QCS: Quantum Correlation Solver

Release 1.0.0

A Practical Guide

Zhi-Guang Lu

Updating

May 16, 2023

Contents

1. Frontmatter

1.1 About This Documentation

This document contains a user guide and manually generated API documentation for QCS. A PDF version of this text only is available at the Github.

1.2 Citing This Project

If you find this project useful, then please cite:

xxx

which may also be download from `http://arxiv.org/abs/2305.08923`.

1.3 About QCS

The correlation function plays a key role in quantum physics. For example, the statistical properties of light could be acquired by the second-order correlation function, and the photon transport can be characterized by single-photon transimission and reflection. Consequently, analytical or numerical calculating the higher-order correlation function is very important in quantum physics, which is the reason that we develop the tool.

In general, the calculation of higher-order correlation function, whether numerical or analytical, is extremely difficult for any open quantum systems. For example, multi-cavity or nonidentical multi-atom systems will face dimensional exponential growth of Hilbert space with the size of the cavity or atom. However, under one or multiple weak coherent inputs, the computation complexity of higher-order equal-time correlation function (ETCF) will be reduced from exponential to polynomial when the system Hamiltonian satisfies U(1) symmetry.

The Quantum Correlation Solver, or QCS, is an open-source framework written in the Python programming language, designed for calculating the higher-order ETCF of systems that satisfy the abovementioned criterion. This framework distinguishes itself from other available software solutions in providing the following advantages:

- QCS relies entirely on open-source software. You are free to modify and use it as you wish with no licensing fees or limitations.
- QCS is based on the Python scripting language, providing easy to read, fast code generation without the need to compile after modification.

- QCS allows for solving the higher-order ETCF, cross-correlation function, transimission, and reflection with arbitrary multiple weak coherent driven cases.
- QCS allows for solving the large size systems, such as multi-cavity and multi-atom.

For detailed information about new features of each release of QCS, see the *Change Log*.

1.4 Contributing to QCS

We welcome anyone who is interested in helping us perfected QCS. If you are interested, please contact me.

2. Installation

2.1 General Requirements

QCS mainly depends on two open-source libraries for scientific computing in the Python programming language. The following packages are currently required:

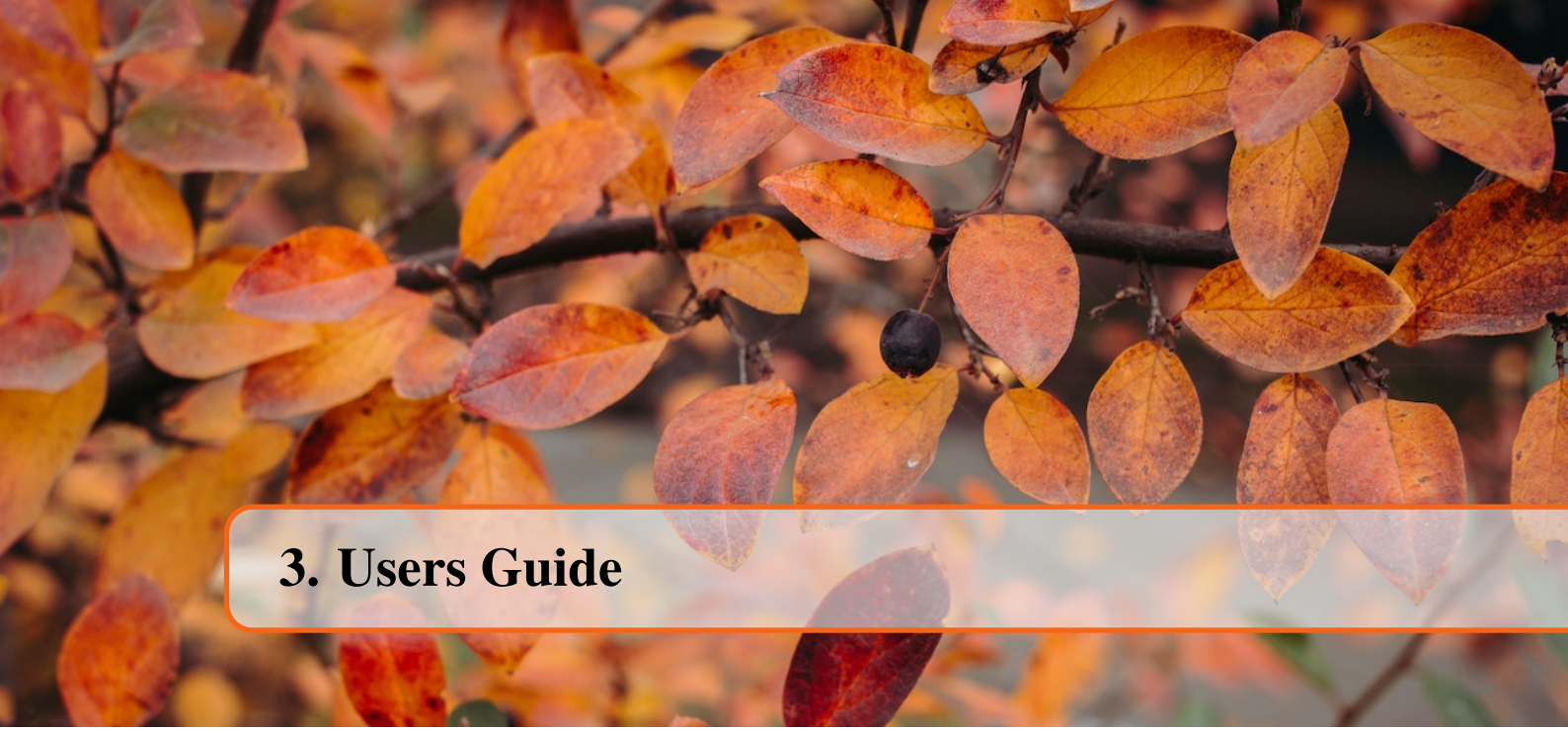
Package	Version	Details
Python	3.0+	Not tested on lower versions.
NumPy	1.8+	Not tested on lower versions.
SciPy	0.15+	Lower versions have missing features.

2.2 Installing via pip

It is often easiest to use the Python package manager pip.

```
pip install qcs_phy
```

Note that our Python package name is `qcs_phy`.



3. Users Guide

3.1 Guide Overview

The goal of this guide is to introduce you to the basic structures and functions that make up QCS. This guide is divided up into several sections, each highlighting a specific set of functionalities. In combination with the examples that can be found on the project web page `https://github.com/ZhiGuangLu/qcs/tree/main/examples`, this guide should provide a more or less complete overview. In addition, the *API documentation* for each function is located at the end of this guide.

3.2 Basic Operations on Quantum Objects

3.2.1 First thing first

To load the qutip modules, we must first call the import statement:

```
In [1]: from qcs_phy import qcs
```

that will load the kernel class `qcs`. Often, we also need to import the NumPy and Matplotlib libraries with:

```
In [2]: import numpy as np
In [3]: import matplotlib.pyplot as plt
```

Note that the class `qcs` contains all of the user available functions.

3.2.2 The symbol stipulations of quantum operators and Hamiltonian

Here, we must have to define a class of creation and annihilation operators about boson and fermion. For the sake of simplicity, we only consider three physical modes, i.e., the cavity mode, the spin-1/2 mode, and the collective spin mode. The specific representations are shown below:

Physical Mode	Symbol: annihilation	Symbol: creation	Python Code: `str`
cavity	a	$a^\dagger$	`"a"`, `"ad"`
spin-1/2	σ	$\sigma^\dagger$	`"sm"`, `"sp"`
n collective spins	$S=\sum_{i=1}^{n}\sigma_i$	$S^\dagger=\sum_{i=1}^{n}\sigma_i^\dagger$	`"Sm N=n"`, `"Sp N=n"`
	$S_z=\sum_{i=1}^{n}\sigma_i^\dagger\sigma_i$		`"Sz N=n"`

Based on the definition above, we could construct Hamiltonian of any form. However, in order to obtain the matrix form of Hamiltonian at the subspace of certain excitation number, we must have to number each physical mode, i.e., existing in the form of `tuple` in Python, such as

$a_k^\dagger \leftrightarrow$("ad", k), $a_k \leftrightarrow$("a", k), $\sigma_k \leftrightarrow$("sm", k), $S_k \leftrightarrow$("Sm N=n", k), $\cdots$

In general, each Hamiltonian carries the corresponding coefficients in addition to containing various physical modes, and the form exists in the form of `list` in Python. Now, we consider a kind of Hamiltonian that only contains one term, and the specific representations are shown in the table below:

One term	Python Code: `list`
$Ea_1^\dagger a_1$	[E, ("ad", 1), ("a", 1)]
$E\sigma_1^\dagger \sigma_1$	[E, ("sp", 1), ("sm", 1)]
$Ea_1^\dagger \sigma_1$	[E, ("ad", 1), ("sm", 1)]
$Ea_1 \sigma_1^\dagger$	[E, ("a", 1), ("sp", 1)]
$Ea_1^\dagger a_1^\dagger a_1 a_1$	[E, ("ad", 1), ("ad", 1), ("a", 1), ("a", 1)]

Notably, the Python code of this Hamiltonian is described by the `list` form:

Hj = [coff, ope1, ope2, $\cdots$] $\Leftrightarrow$ Hj = coff $\times$ ope1 $\times$ ope2 $\times$ $\cdots$

Here, ope1 and ope2 are denote as the `tuple` above, such as ope1 $= a_1 \leftrightarrow$ ope1 = ("a", 1).

Similarly, for the Hamiltonian containing multiple term, we use `list[list]` Python code to represent it. Here, we take the Hamiltonian containing two terms as example, i.e.,

Two terms	Python Code: `list[list]`
$Ea_1^\dagger a_1 + F\sigma_1^\dagger \sigma_1$	[[E, ("ad", 1), ("a", 1)],[F, ("sp", 1), ("sm", 1)]]
$Ea_1^\dagger \sigma_1 + Fa_1 \sigma_1^\dagger$	[[E, ("ad", 1), ("sm", 1)],[F, ("a", 1), ("sp", 1)]]

Obviously, it is also true for multiple terms, and the Python code of this Hamiltonian is described by the `list` form:

H = [H1, H2, H3, $\cdots$] $\Leftrightarrow$ H = H1 + H2 + H3 + $\cdots$

Here, H1, H2, and H3 have the `list` form: Hj = [coff, ope1, ope2, ...].

Note: For the Hamiltonian containing one term, we also use the `list[list]` form in order to ensure consistency.

Example: Constructing effective Hamiltonian

Here, we take four effective Hamiltonian as examples, i.e.,

$$H_1 = (\omega_c - i\kappa/2)a_1^\dagger a_1 + (\omega_e - i\gamma/2)\sigma_1^\dagger \sigma_1 + g(a_1^\dagger \sigma_1 + a_1 \sigma_1^\dagger),$$
$$H_2 = (\omega_c - i\kappa/2)a_1^\dagger a_1 + (\omega_e - i\gamma/2)\sigma_1^\dagger \sigma_1 + g(a_1^{\dagger 2} \sigma_1 + a_1^2 \sigma_1^\dagger),$$
$$H_3 = (\omega_c - i\kappa/2)a_1^\dagger a_1 + (\omega_e - i\gamma/2)\sum_{j=1}^{N} \sigma_j^\dagger \sigma_j + \sum_{j=1}^{N} g_j(a_1^\dagger \sigma_j + a_1 \sigma_j^\dagger),$$
$$H_4 = \sum_{j=1}^{N} [(\omega_c - i\kappa/2)a_j^\dagger a_j + U a_j^\dagger a_j^\dagger a_j a_j] + J\sum_{j=1}^{N-1} (a_j^\dagger a_{j+1} + a_j a_{j+1}^\dagger).$$

The corresponding Python codes are shown below:

```
# construct H1
H1 = []
H1.append([ω_c − iκ/2, ("ad", 1), ("a", 1)])
H1.append([ω_e − iγ/2, ("sp", 1), ("sm", 1)])
H1.append([g, ("ad", 1), ("sm", 1)])
H1.append([g, ("a", 1), ("sp", 1)])

# construct H2
H2 = []
H2.append([ω_c − iκ/2, ("ad", 1), ("a", 1)])
H2.append([ω_e − iγ/2, ("sp", 1), ("sm", 1)])
H2.append([g, ("ad", 1), ("ad", 1), ("sm", 1)])
H2.append([g, ("a", 1), ("a", 1), ("sp", 1)])

# construct H3
H3 = []
H3.append([ω_c − iκ/2, ("ad", 1), ("a", 1)])
for j in range(N):
    H3.append([ω_e − iγ/2, ("sp", j), ("sp", j)])
    H3.append([g[j], ("ad", 1), ("sm", j)])
    H3.append([g[j], ("a", 1), ("sp", j)])

# construct H4
H4 = []
for j in range(N):
    H4.append([ω_c − iγ/2, ("ad", j), ("a", j)])
    H4.append([U, ("ad", j), ("ad", j), ("a", j), ("a", j)])
    if j != N-1:
        H4.append([J, ("ad", j), ("a", j+1)])
        H4.append([J, ("a", j), ("ad", j+1)])
```

3.2.3 The input and output channels

According to the scattering theory, we must have to define the input and output channels in order to obtain the statistical properties of outgoing light. To proceed, we consider a standard input-output formalism that relates μ_{in}, μ_{out}, and o_μ as

$$\mu_{\text{out}}(t) = \mu_{\text{in}}(t) - io_\mu(t).$$

Obviously, the quantum operator o_μ also satisfies the symbol stipulations 3.2.2 above.

For example, we assume that the name of input and output channels are $\mu = b$ and $\mu = c$, respectively, and the quantum operators are $o_b = \sqrt{\kappa}a_1$ and $o_c = \sqrt{\gamma}\sigma_1$. Here, we consider an incoming coherent state from the input channel, which the amplitude and frequency are β and ω_d, respectively. This process is equivalent with a coherent drive, and the Hamiltonian could be written as

$$H_d = \frac{\beta a_1 \exp(i\omega_d t) + \beta^* a_1^\dagger \exp(-i\omega_d t)}{\sqrt{2\pi/\kappa}}.$$

```
In [1]: from qcs_phy import qcs

In [2]: o_b = [√κ, ("a", 1)]
In [3]: o_c = [√γ, ("sm", 1)]
In [4]: Input = qcs.Input_channel("b", o_b, ω_d)
In [5]: Output = qcs.Output_channel("c", o_c)
```

Here, we must use the two functions of the class `qcs` to construct the `Input` and `Output`, and the two functions are given by

```
qcs.Input_channel(para1, para2, para3), qcs.Output_channel(para1, para2)
```

- The first parameter (`para1`) represents the name of channel in the form of `str`, such as `"b"` and `"c"`.
- The second parameter (`para2`) represents the quantum operator o_μ in the form of `list`, such as o_b and o_c.
- The third parameter (`para3`) represents the driving frequency ω_d, and it could be [number, list, or array].

Multiple coherent state inputs

As expected, the case of multiple coherent state inputs is equivalent with the multi-mode coherent drives. For simplicity, we consider the case of two coherent state inputs, and the corresponding input channels are $\mu = b_1$ and $\mu = b_2$, respectively. Meanwhile, we assume the quantum operators are $o_{b_1} = \sqrt{\kappa_1}a_1$ and $o_{b_2} = \sqrt{\kappa_2}a_2$; the amplitudes and driving frequencies are (β_1, ω_1) and (β_2, ω_2), respectively. Thus, the driven Hamiltonian could be written as

$$H_d = \frac{\beta_1 a_1 \exp(i\omega_1 t) + \beta_1^* a_1^\dagger \exp(-i\omega_1 t)}{\sqrt{2\pi/\kappa_1}} + \frac{\beta_2 a_2 \exp(i\omega_2 t) + \beta_2^* a_2^\dagger \exp(-i\omega_2 t)}{\sqrt{2\pi/\kappa_2}}.$$

Note: For the multiple coherent state inputs, although the individual amplitude could be ignored, the ratio between amplitudes are not negligible. Therefore, we must provide a new parameter in the form of `list`, i.e., ratio $= [\beta_1, \beta_2, \cdots]$.

```
In [6]: o_b1 = [√κ_1, ("a", 1)]
In [7]: o_b2 = [√κ_2, ("a", 2)]
In [8]: Input_1 = qcs.Input_channel("b1", o_b1, ω_1) # The first input
In [9]: Input_2 = qcs.Input_channel("b2", o_b2, ω_2) # The second input
In [10]: Input = [Input_1, Input_2] # Two inputs
In [11]: ratio = [β_1, β_2] # The ratio between amplitudes
```

For the case of multiple outputs, the usage is completely identical with the multiple inputs.

3.2.4 Main body

When we construct the effective Hamiltonian `Heff`, the input channel `Input`, and the output channel `Output`, the main body has finished, and the Python code is shown below:

```
In [1]: from qcs_phy import qcs
```

```
In [2]: ... Heff
In [3]: ... Input
In [4]: ... Output
In [5]: system = qcs(Heff, Input, Output)
```

If the `Input` contains multiple coherent state inputs, we will provide the new parameter `ratio`, i.e.,

```
In [1]: from qcs_phy import qcs

In [2]: ... Heff
In [3]: ... Input
In [4]: ... ratio
In [5]: ... Output
In [6]: system = qcs(Heff, Input, Output, ratio)
```

3.3 The Higher-Order Equal-Time Correlation Function

In Sec. 3.2, we have provided the steps how to construct the effective Hamiltonian, Input, and Output. Meanwhile, we also define the name of output channel in the form of `str`, such as `"c1"` or `"c2"`. For the sake of simplicity, we assume the output channel modes are $c_1, c_2, \ldots, c_n$, and the corresponding channel names are `"c1"`, `"c2"`, ..., `"cn"`, respectively. Subsequently, we will introduce the function on `qcs` class that calculates physical quantities, i.e.,

```
system.calculate_quantity(Quantity, tlist=0, zp=0)
```

- The first parameter (`Quantity`) represents the physical quantity in the form of `str`.
- The second parameter (`tlist`) represents time, which is only given in multiple unequal driving frequencies. Default is zero.
- The third parameter (`zp`) represents the coefficient 0^+, which is only given when the system has multiple steadystates. Default is zero.

3.3.1 Steady ETCF

Here, the criteria of steady ETCF is denoted as the system reduced density matrix is time-independent after long-time evolution. In other words, if there is single coherent state input or multiple coherent state inputs with the identical driving frequencies, the system reduced density matrix will satisfy the criteria. Meanwhile, the usage of the parameter `Quantity` is shown in the table below:

Quantity	Equation	Name
`"c1c1"`	$g^{(2)}(0) = \frac{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1^\dagger(t)c_1(t)c_1(t)\|\Psi_{\rm out}\rangle}{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1(t)\|\Psi_{\rm out}\rangle^2}$	2nd-order ETCF
`"c1c1c1"`	$g^{(3)}(0) = \frac{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1^\dagger(t)c_1^\dagger(t)c_1(t)c_1(t)c_1(t)\|\Psi_{\rm out}\rangle}{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1(t)\|\Psi_{\rm out}\rangle^3}$	3rd-order ETCF
`"c1c2"`	$g_c^{(2)}(0) = \frac{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_2^\dagger(t)c_1(t)c_2(t)\|\Psi_{\rm out}\rangle}{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1(t)\|\Psi_{\rm out}\rangle\langle\Psi_{\rm out}\|c_2^\dagger(t)c_2(t)\|\Psi_{\rm out}\rangle}$	2nd-order cross ETCF
`"c1c1c2"`	$g_c^{(3)}(0) = \frac{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1^\dagger(t)c_2^\dagger(t)c_1(t)c_1(t)c_2(t)\|\Psi_{\rm out}\rangle}{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1(t)\|\Psi_{\rm out}\rangle^2\langle\Psi_{\rm out}\|c_2^\dagger(t)c_2(t)\|\Psi_{\rm out}\rangle}$	3nd-order cross ETCF
⋮		

Example: Jaynes-Cummings Model

3.3.2 Dynamical ETCF

Here, the criteria of dynamical ETCF is denoted as the system reduced density matrix is time-dependent even after long-time evolution. In other words, if there is multiple coherent state inputs

with the unequal driving frequencies, the system reduced density matrix will satisfy the criteria. Meanwhile, the usage of the parameter `Quantity` is shown in the table below:

`Quantity`	Equation	Name
`"c1c1"`	$g^{(2)}(t)=\frac{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1^\dagger(t)c_1(t)c_1(t)\|\Psi_{\rm out}\rangle}{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1(t)\|\Psi_{\rm out}\rangle^2}\neq g^{(2)}(0)$	2nd-order ETCF
`"c1c1c1"`	$g^{(3)}(t)=\frac{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1^\dagger(t)c_1^\dagger(t)c_1(t)c_1(t)c_1(t)\|\Psi_{\rm out}\rangle}{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1(t)\|\Psi_{\rm out}\rangle^3}\neq g^{(3)}(0)$	3rd-order ETCF
`"c1c2"`	$g_c^{(2)}(t)=\frac{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_2^\dagger(t)c_1(t)c_2(t)\|\Psi_{\rm out}\rangle}{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1(t)\|\Psi_{\rm out}\rangle\langle\Psi_{\rm out}\|c_2^\dagger(t)c_2(t)\|\Psi_{\rm out}\rangle}\neq g_c^{(2)}(0)$	2nd-order cross ETCF
`"c1c1c2"`	$g_c^{(3)}(t)=\frac{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1^\dagger(t)c_2^\dagger(t)c_1(t)c_1(t)c_2(t)\|\Psi_{\rm out}\rangle}{\langle\Psi_{\rm out}\|c_1^\dagger(t)c_1(t)\|\Psi_{\rm out}\rangle^2\langle\Psi_{\rm out}\|c_2^\dagger(t)c_2(t)\|\Psi_{\rm out}\rangle}\neq g_c^{(3)}(0)$	3nd-order cross ETCF
⋮		

Example: Jaynes-Cummings Model

3.3.3 The single-photon transimission and reflection

Here, we only consider a coherent state input with channel mode b_1, and the corresponding name is `"b1"`. Thus, we have the following table, i.e.,

`Quantity`	Equation	Name
`"cj"`	$T_j=\frac{\langle\Psi_{\rm out}\|c_j^\dagger(t)c_j(t)\|\Psi_{\rm out}\rangle}{\langle\Psi_{\rm in}\|b_1^\dagger(t)b_1(t)\|\Psi_{\rm in}\rangle}$	Transimission
`"b1"`	$R=\frac{\langle\Psi_{\rm out}\|b_1^\dagger(t)b_1(t)\|\Psi_{\rm out}\rangle}{\langle\Psi_{\rm in}\|b_1^\dagger(t)b_1(t)\|\Psi_{\rm in}\rangle}$	Reflection
⋮		

3.4 The Other Functions On Qcs Class

3.4.1 The second-order unequal-time correlation function

4. API documentation

5. Change Log

5.1 Version 1.0.0 (May 11,2023)