Tomography Documentation

Release 0.1

Tjeerd Fokkens, Andreas Fognini, Val Zwiller

Aug 17, 2016

Contents:

CHAPTER

ONE

INTRODUCTION

This implementation of the density matrix reconstruction follows closely the method described in D. F. V. James et al. Phys. Rev. A, 64, 052312 (2001). We tried to keep the naming in the code as similar as possible to that reference.

Measurement

To reconstruct the density matrix of a two qubit photonic system we need to perform 16 coincidence measurements between the two qubits in 16 different polarization projections. A valid basis set for the reconstruction is for example:

```
basis =  ["HH","HV","VV","VH","HD","HR","VD","VR","DH","DV","DD","DR","RH","RV","RD",
↪"RR"]
```

In this example the first coincidence measurement is performed for the basis element "HH". This means that for both photons, the horizontal component of the polarization is measured.

In the second coincidence measurement, "HV", we measure the horizontal component of the first photon, and the vertical polarization component of the second one.

Usage of the library

Once all the coincidence counts have been collected the density matrix describing the quantum system can be reconstructed.

```
import numpy as np
from Tomography import DensityMatrix

round_digits = 2
dm = DensityMatrix(basis= ["HH","HV","VV","VH","RH","RV","DV","DH","DR","DD","RD","HD
↪","VD","VL","HL","RL"])
```

At first, the DensityMatrix class is initialized with the measurement basis. The measured correlation counts in this basis set is given to the function rho which calculates the density matrix.

```
#Compute the raw density matrix, data from: D. F. V. James et al. Phys. Rev. A, 64,␣
↪052312 (2001).
cnts = np.array([34749, 324, 35805, 444, 16324, 17521, 13441, 16901, 17932, 32028,␣
↪15132, 17238, 13171, 17170, 16722, 33586])
rho  = dm.rho(cnts)
```

However, due to measurement imperfections this matrix is not necessarily positive semidefinite. To circumvent this problem a maximum likelihood estimation as described in the aforementioned paper is implemented. Please note

that we initialize the t parameters in our implementation with only ones. In this way, the algorithm can also handle states like HH, since the Cholesky decomposition can effectively be computed. We get the reconstructed positive semidefinite density matrix by evoking:

```
rho_recon=dm.rho_max_likelihood(rho, cnts)
```

The library can also compute some quantum measures on the density matrix like the concurrence:

```
concurrence=dm.concurrence(rho_recon)
```

or the fidelity:

```
f=dm.fidelity_max(rho_recon)
```

The fidelity is calculated here to a maximally entangled state. We used the algorithm described in http://dx.doi.org/10.1103/PhysRevA.66.022307.

In quantum tomography not only the density matrix is of interest but also the pure state which most likely characterizes the system. This is only reasonable if the density matrix reconstructed is already close to a pure state, i.e. its fidelty or concurrence is close to unity.

The following code block is an example how to reconstruct a pure state with this library:

```
closest_state_basis =["HH","HV","VH","VV"]
closest_state = dm.find_closest_pure_state(rho_recon, basis=closest_state_basis)

s = str()
for i in range(3):
        s = s + "\t"+ str(closest_state[i]) + "\t|"+closest_state_basis[i] + "> + \n"

s = s + "\t" + str(closest_state[3]) + "\t|" + closest_state_basis[3] + ">"

print("Closest State: \n" + s + "\n")
```

The density matrix from any pure state can also easily be constructed. For example from the following Bell state: $\frac{1}{\sqrt{2}}(|HH\rangle + i|VV\rangle)$.

```
HH =dm.state("HH")
VV =dm.state("VV")

print(dm.rho_state(state=1/np.sqrt(2)*(HH+1j*VV)))
```

Error estimation

The error estimation is performed based on a Monte Carlo simulation. Each correlation count is assumed to be subjected to counting statistics. Thus, the measured number N of correlation counts will be replaced in each step of the simulation with a draw from a normal distribution with standard deviation $\sigma = \sqrt{N}$ and mean $\mu = N$. In each simulation step a new density matrix is calculated. Based on this set of simulated density matrices the standard deviation can be computed to estimate the error.

To get the error of the above examples do:

```
import numpy as np
from Tomography import Errorize
round_digits = 2
```

```
basis= ["HH", "HV","VV","VH","RH","RV","DV","DH","DR","DD","RD","HD","VD","VL","HL",
↪"RL"]
cnts = np.array([34749, 324, 35805, 444, 16324, 17521, 13441, 16901, 17932, 32028,␣
↪15132, 17238, 13171, 17170, 16722, 33586])
#Data from: D. F. V. James et al. Phys. Rev. A, 64, 052312 (2001).

err = Errorize(basis = basis, cnts = cnts)
err.multiprocessing_simulate(n_cycles_per_core  = 10, nbr_of_cores = 2)

rho_err = err.rho_max_likelihood()

print("Uncertainty of rho: \n" + str(np.around(rho_err, decimals =round_digits)) + "\n
↪")

#Uncertainty of fidelity and concurrence estimates
fid_err=err.fidelity_max()
con_err=err.concurrence()

print("fid_err: \n" + str(fid_err) + "\n")
print("con_err: \n" + str(con_err) + "\n")
```

CHAPTER

TWO

INSTALLATION

Before you start the installation of the library it is advised that you have installed numpy (numpy.org) and scipy (SciPy.org) packages already.

To install the library open a terminal and navigate into the Tomography folder. Then install it either by:

```
python install setup.py
```

Or by:

```
python3 install setup.py
```

Depending on your python installation. Please note that this library needs python3.x .

You can test if the installation worked by importing the library:

```
import Tomography
```

If you not getting an error message, everything works.

CHAPTER

THREE

MODULES DESCRIPTION

In the following the Tomography module is described. It consists of two classes:

```
* DensityMatrix
* Errorize
```

The DensityMatrix class is used to calculate the density matrix, some quantum measures, and the pure state closest to the found density matrix. The Errorize class is used to compute uncertainties of the values computed in the DensityMatrix class by means of a Monte Carlo simulation.

Tomography module

class Tomography.**DensityMatrix**(*basis*)

Bases: object

Computes the density matrix for an optical two qubit system. The measurements are performed with only one detector for each qubit. The code is programmed along the procedure described in D. F. V. James et al. Phys. Rev. A, 64, 052312 (2001).

The measurements need to be performed in horizontal (H) or vertical (V), circular right (R) or left (L), and diagonal (D) or antidiagonal (A) projections.

We use the following vector representation:

- $H = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
- $V = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
- $R = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -i \end{pmatrix}$
- $L = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ i \end{pmatrix}$
- $D = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- $A = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$

Parameters **basis** (*array*) – An array of basis elements in which the measurement is performed.

basis_str_to_object(*pol='H'*)
Relate string of basis element to Stokes vector.

Parameters **pol** (*str*) – String of the measurement basis. Valid elements H, V, R, L, D, A.

Returns Stokes vector of the polarization specified by H, V, R, L, D, A.

Return type numpy array

Example:

```
pol = "A"
```

concurrence(*rho*)
Compute the concurrence of the density matrix.

Parameters **rho** (*numpy_array*) – Density matrix

Returns The concurrence, see https://en.wikipedia.org/wiki/Concurrence_(quantum_computing).

Return type complex

construct_b_matrix(*PSI*, *GAMMA*)
Construct B matrix as in D. F. V. James et al. Phys. Rev. A, 64, 052312 (2001).

Parameters

- **PSI** (*array*) – ψ_ν vector with $\nu = 1, ..., 16$, computed in __init__
- **GAMMA** (*array*) – Γ matrices, computed in __init__

Returns $B_{\nu,\mu} = \langle\psi_\nu|\Gamma_\mu|\psi_\nu\rangle$

Return type numpy array

entropy_neumann(*rho*)
Compute the von Neumann entropy of the density matrix.

Parameters **rho** (*numpy_array*) – Density matrix

Returns The von Neumann entropy of the density matrix. $S = -\sum_j m_j \ln(m_j)$, where m_j denotes the eigenvalues of rho.

Return type complex

fidelity(*m*, *n*)
Compute the fidelity between the density matrces m and n.

Parameters

- **m** (*numpy_array*) – Density matrix
- **n** (*numpy_array*) – Density matrix

Returns The fideltiy between m and n ($\mathrm{Tr}(\sqrt{\sqrt{m}n\sqrt{m}})^2$).

Return type complex

fidelity_max(*rho*)
Compute the maximal fidelity of rho to a maximally entangled state.

Parameters **rho** (*numpy_array*) – Density matrix

Returns

The maximal fidelity of rho (ρ) to a maximally entangled state. $F(\rho) = \frac{1+\lambda_1+\lambda_2-\mathrm{sgn}(\det(\mathrm{R}))\lambda_3}{4}$, where

$R_{i,j} = \mathrm{Tr}(\sigma_i \otimes \sigma_j)$, with $\sigma_i, i = 1, 2, 3$ the Pauli matrices and $\lambda_i, i = 1, 2, 3$ the ordered singular values of R.

Note, the maximally entangled state is not computed. Algorithm from: http://dx.doi.org/10.1103/PhysRevA.66.022307

Return type complex

find_closest_pure_state (*rho, basis=['HH', 'HV', 'VH', 'VV']*)
Finds the closest pure state to the density matrix rho in the given basis.

Parameters

- **rho** (*numpy_array*) – density matrix
- **basis** (*array*) – The basis in which the state is described

Returns state vector describing the closest pure state. By convention the first vector element has vanishing complex component.

Return type numpy array

fun (*t*, *NormFactor*)
Maximum likelihood function to be minimized.

Parameters

- **t** (*numpy_array*) – t values.
- **NormFactor** (*float*) – Normalization factor.

Returns Function value. See for further information D. F. V. James et al. Phys. Rev. A, 64, 052312 (2001).

Return type numpy float

opt_pure_state (*coeff_array*, *rho*, *basis*)
Helper function for *self.find_closest_pure_state*.

Parameters

- **coeff_array** (*numpy_array*) – Coefficient array, to be optimized
- **rho** (*numpy_array*) – Density matrix
- **basis** – The basis state from which the pure state is constructed, e.g. ["HH", "HV", "VH", "VV"]

Returns 1-fidelity, such that the minimizing function finds the maximum of the fidelity.

Return type complex

purity (*rho*)
Compute the purity of the density matrix.

Parameters **rho** (*numpy_array*) – Density matrix

Returns The density matrix's purity $\mathrm{Tr}\rho^2$

Return type complex

rho (*correlation_counts*)
Compute the density matrix from measured correlation counts.

Parameters **correlation_counts** (*array*) – An array containing the correlation counts sorted according to the elements in self.basis.

Returns The density matrix.

Return type numpy array

Example:

```
correlation_counts = np.array([34749, 324, 35805, 444, 16324, 17521,␣
↪13441, 16901, 17932, 32028, 15132, 17238, 13171, 17170, 16722, 33586])
basis = ["HH","HV","VV","VH","RH","RV","DV","DH","DR","DD","RD","HD","VD",
↪"VL","HL","RL"]
```

Data from: D. F. V. James et al. Phys. Rev. A, 64, 052312 (2001).

rho_max_likelihood(*rho*, *corr_counts*)

Compute the density matrix based on the maximum likelihood approach.

Parameters

- **rho** (*numpy_array*) – Density matrix estimated from the measured correlation counts. Does not need to be physical, i.e. does not need to be postive semidefinite.
- **corr_counts** (*numpy_array*) – Measured correlation counts corresponding to the basis specified in __init__.

Returns Density matrix which is positive semidefinite.

Return type numpy array

rho_phys(*t*)

Positive semidefinite matrix based on t values.

Parameters **t** (*numpy_array*) – tvalues

Returns A positive semidefinite matrix which is an estimation of the actual density matrix.

Return type numpy matrix

rho_state(*state*)

Compute the density matrix of a pure state. The state is described either by linear superpositions of *self.state()* or ψ_ν tensor elements.

Parameters **state** – The state expressed as a linear combination of state tensor elements.

Returns The corresponding density matrix.

Example: If the basis in __init__(basis) was chosen as:

```
basis = ['HH', 'HV','VV','VH','RH','RV','DV','DH','DR','DD','RD','HD','VD
↪','VL','HL','RL']
```

The Bell state: $\frac{1}{\sqrt{2}}(|HH\rangle + i|VV\rangle)$

is described in python code with above basis as

```
HH=self.state("HH")
VV=self.state("VV")

state=1/sqrt(2)*(HH+1j*VV)
```

or as:

```
state=1/sqrt(2)*(self.PSI[0]+1j*self.PSI[2])
```

rho_state_optimized(*state*)

Compute the density matrix of a pure state based on the maximum likelihood approach. Aim: To test the maximum likelihood function. The state is described by linear superpositions of *self.state()* or ψ_ν tensor elements.

Parameters **state** – The state expressed as a linear combination of state tensor elements.

Returns The density matrix computed by the maximum likelihood approach.

Return type numpy array

Example: If the basis in __init__(basis) was chosen as:

```
basis = ['HH', 'HV','VV','VH','RH','RV','DV','DH','DR','DD','RD','HD','VD
↪','VL','HL','RL']
```

The Bell state: $\frac{1}{\sqrt{2}}(|HH\rangle + i|VV\rangle)$

is described in python code with above basis as

```
HH=self.state("HH")
VV=self.state("VV")

state=1/sqrt(2)*(HH+1j*VV)

dm=DensityMatrix(basis)
dm.rho_state_optimized(state)
```

or as:

```
state = 1/sqrt(2)*(self.PSI[0]+1j*self.PSI[2])

dm = DensityMatrix(basis)
dm.rho_state_optimized(state)
```

state(*string*)

Compute state from string.

Parameters **state** – Two letter string.

Returns State vector.

Example:

```
state = dm.state("HH")
```

will be: $state = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix}$

test_gamma(*gamma*)

Test if $\Gamma_i, i = 1, ..., 16$ matrices are properly defined.

Parameters **GAMMA** (*array*) – Gamma matrices.

Test for:

$\mathrm{Tr}(\Gamma_i\Gamma_j) = \delta_{i,j}$

Returns True if equation fullfilled for all gamma matrices, False otherwise.

Return type bool

class Tomography.**Errorize**(*basis*, *cnts*)
Bases: *Tomography.DensityMatrix*

Compute +- uncertainty of the density matrix. A Monte Carlo simulation is performed based on counting statistics.

Parameters

- **basis** (*array*) – Basis of measurements
- **cnts** (*array*) – Correlation counts of the measurements

collect_results(*result*)
Helper function for multicore processing.

complex_std_dev(*matrices*)
Compute the standard deviation for the real and complex part of matrices separately.

Parameters **matrices** (*numpy_array*) – An array filled with matrices.

Returns Standard deviation of the real and complex part for every matrix element.

Return type complex

concurrence()
Compute the standard deviation of the concurrence of density matrix.

Returns Its standard deviation.

fidelity_max()
Compute the standard deviation of the maximal fidelity of ρ to a maximally entangled state.

Returns Its standard deviation.

Note, the maximally entangled state is not computed. Function from: http://dx.doi.org/10.1103/PhysRevA.66.022307

multiprocessing_simulate(*n_cycles_per_core=10*, *nbr_of_cores=8*)
Perform Monte Carlo simulation parallel on several CPU cores. Each core will call function self.sim().

Parameters

- **n_cycles_per_core** (*float*) – Number of simulations per core.
- **nbr_of_cores** (*float*) – Number of CPUs

Returns

self.rhos, self.rhosrec

self.rhos array with raw density matrices and

self.rhosrec array with maximum likelihood approximated matrices.

Note: 'rhosrec' stands for rho reconstructed.

rtype numpy matrices

purity()
Compute the standard deviation of the density matrix's purity.

Returns Its standard deviation.

rho()
Compute the standard deviation of the density matrix.

Returns Its standard deviation.

rho_max_likelihood()
Compute the standard deviation of the density matrix reconstructed by the maximum likelihood method.

Returns Its standard deviation.

sim(*n_cycles_per_core*, *basis*)
Perform Monte Carlo simulation on one CPU.

Parameters **n_cycles_per_core** (*float*) – Number of simulations per core.

Returns

a dictionary {'rhos': self.rhos, 'rhosrec': self.rhosrec} where

self.rhos Array with raw density matrices.

self.rhosrec Array with maximum likelihood approximated matrices.

Return type dict

sim_counts(*counts*)
Simulates counting statistics noise.

Parameters **counts** (*numpy_array*) – Measured counts.

Returns Array of simulated counting statistics values.

Return type numpy array

CHAPTER

FOUR

LICENSE

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search