QuantumToolbox.jl: An efficient Julia framework for simulating open quantum systems

Alberto Mercurio*[1,2], Yi-Te Huang†[3,4,5], Li-Xun Cai[3,4], Yueh-Nan Chen[3,4,6], Vincenzo Savona[1,2], and Franco Nori[5,7]

[1]Institute of Physics, Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
[2]Center for Quantum Science and Engineering, Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
[3]Department of Physics, National Cheng Kung University, Tainan 701401, Taiwan
[4]Center for Quantum Frontiers of Research and Technology (QFort), Tainan 701401, Taiwan
[5]RIKEN Center for Quantum Computing, RIKEN, Wakoshi, Saitama 351-0198, Japan
[6]Physics Division, National Center for Theoretical Sciences, Taipei 106319, Taiwan
[7]Physics Department, The University of Michigan, Ann Arbor, Michigan 48109-1040, USA.

We present `QuantumToolbox.jl`, an open-source Julia package for simulating open quantum systems. Designed with a syntax familiar to users of `QuTiP` (Quantum Toolbox in Python), it harnesses Julia's high-performance ecosystem to deliver fast and scalable simulations. The package includes a suite of time-evolution solvers supporting distributed computing and GPU acceleration, enabling efficient simulation of large-scale quantum systems. We also show how `QuantumToolbox.jl` can integrate with automatic differentiation tools, making it well-suited for gradient-based optimization tasks such as quantum optimal control. Benchmark comparisons demonstrate substantial performance gains over existing frameworks. With its flexible design and computational efficiency, `QuantumToolbox.jl` serves as a powerful tool for both theoretical studies and practical applications in quantum science.

GitHub: https://github.com/qutip/QuantumToolbox.jl

1 Introduction

The simulation of quantum systems is a fundamental task in many areas of physics, including quantum optics, condensed matter physics, and quantum information science. The development of efficient numerical methods and software packages for simulating these systems has become increasingly important due to their growing complexity and the need for accurate predictions of their behavior. Although the simulation of these systems generally requires standard linear algebra operations, the computational complexity grows exponentially with the number of degrees of freedom. This makes it challenging to simulate large-scale quantum systems, especially when considering the effects of interactions with the environment.

In many-body physics, approximate techniques render intractable problems accessible. Tensor network methods exploit locality of entanglement to efficiently represent wavefunctions [1–9]. Quantum Monte Carlo utilizes stochastic sampling [10, 11], while the dynamical mean field theory approximate many-body systems as effective impurity problems [12, 13]. Moreover, variational approaches [14–18], including neural quantum states [19–21], have recently emerged as powerful methods. Additionally, quantum cumulant expansions [22, 23] and phase space representations [24] offer complementary insights into the quantum correlations and nonclassical features of the system.

Although these techniques have demonstrated great success, the simulation of quantum systems without approximation remains an important challenge. Therefore, exploring the computational efficiency of different implementations on different programming languages optimized for various hardware architectures is crucial. Since 1999, several software packages have been developed to this purpose, including `QOToolbox` [25] in Matlab [26], `QuTiP` [27–30] and `dynamiqs` [31] in Python [32], as well as `QuantumOptics.jl` [33] in Julia [34, 35].

A common limitation of most existing Python-based packages is their strict dependency on a specific numerical backend (e.g., `SciPy` or `JAX`). The addition of GPU and automatic differentiation support has required major restructuring of these code bases. Moreover, while the JAX backend provides excellent automatic differentiation, it restricts computations to JAX-compatible operations, limiting flexibility in data structures and control flows. These architectural choices imply that, if a new and superior numerical backend emerges in the future, adopting

Alberto Mercurio*: alberto.mercurio96@gmail.com, These authors contributed equally.

Yi-Te Huang†: yitehuang.tw@gmail.com, These authors contributed equally.

it would again require substantial refactoring of the code.

In this work, we introduce `QuantumToolbox.jl` (Quantum Toolbox in Julia), a fully open-source software package written in Julia [34, 35]. The package has been designed from the ground up to overcome these structural limitations, providing a backend-agnostic and extensible platform for simulating open quantum systems. The package can be used with a syntax similar to that of `QuTiP` and relies on the Julia design philosophy: *one can have machine performance without sacrificing human convenience* [34]. It combines the simplicity and efficiency of Julia with advanced features like distributed computing [35] and GPU acceleration [36], making simulation of quantum systems more accessible and efficient. Furthermore, by wrapping some functions from other Julia packages [37, 38], we could further optimize the computational efficiency of simulations and achieve a significant speedup with respect to the corresponding methods in other packages [27, 28, 30, 31, 33].

Julia is a high-level, dynamic programming language designed for high-performance scientific computing. It was created in 2012 with the goal to combine the speed of C [39], the flexibility of Ruby [40], the ease of use of Python [32], and the statistical and numerical computing capabilities of R [41] and Matlab [26]. Its performance advantage comes from a just-in-time (JIT) compiler [42], which translates source code into machine code before execution, ensuring both efficiency and speed. Recently, Julia has gained increasing attention in research communities, particularly in fields requiring intensive computation, such as open quantum system dynamics [23, 33, 43–45], quantum algorithms [46], and quantum information [47]. Moreover, Julia supports UTF-8 variable names, allowing the usage of Greek and mathematical symbols, and thus, making the equations in code more readable and intuitive.

One can easily install `QuantumToolbox.jl` by running the following commands inside Julia's interactive session:

```
import Pkg
Pkg.add("QuantumToolbox")
```

To load the package and check the version information, one can run the following commands:

```
using QuantumToolbox
QuantumToolbox.versioninfo()
```

Note that all examples and code snippets presented in the following sections are tested with `QuantumToolbox.jl` version 0.34.1. The entire ecosystem of `QuantumToolbox.jl` package is summarized in Fig. 1.

2 Package architecture and design philosophy.

In this Section, we summarize the main structure of the package and the most relevant user-accessible functions of `QuantumToolbox.jl`. A schematic is shown in Fig. 1 and a complete list of available functions is shown in Appendix C. By taking advantage of the Julia programming language features (e.g., multiple dispatch, metaprogramming, and distributed computing [34, 35]) and other powerful Julia packages [36–38], simulating large-scale open quantum system dynamics becomes very easy and efficient.

As shown in Fig. 1(a), the `QuantumObject` (or its abbreviated synonym `Qobj`) serves as a fundamental structure in `QuantumToolbox.jl`, representing quantum states, operators, and superoperators in a unified manner. The design of `Qobj` is inspired by similar abstractions found in `QuTiP` and also leverages Julia's multiple dispatch feature, allowing standard arithmetic operations and other advanced features for `Qobj`.

The `Qobj` structure is composed of three fields: `type`, `dimensions` (or `dims`), and `data`. The `type` field specifies whether a `Qobj` represents a quantum `Ket`, `Bra`, `Operator`, `SuperOperator`, `OperatorKet`, or `OperatorBra`, enabling intuitive manipulation of these objects. The `dimensions` (or `dims`) field stores the information about the Hilbert space of each subsystem in a composite system, ensuring compatibility with tensor products and partial traces on multipartite systems.

The `data` field contains the numerical representation of the quantum object. It stores the underlying data as a generic `AbstractArray`-type, allowing for broad compatibility with different computational backends. Depending on the nature of the quantum object and users' computational requirements, the `data` field can hold:

- Standard Julia dense or sparse arrays
- GPU dense or sparse arrays [36]
- Statically sized arrays [48]
- Any other custom array that conforms to Julia's `AbstractArray` type.

This feature makes the `Qobj` highly adaptable across different platforms and architectures, allowing efficient large-scale quantum simulations without modifying the high-level application programming interface (API).

The `QuantumObjectEvolution` (or its abbreviated synonym `QobjEvo`) structure is introduced to encode time-dependent quantum objects for time evolution simulations. It shares the same fields as `Qobj`, with the `data` field leveraging the package

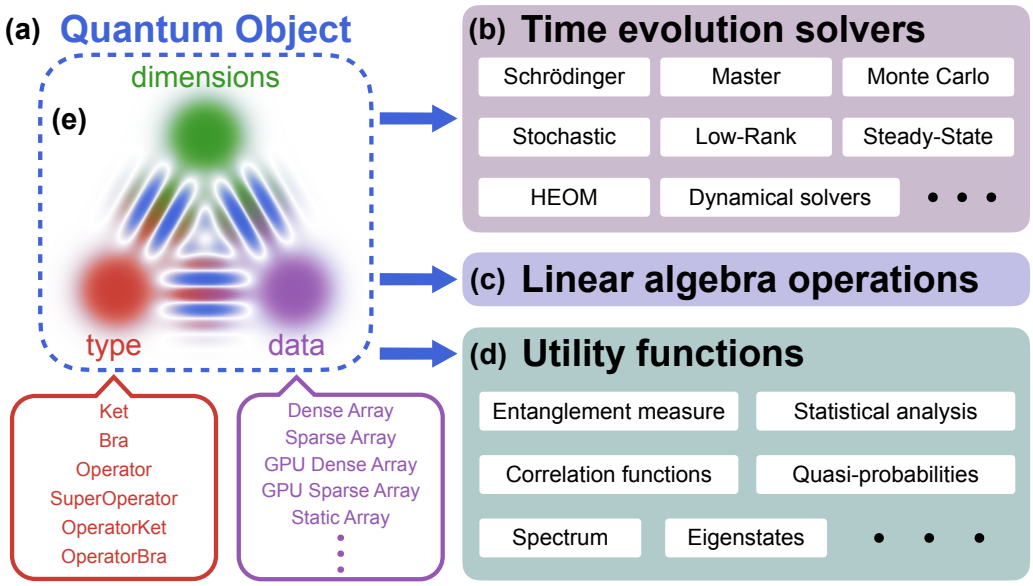

Figure 1: **Ecosystem of the `QuantumToolbox.jl` package. (a)** The quantum object structure encapsulates the properties of generic quantum objects through three main fields: `type`, `dimensions`, and `data`. The `type` field specifies the category of the quantum object, the `dimensions` field defines the Hilbert space structure for composite systems, and the `data` field supports arbitrary array types in Julia. **(b)** Various time evolution solvers are available for studying open quantum system dynamics. **(c)** All quantum objects support fundamental arithmetic and linear algebra operations. **(d)** `QuantumToolbox.jl` also provides a variety of utility functions for analyzing the properties and physical quantities of a given quantum object. **(e)** Logo of the `QuantumToolbox.jl` package.

`SciMLOperators.jl` [49], which is one of the core packages of `DifferentialEquations.jl` [37].

We demonstrate the usage of `Qobj` and `QobjEvo` by constructing the following time-dependent Hamiltonian for a single spin-1/2 system:

$$\hat{H}(t) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} + \cos(\omega_1 t) \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + \sin(\omega_2 t) \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \tag{1}$$

This Hamiltonian can be generated by the following code:

```
# time-dependent coefficient functions
## parameters passed as Vector
p = [ω1, ω2]
f1(p, t) = cos(p[1] * t)
f2(p, t) = sin(p[2] * t)

## or

## parameters passed as NamedTuple
p = (ω1 = ω1, ω2 = ω2)
f1(p, t) = cos(p.ω1 * t)
f2(p, t) = sin(p.ω2 * t)

# Pauli matrices
## same as calling sigmax(), sigmay(), and sigmaz()
σx=Qobj([0  1.0;   1.0   0], type=Operator(), dims=2)
σy=Qobj([0 -1.0im; 1.0im 0], type=Operator(), dims=2)
σz=Qobj([1.0  0;   0  -1.0], type=Operator(), dims=2)

H = QobjEvo((σz, (σx, f1), (σy, f2)))

H(p, 0.1)  # returns the Hamiltonian at t = 0.1
```

A time-dependent quantum object can be built by calling the function `QobjEvo` and providing one or more time-independent `Qobj` along with corresponding time-dependent coefficient functions. The type of parameter `p` for the coefficient function can be either a `Vector` or `NamedTuple`. Notice that the Pauli matrices in this example can be more simply constructed by calling the functions `sigmax()`, `sigmay()`, and `sigmaz()`. `QuantumToolbox.jl` provides many functions for constructing typical quantum states and operators, as listed in Table 3.

Figure 1(b) shows that `QuantumToolbox.jl` can solve a wide range of time evolution problems, including deterministic and stochastic dynamics, variational low-rank [17] and hierarchical equations of motion (HEOM) [44, 50–57]. A summary of the available time evolution solvers is provided in Table 4.

By utilizing the `DifferentialEquations.jl` [37] package, which provides a set of low-level solvers for ordinary and stochastic differential equations, `QuantumToolbox.jl` achieves enhanced performance and scalability for time evolution simulations. To compute stationary states, we leverage the LinearSolve.jl [38] package, which provides a unified and efficient interface to solve linear equations using various algorithms. We emphasize that solving the aforementioned problems on GPUs is straightforward by specifying the `data` field of `Qobj` as a GPU array [36]. Moreover, the stochastic (or quantum trajectory) solvers support distributed computing [35, 37], enabling parallel integration of many trajectories on

large-scale systems.

As can be seen from Fig. 1(b), HEOM can be used to study systems strongly coupled to the environment using the `HierarchicalEOM.jl` package [44]. This package is built on top of `QuantumToolbox.jl` and provides a user-friendly framework for simulating open quantum systems based on the HEOM approach [50–57]. The HEOM method enables a numerically exact characterization of all environmental effects on the system. It assumes that the environment is Gaussian and that the environmental operator in the interaction Hamiltonian couples linearly to the system, thereby preserving Gaussian statistics throughout the system-environment dynamics [50–52]. For a detailed explanation of `HierarchicalEOM.jl` package, we refer the readers to Ref. [44].

As shown in Fig. 1, `QuantumToolbox.jl` overloads most of the linear algebra operations provided by Julia's standard library `LinearAlgebra.jl` [34]. It also offers a variety of utility functions for analyzing the properties and physical quantities of `Qobj`, including the calculation of expectation values, eigen decompositions, quasi-probabilities, entanglement measurements, correlation functions, spectra, and more. The most relevant linear algebra and utility functions supported by `QuantumToolbox.jl` are summarized in Table 5.

Fig. 1(e) shows the logo of `QuantumToolbox.jl`, which is generated by the package itself. The logo represents the Wigner function [58] of a triangular cat state subject to decoherence. The state is constructed as a linear superposition of three coherent states using the `coherent` function, normalized with the `normalize` function, and evolved under the Lindblad master equation using the `mesolve` function, which we will discuss in Section 3.2. The Wigner quasi-probability is then computed using the `wigner` function.

In the following sections, we will demonstrate the features and capabilities of `QuantumToolbox.jl` with several examples. All figures are generated using the `Makie.jl` [59] plotting library, written purely in Julia. The code used to generate the figures is available in a dedicated repository [60].

3 Time evolution solvers

In this section, we demonstrate the features and capability of `QuantumToolbox.jl` with several examples.

3.1 Schrödinger equation

Given a quantum state $|\psi(0)\rangle$, the corresponding time-evolved state $|\psi(t)\rangle$ is given by the Schrödinger equation ($\hbar$ is set to unity throughout this work)

$$i\frac{d}{dt}|\psi(t)\rangle = \hat{H}|\psi(t)\rangle\,, \tag{2}$$

where $\hat{H}$ is the Hamiltonian operator. Aside from a few solutions that can be found analytically, the Schrödinger equation is usually solved numerically. Thanks to the `sesolve` function, `QuantumToolbox.jl` provides a simple way to solve the Schrödinger equation for a given Hamiltonian and initial state.

As a pedagogical example, let us consider the Jaynes-Cummings (JC) model [61], which describes the interaction between a two-level system and a mode of the electromagnetic field. The Hamiltonian of the JC model is given by

$$\hat{H} = \omega_c \hat{a}^\dagger \hat{a} + \frac{\omega_a}{2}\hat{\sigma}_z + g(\hat{a}\hat{\sigma}_+ + \hat{a}^\dagger \hat{\sigma}_-) \tag{3}$$

where ω_c and ω_a are the frequencies of the cavity and the atom, respectively, g is the coupling strength, and $\hat{a}$, $\hat{a}^\dagger$, and $\hat{\sigma}_j$ are the annihilation, creation and Pauli operators, respectively.

Consider the case where $\omega_c = \omega_a = 1$, $g = 0.1$, and the initial state is $|\psi(0)\rangle = |0\rangle \otimes |g\rangle$, where $|0\rangle$ is the vacuum state of the cavity and $|g\rangle$ is the ground state of the two-level system. In the following, we will solve the Schrödinger equation for this system and compute the expectation value of the number operator $\langle \hat{a}^\dagger \hat{a}\rangle$ as a function of time. We first define the parameters and the Hamiltonian of the JC model.

```
ωc = 1
ωa = 1
g = 0.1

# cavity
N = 10 # Cutoff of the cavity Hilbert space
a = tensor(destroy(N), qeye(2)) #annihilation operator

# atom
σz = tensor(qeye(N), sigmaz()) # Pauli-Z operator
σm = tensor(qeye(N), sigmam()) # annihilation operator
σp = tensor(qeye(N), sigmap()) # creation operator

H = ωc * a' * a + ωa/2 * σz + g * (a * σp + a' * σm)
```

The time evolution can be simply simulated by

```
# Initial cavity vacuum state and atomic excited state
ψ0 = tensor(fock(N, 0), basis(2, 0))

e_ops = [a' * a] # average photon number

tlist = range(0, 10*π/g, 1000)
sol_se = sesolve(H, ψ0, tlist, e_ops=e_ops)
sol_se.expect # get expectation values
```

Note how the syntax is mostly similar to the one of `QuTiP`. The expectation value $\langle \hat{a}^\dagger \hat{a}\rangle$ at each time point in `tlist` can be extracted by `sol_se.expect[1,:]`.

As can be seen from Fig. 2(a), the number of photons oscillates between the cavity and the atom, showing the vacuum Rabi oscillations. Its is important to note that the memory allocated does not depend on the number of points in the time grid or the final time. This is because the solver uses the most efficient algorithms given by the `DifferentialEquations.jl` package [37], computing intermediate steps in-place.

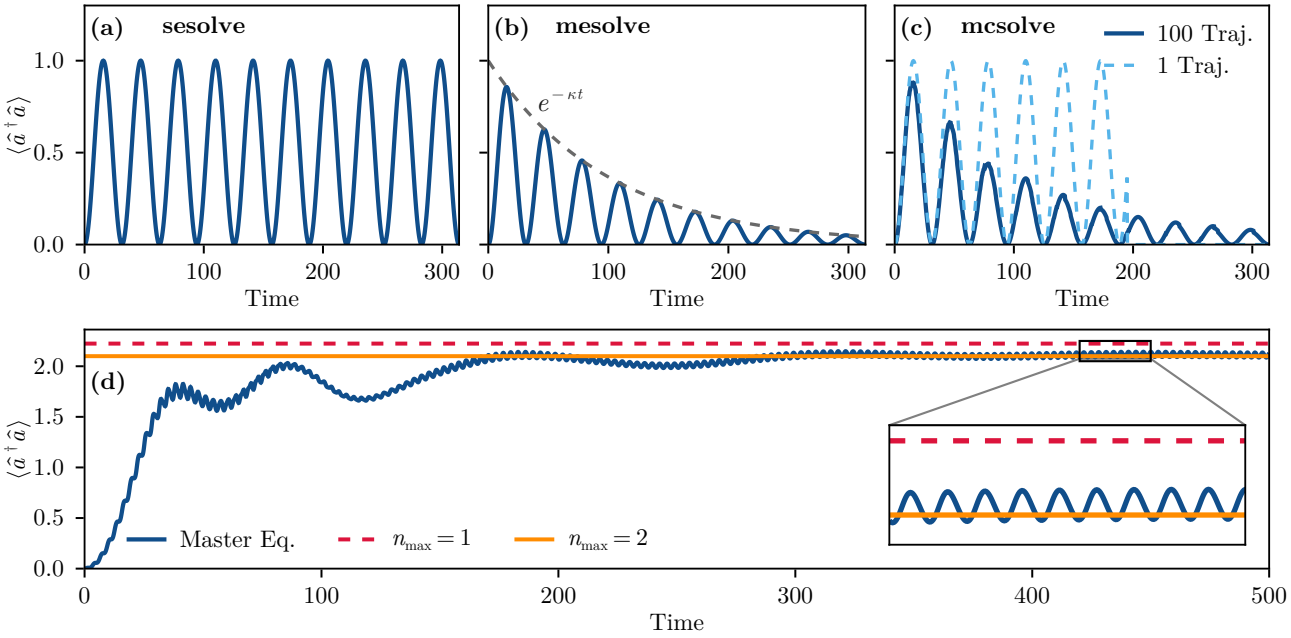

Figure 2: **Examples of time evolution solvers.** We observe the average photon number of the cavity $\langle \hat{a}^\dagger \hat{a} \rangle$ with respect to time in the Jaynes-Cummings model (**(a)**, **(b)** and **(c)**) and in the optomechanical system (**(d)**). **(a)** shows the time evolution following the Schrödinger equation, while **(b)** and **(c)** show the time evolution of the Lindblad master equation and the Monte Carlo wave-function, respectively. The dashed light blue curve in **(c)** represents a single quantum trajectory, where a quantum jump occurs at $t \approx 200$, and the solid blue curve represents the average over 100 quantum trajectories. **(d)** shows the time evolution of the optomechanical system, where the cavity is driven by a time-dependent drive. The dashed red and solid orange curves represent the value of the cavity population at the time-averaged steady state using $n_{\max} = 1$ and $n_{\max} = 2$ fourier components, respectively.

This method only allocates extra memory if the user requests storage of the states with the `saveat` keyword argument, specifying the times at which the states are saved. Note that, when `e_ops` is not defined, `saveat=tlist`.

3.2 Lindblad master equation

The Schrödinger equation is a unitary evolution equation, which describes the dynamics of a closed quantum system. However, in many cases, the system is not isolated as it interacts with an environment. In the case of a Markovian environment, the dynamics of the system is described by the Lindblad master equation [62]

$$\frac{d}{dt}\hat{\rho} = \mathcal{L}\hat{\rho} = -i[\hat{H}, \hat{\rho}] + \sum_k \mathcal{D}[\hat{C}_k]\hat{\rho}\,, \tag{4}$$

where $\mathcal{L}$ is the Liouvillian superoperator, $\hat{\rho}$ is the density matrix of the system, $\hat{H}$ is the Hamiltonian operator, $\hat{C}_k$ are the collapse operators, and

$$\mathcal{D}[\hat{C}]\hat{\rho} = \hat{C}\hat{\rho}\hat{C}^\dagger - \frac{1}{2}\left(\hat{C}^\dagger\hat{C}\hat{\rho} - \hat{\rho}\hat{C}^\dagger\hat{C}\right) \tag{5}$$

is the Lindblad dissipator. `QuantumToolbox.jl` provides the `mesolve` function to solve the master equation for a given Hamiltonian, initial state, and collapse operators.

As an example, we consider the Jaynes-Cummings model, where both the cavity and the atom are coupled to a zero-temperature bath. The collapse operators are given by

$$\hat{C}_1 = \sqrt{\kappa}\hat{a}\,, \quad \hat{C}_2 = \sqrt{\gamma}\hat{\sigma}_-\,, \tag{6}$$

where κ and γ are the decay rates of the cavity and the atom, respectively. We solve the master equation for this system and compute the expectation value of the number operator $\langle \hat{a}^\dagger \hat{a} \rangle$ as a function of time.

```
κ = 0.01
γ = 0.01

C1 = sqrt(κ) * a
C2 = sqrt(γ) * σm

c_ops = [C1, C2]

sol_me = mesolve(H, ψ0, tlist, c_ops, e_ops=e_ops)
sol_me.expect # get expectation values
```

The expectation value $\langle \hat{a}^\dagger \hat{a} \rangle$ at each time point in `tlist` can be extracted by `sol_me.expect[1,:]`.

Figure 2(b) shows the expectation value of the number operator $\langle \hat{a}^\dagger \hat{a} \rangle$ as a function of time. Contrary to the Schrödinger equation, the cavity population decays exponentially due to the cavity decay rate κ. Indeed, the dashed gray curve represents the exponential decay of the cavity population due to the cavity decay rate κ. The master equation solver is able to

capture this behavior, as well as the oscillations due to the atom-cavity coupling.

When only the steady state is of interest, the `steadystate` function can be used to directly compute the steady state, without the need to solve the master equation in time. This function supports multiple algorithms, including direct, iterative, and eigenvalue-based methods [63, 64].

3.3 Monte Carlo wave-function

The Monte Carlo quantum trajectories' approach is a stochastic method to solve the time evolution of an open quantum system [65–68]. The idea is to simulate the evolution of the system by sampling the trajectories of the quantum state through a non-unitary evolution of a pure state $|\psi(t)\rangle$. The density matrix $\hat{\rho}(t) = \overline{|\psi(t)\rangle\langle\psi(t)|}$ obtained by averaging over the statistical ensemble of the trajectories is then an estimate of the actual solution of the Lindblad Master equation. The Monte Carlo method is particularly useful when the Hilbert space of the system is large, as it avoids direct computation of the density matrix and the Liouvillian superoperator. Trajectories can be easily computed in parallel, especially when using the Distributed.jl package in Julia. `QuantumToolbox.jl` provides the `mcsolve` function to solve the open quantum system dynamics using the Monte Carlo method. The system evolves according to the Schrödinger equation expressed in Eq. (2), but with the non-Hermitian effective Hamiltonian

$$\hat{H}_{\text{eff}} = \hat{H} - \frac{i}{2}\sum_k \hat{C}_k^\dagger \hat{C}_k \,. \tag{7}$$

Being the evolution non-unitary, the norm of the state is not conserved. A quantum jump occurs when the norm of the state reaches a random threshold, previously sampled from a uniform distribution between 0 and 1. If many jump operators are present, the jump to be executed is sampled from the probability distribution

$$P_k(t) = \frac{\langle\psi(t)|\hat{C}_k^\dagger \hat{C}_k|\psi(t)\rangle}{\sum_j \langle\psi(t)|\hat{C}_j^\dagger \hat{C}_j|\psi(t)\rangle} \,. \tag{8}$$

We consider the same system as before, where both the cavity and the atom are coupled to a zero-temperature bath. In addition to other keyword arguments, here we also use the `ntraj` and `rng` arguments to specify the number of quantum trajectories and the random seed, respectively. The second argument is optional, but it is useful for reproducibility.

```
using Random
rng = MersenneTwister(10)

sol_mc = mcsolve(H, ψ0, tlist, c_ops, e_ops=e_ops,
    ntraj=100, rng=rng)
sol_mc.expect # get expectation values
```

The average expectation value $\langle\hat{a}^\dagger\hat{a}\rangle$ at each time point in `tlist` can be extracted by `sol_mc.expect[1,:]`.

Fig. 2(c) shows the expectation value of the number operator $\langle\hat{a}^\dagger\hat{a}\rangle$ as a function of time. As can be seen, by averaging over many trajectories, we get the same results as the master equation.

3.4 Time-dependent Lindblad master equation

In many models of interest, the Hamiltonian and-or the collapse operators are time-dependent. For example, this can happen when a drive is applied to the system or when dynamically changing the resonance frequency. Here, we show how the solvers can be used to simulate systems with time-dependent Hamiltonian or collapse operators. As an example, we consider the driven optomechanical system [69–71]

$$\begin{aligned}\hat{H} = \omega_c \hat{a}^\dagger\hat{a} + \omega_m \hat{b}^\dagger\hat{b} + \frac{g}{2}\left(\hat{a}+\hat{a}^\dagger\right)^2\left(\hat{b}+\hat{b}^\dagger\right) \\ + F\cos\left(\omega_{\text{d}} t\right)\left(\hat{a}+\hat{a}^\dagger\right) \,,\end{aligned} \tag{9}$$

where $\hat{a}$ and $\hat{b}$ are the annihilation operators of the cavity and mechanical mode, respectively. The parameter ω_m represents the mechanical frequency, while F and ω_{d} denote the amplitude and frequency of the drive, respectively. It is worth noting that the time-dependence can not be removed by a unitary transformation, as the Hamiltonian contains counter-rotating terms on both the coupling and the drive. The time evolution is solved as before, with Hamiltonian being either a `Tuple` or a `QuantumObjectEvolution` type. Finally, we solve the open dynamics of the system by considering the cavity decay and the mechanical damping, with the collapse operators $\mathcal{D}[\hat{a}]$ and $\mathcal{D}[\hat{b}]$, respectively.

```
Nc = 10 # Cutoff of the cavity Hilbert space
Nm = 7  # Cutoff of the mechanical mode Hilbert space
ωc = 1
ωm = 2 * ωc
g = 0.05
κ = 0.01
γ = 0.01
F = 10*κ
ωd = ωc

# time-dependent coefficient function
coef(p, t) = p[1] * cos(p[2] * t)

# annihilation operators
a = tensor(destroy(Nc), qeye(Nm)) # cavity
b = tensor(qeye(Nc), destroy(Nm)) # mechanical mode

H = ωc * a' * a + ωm * b' * b +
    g / 2 * (a + a')^2 * (b + b')
c_ops = [sqrt(κ) * a, sqrt(γ) * b]
e_ops = [a' * a]
H_td = (H, (a + a', coef))

# Zero bare cavity and mechanical excitations
ψ0 = tensor(fock(Nc, 0), fock(Nm, 0))

params = [F, ωd]
```

```
tlist2 = range(0, 5/κ, 5000)
sol_me_td = mesolve(H_td, ψ0, tlist2, c_ops, e_ops=
    e_ops, params=params)
sol_me_td.expect # get expectation values
```

Note that, as we mentioned in Sec. 2, the `params` argument is also supported as a `NamedTuple` instead of a `Vector`. The evolution of the expectation value $\langle \hat{a}^\dagger \hat{a} \rangle$, extracted by `sol_me_td.expect[1,:]`, is shown in Fig. 2(d), where we can observe a stroboscopic behavior of the steady state due to the counter-rotating terms on both the coupling and the drive. This results in the impossibility of using the `steadystate` function to compute the steady state. To this end, `QuantumToolbox.jl` offers a solution by providing the `steadystate_fourier` function. The method extracts the time-averaged steady state by expanding it into Fourier components. Depending on the internal solver, it then solves a matrix continuation fraction problem or a linear system [72–75]. The case of the linear system solving approach is discussed in Appendix A.

Although we applied this method to the master equation, the same approach can be used for all the time evolution solvers of `QuantumToolbox.jl`.

3.5 Stochastic processes under continuous measurements

Continuous quantum measurements such as homodyne and heterodyne detection describe the gradual acquisition of information about a quantum system in real time, leading to stochastic dynamics governed by measurement backaction [76]. These processes are fundamental for modeling measurement-based state estimation, e.g., the qubit readout [77–80]. In `QuantumToolbox.jl`, we have implemented the stochastic evolution of open quantum systems under homodyne detection, where a single quadrature of the field is continuously monitored. This corresponds to solving stochastic Schrödinger or master equations driven by Wiener noise, following the formalism used in `QuTiP`. The homodyne measurement allows access to real-time information about the system and is widely used in cavity and circuit QED experiments.

3.5.1 Stochastic Schrödinger equation

The stochastic Schrödinger equation (SSE) describes the time evolution of a quantum system under homodyne detection. Following this protocol, a detector does not resolve single photons but rather produces a homodyne current J_x [76]. The SSE is defined in Itô form by the following stochastic differential equation [76]

$$d|\psi(t)\rangle = -i\hat{K}|\psi(t)\rangle dt + \sum_n \hat{M}_n |\psi(t)\rangle dW_n(t)\,, \quad (10)$$

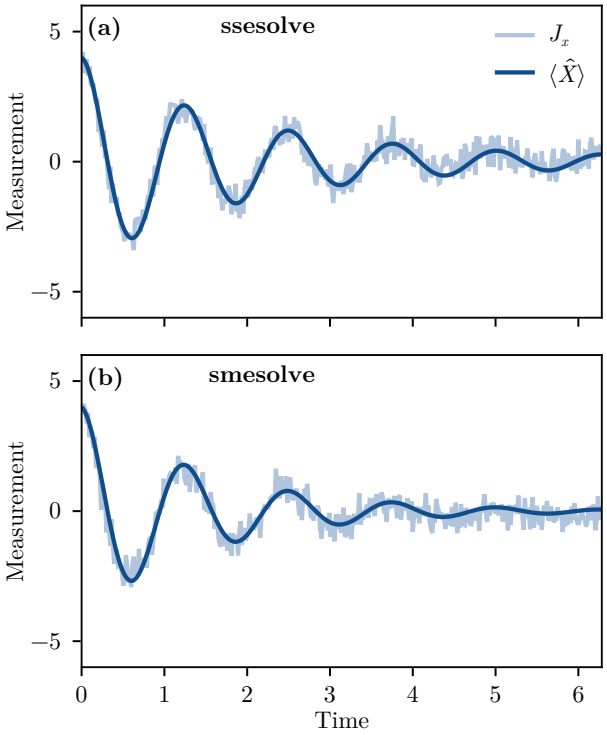

Figure 3: **Stochastic Schrödinger equation and stochastic master equation. (a)** Time evolution under the stochastic Schrödinger equation of the homodyne current J_x (light blue curve) and the expectation value $\langle \hat{X} \rangle$ (dark blue curve) for the Jaynes-Cummings model. **(b)** Time evolution under the stochastic master equation of the same quantities as in **(a)**. Both the homodyne current and the expectation value are averaged over 500 trajectories.

where

$$\hat{K} = \hat{H} + i\sum_n \left(\frac{e_n}{2}\hat{S}_n - \frac{1}{2}\hat{S}_n^\dagger \hat{S}_n - \frac{e_n^2}{8} \right)\,, \quad (11)$$

$$\hat{M}_n = \hat{S}_n - \frac{e_n}{2}\,, \quad (12)$$

$$e_n = \langle \psi(t)|\hat{S}_n + \hat{S}_n^\dagger|\psi(t)\rangle\,. \quad (13)$$

Here, $\hat{H}$ is the Hamiltonian, $\hat{S}_n$ are the stochastic collapse operators, and $dW_n(t)$ is the real Wiener increment (associated to $\hat{S}_n$) with the expectation values of $E[dW_n] = 0$ and $E[dW_n^2] = dt$, with dt being the infinitesimal time step.

As an example, we consider the JC Hamiltonian defined in Eq. (3), where we perform homodyne detection on the cavity field $\hat{S} = \sqrt{\kappa}\hat{a}$. This protocol gives us access to the quadrature $\hat{X} = \hat{S} + \hat{S}^\dagger$ of the cavity field. We initialize the system with the cavity in a coherent state and the qubit in the excited state, $|\psi(0)\rangle = |\alpha, e\rangle$. The solver `ssesolve` will construct the operators $\hat{K}$ and $\hat{M}_n$, once the user passes the Hamiltonian `H` and the stochastic collapse operators list `sc_ops`. The homodyne current J_x can be computed using [76]

$$J_x = \langle \hat{X} \rangle + \frac{dW}{dt}\,. \tag{14}$$

```
N = 20       # Fock space dimension
ωc = 5       # cavity resonance frequency
ωq = 5       # qubit resonance frequency
g = 0.1      # coupling strength
κ = 1        # cavity decay rate
γ = 2        # qubit decay rate
α = 2        # coherence of initial state
ntraj = 500 # number of trajectories

tlist = range(0, 10π / ωc, 400)

# operators
a = tensor(destroy(N), qeye(2))
σm = tensor(qeye(N), sigmam())
σz = tensor(qeye(N), sigmaz())

H = ωc * a' * a + ωq * σz / 2 +
    g * (a' * σm + a * σm')

# initial state |α, e>
ψ0 = tensor(coherent(N, α), basis(2, 0))

# stochastic collapse operators
sc_ops = [sqrt(κ) * a]

X = (a + a') * sqrt(κ)

sol_sse = ssesolve(
    H,
    ψ0,
    tlist,
    sc_ops[1],
    e_ops = [X],
    ntraj = ntraj,
    store_measurement = Val(true),
)
sol_sse.expect      # get expectation values
sol_sse.measurement # get homodyne current
```

The averaged expectation value $\langle \hat{X} \rangle$ can be extracted as in the previous examples with `sol_sse.expect`, while the homodyne current can be obtained using `sol_sse.measurement`. Figure 3(a) shows the time evolution of the homodyne current J_x (light blue curve) and the expectation value $\langle \hat{X} \rangle$ (dark blue curve), averaged over 500 trajectories. It is worth mentioning that if `sc_ops` is a single `QuantumObject` rather than a `Vector` or a `Tuple`, the `ssesolve` function will use a different solver, specific for a single Wiener process. Indeed, when giving a list of operators, the solver uses an algorithm supporting multiple Wiener processes (non-diagonal noise). It is recommended to insert a single operator, when applicable, as this improves both the performance and the stability of the solver.

3.5.2 Stochastic master equation

The stochastic master equation (SME) describes the evolution of the density matrix of a quantum system under continuous measurements. It can also take into account losses and dephasing not related to the measurement process, or inefficient measurement channels. The stochastic master equation in the Itô form is given by [76]

$$\begin{aligned} d\rho(t) = & -i[\hat{H}, \rho(t)]dt + \sum_i \mathcal{D}[\hat{C}_i]\rho(t)dt \\ & + \sum_n \mathcal{D}[\hat{S}_n]\rho(t)dt \\ & + \sum_n \mathcal{H}[\hat{S}_n]\rho(t)dW_n(t)\,, \end{aligned} \tag{15}$$

where

$$\mathcal{H}[\hat{O}]\rho = \hat{O}\rho + \rho\hat{O}^\dagger - \mathrm{Tr}[\hat{O}\rho + \rho\hat{O}^\dagger]\rho\,. \tag{16}$$

The equations now take into account also the presence of additional loss channels $\hat{C}_i$ not related to the measurement processes $\hat{S}_n$. The stochastic master equation can be solved using the `smesolve` function, which takes the same arguments as the `ssesolve` function, but also requires the list of collapse operators `c_ops`. Here, we consider the same system as before, but we also include the qubit decay channel $\hat{C}_1 = \sqrt{\gamma}\hat{\sigma}_-$ and cavity pure dephasing $\hat{C}_2 = \sqrt{\kappa_\varphi}\hat{a}^\dagger\hat{a}$.

```
# Including qubit losses
κϕ = 0.3    # cavity dephasing rate
c_ops  = [sqrt(γ) * σm, sqrt(κϕ) * a' * a]

sol_sme = smesolve(
    H,
    ψ0,
    tlist,
    c_ops,
    sc_ops[1],
    e_ops = [X],
    ntraj = ntraj,
    store_measurement = Val(true)
)
sol_sme.expect      # get expectation values
sol_sme.measurement # get homodyne current
```

As in the previous case, it is possible to extract the expectation value $\langle \hat{X} \rangle$ and the homodyne current J_x from the `sol_sme` object. Also, a simpler algorithm specific for single Wiener process is used to solve the SME when the stochastic collapse operator is a single `QuantumObject`. Figure 3(b) shows the time evolution of the homodyne current J_x (light blue curve) and the expectation value $\langle \hat{X} \rangle$ (dark blue curve), averaged over 500 trajectories.

Support for heterodyne detection, which involves simultaneous monitoring of two orthogonal quadratures and results in a complex-valued measurement current, is planned for future releases for both the SSE and SME solvers. This will further extend the package's capabilities in simulating realistic quantum measurement scenarios and feedback schemes.

3.6 Dynamical solvers

All the time-evolution solvers described above allow the use of user-defined callbacks. Indeed, thanks to the `DiffEqCallbacks.jl`, which is one of the core

packages of `DifferentialEquations.jl` [37], it is possible to apply a given operation to the state of the system, triggered by a condition that is verified along the dynamics. To demonstrate the power of callbacks, here we show the dynamical Fock dimension (DFD) and dynamical shifted Fock (DSF) algorithms. The first is a simple extension of `mesolve` with an additional callback that continuously monitors the population of the Fock states in time, increasing or decreasing the cutoff dimension of the Hilbert space accordingly. In this way, it is no longer required to check for the convergence of the results as a function of a fixed cutoff in the Hilbert space dimension. This method is accessible through the `dfd_mesolve`.

Here, we focus on the DSF method, which is a simple but powerful algorithm that allows the simulation of strongly-driven systems. Indeed, the numerical simulation of strongly-driven, nonlinear, and coupled systems presents several challenges. On one hand, the dynamics of such systems may necessitate the description of states that occupy a significant portion of the Hilbert space, making them difficult to simulate using standard numerical techniques. On the other hand, although semiclassical approximations (e.g., mean-field or cumulant expansion [22–24]) partially address this issue, they may fail to accurately characterize quantum fluctuations. The DSF is a numerical algorithm designed to efficiently tackle these challenges. A detailed description of the algorithm is provided in Appendix B, while here we limit ourselves to a brief overview of the working principle. We apply DSF to the Lindblad master equation, but the same approach can also be used to integrate Monte-Carlo quantum trajectories.

The DSF algorithm efficiently simulates strongly driven systems by maintaining a low-dimensional Hilbert space. This is achieved through monitoring the coherence α and applying unitary transformations once a specified threshold value of the coherence is reached. This approach can be seen as a continuous shift of the origin of the phase space, which is continuously re-centered at α. In order to demonstrate the full potential of the DSF algorithm, we consider a system of two harmonic oscillators with a nonlinear interaction term, a Kerr nonlinearity, and both one- and two-photon losses. More specifically, the Hamiltonian reads

$$\begin{aligned}\hat{H} = &\Delta_1 \hat{a}_1^\dagger \hat{a}_1 + \Delta_2 \hat{a}_2^\dagger \hat{a}_2 - U \hat{a}_2^{\dagger 2} \hat{a}_2^2 \\ &+ J\left(\hat{a}_1 \hat{a}_2^{\dagger 2} + \hat{a}_1^\dagger \hat{a}_2^2\right) + F(\hat{a}_1 + \hat{a}_1^\dagger)\,,\end{aligned} \tag{17}$$

with dissipators $\mathcal{D}[\hat{a}_1]$, $\mathcal{D}[\hat{a}_2]$, and $\mathcal{D}[\hat{a}_2^2]$. Here Δ_1 and Δ_2 are the detunings with respect to the drive frequency of the first and second mode, respectively; U is the Kerr nonlinearity of the second mode; J is the nonlinear coupling strength; and F is the drive amplitude. The system evolves according to the Lindblad master equation, as shown in Eq. (4).

When $J = 0$, the first mode is driven and dissipative but remains in a coherent state. For $J \neq 0$, nonlinear effects become relevant, causing the state of the first mode to deviate from a perfect coherent state while still remaining localized in phase space around it, with quantum fluctuations shaping the dynamics. Under suitable parameters, this model can generate cat states in the second mode. Such states cannot directly benefit from the DSF algorithm, since their coherence vanishes and they significantly delocalize in phase space. However, the DSF algorithm allows the user to selectively choose which subsystems should be displaced and which should be treated in the original Fock basis. This flexibility is a key advantage compared to methods such as the quantum cumulant expansion [22], where quantum fluctuations must be expanded for the entire system.

The DSF algorithm is implemented in `QuantumToolbox.jl` through the function `dsf_mesolve`. The user can specify a threshold for the coherence, a maximum cutoff dimension for the fluctuations, and numerical parameters such as the Krylov subspace dimension used to efficiently compute the action of the displacement operator. In order to support multipartite systems, the argument `op_list` must be a list of operators and each element representing the bosonic annihilation operator of a subsystem on which the DSF algorithm is applied. Because these operators evolve during the solving process, the syntax requires specifying the Hamiltonian, collapse operators, and observables as functions of `op_list`. For example, we demonstrate the case where we apply DSF algorithm to the first mode and the second mode remain in Fock basis.

```
F   = 7.0
Δ1  = 0.5
Δ2  = 0.5
U   = 0.05
κ1  = 1.0
κ2  = 0.0
κ2_2 = 0.01
J   = 0.05

tlist = range(0, 15, 1000)

# Hamiltonian
function H(op_list, p)
    a1 = op_list[1]
    Δ1 * a1'*a1 + Δ2 * a2'*a2 - U * (a2^2)' * a2^2 +
     F * (a1 + a1') + J * (a1'*a2^2 + a1*(a2^2)')
end

# Collapse operators
function c_ops(op_list,p)
    a1 = op_list[1]
    [sqrt(κ1) * a1, sqrt(κ2) * a2, sqrt(κ2_2) * a2^2]
end

# Expectation operators
function e_ops(op_list,p)
    a1 = op_list[1]
    [a1' * a1, a2' * a2, a1, a2]
end
```

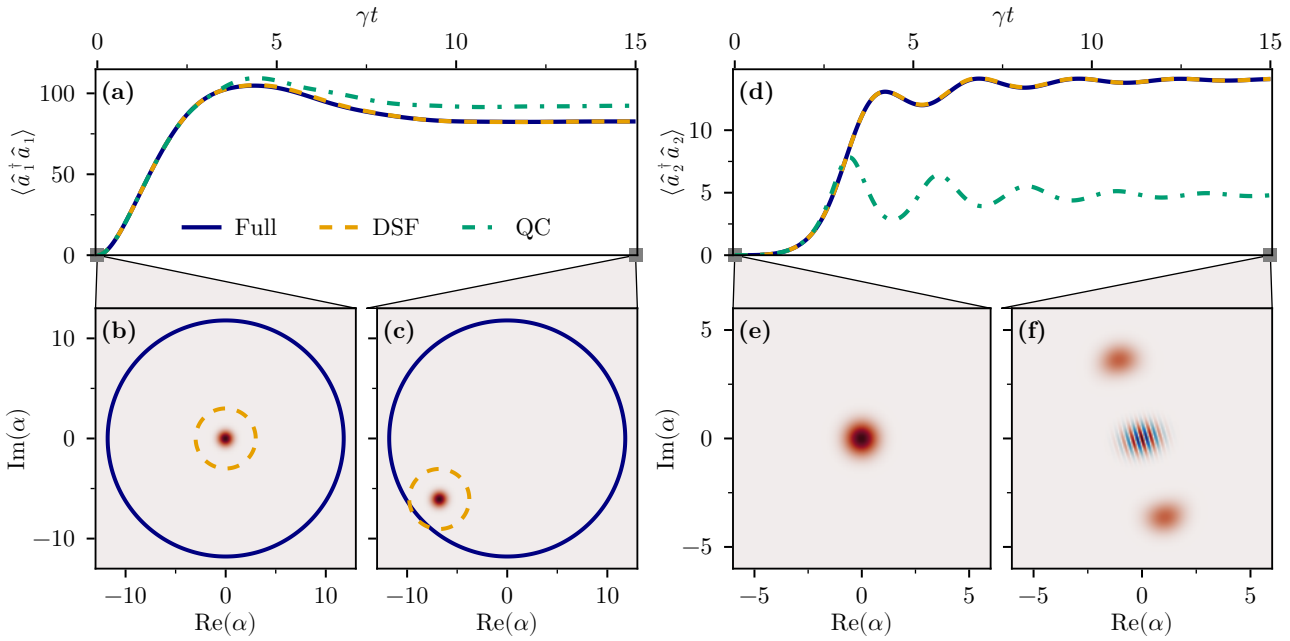

Figure 4: **Dynamical shifted Fock algorithm. (a)** Time evolution of the average photon number $\langle \hat{a}_1^\dagger \hat{a}_1 \rangle$ for two nonlinearly coupled harmonic oscillators with Kerr nonlinearity. In contrast to the second-order quantum cumulant expansion (dash-dotted green curve), the DSF algorithm (dashed orange curve) is able to capture the system's dynamics while still maintaining a low-dimensional Hilbert space in the shifted Fock basis. **(b-c)** Wigner function of the first cavity field at the initial time (**(b)**), and at the final time (**(c)**). The solid blue circle represents the Hilbert space required to simulate the model using `mesolve`, while the dashed orange circle represents the Hilbert space used by the DSF algorithm. As can be seen, the center of the space is shifted, depending on the coherence of the state. **(d)** Average photon number of the second mode, which is simulated in the original Fock basis during the DSF algorithm solving process. However, the quantum cumulants expansion is not able to reproduce this behavior. **(e-f)** Wigner function of the second mode at the initial and final time, respectively. The Wigner function is computed using the `wigner` function.

```
N1 = 4 # Cutoff of the quantum fluctuations
N2 = 40 # Cutoff of the second mode

a1 = tensor(destroy(N1), qeye(N2))
a2 = tensor(qeye(N1), destroy(N2))

# Initial state representing the quantum fluctuations
# around the initial coherent state
ψ0  = kron(fock(N1, 0), fock(N2, 0))

op_list = [a1] # Annihilation operator

sol_dsf = dsf_mesolve(H, ψ0, tlist, c_ops, op_list,
     e_ops = e_ops)
sol_dsf.expect # get expectation values
```

As for the other solvers, `sol_dsf.expect` gives the expectation values of the operators.

Figure 4(a) shows the time evolution of the average photon number $\langle \hat{a}_1^\dagger \hat{a}_1 \rangle$, computed with the standard `mesolve` function (solid blue curve), with the DSF algorithm (dashed orange curve), and with the second-order quantum cumulant expansion (dash-dotted green curve). The quantum cumulant expansion is a systematic method that expresses the dynamics of operator expectation values in terms of connected correlations (cumulants), truncating higher-order terms to approximate the dynamics by a closed set of lower-order equations [22]. Here we employ `QuantumCumulants.jl` [23], which automatically generates the equations of motion and integrates them numerically.

Panels 4(b-c) show the Wigner function of the first mode at different times, illustrating its coherent-like shape. The solid blue circle indicates the Hilbert-space region required for the full `mesolve` simulation, while the dashed orange circle indicates the effective Hilbert space explored by the DSF algorithm. As expected, the displacement shifts the center of the basis, keeping the fluctuations localized. Although the first mode remains close to a coherent state, the cumulant expansion fails to reproduce the correct evolution, as nonlinear correlations induced by the second mode become important. This discrepancy is even more pronounced in Fig. 4(d), which shows the photon number of the second mode. Figures 4(e-f) display the Wigner function of the second mode, illustrating the emergence of a cat state. In this case, the DSF algorithm correctly reverts to the standard Fock basis, while the cumulant expansion completely fails to capture the non-Gaussian features.

It is worth emphasizing that the DSF algorithm does not require the state to remain strictly coherent-like. The essential condition for DSF to provide an advantage over the full Fock basis is that the quantum fluctuations remain small compared to the dis-

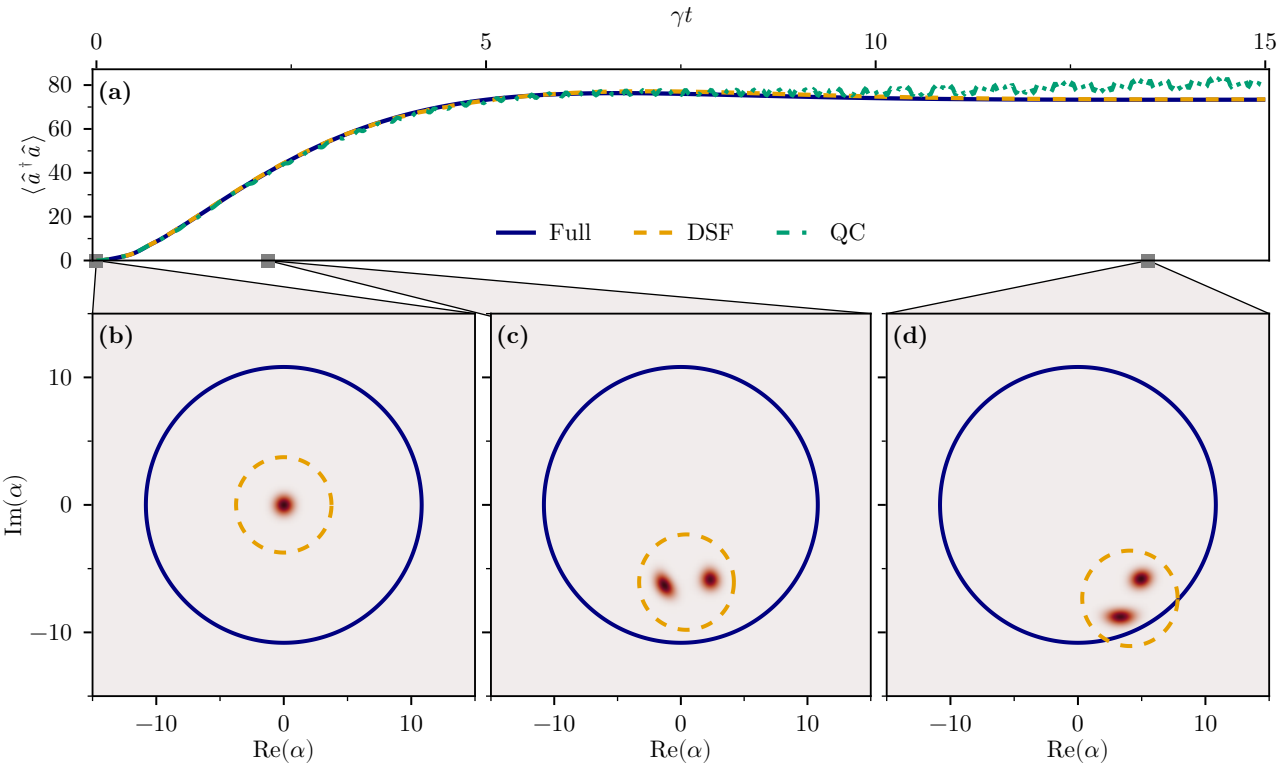

Figure 5: **Dynamical shifted Fock algorithm. (a)** Time evolution of the average photon number $\langle \hat{a}^\dagger \hat{a} \rangle$ for the driven Jaynes–Cummings model with Kerr nonlinearity. Even though the cavity state is not strictly coherent, its localization in phase space allows the DSF algorithm to reproduce the dynamics accurately, while the quantum cumulant expansion fails to capture the correct behavior. **(b–d)** Wigner functions of the cavity state at different times, illustrating its localized, non-coherent structure and the efficiency of the DSF representation with $N_{\mathrm{dsf}} = 13$.

placement in phase space. Larger fluctuations can still be handled, but they demand a correspondingly larger cutoff dimension in the fluctuation space. As an explicit example, we consider the Jaynes–Cummings model with a Kerr nonlinearity, driven at frequency ω_d with amplitude F:

$$\begin{aligned} \hat{H} = \Delta_c \hat{a}^\dagger \hat{a} + \frac{\Delta_q}{2} \hat{\sigma}_z + U \hat{a}^{\dagger 2} \hat{a}^2 \\ + g\left(\hat{a}\hat{\sigma}_+ + \hat{a}^\dagger \hat{\sigma}_-\right) + F(\hat{a} + \hat{a}^\dagger), \end{aligned} \tag{18}$$

where $\Delta_c = \omega_c - \omega_d$ and $\Delta_q = \omega_q - \omega_d$ are the detunings of the cavity and qubit with respect to the drive frequency, and U is the Kerr nonlinearity.

Figure 5(a) displays the time evolution of the cavity photon number, showing that the DSF algorithm reproduces the exact dynamics with high accuracy, while the quantum cumulant expansion deviates from the true case. Figure 5(b-d) show the corresponding Wigner functions at different times. In this case the state is not coherent, but it remains localized in phase space, requiring only a modest cutoff of $N_{\mathrm{dsf}} = 13$ for the fluctuation space.

The DSF algorithm can be also extended to the quantum trajectories method. This can have some benefits in the presence of bistable states, where the density matrix is too spread out in the Hilbert space, but the single trajectory is localized. It can be used with the `dsf_mcsolve` function, with a very similar syntax.

4 GPU and distributed computing

In recent years, graphics processing units (GPUs) have become essential tools for scientific computing due to their massive parallelism and high memory bandwidth. Originally developed for graphics rendering, GPUs gained widespread attention with the rise of deep learning [81, 82], where they dramatically accelerated training times for large neural networks. This success sparked a broader interest across various scientific domains, where parallel numerical tasks can benefit from GPU acceleration.

The field of quantum simulation is no exception. As the complexity of quantum systems scales exponentially with system size, GPU computing offers a powerful way to tackle large Hilbert spaces and intensive linear algebra operations. This has led to a growing number of quantum toolkits supporting GPU backends to enable faster and more scalable simulations.

As already mentioned in Section 2, the `QuantumObject` constructor supports any general `AbstractArray` type, including GPU arrays.

This means that the user can easily create a `QuantumObject` on the GPU by passing a GPU array to the constructor. Moreover, `QuantumToolbox.jl` has special functions that allows to convert a `QuantumObject` from the CPU to the GPU and vice versa. This is particularly useful when the user wants to perform some operations on the CPU and then transfer the result to the GPU for further processing. Here we show how easily this can be done in the case of CUDA arrays, by applying the `cu` function to a generic `QuantumObject`. For instance, the following code runs the master equation solver on the GPU:

```
using QuantumToolbox
using CUDA

N = 20 # cutoff of the Hilbert space dimension
ω = 1.0 # frequency of the harmonic oscillator
γ = 0.1 # damping rate

tlist = range(0, 10, 100) # time list

# The only difference in the code is the cu() function
a = cu(destroy(N))

H_gpu = ω * a' * a

ψ0_gpu = cu(fock(N, 3))

c_ops = [sqrt(γ) * a]
e_ops = [a' * a]

sol = mesolve(H_gpu, ψ0_gpu, tlist, c_ops, e_ops =
    e_ops)
sol.expect # get expectation values
```

This can also be applied to many other functions. For example, the computation of the Wigner function can become computationally demanding in the cases of a large Hilbert space or of a fine grid in phase space. By converting the phase space grid to a GPU array, the computation can be significantly accelerated.

```
N = 200
n_ph = 100
α = sqrt(n_ph)

ψ = normalize(coherent(N, α) + coherent(N, -α))

xvec = CuVector(range(-15, 15, 500))
yvec = CuVector(range(-15, 15, 500))

wig = wigner(ψ, xvec, yvec)
```

This can be very useful when rendering animations of the time evolution of the Wigner function, reducing the time needed to compute each frame. A performance comparison between the CPU and GPU implementations together with the comparison with other packages will be shown in Section 6.

`QuantumToolbox.jl` also supports distributed computing using Julia's built-in `Distributed.jl` package. This allows users to parallelize simulations across multiple CPU cores or nodes, which is especially useful for parameter sweeps, Monte Carlo trajectories, or ensemble averages. The framework is designed to distribute workloads efficiently while keeping the user interface simple and intuitive. Combined with GPU acceleration, this enables hybrid parallelism to tackle large and computationally demanding quantum simulations.

We now demonstrate distributed computing in the context of embarrassingly parallel problems, such as Monte Carlo trajectories and ensemble simulations. Thanks to Julia's built-in `Distributed.jl` package, these tasks can be parallelized with minimal effort. We also note that `QuantumToolbox.jl` is fully compatible with Julia's `AbstractArray` interface, which opens the possibility of using distributed array backends such as `DistributedArrays.jl` [83], `PartitionedArrays.jl` [84], or `Dagger.jl` [85] for more sophisticated distributed linear algebra, although this functionality is beyond the scope of the present demonstration.

Let us now consider the two-dimensional transverse field Ising model on a 4×4 lattice. The Hamiltonian is given by

$$\hat{H} = J_z \sum_{\langle i,j \rangle} \hat{\sigma}_i^z \hat{\sigma}_j^z + h_x \sum_i \hat{\sigma}_i^x \,, \tag{19}$$

where the sums are over nearest neighbors. We now study the evolution of the system using the quantum Monte Carlo method, under the local dissipators $\hat{c}_i = \sqrt{\gamma}\hat{\sigma}_i^-$, where γ is the decay rate. Quantum trajectories are very useful here, as they avoid the need to compute the full Liouvillian of such a large system. Moreover, they can be seamlessly parallelized by simply adding the `ensemblealg` keyword argument to the `mcsolve` function. Indeed, thanks to the Distributed.jl and SlurmClusterManager.jl packages, it is possible to parallelize on a cluster with minimal effort. The following examples are applied to a cluster with the SLURM workload manager, but the same principles can be applied to other workload managers. The script file simulating the dynamics of the system is given by

```
using Distributed
using SlurmClusterManager

const SLURM_CPUS_PER_TASK = get(ENV, "
    SLURM_CPUS_PER_TASK", 1)

exeflags = ["--project=.", "-t $SLURM_CPUS_PER_TASK"]
addprocs(SlurmManager(); exeflags=exeflags)

@everywhere begin
    using QuantumToolbox
    import SciMLBase: EnsembleSplitThreads

    BLAS.set_num_threads(1)
end

# Define lattice
Nx = 4
Ny = 4
latt = Lattice(Nx = Nx, Ny = Ny)

# Define Hamiltonian and collapse operators
Jx = 0.0
```

```
Jy = 0.0
Jz = 1.0
hx = 0.2
hy = 0.0
hz = 0.0
γ = 1

Sx = mapreduce(i->multisite_operator(latt, i=>sigmax()
    ), +, 1:latt.N)
Sy = mapreduce(i->multisite_operator(latt, i=>sigmay()
    ), +, 1:latt.N)
Sz = mapreduce(i->multisite_operator(latt, i=>sigmaz()
    ), +, 1:latt.N)

H, c_ops = DissipativeIsing(Jx, Jy, Jz, hx, hy, hz, γ,
     latt; boundary_condition = Val(:periodic_bc),
    order = 1)
e_ops = [Sx, Sy, Sz]

# Time Evolution

ψ0 = fock(2^latt.N, 0, dims = ntuple(i->2, Val(latt.N)
    ))

tlist = range(0, 10, 100)

sol_mc = mcsolve(H, ψ0, tlist, c_ops, e_ops=e_ops,
    ntraj=5000, ensemblealg=EnsembleSplitThreads())

rmprocs(workers())

sol_mc.expect # get expectation values
```

The average expectation value $\sum_i \langle \hat{\sigma}_i^x \rangle$, $\sum_i \langle \hat{\sigma}_i^y \rangle$, and $\sum_i \langle \hat{\sigma}_i^z \rangle$ at each time point in `tlist` can be extracted by `sol_mc.expect[1,:]`, `sol_mc.expect[2,:]`, and `sol_mc.expect[3,:]`, respectively.

Here, we used some helper functions to easily define the Hamiltonian and the collapse operators. The simulation of 5000 trajectories took 6 minutes on 10 nodes with 72 threads each, for a total of 720 threads on a Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40 GHz processor.

5 Automatic differentiation

In many applications, one is interested not only in simulating the dynamics of an open quantum system, but also in optimizing some cost functional that depends on the system's evolution. Typical examples include quantum optimal control, variational ansätze for open dynamics, or parameter estimation [86–89]. These tasks require the efficient computation of derivatives of observables with respect to physical parameters.

Let us consider a generic quantum system described by a time- and parameter-dependent Hamiltonian $\hat{H}(t, \mathbf{p})$, and a set of collapse operators $\hat{C}_k(t, \mathbf{p})$, where $\mathbf{p} = (p_1, \ldots, p_m)$ denotes a vector of parameters. The dynamics of the density matrix $\hat{\rho}(t, \mathbf{p})$ is governed by the Lindblad master equation in Eq. (4), namely

$$\frac{d}{dt}\hat{\rho}(t,\mathbf{p}) = -i\big[\hat{H}(t,\mathbf{p}),\, \hat{\rho}(t,\mathbf{p})\big] + \sum_k \mathcal{D}\big[\hat{C}_k(t,\mathbf{p})\big]\hat{\rho}(t,\mathbf{p})\,. \tag{20}$$

The expectation value of an observable $\hat{O}$ at a given time T is then

$$f(\mathbf{p}) = \langle \hat{O} \rangle(T,\mathbf{p}) = \mathrm{Tr}\Big[\hat{O}\,\hat{\rho}(T,\mathbf{p})\Big]\,. \tag{21}$$

In many practical scenarios, one is interested in the gradient

$$\nabla_{\mathbf{p}} f(\mathbf{p}) = \left(\frac{\partial f}{\partial p_1}, \cdots, \frac{\partial f}{\partial p_m}\right), \tag{22}$$

which quantifies how the final observable depends on the system's parameters. This gradient is essential in optimization and control, where $f(\mathbf{p})$ is used as a cost function. A straightforward approach is to approximate these derivatives using finite differences. However, this is computationally expensive, requiring at least $m+1$ full simulations, and numerically unstable, especially when $f(\mathbf{p})$ results from the integration of stiff differential equations.

Automatic differentiation (AD) provides an efficient alternative. It computes derivatives of $f(\mathbf{p})$, with respect to all parameters $\mathbf{p}$, to machine precision by applying the chain rule directly to the solver's sequence of elementary operations.

In `QuantumToolbox.jl`, we have introduced preliminary support for automatic differentiation. Many of the core functions are compatible with AD engines such as `ForwardDiff.jl` [90], `Zygote.jl` [91] and `Enzyme.jl` [92–94], allowing users to compute gradients of observables or cost functionals involving the time evolution of open quantum systems. Both forward and reverse AD are supported, in particular:

- **Forward-mode AD** (via `ForwardDiff.jl` or `Enzyme.jl`): propagates derivatives of each parameter alongside the system state during time evolution. Its cost scales linearly with the number of parameters m, making it efficient when m is small (for example, when optimizing only a few drive amplitudes or detunings).
- **Reverse-mode AD** (via `Zygote.jl` or `Enzyme.jl`): propagates sensitivities backward from the output quantity to all parameters at once. The cost scales with the number of outputs rather than the number of parameters. For scalar objectives such as $f(\mathbf{p})$ or a cost functional in optimal control, the computational cost of reverse-mode AD is essentially comparable to one (or a few) primal solves, regardless of m. This makes it the method of choice for problems with many parameters, as in pulse-shaping or

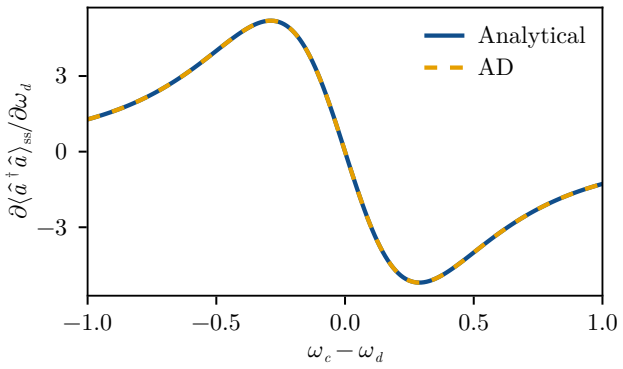

Figure 6: **Automatic differentiation.** Derivative of the expectation value $\langle\hat{a}^\dagger\hat{a}\rangle$ at the steady state with respect to the drive frequency ω_d. The solid blue curve represents the analytical derivative, while the dashed orange curve represents the derivative computed using automatic differentiation, AD.

variational optimization. Reverse-mode AD corresponds to the "backpropagation" approach widely used in machine learning.

Although `QuantumToolbox.jl` was not originally designed with AD in mind, its architecture facilitated the integration of AD capabilities. Many core functions were already compatible with AD engines out of the box.

At present, this functionality is considered experimental and not all parts of the library are AD-compatible. Nevertheless, these initial results demonstrate the flexibility of the `QuantumToolbox.jl` design and open the door to advanced applications, such as quantum optimal control and variational optimization, within the same simulation framework. Here we provide a simple example of how to use the reverse-mode AD capabilities of `QuantumToolbox.jl`, by computing the gradient of a master equation time evolution with respect to some Hamiltonian parameters.

Let us consider the case of a driven-dissipative quantum harmonic oscillator, where the Hamiltonian is given by

$$\hat{H}(t) = \omega_c\hat{a}^\dagger\hat{a} + F(\hat{a}e^{i\omega_d t} + \hat{a}^\dagger e^{-i\omega_d t}), \tag{23}$$

where ω_d is the drive frequency, and F is the drive amplitude. The system evolves according to the Lindblad master equation in Eq. (4), where the collapse operator is given by $\hat{C} = \sqrt{\gamma}\hat{a}$, with γ being the decay rate. The time evolution of the system can be computed analytically, and the number of photons at the steady state is given by

$$\langle\hat{a}^\dagger\hat{a}\rangle_{\rm ss} = \frac{F^2}{(\omega_c - \omega_d)^2 + \frac{\gamma^2}{4}}, \tag{24}$$

and its derivative with respect to the drive frequency is given by

$$\frac{\partial\langle\hat{a}^\dagger\hat{a}\rangle_{\rm ss}}{\partial\omega_d} = \frac{2F^2(\omega_c - \omega_d)}{\left[(\omega_c - \omega_d)^2 + \frac{\gamma^2}{4}\right]^2}. \tag{25}$$

Thanks to the `SciMLSensitivity.jl` [95] package, which is part of the `DifferentialEquations.jl` ecosystem, we can compute the derivative of the ordinary differential equation (ODE) using various algorithms. Here we use the adjoint sensitivity algorithm using a backwards solution of the ODE to compute the gradient with respect to ω_d and F.

```
using SciMLSensitivity
using Zygote

N  = 20
a  = destroy(N)
ωc = 5.0
γ  = 1.0

coef_a(p, t) = p[2] * exp(1im * p[1] * t)
coef_ac(p, t) = conj(coef_a(p, t))

H = ωc * a' * a + QobjEvo(a, coef_a) + QobjEvo(a',
    coef_ac)
c_ops = [sqrt(γ) * a]
L = liouvillian(H, c_ops)
ψ0 = fock(N, 0)
tlist = range(0, 40, 100)

function my_f_mesolve(p)
    sol = mesolve(
        L,
        ψ0,
        tlist,
        progress_bar = Val(false),
        params = p,
        sensealg = BacksolveAdjoint(autojacvec =
    EnzymeVJP()),
    )

    return real(expect(a' * a, sol.states[end]))
end

ωd = 1.0
F = 1.0
params = [ωd, F]

grad = Zygote.gradient(my_f_mesolve, params)[1]
```

Once again, the syntax is very simple, as the only difference is the addition of the `params` and `sensealg` keyword arguments to the `mesolve` function. The parameters are passed as a `Vector`, and the dependence of the Hamiltonian on the parameters is defined using the `QobjEvo` constructor. Figure 6 shows the comparison between the analytical derivative (solid blue curve) and the derivative computed using automatic differentiation (dashed orange curve). As can be seen, the two curves are in perfect agreement, demonstrating the accuracy of the AD implementation.

The previous example illustrates how gradients can be obtained by back-propagating a master equation evolution. Reverse-mode AD becomes especially powerful when optimizing over many parameters, such as in pulse-shaping tasks aimed at preparing a desired

target state. As a demonstration, we consider a two-qubit system initialized in $|0,0\rangle$, with the goal of generating the Bell state $|\Phi^+\rangle = (|0,0\rangle + |1,1\rangle)/\sqrt{2}$ at a final time t_f.

In the absence of losses, this state can be ideally prepared by applying a Hadamard gate on the first qubit, followed by a CNOT gate [96]. This corresponds to the time-dependent Hamiltonian

$$\hat{H}_{\text{Bell}}(t) = \hat{H}_{\text{H}}^{(1)}(t) + \hat{H}_{\text{CNOT}}(t), \tag{26}$$

with

$$\hat{H}_{\text{H}}^{(1)}(t) = f_{\text{H}}(t)\,\frac{\pi}{t_f\sqrt{2}}\left(\hat{\sigma}_x^{(1)} - \hat{\sigma}_z^{(1)}\right), \tag{27}$$

representing the Hadamard rotation on qubit 1, active during the first half of the evolution through $f_{\text{H}}(t) = \Theta(t)\Theta(t_f/2 - t)$, where Θ is the Heaviside step function, and

$$\hat{H}_{\text{CNOT}}(t) = f_{\text{CNOT}}(t)\,\frac{\pi}{t_f}\left(\hat{I}^{(1)} + \hat{\sigma}_z^{(1)}\right)\otimes\left(\hat{I}^{(2)} - \hat{\sigma}_x^{(2)}\right), \tag{28}$$

implementing the CNOT gate during the second half of the protocol, with $f_{\text{CNOT}}(t) = \Theta(t - t_f/2)\Theta(t_f - t)$.

When including dissipation via collapse operators $\hat{C}_j = \sqrt{\gamma}\,\hat{\sigma}_-^{(j)}$ with $\gamma = 10^{-2}$ and $t_f = 100$, the resulting fidelity is $\mathcal{F} = \langle\Phi^+|\hat{\rho}(t_f)|\Phi^+\rangle \approx 0.77$, clearly below the ideal value. We emphasize that the chosen decay rate γ is deliberately large compared to the gate time, in order to highlight a noticeable drop in fidelity. In standard experimental architectures, gate times are typically much shorter than decoherence times.

We now assume the functional forms of $f_{\text{H}}(t)$ and $f_{\text{CNOT}}(t)$ are unknown, and instead optimize their shapes directly through gradient-based algorithms.

For this purpose, each function is discretized into N piecewise-constant segments, with amplitudes treated as free parameters:

$$\tilde{f}_{\text{H,opt}}(t) = \sum_{i=1}^{N} \theta_i^{\text{H}}\,\chi_i(t), \tag{29}$$

$$\tilde{f}_{\text{CNOT,opt}}(t) = \sum_{i=1}^{N} \theta_i^{\text{CNOT}}\,\chi_i(t), \tag{30}$$

where $\chi_i(t)$ is unity in the interval $t_i < t < t_{i+1}$ and zero otherwise. To ensure smoothness, the final driving fields $f_{\text{H,opt}}(t)$ and $f_{\text{CNOT,opt}}(t)$ are obtained by cubic interpolation of the piecewise-constant functions.

The optimization thus reduces to finding the parameter vectors $\boldsymbol{\theta}^{\text{H}}$ and $\boldsymbol{\theta}^{\text{CNOT}}$ that minimize the objective function, i.e., the infidelity $1 - \mathcal{F}$. Using $N = 100$ points per pulse, we optimize over 200 parameters $\{\theta_i^{\text{H}}, \theta_i^{\text{CNOT}}\}_i$ with the Adam algorithm [97]. Figure 7(a) shows the infidelity as a function of optimization steps, converging to $\mathcal{F}_{\text{opt}} \approx 0.94$, clearly above the standard gate sequence. To avoid unphysical values for the amplitudes, we also introduce the boundaries from -5 to 5, higher values would give higher fidelities. Figure 7(b) illustrates the standard pulse shapes, while Fig. 7(c) displays the optimized ones after interpolation.

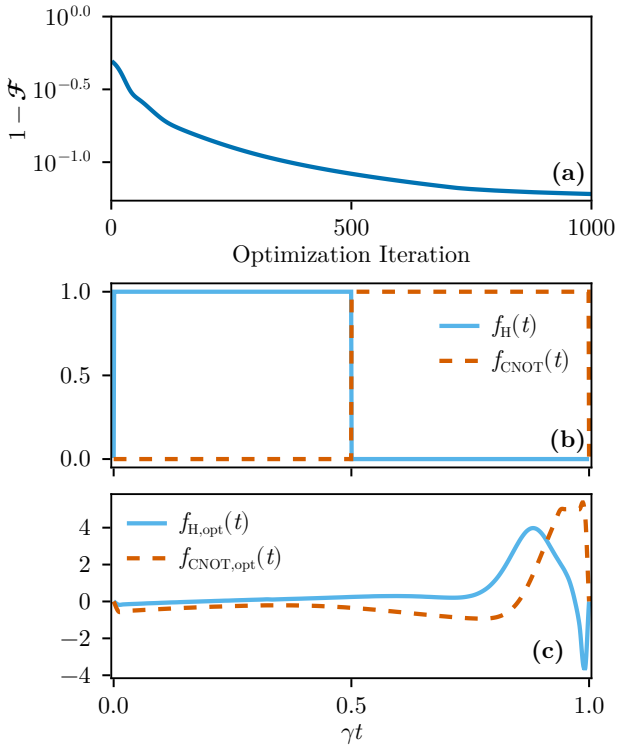

Figure 7: **Bell-state preparation via optimal control. (a)** Infidelity as a function of optimization steps, converging to a final fidelity $\mathcal{F}_{\text{opt}} \approx 0.94$. **(b)** Ideal pulse shapes for the Hadamard and CNOT sequence. **(c)** Optimized pulse shapes obtained through cubic interpolation of piecewise-constant controls.

This example highlights how `QuantumToolbox.jl` leverages automatic differentiation for quantum optimal control, enabling the optimization of pulse shapes involving hundreds of parameters with minimal user effort.

6 Performance comparison with other packages

To assess the computational efficiency of `QuantumToolbox.jl`, we benchmarked its performance against leading quantum simulation packages: `QuTiP` (Python) [27–30], `QuantumToolbox.jl` (Julia) [33], and `dynamiqs` (Python) [31]. The benchmarks include different types of solvers (mesolve, mcsolve, and smesolve), and a range of Hilbert space dimensions, both on CPU and GPU. The code used to generate these benchmarks is available in a dedicated repository [60].

In Fig. 8(a-d), we simulate the open evolution of the driven Kerr oscillator, with the Hamiltonian

$$\hat{H} = \Delta\hat{a}^\dagger\hat{a} - U\hat{a}^{\dagger 2}\hat{a}^2 + F(\hat{a} + \hat{a}^\dagger) \tag{31}$$

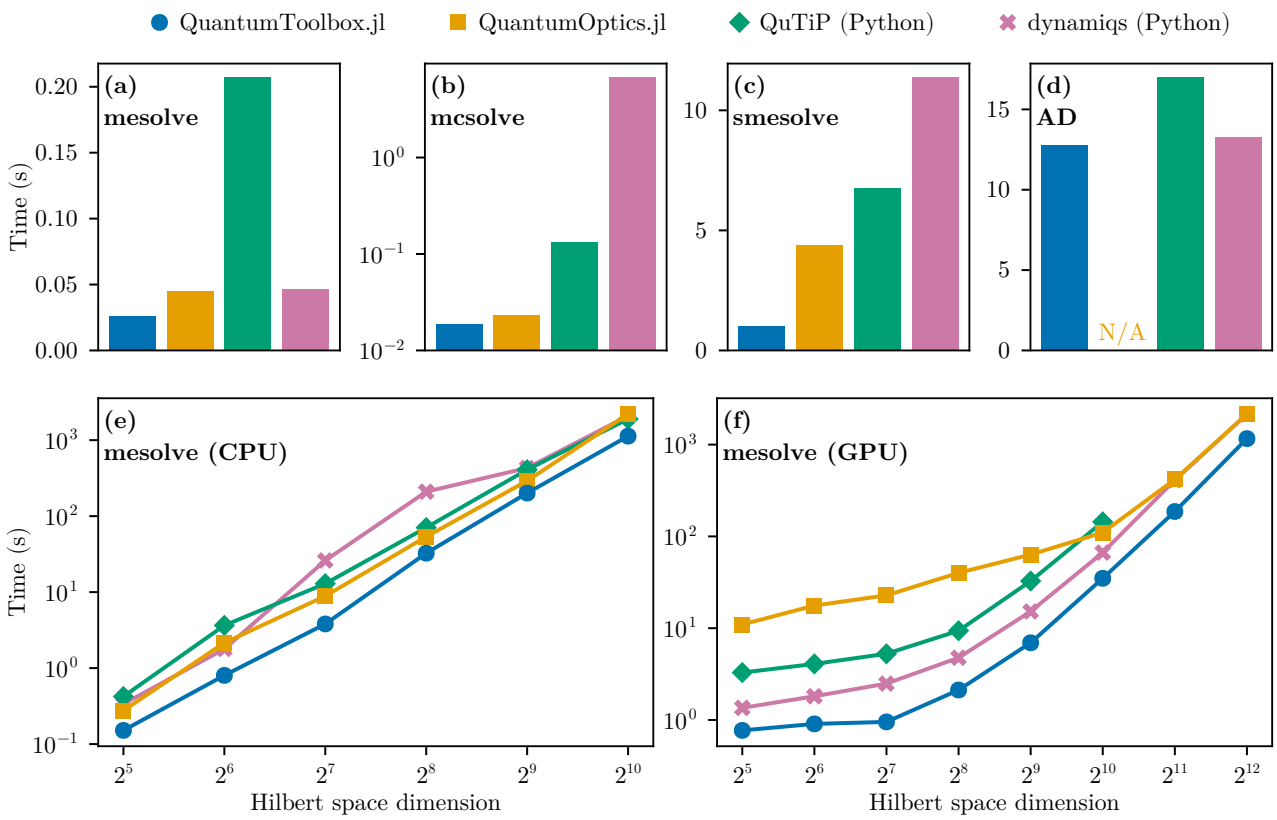

Figure 8: **Comparison of the performance with other packages.** **(a)** Master equation. **(b)** Monte Carlo quantum trajectories. **(c)** Stochastic master equation. **(d)** AD of the master equation. **(e-f)** Scaling of the master equation computation time on the CPU (**e**) and GPU (**f**) with respect to the Hilbert space dimension. `QuantumToolbox.jl` outperforms other packages in all scenarios. While the AD support in `QuantumToolbox.jl` should still be considered experimental, it is already competitive for backpropagation of the master equation. The benchmarks were performed on a workstation with an Intel(R) Core(TM) i9-13900KF CPU and 64 GB of RAM. The GPU simulations were performed on a NVIDIA GeForce RTX 4090 GPU. The versions of the packages used in the benchmarks are summarized in Table 1.

and single photon loss with rate γ. In Fig. 8(e-f), we simulate the 1D Ising model

$$\hat{H} = h_z \sum_{k=1}^{N} \hat{\sigma}_k^z + J_x \sum_{j=1}^{N-1} \hat{\sigma}_k^x \hat{\sigma}_{k+1}^x \tag{32}$$

with local dissipators $\{\hat{\sigma}_j^-\}$, and we vary the number of spins N up to 12 in the GPU case.

As shown in Fig. 8, `QuantumToolbox.jl` consistently outperforms the other libraries across all tested scenarios. Notably, its GPU implementation achieves significant speedups already from small Hilbert sizes, demonstrating superior scalability. These results highlight the advantage of Julia's performance-oriented design combined with QuantumToolbox's efficient solver architecture.

The benchmarks were done on a workstation with an Intel(R) Core(TM) i9-13900KF CPU and 64 GB of RAM. The GPU simulations were performed on a NVIDIA GeForce RTX 4090 GPU. The GPU version of `QuTiP` was performed using its JAX backend, i.e., `qutip-jax`. Other information about the packages used in the benchmarks is summarized in Table 1.

Package	Version	Package	Version
Julia	1.11.6	Python	3.13.5
`QuantumToolbox.jl`	0.34.1	`QuTiP`	5.2.1
		`QuTiP-JAX`	0.1.1
`QuantumOptics.jl`	1.2.3	`dynamiqs`	0.3.3

Table 1: **Versions of the packages used to generate all the figures.** We list the latest version of each package as of Aug. 28, 2025.

7 Conclusions

We presented `QuantumToolbox.jl`, a high-performance Julia package for simulating open quantum systems. By leveraging Julia's JIT compilation and multiple dispatch, the package achieves performance on par with or exceeding that of established tools like `QuTiP` and `QuantumToolbox.jl`, particularly in time evolution and master equation solvers. Its modular design and native compatibility with GPU computing, automatic differentiation, and advanced visualization tools make it a flexible and efficient framework for both research and teaching.

The package reached its first stable release, but we plan to add new features in the future.

It is worth mentioning that, while Julia provides substantial performance advantages, it still suffers from a relatively young ecosystem, longer initial compilation times, and limited community support compared to Python. Nonetheless, we believe that Julia's rapid evolution and growing adoption in scientific computing make it a strong foundation for building next-generation quantum simulation tools. We hope that `QuantumToolbox.jl` will contribute to this ecosystem and become a useful resource for researchers and educators working in quantum science.

Acknowledgements

We acknowledge Fabrizio Minganti and Luca Gravina for the useful discussions, especially for the dynamical shifted Fock algorithm. We also acknowledge Filippo Ferrari, Lorenzo Fioroni, Po-Chen Kuo, Jhen-Dong Lin, Po-Rong Lai, and Hsiang-Wei Huang for their support and beta testing in the development of the package. We thank all the contributors to the package, as well as the `QuTiP` team for their support and the useful discussions. In particular, we thank Eric Giguère and Neill Lambert for their insightful discussions on the benchmarks. YTH acknowledges the support of the National Science and Technology Council, Taiwan (NSTC Grant No. 113-2917-I-006-024). YNC acknowledges the support of the National Center for Theoretical Sciences and the National Science and Technology Council, Taiwan (NSTC Grant No. 113-2123-M-006-001 and 114-2112-M-006 -015 -MY3). V.S. acknowledges the Swiss National Science Foundation through Projects No. 200020_215172, 200021-227992, and 20QU-1_215928. F.N. is supported in part by: the Japan Science and Technology Agency (JST) [via the CREST Quantum Frontiers program Grant No. JPMJCR24I2, the Quantum Leap Flagship Program (Q-LEAP), and the Moonshot R&D Grant Number JPMJMS2061], and the Office of Naval Research (ONR) Global (via Grant No. N62909-23-1-2074).

A Solving the steady state of time-dependent systems

In this section we explain one the algorithms behing the `steadystate_fourier` function, used for extracting the time-averaged steady state for time-dependent systems. We first define the time-dependent equation of motion for the density matrix

$$\frac{d}{dt}\hat{\rho} = \left[\mathcal{L}_0 + \mathcal{L}_1 e^{i\omega_{\mathrm{d}}t} + \mathcal{L}_{-1} e^{-i\omega_{\mathrm{d}}t}\right]\hat{\rho}\,, \tag{33}$$

where $\mathcal{L}_0$ is the time-independent Liouvillian superoperator, while $\mathcal{L}_{\pm 1}$ are the superoperators containing the drive terms. For example, in the case of the optomechanical system in Eq. (9), the Liouvillian superoperators are given by

$$\begin{aligned}\mathcal{L}_0\hat{\rho} &= -i\left[\hat{H},\hat{\rho}\right] + \kappa\mathcal{D}[\hat{a}]\hat{\rho} + \gamma\mathcal{D}[\hat{b}]\hat{\rho},\\ \mathcal{L}_{\pm 1}\hat{\rho} &= -i\left[\frac{F}{2}\left(\hat{a}+\hat{a}^\dagger\right),\hat{\rho}\right].\end{aligned} \tag{34}$$

At long times, all the transient dynamics are washed out, and the density matrix can be expanded in Fourier components of the form

$$\hat{\rho}(t) = \sum_{n=-\infty}^{+\infty} \hat{\rho}_n e^{in\omega_{\mathrm{d}}t}\,. \tag{35}$$

By substituting the expansion in the equation of motion, we obtain

$$\sum_{n=-\infty}^{+\infty} in\omega_{\mathrm{d}}\hat{\rho}_n e^{in\omega_{\mathrm{d}}t} = \sum_{n=-\infty}^{+\infty}\left[\mathcal{L}_0 + \mathcal{L}_1 e^{i\omega_{\mathrm{d}}t} + \mathcal{L}_{-1}e^{-i\omega_{\mathrm{d}}t}\right]\hat{\rho}_n e^{in\omega_{\mathrm{d}}t}\,. \tag{36}$$

By equating the coefficients of the series yields the tridiagonal recursion relation

$$(\mathcal{L}_0 - in\omega_{\mathrm{d}})\hat{\rho}_n + \mathcal{L}_1\hat{\rho}_{n-1} + \mathcal{L}_{-1}\hat{\rho}_{n+1} = 0\,. \tag{37}$$

After choosing a cutoff $n_{\max}$ for the Fourier components, the equation above defines a tridiagonal linear system of the form $\mathbf{A}\cdot\mathbf{b} = 0$, where

$$\mathbf{A} = \begin{pmatrix} \mathcal{L}_0 - i(-n_{\max})\omega_{\mathrm{d}} & \mathcal{L}_{-1} & 0 & \cdots & 0 \\ \mathcal{L}_1 & \mathcal{L}_0 - i(-n_{\max}+1)\omega_{\mathrm{d}} & \mathcal{L}_{-1} & \cdots & 0 \\ 0 & \mathcal{L}_1 & \mathcal{L}_0 - i(-n_{\max}+2)\omega_{\mathrm{d}} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \mathcal{L}_0 - in_{\max}\omega_{\mathrm{d}} \end{pmatrix} \tag{38}$$

and

$$\mathbf{b} = \begin{pmatrix} \hat{\rho}_{-n_{\max}} & \dots & \hat{\rho}_0 & \dots & \hat{\rho}_{n_{\max}} \end{pmatrix}^\top . \tag{39}$$

This allows to simultaneously obtain all the Fourier components $\hat{\rho}_n$.

B The dynamical shifted Fock algorithm

The DSF algorithm is based on the efficient manipulation of coherent states and displacement operators. A coherent state is defined as:

$$|\alpha\rangle = e^{-|\alpha|^2/2} \sum_{n=0}^{+\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \,, \tag{40}$$

which can be also seen as the eigenstate of the bosonic annihilation operator $\hat{a}|\alpha\rangle = \alpha|\alpha\rangle$. We define $\alpha = \langle\alpha|\hat{a}|\alpha\rangle$ as the coherence of the state. A coherent state can be obtained from the action of the displacement operator

$$\hat{D}(\alpha) = e^{\alpha\hat{a}^\dagger - \alpha^*\hat{a}} \,, \quad \hat{D}(\alpha)\hat{D}^\dagger(\alpha) = \hat{D}(\alpha)\hat{D}(-\alpha) = 1 \tag{41}$$

to the vacuum state $|0\rangle$, namely $|\alpha\rangle = \hat{D}(\alpha)|0\rangle$. From this definition and from Eq. (41), it follows that

$$|0\rangle = \hat{D}^\dagger(\alpha)|\alpha\rangle \,. \tag{42}$$

It can also be proven that, given any two coherent states $|\alpha\rangle$ and $|\beta\rangle$, one has

$$|\alpha\rangle \propto \hat{D}(\alpha - \beta)\,|\beta\rangle \,. \tag{43}$$

The equations above are the key points of the DSF algorithm. Indeed, a coherent state defined in the Fock basis as in Eq. (40) can have large occupation probabilities in the high-energy states. However, thanks to Eq. (42), it is possible to rotate a high-energy coherent state into the vacuum state $|0\rangle$. Numerically speaking, this allows us to treat a coherent state with a vector where all the elements are zero, except for the first one. Hence, we can describe it with a smaller vector, given that we keep track of its coherence.

This approach is also valid when the state is almost-coherent-like, with small quantum fluctuations around a macroscopic coherent state. In this case, given such a state $|\psi\rangle$, we have

$$\hat{D}^\dagger(\alpha)|\psi\rangle = \sum_n c_n |n\rangle \,, \tag{44}$$

where $|c_n|$ rapidly decreases as n increases. Once again, this state can still be represented with a small Fock basis, and the accuracy of the DSF depends on the size of this basis.

Let us start by considering a closed quantum system with initial state $|\psi(t_0 = 0)\rangle$, whose unitary dynamics is governed by a generic Hamiltonian $\hat{H}$. For fixed-time-step integration, the algorithm is presented in Table 2, and its extension to adaptive-time-step integrators is straightforward as `DiffEqCallbacks.jl` supports very general integration algorithms.

1: **Initialize:** state $|\psi(0)\rangle$, Hamiltonian $\hat{H}$, dissipators $\{\hat{O}_n\}_n$ and threshold $\delta_\alpha^{(\max)}$.
2: Set $t \leftarrow 0$.
3: **while** $t < t_f$ **do**
4: **Evolve:** Integrate

$$i\frac{d}{dt}|\psi(t)\rangle = \hat{H}|\psi(t)\rangle$$

from t to $t + \Delta t$.
5: **Compute coherence**:

$$\alpha_{t+\Delta t} \leftarrow \langle\psi(t+\Delta t)|\hat{a}|\psi(t+\Delta t)\rangle.$$

6: **Compute shift**:

$$\delta_\alpha \leftarrow \alpha_{t+\Delta t} - \alpha_t.$$

7: **if** $|\delta_\alpha| > \delta_\alpha^{(\max)}$ **then**
8: **Displace state:**

$$|\psi(t+\Delta t)\rangle \leftarrow \hat{D}(-\delta_\alpha)|\psi(t+\Delta t)\rangle.$$

9: **Update operators:**

$$\hat{H} \leftarrow \hat{D}(-\delta_\alpha)\hat{H}\hat{D}(\delta_\alpha).$$

$$\hat{O}_n \leftarrow \hat{D}(-\delta_\alpha)\hat{O}_n\hat{D}(\delta_\alpha).$$

10: **end if**
11: Set $t \leftarrow t + \Delta t$.
12: **end while**

Table 2: Fixed-time-step DSF Algorithm

To summarize, the DSF algorithm allows us to simulate the dynamics of a system by keeping the Hilbert space dimension small, at the expense of tracking the coherence and performing the unitary transformations when reaching a given threshold.

C Tables of functions

In this subsection, we summarize the most relevant user-accessible functions in `QuantumToolbox.jl` into several tables. Table 3 summarizes the functions that can easily construct typical quantum states and operators. Table 4 summarizes the time evolution solvers that compute the dynamics of a quantum system according to different methods. Table 5 summarizes the standard linear algebra operator and some utility functions for analyzing the properties and physical quantities of given quantum objects. Note that the complete list of functions is shown in the `QuantumToolbox.jl` documentation website (https://qutip.org/QuantumToolbox.jl/stable/).

Furthermore, additional information about each function can be obtained by calling “ `?function_name`” in Julia.

Function	Description
User-defined quantum object	
`QuantumObject` / `Qobj`	General time-independent quantum object
`QuantumObjectEvolution` / `QobjEvo`	General time-dependent quantum object
States	
`basis` / `fock` / `fock_dm`	Single basis (Fock) state
`coherent` / `coherent_dm`	Single-mode coherent state
`rand_ket` / `rand_dm`	Random state
`thermal_dm`	Thermal state
`maximally_mixed_dm`	Maximally mixed state
`spin_state`	Spin state
`spin_coherent`	Coherent spin state
`bell_state`	Bell state
`singlet_state`	Singlet state
`triplet_states`	Triplet states
`w_state`	W state
`ghz_state`	Greenberger–Horne–Zeilinger state
Operators	
`eye` / `qeye`	Identity operator
`destroy`	Bosonic annihilation operator
`create`	Bosonic creation operator
`fdestroy`	Fermionic annihilation operator
`fcreate`	Fermionic creation operator
`projection`	Projection operator
`displace`	Displacement operator
`squeeze`	Single-mode squeeze operator
`num`	Bosonic number operator
`phase`	Single-mode Pegg-Barnett phase operator
`position`	Position operator
`momentum`	Momentum operator
`rand_unitary`	Random unitary operator
`sigmax`	Spin-1/2 Pauli-X operator
`sigmay`	Spin-1/2 Pauli-Y operator
`sigmaz`	Spin-1/2 Pauli-Z operator
`sigmap`	Spin-1/2 Pauli ladder (σ^+) operator
`sigmam`	Spin-1/2 Pauli ladder (σ^-) operator
`jmat`	General spin-j operator
`qft`	Discrete quantum Fourier transform operator

Table 3: **List of functions in `QuantumToolbox.jl` for constructing typical quantum states and operators.** Note that "`_dm`" represents the state is constructed in density matrix formalism.

Solver	Description
`sesolve`	Solve Schrödinger equation
`mesolve`	Solve master equation
`mcsolve`	Solve time evolution using Monte Carlo wave-function method
`ssesolve`	Solve stochastic Schrödinger equation [76]
`smesolve`	Solve stochastic master equation [76]
`lr_mesolve`	Solve low-rank master equation [17]
`dfd_mesolve`	Solve master equation using the dynamical Fock dimension algorithm
`dsf_mesolve`	Solve master equation using the dynamical shifted Fock algorithm
`dsf_mcsolve`	Solve Monte Carlo quantum trajectories using the dynamical shifted Fock algorithm
`brmesolve`	Solve Bloch-Redfield master equation [62]
`heomsolve`	Solve hierarchical equations of motion [44]
`steadystate`	Calculate the steady state
`steadystate_fourier`	Calculate the steady state of a periodically driven system

Table 4: **List of time evolution solvers in QuantumToolbox.jl.**

Function	Description
Linear algebra operations	
`conj`	Conjugation of a quantum object
`transpose` / `trans`	Transpose of a quantum object
`adjoint` / `dag` / `'`	Adjoint of a quantum object
`partial_transpose`	Partial transpose of a quantum object
`dot`	Dot product between quantum objects
`matrix_element`	Extract single matrix element from a quantum object
`tr`	Trace of a quantum object
`ptrace`	Partial trace of a quantum object
`svdvals`	Compute singular values of a quantum object
`norm`	Compute standard vector or Schatten p-norm of a quantum object
`normalize` / `normalize!` / `unit`	Normalize a quantum object
`inv`	Compute matrix inverse of a quantum object
`sqrt` / `sqrtm` / `√`	Compute matrix square root of a quantum object
`log` / `logm`	Compute matrix logarithm of a quantum object
`exp` / `expm`	Compute matrix exponential of a quantum object
`sin` / `sinm`	Compute matrix sine of a quantum object
`cos` / `cosm`	Compute matrix cosine of a quantum object
`diag`	Extract diagonal elements from a quantum object
`proj`	Projection operator of a quantum `Ket` state
`permute`	Permute the tensor structure of a quantum object
`tidyup` / `tidyup!`	Remove small matrix elements in a quantum object
`kron` / `tensor` / `⊗`	Kronecker (tensor) product of quantum objects
Quantum object conversions	
`cu`	Convert `data` into CUDA (GPU) array
`to_dense`	Convert `data` into dense array
`to_sparse`	Convert `data` into sparse array
`ket2dm`	Convert a `Ket` state into density matrix (`Operator`)
`mat2vec` / `operator_to_vector`	Vectorize an `Operator` into `OperatorKet`
`vec2mat` / `vector_to_operator`	Reshape (un-vectorize) an `OperatorKet` back to `Operator`
`spre`	Generate left multiplication `SuperOperator`
`spost`	Generate right multiplication `SuperOperator`
`sprepost`	Generate left-and-right multiplication `SuperOperator`
`liouvillian`	Generate Liouvillian `SuperOperator`
`lindblad_dissipator`	Generate Lindblad dissipator (`SuperOperator`)
Utility functions	
`expect`	Compute expectation value of a quantum operator
`variance`	Compute variance of a quantum operator
`eigvals` / `eigenenergies`	Compute eigenvalues of a quantum object
`eigen` / `eigsolve` / `eigenstates`	Compute eigenvectors and eigenvalues of a quantum object
`eigsolve_al`	The Arnoldi-Lindblad eigensolver [98]
`purity`	Compute purity of a quantum state
`entropy_vn`	Compute Von Neumann entropy [96] of a quantum state
`entropy_linear`	Compute quantum linear entropy [96] of a quantum state
`entropy_conditional`	Compute quantum conditional entropy [96] of a quantum state
`entropy_relative`	Compute quantum relative entropy [96] of a quantum state
`entropy_mutual`	Compute quantum mutual information [96] of a quantum state
`entanglement`	Compute entanglement entropy [96] of a bipartite quantum state
`concurrence`	Compute concurrence [99] of a bipartite quantum state
`negativity`	Compute negativity [100] of a bipartite quantum state
`fidelity`	Compute fidelity [96] between two quantum states
`tracedist`	Compute trace distance [96] between two quantum states
`correlation_2op_1t`	Compute correlation function of two operators with one time coordinate
`correlation_2op_2t`	Compute correlation function of two operators with two time coordinates
`correlation_3op_1t`	Compute correlation function of three operators with one time coordinate
`correlation_3op_2t`	Compute correlation function of three operators with two time coordinates
`spectrum`	Compute the power spectrum of a correlation function
`spectrum_correlation_fft`	Compute the fast Fourier transform from a given correlation function data
`wigner`	Generate the Wigner quasipropability distribution of a quantum state

Table 5: **List of standard linear algebra and other utility functions for quantum objects in `QuantumToolbox.jl`.**

References

[1] Steven R. White. "Density matrix formulation for quantum renormalization groups". Physical Review Letters **69**, 2863–2866 (1992).

[2] Steven R. White. "Density-matrix algorithms for quantum renormalization groups". Physical Review B **48**, 10345–10356 (1993).

[3] Guifré Vidal. "Efficient Classical Simulation of Slightly Entangled Quantum Computations". Phys. Rev. Lett. **91**, 147902 (2003).

[4] M. C. Bañuls, M. B. Hastings, F. Verstraete, and J. I. Cirac. "Matrix Product States for Dynamical Simulation of Infinite Chains". Phys. Rev. Lett. **102**, 240603 (2009).

[5] Ulrich Schollwöck. "The density-matrix renormalization group in the age of matrix product states". Annals of Physics **326**, 96–192 (2011).

[6] Román Orús. "A practical introduction to tensor networks: Matrix product states and projected entangled pair states". Annals of Physics **349**, 117–158 (2014).

[7] Román Orús. "Tensor networks for complex quantum systems". Nature Reviews Physics **1**, 538–550 (2019).

[8] J. Ignacio Cirac, David Pérez-García, Norbert Schuch, and Frank Verstraete. "Matrix product states and projected entangled pair states: Concepts, symmetries, theorems". Rev. Mod. Phys. **93**, 045003 (2021).

[9] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. "The ITensor Software Library for Tensor Network Calculations". SciPost Phys. Codebases **4** (2022).

[10] David Ceperley and Berni Alder. "Quantum Monte Carlo". Science **231**, 555–560 (1986).

[11] Federico Becca and Sandro Sorella. "Quantum Monte Carlo Approaches for Correlated Systems". Cambridge University Press. (2017).

[12] Antoine Georges and Gabriel Kotliar. "Hubbard model in infinite dimensions". Physical Review B **45**, 6479–6483 (1992).

[13] Antoine Georges, Gabriel Kotliar, Werner Krauth, and Marcelo J. Rozenberg. "Dynamical mean-field theory of strongly correlated fermion systems and the limit of infinite dimensions". Reviews of Modern Physics **68**, 13–125 (1996).

[14] W. L. McMillan. "Ground State of Liquid He^4". Phys. Rev. **138**, A442–A451 (1965).

[15] D. Ceperley, G. V. Chester, and M. H. Kalos. "Monte Carlo simulation of a many-fermion study". Phys. Rev. B **16**, 3081–3099 (1977).

[16] Tao Shi, Eugene Demler, and J. Ignacio Cirac. "Variational study of fermionic and bosonic systems with non-Gaussian states: Theory and applications". Annals of Physics **390**, 245–302 (2018).

[17] Luca Gravina and Vincenzo Savona. "Adaptive variational low-rank dynamics for open quantum systems". Phys. Rev. Res. **6**, 023072 (2024).

[18] Dian Wu, Riccardo Rossi, Filippo Vicentini, Nikita Astrakhantsev, Federico Becca, Xiaodong Cao, Juan Carrasquilla, Francesco Ferrari, Antoine Georges, Mohamed Hibat-Allah, Masatoshi Imada, Andreas M. Läuchli, Guglielmo Mazzola, Antonio Mezzacapo, Andrew Millis, Javier Robledo Moreno, Titus Neupert, Yusuke Nomura, Jannes Nys, Olivier Parcollet, Rico Pohle, Imelda Romero, Michael Schmid, J. Maxwell Silvester, Sandro Sorella, Luca F. Tocchio, Lei Wang, Steven R. White, Alexander Wietek, Qi Yang, Yiqi Yang, Shiwei Zhang, and Giuseppe Carleo. "Variational benchmarks for quantum many-body problems". Science **386**, 296–301 (2024).

[19] Giuseppe Carleo and Matthias Troyer. "Solving the quantum many-body problem with artificial neural networks". Science **355**, 602–606 (2017).

[20] Giuseppe Carleo, Kenny Choo, Damian Hofmann, James E. T. Smith, Tom Westerhout, Fabien Alet, Emily J. Davis, Stavros Efthymiou, Ivan Glasser, Sheng-Hsuan Lin, Marta Mauri, Guglielmo Mazzola, Christian B. Mendl, Evert van Nieuwenburg, Ossian O'Reilly, Hugo Théveniaut, Giacomo Torlai, Filippo Vicentini, and Alexander Wietek. "NetKet: A Machine Learning Toolkit for Many-Body Quantum Systems". SoftwareXPage 100311 (2019).

[21] Filippo Vicentini, Damian Hofmann, Attila Szabó, Dian Wu, Christopher Roth, Clemens Giuliani, Gabriel Pescia, Jannes Nys, Vladimir Vargas-Calderón, Nikita Astrakhantsev, and Giuseppe Carleo. "NetKet 3: Machine Learning Toolbox for Many-Body Quantum Systems". SciPost Phys. Codebases **7** (2022).

[22] Ryogo Kubo. "Generalized Cumulant Expansion Method". Journal of the Physical Society of Japan **17**, 1100–1120 (1962).

[23] David Plankensteiner, Christoph Hotter, and Helmut Ritsch. "QuantumCumulants.jl: A Julia framework for generalized mean-field equations in open quantum systems". Quantum **6**, 617 (2022).

[24] Anatoli Polkovnikov. "Phase space representation of quantum dynamics". Annals of Physics **325**, 1790–1852 (2010).

[25] Sze M. Tan. "A computational toolbox for quantum and atomic optics". Journal of Optics B: Quantum and Semiclassical Optics **1**, 424–432 (1999).

[26] Desmond J. Higham and Nicholas J. Higham. "MATLAB Guide". Pages xxvi+476. Society for Industrial and Applied Mathematics. Philadelphia, PA, USA (2017).

[27] J.R. Johansson, P.D. Nation, and Franco

Nori. "QuTiP: An open-source Python framework for the dynamics of open quantum systems". Computer Physics Communications **183**, 1760–1772 (2012).

[28] J.R. Johansson, P.D. Nation, and Franco Nori. "QuTiP 2: A Python framework for the dynamics of open quantum systems". Computer Physics Communications **184**, 1234–1240 (2013).

[29] Boxi Li, Shahnawaz Ahmed, Sidhant Saraogi, Neill Lambert, Franco Nori, Alexander Pitchford, and Nathan Shammah. "Pulse-level noisy quantum circuits with QuTiP". Quantum **6**, 630 (2022).

[30] Neill Lambert, Eric Giguère, Paul Menczel, Boxi Li, Patrick Hopf, Gerardo Suárez, Marc Gali, Jake Lishman, Rushiraj Gadhvi, Rochisha Agarwal, Asier Galicia, Nathan Shammah, Paul Nation, J. R. Johansson, Shahnawaz Ahmed, Simon Cross, Alexander Pitchford, and Franco Nori. "QuTiP 5: The Quantum Toolbox in Python" (2024). url: https://arxiv.org/abs/2412.04705.

[31] Pierre Guilmin, Ronan Gautier, Adrien Bocquet, Élie Genois, and Daniel Weiss. "Dynamiqs: an open-source Python library for GPU-accelerated and differentiable simulation of quantum systems". https://github.com/dynamiqs/dynamiqs (2024).

[32] Guido Van Rossum and Fred L. Drake. "Python 3 Reference Manual". CreateSpace. Scotts Valley, CA (2009).

[33] Sebastian Krämer, David Plankensteiner, Laurin Ostermann, and Helmut Ritsch. "QuantumOptics. jl: A Julia framework for simulating open quantum systems". Computer Physics Communications **227**, 109 (2018).

[34] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. "Julia: A Fast Dynamic Language for Technical Computing" (2012). arXiv:1209.5145.

[35] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. "Julia: A Fresh Approach to Numerical Computing". SIAM Review **59**, 65 (2017).

[36] Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". IEEE Transactions on Parallel and Distributed Systems **30**, 827–841 (2019).

[37] Christopher Rackauckas and Qing Nie. "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". Journal of Open Research Software **5**, 15 (2017).

[38] Will Kimmerer, Vedant Puri, and Chris Rackauckas. "LinearSolve.jl". https://github.com/SciML/LinearSolve.jl.

[39] Brian W Kernighan and Dennis M Ritchie. "The C programming language". Prentice Hall Professional Technical Reference. (2006).

[40] David Flanagan and Yukihiro Matsumoto. "The Ruby Programming Language". O'Reilly Media, Inc. (2007).

[41] R Core Team. "R: A Language and Environment for Statistical Computing". R Foundation for Statistical Computing. Vienna, Austria (2021). url: https://www.R-project.org/.

[42] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In International Symposium on Code Generation and Optimization, 2004. CGO 2004. Page 75. (2004).

[43] Huo Chen and Daniel A. Lidar. "Hamiltonian open quantum system toolkit". Communications Physics **5**, 112 (2022).

[44] Yi-Te Huang, Po-Chen Kuo, Neill Lambert, Mauro Cirio, Simon Cross, Shen-Liang Yang, Franco Nori, and Yueh-Nan Chen. "An efficient Julia framework for hierarchical equations of motion in open quantum systems". Communications Physics **6**, 313 (2023).

[45] Matias Bundgaard-Nielsen, Dirk Englund, Mikkel Heuck, and Stefan Krastanov. "WaveguideQED.jl: An Efficient Framework for Simulating Non-Markovian Waveguide Quantum Electrodynamics" (2024). arXiv:2412.13332.

[46] Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang, and Lei Wang. "Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design". Quantum **4**, 341 (2020).

[47] Piotr Gawron, Dariusz Kurzyk, and Łukasz Pawela. "QuantumInformation.jl—A Julia package for numerical computation in quantum information theory". PLOS ONE **13**, e0209358 (2018).

[48] "StaticArrays.jl: Statically sized arrays for Julia". https://github.com/JuliaArrays/StaticArrays.jl.

[49] "SciMLOperators.jl: Matrix-Free Operators for the SciML Scientific Machine Learning Common Interface in Julia". https://github.com/SciML/SciMLOperators.jl.

[50] Yoshitaka Tanimura and Ryogo Kubo. "Time Evolution of a Quantum System in Contact with a Nearly Gaussian-Markoffian Noise Bath". Journal of the Physical Society of Japan **58**, 101–114 (1989).

[51] Yoshitaka Tanimura. "Nonperturbative Expansion Method for a Quantum System Coupled to a Harmonic-Oscillator Bath". Physical Review A **41**, 6676–6687 (1990).

[52] Yoshitaka Tanimura. "Numerically "Exact" Approach to Open Quantum Dynamics: The Hierarchical Equations of Motion

(HEOM)". The Journal of Chemical Physics **153**, 020901 (2020).

[53] Jinshuang Jin, Xiao Zheng, and YiJing Yan. "Exact Dynamics of Dissipative Electronic Systems and Quantum Transport: Hierarchical Equations of Motion Approach". The Journal of Chemical Physics **128**, 234703 (2008).

[54] ZhenHua Li, NingHua Tong, Xiao Zheng, Dong Hou, JianHua Wei, Jie Hu, and YiJing Yan. "Hierarchical Liouville-Space Approach for Accurate and Universal Characterization of Quantum Impurity Systems". Physical Review Letters **109**, 266403 (2012).

[55] Neill Lambert, Tarun Raheja, Simon Cross, Paul Menczel, Shahnawaz Ahmed, Alexander Pitchford, Daniel Burgarth, and Franco Nori. "QuTiP-BoFiN: A bosonic and fermionic numerical hierarchical-equations-of-motion library with applications in light-harvesting, quantum control, and single-molecule electronics". Physical Review Research **5**, 013181 (2023).

[56] Po-Chen Kuo, Neill Lambert, Mauro Cirio, Yi-Te Huang, Franco Nori, and Yueh-Nan Chen. "Kondo QED: The Kondo effect and photon trapping in a two-impurity Anderson model ultrastrongly coupled to light". Physical Review Research **5**, 043177 (2023).

[57] Po-Chen Kuo, Shen-Liang Yang, Neill Lambert, Jhen-Dong Lin, Yi-Te Huang, Franco Nori, and Yueh-Nan Chen. "Non-Markovian skin effect". Physical Review Research **7**, L012068 (2025).

[58] E. Wigner. "On the Quantum Correction For Thermodynamic Equilibrium". Phys. Rev. **40**, 749–759 (1932).

[59] Simon Danisch and Julius Krumbiegel. "Makie.jl: Flexible high-performance data visualization for Julia". Journal of Open Source Software **6**, 3349 (2021).

[60] "Repository containing the code used to generate the figures in the QuantumToolbox.jl paper.". https://github.com/albertomercurio/QuantumToolbox.jl-Paper-Figures.

[61] E.T. Jaynes and F.W. Cummings. "Comparison of quantum and semiclassical radiation theories with application to the beam maser". Proceedings of the IEEE **51**, 89–109 (1963).

[62] Heinz-Peter Breuer and Francesco Petruccione. "The Theory of Open Quantum Systems". Oxford University PressOxford. (2007).

[63] P. D. Nation, J. R. Johansson, M. P. Blencowe, and A. J. Rimberg. "Iterative solutions to the steady-state density matrix for optomechanical systems". Phys. Rev. E **91**, 013307 (2015).

[64] P. D. Nation. "Steady-state solution methods for open quantum optical systems" (2015). arXiv:1504.06768.

[65] Jean Dalibard, Yvan Castin, and Klaus Mølmer. "Wave-function approach to dissipative processes in quantum optics". Physical Review Letters **68**, 580–583 (1992).

[66] R. Dum, P. Zoller, and H. Ritsch. "Monte Carlo simulation of the atomic master equation for spontaneous emission". Physical Review A **45**, 4879–4887 (1992).

[67] Klaus Mølmer, Yvan Castin, and Jean Dalibard. "Monte Carlo wave-function method in quantum optics". Journal of the Optical Society of America B **10**, 524 (1993).

[68] Howard Carmichael. "An Open Systems Approach to Quantum Optics: Lectures Presented at the Université Libre de Bruxelles October 28 to November 4, 1991". Springer Berlin Heidelberg. (1993).

[69] C. K. Law. "Interaction between a moving mirror and radiation pressure: A Hamiltonian formulation". Physical Review A **51**, 2537–2541 (1995).

[70] Markus Aspelmeyer, Tobias J. Kippenberg, and Florian Marquardt. "Cavity optomechanics". Reviews of Modern Physics **86**, 1391–1452 (2014).

[71] Vincenzo Macrì, Alessandro Ridolfo, Omar Di Stefano, Anton Frisk Kockum, Franco Nori, and Salvatore Savasta. "Nonperturbative Dynamical Casimir Effect in Optomechanical Systems: Vacuum Casimir-Rabi Splittings". Physical Review X**8** (2018).

[72] Arka Majumdar, Alexander Papageorge, Erik D. Kim, Michal Bajcsy, Hyochul Kim, Pierre Petroff, and Jelena Vučković. "Probing of single quantum dot dressed states via an off-resonant cavity". Phys. Rev. B **84**, 085310 (2011).

[73] Alexander Papageorge, Arka Majumdar, Erik D Kim, and Jelena Vučković. "Bichromatic driving of a solid-state cavity quantum electrodynamics system". New Journal of Physics **14**, 013028 (2012).

[74] M. Maragkou, C. Sánchez-Muñoz, S. Lazić, E. Chernysheva, H. P. van der Meulen, A. González-Tudela, C. Tejedor, L. J. Martínez, I. Prieto, P. A. Postigo, and J. M. Calleja. "Bichromatic dressing of a quantum dot detected by a remote second quantum dot". Phys. Rev. B **88**, 075309 (2013).

[75] Vincenzo Macrì, Alberto Mercurio, Franco Nori, Salvatore Savasta, and Carlos Sánchez Muñoz. "Spontaneous Scattering of Raman Photons from Cavity-QED Systems in the Ultrastrong Coupling Regime". Physical Review Letters**129** (2022).

[76] Howard M. Wiseman and Gerard J. Milburn. "Quantum Measurement and Control". Cambridge University Press. (2009).

[77] David P. DiVincenzo. "The Physical Implementation of Quantum Computation". Fortschritte der Physik **48**, 771–783 (2000).

[78] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver. "A quantum engineer's guide to superconducting qubits". Applied Physics Reviews **6**, 021318 (2019).

[79] Iulia Buluta and Franco Nori. "Quantum Simulators". Science **326**, 108–111 (2009).

[80] I. M. Georgescu, S. Ashhab, and Franco Nori. "Quantum simulation". Reviews of Modern Physics **86**, 153–185 (2014).

[81] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, Advances in Neural Information Processing Systems. Volume 25. Curran Associates, Inc. (2012). url: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[82] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". Nature **521**, 436–444 (2015).

[83] "DistributedArrays.jl: Distributed Arrays in Julia". https://github.com/JuliaParallel/DistributedArrays.jl.

[84] "PartitionedArrays.jl: Large-scale, distributed, sparse linear algebra in Julia.". https://github.com/PartitionedArrays/PartitionedArrays.jl.

[85] "Dagger.jl: A framework for out-of-core and parallel execution". https://github.com/JuliaParallel/Dagger.jl.

[86] Anthony P. Peirce, Mohammed A. Dahleh, and Herschel Rabitz. "Optimal control of quantum-mechanical systems: Existence, numerical approximation, and applications". Phys. Rev. A **37**, 4950–4964 (1988).

[87] J Werschnik and E K U Gross. "Quantum optimal control theory". Journal of Physics B: Atomic, Molecular and Optical Physics **40**, R175–R211 (2007).

[88] Domenico D'Alessandro. "Introduction to Quantum Control and Dynamics". Chapman and Hall/CRC. (2021).

[89] Christiane P. Koch, Ugo Boscain, Tommaso Calarco, Gunther Dirr, Stefan Filipp, Steffen J. Glaser, Ronnie Kosloff, Simone Montangero, Thomas Schulte-Herbrüggen, Dominique Sugny, and Frank K. Wilhelm. "Quantum optimal control in quantum technologies. Strategic report on current status, visions and goals for research in Europe". EPJ Quantum Technology **9**, 19 (2022).

[90] "ForwardDiff.jl: Forward Mode Automatic Differentiation for Julia". https://github.com/JuliaDiff/ForwardDiff.jl.

[91] Michae Innes. "Don't Unroll Adjoint: Differentiating SSA-Form Programs" (2019). arXiv:1810.07951.

[92] William Moses and Valentin Churavy. "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients". In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, Advances in Neural Information Processing Systems. Volume 33, pages 12472–12485. Curran Associates, Inc. (2020). url: https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf.

[93] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. "Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme". In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '21New York, NY, USA (2021). Association for Computing Machinery.

[94] William S. Moses, Sri Hari Krishna Narayanan, Ludger Paehler, Valentin Churavy, Michel Schanen, Jan Hückelheim, Johannes Doerfert, and Paul Hovland. "Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation". In SC22: International Conference for High Performance Computing, Networking, Storage and Analysis. Page 1–18. IEEE (2022).

[95] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. "Universal Differential Equations for Scientific Machine Learning" (2021). arXiv:2001.04385.

[96] Michael A. Nielsen and Isaac L. Chuang. "Quantum Computation and Quantum Information: 10th Anniversary Edition". Cambridge University Press. (2012).

[97] Diederik P. Kingma and Jimmy Ba. "Adam: A method for stochastic optimization" (2017). arXiv:1412.6980.

[98] Fabrizio Minganti and Dolf Huybrechts. "Arnoldi-Lindblad time evolution: Faster-than-the-clock algorithm for the spectrum of time-independent and Floquet open quantum systems". Quantum **6**, 649 (2022).

[99] Sam A. Hill and William K. Wootters. "Entanglement of a Pair of Quantum Bits". Phys. Rev. Lett. **78**, 5022–5025 (1997).

[100] G. Vidal and R. F. Werner. "Computable

measure of entanglement". Phys. Rev. A **65**, 032314 (2002).