is no longer possible to neglect the interactions with the environment. Those so-called *open quantum systems* occur in a variety of contexts including quantum optics, atomic and molecular physics, and quantum computers. Open quantum systems can often be described by a master equation [1], a first-order differential equation for the density operator, in which the internal dynamics of the system is represented by the system Hamiltonian $\widehat{H}$, which is a Hermitian Hilbert-space operator, and the interaction with the environment is represented by one or more Lindblad operators $\widehat{L}_j$ which are not necessarily Hermitian.

By analogy with the solution of a Fokker–Planck equation by numerical simulation of the corresponding stochastic differential equation (or Langevin equation), a master equation can be solved by simulating the

stochastic evolution of a vector in Hilbert space. The correspondence between master equation and stochastic equation is not unique; there are many ways to *unravel* the master equation into stochastic quantum trajectories [2–7].

The main challenge of this software project was to develop a general program flexible enough to accommodate different integration algorithms and unravellings of the master equation, as well as the vast range of possible physical systems. In particular, we wanted to make it easy to add new algorithms and unravellings, and we wanted a program capable of dealing with arbitrary Hamiltonian and Lindblad operators in Hilbert spaces with an arbitrary number of degrees of freedom. This task turned out to be ideal for the application of object-oriented programming. We chose the C++ language both because of its wide availability and because it allowed us to use standard mathematical notation for Hilbert-space operations by overloading algebraic operators like '+' and '*'.

The core of the program are the C++ classes State and Operator, which represent state vectors and operators in Hilbert space. Because of the object-oriented features of C++, it is possible to hide the implementation details of these classes completely from the classes dealing with the simulation of quantum trajectories. These implementation details need not to be known either by a user of the program who wants to choose the quantum operators defining the physical problem of interest or by a programmer who wants to add a new unravelling of the master equation to the software. A welcome side effect of this encapsulation is that the State and Operator classes can be used independently of the rest of the code. They should prove useful in many numerical schemes involving Hilbert spaces for systems with several degrees of freedom.

Many Hamiltonian and Lindblad operators can be written as sums of products of simple operators acting on a single degree of freedom. Here is an example of a Hamiltonian operator coupling a two-level atom (with raising and lowering operators $\hat{\sigma}_+$ and $\hat{\sigma}_-$) to an electromagnetic field mode (with annihilation and creation operators $\hat{a}$ and $\hat{a}^\dagger$),

$$\widehat{H} = g\left(\hat{\sigma}_+\hat{a} + \hat{\sigma}_-\hat{a}^\dagger\right) , \tag{1}$$

where the parameter g is the coupling strength. In the following code segment, the atomic and field degrees of freedom are labeled 0 and 1, respectively. The Hamiltonian is defined in terms of the predefined *primary operators* SigmaPlus and AnnihilationOperator using standard algebraic notation. The class AdaptiveStep is a *stepper* routine advancing the quantum trajectory by a single time step.

```
double g = 0.5;
SigmaPlus Sp(0);            // operates on the 1st degree of freedom
AnnihilationOperator A(1);  // operates on the 2nd degree of freedom
Operator Sm = Sp.hc();      // Hermitian conjugate
Operator Ac = A.hc();
Operator H = g*( Sp*A + Sm*Ac );   // Hamiltonian
    ...
AdaptiveStep theStepper(..., H, ...);  // ... denotes further arguments
```

The important feature illustrated by this example is that the stepper routine is passed an object of type Operator without any reference to details like the number of degrees of freedom. All the stepper needs to know is that operators can be added, multiplied, etc., and that they can be applied to state vectors.

Internally, the primary operators SigmaPlus and AnnihilationOperator are represented as simple loops acting on a single-degree-of-freedom state vector. An instance of the more general Operator class is represented by a stack that indicates which primary operators are used and the operations by which they are combined. For example, the sequence of steps executed by the program when the operator $\widehat{H}$ defined above is applied to a state $|\psi\rangle$ is summarized in the expression

$$\widehat{H}|\psi\rangle = g\left(\hat{\sigma}_+(\hat{a}|\psi\rangle) + \hat{\sigma}_-(\hat{a}^\dagger|\psi\rangle)\right) , \tag{2}$$

in which the elementary steps are applying a primary operator to a state, adding two states, and multiplying a state by a scalar.

The object-oriented concept of inheritance is used throughout the program library. Inheritance plays a particularly important role for the Operator class. Primary-operator classes like SigmaPlus and AnnihilationOperator are *derived*, via an intermediary interface class, from their common *parent class* Operator. The primary operators thus *inherit* the properties of their parent class. Like general operators, they can be added, applied to multiple-degree-of-freedom states, etc. Technical details about how inheritance is used in the Operator class and its derived classes are given in Section 4.

2. Quantum trajectories

2.1. Master equations

An open quantum system cannot be described by a Hilbert-space vector $|\psi\rangle$ evolving according to the Schrödinger equation; instead, the state must be described by a density operator $\hat{\rho}$ whose time evolution generally does not follow any simple law. Fortunately it turns out that for a large class of systems the time evolution of the density operator $\hat{\rho}$ is *Markovian* to an excellent approximation, i.e., the rate of change of $\hat{\rho}$ at time t, $d\hat{\rho}/dt$, depends only on $\hat{\rho}(t)$, not on the value of $\hat{\rho}$ at any earlier time. It has been shown that under the Markov approximation the density operator of any open quantum system obeys a *master equation* of Lindblad form [1],

$$\frac{d}{dt}\hat{\rho} = -\frac{i}{\hbar}\,[\hat{H}, \hat{\rho}] + \sum_j \left(\hat{L}_j\hat{\rho}\hat{L}_j^\dagger - \tfrac{1}{2}\hat{L}_j^\dagger\hat{L}_j\hat{\rho} - \tfrac{1}{2}\hat{\rho}\hat{L}_j^\dagger\hat{L}_j \right) , \tag{3}$$

where $\hat{H}$ is the system Hamiltonian and the $\hat{L}_j$ are the Lindblad operators representing the interaction with the environment.

In many cases, no analytical methods for the solution of the master equation are known; one has to use numerical methods. But even a numerical solution of the master equation can be very hard. If a state requires D basis vectors in Hilbert space to represent it, the corresponding density operator will require $D^2 - 1$ real numbers; this can often be too large for even the most powerful machines to handle, particularly if the system involves more than one degree of freedom.

This problem can be overcome by unravelling the density operator evolution into *quantum trajectories* [2–7]. Since quantum trajectories represent the system as a state vector rather than a density operator, they often have a numerical advantage over solving the master equation directly, even though one has to average over many quantum trajectories to recover the solution of the master equation. A single quantum trajectory can give an excellent, albeit qualitative, picture of a single experimental run.

2.2. Unravellings

The three unravellings of the master equation currently implemented are given by the following three nonlinear stochastic differential equations for a normalized state vector $|\psi\rangle$:

(i) the quantum state diffusion (QSD) equation [3],

$$|d\psi\rangle = -\frac{i}{\hbar}\,\hat{H}|\psi\rangle\, dt + \sum_j \left(\langle\hat{L}_j^\dagger\rangle_\psi\hat{L}_j - \tfrac{1}{2}\hat{L}_j^\dagger\hat{L}_j - \tfrac{1}{2}\langle\hat{L}_j^\dagger\rangle_\psi\langle\hat{L}_j\rangle_\psi \right) |\psi\rangle\, dt$$

$$+ \sum_j \left(\hat{L}_j - \langle\hat{L}_j\rangle_\psi \right) |\psi\rangle\, d\xi_j ; \tag{4}$$

(ii) the quantum jump equation [4–6],

$$|d\psi\rangle = -\frac{i}{\hbar}\widehat{H}|\psi\rangle\,dt + \sum_j \left(\tfrac{1}{2}\langle\widehat{L}_j^\dagger\widehat{L}_j\rangle_\psi - \tfrac{1}{2}\widehat{L}_j^\dagger\widehat{L}_j\right)|\psi\rangle\,dt + \sum_j \left(\frac{\widehat{L}_j}{\sqrt{\langle\widehat{L}_j^\dagger\widehat{L}_j\rangle_\psi}} - 1\right)|\psi\rangle\,dN_j\,; \qquad (5)$$

(iii) the orthogonal jump equation [2,7],

$$|d\psi\rangle = -\frac{i}{\hbar}\widehat{H}|\psi\rangle\,dt + \sum_j \left(\langle\widehat{L}_j^\dagger\rangle_\psi\widehat{L}_j - \tfrac{1}{2}\widehat{L}_j^\dagger\widehat{L}_j + \tfrac{1}{2}\langle\widehat{L}_j^\dagger\widehat{L}_j\rangle_\psi - \langle\widehat{L}_j^\dagger\rangle_\psi\langle\widehat{L}_j\rangle_\psi\right)|\psi\rangle\,dt$$

$$+ \sum_j \left(\frac{\widehat{L}_j - \langle\widehat{L}_j\rangle_\psi}{\sqrt{\langle\widehat{L}_j^\dagger\widehat{L}_j\rangle_\psi - \langle\widehat{L}_j^\dagger\rangle_\psi\langle\widehat{L}_j\rangle_\psi}} - 1\right)|\psi\rangle\,dN_j\,. \qquad (6)$$

The first sum in each of these equations represents the deterministic drift of the state vector due to the environment, and the second sum the random fluctuations. Angular brackets denote the quantum expectation $\langle\widehat{G}\rangle_\psi = \langle\psi|\widehat{G}|\psi\rangle$ of the operator $\widehat{G}$ in the state $|\psi\rangle$. The $d\xi_j$ are independent complex differential Gaussian random variables satisfying the conditions

$$\mathrm{M}\,d\xi_j = \mathrm{M}\,d\xi_i\,d\xi_j = 0\,, \quad \mathrm{M}\,d\xi_i^*\,d\xi_j = \delta_{ij}\,dt\,, \qquad (7)$$

where M denotes the ensemble mean. The dN_j are independent real discrete Poissonian random variables satisfying the conditions

$$dN_j^2 = dN_j\,, \quad dN_i\,dN_j = 0\,, \quad \mathrm{M}_{|\psi\rangle}\,dN_j = (\langle\widehat{L}_j^\dagger\widehat{L}_j\rangle_\psi - \lambda\langle\widehat{L}_j^\dagger\rangle_\psi\langle\widehat{L}_j\rangle_\psi)\,dt\,, \qquad (8)$$

where the "conditional mean" $\mathrm{M}_{|\psi\rangle}$ is defined as the mean over all trajectories for which $|\psi(t)\rangle = |\psi\rangle$, and where $\lambda = 0$ for the quantum jump equation (5) and $\lambda = 1$ for the orthogonal jump equation (6).

The density operator is given by the mean over the projectors onto the quantum states of the ensemble

$$\hat{\rho} = \mathrm{M}|\psi\rangle\langle\psi|\,. \qquad (9)$$

If the pure states of the ensemble satisfy one of the quantum trajectory equations (4), (5), or (6), then the density operator satisfies the master equation (3),

$$\mathrm{M}|\psi(t)\rangle\langle\psi(t)| = \hat{\rho}(t)\,, \qquad (10)$$

where we have assumed that initially the system is in a pure state $|\psi_0\rangle$ at time $t = 0$. From this it is clear that the expectation value of an operator $\widehat{O}$ is given by

$$\mathrm{Tr}\{\widehat{O}\hat{\rho}\} = \mathrm{M}\langle\psi|\widehat{O}|\psi\rangle\,. \qquad (11)$$

3. Program structure

Our C++ library can be divided roughly into three main parts:
(1) The State class and its associated friend functions, defined in the file State.cc. A State includes as member data the number of degrees of freedom it represents, how many basis vectors are allocated for each degree of freedom, the physical type of each degree of freedom, and (of course) the complex amplitudes of each basis vector in the total Hilbert space. The member functions include constructors for a number of common State types; arithmetic functions enabling States to be added, subtracted, multiplied by scalars, and normalized; functions relating to the efficient use of memory, so that a State

can be dynamically resized; and functions controlling the action of Operators on the State. There are also member data and functions relating to the moving basis algorithm, described below. States (and Operators) can be used like ordinary variables. In particular, when a locally defined State (or Operator) goes out of scope, all memory used by it is properly returned to the system; the user of the program need not worry about memory allocation and deallocation as this is done automatically.

(2) The Operator class, defined in Operator.cc. Operators are defined in terms of their actions on States. There is a small class of PrimaryOperators, whose actions on a single degree of freedom are given by predefined functions. More complex Operators are defined in terms of these PrimaryOperators; they can be added, multiplied, multiplied by scalars or time-dependent functions, conjugated, or raised to powers. An Operator's member data includes a number of dynamically allocated stacks which indicate which PrimaryOperators are used, and the operations by which they are combined. Arithmetic operations on Operators are then defined by operations on these stacks. The file PrimOp.cc defines the PrimaryOperator class, which acts as an interface to the various special PrimaryOperators defined in FieldOp.cc, SpinOp.cc and AtomOp.cc. The operators defined in FieldOp.cc include the annihilation and creation operators $\hat{a}$ and $\hat{a}^{\dagger}$ and the number operator $\hat{n}$, which act on a field mode. The operators defined in SpinOp.cc include the Pauli operators $\hat{\sigma}_x$, $\hat{\sigma}_y$ and $\hat{\sigma}_z$ acting on a spin degree of freedom. The file AtomOp.cc defines the class TransitionOperator which effects a transition between two specified levels of an atom.

(3) The Trajectory class and associated classes, defined in Traject.cc. These encode the numerical algorithms for solving the quantum trajectory equations and generating output, with associated integration routines, random number generators, and other utilities. Several different integration algorithms are currently included, including second- and fourth-order Runge–Kutta and Cash–Karp Runge–Kutta with adaptive time steps [8]. These algorithms are used to solve the deterministic part of the quantum trajectory equations (4), (5), and (6). The stochastic terms are solved using first-order Euler integration. The implementation of more sophisticated stochastic integration methods (see, e.g., [9]) is straightforward. Note that it is only in this part of the program that there is any reference at all to the details of quantum unravellings. The Operator and State classes are very general.

The remaining files included in the source code have a supporting role. The file Complex.cc defines the (slightly modified) Gnu complex-number class. The files RNG.cc, ACG.cc, MLCG.cc, Random.cc, Uniform.cc and Normal.cc have been taken from the Gnu g++ class library and define various random-number generators and random deviates (see Table 1 for more details). Finally, the file CmplxRan.cc defines two classes for complex random deviates. One of them is based on the Gnu random-number generators, whereas the other is primitive but permits one to compare single stochastic runs between different computing platforms.

The three main parts of the program described above are roughly equal in size, but quite different in internal structure. The State class is a single monolithic C++ class with associated functions; the Operator class is a parent class with numerous descendent classes representing the different PrimaryOperators. The numerical integration classes are independent of the details of State and Operator, and of each other. Because of the object-oriented nature of C++, these three groups need know very little about each other's internal workings. The following more detailed discussion is not exhaustive; a complete description of the code can be found in the extensively commented #include files, particularly in State.h, Operator.h, and Traject.h.

Table 1
List of files contained in the distribution QSD-1.3

Contents of the main directory QSD-1.3/

README	Installation instructions
CHANGES	What has changed since the previous version
TEST	How to test the package
EXAMPLES	Short description of the example programs
VERSION	Version number
License	Gnu general public license
Makefile	Makefile
include/	Directory containing the #include files (.h files)
src/	Directory containing the source code (.cc files)
testprog.cc	Test program
template.cc	Template program described in Section 6
simple.cc	Basic example program
moving.cc	Program illustrating the moving basis and different unravellings
spins.cc	Program for interacting spins, time-dependent Hamiltonian
sums.cc	Example program for a sum over trajectories
example_output/	Directory containing the output of all example programs

Contents of the subdirectory QSD-1.3/src/

State.cc	The State class
Operator.cc	The Operator class
PrimOp.cc	Abstract PrimaryOperator class, interface for derived classes
FieldOp.cc	Special operators acting on field modes
SpinOp.cc	Special operators acting on spin degrees of freedom
AtomOp.cc	Special operators acting on atomic degrees of freedom
Traject.cc	The Trajectory class
Complex.cc	Gnu Complex class, slightly modified
RNG.cc	Gnu random-number generator interface
ACG.cc	Gnu additive congruential random-number generator
MLCG.cc	Gnu linear congruential random-number generator
Random.cc	Gnu random-deviate interface
Uniform.cc	Gnu uniform random deviates
Normal.cc	Gnu Gaussian random deviates
CmplxRan.cc	Complex random deviates

4. The state and operator classes

4.1. One degree of freedom

4.1.1. States

We represent a state $|\psi\rangle$ with a single degree of freedom by an array of N complex amplitudes c_j in a given basis $\{|\phi_j\rangle\}$,

$$|\psi\rangle = \sum_{j=1}^{N} c_j |\phi_j\rangle .$$ (12)

The choice of basis vectors depends on the physical type of the system. For field modes, we use Fock states $|n\rangle$; for spins ($s = 1/2$), we use $\hat{\sigma}_z$ eigenstates $|\downarrow\rangle$ and $|\uparrow\rangle$; for N-level atoms, we use energy levels $|j\rangle$. Other types, e.g., molecules or higher spins, can be added easily. Of course, a true field mode has an infinite-dimensional

Hilbert space. The `State` class represents fields by a finite number of basis states, which should be taken as a truncation of the true infinite expansion.

To represent a state then requires the physical type (currently `FIELD`, `SPIN` or `ATOM`), the number of basis vectors N, and an array of N complex amplitudes. The state class contains constructors for many typical situations. For instance, the expression

```
State psi(2,SPIN);
```

defines `psi` to be the $|\downarrow\rangle$ state of a spin ($N = 2$), and

```
Complex alpha(0.2,0.3); State psi(100,alpha,FIELD);
```

defines a coherent state $|\alpha\rangle$ with $\alpha = 0.2 + 0.3i$ truncated to $N = 100$ basis states.

Arithmetic operations for `State`s are defined internally as operations on the complex amplitudes. In the following code examples, the state $|\psi_3\rangle = 0.5|\psi_1\rangle - |\psi_2\rangle$ is formed from the Fock states $|\psi_1\rangle = |0\rangle$ and $|\psi_2\rangle = |3\rangle$, added to $|\psi_1\rangle$, and then renormalized; finally, the inner product $z = \langle\psi_2|\psi_3\rangle$ is evaluated. Here $N = 10$ basis states are more than sufficient to represent all states without any truncation.

```
State psi1(10,0,FIELD);
State psi2(10,3,FIELD);
State psi3 = 0.5*psi1 - psi2;
psi1 += psi3;
psi3.normalize();
Complex z = psi2*psi3;
```

The expression `psi1+=psi3` is superior to the alternative `psi1=psi1+psi3` because it avoids the creation of temporary `State` objects, which is an important consideration in high-dimensional Hilbert spaces.

4.1.2. Operators

A general way of representing operators is as $N \times N$ complex matrices acting on vectors in N-dimensional Hilbert space. For large N, however, this can be very inefficient, as these matrices become very large, and applying them to states requires $O(N^2)$ operations. Fortunately, most of the operators of interest in quantum systems are *sparse*, consisting of sums and products of a few primary operators. For `FIELD`s, such primary operators are the annihilation and creation operators $\hat{a}$ and $\hat{a}^\dagger$ and the position and momentum operators $\widehat{X}$ and $\widehat{P}$; for `SPIN`s, the primary operators are the Pauli matrices $\hat{\sigma}_i$; for `ATOM`s, we have the transition operators $|i\rangle\langle j|$.

In the program, these primary operators are implemented as simple classes, as illustrated for the `SPIN` operator $\hat{\sigma}_+$ in the following code section.

```
class SigmaPlus: public PrimaryOperator {
public:
  SigmaPlus() : PrimaryOperator(0,SPIN) {};
  SigmaPlus(int freedom) : PrimaryOperator(freedom,SPIN) {};
  virtual void applyTo(State&,int,double);
};
void SigmaPlus::applyTo(State& v, int hc, double) {
  switch( hc ) {
  case NO_HC:
    v[1] = v[0]; v[0] = 0; break;
  case HC:
    v[0] = v[1]; v[1] = 0; break;
  }
}
```

The `SigmaPlus` class is derived from the abstract class `PrimaryOperator` which serves as an interface to the different special classes like `SigmaPlus`. Apart from the two constructors, the class contains only the method `applyTo`. The three arguments of `applyTo` are a single-degree of freedom `State`, an integer switch determining whether to apply $\hat{\sigma}_+$ or its Hermitian conjugate, and a `double` argument specifying the time for time-dependent operators, which is not used here.

The program represents *composite operators*, i.e., sums and products of primary operators, by stacks containing pointers to primary operators as illustrated in Fig. 1. Those stacks are the principal member data of the `Operator` class, which is the parent class of `PrimaryOperator` and therefore of all special classes derived from `PrimaryOperator`. For a primary operator like `SigmaPlus`, the stack consists just of the pointer to `*this`, which points to the primary operator itself. Fig. 2 shows the hierarchy of operator classes.

The example stack in box 3 in Fig. 1 is generated by the code segment

```
Operator O1 = a + b;
Operator O2 = (3 * c) * d;
Operator O3 = O1 - O2;
```

where `a`, `b`, `c`, `d` are assumed to be primary operators defined earlier in the program. The example illustrates how addition, subtraction, and multiplication of `Operator`s is implemented in terms of operations on the stack. Further operations defined for `Operator`s include Hermitian conjugation and raising to an integer power. The C++ inheritance mechanism ensures that all these operations are also defined for the derived primary-operator classes like `SigmaPlus`.

To apply an `Operator` to a `State`, the '`*`' operator can be used as in the following example, where `psi` is a `State` and `O3` is defined above:

```
State psi1 = O3 * psi;
```

Internally, this is implemented as a recursive evaluation of the stack. The order in which the primary operators are applied in the example can be inferred from the parentheses in

$$\hat{O}_3|\psi\rangle = \left(\hat{a} + \hat{b} - 3\hat{c}\hat{d}\right)|\psi\rangle = \left(\hat{a}|\psi\rangle + \hat{b}|\psi\rangle\right) - \hat{c}\left(3(\hat{d}|\psi\rangle)\right) . \tag{13}$$

The program keeps the number of operations and the number of temporary `State`s it creates to a minimum. Some care has to be exercised, however, to avoid an inefficient evaluation order. E.g., in the code segment

```
double x=1.5;
SigmaPlus Sp;
State psi1 = 2.0*x*Sp*psi;
```

the state `Sp*psi` is first multiplied by 1.5, then by 2.0, whereas in

```
double x=1.5;
SigmaPlus Sp;
State psi1 = (2.0*x)*Sp*psi;
```

there is only one multiplication by 3.0.

The creation of unnecessary temporary `State`s can be avoided by applying `Operator`s to `State`s using the '`*=`' operator as in

```
SigmaPlus Sp;
State psi(2,SPIN);
psi *= Sp;
```

When this code segment is executed, no temporary `State`s are created, in contrast to the otherwise equivalent code segment

```
SigmaPlus Sp;
State psi(2,SPIN);
```
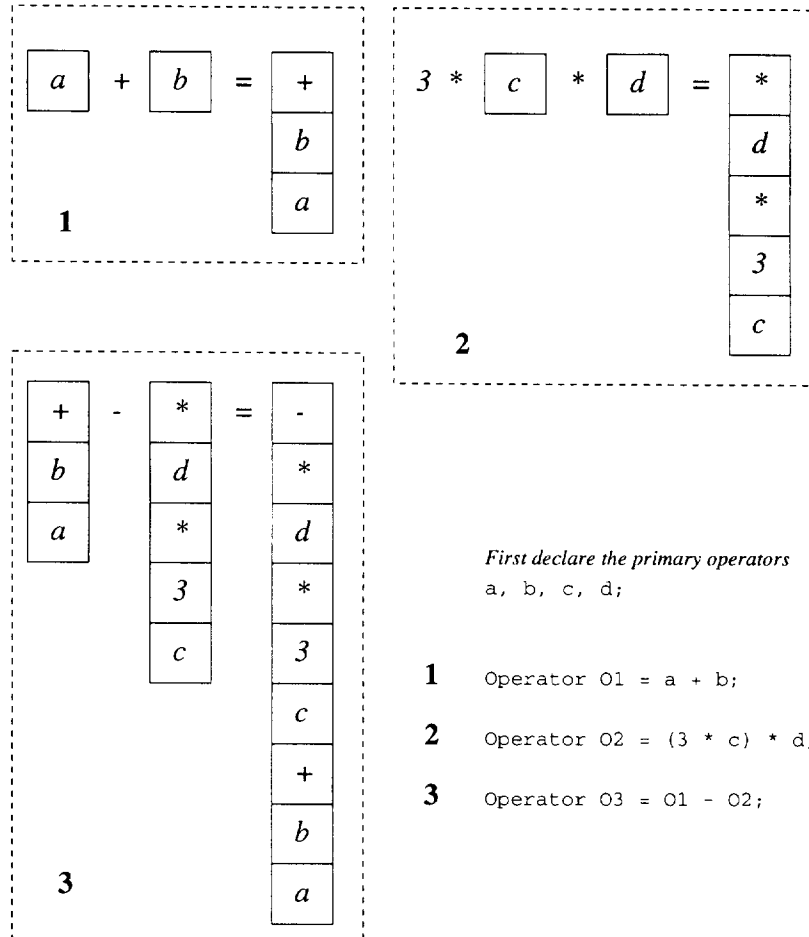
Fig. 1. These examples show how the internal stack representations of primary and composite Operators are combined in arithmetic operations. Notice that while arithmetic expressions are parsed from left to right, the order in which Operators are applied to States is from right to left.

```
psi = Sp*psi;
```

A detailed explanation of the stack and the recursive evaluation procedure can be found in the extensively commented file Operator.cc.

4.2. Multiple degrees of freedom

A quantum system with M degrees of freedom can be represented in a product Hilbert space $\mathcal{H}_1 \otimes \cdots \otimes \mathcal{H}_M$. We assume that there is a finite, perhaps truncated, product basis $\{|\phi_{n_1}\rangle \otimes \cdots \otimes |\phi_{n_M}\rangle \mid 1 \leq n_j \leq N_j\}$. Any state $|\psi\rangle \in \mathcal{H}_1 \otimes \cdots \otimes \mathcal{H}_M$ can then be written in the form

Operator.h

$$\boxed{\begin{array}{c} \textbf{Operator} \\[4pt] =,\ +,\ -,\ *,\ +=,\ -=,\ *= \\ \texttt{hc()},\ \texttt{pow()} \end{array}}$$

PrimOp.h

$$\boxed{\begin{array}{l} \textbf{PrimaryOperator} \\[4pt] \texttt{applyTo()} \end{array}}$$

FieldOp.h   SpinOp.h   AtomOp.h

$$\boxed{\begin{array}{l} \textbf{AnnihilationOperator} \\ a = (x + ip)/\sqrt{2} \\ \textbf{NumberOperator} \\ n = a^\dagger a \\ \textbf{IdentityOperator} \\ 1 \\ \textbf{XOperator} \\ x = (a + a^\dagger)/\sqrt{2} \\ \textbf{POperator} \\ p = i(a^\dagger - a)/\sqrt{2} \end{array}}$$

$$\boxed{\begin{array}{l} \textbf{SigmaX} \\ \sigma_x \\ \textbf{SigmaY} \\ \sigma_y \\ \textbf{SigmaZ} \\ \sigma_z \\ \textbf{SigmaPlus} \\ \sigma_+ \end{array}}$$

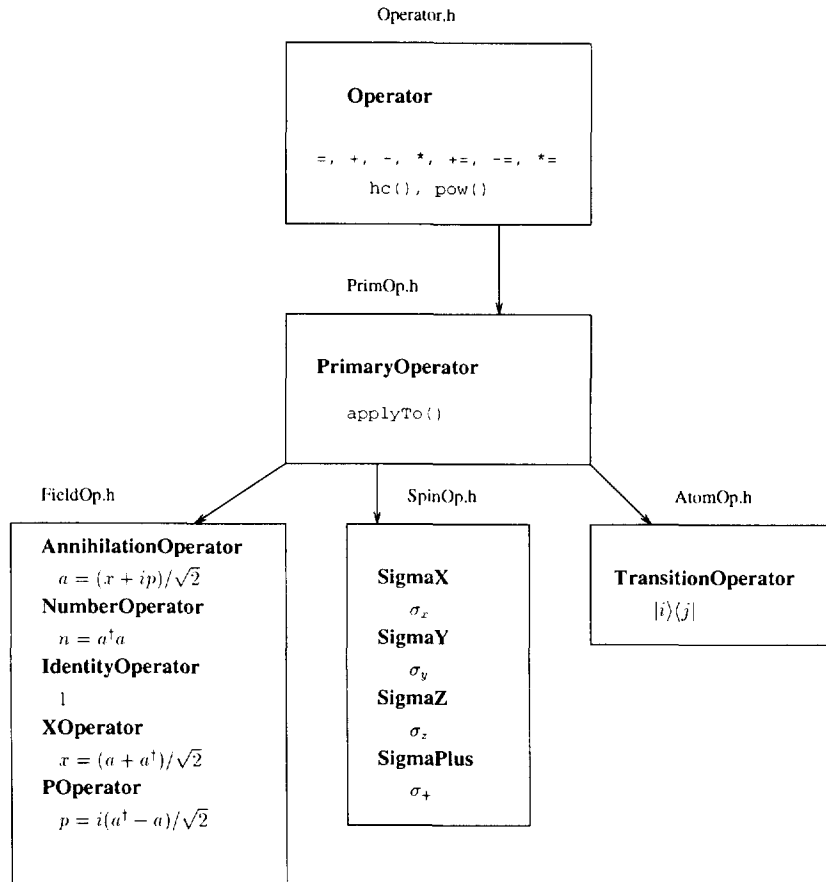$$\boxed{\begin{array}{l} \textbf{TransitionOperator} \\ |i\rangle\langle j| \end{array}}$$

Fig. 2. In this diagram, the arrows point from parent classes to derived classes. The classes listed in each box are declared in the #include file given above the box. Arithmetic operations are defined in the Operator class. The PrimaryOperator class serves as an interface for the specific FIELD, SPIN and ATOM operators. Adding operators of either an existing or a new type is straightforward.

$$|\psi\rangle = \sum_{n_1,\ldots,n_M} C_{n_1,\ldots,n_M} |\phi_{n_1}\rangle \otimes \cdots \otimes |\phi_{n_M}\rangle, \tag{14}$$

requiring a total of $N_{tot} = N_1 N_2 \cdots N_M$ basis vectors. To represent a state with multiple degrees of freedom, the State class contains as member data the number of freedoms M, an array of M physical types, an array of M subspace dimensions N_j, and an array of N_{tot} amplitudes $C_{n_1,\ldots,n_M}$.

Product states can be initialized by passing a list of single-degree-of-freedom states to the appropriate State constructor. This is illustrated in the following example, where the state $|0\rangle \otimes |0\rangle \otimes |\downarrow\rangle$ is assigned to psiIni:

```
State phi1(50,FIELD);
State phi2(50,FIELD);
State phi3(2,SPIN);
```

```
State stateList[3] = {phi1, phi2, phi3};
State psiIni(3,stateList);
```

Entangled states can be constructed by adding several product states or by explicitly initializing the array of amplitudes $C_{n_1,...,n_M}$.

Operators acting on multiple degrees of freedom are represented as sums and products of primary operators each acting on a single degree of freedom. Take the example of a primary operator $\hat{b}$ acting on the first degree of freedom. It can be rewritten as the operator $\hat{b} \otimes \hat{1}$ on the product Hilbert space, where $\hat{1}$ is the identity operator acting on all the other degrees of freedom. We can write any state $|\psi\rangle$ as

$$|\psi\rangle = \sum_{n_2,...,n_M} |\psi_{n_2,...,n_M}\rangle \otimes |\phi_{n_2}\rangle \otimes \cdots \otimes |\phi_{n_M}\rangle ; \tag{15}$$

the action of $\hat{b} \otimes \hat{1}$ on $|\psi\rangle$ is therefore given by the action of $\hat{b}$ on the first degree of freedom inside a hierarchy of loops over all the other degrees of freedom,

$$\left(\hat{b} \otimes \hat{1}\right) |\psi\rangle = \sum_{n_2,...,n_M} \left(\hat{b}|\psi_{n_2,...,n_M}\rangle\right) \otimes |\phi_{n_2}\rangle \otimes \cdots \otimes |\phi_{n_M}\rangle . \tag{16}$$

In the program, the loops are unrolled into one big loop if the primary operator acts on the first or last degree of freedom; otherwise the loops are unrolled into two loops, an "inner" and an "outer" loop.

To define, e.g., a primary SigmaPlus operator acting on the $n = $ 3rd degree of freedom, the constructor has to be called with the argument $n - 1 = 2$:

```
SigmaPlus Sp(2);
```

The Operator class is virtually unaffected by the complications arising from multiple degrees of freedom (see Fig. 3). Whenever an Operator is applied to a State psi, the recursive evaluation of the Operator stack will eventually come across a pointer to some primary operator B acting on a particular freedom. At that stage, the pointer to B will be passed to the method psi.apply() of the State class, which controls the loops over all the other degrees of freedom. Each time the loop is executed, the State class passes a single-degree-of-freedom state to the method B.applyTo() of the primary operator B. The complex amplitudes of this single-freedom state are typically stored at widely spaced locations in the array of complex amplitudes $C_{n_1,...,n_M}$, but this fact is completely hidden from the primary operator B.

This way of organizing the program has great advantages. Most importantly, all the implementation details of multiple-freedom states are hidden from the Operator class. Apart from leading to a transparent program, this makes adding new primary operators very easy, as was seen in Section 4.1.2. The definition of the primary-operator class SigmaPlus given there is used without modification in the multiple-freedom case.

Our class library realizes its full potential when all operators are sums and products of a few simple primary operators. Although this situation is extremely common in many fields, there are important exceptions like the Coulomb potential. While the program could be adapted to implement such a case, some of its unique features would be lost in the process.

For efficiency, the State class distinguishes internally between single-freedom and multiple-freedom states; many actions are more efficient for a single degree of freedom. This distinction, however, is completely transparent. The user needs to distinguish between the two only when constructing the initial state.

4.3. The moving basis

In quantum-trajectory simulations, one often encounters FIELD states that are well *localized* in phase space [3,10–14]. In cases with strong localization, it is often possible to reduce drastically the number N of basis states needed by continually changing the basis. If a state is localized about a point (q,p) in phase
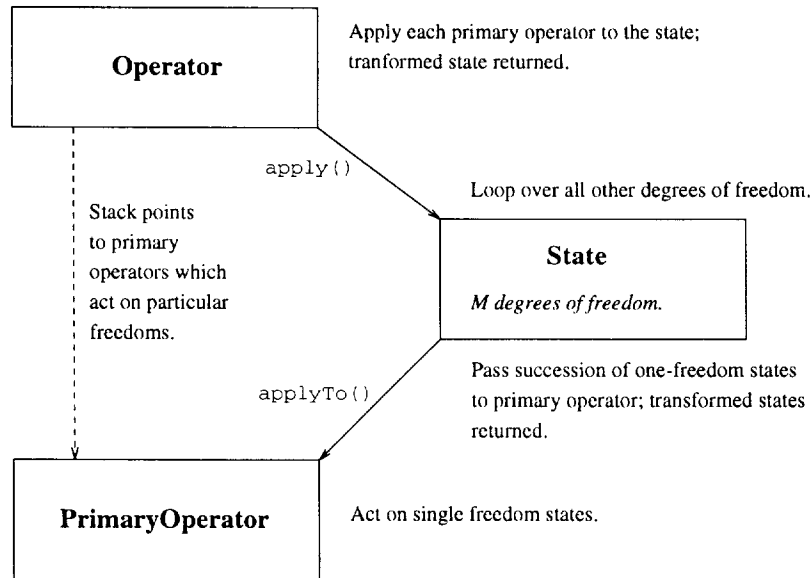
Fig. 3. When a pointer to a primary operator acting on a particular freedom is encountered during the evaluation of an Operator stack, control is passed to the State class, where within loops over the basis states of all the other degrees of freedom, the primary operator is applied to a succession of single-freedom states. This means that the Operator class and its derived classes do not need to distinguish between single and multiple freedom states; all details concerning the multiple-freedom case are hidden within the State class.

space far from the origin, it requires many number states $|n\rangle$ to represent it. But relatively few displaced number states (or *excited coherent states*) $|q,p,n\rangle = \widehat{D}(q,p)|n\rangle$ are needed, with corresponding savings in computer storage space and computation time. The operator $\widehat{D}(q,p)$ is the usual coherent state displacement operator [15],

$$\widehat{D}(q,p) = \exp\frac{i}{\hbar}\left(p\widehat{X} - q\widehat{P}\right) , \qquad (17)$$

where $\widehat{X}$ and $\widehat{P}$ are the position and momentum operators. The separation of the representation into a classical part (q,p) and a quantum part $|q,p,n\rangle$ is called the *moving basis* [16] or, as in [13], the *mixed* representation. To represent a state of type FIELD in the moving basis requires to store the complex center of coordinates $\alpha = (q + ip)/\sqrt{2}$ in addition to the complex amplitudes. A multiple-freedom state in the moving basis with several freedoms of type FIELD requires an array of centers of coordinates.

Implementing the moving basis algorithm is straightforward. Suppose that at time $t = t_0$ the state $|\psi(t_0)\rangle$ is represented in the basis $|q_0,p_0,n\rangle$, centred at

$$(q_0,p_0) = \left(\langle\psi(t_0)|\widehat{X}|\psi(t_0)\rangle, \langle\psi(t_0)|\widehat{P}|\psi(t_0)\rangle\right) . \qquad (18)$$

Then after one discrete time step, the expectations in this basis shift to

$$(q_1,p_1) = \left(\langle\psi(t_0 + \delta t)|\widehat{X}|\psi(t_0 + \delta t)\rangle, \langle\psi(t_0 + \delta t)|\widehat{P}|\psi(t_0 + \delta t)\rangle\right) \neq (q_0,p_0) . \qquad (19)$$

The computational advantage of a small number of basis states is then retained by changing the representation to the shifted basis $|q_1,p_1,n\rangle$ centred at q_1 and p_1. This shift in the origin of the basis represents the elementary single step of the moving basis.

The components of $|\psi(t_0 + \delta t)\rangle$ can be computed using the expressions given above. The computing time needed for the basis shift is of the same order of magnitude as for computing a single discrete time step of one of the quantum trajectory equations. Shifting the basis once every discrete time step could therefore double the computing time, depending on the complexity of the Hamiltonian and the number of degrees of freedom. On the other hand, the reduced number of basis vectors needed to represent states in the moving basis can lead to savings far bigger than a factor of 2.

In the example of second harmonic generation discussed in [16], two modes of the electromagnetic field interact. Using the moving basis reduces the number of basis vectors needed by a factor of 100 in each mode. The total number of basis vectors needed is thus reduced by a factor of 10000, leading to reduction in computing time by a factor of roughly $10000/2 = 5000$. Furthermore, the fixed basis would exceed the memory capacity of most computers.

The State class includes a variety of basis-changing methods. The most important is the method

```
void moveCoords( const Complex& displacement, int theFreedom,
                 double shiftAccuracy );
```

which performs a relative shift of the center of coordinates $\alpha = (q + ip)/\sqrt{2}$ by an amount given by the complex argument displacement. The integer argument theFreedom specifies which degree of freedom is to be shifted – this degree of freedom must be of type FIELD. The double argument shiftAccuracy gives the numerical accuracy with which to make the shift. The physical state is unchanged by applying moveCoords(), but it is represented in a new basis. The method moveCoords() is used in the stochastic integration algorithms of the Trajectory class described in Section 5. The primary operators of type FIELD defined in the files FieldOp.h and FieldOp.cc are implemented in such a way that they can handle moving-basis states as well as ordinary states.

The quantum trajectory equations can contain both localizing and delocalizing terms [3,10–14]. Nonlinear terms in the Hamiltonian tend to spread the wave function in phase space, whereas the Lindblad terms often cause it to localize. Accordingly, the width of the wave packets varies along a typical trajectory. We use this to reduce the computing time even further by dynamically adjusting the number of basis vectors. Our criterion for this adjustment depends on parameters $\epsilon \ll 1$, the *cutoff probability*, and N_{pad}, the *pad size*, which represents the number of boundary basis states that are checked for significant probability. We require the total probability of the top N_{pad} states to be no greater than ϵ, increasing or decreasing the number of states actually used accordingly, as the integration proceeds along the quantum trajectory. The method of the State class used to adjust the basis size is

```
void adjustCutoff(int theFreedom, double epsilon, int padSize);
```

where the arguments specify the degree of freedom to be adjusted, the cutoff probability ϵ, and the pad size N_{pad}, respectively.

Like the basis-changing methods discussed above, the method adjustCutoff() is typically only used inside integration routines of the Trajectory class. Those methods will not normally be called from a top-level program, so the user need not be concerned by them.

5. The Trajectory class

The Trajectory class and its associated classes, defined in the files Traject.h and Traject.cc, implement the integration of the quantum trajectory equations (4), (5), and (6). At the heart of this part of the code is the abstract class IntegrationStep which serves as an interface for the specific stepper classes implementing single integration steps of length dt. The stepper classes derived from the class IntegrationStep include the class Order4Step for a single fourth-order Runge–Kutta step of the QSD equation (4) as well as a group of classes using adaptive Cash–Karp Runge–Kutta time steps: the class AdaptiveStep for a time step of total

length dt of the QSD equation (4), the class AdaptiveJump for a time step of total length dt of the quantum jump equation (5), and the class AdaptiveOrthoJump for a time step of total length dt of the orthogonal jump equation (6). All those classes use a single first-order Euler integration step of length dt for the stochastic part. Due to the modular structure of the class library, it is straightforward to add more sophisticated stochastic integration methods (see, e.g., [17,9]).

To initialize a stepper, including all temporary memory needed for the integration algorithm, all one has to do is call the appropriate constructor, as in the code segment

```
State psiIni(2,SPIN);
SigmaPlus Sp;
Operator H = Sp + Sp.hc();
int nL = 1;
Operator L[nL] = {0.1*Sp.hc()}
AdaptiveStep stepper(psiIni, H, nL, L);
```

A less trivial example can be found in the sample program in Section 6. Entire quantum trajectories are computed by repeatedly calling a stepper from within the Trajectory class. A trajectory is initialized as in the following example which is taken from the sample program below:

```
double dt=0.01;      // basic time step passed to the stepper
ACG gen(38388389);   // random number generator defined in ACG.h
ComplexNormal rndm(&gen);
                     // Gaussian random numbers defined in CmplxRan.h
Trajectory traj(psiIni, dt, stepper, &rndm);
```

The Trajectory class comprises two methods to launch the simulation, compute expectation values of operators of interest, and produce output. The use of the method plotExp(), designed to simulate a single trajectory, is explained in Section 6. The method sumExp(), which is very similar to plotExp(), can be used to compute the mean expectation values of operators averaged over many trajectories.

6. Sample program and template

In this section, we illustrate the main features of the class library in a complete example program which can be used as a template. The example program computes expectation values for a single trajectory of the quantum state diffusion equation (4); to compute means over many trajectories, one simply replaces the call to traj.plotExp() in the template by a call to traj.sumExp(). The system has three degrees of freedom: two nonlinearly coupled field modes described by annihilation operators $\hat{a}_1$ and $\hat{a}_2$, and a spin described by raising and lowering operators $\hat{\sigma}_+$ and $\hat{\sigma}_-$. The Hamiltonian in the interaction picture is [18]

$$\widehat{H} = Ei(\hat{a}_1^\dagger - \hat{a}_1) + \tfrac{1}{2}\chi i(\hat{a}_1^{\dagger 2}\hat{a}_2 - \hat{a}_1^2\hat{a}_2^\dagger) + \omega\hat{\sigma}_+\hat{\sigma}_- + \eta i(\hat{a}_2\hat{\sigma}_+ - \hat{a}_2^\dagger\hat{\sigma}_-) , \qquad (20)$$

where E is the strength of an external pump field, χ is the strength of the nonlinear interaction, ω is the detuning between the frequency of the field mode $\hat{a}_2$ and the spin transition frequency, and η is the strength of the coupling of the spin to the field mode $\hat{a}_2$. The Lindblad operators

$$\widehat{L}_1 = \sqrt{2\gamma_1}\,\hat{a}_1 , \quad \widehat{L}_2 = \sqrt{2\gamma_2}\,\hat{a}_2 , \quad \widehat{L}_3 = \sqrt{2\kappa}\,\hat{\sigma}_- \qquad (21)$$

describe dissipation of the field modes and the spin with coefficients γ_1, γ_2, and κ, respectively.

The trajectory's initial state is the product state $|\psi_{\text{ini}}\rangle = |0\rangle \otimes |0\rangle \otimes |\downarrow\rangle$. The integration step-size is dt=0.01 and the total integration time is 500*dt = 5.0. The integration stepper AdaptiveStep implements a single time step of length dt of the QSD equation (4) using the Cash–Karp Runge–Kutta algorithm with adaptive time steps [8] for the deterministic part and first-order Euler integration for the stochastic part.

At times that are integer multiples of 100*dt = 1.0, the expectation values of the operators specified in the array outlist are computed and written to the files specified in the array flist. E.g., the first element of outlist is the operator $\widehat{X}_1 \equiv \hat{\sigma}_+ \hat{a}_2 \hat{\sigma}_- \hat{\sigma}_+$. At times $t = 0, 1.0, \ldots, 5.0$, the method plotExp computes the expectation values $\langle \widehat{X}_1 \rangle$ and var$(\widehat{X}_1) \equiv \langle \widehat{X}_1^2 \rangle - \langle \widehat{X}_1 \rangle^2$ and writes t, Re($\langle \widehat{X}_1 \rangle$), Im($\langle \widehat{X}_1 \rangle$), Re(var($\widehat{X}_1$)), and Im(var($\widehat{X}_1$)) to the file X1.out. In addition, each time a set of expectation values is computed, the program writes 7 numbers to standard output (see the sample output below): the time t, 4 expectation values determined by the integer array pipe, the number of basis states used, and the number of adaptive steps taken. The integers in the array pipe correspond to the columns in the output files containing expectation values (i.e., columns 2 through 5 of each output file). In the present example, expectation values are computed for the 3 operators $\hat{\sigma}_+ \hat{a}_2 \hat{\sigma}_- \hat{\sigma}_+$, $\hat{\sigma}_- \hat{\sigma}_+ \hat{a}_2 \hat{\sigma}_-$ and $\hat{a}_2$, which are written to 3 output files with numbered columns 1 through 12. According to the expression int pipe[]={1,5,9,11}, the expectation values written to standard output are Re($\langle \hat{\sigma}_+ \hat{a}_2 \hat{\sigma}_- \hat{\sigma}_+ \rangle$), Re($\langle \hat{\sigma}_- \hat{\sigma}_+ \hat{a}_2 \hat{\sigma}_- \rangle$), Re($\langle \hat{a}_2 \rangle$) and Re(var($\hat{a}_2$)).

The moving basis is used for both FIELD degrees of freedom. The basis size is dynamically adjusted with a cutoff probability $\epsilon = 0.01$ and a pad size $N_{pad} = 2$. The sample output below shows how the basis size changes with time. Initially, $5000 = 50 * 50 * 2$ states are allocated, but at time $t = 1.0$, only 60 states are needed. Subsequently, the basis size fluctuates around a typical size of 70 states.

Here is the complete program:

```
#include "Complex.h"
#include "ACG.h"
#include "CmplxRan.h"
#include "State.h"
#include "Operator.h"
#include "FieldOp.h"
#include "SpinOp.h"
#include "Traject.h"


int main()
{
// Primary Operators
  AnnihilationOperator A1(0);   // 1st freedom
  NumberOperator N1(0);
  AnnihilationOperator A2(1);   // 2nd freedom
  NumberOperator N2(1);
  SigmaPlus Sp(2);              // 3rd freedom
  Operator Sm = Sp.hc();        // Hermitian conjugate
  Operator Ac1 = A1.hc();
  Operator Ac2 = A2.hc();
// Hamiltonian
  double E = 20.0;
  double chi = 0.4;
  double omega = -0.7;
  double eta = 0.001;
  Complex I(0.0,1.0);
  Operator H = (E*I)*(Ac1-A1)
            + (0.5*chi*I)*(Ac1*Ac1*A2 - A1*A1*Ac2)
            + omega*Sp*Sm + (eta*I)*(A2*Sp-Ac2*Sm);
```

```
// Lindblad operators
  double gamma1 = 1.0;
  double gamma2 = 1.0;
  double kappa = 0.1;
  const int nL = 3;
  Operator L[nL]={sqrt(2*gamma1)*A1,sqrt(2*gamma2)*A2,sqrt(2*kappa)*Sm};
// Initial state
  State phi1(50,FIELD);        // see Section 4.2
  State phi2(50,FIELD);
  State phi3(2,SPIN);
  State stateList[3] = {phi1,phi2,phi3};
  State psiIni(3,stateList);
// Trajectory
  double dt = 0.01;     // basic time step
  int numdts = 100;     // time interval between outputs = numdts*dt
  int numsteps = 5;     // total integration time = numsteps*numdts*dt
  int nOfMovingFreedoms = 2;
  double epsilon = 0.01;       // cutoff probability
  int nPad = 2;                // pad size
  ACG gen(38388389);           // random number generator with seed
  ComplexNormal rndm(&gen);  // Complex Gaussian random numbers
  AdaptiveStep stepper(psiIni, H, nL, L);       // see Section 5
  Trajectory traj(psiIni, dt, stepper, &rndm);  // see Section 5
// Output
  const int nOfOut = 3;
  Operator outlist[nOfOut]={ Sp*A2*Sm*Sp, Sm*Sp*A2*Sm, A2 };
  char *flist[nOfOut]={"X1.out","X2.out","A2.out"};
  int pipe[] = { 1, 5, 9, 11 };    // controls standard output
// Simulate one trajectory
  traj.plotExp( nOfOut, outlist, flist, pipe, numdts, numsteps,
                nOfMovingFreedoms, epsilon, nPad );

}
```

In addition to the output files X1.out, X2.out and A2.out, the program writes the following numbers to standard output:

t	$\mathrm{Re}(\langle \hat{\sigma}_+ \hat{a}_2 \hat{\sigma}_- \hat{\sigma}_+ \rangle)$	$\mathrm{Re}(\langle \hat{\sigma}_- \hat{\sigma}_+ \hat{a}_2 \hat{\sigma}_- \rangle)$	$\mathrm{Re}(\langle \hat{a}_2 \rangle)$	$\mathrm{Re}(\mathrm{var}(\hat{a}_2))$	basis size	steps
0	0	0	0	0	5000	0
1	0.0131402	0.0131173	-5.91709	-0.247349	60	125
2	0.0425276	0.0455457	-6.4369	-0.369512	70	106
3	0.0260639	0.0626976	-5.55435	-0.162967	80	260
4	0.0796275	-0.0209383	-4.34486	0.0635076	70	266
5	-0.00616844	0.0110794	-5.38422	-0.253009	50	314

The output in the file A2.out, e.g., should be

t	$\mathrm{Re}(\langle \hat{a}_2 \rangle)$	$\mathrm{Im}(\langle \hat{a}_2 \rangle)$	$\mathrm{Re}(\mathrm{var}(\hat{a}_2))$	$\mathrm{Im}(\mathrm{var}(\hat{a}_2))$
0	0	0	0	0
1	-5.91709	0.0897429	-0.247349	0.036083
2	-6.4369	0.426733	-0.369512	0.0381381
3	-5.55435	2.88847	-0.162967	0.303524

```
4    -4.34486    -4.36856        0.0635076    -0.307045
5    -5.38422     0.530589      -0.253009      0.189403
```

This output has been obtained using Gnu g++ Version 2.7.2 under Digital Unix on a Dec Alpha 200 and corresponds to a single stochastic trajectory. The random number generator used in the template program tends to generate different sequences of random numbers on different machines. The output obtained from the template program in another environment may thus differ from the above, corresponding to a different but equally valid realisation of the stochastic process. The next section describes a test program using a primitive but platform-independent random-number generator which should lead to identical output on all computing platforms.

7. Installation and testing

The version of the software described in this article is distributed in the form of a compressed file, QSD-1.3.tar.Z. To install the software on a UNIX system, execute the commands

```
uncompress QSD-1.3.tar.Z
tar -xvf QSD-1.3.tar
```

To test that the program has been correctly installed and runs properly, execute the commands

```
cd QSD-1.3
make testprog
testprog
```

On a DEC Alpha 200 workstation, the test run takes about 2 minutes. You should get the following output on the screen, independently of the computing platform used:

```
0 0 0 0 0 5000 0
1 0.012356 0.0122021 -5.69338 -0.176628 60 125
2 0.0470506 0.0447548 -6.49036 -0.406 108 100
3 0.0556004 0.0477218 -6.30988 -0.417363 108 137
4 0.0287731 0.0509956 -6.2915 -0.332276 90 147
5 0.04361 -0.0130806 -5.73311 -0.311603 84 169
```

Furthermore, the output files X1.out, X2.out and A2.out should be created and be identical to the corresponding files in the directory QSD-1.3/example_output/testprog/.

In addition to the test program, testprog.cc, and the template program described in the last section, template.cc, the distribution contains four example programs illustrating various features of the package. Their output is contained in the subdirectories of the directory QSD-1.3/example_output/. To compile and run, e.g., the program spins.cc, type

```
make spins
spins
```

The program simple.cc does a very basic simulation of a single trajectory for a simple harmonic oscillator. The program moving.cc also simulates one trajectory for a simple harmonic oscillator, but uses the moving basis. In moving.cc, different unravellings can be chosen by uncommenting or commenting out the appropriate lines. The program spins.cc simulates a single trajectory for two interacting spins with a time-dependent Hamiltonian. Finally, sums.cc sums over many trajectories for a simple harmonic oscillator.

Acknowledgements

References

[1] G. Lindblad, Commun. Math. Phys. 48 (1976) 119.
[2] L. Diósi, Phys. Lett. A 114 (1986) 451.
[3] N. Gisin and I.C. Percival, J. Phys. A 25 (1992) 5677.
[4] H.J. Carmichael, An Open Systems Approach to Quantum Optics (Springer, Berlin, 1993).
[5] J. Dalibard, Y. Castin and K. Mølmer, Phys. Rev. Lett. 68 (1992) 580.
[6] C.W. Gardiner, A.S. Parkins and P. Zoller, Phys. Rev. A 46 (1992) 4363.
[7] J.K. Breslin, G.J. Milburn and H.M. Wiseman, Phys. Rev. Lett. 74 (1995) 4827.
[8] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, Numerical Recipes in C, 2nd ed. (Cambridge Univ. Press, Cambridge, 1992).
[9] J. Steinbach, B.M. Garraway and P.L. Knight, Phys. Rev. A 51 (1995) 3302.
[10] L. Diósi, Phys. Lett. A 132 (1988) 233.
[11] N. Gisin and I.C. Percival, J. Phys. A 26 (1993) 2245.
[12] I.C. Percival, J. Phys. A 27 (1994) 1003.
[13] T. Steimle, G. Alber and I.C. Percival, J. Phys. A 28 (1995) L491.
[14] M. Holland, S. Marksteiner, P. Marte and P. Zoller, Phys. Rev. Lett. 76 (1996) 3683.
[15] W.H. Louisell, Quantum Statistical Properties of Radiation (Wiley, New York, 1973).
[16] R. Schack, T.A. Brun and I.C. Percival, J. Phys. A 28 (1995) 5401.
[17] P.E. Kloeden and E. Platen, Numerical Solution of Stochastic Differential Equations (Springer, Berlin, 1992).
[18] R. Schack, T.A. Brun and I.C. Percival, Phys. Rev. A 53 (1996) 2694.