`QOptCraft` : user guide

Daniel Gómez Aguado,[1] Juan Carlos García Escartín,[2]

[1]Facultad de Ciencias, Universidad de Valladolid,
Campus Miguel Delibes, Paseo Belén 15, 47011 Valladolid, Spain
[2]Dpto. de Teoría de la Señal y Comunicaciones e Ingeniería Telemática,
ETSI de Telecomunicación, Universidad de Valladolid,
Campus Miguel Delibes, Paseo Belén 15, 47011 Valladolid, Spain

We developed a software package that automates multiple tasks in the study of the evolution of the quantum states of light in linear optical devices. First, we build unitary matrices S giving the classical description of the system, using basic linear optical elements. Following that, we find the quantum n-photon evolution U induced by S. This work also gives functions that solve the inverse problem: rebuilding the S matrix which could generate said evolution. Given the limitations of linear optical devices, the number of possible evolutions $U \in im(\varphi)$ using this method is narrow. For the remaining $U \notin im(\varphi)$, the package gives an alternative procedure based on Toponogov's theorem, which obtains the closest approximation to the evolution to U, $U_a \in im(\varphi)$ in terms of a matrix distance. The scattering matrix results are also generalized to lossy devices, with photon loss and gain between the input and the output, which can be closer to certain real-world experiments.

Contents

1 Introduction

In this file, we present a Python package for quantum linear optics with applications to quantum computing. From simple linear optics instruments, we are able to build complex quantum systems working with photons.

All code has been developed in Python 3 (v3.9.5:0a7dcbd, May 3 2021 17:27:52) due to its flexibility. It includes an appropiate amount of linear algebra functions for our benefit.

2 User guide

This guide discusses the intricacies of using the software package **`QOptCraft`** , a builder of quantum experiments with linear optics elements.

Provided the .zip file *QOptCraft.zip* to the user, the following items can be found:

- *QOptCraft-1.0.tar.gz*, the software package itself.

  For its **installation**, read the next section.

- Three *.py* files, used as Examples of **`QOptCraft`** 's execution:

  - *ex1_Evol4phMatrix2dimU.py*: standard n-photonic evolution of an unitary matrix.
  - *ex2_Build3dimQFTMatrix.py*: concatenation of multiple **`QOptCraft()`** processes.
  - *ex3_QuaSystemfrom2x3dimT.py*: an introduction to quasiunitarity.

We will focus on explaining Python and the package installation, as well as the main functions' capabilities.

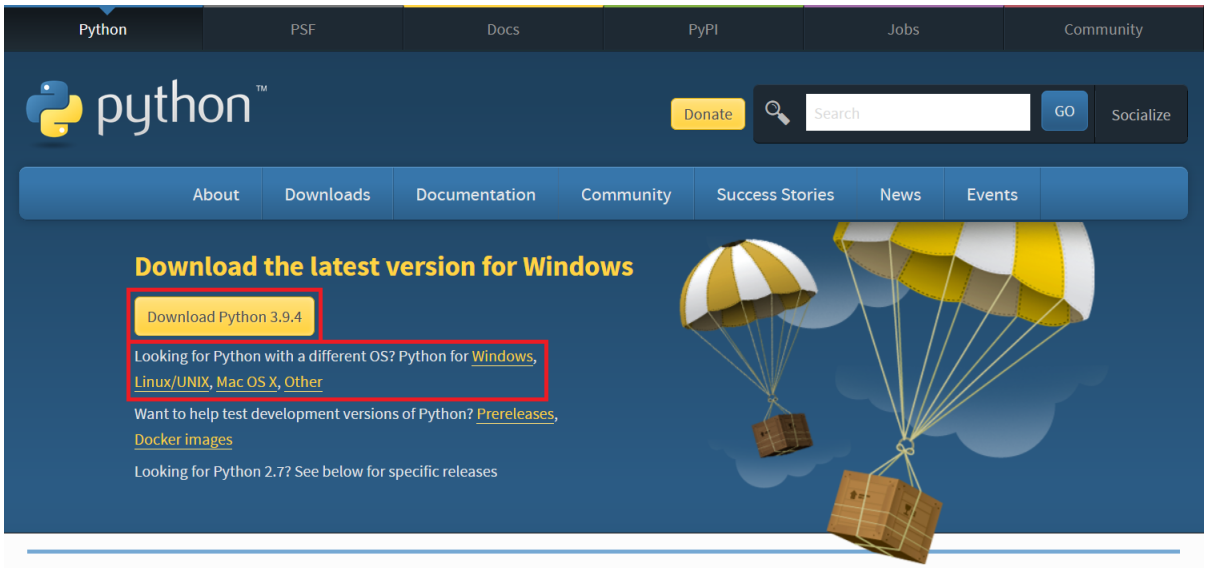
2.1 Installation (for Windows)

The file *QOptCraft.zip* contains the code package in a *.tar.gz* file and a folder with examples of execution.

2.1.1 The code language: Python

The first step consists of obtaining Python, depending on whether the user already successfully installed a compatible version or not. If the latter is true, this section may be skipped.

Go into the website https://www.python.org/downloads/ and download the latest version available. Despite this being a guide for Windows, the full process is identical for other Operating Systems with their respective versions of Python:

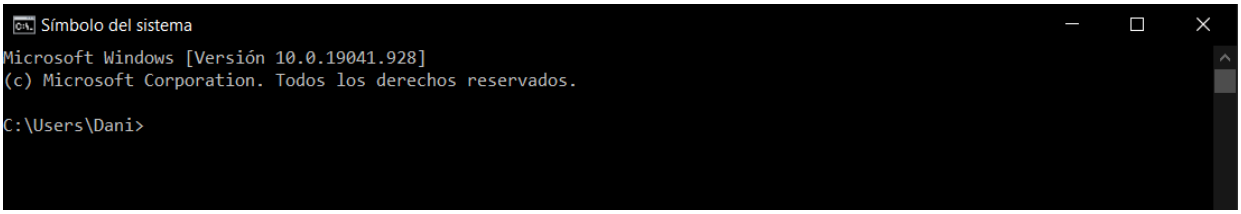


Follow the installer's instructions: when the app finishes, search in the Windows Taskbar for the *IDLE (Python [your version] 32/64-bit)*. You can now access to Python.

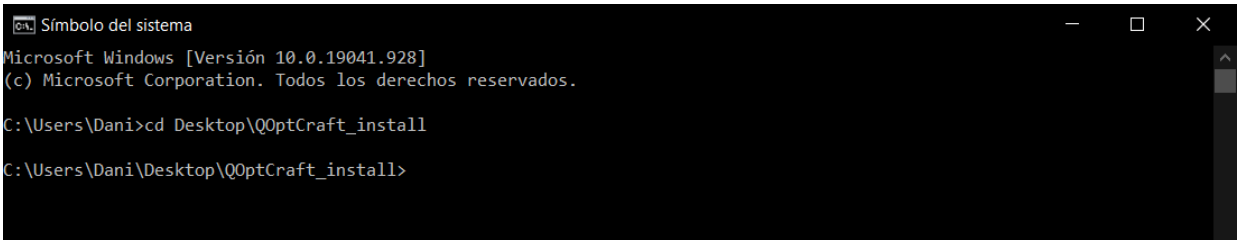
Of course, there are other ways to call Python functions/files. For this package's creation, a text editor (Sublime Text 3) was useful for writing the .py files, since it allows for an easier time identifying a code elements. However, since this manual addresses its use only, such complements will not be required.

2.1.2 The package `QOptCraft`

As for **`QOptCraft`** , extract the .zip files into any path (directory) of choice. Following that, enter into the console by typing **`cmd`** in the Windows Taskbar.



Navigate into your path (directory) of choice with the .zip elements (cd command). For instance:



Following that, type **`pip3 install QOptCraft-1.0.tar.gz`** and the package will be incorporated into the user's Python setup.

Some additional notes:

- For the installation of new versions, do as already indicated for a fresh installation. The previous version will be replaced by the new code.
- In order to uninstall the package, type in any directory `pip3 uninstall QOptCraft`.

There is still some coverage to do, regarding required libraries.

2.1.3 Other libraries

QOptCraft makes use of the well-known packages NumPy, SciPy and SymPy, as well as XlsxWriter for occasional outputs into Excel format. Any Python library can be installed by typing the following in the console, in any path:

```
py -m pip install [library_name]
```

Should the user execute **QOptCraft** or any .py file requiring of an uninstalled library, the program will fail and point out the missing function in the system.

2.1.4 Calling `QOptCraft` functions

The main function **`QOptCraft()`** and all its different blocks are imported to a .py code by writing:

```
from QOptCraft.Main_Code import *
```

It can be specified to simply import **`QOptCraft()`** or any of its components, shall the former not be fully required:

```
from QOptCraft.Main_Code import QOptCraft
```

Alternatives are **`import QOptCraft.Main_Code`** or other uses of the **`import`** command, although the functions' calling will vary depending on your choice. By sticking to our notation, no additional text other than each function's name is needed.

The following section will explain the basics of the general function **`QOptCraft()`** and its corresponding subfunctions. Additional blocks such as **`QOCGen()`**, **`QOCTest()`** or **`QOCLog()`** are included in the guide.

If doubts of the role or total amount of parameters of any of these functions arise, use the **`help()`** command in the Python shell, after importing them from the package.

2.1.5 .txt file format

In case the user wishes to write their .txt input matrix files by hand, the format followed is the following:

```
   (a_{11}), (a_{12}), ..., (a_{1N})
   (a_{21}), (a_{12}), ..., (a_{2N})
      :          :              :
   (a_{N1}), (a_{N2}), ..., (a_{NN})
```

That is, each index **a_{ij}** is written within parenthesis.

- For complex numbers, the notation would be **n_{real}+n_{imag}j** .
- For decimal numbers, a character **.** separating the integer and decimal is required.
- Scientific notation is written as **e+d** or **e-d** , **d** being the number's order, at the end of the index number.

An example of all aspects mentioned is given with the following 3 x 3 unitary matrix:

```
(-2.680e-01+2.273e-01j), (3.054e-01+3.925e-01j), (1.308e-01+7.823e-01j)
(5.162e-01+2.443e-01j), (-7.226e-01+1.498e-01j), (2.331e-01+2.738e-01j)
(7.263e-01+1.513e-01j), (4.559e-01-1.421e-02j), (-4.781e-01+1.139e-01j)
```

When working with the physics surrounding **QOptCraft** , the usage and appearance of complex matrices should be the norm, and expected in the results by the user.

2.2 `QOptCraft` - The full algorithm

Source code: *_0_FullAlgorithm.py*.

`QOptCraft` compresses all the phases of the algorithm in one function.

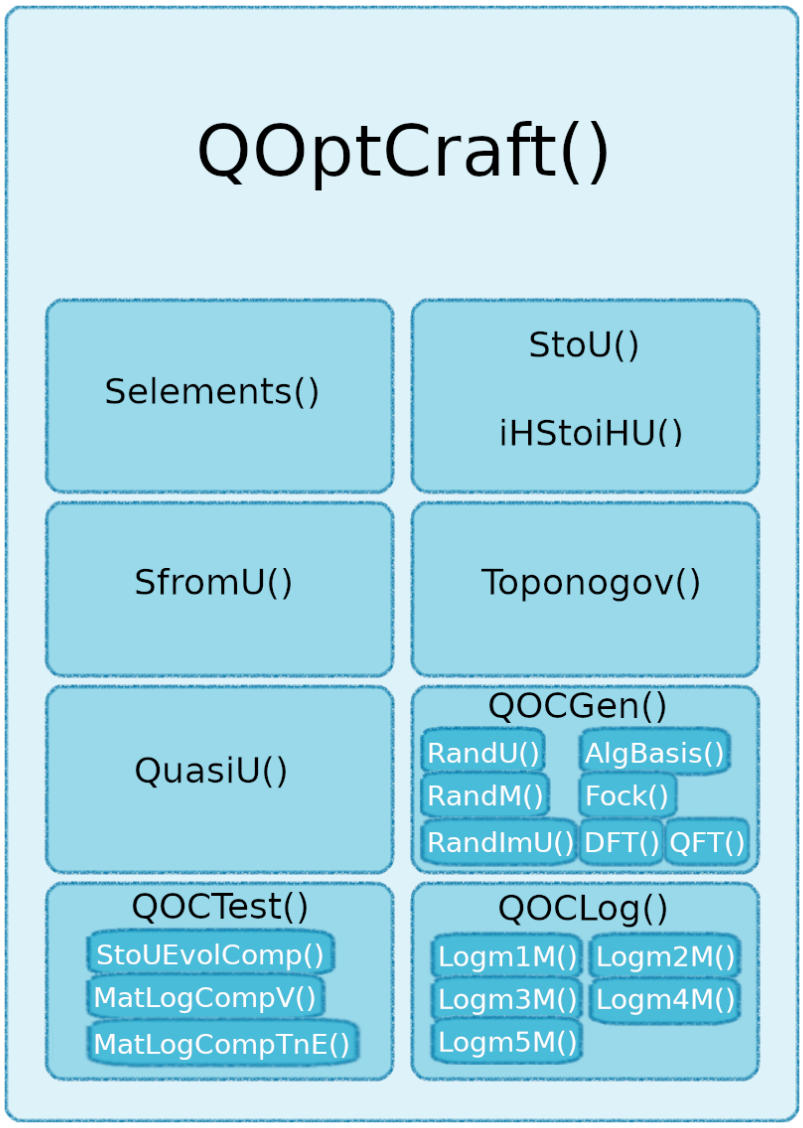


Figure 1: `QOptCraft` schematization. The main function `QOptCraft()` allows the execution of different blocks or subfunctions. For the latter, there exist multiple parameters, ommited in this Figure for simplicity.

Although there is the option of executing each subfunction on it own, `QOptCraft()` compresses them all given the right parameters, which are the following:

- `module` **(int)** chooses the subfunction:
  - Press `1`, or any other int number, for `Selements()` .
  - Press `2` for `StoU()` .
  - Press `3` for `SfromU()` .
  - Press `4` for `Toponogov()` .
  - Press `5` for `QuasiU()` .
  - Press `6` for `iHStoiHU()` .
  - Press `7` for `QOCGen()` .
  - Press `8` for `QOCTest()` .
  - Press `9` for `QOCLog()` .
- `file_input` **(boolean)** activates giving the input either via an external file (`True`) or an array already in the code (`False`).
- `M_input` **(numpy.array)** allows the latter situation mentioned in the previous paragraph. This input serves equally for any function. Not required when using a file as the input.
- `file_output` **(boolean)** must be `True` for generating new .txt files with the code's results. In case it is unnecessary, press `False` .

  NOTE: a standard array output will always be given.
- `filename` **(str)** is the name (without extension) of the new/loaded .txt file, containing the unitary matrix.

  If not given, the user will be asked to input its value on-screen.

- `newfile` **(boolean)** toggles the creation of a new file. Write `True` for its activation, or `False` for opening an already created one.

  NOTE: this parameter is only useful for `Selements` and `QuasiU`.

- `base_input` **(boolean)** activates giving any required basis from a .txt file by toggling (`True`). Since these arrays will be used frequently, this option saves computation time.

  NOTE: this parameter is only useful for `SfromU` and `Toponogov`.

- `txt` **(boolean)** configures the printing of the process in the console.

  Press `True` for a more detailed, property-checking study, or `False` for text omission.

- `acc_d` **(int)** is the amount of decimals numerical results will show on-screen. Only relevant when `txt=True`.

  Since the machine has its limits, writing an abnormaly high number will not ensure displays of said decimal precission.

Were any input to be invalid (for example: `N="hello_world"`, a **str** in a supposedly **int** variable) the code will ask again for its value.

The remaining parameters, normally specific for each particular function, are to be discussed in the following sections.

2.3 `Selements` - Generate unitary compositions of linear optics instruments

Source code [1]: *_1_Unitary_matrix_U_builder.py*

`Selements` creates/loads .txt files containing an unitary matrix and decomposes them into linear optics devices plus the remaining diagonal.

- **`U_un` (numpy.array)**, an unitary matrix, serves the same purpose as **`M_input`** in the full algorithm **`QOptCraft()`**. If newly generated, or introduced from a file, it is not required.
- **`impl` (int)** references the implementation of choice, be it Clements (**`impl=0`**, default) or Reck (**`impl=any other int number`**).
- **`N` (int)** is the matrix's size. Only relevant when creating a new file, in which case N shall be asked on-screen should it not to be inputted in the function.
  It cannot be lower than 2 since that is the lowest matrix dimension available for a linear optics device.

Were any input to be invalid (for example: **`N="hello_world"`**, a **str** in a supposedly **int** variable) the code will ask again for its value.

The output consists of two .txt files (three if {*filename*}*.txt* is a new file):

- {*filename*}*_TmnList.txt* contains, in succession, the linear optics elements decomposition of the unitary matrix.
- {*filename*}*_D.txt* is the resulting diagonal, indicating a different phase for each mode.

2.4 `StoU` - Evolve unitary scattering matrices for an optic system of n photons

Source code [2] [3] [4] [5]: *_2_Get_U_matrix_of_photon_system_evolution.py*

`StoU` loads .txt files containing an unitary matrix (the scattering matrix S). Depending on the total number of photons within the modes, a different evolution matrix U will be obtained.

- **`S` (numpy.array)**, the S-matrix, serves the same function as **`M_input`** in the full algorithm **`QOptCraft()`**. If introduced from a file, it is not required.
- **`method` (int)** references the evolution method of choice:
  - **`method=0`** Quantum mechanical description.
  - **`method=1`** Standard permanents.
  - **`method=2`** Ryser permanents (default: generally, the fastest option for our tests).
  - **`method=3`** Computation via effective Hamiltonians.
- **`n` (int)**, the number of photons. Needed for the dimension **`M=comb(n,m)`** of the output. At least 1 photon (case in which the evolution corresponds to the S-matrix) is needed.

Were any input to be invalid (for example: **`n="hello_world"`**, a **str** in a supposedly **int** variable) the code will ask again for its value.

The output consists of three .txt files:

- {*filename*}_*m*_{*m*}_*n*_{*n*}_*coefs_method*_{*method*}*.txt* contains the evolution matrix U.
- {*filename*}_*m*_{*m*}_*n*_{*n*}_*probs_method*_{*method*}*.txt* contains U's indexes but squared, each corresponding to the probability of a photon to go from an initial to final mode.
- *m*_{*m*}_*n*_{*n*}_*vec_base.txt* is an auxiliar file, also generated by **`QOCGen()`**. It stores the Fock states photon basis: each posible distribution of **`n`** photons in **`m`** modes, ordered.

2.5 `SfromU` - Rebuild the original matrix S from any viable evolution matrix U

Source code [6]: *_3_Get_S_from_U_inverse_problem.py*

`SfromU` loads .txt files containing an evolution matrix U. Should it be buildable via linear optics elements, its scattering matrix of origin S will be rebuilt. Modes can be permuted for different ways of placing the instruments.

- `U` **(numpy.array)**, an already evolved matrix, serves the same function as `M_input` in the full algorithm `QOptCraft()`. If introduced from a file, it is not required.
- `perm` **(boolean)** lets the user choose between permuting the Fock states modes' positions or simply computing `S` for the standard disposition. Write `True` for its activation. Useful for finding new combinations of devices. `False` by default, for a faster pace.
- `m` **(int)** is the rebuilt S-matrix's number of modes. At least 2 modes (for having the minimal square matrix compatible with linear optics elements) are needed.
- `n` **(int)** is the number of photons employed in the evolution of our desired S-matrix output. At least 1 photon (case in which the evolution corresponds to the S-matrix) is needed.

Were any input to be invalid (for example: `m="hello_world"`, a **str** in a supposedly **int** variable) the code will ask again for its value.

The output consists of three .txt files:

- {*filename*}_*m*_{*m*}_*n*_{*n*}_*S_recon_main.txt* contains the rebuilt S-matrix for the standard disposition of modes.
- {*filename*}_*m*_{*m*}_*n*_{*n*}_*S_recon_all.txt* and {*filename*}_*m*_{*m*}_*n*_{*n*}_*S_recon_all_U.txt* contain the rebuilt S-matrix for all permutations and their corresponding U, respectively.
- *m*_{*m*}_*n*_{*n*}_*vec_base.txt*: see the section for `StoU` or, most notably, `QOCGen`.

2.6 `Toponogov` - Find valid approximations for unavailable evolution matrices U

Source code [7]: *_4_toponogov_theorem_for_uncraftable_matrices_U.py*

`Toponogov` loads .txt files containing an unavailable for implementation evolution matrix U. Given an amount of tries and a number of modes m and photons n, by the Toponogov theorem the closer viable evolutions are found, executing a matrix bombing from an initial point.

- **`U` (numpy.array)**, an unavailable evolved matrix, serves the same function as **`M_input`** in the full algorithm **`QOptCraft()`**. If introduced from a file, it is not required.
- **`tries` (int)** is the amount of times the algorithm will be executed. Different solutions can be found depending on the initial matrix and the bombing. Its minimal amount is 1.
- **`m` (int)** is the random unitary S-matrices' number of modes Required for the random bombing. At least 2 modes for the minimal craftable square matrix are needed.
- **`n` (int)** is the number of photons employed in the evolution of the random S-matrices. Also required for the random bombing.
  At least 1 photon (case in which the evolution corresponds to the S-matrix) is needed.

Were any input to be invalid (for example: **`m="hello_world"`**, a **str** in a supposedly **int** variable) the code will ask again for its value.

The output consists of **`tries+1`** .txt files:

- {*filename*}*_toponogov_general.txt* contains all valid approximations of U found for each attempt, and their separation lengths with the original.
- {*filename*}*_toponogov_N.txt*, N corresponding to the number of each try, gives the output matrices separately, ready to be imported.

2.7 `QuasiU` - Generate compositions of linear optics instruments with loss

Source code [8]: *_5_Quasiunitary_S_with_or_without_loss_builder.py*

`QuasiU` creates/loads .txt files containing a quasiunitary matrix and decomposes them via the singular value decomposition, and already developed algorithms for **`Selements()`**, into linear optics devices plus the remaining diagonal, as the total scattering matrix S.

- **`T`** **(numpy.array)**, a quasiunitary matrix, serves the same function as **`M_input`** in the full algorithm **`QOptCraft()`**. If newly generated or introduced from a file, it is not required.
- **`N1`** and **`N2`** **(both int)** are the first and second matrix dimensions. Only relevant when creating a new file, **`N1`** or **`N2`** shall be asked on-screen should any of these parameters not to be inputted in the function.

  They cannot be lower than 1.

Were any input to be invalid (for example: **`N1="hello_world"`**, a **str** in a supposedly **int** variable) the code will ask again for its value.

The output consists of five .txt files:

- {*filename*}*_SU.txt*, {*filename*}*_SD.txt* and {*filename*}*_SW.txt* are the scattering evolution for the matrices U, D and W of the singular value decomposition, respectively.
- {*filename*}*_S_quasiunitary.txt* is the total scattering matrix for our input T.
- {*filename*}*_S.txt* is a shorter version of the previous file, containing only half its dimensions. Useful for only considering a type of both creation/annihilation operators.

2.8 `QOCGen` - A mutiple types of matrices generator

Source code:

- *_0_FullAlgorithm.py* `QOCGen()`
- *_8_generators.py* `RandU() RandM() Fock() AlgBasis() DFT() QFT() RandImU()`

`QOCGen` allows the user to generate any type of matrix covered by `QOptCraft` individually, incluiding unitary, random, Discrete Fourier Transform and Quantum Fourier Transform matrices. The option to generate vector basis for Fock states (relevant in `StoU() or SfromU()`) and subalgebra $\mathbf{u}(\mathrm{m})$, $\mathbf{u}(\mathrm{M})$ matrices (`SfromU() or Toponogov()`) is given as well.

- `choice` **(int)** determines the new matrix's type:
  - Press `0` or any other int number for unitary matrices via `RandU()`.
    Requires a dimension `N`, given as a parameter or in the console.
  - Press `1` for random matrices via `RandM()`.
    Requires dimensions `N1` and `N2`, given as parameters or in the console.
  - Press `2` for n-photonic Fock states basis via `Fock()`.
    Requires a number of modes `m` and photons `n`.
  - Press `3` for subalgebras basis $\mathbf{u}(\mathrm{m})$ and $\mathbf{u}(\mathrm{M})$ via `AlgBasis()`.
    Requires a number of modes `m` and photons `n`, and their combination `M`. It **will only work if** `M=comb(n,m)`. In the contrary case, new `m` and `n` values are asked.
  - Press `4` for a DFT matrix via `DFT()`.
    Requires a dimension `N`.
  - Press `5` for a QFT matrix via `QFT()`.
    Requires a dimension `N`. There is an optional **boolean** argument, `inverse`, for the computation of inverse QFTs.

– Press `6` for an evolution matrix $\mathbf{U} \in \mathbf{im}(\varphi)$, craftable with linear optic devices, via `RandImU()`.

Requires a number of modes `m` and photons `n`, for the size of its available S-matrix.

Both values decide U's size `M`.

- We also remind from the general `QOptCraft()` function the parameter `file_output` **(boolean)**, since it could be crucial for the generator. It must be `True` for generating new .txt files with the code's results. In case it is unnecessary, press `False`.

Were any input to be invalid (for example: `choice="hello_world"`, a **str** in a supposedly **int** variable) the code will ask again for its value.

Some of the following list of files are already being generated on previous functions. By pointing out where, the user can understand their purpose on the library.

- *m_{m}_n_{n}_vec_base.txt* is an auxiliar file generated by `Fock()`, storing the Fock states photon basis employed for computation. Such relevant information, reminding the user of the different dispositions of n photons in m modes, and most importantly, their order on the basis.

  For example, for `m=2, n=3` its basis would be $\{|30\rangle, |21\rangle, |12\rangle, |03\rangle\}$, interpreted in computation as the following:

$$|30\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |21\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad |12\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad |03\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

- *base_u_m_{m}.txt* and *base_u__M_{M}.txt* are auxiliar files generated by `AlgBasis()`, storing the m x m and M x M matrix bases in the subalgebra $\mathbf{u(m)}$ and $\mathbf{u(M)}$, respectively.

The craftable matrices, they can be found in a filename.txt file, should the option `file_output` be toggled on.

2.9 `QOCTest` - Verify the validity of `QOptCraft` 's logarithm and matrix evolution algorithms

Source code [2] [3] [4] [5] [9]:

- *_0_FullAlgorithm.py* **`QOCTest()`**
- *_2_aux_a_computation_time_evolutions_comparison.py* **`StoUEvolComp()`**
- *_2_aux_b_logarithm_algorithms_equalities.py* **`MatLogCompV()`**
- *_2_aux_c_logarithm_algorithms_timesanderror.py* **`MatLogCompTnE()`**

`QOCTest()` holds all the comparative functions between functions such as the different S-matrix evolution methods developed for **`StoU`**, or varying implementations for matrix logarithms. By comparison of their results, we can analyse their performance in terms of value, time of computation and error.

- **`choice`** **(int)** decides which function to execute:
  - Press **`0`** or any other int number for **`StoUEvolComp()`**, testing each evolution method's computing speeed for an interval of [**`m1, m2`**] modes and [**`n1, n2`**] photons, introducer either via parameters, or in the console.
  - Press **`1`** for **`MatLogCompV()`**, comparing for matrices in an interval [**`N1, N2`**] the value of their different matrix logarithms. Another parameter **`exp`** is included, enabling comparisons between the exponential of those logarithms (**`exp=2`**).
  - Press **`2`** for **`MatLogCompTnE()`**, comparing for matrices in an interval [**`N1, N2`**] the computation times and error of the matrix logarithms. **`exp`** is again included, this time choosing between $N_{total} = 2^{N_{input}}$ (**`exp=2`**) and $N_{total} = N_{input}$ (**`exp=1`**).

Were any input to be invalid (for example: **`choice="hello_world"`**, a **str** in a supposedly **int** variable) the code will ask again for its value.

2.10 `QOCLog` - Compute the logarithm of a matrix, chosing between five metodologies

Source code [9]:

- *_0_FullAlgorithm.py* `QOCLog()`
- *_10_friendly_logarithm_algorithms.py* `Logm1M()` `Logm2M()` `Logm3M()` `Logm4M()` `Logm5M()`

An easy access towards the tested logarithms on `QOCTest` could be of interest for many developers, so there it is. Given a matrix `A`, introduced from a .txt file or as a parameter, `QOCLog()` can compute its logarithm in five different ways, and export it to a new file.

- `A` **(numpy.array)** is the input matrix whose logarithm we want to find. If introduced from a file by typing `file_input=True`, it is not required.
- `choice` **(int)** determines the new matrix's type:
  - Press `1` for `Logm1M()`.
  - Press `2` for `Logm2M()`.
  - Press `3` or any other nonmentioned int number for `Logm3M()`.
  - Press `4` for `Logm4M()`.
  - Press `5` for `Logm5M()`.
- We might as well remind from the general `QOptCraft()` function the parameter `file_output` **(boolean)**. It must be `True` for generating new .txt files with the code's results. In case it is unnecessary, press `False`.

The output consists of a .txt file:

- {*filename*}*_Logm*{*i*}*.txt* contains the matrix logarithm `Logm{i}M(A)` of `A`.

3 Examples

Examples of usage are showcased for a better understanding of how to work with **`QOptCraft`**.

Said examples will cover the following:

- Evolution for a craftable, unitary, scattering matrix **`S`**.
- An attempt to build a 3 x 3 QFT matrix **`QFTM`** by linear optic devices.
- The treatment of a random matrix **`T`**, resulting in a quasiunitary system.

First, we will introduce each problem and its respective full code. Furthermore, we will analyze the latter by explaining how each command is called.

3.1 4-photonic evolution of new 2-dimensional matrix `S`

Trial code name: *ex1_Evol4phMatrix2dimU.py*

Our goal is to obtain the unitary evolution matrix **`U`** for **`n=4`** photons in a system of modes or dimensions **`m=2`**. By using the general function **`QOptCraft()`** we only need to call it twice, specifying for each the parameter **`module`**. This is the full code:

```
# Evolution of S (m=2, n=4)

# We first generate and decompose the unitary scattering matrix in
↪ linear optic devices:
QOptCraft(file_input=False,filename="S_dim2",newfile=True,
file_output=True,module=1,impl=0,N=2)

# Obtaination of "S_dim2.txt"'s evolution of n=4 photons U.
QOptCraft(file_input=True,filename="S_dim2",
file_output=True,module=2,n=4,method=2)
```

We will follow by making a **breakdown** on said code.

3.1.1 Creation of `S`

- **`QOptCraft(module=1)=Selements()`**. First, a matrix **`S`** of dimension **`N=2`** is generated (**`newfile=True, file_input=False`**) and exported to a new file *S_dim2.txt* (**`file_output=True`**). **`impl=0`** for using Clements' approach towards decomposition.

```
# We first generate and decompose the unitary scattering matrix in
↪ linear optic devices:
QOptCraft(file_input=False,filename="S_dim2",newfile=True,
file_output=True,module=1,impl=0,N=2)
```

An easier to understand alternative for new users is to use **`QOCGen()`** and **`Selements()`** separately: both functions give access to **`RandU()`**, nonetheless the former is the one associated with generating new matrices (and it is uncommon as well for other functions to hold the same capability, being **`RandU()`** in **`Selements()`** and **`RandM()`** in **`QuasiU()`** the only exceptions):

- **`QOptCraft(module=8,choice=0)=QOCGen(choice=0)=RandU()`**. First, a matrix **`S`** of dimension **`N=2`** is generated (we omit **`file_output=True`** this time as it is toggled by default, both in this command and previous examples).

- **`QOptCraft(module=1)=Selements()`**. The only difference with **`Selements()`**'s previous execution is **`newfile=False, file_input=True`**, since *S_dim2.txt* was already generated.

```
# We first generate the unitary scattering matrix:
QOptCraft(filename="S_dim2",module=8,choice=0,N=2)

# Then, it is decomposed:
QOptCraft(file_input=True,filename="S_dim2",
newfile=False,file_output=True,module=1,impl=0)
```

3.1.2 Photonic evolution

- `QOptCraft(module=2)=StoU()`. We load *S_dim2.txt* (`file_input=True`) and introduce `n=4` photons. Our algorithm of choice for `U`'s computation is the Ryser permanents (`method=2`).

```
# Obtaination of "S_dim2.txt"'s evolution of n=4 photons U.
QOptCraft(file_input=True,filename="S_dim2",
file_output=True,module=2,n=4,method=2)
```

As a result, we obtained the file *S_dim2_m_2_n_4_coefs_method_2.txt*, containing the desired evolution `U`, as well as *S_dim2_TmnList.txt* and *S_dim2_D.txt* for building its source, the scattering matrix `S`.

3.2 Difficulties when building a 3-dimensional QFT matrix

Trial code name: *ex2_Build3dimQFTMatrix.py*

For a fresly generated evolution matrix *QFT_matrix_3.txt*, an adequate use of QOptCraft can return the optics devices needed for building one of its implementable approximations, given by the Toponogov theorem.

This time, we will call `QOptCraft()` five times: first, we create `QFTM`, for the further attempt of finding a compatible 2-dimensional matrix `S`. Given the negative result, we search for an approximation `QFTM_t` instead.

```
# QFT_matrix (N=3)

# We first generate the 3 x 3 QFT matrix:
QOptCraft(filename="QFT_matrix_3",module=8,choice=5,N=3)

# Is the original matrix aleady plausible?
QOptCraft(file_input=True,filename="QFT_matrix_3",
txt=True,acc_d=2,module=3,m=2,n=2)
```

```
9
10    # Obtaination of "QFT_matrix_3.txt"'s closest evolution matrix U.
11    QOptCraft(file_input=True,filename="QFT_matrix_3",base_input=False,
12    file_output=True,module=4,m=2,n=2,tries=20)
13
14    # Obtaination of "QFT_matrix_3_toponogov_2.txt"'s S-matrix.
15    QOptCraft(file_input=True,filename="QFT_matrix_3_toponogov_2",
16    file_output=True,module=3,m=2,n=2)
17
18    # Decomposition of "QFT_matrix_3_toponogov_2.txt's S-matrix".
19    QOptCraft(file_input=True,file_output=True,module=1,newfile=False,
20    impl=0,filename="QFT_matrix_3_toponogov_2_m_2_n_2_S_recon_main")
```

We will follow by making a **breakdown** on said code.

3.2.1 Creation of `QFTM` and compatibility issues

- `QOptCraft(module=8,choice=5)=QOCGen(choice=5)=QFT()`. We pick `N=3` for its dimensions. The original `QFTM` matrix will be stored in *QFT_matrix_3.txt*.

```
1     # We first generate the 3 x 3 QFT matrix:
2     QOptCraft(filename="QFT_matrix_3",module=8,choice=5,N=3)
```

- `QOptCraft(module=3)=SfromU()`. There is the possibility of `QFTM` being craftable as it is already. We wish for a 2-dimensional (`m=2`) `S` matrix, which requires `n=2` photons for reaching `M=3`, the dimension of `QFTM`.

  Despite the file output telling whether `QFTM` is craftable or not, this process is easier to check by writing `txt=True`, giving the user information through the console.

```
1     # Is the original matrix aleady plausible?
2     QOptCraft(file_input=True,filename="QFT_matrix_3",
3     txt=True,acc_d=2,module=3,m=2,n=2)
```

3.2.2 Application of Toponogov's theorem

- **`QOptCraft(module=4)=Toponogov()`**. It is time to search for the closest, available approach to **`QFTM`**. Again, we want our **`S`** matrix to be 2-dimensional. We introduce our QFT matrix from *QFT_matrix_3.txt* (**`file_input=True`**), and will make 20 attempts via **`tries=20`** for finding a solution. We do not need to introduce our matrix basis of $\mathbf{u}(\mathbf{m})$ and $\mathbf{u}(\mathbf{M})$ through a file (**`base_input=False`**).

```
# Obtaination of "QFT_matrix_3.txt"'s closest evolution matrix U.
QOptCraft(file_input=True,filename="QFT_matrix_3",base_input=False,
file_output=True,module=4,m=2,n=2,tries=20)
```

3.2.3 Available `S` and decomposition

- **`QOptCraft(module=3)=SfromU()`**. Now, the file we introduce will be one of the multiple solutions obtained by the previous command. In our experiment, the closest solution **`QFTM_t`** to the original **`QFTM`** by metric was the second (*QFT_matrix_3_toponogov_2.txt*). A **`S`** matrix for **`m=2, n=2`** is found.

```
# Obtaination of "QFT_matrix_3_toponogov_2.txt"'s S-matrix.
QOptCraft(file_input=True,filename="QFT_matrix_3_toponogov_2",
file_output=True,module=3,m=2,n=2)
```

- **`QOptCraft(module=1)=Selements()`**. By using Clements' algorithm (**`impl=0`**), we decompose **`S`** (*QFT_matrix_3_toponogov_2_m_2_n_2_S_recon_main_S.txt*) successfully.

```
# Decomposition of "QFT_matrix_3_toponogov_2.txt's S-matrix".
QOptCraft(file_input=True,file_output=True,module=1,newfile=False,
impl=0,filename="QFT_matrix_3_toponogov_2_m_2_n_2_S_recon_main")
```

Our approximate evolution **`QFTM_t`** can be found in *QFT_matrix_3_toponogov_2.txt*. This file is not the only Toponogov result, as there are nine solutions. However, it is the closest to

the original *QFT_matrix_3.txt* in terms of metric. All solutions with their respective distances to the original are stored as well in *QFT_matrix_3_toponogov_general.txt*.

We also obtained *QFT_matrix_3_toponogov_2_m_2_n_2_S_recon_main_S.txt*, containing the `S` matrix, and its decomposition (*QFT_matrix_3_toponogov_2_m_2_n_2_S_recon_main_S_TmnList.txt* and *QFT_matrix_3_toponogov_2_m_2_n_2_S_recon_main_S_D.txt*).

3.3 Quasiunitary system from a random 2 x 3 matrix `T`

Trial code name: *ex3_QuaSystemfrom2x3dimT.py*

Systems with loss can be simulated by introducing quasiunitarity on matrices. Their creation can be given by an item as simple as a random matrix, be it square or not.

Considering `QuasiU()`'s integration of `RandM()`, only one command is really required:

```
# Quasiunitary system S from random N1=2 x N2=3 matrix T

# We generate "T_dim2x3.txt", for later obtaination of its
↪  quasiunitary representation S.
QOptCraft(file_input=False,filename="T_dim2x3",
newfile=True,file_output=True,module=5)
```

However, since we want to explain step-by-step the process, we will call `QOptCraft()` twice: first, the generator `QOCGen()`, and last, `QuasiU()`.

```
# Quasiunitary system S from random N1=2 x N2=3 matrix T

# We first generate the random matrix:
QOptCraft(filename="T_dim2x3",module=8,choice=1,N1=2,N2=3)

# Obtaination of "T_dim2x3.txt"'s quasiunitary representation S.
QOptCraft(file_input=True,filename="T_dim2x3",
newfile=False,file_output=True,module=5)
```

We will follow by making a **breakdown** on said code.

3.3.1 Creation of `T`

- `QOptCraft(module=8,choice=1)=QOCGen(choice=1)=RandM()`. Since we want the most general case possible, our dimension choice is not quadratic: `N1=2, N2=3`.

```
# We first generate the random matrix:
QOptCraft(filename="T_dim2x3",module=8,choice=1,N1=2,N2=3)
```

3.3.2 Quasiunitary matrix `S`

- `QOptCraft(module=5)=QuasiU()`. This command will export multiple files regarding the quasiunitary adaptation of `T`. We introduce the previous file matrix *T_dim2x3* (`file_input=True, newfile=False`) and obtain anew: *T_dim2x3_S_quasiunitary.txt*.

```
# Obtaination of "T_dim2x3.txt"'s quasiunitary representation S.
QOptCraft(file_input=True,filename="T_dim2x3",
newfile=False,file_output=True,module=5)
```

As said before, we obtained the file *T_dim2x3_S_quasiunitary.txt*, containing the full quasiunitary scattering matrix `S`. For particular cases (such as no parametric amplifiers), non-diagonal blocks of the matrix become zero. Thus, diagonal blocks become unitary, and only one of them would be needed to fully describe the system. Given those cases exist, the code also stores the first diagonal block on its own in another file *T_dim2x3_S.txt*.

References and Notes

[1] W. R. Clements, P. C. Humphreys, B. J. Metcalf, W. S. Kolthammer, and I. A. Walsmley, "Optimal Design for Universal Multiport Interferometers", Optica 3, 1460 (2016).

[2] J. Skaar, J. C. García Escartín, and H. Landro, "Quantum mechanical description of linear optic", American Journal of Physics 72, 1385 (2004).

[3] S. Scheel, "Permanents in linear optics network", Acta Physica Slovaca 58, 675 (2008).

[4] "Permanents and Ryser's algorithm", numbersandshapes.net.

[5] J. C. García Escartín, V. Gimeno, and J. J. Moyano-Fernández, "Multiple photon effective Hamiltonians in linear quantum optical networks", Optics Communications 430 (2019) 434–439.

[6] J. C. García Escartín, V. Gimeno, and J. J. Moyano Fernández, "A method to determine which quantum operations can be realized with linear optics with a constructive implementation recipe", Physical Review A 100, 022301 (2019).

[7] J. C. García Escartín and J. J. Moyano Fernández, "Optimal approximation to unitary quantum operators with linear optics", arXiv:2011.15048v1 [quant-ph].

[8] N. Tischler, C. Rockstuhl, and K. Slowik, "Quantum Optical Realization of Arbitrary Linear Transformations Allowing for Loss and Gain", Physical Review X 8, 021017 (2018).

[9] T. A. Loring, "Computing a logarithm of a unitary matrix with general spectrum", Numerical Linear Algebra wth Applications, 21 (6) 744–760 (2014).