Source code for qutip.sesolve

```python
# This file is part of QuTiP: Quantum Toolbox in Python.
#
#    Copyright (c) 2011 and later, Paul D. Nation and Robert J. Johansson.
#    All rights reserved.
#
#    Redistribution and use in source and binary forms, with or without
#    modification, are permitted provided that the following conditions are
#    met:
#
#    1. Redistributions of source code must retain the above copyright notice,
#       this list of conditions and the following disclaimer.
#
#    2. Redistributions in binary form must reproduce the above copyright
#       notice, this list of conditions and the following disclaimer in the
#       documentation and/or other materials provided with the distribution.
#
#    3. Neither the name of the QuTiP: Quantum Toolbox in Python nor the names
#       of its contributors may be used to endorse or promote products derived
#       from this software without specific prior written permission.
#
#    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
#    "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
#    LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
#    PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
#    HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
#    SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
#    LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
#    DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
#    THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
#    (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
#    OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
###############################################################################
"""
This module provides solvers for the unitary Schrodinger equation.
"""

__all__ = ['sesolve']

import os
import types
from functools import partial
import numpy as np
import scipy.integrate
from scipy.linalg import norm

from qutip.qobj import Qobj, isket
from qutip.rhs_generate import rhs_generate
from qutip.solver import Result, Options, config, _solver_safety_check
from qutip.rhs_generate import _td_format_check, _td_wrap_array_str
from qutip.interpolate import Cubic_Spline
from qutip.settings import debug
from qutip.cy.spmatfuncs import (cy_expect_psi, cy_ode_rhs,
                                 cy_ode_psi_func_td,
                                 cy_ode_psi_func_td_with_state)
from qutip.cy.codegen import Codegen

from qutip.ui.progressbar import BaseProgressBar

if debug:
    import inspect


def sesolve(H, rho0, tlist, e_ops=[], args={}, options=None,                [docs]
            progress_bar=BaseProgressBar(),
            _safe_mode=True):
    """
    Schrodinger equation evolution of a state vector for a given Hamiltonian.

    Evolve the state vector or density matrix (`rho0`) using a given
    Hamiltonian (`H`), by integrating the set of ordinary differential
    equations that define the system.

    The output is either the state vector at arbitrary points in time
    (`tlist`), or the expectation values of the supplied operators
    (`e_ops`). If e_ops is a callback function, it is invoked for each
    time in `tlist` with time and the state as arguments, and the function
    does not use any return values.

    Parameters
    ----------

    H : :class:`qutip.qobj`
        system Hamiltonian, or a callback function for time-dependent
        Hamiltonians.

    rho0 : :class:`qutip.qobj`
        initial density matrix or state vector (ket).

    tlist : *list* / *array*
        list of times for :math:`t`.

    e_ops : list of :class:`qutip.qobj` / callback function single
        single operator or list of operators for which to evaluate
        expectation values.

    args : *dictionary*
        dictionary of parameters for time-dependent Hamiltonians and
        collapse operators.

    options : :class:`qutip.Qdeoptions`
        with options for the ODE solver.

    Returns
    -------

    output: :class:`qutip.solver`

        An instance of the class :class:`qutip.solver`, which contains either
        an *array* of expectation values for the times specified by `tlist`, or
        an *array* or state vectors or density matrices corresponding to the
        times in `tlist` [if `e_ops` is an empty list], or
        nothing if a callback function was given inplace of operators for
        which to calculate the expectation values.

    """

    if _safe_mode:
        _solver_safety_check(H, rho0, c_ops=[], e_ops=e_ops, args=args)
    if isinstance(e_ops, Qobj):
        e_ops = [e_ops]

    if isinstance(e_ops, dict):
        e_ops_dict = e_ops
        e_ops = [e for e in e_ops.values()]
    else:
        e_ops_dict = None

    # convert array based time-dependence to string format
    H, _, args = _td_wrap_array_str(H, [], args, tlist)
    # check for type (if any) of time-dependent inputs
    n_const, n_func, n_str = _td_format_check(H, [])

    if options is None:
        options = Options()

    if (not options.rhs_reuse) or (not config.tdfunc):
        # reset config time-dependence flags to default values
        config.reset()

    if n_func > 0:
        res = _sesolve_list_func_td(H, rho0, tlist, e_ops, args, options,
                                    progress_bar)

    elif n_str > 0:
        res = _sesolve_list_str_td(H, rho0, tlist, e_ops, args, options,
                                   progress_bar)

    elif isinstance(H, (types.FunctionType,
                        types.BuiltinFunctionType,
                        partial)):
        res = _sesolve_func_td(H, rho0, tlist, e_ops, args, options,
                               progress_bar)

    else:
        res = _sesolve_const(H, rho0, tlist, e_ops, args, options,
                             progress_bar)

    if e_ops_dict:
        res.expect = {e: res.expect[n]
                      for n, e in enumerate(e_ops_dict.keys())}

    return res



# -----------------------------------------------------------------------------
# A time-dependent unitary wavefunction equation on the list-function format
#
def _sesolve_list_func_td(H_list, psi0, tlist, e_ops, args, opt,
                          progress_bar):
    """
    Internal function for solving the master equation. See mesolve for usage.
    """

    if debug:
        print(inspect.stack()[0][3])

    #
    # check initial state
    #
    if not isket(psi0):
        raise TypeError("The unitary solver requires a ket as initial state")

    #
    # construct liouvillian in list-function format
    #
    L_list = []
    if not opt.rhs_with_state:
        constant_func = lambda x, y: 1.0
    else:
        constant_func = lambda x, y, z: 1.0

    # add all hamitonian terms to the lagrangian list
    for h_spec in H_list:

        if isinstance(h_spec, Qobj):
            h = h_spec
            h_coeff = constant_func

        elif isinstance(h_spec, list):
            h = h_spec[0]
            h_coeff = h_spec[1]

        else:
            raise TypeError("Incorrect specification of time-dependent " +
                            "Hamiltonian (expected callback function)")

        L_list.append([-1j * h.data, h_coeff])

    L_list_and_args = [L_list, args]

    #
    # setup integrator
    #
    initial_vector = psi0.full().ravel()
    if not opt.rhs_with_state:
        r = scipy.integrate.ode(psi_list_td)
    else:
        r = scipy.integrate.ode(psi_list_td_with_state)
    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])
    r.set_f_params(L_list_and_args)

    #
    # call generic ODE code
    #
    return _generic_ode_solve(r, psi0, tlist, e_ops, opt, progress_bar,
                              dims=psi0.dims)


#
# evaluate dpsi(t)/dt according to the master equation using the
# [Qobj, function] style time dependence API
#
def psi_list_td(t, psi, H_list_and_args):
    H_list = H_list_and_args[0]
    args = H_list_and_args[1]

    H = H_list[0][0] * H_list[0][1](t, args)
    for n in range(1, len(H_list)):
        #
        # args[n][0] = the sparse data for a Qobj in operator form
        # args[n][1] = function callback giving the coefficient
        #
        H = H + H_list[n][0] * H_list[n][1](t, args)

    return H * psi


def psi_list_td_with_state(t, psi, H_list_and_args):

    H_list = H_list_and_args[0]
    args = H_list_and_args[1]

    H = H_list[0][0] * H_list[0][1](t, psi, args)
    for n in range(1, len(H_list)):
        #
        # args[n][0] = the sparse data for a Qobj in operator form
        # args[n][1] = function callback giving the coefficient
        #
        H = H + H_list[n][0] * H_list[n][1](t, psi, args)

    return H * psi


# -----------------------------------------------------------------------------
# Wave function evolution using a ODE solver (unitary quantum evolution) using
# a constant Hamiltonian.
#
def _sesolve_const(H, psi0, tlist, e_ops, args, opt, progress_bar):
    """!
    Evolve the wave function using an ODE solver
    """
    if debug:
        print(inspect.stack()[0][3])

    if not isket(psi0):
        raise TypeError("psi0 must be a ket")

    #
    # setup integrator.
    #
    initial_vector = psi0.full().ravel()
    r = scipy.integrate.ode(cy_ode_rhs)
    L = -1.0j * H
    r.set_f_params(L.data.data, L.data.indices, L.data.indptr)  # cython RHS
    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])

    #
    # call generic ODE code
    #
    return _generic_ode_solve(r, psi0, tlist, e_ops, opt,
                              progress_bar, dims=psi0.dims)


#
# evaluate dpsi(t)/dt [not used. using cython function is being used instead]
#
def _ode_psi_func(t, psi, H):
    return H * psi


# -----------------------------------------------------------------------------
# A time-dependent disipative master equation on the list-string format for
# cython compilation
#
def _sesolve_list_str_td(H_list, psi0, tlist, e_ops, args, opt,
                         progress_bar):
    """
    Internal function for solving the master equation. See mesolve for usage.
    """

    if debug:
        print(inspect.stack()[0][3])

    #
    # check initial state: must be a density matrix
    #
    if not isket(psi0):
        raise TypeError("The unitary solver requires a ket as initial state")

    #
    # construct liouvillian
    #
    Ldata = []
    Linds = []
    Lptrs = []
    Lcoeff = []
    Lobj = []

    # loop over all hamiltonian terms, convert to superoperator form and
    # add the data of sparse matrix representation to h_coeff
    for h_spec in H_list:

        if isinstance(h_spec, Qobj):
            h = h_spec
            h_coeff = "1.0"

        elif isinstance(h_spec, list):
            h = h_spec[0]
            h_coeff = h_spec[1]

        else:
            raise TypeError("Incorrect specification of time-dependent " +
                            "Hamiltonian (expected string format)")

        L = -1j * h

        Ldata.append(L.data.data)
        Linds.append(L.data.indices)
        Lptrs.append(L.data.indptr)
        if isinstance(h_coeff, Cubic_Spline):
            Lobj.append(h_coeff.coeffs)
        Lcoeff.append(h_coeff)

    # the total number of liouvillian terms (hamiltonian terms +
    # collapse operators)
    n_L_terms = len(Ldata)

    #
    # setup ode args string: we expand the list Ldata, Linds and Lptrs into
    # and explicit list of parameters
    #
    string_list = []
    for k in range(n_L_terms):
        string_list.append("Ldata[%d], Linds[%d], Lptrs[%d]" % (k, k, k))
    # Add object terms to end of ode args string
    for k in range(len(Lobj)):
        string_list.append("Lobj[%d]" % k)

    for name, value in args.items():
        if isinstance(value, np.ndarray):
            string_list.append(name)
        else:
            string_list.append(str(value))
    parameter_string = ",".join(string_list)

    #
    # generate and compile new cython code if necessary
    #
    if not opt.rhs_reuse or config.tdfunc is None:
        if opt.rhs_filename is None:
            config.tdname = "rhs" + str(os.getpid()) + str(config.cgen_num)
        else:
            config.tdname = opt.rhs_filename
        cgen = Codegen(h_terms=n_L_terms, h_tdterms=Lcoeff, args=args,
                       config=config)
        cgen.generate(config.tdname + ".pyx")

        code = compile('from ' + config.tdname + ' import cy_td_ode_rhs',
                       '<string>', 'exec')
        exec(code, globals())
        config.tdfunc = cy_td_ode_rhs

    #
    # setup integrator
    #
    initial_vector = psi0.full().ravel()
    r = scipy.integrate.ode(config.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(' + parameter_string + ')',
                   '<string>', 'exec')

    exec(code, locals(), args)
```

```python
    #
    # call generic ODE code
    #
    return _generic_ode_solve(r, psi0, tlist, e_ops, opt, progress_bar,
                              dims=psi0.dims)


# -----------------------------------------------------------------------------
# Wave function evolution using a ODE solver (unitary quantum evolution), for
# time dependent hamiltonians
#
def _sesolve_list_td(H_func, psi0, tlist, e_ops, args, opt, progress_bar):
    """!
    Evolve the wave function using an ODE solver with time-dependent
    Hamiltonian.
    """

    if debug:
        print(inspect.stack()[0][3])

    if not isket(psi0):
        raise TypeError("psi0 must be a ket")

    #
    # configure time-dependent terms and setup ODE solver
    #
    if len(H_func) != 2:
        raise TypeError('Time-dependent Hamiltonian list must have two terms.')
    if (not isinstance(H_func[0], (list, np.ndarray))) or \
       (len(H_func[0]) <= 1):
        raise TypeError('Time-dependent Hamiltonians must be a list with two '
                        + 'or more terms')
    if (not isinstance(H_func[1], (list, np.ndarray))) or \
       (len(H_func[1]) != (len(H_func[0]) - 1)):
        raise TypeError('Time-dependent coefficients must be list with ' +
                        'length N-1 where N is the number of ' +
                        'Hamiltonian terms.')
    tflag = 1
    if opt.rhs_reuse and config.tdfunc is None:
        print("No previous time-dependent RHS found.")
        print("Generating one for you...")
        rhs_generate(H_func, args)
    lenh = len(H_func[0])
    if opt.tidy:
        H_func[0] = [(H_func[0][k]).tidyup() for k in range(lenh)]
    # create data arrays for time-dependent RHS function
    Hdata = [-1.0j * H_func[0][k].data.data for k in range(lenh)]
    Hinds = [H_func[0][k].data.indices for k in range(lenh)]
    Hptrs = [H_func[0][k].data.indptr for k in range(lenh)]
    # setup ode args string
    string = ""
    for k in range(lenh):
        string += ("Hdata[" + str(k) + "], Hinds[" + str(k) +
                   "], Hptrs[" + str(k) + "],")

    if args:
        td_consts = args.items()
        for elem in td_consts:
            string += str(elem[1])
            if elem != td_consts[-1]:
                string += (",")

    # run code generator
    if not opt.rhs_reuse or config.tdfunc is None:
        if opt.rhs_filename is None:
            config.tdname = "rhs" + str(os.getpid()) + str(config.cgen_num)
        else:
            config.tdname = opt.rhs_filename
        cgen = Codegen(h_terms=n_L_terms, h_tdterms=Lcoeff, args=args,
                       config=config)
        cgen.generate(config.tdname + ".pyx")

        code = compile('from ' + config.tdname + ' import cy_td_ode_rhs',
                       '<string>', 'exec')
        exec(code, globals())
        config.tdfunc = cy_td_ode_rhs
    #
    # setup integrator
    #
    initial_vector = psi0.full().ravel()
    r = scipy.integrate.ode(config.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)

    #
    # call generic ODE code
    #
    return _generic_ode_solve(r, psi0, tlist, e_ops, opt, progress_bar
                              , dims=psi0.dims)


# ------------------------------------------------------------------------------
# Wave function evolution using a ODE solver (unitary quantum evolution), for
# time dependent hamiltonians
#
def _sesolve_func_td(H_func, psi0, tlist, e_ops, args, opt, progress_bar):
    """!
    Evolve the wave function using an ODE solver with time-dependent
    Hamiltonian.
    """

    if debug:
        print(inspect.stack()[0][3])

    if not isket(psi0):
        raise TypeError("psi0 must be a ket")

    #
    # setup integrator
    #
    new_args = None

    if type(args) is dict:
        new_args = {}
        for key in args:
            if isinstance(args[key], Qobj):
                new_args[key] = args[key].data
            else:
                new_args[key] = args[key]

    elif type(args) is list or type(args) is tuple:
        new_args = []
        for arg in args:
            if isinstance(arg, Qobj):
                new_args.append(arg.data)
            else:
                new_args.append(arg)

        if type(args) is tuple:
            new_args = tuple(new_args)
    else:
        if isinstance(args, Qobj):
            new_args = args.data
        else:
            new_args = args

    initial_vector = psi0.full().ravel()

    if not opt.rhs_with_state:
        r = scipy.integrate.ode(cy_ode_psi_func_td)
    else:
        r = scipy.integrate.ode(cy_ode_psi_func_td_with_state)

    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])
    r.set_f_params(H_func, new_args)

    #
    # call generic ODE code
    #
    return _generic_ode_solve(r, psi0, tlist, e_ops, opt, progress_bar,
                              dims=psi0.dims)


#
# evaluate dpsi(t)/dt for time-dependent hamiltonian
#
def _ode_psi_func_td(t, psi, H_func, args):
    H = H_func(t, args)
    return -1j * (H * psi)


def _ode_psi_func_td_with_state(t, psi, H_func, args):
    H = H_func(t, psi, args)
    return -1j * (H * psi)


# -------------------------------------------------------------------------------
# Solve an ODE which solver parameters already setup (r). Calculate the
# required expectation values or invoke callback function at each time step.
#
def _generic_ode_solve(r, psi0, tlist, e_ops, opt, progress_bar, dims=None):
    """
    Internal function for solving ODEs.
    """
    if opt.normalize_output:
        state_norm_func = norm
    else:
        state_norm_func = None


    #
    # prepare output array
    #
    n_tsteps = len(tlist)
    output = Result()
    output.solver = "sesolve"
    output.times = tlist

    if opt.store_states:
        output.states = []

    if isinstance(e_ops, types.FunctionType):
        n_expt_op = 0
        expt_callback = True

    elif isinstance(e_ops, list):

        n_expt_op = len(e_ops)
        expt_callback = False

        if n_expt_op == 0:
            # fallback on storing states
            output.states = []
            opt.store_states = True
        else:
            output.expect = []
            output.num_expect = n_expt_op
            for op in e_ops:
                if op.isherm:
                    output.expect.append(np.zeros(n_tsteps))
                else:
                    output.expect.append(np.zeros(n_tsteps, dtype=complex))
    else:
        raise TypeError("Expectation parameter must be a list or a function")

    #
    # start evolution
    #
    progress_bar.start(n_tsteps)

    dt = np.diff(tlist)
    for t_idx, t in enumerate(tlist):
        progress_bar.update(t_idx)

        if not r.successful():
            raise Exception("ODE integration error: Try to increase "
                            "the allowed number of substeps by increasing "
                            "the nsteps parameter in the Options class.")

        if state_norm_func:
            data = r.y / state_norm_func(r.y)
            r.set_initial_value(data, r.t)

        if opt.store_states:
            output.states.append(Qobj(r.y, dims=dims))

        if expt_callback:
            # use callback method
            e_ops(t, Qobj(r.y, dims=psi0.dims))

        for m in range(n_expt_op):
            output.expect[m][t_idx] = cy_expect_psi(e_ops[m].data,
                                                    r.y, e_ops[m].isherm)

        if t_idx < n_tsteps - 1:
            r.integrate(r.t + dt[t_idx])

    progress_bar.finished()

    if not opt.rhs_reuse and config.tdname is not None:
        try:
            os.remove(config.tdname + ".pyx")
        except:
            pass

    if opt.store_final_state:
        output.final_state = Qobj(r.y, dims=dims)

    return output
```