pyqo Documentation

Release 0.1

Sebastian Krämer

October 25, 2011

CONTENTS

Contents:

CHAPTER

ONE

PYQO TUTORIAL

pyqo is a python library similar to the *quantum optics toolbox* for Matlab. It lets the user define state vectors and operators and provides functionality to solve typical problems occurring in quantum optics, e.g. solving Schroedinger or master equations.

1.1 Importing pyqo

To be able to use **pyqo** from a python script the **pyqo** library has to be in your `PYTHONPATH` or in the same directory as the script that uses it. Then it can for example be imported as:

```
>>> import pyqo as qo
```

In all following examples it is assumed that **pyqo** was imported in that way.

1.2 Defining states

All objects in **pyqo** are defined as tensors of different ranks. Taking advantage of the power of numpy, they inherit from the mighty `numpy.ndarray`. Obviously their dimensionality is limited by the memory available on your system so it is impossible to calculate in infinite dimensional Hilbert spaces. For quantum states **pyqo** provides the class `pyqo.StateVector`. It can be used to directly create states from nested lists or tuples (or anything else that a numpy array can handle):

```
>>> psi = qo.StateVector([1,0])
>>> print(psi)
StateVector(2)
[ 1.+0.j  0.+0.j]
```

Alternatively there are some functions that create commonly used state vectors:

```
>>> psi = qo.basis(4,0)
>>> print(psi)
StateVector(4)
[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
>>> psi = qo.coherent(10, 0.5)
```

Composing systems can be done with the tensor product between two states. For this the operator ^ can be used.

```
>>> psi1 = qo.basis(2,0)
>>> psi2 = qo.basis(2,1)
>>> print(psi1 ^ psi2)
StateVector(2 x 2)
```

```
[[ 0.+0.j  1.+0.j]
 [ 0.+0.j  0.+0.j]]
```

Note: The ^ operator follows the built-in operator precedence. That means "*" and "+" have higher precedence!

1.3 Defining operators

Operators are represented by the `pyqo.Operator`. Like in the case of state vectors operators can be constructed directly from a list or tuple:

```
>>> A = qo.Operator([[1,0], [0,-1]])
>>> print(A)
Operator
2 -> 2
[[ 1.+0.j  0.+0.j]
 [ 0.+0.j -1.+0.j]]
```

Operators have some constraint on their shape - it has to be of the form $(n_1, n_2, .., n_N, n_1, n_2, .., n_N)$.

Many commonly used operators are already defined:

```
>>> print(qo.sigmax)
Operator
2 -> 2
[[ 0.+0.j  1.+0.j]
 [ 1.+0.j  0.+0.j]]
>>> print(qo.create(3))
Operator
3 -> 3
[[ 0.00000000+0.j  0.00000000+0.j  0.00000000+0.j]
 [ 1.00000000+0.j  0.00000000+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  1.41421356+0.j  0.00000000+0.j]]
```

Composing operators of different systems can be done in the following way:

```
>>> s_z = qo.sigmaz
>>> s_p = qo.sigmap
>>> print(s_z^s_p)
Operator
2 x 2 -> 2 x 2
[[[[ 0.+0.j  0.+0.j]
   [ 0.+0.j  0.+0.j]]

  [[ 1.+0.j  0.+0.j]
   [ 0.+0.j  0.+0.j]]]


 [[[ 0.+0.j  0.+0.j]
   [-0.+0.j -0.+0.j]]

  [[ 0.+0.j  0.+0.j]
   [-1.+0.j -0.+0.j]]]]
```

CHAPTER

TWO

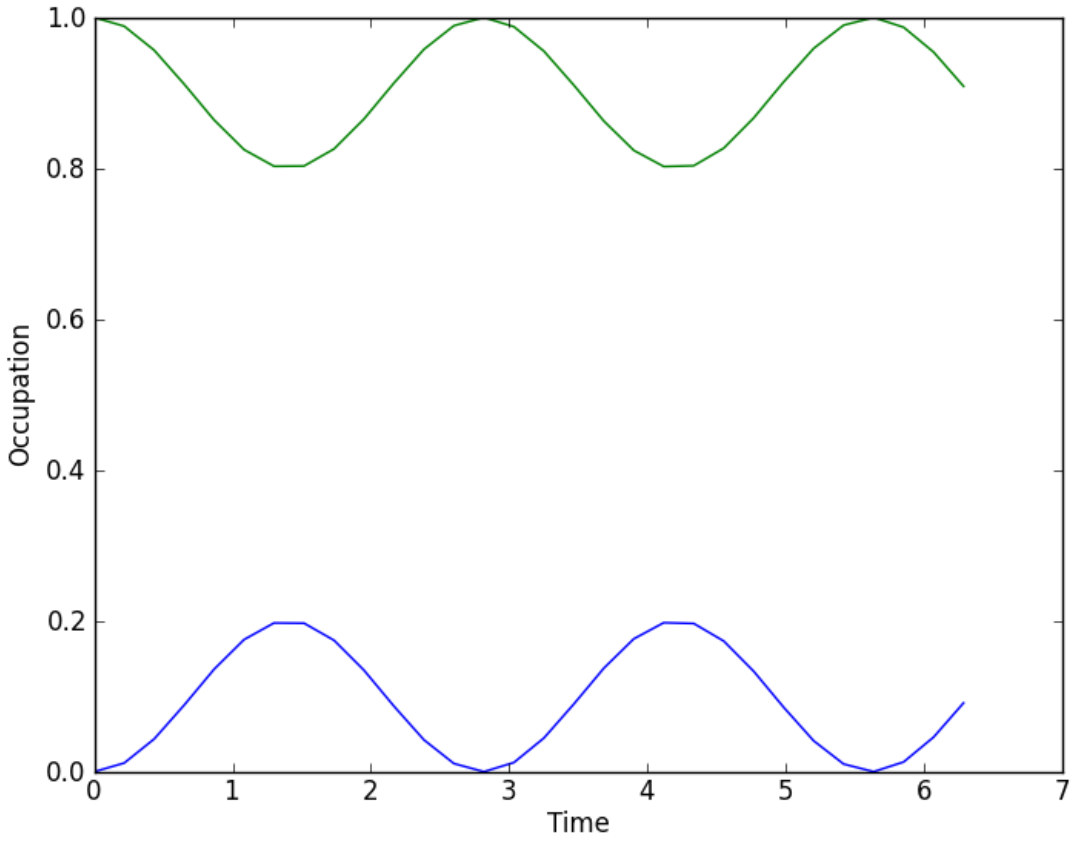
EXAMPLES

2.1 Rabi oscillation

2.1.1 Code

```
import imp
qo = imp.load_module("pyqo", *imp.find_module("pyqo", [".."]))
import numpy as np

Delta = 2
Omega = 1
phi = np.pi

H = 1./2*(- Delta * qo.sigmaz\
          + Omega * np.exp(1j*phi) * qo.sigmam\
          + Omega * np.exp(-1j*phi) * qo.sigmap)

psi_0 = qo.basis(2,0)

T = np.linspace(0,2*np.pi,30)
psi = qo.solve_ode(H, psi_0, T)

e0 = qo.expect(qo.sigmap*qo.sigmam, psi)
e1 = qo.expect(qo.sigmam*qo.sigmap, psi)

import pylab
pylab.plot(T, e0, T, e1)
pylab.xlabel("Time")
pylab.ylabel("Occupation")
pylab.show()
```

2.1.2 Output



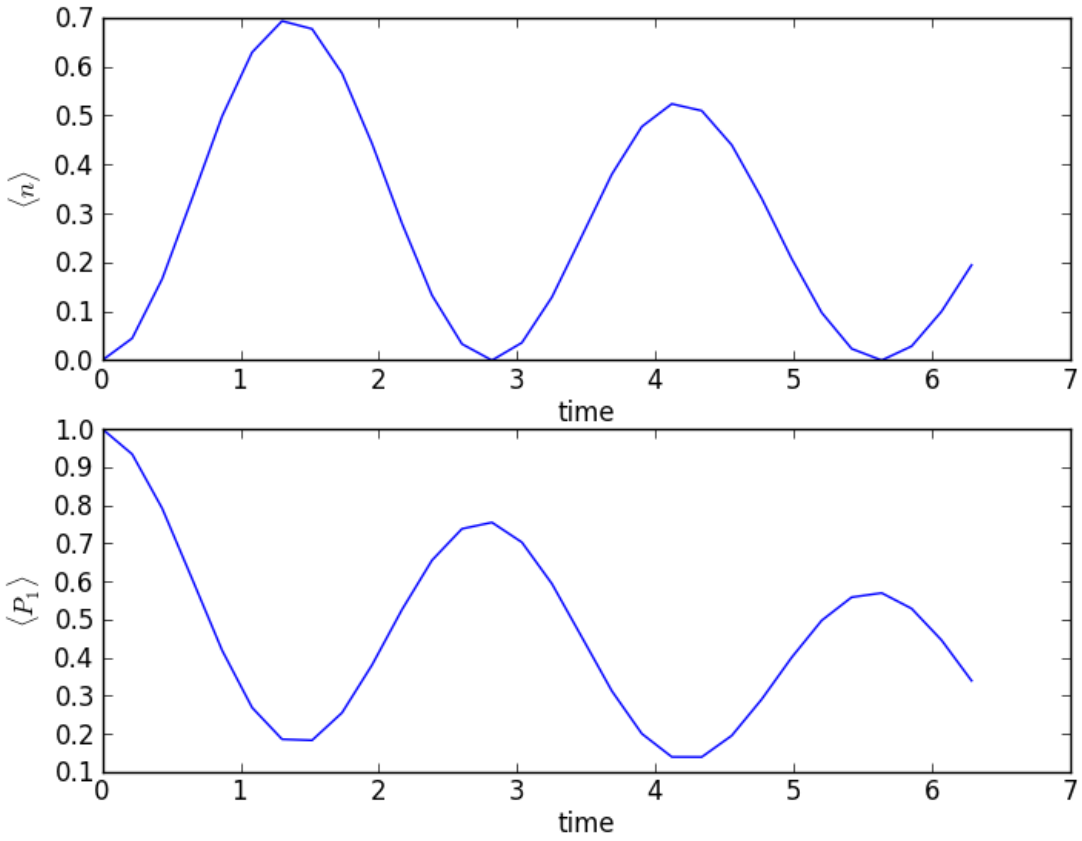
2.2 Jaynes-Cummings model

2.2.1 Code

```
import imp
qo = imp.load_module("pyqo", *imp.find_module("pyqo", [".."]))
import numpy as np

N = 10 # dimension of field Hilbert space
delta_c = 1
delta_a = 2
g = 1
gamma = 0.1
kappa = 0.1

# Field
id_f = qo.identity(N)
a = qo.destroy(N)
at = qo.create(N)
n = qo.number(N)

# Atom
```

```
id_a = qo.identity(2)

# Initial state
psi_0 = qo.basis(N,0) ^ qo.basis(2,1)

# Hamiltonian
H = delta_c*(at*a^id_a)\
    + delta_a*(id_f^qo.sigmap*qo.sigmam)\
    + g*(a^qo.sigmap) + g*(at^qo.sigmam)

# Solve Master equation
T = np.linspace(0, 2*np.pi, 30)
rho = qo.solve_ode(H, psi_0, T,
        [gamma**(1/2)*(id_f^qo.sigmam), kappa**(1/2)*(a^id_a)])

# Expectation values
n_exp = qo.expect(n^id_a, rho)
e_exp = qo.expect(id_f^qo.sigmap*qo.sigmam, rho)

# Q-function
x = np.linspace(-4,4,40)
y = np.linspace(-4,4,40)
X, Y = np.meshgrid(x,y)


# Visualization
import pylab
pylab.figure(1)
pylab.subplot(211)
pylab.xlabel("time")
pylab.ylabel(r"$\langle n \rangle$")
pylab.plot(T, n_exp)
pylab.subplot(212)
pylab.xlabel("time")
pylab.ylabel(r"$\langle P_1 \rangle$")
pylab.plot(T, e_exp)
pylab.show()

Q = []
for rho_t in rho:
    rho_f = qo.ptrace(rho_t,1)
    Q.append(np.abs(qo.qfunc(rho_f,X,Y)))

def qplot(fig,step):
    axes = fig.add_subplot(111)
    axes.clear()
    axes.imshow(Q[step])

qo.animate(len(rho), qplot)
```

2.2.2 Output



2.3 Single atom laser

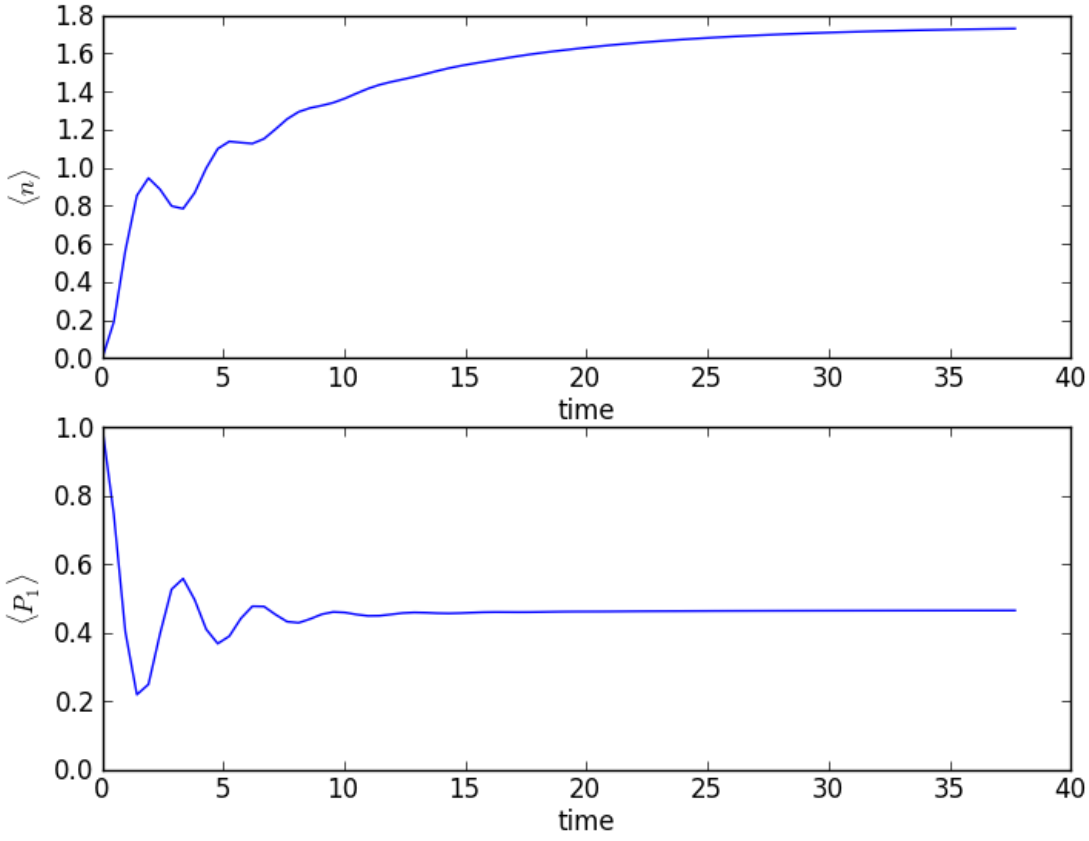
2.3.1 Code

```
import imp
qo = imp.load_module("pyqo", *imp.find_module("pyqo", [".."]))
import numpy as np

N = 10 # dimension of field Hilbert space
delta_c = 1
delta_a = 1
g = 1
gamma = 0.2
kappa = 0.1
R = 0.5

# Field
id_f = qo.identity(N)
a = qo.destroy(N)
at = qo.create(N)
n = qo.number(N)
```

```
# Atom
id_a = qo.identity(2)

# Initial state
psi_0 = qo.basis(N,0) ^ qo.basis(2,1)

# Hamiltonian
H = delta_c*(at*a^id_a) + delta_a*(id_f^qo.sigmap*qo.sigmam) + g*(a^qo.sigmap) + g*(at^qo.sigmam)

# Jump operators
j1 = gamma**(1./2)*(id_f^qo.sigmam)
j2 = kappa**(1./2)*(a^id_a)
j3 = R**(1./2)*(id_f^qo.sigmap)
J = [j1, j2, j3]

# Solve Master equation
T = np.linspace(0, 12*np.pi, 80)
rho = qo.solve_ode(H, psi_0, T, J)
#rho = qo.solve_es(H, psi_0, T, J)

# Expectation values
n_exp, e_exp = qo.expect((n^id_a, id_f^qo.sigmap*qo.sigmam), rho)

# Calculate Q-function and photon number distribution
x_min, x_max = -5, 5
y_min, y_max = -5, 5
x = np.linspace(x_min, x_max, 30)
y = np.linspace(y_min, y_max, 30)
X, Y = np.meshgrid(x,y)

Q = []
F = []
for rho_t in rho:
    rho_f = qo.ptrace(rho_t,1)
    Q.append(np.abs(qo.qfunc(rho_f,X,Y)))
    F.append(np.abs(np.diag(rho_f)))

# Visualization
import pylab
pylab.figure(1)
pylab.subplot(211)
pylab.xlabel("time")
pylab.ylabel(r"$\langle n \rangle$")
pylab.plot(T, n_exp)
pylab.ylim(ymin=0)
pylab.subplot(212)
pylab.xlabel("time")
pylab.ylabel(r"$\langle P_1 \rangle$")
pylab.ylim((0,1))
pylab.plot(T, e_exp)
pylab.show()

from scipy.misc import factorial

F_x = np.arange(0, N)
def fplot(fig, step):
    axes = fig.add_subplot(111)
    axes.clear()
```

```
    pylab.plot(F_x, F[step], "o")
    n_ = np.abs(n_exp[step])
    pylab.plot(F_x, np.exp(-n_)*n_**F_x/factorial(F_x))
qo.animate(len(rho), fplot)

def qplot(fig,step):
    axes = fig.add_subplot(111)
    axes.clear()
    axes.imshow(Q[step], interpolation='bilinear', origin='lower',
                extent=(x_min, x_max, y_min, y_max))

qo.animate(len(rho), qplot)
```

2.3.2 Output



1. *Rabi oscillation*
2. *Jaynes-Cummings model*
3. *Single atom laser*

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*