QuTiP²
The Quantum Toolbox in Python

QuTiP: Quantum Toolbox in Python

Robert Johansson
robert@riken.jp
RIKEN

Paul Nation
pnation@riken.jp
RIKEN / Korea University
What is QuTiP?

- Framework for **computational quantum dynamics**
  - Efficient and easy to use for quantum physicists
  - Thoroughly tested (100+ unit tests)
  - Well documented (200+ pages, 50+ examples)
- Suitable for
  - theoretical work
  - modeling experiments on engineered quantum systems
- 100% open source
- Implemented in Python/Cython using SciPy, Numpy, and matplotlib
Project information

Authors: Paul Nation and Robert Johansson
Web site: http://qutip.googlecode.com
Discussion: Google group “qutip”
Blog: http://qutip.blogspot.com
Platforms: Linux and Mac
License: GPLv3
Download: http://code.google.com/p/qutip/downloads
SVN Repository: https://qutip.googlecode.com/svn/
QuTiP$^2$
The Quantum Toolbox in Python

What can QuTiP be used for?

*Simulate engineered quantum devices*

- Superconducting circuits
  - Martinis group at UCSB
- Semiconducting circuits
  - Petta group at Princeton
- Nanomechanical devices
  - Roukes group at Caltech
- Ion traps
  - Monroe group at Maryland
Quantum mechanics in one slide …
… not as difficult as one might think: It's only linear algebra!

<table>
<thead>
<tr>
<th>Key concepts</th>
<th>Practical representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Wavefunction / state:</strong></td>
<td><strong>Vectors or matrices:</strong></td>
</tr>
<tr>
<td>Probability amplitude describing the state of a quantum system.</td>
<td>Complex elements, unitary norm/trace.</td>
</tr>
<tr>
<td><strong>Hamiltonian / operators:</strong></td>
<td><strong>Matrices:</strong></td>
</tr>
<tr>
<td>The Hamiltonian is the total energy function of a system, describes the energies of the possible states. Operators represents physical observables, and are used to construct Hamiltonians.</td>
<td>Hermitian, complex elements, usually sparse.</td>
</tr>
<tr>
<td><strong>Equation of motion:</strong></td>
<td><strong>Ordinary differential equations (ODEs)</strong></td>
</tr>
<tr>
<td>Describes how the states of a system evolve in time for a given Hamiltonian. Examples: Schrödinger equation, Master equations and Monte-Carlo.</td>
<td>Systems of coupled ODEs in matrix form. In general time-dependent. Sometimes including stochastic processes.</td>
</tr>
<tr>
<td><strong>Observables and expectation values:</strong></td>
<td><strong>Inner product</strong></td>
</tr>
<tr>
<td>Observable physical quantities correspond to operators. Measurement outcomes are “stochastic”, expectation values calculated using the wavefunction.</td>
<td>Measurement outcomes are calculated as inner product between state vectors and operator matrices, resulting in a real number for physical observables.</td>
</tr>
</tbody>
</table>
What we want to accomplish with QuTiP

Objectives

To provide a powerful framework for quantum mechanics that closely resembles the standard mathematical formulation

- Efficient and easy to use
- General framework, able to handle a wide range of different problems

Design and implementation

- Object-oriented design
- Qobj class used to represent quantum objects
  - Operators
  - State vectors
  - Density matrices
- Library of utility functions that operate on Qobj instances

QuTiP core class: Qobj
Quantum object class: Qobj

Abstract representation of quantum states and operators
- Matrix representation of the object
- Structure of the underlaying state space, Hermiticity, type, etc.
- Methods for performing all common operations on quantum objects:
  - eigs(), dag(), norm(), unit(), expm(), sqrt(), tr(), ...
- Operator arithmetic with implementations of: +, -, *, ...

Example: built-in operator $\hat{\sigma}_x$

```python
>>> sigmax()
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = True
Qobj data =
[[ 0.  1.]
 [ 1.  0.]]
```

Example: built-in state $|\alpha = 0.5\rangle$

```python
>>> coherent(5, 0.5)
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.88249693]
 [ 0.44124785]
 [ 0.15601245]
 [ 0.04496584]
 [ 0.01173405]]
```
Calculating using Qobj instances

Basic operations

# operator arithmetic
>> H = 2 * sigmax() + 0.5 * sigmax()

Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = True
Qobj data =
[[ 2.  0.5]
 [ 0.5 -2. ]]

# superposition states
>> psi = (basis(2,0) + basis(2,1))/sqrt(2)

Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.70710678]
 [ 0.70710678]]

# expectation values
>> expect(num(2), psi)

0.49999999999999999

>> N = 25
>> psi = (coherent(N,1) + coherent(N,3)).unit()
>> expect(num(N), psi)

4.761589143572134

Composite systems

# operators
>> sx = sigmax()
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = True
Qobj data =
[[ 0.  1.]
 [ 1.  0.]]

>> sxsx = tensor([sx,sx])
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isHerm = True
Qobj data =
[[ 0.  0.  0.  1.]
 [ 0.  0.  1.  0.]
 [ 0.  1.  0.  0.]
 [ 1.  0.  0.  0.]]

# states
>> psi_a = fock(2,1); psi_b = fock(2,0)
>> psi = tensor([psi_a, psi_b])
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.70710678]
 [ 0.70710678]]

# transform an operator to the eigenbasis of H
>> sx_eb = sigmax().transform(evecs)
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = True
Qobj data =
[[ 1.  0.]
 [ 0.  1.]]

Basis transformations

# eigenstates and values for a Hamiltonian
>> H = sigmax()
>> evals, evects = H.eigenstates()
>> evals
array([-1.,  1.])
>> evects
array([

Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[-0.70710678]
 [ 0.70710678]],
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.70710678]
 [ 0.70710678]], dtype=object)

# transform an operator to the eigenbasis of H
>> sx_eb = sigmax().transform(evects)
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = True
Qobj data =
[[ 1.  0.]
 [ 0.  1.]]
QuTiP²
The Quantum Toolbox in Python

Organization

- States
- Operators
- Time evolution
- Visualization
- Gates
- Core/Utility functions
Evolution of quantum systems

The main use of QuTiP is quantum evolution. A number of solvers are available.

Typical simulation workflow:

i. Define parameters that characterize the system

ii. Create Qobj instances for operators and states

iii. Create Hamiltonian, initial state and collapse operators, if any

iv. Choose a solver and evolve the system

v. Post-process, visualize the data, etc.

Available evolution solvers:

- Unitary evolution: Schrödinger and von Neumann equations
- Lindblad master equations
- Monte-Carlo quantum trajectory method
- Bloch-Redfield master equation
- Floquet-Markov master equation
- Propagators
Example: Jaynes-Cummings model

Mathematical formulation:

**Hamiltonian**

\[ \hat{H} = \hbar \omega_c \hat{a}^{\dagger} \hat{a} - \frac{\hbar \omega_q}{2} \hat{\sigma}_z + \frac{\hbar g}{2} (\hat{a} \hat{\sigma}_+ + \hat{a}^{\dagger} \hat{\sigma}_-) \]

**Initial state**

\[ |\psi(t = 0)\rangle = |1\rangle_c \otimes |0\rangle_q \]

**Time evolution**

\[ \frac{d}{dt} |\psi(t)\rangle = \hat{H} |\psi(t)\rangle \]

**Expectation values**

\[ \langle \hat{a}^{\dagger} \hat{a} \rangle = \langle \psi(t) | \hat{a}^{\dagger} \hat{a} | \psi(t) \rangle \]

QuTiP code:

```python
from qutip import *
N = 10

a = tensor(destroy(N),qeye(2))
sz = tensor(qeye(N),sigmaz())
s = tensor(qeye(N),destroy(2))
wc = wq = 1.0 * 2 * pi
g = 0.5 * 2 * pi
H = wc * a.dag() * a - 0.5 * wq * sz +
   0.5 * g * (a * s.dag() + a.dag() * s)
psi0 = tensor(basis(N,0), basis(2,0))
tlist = linspace(0, 10, 100)
out = mesolve(H, psi0, tlist, [a.dag()*a])

from pylab import *
p = plot(tlist, out.expect[0])
show()
```

(a two-level atom in a cavity)
Example: time-dependence

Multiple Landau-Zener transitions

\[ \hat{H}(t) = -\frac{\Delta}{2} \hat{\sigma}_z - \frac{\epsilon}{2} \hat{\sigma}_x - A \cos(\omega t) \hat{\sigma}_z \]

```python
from qutip import *

# Parameters
epsilon = 0.0
delta = 1.0

# Initial state: start in ground state
psi0 = basis(2,0)

# Hamiltonian
H0 = - delta * sigmaz() - epsilon * sigmax()
H1 = - sigmaz()
h_t = [H0, [H1, 'A * cos(w*t)']] args = {'A': 10.017, 'w': 0.025*2*pi}

# No dissipation
c_ops = []

# Expectation values
e_ops = [sigmax(), sigmay(), sigmaz()]

# Evolve the system
tlist = linspace(0, 160, 500)
output = mesolve(h_t, psi0, tlist, c_ops, e_ops, args)

# Process and plot result
# ...
```
Example: open quantum system

\[ \hat{H} = g (\hat{\sigma}_x \otimes \hat{\sigma}_x + \hat{\sigma}_y \otimes \hat{\sigma}_y), \quad t \in [0, T = \pi/4g] \]

```python
from qutip import *
g = 1.0 * 2 * pi  # coupling strength
g1 = 0.75          # relaxation rate
g2 = 0.25          # dephasing rate
n_th = 1.5          # environment temperature
T = pi/(4*g)

H = g * (tensor(sigmax(), sigmax()) + tensor(sigmay(), sigmay()))
c_ops = []
# qubit 1 collapse operators
sm1 = tensor(sigmam(), qeye(2))
sz1 = tensor(sigmaz(), qeye(2))
c_ops.append(sqrt(g1 * (1+n_th)) * sm1)
c_ops.append(sqrt(g1 * n_th) * sm1.dag())
c_ops.append(sqrt(g2) * sz1)
# qubit 2 collapse operators
sm2 = tensor(qeye(2), sigmam())
sz2 = tensor(qeye(2), sigmaz())
c_ops.append(sqrt(g1 * (1+n_th)) * sm2)
c_ops.append(sqrt(g1 * n_th) * sm2.dag())
c_ops.append(sqrt(g2) * sz2)

U = propagator(H, T, c_ops)
qpt_plot(qpt(U, op_basis), op_labels)
```
Advanced example

- 21 spin-$\frac{1}{2}$ with nearest-neighbor interaction
- $\sim$40 lines of code
- 2'097'152 basis states
- simulation time: $\sim$7 hours

```python
from qutip import *

def system_ops(N, h, J, gamma):
    # pre-compute operators
    si = qeye(2); sx = sigmax(); sz = sigmaz()
    sx_list = []; sz_list = []
    for n in range(N):
        op_list = [si for m in range(N)]
        op_list[n] = sx
        sx_list.append(tensor(op_list))
        op_list[n] = sz
        sz_list.append(tensor(op_list))
    # construct the hamiltonian
    H = 0
    for n in range(N):
        # energy splitting terms
        H += - 0.5 * h[n] * sz_list[n]
    for n in range(N-1):
        # interaction terms
        H += - 0.5 * J[n] * sx_list[n] * sx_list[n+1]
    # collapse operators for spin dephasing
    c_op_list = []
    for n in range(N):
        if gamma[n] > 0.0:
            c_op_list.append(sqrt(gamma[n]) * sz_list[n])
    # initial state
    psi_list = [basis(2,0) for n in range(N-2)]
    psi_list.insert(0, basis(2,1)) # first
    psi_list.append(basis(2,1))    # last
    psi0 = tensor(psi_list)
    return H, c_op_list, sz_list, psi0

N = 21           # number of spins
h = 1.0 * 2 * pi * ones(N)    # energy splittings
J = 0.05 * 2 * pi * ones(N)  # coupling
gamma = 0.0 * ones(N)         # dephasing rate

# pre-compute operators
H, c_ops, e_ops, psi0 = system_ops(N, h, J, gamma)

# evolve
tlist = linspace(0, 150, 150)
output = mesolve(H, psi0, tlist, c_ops, e_ops)
sn_expt = (1 - array(output.expect))/2.0

# post-process and plot ...
```
Visualization

- Objectives of visualization in quantum mechanics:
  - Visualize the composition of complex quantum states (superpositions and statistical mixtures).
  - Distinguish between quantum and classical states. Example: Wigner function.

- In QuTiP:
  - Wigner and Q functions, Bloch spheres, process tomography, ...
  - *most common visualization techniques used in quantum mechanics are implemented*
What do we use to implement QuTiP?

- Python core language and the *multiprocessing package*
- *Numpy* and *Scipy*
  - Sparse matrix library in Scipy
  - ODE solve in Scipy
- *Matplotlib*
  - With custom functions for performing many common visualizations for quantum systems, e.g. plotting Wigner distributions, Bloch sphere representations
- *Cython*
  - For various optimizations, most notably for faster sparse-matrix-vector multiplication and for evaluating the derivative callback function for the Scipy ODE solver
  - Dynamic generation and compilation of Cython code using a home-brewed code generator for time-dependent ODE callback functions
Parallelization with multiprocessing package

- Many tasks in QuTiP can be parallelized:
  - Quantum Monte-Carlo: solving and averaging over many ODEs for trajectories with stochastic quantum jumps
  - In applications, we typically need to repeat calculations over a range or grid of parameters

- In QuTiP we use a `parfor` function to parallelize such tasks on multicore machines.
  - `parfor` is implemented using the Python multiprocessing standard package
  - Powerful and very easy-to-use interface for parallel computations

```python
# example use of parfor in QuTiP
#
def task(args):
    rank, param = args
    # compute ...
    return [rank, result]

param_list = linspace(0, 10, 100)
result_list = parfor(task, enumerate(param_list))
```
Optimizations using Cython

- Significant performance gains by replacing our Python callback functions for the SciPy ODE solver with Cython implementations.

- Problem with this approach:
  - Our ODE callback functions are dynamically generated from our abstract quantum object representation, and can in general be time-dependent.
  - The ODE RHS cannot be represented by a static Cython RHS function for general time-dependent problems.

---

```
# python code: from mesolve function, for time-independent Hamiltonian
...
initial_vector = psi0.full()
L = -1.0j * H
r.set_f_params(L.data.data, L.data.indices, L.data.indptr)
L.init
r.set_integrator('zvode', method=opt.method, order=opt.order, ...)
r.set_initial_value(initial_vector, tlist[0])
...```

---

```
# cython code
@cython.boundscheck(False)
cdef int row, jj, row_start, row_end
nrows=len(rho)
cdef CTYPE_t dot
cdef np.ndarray[CTYPE_t, ndim=2] out = np.zeros((nrows,1),dtype=np.complex)
for row in range(nrows):
    row_start = ptr[row]
    row_end = ptr[row+1]
    for jj in range(row_start, row_end):
        dot += data[jj] * rho[idx[jj]]
    out[row,0] = dot
return out
```

---

```
# python code: from mesolve function, for time-independent Hamiltonian
...
initial_vector = psi0.full()
L = -1.0j * H
r.set_f_params(L.data.data, L.data.indices, L.data.indptr)
L.init
r.set_integrator('zvode', method=opt.method, order=opt.order, ...)
r.set_initial_value(initial_vector, tlist[0])
...```
Optimizations using Cython

- Significant performance gains by replacing our Python callback functions for the SciPy ODE solver with Cython implementations

- Solution:
  - Dynamically generate, compile and load Cython code for the ODE RHS

---

Example use of Cython in QuTiP

```python
# python code: from mesolve function, for time-dependent Hamiltonian
...
# code generator
if not opt.rhs_reuse:
    odeconfig.tdname="rhs"+str(odeconfig.cgen_num)
    cgen=Codegen(h_terms=n_L_terms,h_tdterms=Lcoeff, args=args)
    cgen.generate(odeconfig.tdname+".pyx")
    os.environ['CFLAGS']='-O3 -w'
    import pyximport
    pyximport.install(setup_args=['include_dirs':[numpy.get_include()]])
    code = compile('from '+odeconfig.tdname+' import cyq_td_ode_rhs', 'string', 'exec')
    exec(code, globals())
    odeconfig.tdfunc=cyq_td_ode_rhs

# setup integrator
initial_vector = mat2vec(rho0.full())
r = scipy.integrate.ode(odeconfig.tdfunc)
r.set_integrator('zvode', method=opt.method, order=opt.order,
    atol=opt.atol, rtol=opt.rtol, nsteps=opt.nsteps,
    first_step=opt.first_step, min_step=opt.min_step,
    max_step=opt.max_step)
r.set_initial_value(initial_vector, tlist[0])
    code = compile('r.set_f_params('+string+')', 'string', 'exec')
    exec(code)
r.set_initial_value(initial_vector, tlist[0])
...```
Summary

- **QuTiP**: framework for numerical simulations of quantum systems
  - Generic framework for representing quantum states and operators
  - Large number of dynamics solvers

- **Main strengths:**
  - Ease of use: complex quantum systems can programmed rapidly and intuitively
  - Flexibility: Can be used to solve a wide variety of problems
  - Performance: Near C-code performance due to use of Cython for time-critical functions

- **Future developments:**
  - OpenCL? Stochastic master equations?
  - Non-markovian master equations?