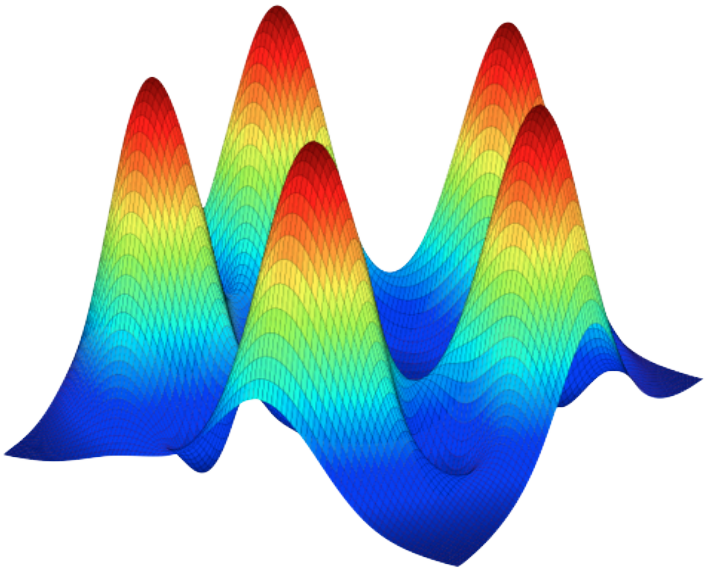
QuTiP: Quantum Toolbox in Python

an open-source framework for the dynamics of open quantum systems

http://code.google.com/p/qutip/



by
P.D. Nation
J.R. Johansson

(RIKEN)

Python and SCIPY

- Python is a modern script language
  - open source
  - very easy to learn and use
  - cross platform (Linux, Mac OS, etc.)
- SCIPY is a collection of high-performance scientific libraries for python, including
  - matrices, vectors, math functions, ode solvers, and much more
  - close integration with matplotlib, a powerful graphics system for production-quality plots
- The Python+SCIPY combination is a competitive alternative to propriety software for numerical calculations, such as MATLAB
  - no license fees
  - in widespread use in the computational physics communities
- QuTiP uses the Python+SCIPY framework to provide an environment for the simulation of open quantum systems

Why use QuTiP?

- Easy to use
- Efficient
  - Performance near that of compiled code: under the hood, it uses optimized low-level BLAS and LAPACK libraries for matrix operations, ARPACK for sparse matrices, etc.
  - Built-in support for CPUs with multiple cores
- Actively developed and easy to extend
- Extensively tested
- A large collection of quantum mechanics related functions
- 100% open source. No propriety software needed: no license fees
- Well suited for running on clusters and supercomputers
- Built-in quantum mechanics related plot functions:
  - Bloch sphere, P-functions, Q-functions, Wigner functions, etc.
- Closely resembles the Quantum Optics toolbox for MATLAB
  - very easy to convert old programs using the QO toolbox to QuTiP

Features

- System for easily creating
  - Hamiltonians and Liouvillians
  - wave functions and density matrices
  - tracing out and combining components of quantum systems
- Functions for calculating
  - Expectation values, entanglement metrics, entropy, fidelity, ...
  - Wigner function, P functions, Q functions, ...
- ODE solvers for the Schrödinger and master equations
- Monte-Carlo stochastic wave function solvers
- Diagonalization, exponentiation, steady state solvers
- Calculating correlation functions using the quantum regression theorem
- Can do anything that the Quantum Optics toolbox can do, and more
  - Arbitrarily time-dependent Hamiltonians and Liouvillians
  - Visualization in Bloch spheres

Basics

- Installation
  - Vary depending on platform
  - See instructions on the project web page for details:

    → http://code.google.com/p/qutip

- Importing the QuTiP environment
  - In the beginning of the program:

```
1: #!/usr/bin/python
2: from qutip import *
```

  - Or directly from the interactive python prompt:

```
>> from qutip import *
```

Quantum objects

- The Qobj class is used for representing all quantum objects:
  - operators, states, Hamiltonians and Liouvillians
  - maintains information about the size and structure of the quantum objects
- Qobj instances can be created
  - manually, using the Qobj constructor that takes an array as argument:

```
>> psi = Qobj([[1], [0]])
>> print psi
Quantum object: dims = [[2], [1]], shape = [2, 1]
Qobj data =
[[1]
 [0]]
```

```
>> sigma_z = Qobj([[0, 1], [1, 0]])
>> print sigma_z
Quantum object: dims = [[2], [2]], shape = [2, 2]
Qobj data =
[[0 1]
 [1 0]]
```

  - From a large library of predefined operators and states (see the following slides)
- Operations on quantum objects:
  - Arithmetic operation: plus, minus, multiplication and division with a C number, as overloaded operators:

```
>> psi_up   = Qobj([[0], [1]])
>> psi_down = Qobj([[1], [0]])
>> psi      = (psi_up + psi_down) / sqrt(2)
>> print psi
Quantum object: dims = [[2], [1]], shape = [2, 1]
Qobj data =
[[ 0.70710678]
 [ 0.70710678]]
```

States and density matrices

- Predefined states (N = size of the Hilbert space)
  - Fock states

```
fock(N, n)                ← wave function
fock_dm(N, n)             ← density matrix
n = the fock state number
```

  - Coherent states

```
coherent(N, alpha)    ← wave function
coherent_dm(N, alpha)    ← density matrix
```

  - Thermal states

```
thermal_dm(N, n)          ← density matrix
n = average number of photons
```

- Superposition states
  - With simple arithmetic syntax:

```
>> psi = ( psi1 + psi2 ) / sqrt(2)
```

- Tensor combine states

```
>> tensor([list of quantum states …])
```

- Combined objects keeps track of its underlying structure

```
>> print tensor([fock(2,0), qeye()])
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4]
...
```

- So that they easily can be decomposed again, by doing a partial trace:

```
>> rho_sub = ptrace(tensor([fock_dm(2,0), fock_dm(2)], 1)
Quantum object: dims = [[2], [2]], shape = [2, 2]
```

```
>> print fock(4, 1)
Quantum object: dims = [[4], [1]], shape = [4, 1]
Qobj data =
[[ 0.]
 [ 1.]
 [ 0.]
 [ 0.]]
>> print fock_dm(4, 1)
Quantum object: dims = [[4], [4]], shape = [4, 4]
Qobj data =
[[ 0.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]

>> rho=tensor([fock_dm(2,0),fock_dm(2,1)])
>> print ptrace(rho,0)
Quantum object: dims = [[2], [2]], shape = [2, 2]
Qobj data =
[[ 1.  0.]
 [ 0.  0.]]
>> print ptrace(rho,1)
Quantum object: dims = [[2], [2]], shape = [2, 2]
Qobj data =
[[ 0.  0.]
 [ 0.  1.]]
```

Operators

- Predefined operators

  (N = number of states in the Hilbert space)

  - Paul spin matrices

    `sigmax(), sigmay() and sigmaz()`
    `sigmap() and sigmam()`

  - Creation and annihilation operators

    `create(N), destroy(N)`

  - Identity operator

    qeye(N)

```
>> print tensor([sigmax(), sigmax()])
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4]
Qobj data =
[[ 0.  0.  0.  1.]
 [ 0.  0.  1.  0.]
 [ 0.  1.  0.  0.]
 [ 1.  0.  0.  0.]]

>> print tensor([sigmax(), qeye(2)])
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4]
Qobj data =
[[ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]
 [ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]]
```

- Tensor combine operators

  `tensor([list of operators, …])`

- Combined objects keeps track of its underlying structure

```
>> print tensor([sigmax(), qeye()])
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4]
...
```

Hamiltonian: examples

- Two-qubit Hamiltonian: $H = \sigma_z^{(1)} + \sigma_z^{(2)} + 0.01\sigma_x^{(1)}\sigma_x^{(2)}$

```
>> sz1  = tensor([sigmaz(), qeye()])
>> sz2  = tensor([qeye(), sigmaz()])
>> sxsx = tensor([sigmax(), sigmax()])
>> H = sz1 + sz2 + 0.1 * sxsx
>> print H
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4]
Qobj data =
[[ 2.   0.   0.   0.1]
 [ 0.   0.   0.1  0. ]
 [ 0.   0.1  0.   0. ]
 [ 0.1  0.   0.  -2. ]]
```

- Jaynes-Cumings Hamiltonian: $H = \omega a^\dagger a + \epsilon\sigma_z + g(a^\dagger\sigma^- + a\sigma^+)$

```
>> N = 10; w = 1; eps = 1; g = 0.01
>> a = tensor([destroy(N), qeye(2)])
>> sz = tensor([qeye(N), sigmaz()])
>> sm = tensor([qeye(N), sigmam()])
>> H = w * a.dag() * a + eps * sz + g * (a.dag() * sm + a * sm.dag())
>> print H
Quantum object: dims = [[10, 2], [10, 2]], shape = [20, 20]
Qobj data =
...
```

Unitary time evolution

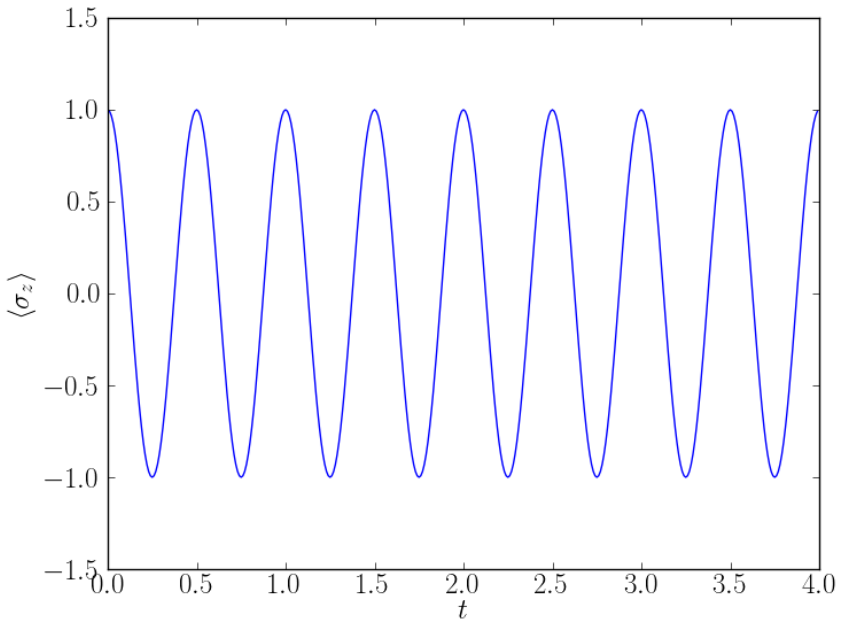
- ODE solver for the Schrödinger equation:

```
expt_vals = ode_solve(H, psi0, tlist, [], expt_op_list)
```

  - H = a Hamiltonian
  - psi0 = an initial wave function
  - tlist = a list of times at which to store the results
  - expt_op_list = list of operators for which to calculate the expectation value for, at the times specified in tlist

Example:

```
>> H = 2 * pi * sigmax()
>> psi0 = fock(2, 0)                    # |0> state
>> tlist = arange(0.0, 4.0, 0.01)
>> expt_sz = ode_solve(H, psi0, tlist,[], [sigmaz()])
>> plot(tlist, expt_sz[0])
>> show()
```



Dissipation: Master equation

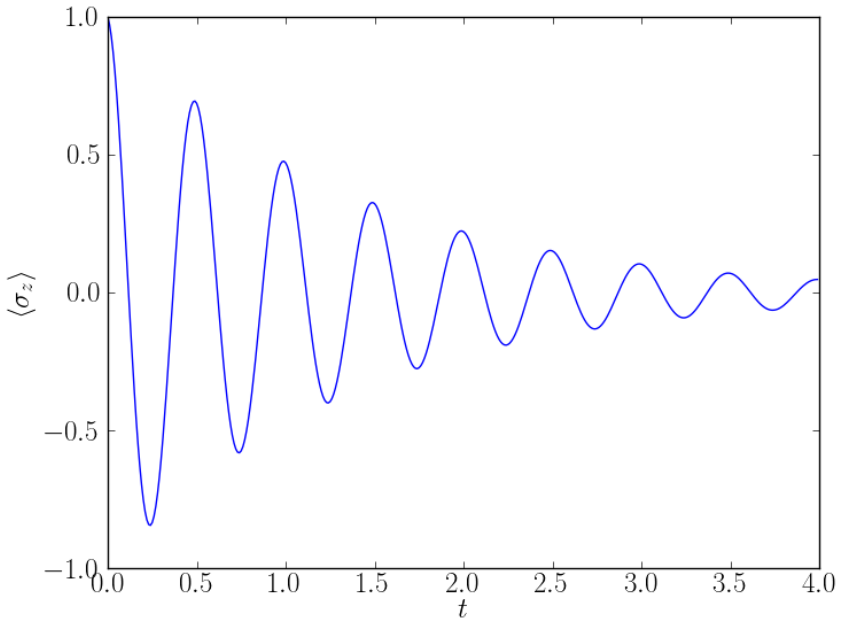
- ODE solver for the master equation:

```
expt_vals = ode_solve(H, psi0, tlist, c_op_list, expt_op_list)
```

  - H = a Hamiltonian
  - psi0 = an initial wave function
  - tlist = a list of times at which to store the results
  - c_op_list = list of collapse operators
  - expt_op_list = list of operators for which to calculate the expectation value for, at the times specified in tlist

Example:

```
>> H = 2 * pi * sigmax()
>> psi0 = fock(2, 0)                    # |0> state
>> gamma1 = 1.0
>> c_ops = [sqrt(gamma1) * sigmam()] # relaxation
>> expt_ops = [sigmaz()]
>> tlist = arange(0.0, 4.0, 0.01)
>> expt_sz = ode_solve(H, psi0,tlist,c_ops,expt_ops)
>> plot(tlist, expt_sz[0])
```



Dissipation: Steady state

- Finding the steady state of a open quantum system:

```
rho_ss = steadystate(H, c_op_list)
```

  - H = a Hamiltonian
  - c_op_list = list of collapse operators
  - rho_ss = the steady state density matrix

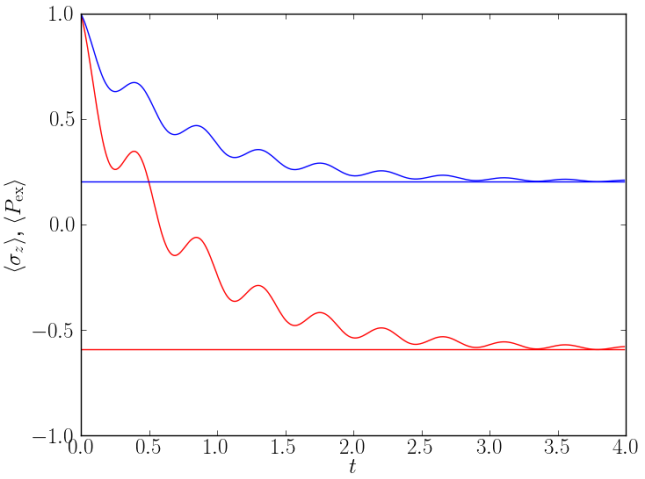
Example:

```
>> H = 2 * pi * (1.0 * sigmaz() + 0.5 * sigmax())
>> sm   = sigmam()
>> g1   = 1.0  # rate
>> n_th = 0.25 # temperature factor
>> c_op_list = [sqrt(g1*(n_th+1)) * sm, sqrt(g1*n_th) * sm.dag()]
>> rho_ss = steadystate(H, c_op_list)
>> print "<sigma_z>_ss =", expect(sigmaz(), rho_ss)

<sigma_z>_ss = -0.592826392093

>> print "<P_ex>_ss    =", expect(sm.dag() * sm, rho_ss)

<P_ex>_ss    = 0.203586803953
```



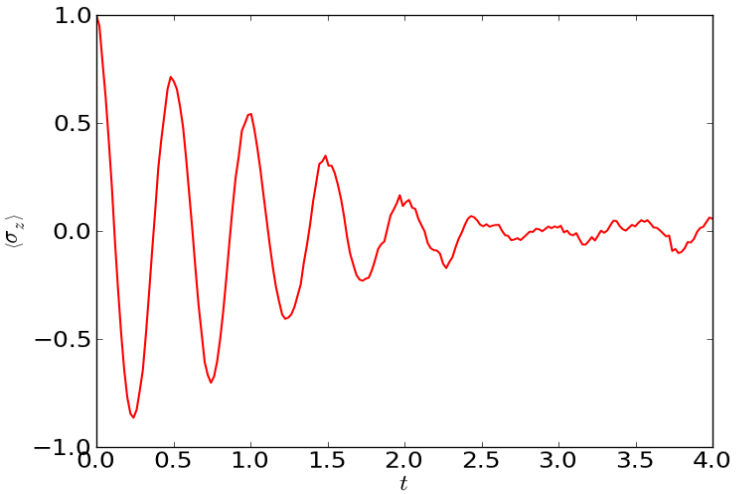
Dissipation: Monte-Carlo solver

- Stochastic wavefunction Monte-Carlo solver

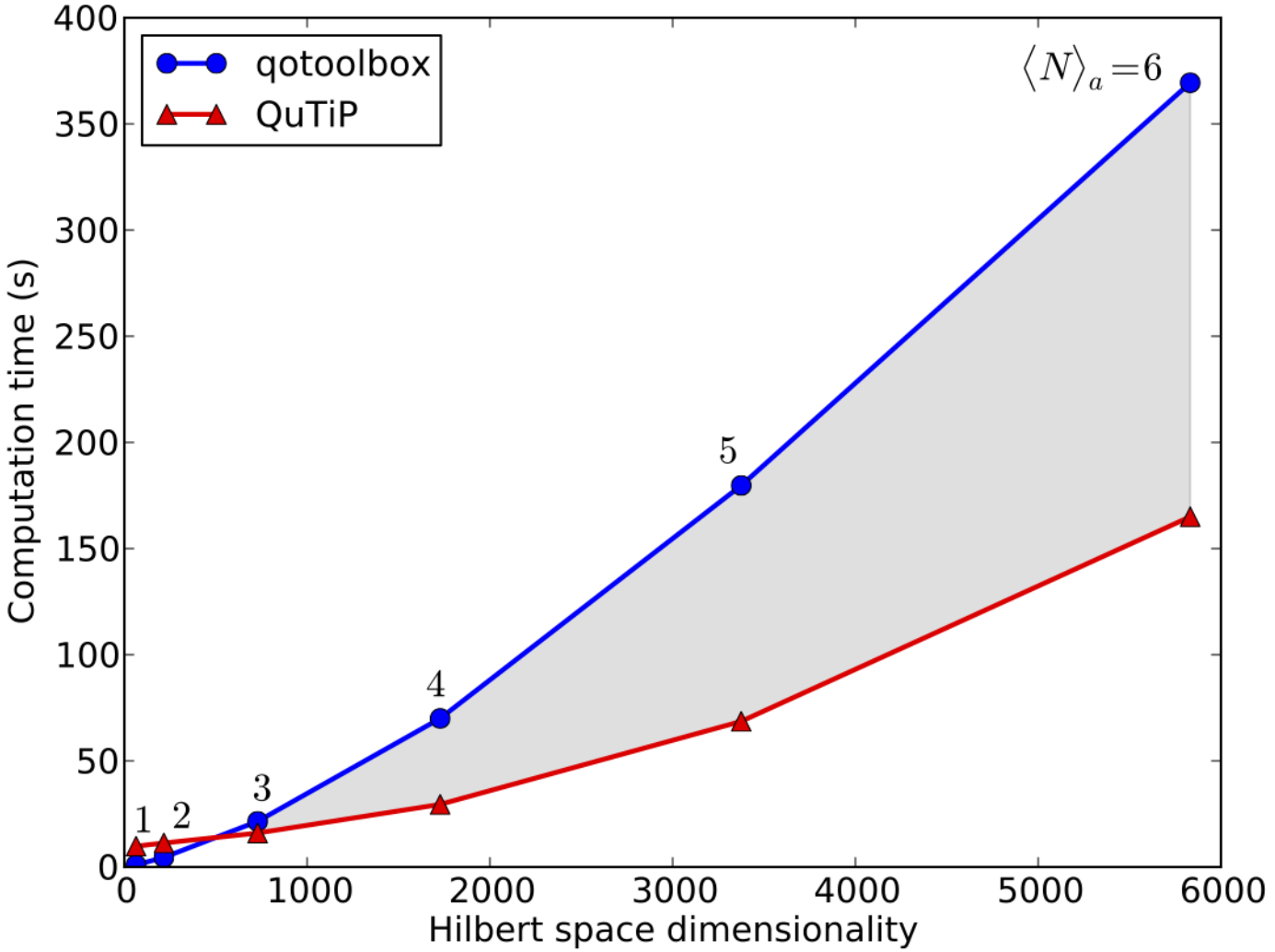
```
expt_vals = mcsolve(H, psi0, tlist, ntraj, c_op_list, expt_op_list)
```

  - H = a Hamiltonian
  - psi0 = an initial wave function
  - tlist = a list of times at which to store the results
  - ntraj = number of monte carlo trajectories
  - c_op_list = list of collapse operators
  - expt_op_list = list of operators for which to calculate the expectation value for, at the times specified in tlist

```
>> H = 2 * pi * sigmax()
>> psi0 = fock(2, 0)                    # |0> state
>> gamma1 = 1.0
>> c_ops = [sqrt(gamma1) * sigmam()] # relaxation
>> expt_ops = [sigmaz()]
>> tlist = arange(0.0, 4.0, 0.01)
>> ntraj = 100
>> expt_sz = mcsolve(H, psi0,tlist,ntraj,c_ops,expt_ops)
>> plot(tlist, expt_sz[0])
```



Monte-Carlo solver: Performance



Time-dependent Hamiltonian

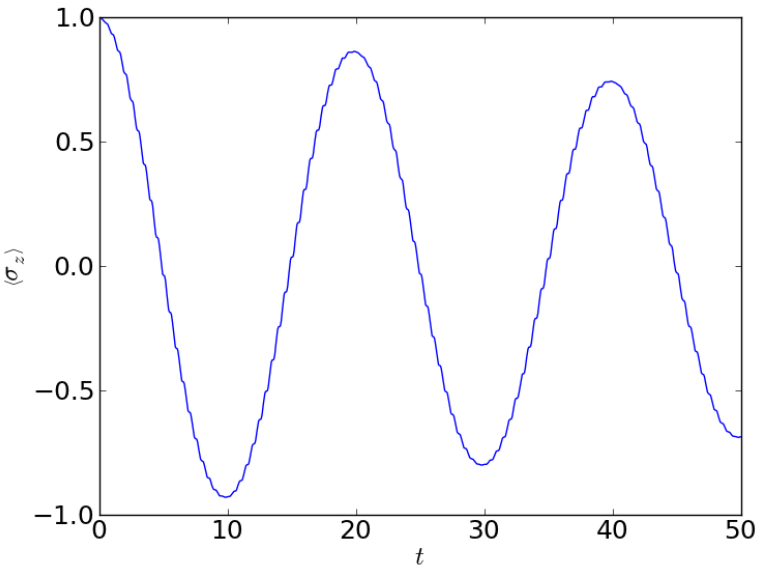
- ODE solver for the time-dependent master equation:

```
expt_vals = ode_solve(H_func, psi0, tlist, c_list, e_list, H_param)
```

  - H _func = a callback function returning the Hamiltonian at a time t
  - psi0 = an initial wave function
  - tlist = a list of times at which to store the results
  - c_list = list of collapse operators
  - e_list = list of operators for which to calculate the expectation value for
  - H_args = arguments for the function H_func(t) that calculates the Hamiltonian

Example: Rabi oscillations with damping

```
>> def H_t(t, args):
>>     H0 = args[0]
>>     H1 = args[1]
>>     w  = args[2]
>>     return H0 + H1 * sin(w * t)
>>
>> H0 = - 2*pi * 1.0/2.0  * sigmaz()
>> H1 = - 2*pi * 0.05 * sigmax()
>> H_args = (H0, H1, 2*pi*1.0)
>> psi0 = fock(2, 0)                    # |0> state
>> clst = [sqrt(0.01) * sigmam()] # relaxation
>> tlst = arange(0.0, 50.0, 0.01)
>> expt_sz = ode_solve(H_t,psi0,tlst,clst,[sigmaz()],H_args)
```



Conclusions

- Introduced QuTiP:
  - an open-source framework for the time evolution of open quantum systems
- Described how to setup simple calculations
- Described the main solvers for quantum evolution
  - Master equation, Monte-Carlo, steadystate, time-dependent solver
- Feel free to try it out. It's free!
  - http://code.google.com/p/qutip