QuTiP: A framework for the dynamics of open quantum systems using SciPy and Cython

Robert J. Johansson‡*, Paul D. Nation‡

Abstract—We present QuTiP (http://www.qutip.org), an object-oriented, open-source framework for solving the dynamics of open quantum systems. Written in Python, and using a combination of Cython, NumPy, SciPy, and matplotlib, QuTiP provides an environment for computational quantum mechanics that is both easy and efficient to use. Arbitrary quantum systems, including time-dependent systems, may be built up from operators and states defined by a quantum object class, and then passed on to a choice of unitary or dissipative evolution solvers. Here we give an overview of the basic structure for the framework, and the techniques used in its implementation. We also present a few selected examples from current research on quantum mechanics that illustrate the strengths of the framework, as well as the types of calculations that can be performed. This framework is particularly well suited to the fields of quantum optics, superconducting circuit devices, nanomechanics, and trapped ions, while also being ideal as an educational tool.

Index Terms—quantum mechanics, master equation, monte-carlo

Introduction

One of the main goals of contemporary physics is to control the dynamics of *individual* quantum systems. Starting with trapped-ion experiments in the 1970s [Hor97], the ability to manipulate single realizations, as opposed to ensembles, of quantum systems allows for fundamental tests of quantum mechanics [Har06] and quantum computation [Lad10]. Traditionally, the realm of quantum mechanics has been confined to systems with atomic and molecular characteristic length scales with system properties fixed by nature. However, during the last two decades, advances in experimental and manufacturing techniques have opened up the possibility of producing man-made micro and nanometer scale devices with controllable parameters that are governed by the laws of quantum mechanics. These engineered systems can now be realized with a wide range of different technologies such as quantum optics [Obr09], superconducting circuits [You11], semiconducting quantum dots [Han08], nanomechanical devices [Oco10], and ion traps [Bla12], and have received considerable experimental and theoretical attention.

With the increasing interest in engineered quantum devices, the demand for efficient numerical simulations of open quantum dynamics has never been greater. By definition, an open quantum system is coupled to an environment, also called a reservoir or bath, where the complexity of the environmental dynamics renders the combined evolution of system plus reservoir analytically intractable and must therefore be simulated numerically. With a quantum computer out of reach for the foreseeable future, these simulations must be performed using classical computing techniques, where the exponentially increasing dimensionality of the underlying Hilbert space severely limits the size of system that can be efficiently simulated [Fey82]. Fortunately, recent experimental advances have lead to quantum systems fabricated from a small number of oscillator and spin components, containing only a few quanta, that lend themselves to simulation in a truncated state space.

Here we introduce QuTiP [Joh12], a framework designed for simulating open quantum systems using SciPy and Cython. Although other software solutions exist [Sch97], [Vuk07], [Tan99], QuTiP goes beyond these earlier tools by providing a completely open-source solution with an easy to read syntax and extended

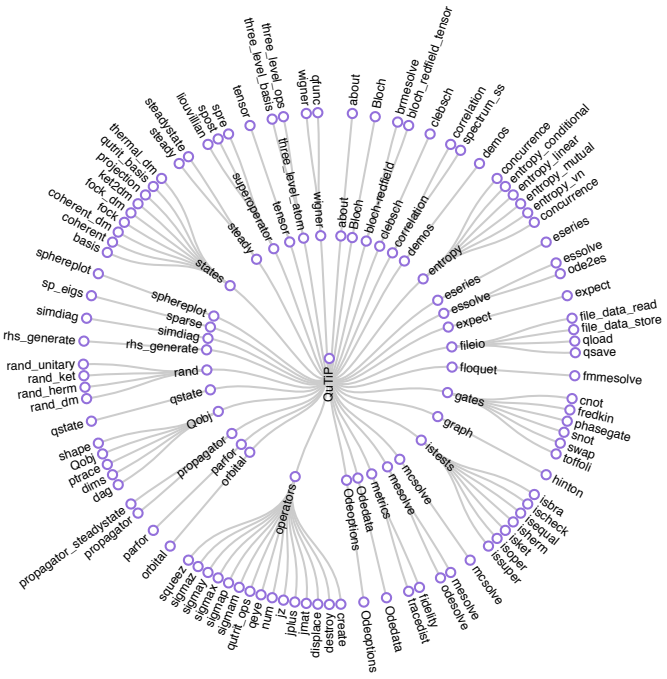

Fig. 1: The organization of user available functions in the QuTiP framework. The inner circle represents submodules, while the outer circle gives the public functions and classes contained in each submodule.

* *Corresponding author: robert@riken.jp*
‡ *Advanced Science Institute, RIKEN, Wako-shi, 351-0198 Japan*

functionality, such as built-in multiprocessing. Our objective with QuTiP is to provide a thoroughly tested and well documented generic framework that can be used for a diverse set of quantum mechanical problems, that encourages openness, verifiability, and reproducibility of published results in the computational quantum mechanics community.

Numerical quantum mechanics

In quantum mechanics, the state of a system is represented by the wavefunction Ψ, a probability amplitude that describes, for example, the position and momentum of a particle. The wavefunction is in general a function of space and time, and its evolution is ideally governed by the Schrödinger equation, $-i\partial_t\Psi = \hat{H}\Psi$, where $\hat{H}$ is the Hamiltonian that describes the energies of the possible states of the system (total energy function). In general, the Schrödinger equation is a linear partial differential equation. For computational purposes, however, it is useful to expand the wavefunction, Hamiltonian, and thus the equation of motion, in terms of basis functions that span the state space (Hilbert space), and thereby obtain a matrix and vector representation of the system. Such a representation is not always feasible, but for many physically relevant systems it can be an effective approach when used together with a suitable truncation of the basis states that often are infinite. In particular, systems that lend themselves to this approach includes resonator modes and systems that are well characterized by a few quantum states (e.g., the two quantum levels of an electron spin). These components also represent the fundamental building blocks of engineered quantum devices.

In the matrix representation the Schrödinger equation can be written as

$$-i\frac{d}{dt}|\psi\rangle = H(t)|\psi\rangle, \tag{1}$$

where $|\psi\rangle$ is a state vector and H is the Hamiltonian matrix. Note that the introduction of complex values in (1) is a fundamental property of evolution in quantum mechanics. In this representation, the equations of motion are a system of ordinary differential equations (ODEs) in matrix form with, in general, time-dependent coefficients. Therefore, to simulate the dynamics of a quantum system we need to obtain the matrix representation of the Hamiltonian and the initial state vector in the chosen basis. Once this is achieved, we have a numerically tractable problem, that involves solving systems of coupled ODEs defined by complex-valued matrices and state vectors.

The main challenge in numerical simulation of quantum systems is that the required number basis states, and thus the size of the matrices and vectors involved in the numerical calculations, quickly become excessively large as the size of the system under consideration is increased. In a composite system, the state space increases exponentially with the number of components. Therefore, in practice only relatively small quantum systems can be simulated on classical computers with reasonable efficiency. Fortunately, in essentially all experimentally realizable systems, the local nature of the physical interactions gives rise to system Hamiltonians containing only a few nonzero elements that are thus highly sparse. This sparsity plays a fundamental role in the efficiency of simulations on quantum computers as well [Aha03]. The exact number of states that can be managed depends on the detailed nature of the problem at hand, but the upper limit is typically on the order of a few thousand quantum states. Many experimentally relevant systems fall within this limit, and numerical simulations of quantum systems on classical computers is therefore an important subject.

Although the state of an ideal quantum systems is completely defined by the wavefunction, or the corresponding state vector, for realistic systems we also need to describe situations where the true quantum state of a system is not fully known. In such cases, the state is represented as a statistical mixture of state vectors $|\psi_n\rangle$, that can conveniently be expressed as a state (density) matrix $\rho = \sum_n p_n |\psi_n\rangle\langle\psi_n|$, where p_n is the classical probability that the system is in the state $|\psi_n\rangle$. The need for density matrices, instead of wavefunctions, arises in particular when modeling open quantum system, where the system's interaction with its surrounding is included. In contrast to the Schrödinger equation for closed quantum systems, the equation of motion for open systems is not unique, and there exists a large number of different equations of motion (e.g., Master equations) that are suitable for different situations and conditions. In QuTiP, we have implemented many of the most common equations of motion for open quantum systems, and provide a framework that can be extended easily when necessary.

The QuTiP framework

As a complete framework for computational quantum mechanics, QuTiP facilitates automated matrix representations of states and operators (i.e. to construct Hamiltonians), state evolution for closed and open quantum systems, and a large library of common utility functions and operations. For example, some of the core functions that QuTiP provides are: `tensor` for constructing composite states and operators from its fundamental components, `ptrace` for decomposing states into their components, `expect` for calculating expectation values of measurement outcomes for an operator and a given state, an extensive collection of functions for generating frequently used states and operators, as well as additional functions for entanglement measures, entropy measures, correlations and much more. A visual map of the user-accessible functions in QuTiP is shown in Fig. 1. For a complete list of functions and their usage, see the QuTiP user guide [Nat12].

The framework is designed so that its syntax and procedures mirror, as closely as possible, the standard mathematical formulation of a quantum mechanical problem. This is achieved thanks to the Python language syntax, and an object-oriented design that is centered around the class `Qobj`, used for representing quantum objects such as states and operators.

In order to simulate the quantum evolution of an arbitrary system, we need an object that not only incorporates both states and operators, but that also keeps track of important properties for these objects, such as the composite structure (if any) and the Hermicity. This later property is especially important as all physical observables are Hermitian, and this dictates when real values should be returned by functions corresponding to measurable quantities. In QuTiP, the complete information for any quantum object is included in the `Qobj` class. This class is the fundamental data structure in QuTiP. As shown in Fig. 2, the `Qobj` object can be thought of as a container for the necessary properties need to completely characterize a given quantum object, along with a collection of methods that act on this operator alone.

A typical simulation in QuTiP takes the following steps:

- Specify system parameters and construct Hamiltonian, initial state, and any dissipative quantum (`Qobj`) objects.

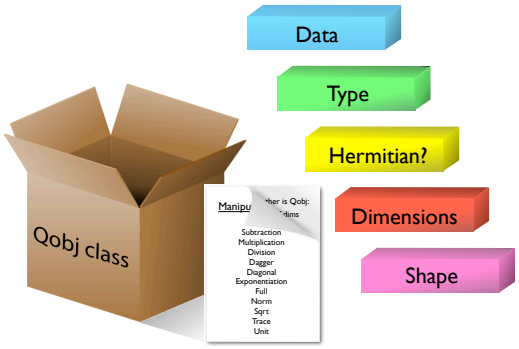

Fig. 2: *`Qobj` class used for defining quantum objects. The class properties include the sparse matrix representation of the object (data), the type of object represented, a nested list describing the composite structure (dimensions), whether the object is Hermitian, and the shape of the underlying data matrix. Also included is a lengthy set of operations acting on the `Qobj`, a list of which can be found at [Nat12].*

- Calculate the evolution of the state vector, or density matrix, using the system Hamiltonian in the appropriate solver.
- Post-process output `Qobj` and/or arrays of return values, including visualization.

Given the generality of this process, we highlight each of these steps below by demonstrating the setup and simulation of select real-world examples.

Constructing Hamiltonians and states

The first step in any QuTiP simulation is the creation of the Hamiltonian that describes the system of interest, initial state, and any possible operators that characterize the interaction between the system and its environment. Although it is possible to directly input a system Hamiltonian into a `Qobj` class object, QuTiP includes a number of predefined operators for oscillator and spin systems out of which a large collection of Hamiltonians can be composed. The simplest, and most common, example is the so-called Jaynes-Cummings model for a two-level atom (qubit) interacting with a single harmonic oscillator [Jay63]

$$\hat{H} = \hbar\omega_c \hat{a}^\dagger \hat{a} + \hbar\omega_q \hat{\sigma}_z/2 + \hbar g/2\left(\hat{a}\hat{\sigma}_+ + \hat{a}^\dagger \hat{\sigma}_-\right) \qquad (2)$$

where the first term in (2) describes the oscillator in terms of creation operators, the second gives the bare qubit Hamiltonian, and the final term characterizes the interaction between oscillator and qubit. Here, ω_c is the oscillator frequency, ω_q is the qubit energy splitting frequency, and g gives the strength of the oscillator-qubit coupling. Typically one is interested in the exchange of a single excitation between the qubit and oscillator. Although the oscillator has an infinite number of states, in this case, we can truncate the Hilbert space. For the initial state with the excitation in the qubit, this state may be written in QuTiP as (we omit the `from qutip import *` statement):

```
N = 4 # number of oscillator levels to consider
psi_osc = basis(N)
psi_qubit = basis(2,1)
psi_sys = tensor(psi_osc,psi_qubit)
```

where `basis(N,m)` creates a basis function of size `N` with a single excitation in the `m` level, and the `tensor` function creates the composite initial state from the individual state vectors for the oscillator and qubit subsystems. The total Hamiltonian (2) can be created in a similar manner using built-in operators and user defined system parameters:

```
wc = wq = 1.0
g = 0.1
a = tensor(destroy(N),qeye(2))
sz = tensor(qeye(N),sigmaz())
sp = tensor(qeye(N),sigmap())
sm = tensor(qeye(N),sigmam())
H = wc*a.dag()*a + wq/2.*sz + g/2.*(a*sp+a.dag()*sm)
```

This final Hamiltonian is a `Qobj` class object representing the Jaynes-Cummings model and is created with a syntax that closely resembles the mathematical formulation given in Eq. (2). Using the `print` function, we can list all of the properties of `H` (omitting the underlying data matrix):

```
Quantum object: dims = [[4, 2], [4, 2]],
shape = [8, 8], type = oper, isHerm = True
```

showing the composite (4×2) structure, the type of object, and verifying that indeed the Hamiltonian is Hermitian as required. Having created collapse operators, if any, we are now in a position to pass the Hamiltonian and initial state into the QuTiP evolution solvers.

Time-evolution of quantum systems

The time-evolution of an initial state of a closed quantum system is completely determined by its Hamiltonian. The evolution of an open quantum system, however, additionally depends on the environment surrounding the system. In general, the influence of such an environment cannot be accounted for in detail, and one need to resort to approximations to arrive at a useful equation of motion. Various approaches to this procedure exist, which results in different equations of motion, each suitable for certain situations. However, most equations of motion for open quantum systems can be characterized with the concept of collapse operators, which describe the effect of the environment on the system and the rate of those processes. A complete discussion of dissipative quantum systems, which is outside the scope of this paper, can be found in [Joh12] and references therein.

QuTiP provides implementations of the most common equations of motion for open quantum systems, including the Lindblad master equation (`mesolve`), the Monte-Carlo quantum trajectory method (`mcsolve`), and certain forms of the Bloch-Redfield (`brmesolve`) and Floquet-Markov (`fmmesolve`) master equations. In QuTiP, the basic type signature and the return value are the same for all evolution solvers. The solvers take following parameters: a Hamiltonian `H`, an initial state `psi_sys`, a list of times `tlist`, an optional list of collapse operators `c_ops` and an optional list of operators for which to evaluate expectation values. For example,

```
c_ops = [sqrt(0.05) * a]
expt_ops = [sz, a.dag() * a]
tlist = linspace(0, 10, 100)
out = mesolve(H, psi_sys, tlist, c_ops, expt_ops)
```

Each solver returns (`out`) an instance of the class `Odedata` that contains all of the information about the solution to the problem, including the requested expectation values, in `out.expect`. The evolution of a closed quantum system can also be computed using the `mesolve` or `mcsolve` solvers, by passing an empty list in place of the collapse operators in the fourth argument. On top of this shared interface, each solver has a set of optional function parameters and class members in `Odedata`, allowing

for modification of the underlying ODE solver parameters when necessary.

Visualization

In addition to providing a computational framework, QuTiP also implements of a number of visualization methods often employed in quantum mechanics. It is of particular interest to visualize the state of a quantum system. Quantum states are often complex superpositions of various basis states, and there is an important distinction between pure quantum coherent superpositions and statistical mixtures of quantum states. Furthermore, the set of all quantum states also includes the classical states, and it is therefore of great interest to visualize states in ways that emphasize the differences between classical and quantum states. Such properties are not usually apparent by inspecting the numerical values of the state vector or density matrix, thus making quantum state visualization techniques an important tool.

Bloch sphere

A quantum two-level system (qubit), can not only occupy the two classical basis states, e.g., "0" and "1", but an arbitrary complex-valued superposition of those two basis states. Such states can conveniently be mapped to, and visualized as, points on a unit sphere, commonly referred to as the Bloch sphere. QuTiP provides a class `Bloch` for visualizing individual quantum states, or lists of data points, on the Bloch sphere. Internally it uses matplotlib to render a 3D view of the sphere and the data points. The following code illustrates how the `Bloch` class can be used

```
bs = Bloch()
bs.add_points([x, y, z])
bs.show()
```

where `x`, `y`, and `z` are the expectation values for the operators σ_x, σ_y, and σ_z, respectively, for the given states. The expectation values can be obtained from the `Odedata` instance returned by a time-evolution solver, or calculated explicitly for a particular state, for example

```
psi = (basis(2,0) + basis(2,1)).unit()
op_axes = sigmax(), sigmay(), sigmaz()
x, y, z = [expect(op, psi) for op in op_axes]
```

In Fig. 5, the time-evolution of a two-level system is visualized on a Bloch sphere using the `Bloch` class.

Quasi-probability distributions

One of goals in engineered quantum systems is to manipulate the system of interest into a given quantum state. Generating quantum states is a non-trivial task as classical driving fields typically lead to classical system states, and the environment gives rise to noise sources that destroy the delicate quantum superpositions and cause unwanted dissipation. Therefore, it is of interest to determine whether the state of the system at a certain time is in a non-classical state. One way to verify that the state of a system is indeed quantum mechanical is to visualize the state of the system as a Wigner quasi-probability distribution. This Wigner function is one of several quasi-probability distributions that are linear transformations of the density matrix, and thus give a complete characterization of the state of the system [Leh10]. The Wigner function is of particular interest since any negative Wigner values indicate an inherently quantum state. Here we demonstrate the ease of calculating Wigner functions in QuTiP by visualizing

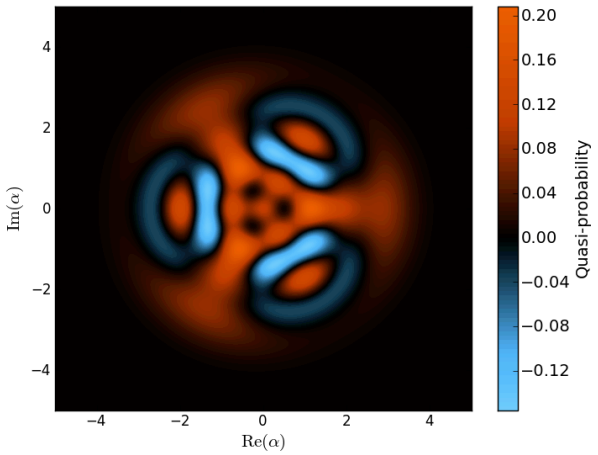

Fig. 3: *Wigner function for the state* $|\Psi\rangle = \frac{1}{\sqrt{3}}[|0\rangle + |3\rangle + |6\rangle]$ *as reconstructed experimentally in [Hof09] . Negative (blue) values indicate that this state is inherently quantum mechanical. The x- and y-axes represent the oscillator position and momentum, respectively.*

the quantum oscillator state $|\Psi\rangle = \frac{1}{\sqrt{3}}[|0\rangle + |3\rangle + |6\rangle]$ recently generated in a superconducting circuit device [Hof09]:

```
psi = (basis(10)+basis(10,3)+basis(10,6)).unit()
xvec = linspace(-5,5,250)
X,Y = meshgrid(xvec, xvec)
W = wigner(psi,xvec,xvec)
```

Again, the quantum state is written in much the same manner as the corresponding mathematical expression with the `basis` functions representing the Fock states $|0\rangle$, $|3\rangle$, and $|6\rangle$ in a truncated Hilbert space with $N = 10$ levels. Here, the `unit` method of the `Qobj` class automatically normalizes the state vector. The `wigner` then takes this state vector (or a density matrix) and generates the Wigner function over the requested interval. The result in shown in Fig. 3.

Example: Multiple Landau-Zener transitions

To demonstrate additional features in QuTiP, we now consider a quantum two-level system, with static tunneling rate Δ and energy-splitting ε, that is subject to a strong driving field of amplitude A coupled to the σ_z operator. In recent years, this kind of system has been actively studied experimentally [Oli05], [Sil06], [Ste12] for its applications in amplitude spectroscopy and Mach-Zehnder interferometry. The system is described by the Hamiltonian

$$\hat{H} = -\frac{\Delta}{2}\hat{\sigma}_x - \frac{\varepsilon}{2}\hat{\sigma}_z - \frac{A}{2}\cos(\omega t)\hat{\sigma}_z, \tag{3}$$

and the initial state $|\psi(t=0)\rangle = |0\rangle$. This is a time-dependent problem, and we cannot represent the Hamiltonian with a single `Qobj` instance. Instead, we can use a nested list of `Qobj` instances and their time-dependent coefficients. In this notation (referred to as list-string notation in QuTiP), the Hamiltonian in Eq. 3 can be defined as

```
H0 = -delta/2 * sigmax() - epsilon/2 * sigmaz()
H1 = sigmaz()
H_td = [H0, [H1, 'A/2 * cos(omega * t)']]
args = {'omega': omega, 'A': A}
```

The QuTiP time-evolution solvers, as well as other functions that use time-dependent operators, then know how to evaluate

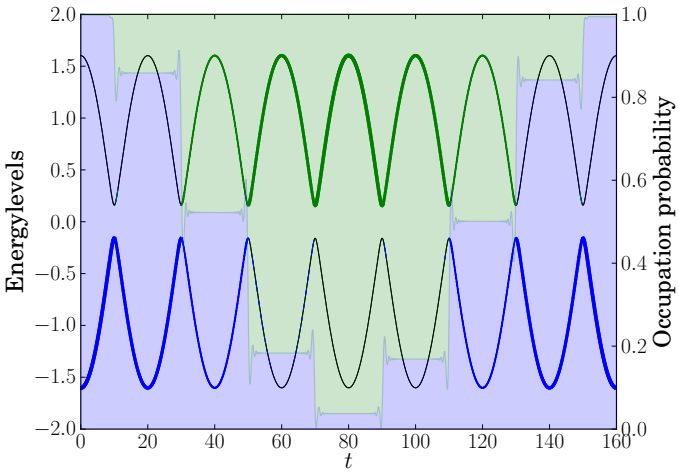

Fig. 4: *Repeated Landau-Zener-like transitions in a quantum two-level system. In each successive sweep through the avoided-level crossing, a small additative change in the occupation probability occurs, and after many crossings a nearly complete state transfer has been achieved. This is an example of constructive interference.*

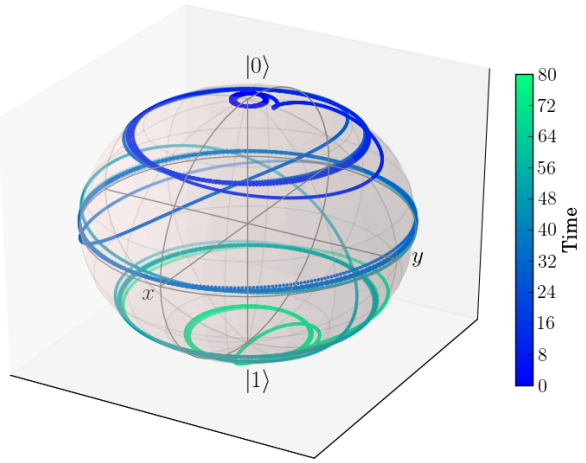

Fig. 5: *Bloch-sphere visualization of the dynamics of a quantum two-level system subject to repeated Landau-Zener-like avoided-level crossings. All the points lay on the surface of the Bloch sphere, so we can immediately conclude that the dynamics is the unitary evolution of a closed quantum system (we did not include any collapse operators in this example).*

the nested list `H_td` to the appropriate operator expression. In this list-string format, this nested list is converted into a Cython source file and compiled. Here, the dictionary `args` is used for passing values of variables that occur in the expression for the time-dependent coefficients. Given this QuTiP representation of the Hamiltonian 3, we can evolve an initial state, using for example the Lindblad master equation solver, with the following lines of code:

```
psi0 = basis(2,0)
tlist = linspace(0, 160, 500)
output = mesolve(H_td, psi0, tlist, [], [], args)
```

Note that here we passed empty lists as fourth and and fifth arguments to the solver `mesolve`, that indicates that we do not have any collapse operators (that is, a closed quantum system) and we do not request any expectation values to be calculated directly by the solver. Instead, we will obtain a list `output.states` that contains the state vectors for each time specified in `tlist`.

These states vectors can be used in further calculations, or for example to visualizing the occupation probabilities of the two states, as show in Figs. 4 and 5. In Fig. 5 we used the previously discussed `Bloch` class to visualize the trajectory of the two-level system.

Implementation and optimization techniques

In implementing the QuTiP framework, we have relied heavily on the excellent Scipy and Numpy packages for Python. Internally, in the class for representing quantum objects, `Qobj`, and in the various time-evolution solvers, we use the sparse matrix from Scipy (in particular the compressed-row format), and in some special cases dense Numpy arrays, for the matrix and vector representation quantum operators and states. Most common quantum mechanics operations can be mapped to the linear algebra operations that are implemented in Scipy for sparse matrices, including matrix-matrix and matrix-vector multiplication, outer and inner products of matrices and vectors, and eigenvalue/eigenvector decomposition. Additional operations that do not have a direct correspondence in matrix algebra, such as the `ptrace` function for decomposing composite states, have been implemented mostly in Python and NumPy. Note that in quantum mechanics it is essential that all matrix and vector elements are complex numbers, and Scipy's thorough support for complex-valued sparse matrices has been a fundamental prerequisite for using Scipy in QuTiP. Overall, Scipy's sparse matrices, and the corresponding functions, have delivered excellent performance. However, we have found that by replacing the built-in matrix-vector multiplication in selected locations with a less general Cython implementation (without, for example type and out-of-bounds checks) we can obtain additional speed-ups.

The ordinary differential equation solver is another feature in Scipy that is used extensively in QuTiP, as most time-evolution solvers use the `scipy.integrate.ode` interface at some level. The configurability and flexibility of Scipy's ODE solver has significantly simplified the implementation of many time-evolution solvers in QuTiP. The Monte-Carlo solver in particular, which is a hybrid method that mixes evolution according to an ODE with stochastic processes, uses some of the more advanced modes of operating Scipy's ODE solver including the high level of control of step size, selectively stopping and restarting the solver, etc.

In a typical simulation using QuTiP, the vast majority of the elapsed time is devoted to evolving ODEs. Fine-tuning Scipy's ODE solver and ensuring that we obtain optimal performance from it has therefore been a priority. Among the optimization measures we have used, the largest impact has been gained by implementing the callback function for the right-hand side (RHS) of the ODE in standard form using Cython. By doing so, a significant amount of overhead related to Python function calls can be avoided, and with the additional speed-up that is gained by evaluating the callback using Cython, this technique has given speed-up factors of up to an order of magnitude or greater [Joh12]. Given this level of speed-up, for any computational problem using Scipy's ODE solver, we would recommend investigating if the callback function

can be implemented in Cython as one of the first performance optimization measures.

One complicating factor that prevents using static Cython implementations for the RHS function with Scipy's ODE, is that in QuTiP the ODEs are generated dynamically by the QuTiP framework. For time-independent problems the RHS function for the ODEs reduce to matrix-vector multiplication, and can be delegated to a pre-compiled Cython function, but in a general time-dependent problem this is not possible. To circumvent this problem, we have employed a method of dynamically generating, compiling and loading Cython code for the RHS callback function. This approach allows us to benefit from the speed-ups gained with a Cython implementation with nontrivial time-dependent RHS functions.

Finally, in implementing QuTiP we have used the Python `multiprocessing` package to parallellize of many time-consuming tasks using the QuTiP `parfor` function, ensuring efficient use of the resources commonly available on modern multicore systems. The Monte-Carlo solver, which requires the evolution of many hundreds of independent ODE systems, is particularly easy to parallelize and has benefited greatly from the `multiprocessing` package, and its good scaling properties as a function of the number of CPU cores.

Conclusions

The Python, Numpy/Scipy and matplotlib environment provides and encourages a unique combination of intuitive syntax and good coding practices, rapid code development, good performance, tight integration between the code and its documentation and testing. This has been invaluable for the QuTiP project. With the additional selective optimization using Cython, QuTiP delivers performance that matches and in many cases exceeds those of natively compiled alternatives [Tan99], accessible through an easy to use environment with a low learning curve for quantum physicists. As a result, sophisticated quantum systems and models can be programmed easily and simulated efficiently using QuTiP.

Acknowledgements

We would like to thank all of the contributors who helped test and debug QuTiP. RJJ and PDN were supported by Japanese Society for the Promotion of Science (JSPS) Fellowships P11505 and P11202, respectively. Additional support comes from Kakenhi grant Nos. 2302505 (RJJ) and 2301202 (PDN).

References

[Aha03] D. Aharonov and A. Ta-Shma, *Adiabatic quantum state generation and statistical zero knowledge*, ACM Symposium on Theory of Computing 20, 2003, available at quant-ph/0301023.

[Bla12] R. Blatt and C. F. Roos, *Quantum simulations with trapped ions*, Nat. Physics, 8:277, 2012.

[Fey82] R. Feynman, *Simulating Physics with Computers*, Int. J. Theor. Phys., 21(6):467, 1982.

[Han08] R. Hanson and D. D. Awschalom, *Coherent manipulation of single spins in semiconductors*, Nature, 453:1043, 2008.

[Har06] S. Haroche and J-M. Raimond, *Exploring the Quantum: Atoms, Cavities, and Photons*, Oxford University Press, 2006.

[Hof09] M. Hofheinz et al., *Synthesizing arbitrary quantum states in a superconducting resonator*, Nature, 459:546, 2009.

[Hor97] G. Z. K. Horvath et al., *Fundamental physics with trapped ions*, Contemp. Phys., 38:25, 1997.

[Jay63] E. T. Jaynes and F. W. Cummings, *Comparison of quantum and semiclassical radiation theories with application to the beam maser*, Proc. IEEE 51(1):89 (1963).

[Joh12] J. R. Johansson et al., *QuTiP: An open-source Python framework for the dynamics of open quantum systems*, Comp. Phys. Commun., 183:1760, 2012, available at arXiv:1110.0573.

[Lad10] T. D. Ladd et al., *Quantum computers*, Nature, 464:45, 2010.

[Leh10] U. Leonhardt, *Essential Quantum Optics*, Cambridge, 2010.

[Nat12] P. D. Nation and J. R. Johansson, *QuTiP: Quantum Toolbox in Python*, Release 2.0, 2012, available at www.qutip.org.

[Obr09] J. L. O'Brien et al., *Photonic quantum technologies*, Nat. Photonics, 3:687, 2009.

[Oco10] A. D. O'Connell et al., *Quantum ground state and single-phonon control of a mechanical resonator*, Nature, 464:697, 2010.

[Sch97] R. Schack and T. A. Brun, *A C++ library using quantum trajectories to solve quantum master equations*, Comp. Phys. Commun., 102:210, 1997.

[Tan99] S. M. Tan, *A computational toolbox for quantum and atomic optics*, J. Opt. B: Quantum Semiclass. Opt., 1(4):424, 1999.

[Vuk07] A. Vukics and H. Ritsch, *C++QED: an object-oriented framework for wave-function simulations of cavity QED systems*, Eur. Phys. J. D, 44:585, 2007.

[You11] J. Q. You and F. Nori, *Atomic Physics and Quantum Optics Using Superconducting Circuits*, Nature, 474:589, 2011.

[Oli05] W. D. Oliver et al., *Mach-Zehnder Interferometry in a Strongly Driven Superconducting Qubit*, Science. 310:1653, 2005.

[Sil06] M. Sillanpää et al., *Continuous-Time Monitoring of Landau-Zener Interference in a Cooper-Pair Box*, Phys. Rev. Lett., 96:187002, 2006.

[Ste12] J. Stehlik et al., *Landau-Zener-Stuckelberg Interferometry of a Single Electron Charge Qubit*, ArXiv:1205.6173, 2012.