DM_solver Documentation

Release 1.0

Stephan Philips

May 09, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

This is a simple program that solves the time evolution of a density matrix, given a time dependent Hamiltonian. I tried to add the most convenient functions that you will find in experiments (pulse shapes, pink/white noise), but feel free to extend this. To speed up the execution of the code, it it written in c++ (decent multi-threading and defined data types). Not to worry though! The code is cythonized, which means that you can fully access the library from python.

This modules solves the Von Neumann equation:

$$\rho(t + \Delta t) = U^{\dagger}\rho(t)U$$

Where Δt will be determined by the number of points calculated in the simulation. U is given by

$$U = e^{iH\Delta t}$$

Where H is the hamiltonian at time t. This module contains a toolbox that allows to easily make H time dependent.

CHAPTER

TWO

INSTALLATION

2.1 Requirements

Before you can compile the library, you will need to install the following dependencies:

- Armadillo
- Openblas (Note that you can also use INTEL MKL if you like)
- Open MPI
- GOMP
- DSP lib (Already included, no need to install)
- cython (install with pip)
- matplotlib (install with pip)

2.2 Installation

Compile and install all the c++ libraries as given in the requirements. In Linux you can install most of them probably with your package manager. On Windows I can imagine that the installation might be mode cumbersome.

Once you installed the dependencies, you can compile the program using the python interpreter:

```
python setup.py build_ext --inplace
```

You can than import the module by typing:

```
import sys
sys.path.append('folder/whereever/you/saved/the/setup/file')
import DM_solver
```

CHAPTER

THREE

USER GUIDE

The main idea of the module is to make the solving of time dependent Hamiltonian easy for typical pulse schemes used in the lab. In a python script you will have to make your own class that generates the Hamiltonian and signals. An example where this is done can be found in the repository (example folder). In the following, the main functionality that you can use to generate and preform operations with/on your Hamiltonians is explained.

The solver is initiated by typing:

```
import DM_solver
my_solver_object = DM_solver.VonNeumann(N)
```

Where N is the size of the Hilbert space. This function makes call to the c library creates the c object that does the solving. The next step is to define the Hamiltonian. The Hamiltonian will be spitted up in a so called time dependent and time independent part.

$H = H_0 + \sum_i H_1^i(t)$

The next section will describe how to create these elements. In this solver, $\hbar$ is set to 1.

3.1 Time independent part

The time independent part is defined as a static part of the Hamiltonian (H_0). These elements should not have any time dependency. You can create this part by typing

```
my_numpy_hamiltonian = np.qeye(5, dtype=np.complex)
# This can be any matrix, as long as it is complex.
my_solver_object.add_H0(my_numpy_hamiltonian)
```

Note that the matrix must be a 128 bit complex matrix, otherwise this will result in a type error. Numpy will only do this by default if you matrix contains something complex. Note that all methods expect complex arguments except when it does not make sense (e.g. a frequency will be a `double`). When in doubt, you can always look in the cython source file (python_wrapper.pyx). If you add multiple elements (`add_H0()`) they will just be summed.

3.2 Time dependent part

3.2.1 Add a list of points (not the recommended method).

The input will represent the following function $A(t)H_{input}$. Where $A(t)$ is defined by the given list. The number of points must be exactly the same as the number of points used in simulation. The input matrix H_{input}, is the matrix that is multiplied with the points $A(t)$ (e.g. $H_{input} = I \otimes X$). The code needed to input such a matrix can be given by:

```
amplitude_list = np.linspace(0,1e9,10000,dtype=np.complex)
my_solver_object.add_H1_list(input_matrix, amplitude_list)
```

Note that this method is only added for completeness and not recommended for general usage (+ I did not test this).

3.2.2 AWG pulses

AWG pulses can be constructed by giving a Numpy array that contains times with voltages. Note that in the following explanation I will quickly show some code to plot your pulse, later on you can input this pulse in to the Hamiltonian. An example could be

```
my_pulse = [
[1e-8,0],
[2e-8,1],
[3e-8,1],
[3e-8,0],
[5e-8,0],
[5e-8,2],
[7e-8,2],
[7e-8,0],
]
```

This will give a pulse with sharp edges (unlike the pulse generated by an AWG). Therefore we will want to apply some filtering (e.g. a first order Bessel IIR filter), this can be done like

```
example_pulse = me.test_pulse()
example_pulse.init(np.array(my_pulse))
example_pulse.plot_pulse(0,100e-9,500)

#Note for the filtering, now only Butterworth and Bessel are supported (this can be␣
↪extended if needed) (up to 10th order).
# overwrite pulse (do Butterworth filter of the order 1 with a cutoff at 340MHz)
example_pulse.init(np.array(my_pulse), ['butt', 1 , 340e6])
example_pulse.plot_pulse(0,100e-9,500)

# overwrite pulse (cascade multiple filters (Butterworth and Bessel))
example_pulse.init(np.array(my_pulse), [['butt', 1 , 340e6], ['bessel', 2 , 380e6]])
example_pulse.plot_pulse(0,100e-9,500)

# show plot of pulses
plt.show()
```

This procedure generally does not need to be done, but it is handy to get a feeling for what kind of filter settings you want. To make a pulse in the Hamiltonian, you can just write:

```
my_filter = [['butt', 1 , 340e6], ['bessel', 2 , 380e6]]
my_solver_object.add_H1_AWG(my_pulse, input_matrix, my_filter )
```

3.2.3 Microwaves

To add microwaves there are basically two approaches. For a block pulse you can use:

```
my_solver_object.add_H1_MW_RF(input_matrix, rabi_f*np.pi, phase, frequency, start,␣
↪stop)
#rabi_f, phase, frequency, start and stop are doubles here.
```

Where the `input_matrix` is your rotation matrix. `rabi_f` is the amplitude of the drive. `phase` is the phase of the incoming signal. `frequency` is the frequency (can be 0 in a rotating frame). Note that the input is not a cosine, but $e^{i\omega t}$. This was done since it is the most convenient for making rotating frames. You can still make a cos out of the exponential functions.

The other way of adding microwaves is by using the microwave object. In this object you can for example specify pulse shaped. At the moment only Gaussians are supported (note that is is quite straightforward to add new pulses in the c code). In the following an example piece of code is given. Here a pulse is send, where you put over it a Gaussian envelope. The envelope has as center the middle of the pulse.

```
# first define the general properties of the microwave. Here in an example of 2␣
↪qubits.
mw_obj_1 = DM_solver.microwave_RWA()
mw_obj_1.init(rabi_f*np.pi), phase, freq_RF-f_qubit, t_start, t_stop)
mw_obj_1.add_gauss_mod(sigma_Gauss) # sigma is here the standard deviation of the␣
↪Gaussian distribution
mw_obj_2 = DM_solver.microwave_RWA()
mw_obj_2.init(rabi_f*np.pi), phase, freq_RF-f_qubit2, t_start, t_stop)
mw_obj_2.add_gauss_mod(sigma_Gauss)
# Couple object to the active matrix element.
my_solver_object.add_H1_MW_RF_obj(H_mw_qubit_1, mw_obj_1)
my_solver_object.add_H1_MW_RF_obj(H_mw_qubit_2, mw_obj_2)
```

3.2.4 Global time dependency

When you have some parts of you Hamiltonian that continuously oscillates, you can add a global time dependency. This will be added latest to you Hamiltonian when constructing it. This means it will also be added on top of the noise you add. This can be a handy feature when you have a time dependency due to a transformed Hamiltonian.

In the following a example is given of how to add a time dependency to a parameter. The dependency is given by:

$parameter(t) * e^{i2\pi f}$

Where f is the frequency of the oscillations. Example:

```
# This adds a time depend parameter to location (1,4) in the matrix.
# Note that the matrix is by nature hermitian, so you do not have to specify (4,1)
# Make sure the data types are set as here.
locations_1 = np.array([[1,4],[1,5]],dtype=np.int32)
locations_2 = np.array([[2,4],[2,5]],dtype=np.int32)
my_solver_object.add_cexp_time_dep(locations_1, frequency_1)
my_solver_object.add_cexp_time_dep(locations_2, frequency_2)
```

Note that you only need to add to top part of the matrix, the hermitian conjugate of the parameter is taken by default.

3.3 Noise functions

In the noise department we have tree flavors, static noise and white noise and pink noise.

3.3.1 Static noise

Static noise can be added in two ways. The most simple way is the following

```
self.solver_obj.add_static_gauss_noise(my_noisy_hamiltonian, T2_qubit)
# where my_noisy_hamiltonian is a matrix that is multiplied with the amount of noise
# e.g. for two qubits this could be S_z x I
```

In this case magnetic noise will be added onto the first qubit. Note if you add more objects, the noise is uncorrelated.

The second way to add noise (for more complex cases) goes as following (with the possibility of making correlations):

```
# init object
charge_noise = DM_solver.noise_py()
# iniy gausian noise
charge_noise.init_gauss(np.zeros([6,6],dtype=np.complex),T2)
# add parameter dependency (arguments: Hamiltonian -- location -- parm_dep_matrix␣
↪(complex))
charge_noise.add_param_matrix_dep(2.4*H_B_field1 + 0.78*H_B_field2 + H_B_field1*H_B_
↪field2 , (4, 4), np.array([[0,1/detuningE],[0,chargingE]], dtype=np.complex))
charge_noise.add_param_matrix_dep(0.45*H_B_field1 + 0.93*H_B_field2, (4, 4), np.
↪array([[0,-1/detuningE],[0,chargingE-detuningE]], dtype=np.complex))

my_solver_object.add_noise_obj(charge_noise)
```

The main difference between the first method is that you can add a parameter dependency. This basically means that you can give a location (here e.g. `(4,4)`) in your Hamiltonian and depending on its value, you will add a certain amount of noise. To construct the dependency, you can enter a formula in the following form:

$y = \sum_i a_i (x_i - x_0)^i$

The input matrix is given in an array of the following shape

$$\begin{pmatrix} x_0 & x_1 & ... & x_n \\ a_0 & a_1 & ... & a_n \end{pmatrix}$$

Note that this matrix has to have the type complex.

3.3.2 White noise

Can be made using:

```
charge_noise = DM_solver.noise_py()
# init Gaussian noise
charge_noise.init_white(noise_hamiltonian (e.g. S_z x I), amplitude)
# dependencies can also be added here.
```

The rest of the procedure is again exactly the same as for static noise.

3.3.3 Pink noise

The $\frac{1}{f^\alpha}$ noise is generated by sampling from a Gaussian distribution (generates white noise). This data is than transformed with a FFT. Then the amplitudes of the frequency components are adjusted and a reverse FFT is taken. Note that this inefficient for long sequences. A FIR/IIR implementation would make sense here.

The noise can be added by:

```
charge_noise = DM_solver.noise_py()
# init Gaussian noise
charge_noise.init_pink(noise_hamiltonian (e.g. S_z x I), amplitude, alpha)
```

Rest of the procedure is again exactly the same as for the others.

3.3.4 Samples

Where running noisy simulations you will need to average these simulations. To do this, you can specify the number of simulations (samples) you want to average. This can be done by calling the following function (the default value is 1).

```
self.solver_obj.set_number_of_evalutions(number)
```

Where the number must be an integer.

3.4 Running the solver

The solver will start whenever you call the following:

```
my_solver_object.calculate_evolution(my_init_densitymatrix, t_start, t_stop,␣
↪numberofsteps)
```

3.5 Getting your results

To get the unitary representing your operation, you can type:

```
my_solver_object.get_unitary()
time_points_sim = my_solver_oject.times
```

To get expectation values for a certain property you can type:

```
my_solver_object.return_expectation_values(operators)
```

where operators is a matrix of dimension 3, meaning a list of you operators (see example provided).

To plot the expectation values, you can use:

```
my_solver_object.plot_expectation(operators, label, figure_number)
plt.show()
```

For more info see source code and the example

3.6 Clearing memory

If you run many loops, you might see that python's garbage collector does not automatically delete the c object created that solves the Von Neumann equation. The memory can be freed up by calling:

```
my_solver_object.clear()
```