

QuTiP 5: The Quantum Toolbox in Python

Neill Lambert^{a,*}, Eric Giguère^b, Paul Menczel^{a,*}, Boxi Li^c, Patrick Hopf^{a,d}, Gerardo Suárez^{a,e}, Marc Gali^f, Jake Lishman^g, Rushiraj Gadhwī^h, Rochisha Agarwal^a, Asier Galiciaⁱ, Nathan Shammah^j, Paul Nation^k, J. R. Johansson^l, Shahnawaz Ahmed^m, Simon Crossⁿ, Alexander Pitchford^o, Franco Nori^{a,p,*}

^aQuantum Information Physics Theory Research Team,

RIKEN Center for Quantum Computing, RIKEN, Wakoshi, Saitama 351-0198, Japan,

^bInstitut quantique, Université de Sherbrooke, Sherbrooke J1K 2R1 Quebec, Canada,

^cPeter Grünberg Institute -Quantum Control (PGI-8), Forschungszentrum Jülich GmbH, D-52425 Jülich, Germany,

^dTechnical University of Munich, Munich, Germany,

^eInternational Centre for Theory of Quantum Technologies (ICTQT), University of Gdańsk, 80-308 Gdańsk, Poland,

^fGlobal Research and Development Center for Business by Quantum-AI Technology (G-QuAT), National Institute of Advanced Industrial Science and Technology (AIST), Tsukuba, Ibaraki 305-8568, Japan,

^gIBM Quantum, IBM Research Europe, Hursley, United Kingdom,

^hPlaksha University, Mohali, India,

ⁱPeter Grünberg Institute -Functional Quantum Systems (PGI-13),

Forschungszentrum Jülich GmbH, D-52425 Jülich, Germany,

^jUnitary Foundation, Walnut, California 91789, USA,

^kIBM Quantum, IBM T. J. Watson Research Center, Yorktown Heights, New York 10598, USA,

^lData and AI Division, Rakuten, Tokyo, Japan,

^mWallenberg Centre for Quantum Technology, Department of Microtechnology and Nanoscience, Chalmers University of Technology, 412 96 Gothenburg, Sweden,

ⁿZurich Instruments, Zurich, Switzerland,

^oDepartment of Mathematics, Aberystwyth University, Penglaus Campus, Aberystwyth, SY23 3BZ, Wales, United Kingdom,

^pQuantum Research Institute and Physics Department, University of Michigan, Ann Arbor, MI 48109-1040, USA,

Abstract

QuTiP, the Quantum Toolbox in Python [1, 2], has been at the forefront of open-source quantum software for the past 13 years. It is used as a research, teaching, and industrial tool, and has been downloaded millions of times by users around the world. Here we introduce the latest developments in QuTiP v5, which are set to have a large impact on the future of QuTiP and enable it to be a modern, continuously developed and popular tool for another decade and more. We summarize the code design and fundamental data layer changes as well as efficiency improvements, new solvers, applications to quantum circuits with QuTiP-QIP, and new quantum control tools with QuTiP-QOC. Additional flexibility in the data layer underlying all “quantum objects” in QuTiP allows us to harness the power of state-of-the-art data formats and packages like JAX, CuPy, and more. We explain these new features with a series of both well-known and new examples. The code for these examples is available in a static form on GitHub [3] and as continuously updated and documented notebooks in the qutip-tutorials package [4].

*Corresponding authors

Email addresses: nw.lambert@gmail.com (Neill Lambert), paul@menczel.net (Paul Menczel), fnori@riken.jp (Franco Nori)

Contents

1	Introduction	3
1.1	QuTiP v5	4
1.1.1	A new flexible data layer	4
1.1.2	Additional substantial changes	4
2	The QuTiP project	4
3	Core features	6
3.1	Quantum objects and the data layer	6
3.1.1	The <code>Qobj</code> class	6
3.1.2	The <code>QobjEvo</code> class	11
3.2	Solvers	12
3.2.1	A new solver class	12
3.2.2	Solver and integrator options	14
3.2.3	<code>mesolve</code> part 1: A master equation solver for Lindblad dynamics and beyond	15
3.2.4	<code>mesolve</code> part 2: Time-dependent systems	18
3.2.5	<code>mesolve</code> part 3: JAX and GPU acceleration with DiffraX	21
3.2.6	<code>steadystate</code> : A steady-state solver for master equations	24
3.2.7	<code>mcsolve</code> : A Monte Carlo solver for quantum trajectories	24
3.2.8	<code>nm_mcsolve</code> : A Monte Carlo solver for non-Markovian baths	26
3.2.9	<code>brmesolve</code> : Bloch–Redfield master equation solver	29
3.2.10	Floquet methods: the Floquet basis	31
3.2.11	<code>somesolve</code> : Stochastic master equation solver	33
3.2.12	<code>HEOMSolver</code> : Hierarchical Equations of Motion	34
3.2.13	Visualization of solver results	39
3.3	Additional features in QuTiP v5	41
3.3.1	Excitation number restricted states	41
3.3.2	Automatic Differentiation: JAX	45
3.3.3	MPI support for high performance computing	49
4	QuTiP’s other main packages	50
4.1	Optimal control: QuTiP-QOC	50
4.1.1	The GRAPE algorithm	51
4.1.2	The CRAB algorithm	52
4.1.3	The GOAT algorithm	53
4.1.4	Integration with QuTiP-JAX	54
4.2	Quantum circuits: QuTiP-QIP	55
4.2.1	Circuit Visualization	55
4.2.2	Simulating Hamiltonian dynamics	56
4.2.3	Simulating master equation dynamics	58
5	QuTiP’s community	60
5.1	The admin team and governance	60
5.2	RIKEN	60
5.3	NumFOCUS and Google Summer of Code	61
5.4	The Unitary Foundation	61
5.5	Packages that use QuTiP	61
6	Future development	62
6.1	QuTiP’s role in the quantum computing revolution	63
6.2	QuTiP’s role in fundamental scientific research	63

7 Conclusion	64
8 Acknowledgements	65
Appendix A Tables	66
Appendix B Summary of tutorials and example notebooks	70
Appendix B.1 Time-evolution tutorials	70
Appendix B.2 Lectures	74
Appendix B.3 Quantum circuits and Pulse-level-circuit-simulation	74
Appendix B.4 Visualization	74
Appendix B.5 HEOM: Hierarchical Equations of Motion	74
Appendix B.6 Miscellaneous	74
Appendix B.7 QuTiP-notebooks	75

1. Introduction

Open-source software plays an important role across a range of scientific disciplines, and is important for reproducibility in scientific research [5], enabling scientific education, and the transfer of academic ideas into industrial applications. Examples include the KWANT library for condensed matter physics [6], Quantum ESPRESSO for density functional theory [7], MDTraj for molecular dynamics [8], and the The Astropy Project for astrophysics [9], to name just a few. With the increasing interest in quantum computing, the need for open-source tools for the study of quantum noise, quantum dynamics and quantum circuits has exploded [10–14]. Among these tools, QuTiP, the Quantum Toolbox in Python, has remained one of the most widely used, consistently maintained, and academically independent.

In its first release about 13 years ago [1, 2], QuTiP originally aimed to reproduce, in Python, the functionality of the famous Quantum Toolbox for Matlab [15]. QuTiP’s initial design approach was focused around a flexible “quantum object” class which represents quantum states, operators and superoperators, and which allows the user to quickly and easily solve many of the standard problems that occur in the fields of quantum optics and open quantum systems [16, 17]. For this purpose, it relied – and still relies – on the extensive scientific computing infrastructure available in Python, including libraries such as SciPy [18], NumPy [5], Cython [19], and Matplotlib [20]. Many of QuTiP’s original examples focused on traditional models from cavity quantum electrodynamics, like Lindblad master equation simulations of the open Jaynes-Cummings, Rabi and Dicke models, and tools to quickly calculate standard observable quantities associated with such systems, such as the Wigner function or the photonic $g^{(2)}(t)$ function.

Continuous development over the last 13 years by a large team of international contributors expanded the scope of QuTiP beyond traditional quantum optics, adding important features like:

- optimal control methods,
- quantum circuit simulators [21],
- a solver for the hierarchical equation of motion that describe non-Markovian dynamics [22],
- a solver that takes advantage of permutational symmetries [23],
- stochastic master equation solvers,
- Floquet methods

and more. Many of these improvements were done by students and first-time developers at Franco Nori’s group in RIKEN, including some who were supported by the Google Summer of Code program [24]. For a more detailed overview of this era of QuTiP’s development history, we refer to Sec. 2 below.

1.1. QuTiP v5

In the last several years, the development team has pushed towards a new milestone release for QuTiP, v5, which combines multiple deep changes to the internals of QuTiP. These changes were originally designed and spearheaded in a project by Jake Lishman under the supervision of Eric Giguère and Alex Pitchford. The changes primarily focus on a generalization of the QuTiP data layer, which previously only supported a single data format: the Compressed Sparse Row (CSR) format. This format is a good choice for many applications, but other formats had to be introduced in an ad-hoc way sometimes. An example is QuTiP’s optimal control library, where the exponentiation and multiplication of many small matrices can be performed more efficiently with a dense data format.

1.1.1. A new flexible data layer

It was thus recognized that a flexible data format would make significant performance improvements possible, and that it would also allow for the use of packages like CuPy, JAX, and cuquantum, which make integration with GPU and XLA hardware easier and provide powerful features like automatic differentiation. For example, an earlier effort [25] to make use of JAX had required sweeping changes throughout the entire QuTiP repository, which implied having to copy and maintain the code in parallel.

In QuTiP v5, a new flexible data layer takes center stage and facilitates new features to be included easily and quickly. At the time of the release of QuTiP v5, it already supports several new data formats in addition to the existing CSR format: the native “Dia” and “Dense” formats (for diagonal sparse matrices and for dense matrices, respectively), and two JAX-based formats for diagonal sparse and dense matrices provided through the QuTiP-JAX optional sub-package. In this article, we will carefully demonstrate, using a variety of examples and benchmarks, the circumstances in which these different data formats can be used and taken advantage of.

1.1.2. Additional substantial changes

In addition, QuTiP v5 includes a solver class interface, new methods for the integration of differential equations, updated tutorials and examples (in the form of Jupyter notebooks), and a large amount of miscellaneous improvements and bug fixes identified by new unit tests. Furthermore, starting with QuTiP v4.7, there has been a concerted effort to reduce the complexity and weight of the core QuTiP package by moving feature-rich aspects into their own sub-packages. This is exemplified by QuTiP-QIP and QuTiP-QOC, which now contain the most recent versions of the circuit simulator and optimal control libraries, respectively. This strategy reduces the maintenance cost of the core package, and the chance that dependencies in these sub-packages will break or interrupt core features.

The aim of this article is to provide a detailed explanation, with examples, of existing and new features of QuTiP. We chose examples which either demonstrate unusual use cases for QuTiP not covered already in the documentation, or ones which allow us to compare the regime of validity of different solvers. We end with an outlook and strategy for the future development of QuTiP into the next decade.

2. The QuTiP project

QuTiP began more than ten years ago as a collaborative project between Robert Johansson and Paul Nation, then two postdoctoral researchers in the group of Franco Nori in RIKEN, Japan. At that time, well-maintained and easy-to-use open source software packages for implementing common numerical methods in the fields of open quantum systems, quantum optics and quantum information were limited. One of the most widely known packages was the Quantum Toolbox for Matlab, developed by Sze M. Tan at the University of Auckland. However, it had not seen active development since 2002, and relied on the commercial closed-source Matlab environment.

The programming language Python has seen extensive adoption across academia, particularly in the data science community. Its easy-to-read philosophy and its quickly increasing support for scientific calculations through packages like SciPy and NumPy made it very appealing as a platform for the development of a modern re-implementation of the Quantum Toolbox. To make the new package appealing to the community,

its developers adopted the philosophy of mirroring the feature set and some of the syntax of the Quantum Toolbox in Matlab (nowadays, new packages like the `quantumtoolbox.jl` library for the Julia programming language are, in turn, being built to mirror QuTiP). This culminated in the first release of QuTiP in 2012 with a feature-set comparable to Matlab's toolbox [1].

Only one year later, QuTiP v2 was released [2]. Alongside API and efficiency improvements, it included solvers supporting arbitrary time-dependent Hamiltonians and collapse operators, and new solvers for Bloch–Redfield and Floquet–Markov master equations.

QuTiP v3 was released in 2014, including stochastic master equation and stochastic Schrödinger equation solvers, a broader range of methods for finding steady states, and the first version of a circuit simulator called `qutip.qip`.

The road from QuTiP v3 to v4 took more time, with the release of version 4.0 occurring only in late 2016. Minor releases in between introduced important new features like a hierarchical equations of motion (HEOM) solver and a new module for optimal control. The optimal control module, then named `qutip.control`, included support for the powerful gradient ascent pulse engineering (GRAPE) and chopped random basis (CRAB) algorithms.

After the release of version 4.0, the development of QuTiP underwent a transition to a series of many minor releases. In addition, the development team grew to a larger international team including full-time developers, volunteers contributing specialized functionality (like improvements of the optimal control and HEOM modules, or the permutational invariant quantum solver) and students who joined the team through internships at RIKEN or through Google Summer of Code projects.

Significant releases during this time include:

- a time-dependent Bloch–Redfield equation solver in v4.2 (2017),
- the permutational invariant quantum solver (PIQS [23]) in v4.3.1 (2018),
- the introduction of the `QObjEvo` class in v4.4 (2019) and
- the first major update to the circuit simulator QuTiP-QIP in v4.5 (2020).

QuTiP v4.6 (2021) saw:

- further improvements to QuTiP-QIP,
- OpenQASM support,
- and the release of binary wheels on pip, which made continued support for Windows and other platforms much easier.

Finally, QuTiP v4.7 (2022) brought:

- a major update to the HEOM solver and
- the introduction of a Krylov subspace solver.

In February 2023, the first alpha pre-release version of QuTiP v5 was published.

As described earlier, version 5 is a substantial new release. It includes deep and far-reaching changes to many of the core components of QuTiP. To avoid bugs or errors plaguing users, or incompatibility with old code, the pre-alpha and alpha development stages stretched over all of 2023. In March 2024, a large QuTiP developer's workshop was held in Franco Nori's group at RIKEN and, during this workshop, QuTiP v5 was finally fully released. It represents a substantial reinvention of what QuTiP can achieve and validates the success of the academic support of open-source science, of programs like JST Moonshot and GSoC, and of non-profit organizations such as NumFOCUS and the Unitary Foundation.

3. Core features

3.1. Quantum objects and the data layer

At its core, QuTiP is a library for manipulating arbitrary quantum objects and for solving the time evolution of both open and closed quantum systems. Its aim is to remove burdens from the researcher and to give them interactive programmatic access to descriptive objects and solvers.

During its history, it has generally aspired to versatility and ease of use rather than optimization or keeping up with state-of-the-art benchmarks. However, as the community using quantum software has grown and demands for the support of high-performance computing platforms have risen, a fundamental change in QuTiP's concept of a quantum object was needed to remain competitive and useful. Towards this goal, QuTiP v5's design enables the support of arbitrary data formats through a flexible and powerful data layer, which features dynamic conversion between data types. This means automatic conversion when objects of different type are combined together, and the option to choose the data type that is optimal for a particular task.

Method	Description
<code>qobj.to(x)</code>	Change the data layer to the format specified by the string <code>x</code> , which may be " <code>dense</code> ", " <code>csr</code> ", or " <code>dia</code> ". If QuTiP-JAX is installed, " <code>jax</code> " and " <code>jaxdia</code> " are also available.
<code>dtype=x</code>	When creating quantum objects, most functions allow the user to specify the data-layer type <code>x</code> (allowed values described above).
<code>qt.CoreOptions(default_dtype=x)</code>	Setting this option will cause quantum objects created with most internal functions to use the same data-layer type <code>x</code> (allowed values described above) by default.
<code>qobj.data_as()</code>	Returns the raw data defining the quantum object in its current format.

Table 1: Summary of methods to change the data-layer in a Qobj

3.1.1. The Qobj class

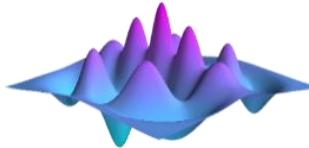
To understand this fundamental change, we have to introduce the cornerstone of QuTiP, the Python class `Qobj`, which provides an intuitive way to store and manipulate commonly used quantum objects. In most instances, `Qobjs` are used to store quantum states (vectors), operators or super-operators, and manipulate them according to the rules of linear algebra.

For example, a Pauli-Z matrix can be simply defined either using the built-in function `sigmaz()` or by constructing a `Qobj()` from a matrix:

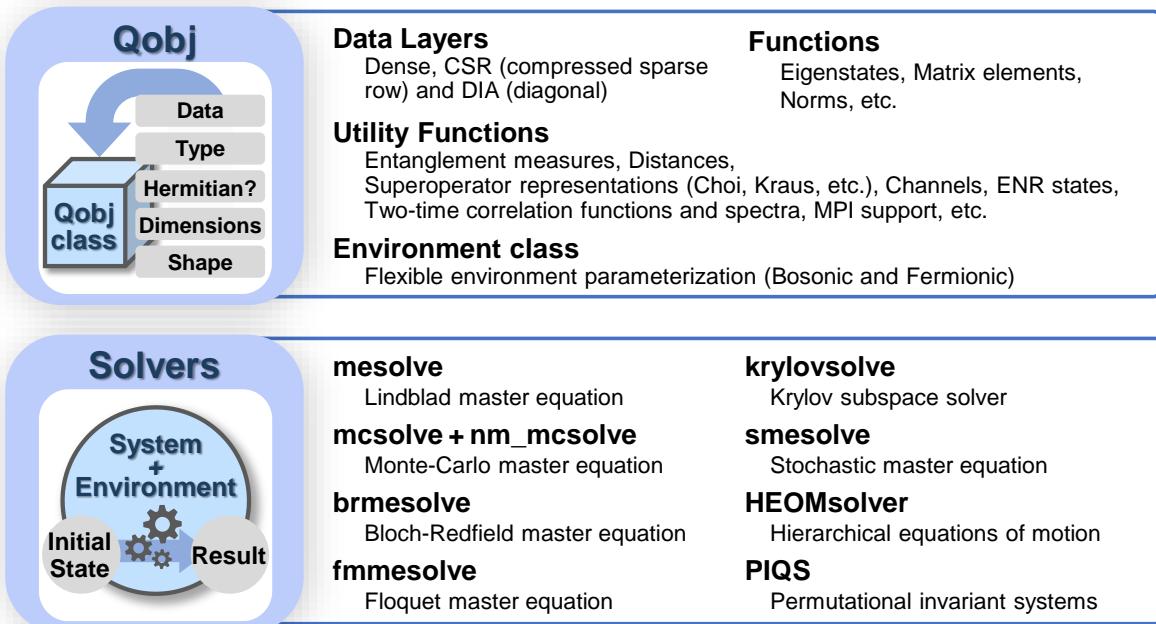
```
import qutip as qt
sz = qt.Qobj([[1, 0], [0, -1]])
print(sz)
Quantum object: dims=[[2], [2]], shape=(2, 2), type='oper', dtype=Dense, isherm=True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]
```

This `Qobj` is an operator, as indicated by its `type`, the `shape` of its matrix representation, and its dimensions (`dims`). The dimensions allow us to keep track of subsystems and indirectly infer the type of the quantum object itself. For example, in this case the object is an operator on a single sub-system, and can be thought of as a map which takes vectors to vectors.

Overview of QuTiP



1 QuTiP - Core



2 QuTiP - Packages

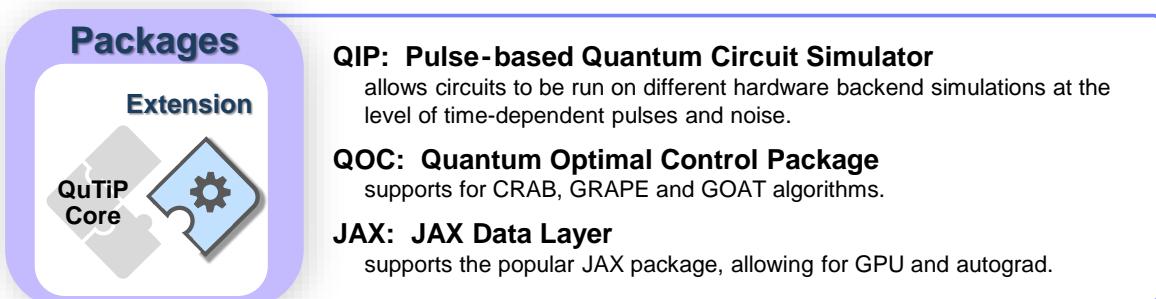


Figure 1: A schematic overview of the QuTiP project, describing **Qobj** and its features/functions, solvers, and QuTiP sub-packages. For a complete list of **Qobj** methods and attributes see Table A.11, for a list of libraries which QuTiP uses see Table 2, for a glossary of terms commonly used in QuTiP see Table 3, and for a list of state, operator, superoperator, entanglement measure and metric functions see Tables 4, 5, A.12, A.13 and A.14.

Acronym	Description
NumPy	Numerical Python: A library for the Python programming language that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
SciPy	Scientific Python: A Python library used for scientific and technical computing, building on the capabilities of NumPy. It includes functions for optimization, integration, interpolation, eigenvalue problems, and other advanced mathematical operations.
Matplotlib	A plotting library for the Python programming language, used for creating static, interactive, and animated visualizations.
JAX	A library developed by Google for high-performance numerical computing that is particularly useful for machine learning and scientific computing, with automatic differentiation capabilities.
Diffrax	Differential Equations in JAX: A library that provides numerical solvers for differential equations, compatible with JAX for high-performance computation.
MKL	Math Kernel Library: A library developed by Intel that provides optimized mathematical routines.
MPI	Message Passing Interface: A standardized and portable message-passing system used for parallel computing.

Table 2: Glossary of acronyms and terms for libraries used by QuTiP

Compare the output above to a tensor product of two Pauli operators:

```
sz = qt.Qobj([[1, 0], [0, -1]])
sz2 = sz & sz
print(sz2)
Quantum object: dims=[[2, 2], [2, 2]], shape=(4, 4), type='oper',
dtype=Dense, isherm=True
Qobj data =
[[ 1.  0.  0.  0.]
 [ 0. -1.  0. -0.]
 [ 0.  0. -1. -0.]
 [ 0. -0. -0.  1.]]
```

Here, we used the `&` operator introduced in version 5, which denotes the tensor product (instead, we could also have used the `tensor()` function like in earlier versions of QuTiP). We see that the dimensions have the structure `[[2,2],[2,2]]`, which implies that the operator acts on vectors that live in the tensor product of two 2-state subsystems. In contrast, an operator acting on a single 4-state system would have the dimensions `[[4],[4]]`. A more complex version of this sub-system labeling can be seen with superoperators, which map operators to operators, and will be explored later when the Lindblad master equation solver is introduced.

In previous versions of QuTiP, the data actually defining the quantum object, a vector or matrix of complex numbers, was usually represented in terms of SciPy’s implementation of the “Compressed Sparse Row” (CSR) matrix format for complex numbers. This was a convenient format for many problems in open quantum systems, where density operators and superoperators are often naturally sparse, since the CSR format provides a fast matrix-vector product. However, for studying the dynamics of small quantum systems or performing computations on specialized hardware like GPUs, other types of data formats are preferred.

QuTiP v5 introduces a new data layer which allows for custom data formats. The standard formats included and available in v5 are `CSR`, `Dense` and `Dia` (diagonal sparse). For example, the operator we defined earlier was constructed from dense data provided by the user and is hence in the `Dense` format, as indicated by the `dtype` in the output above. It can be converted to another format with `.to()`:

Term	Description
CSR	Compressed Sparse Row: A SciPy data format for sparse matrices.
ENR States	Excitation Number Restricted States: Quantum states where the total number of excitations across subsystems is restricted, reducing the Hilbert space dimension size drastically.
Floquet	Floquet theory is a tool for the description of periodically driven systems. Some QuTiP solvers are based on Floquet theory.
HEOM	Hierarchical Equations of Motion: A formalism used in quantum dynamics to describe non-Markovian open quantum systems.
ODE	Ordinary Differential Equation: The QuTiP solvers allow users to choose from various numerical ODE integration methods.
PIQs	Permutationally Invariant Quantum Solver [23]: A solver in QuTiP for efficiently simulating systems that have permutational symmetry, reducing computational complexity.
QuTiP-JAX	QuTiP-JAX: A module in QuTiP that leverages JAX, allowing the use of GPUs and automatic differentiation.
QuTiP-QIP	QuTiP Quantum Information Processing: A module in QuTiP that provides tools for simulating quantum circuits, quantum algorithms, and other aspects of quantum information processing.
QuTiP-QOC	QuTiP Quantum Optimal Control: A module in QuTiP for designing and optimizing quantum control pulses, allowing users to find control solutions for specific quantum dynamics tasks.
GRAPE	Gradient Ascent Pulse Engineering: A numerical algorithm used in quantum optimal control. It uses time discretization in order to identify the optimal control pulses for quantum systems.
CRAB	Chopped Random Basis: A method for quantum optimal control that uses a randomly truncated basis to optimize control pulses.
GOAT	Gradient Optimization of Analytic Controls: An optimal control method that uses analytical functions for the available controls.

Table 3: Glossary of acronyms and terms for large subpackages and methods available in QuTiP

```

print(sz.data)
| Dense(shape=(2, 2), fortran=False)
print(sz.to('CSR').data)
| CSR(shape=(2, 2), nnz=2)

```

Here, the parameter `fortran` refers to the layout of the dense data in the computer memory, and the parameter `nnz` to the number of non-zero entries in the sparse matrix.

When combining data formats, such as multiplying a sparse matrix with a dense vector, QuTiP performs conversions automatically. When new data formats are added, it is sufficient to define conversion methods between that new data format and only one existing format. QuTiP will then automatically use a graph of conversion methods to allow operations between that new format and any existing one. However, converting between data formats is generally not numerically optimal, and it is good practice to restrict oneself to the format that is most useful for the problem at hand. Specific examples will be provided later.

The true power of this approach lies in allowing new flexible data formats to be developed in the future. As an example, together with QuTiP v5 we are also releasing QuTiP-JAX [26], which takes advantage of the

Function	Description
<code>basis(N,n)</code>	Generates the vector representation of a Fock state, $ n\rangle$ in an N -dimensional Hilbert space. Also supports lists for construction of tensor spaces.
<code>qutrit_basis()</code>	Returns a list of basis states for a qutrit (three-level) system.
<code>bra(x)</code>	Produces a bra state given a list or string of excitation numbers. For example, $x = "01010"$ produces the bra state $\langle 01010 $.
<code>ket(x)</code>	Produces a ket state given a list or string, as with <code>bra</code> .
<code>fock(N,n)</code>	Bosonic Fock (number) state. Same as <code>basis</code> .
<code>coherent(N, α)</code>	Generates a coherent state with eigenvalue α for a harmonic oscillator in an N -dimensional truncated Hilbert space.
<code>spin_state(j,m)</code>	Generates the spin state with the quantum numbers j and m .
<code>spin_coherent(j, θ, φ)</code>	Generates the spin coherent state $ \theta, \phi\rangle$ for a spin- j system.
<code>projection(N,n,m)</code>	Generates the projection operator from state m onto state n , i.e., $ n\rangle\langle m $, for a system with Hilbert space dimension N .
<code>maximally_mixed_dm(N)</code>	Generates the maximally mixed density matrix for dimension N .
<code>fock_dm(N,n)</code>	Density matrix representation of the Fock state.
<code>coherent_dm(N, α)</code>	Density matrix representation of the coherent state.
<code>thermal_dm(N,n)</code>	Density matrix for the thermal state of a harmonic oscillator in an N -dimensional truncated Hilbert space.
<code>singlet_state()</code>	Returns the singlet state $ S\rangle = \frac{1}{\sqrt{2}}(01\rangle - 10\rangle)$.
<code>triplet_states()</code>	Returns a list of the triplet states $ 11\rangle$, $\frac{1}{\sqrt{2}}(01\rangle + 10\rangle)$ and $ 00\rangle$.
<code>bell_state(['00', '01', '10', '11'])</code>	Returns the selected Bell state, $ B_{00}\rangle = \frac{1}{\sqrt{2}}(00\rangle + 11\rangle)$, $ B_{01}\rangle = \frac{1}{\sqrt{2}}(00\rangle - 11\rangle)$, $ B_{10}\rangle = \frac{1}{\sqrt{2}}(01\rangle + 10\rangle)$, $ B_{11}\rangle = \frac{1}{\sqrt{2}}(01\rangle - 10\rangle)$.
<code>ghz_state(N)</code>	Produces the N -qubit GHZ-state, e.g., $\frac{1}{\sqrt{2}}(0000\rangle + 1111\rangle)$ for 4 qubits.
<code>w_state(N)</code>	Returns the W state of N qubits, $\frac{1}{\sqrt{N}}(100\cdots 0\rangle + 010\cdots 0\rangle + \cdots + 00\cdots 1\rangle)$.

Table 4: List of commonly used functions to create pre-defined states

powerful JAX library for GPU-based performance enhancements and automatic differentiation [27]. Data formats for CuPy [28], TensorFlow [29] and tensor networks [30] have also been explored, and alpha versions of these data layer implementations are available. They will be expanded upon in future releases. We refer to Table 1 for a summary of methods to convert between data layer formats.

Note that the Array API consortium [31] aims to solve a similar problem (to standardize functionality available across Python libraries and frameworks), but our custom approach to this problem offers several benefits. Firstly, it enables incremental development of new data formats (if some solver or function does not support that new format, the dispatcher will convert to a format which is supported). Secondly, it enables interaction between different data formats; for example, it supports sparse-dense matrix-vector multiplication, which can be an optimal choice for some solvers. Finally, both of these features are powered by a multiple-dispatch system which, to our knowledge, is a unique innovation. For example, it goes beyond Julia’s multiple-dispatch system, which has “promote” rules that cannot handle the arbitrary heterogeneous data inputs of QuTiP’s implementation.

In addition to storing the data describing a quantum object, the `Qobj` class has a large range of built-

Function	Description
<code>commutator(A,B)</code>	Computes the commutator (or anti-commutator, given an additional optional argument) of two operators A and B .
<code>identity(dimensions)</code>	Returns the identity operator for a single or multipartite system described by the integer or list <code>dimensions</code> .
<code>qeye(dimensions)</code>	Alias for <code>identity(dimensions)</code> .
<code>qeye_like(qobj)</code>	Generates the identity operator with the same dimensions and type as the reference quantum object <code>qobj</code> .
<code>create(N), destroy(N)</code>	Generates the creation (raising) or annihilation (lowering) operator for an N -dimensional Fock-space.
<code>momentum(N), position(N)</code>	The momentum or position operator of a single N -dimensional harmonic oscillator, i.e., $p = \frac{i}{\sqrt{2}}(a - a^\dagger)$ or $x = \frac{1}{\sqrt{2}}(a + a^\dagger)$.
<code>num(N)</code>	Generates the number operator of a single N -dimensional harmonic oscillator, i.e., $n = a^\dagger a$.
<code>displace(N,α)</code>	Generates the displacement operator for a distance α in a single N -dimensional Fock space, i.e., $D(\alpha) = \exp(\alpha a^\dagger - \alpha^* a)$.
<code>squeeze(N,z)</code>	Generates the single-mode squeezing operator in a single N -dimensional Fock space, i.e., $S(z) = \exp[(z^* a^2 - z a^\dagger)^2/2]$.
<code>squeezing(a1,a2,z)</code>	Generates the two-mode squeezing operator for two harmonic oscillators predefined with annihilation operators <code>a1</code> and <code>a2</code> , i.e., $S_2(z) = \exp[(z^* a_1 a_2 - z a_1^\dagger a_2^\dagger)/2]$.
<code>fcreate(n,m), fdestroy(n,m)</code>	Generates the fermionic creation or annihilation operator using the Jordan-Wigner transformation. Returns the m -th fermionic operator out of n total fermions.
<code>sigmax(), sigmay(), sigmaz()</code>	Generates the Pauli-x, y or z operator.
<code>sigmam(), sigmap()</code>	Generates the lowering or raising operator for Pauli spins.
<code>spin_Jx(j),...</code>	Generates the spin-j, x, y or z operator.
<code>spin_Jm(j), spin_Jp(j)</code>	Generates the spin-j lowering or raising operator.
<code>jmat(j, which)</code>	Alias for <code>spin_Jx(j)</code> , <code>spin_Jy(j)</code> , <code>spin_Jz(j)</code> , <code>spin_Jm(j)</code> or <code>spin_Jp(j)</code> , depending on <code>which</code> ∈ "x", "y", "z", "-", "+".

Table 5: List of commonly used functions to create pre-defined operators

in utility functions to calculate common properties of quantum systems. For example, `eigenstates()` computes the eigenvalues and eigenstates of an operator, `expm()` takes the matrix exponential, `norm()` finds the norm of states and operators, and so on. We provide a full list of these functions in Table A.11 and lists of commonly used functions to create states, operators and superoperators in Tables 4, 5 and A.12. Examples of commonly used utility functions to calculate entropies, entanglement measurements, and distances between states are given in Tables A.13 and A.14.

3.1.2. The QobjEvo class

The `QobjEvo` class provides a useful extension of the `Qobj` class to describe time-dependent quantum objects and to optimize their use by QuTiP's solvers. As we will demonstrate later, when calling a solver with a time-dependent Hamiltonian or time-dependent bath operators, we typically specify the time-dependent operator by a list of tuples. Each tuple defines an operator and its time-dependent prefactor in the form of a Python function, an array or a string. This list of tuples is then internally converted into the `QobjEvo` class, which involves optimization in preparation for their use by the solvers.

However, `QobjEvo` objects can also be manually created and manipulated by the user, and they provide a flexible framework for dealing with multiple time-dependent systems that have their time-dependence specified in different ways. For example, they can be instantiated from continuously defined functions or from discrete time-dependent data. In the latter case, times in between data points, where the dependence is undefined, are filled in using a cubic spline interpolation. Different `QobjEvo` objects can be added, multiplied, etc., with each other and with constant `Qobj` and scalar objects. They also support many of the utility methods available to `Qobjs` like `dag()` (Hermitian conjugation) or `conj()` (complex conjugation). Further, they support tensor products with other objects and they can be converted into superoperators. We will demonstrate the utility of these features later.

3.2. Solvers

Armed with the `Qobj` and `QobjEvo` classes, one can simulate a large range of open quantum system dynamics using the solvers provided by QuTiP. These simulations are, for the most part, initial-value ordinary differential equation problems: given an initial state for a system, a Hamiltonian (possibly time-dependent) describing its energy and interactions, and an environment described for example in terms of rates or coupling strengths (depending on the solver), QuTiP uses numerical integration (either a custom method or one provided by SciPy) to find the evolution of the system as a function of time.

These solvers are also employed as needed in other QuTiP packages like QuTiP-QIP or QuTiP-QOC. For general tasks, one should consider the effects of the environment on the system under study and choose the solver based on the level of approximation one wishes to take. This concept, core to QuTiP and to open quantum systems in general, will be developed step by step in the following sections. A brief guide to the solvers, their fields of application and their memory usage is provided in Table 6.

3.2.1. A new solver class

A convenient change in QuTiP v5 is the introduction of a unified class interface to control how a user interfaces with the solvers. Using this class interface is optional, but useful when reusing the same Hamiltonian data with different initial conditions, time steps or options. This procedure usually provides only a minor numerical advantage, but the speed-up can be significant if the solver is reused many times.

When a solver is instantiated, one first supplies only the Hamiltonian and the operators defining the bath (e.g., collapse operators for a Lindblad master equation). Then, the initial condition and time steps are passed to the `Solver.run()` method, which performs the time evolution. Alternatively, one can also use the `Solver.start()` and `Solver.step()` methods in order to manually control the spacing of the time steps during the simulation.

We will first illustrate the usage of the solvers with an elementary example that builds upon the concepts introduced in the section on the `Qobj` class. We consider two interacting qubits, which are not interacting with an environment and thus are entirely defined by their Hamiltonian:

$$H = \frac{\epsilon_1}{2} \sigma_z^{(1)} + \frac{\epsilon_2}{2} \sigma_z^{(2)} + g \sigma_x^{(1)} \sigma_x^{(2)}. \quad (1)$$

```

epsilon1 = 1.0 #qubit 1 energy
epsilon2 = 1.0 #qubit 2 energy
g = 0.1 #coupling strength

sx1 = qt.sigmax() & qt.qeye(2)
sx2 = qt.qeye(2) & qt.sigmax()
sz1 = qt.sigmax() & qt.qeye(2)
sz2 = qt.qeye(2) & qt.sigmax()

#Hamiltonian
H = (epsilon1 / 2) * sz1 + (epsilon2 / 2) * sz2 + g * sx1 * sx2

print(H)
Quantum object: dims=[[2, 2], [2, 2]], shape=(4, 4), type='oper',
dtype=CSR, isherm=True
Qobj data =
[[ 1.  0.  0.  0.1]
 [ 0.  0.  0.1  0. ]
 [ 0.  0.1  0.  0. ]
 [ 0.1  0.  0. -1. ]]

```

The Hamiltonian is an operator with the same dimensions that we saw earlier, acting on the tensor product of two 2-state Hilbert spaces. We can see that the sparse CSR data format is used now, since we used the built-in functions `sigmax()` and `qeye()` to construct the operator.

The dynamics of a pure state of this system obeys the Schrödinger equation

$$i\hbar \frac{d}{dt} |\psi\rangle = H |\psi\rangle . \quad (2)$$

Here, the Hamiltonian H is the matrix in the code snippet above. The initial condition $|\psi(t=0)\rangle$ and the final time t must be provided to the solver at run time. For example, with the traditional approach (that is, without using the class interface), we would use the Schrödinger equation solver `sesolve()` with:

```

psi0 = qt.basis(2, 0) & qt.basis(2, 1) #initial state
tlist = np.linspace(0, 40, 100) #time steps
result = qt.sesolve(H, psi0, tlist) #solver result

```

Note that we have used the function `basis(N,n)` to construct the initial condition. This function creates a vector in an N -state Hilbert space with a 1 as the amplitude for the state $|n\rangle$ (note that for historical reasons, the operator `sigmax()` is defined such that `basis(2,0)` is the excited state and `basis(2,1)` is the ground state, a convention which sometimes confuses users). Alongside the standard rules of linear algebra and the use of the `tensor` function (or the equivalent `&` operator), this allows us to create any desired state.

With the new solver class interface, we can perform the same calculation with the following:

```

solver = qt.SESolver(H) #create solve object
result2 = solver.run(psi0, tlist) #get result from solver object

```

The result objects returned from both methods are equivalent. Since we did not specify any particular observables to evaluate, the solver returns the state vector of the system at all time points specified in the `tlist` (100 equally spaced steps between $t = 0$ and $t = 40$).

Finally, the aforementioned manual stepping interface would be used as follows:

```

t = 0
dt = 40 / 100
solver.start(psi0, t)
while t < 40:
    t = t + dt
    psi = solver.step(t)
    dt = ... # process the result psi and calculate the next time step

```

This interface is particularly useful if the Hamiltonian depends on external control parameters such as field strengths. Such parameters can be updated in each step using the optional parameter `args`.

Solver	When to use	Memory
<code>sesolve</code>	Closed system evolution following Schrödinger's equation.	d^2
<code>mesolve</code>	Describes completely positive (CP) Lindblad evolution. Used for Markovian environments with a flat spectral density and typically at weak system-environment coupling. This solver can also be used to simulate arbitrary quantum master equations with superoperator generators.	d^4
<code>brmesolve</code>	Also used for Markovian environments. In contrast to <code>mesolve</code> , no secular approximation is made, leading to non-CP dynamics. This solver can automatically diagonalize time-dependent system Hamiltonians at each time step, as required by the Born-Markov approximation.	d^4
<code>mcsolve</code>	Monte Carlo Wavefunction method. Converges to <code>mesolve</code> for large number of trajectories N_T . Use to simulate statistical transport properties, or if <code>mesolve</code> consumes too much memory.	$N_T d^2$
<code>nm_mcsolve</code>	Like <code>mcsolve</code> , but can be applied to non-CP master equations.	$N_T d^2$
<code>ssesolve</code>	To simulate systems subject to continuous (homodyne or heterodyne) measurement. Can include measurement feedback.	$N_T d^2$
<code>somesolve</code>	Like <code>ssesolve</code> , supports additional deterministic decay channels.	$N_T d^4$
<code>heomsolve</code>	Open quantum systems with Gaussian, bosonic or fermionic environments at arbitrary coupling strength and temperature. In the right column, N_c , N_R and N_I are the hierarchy cutoff and the number of real and imaginary exponents, see Sec. 3.2.12.	$d^4 \binom{N_c + N_R + N_I}{N_c}^2$
<code>FloquetBasis</code>	Periodically driven closed systems. One driving period is discretized into N_t time steps.	$N_t d^2$

Table 6: Overview of most solvers in QuTiP. The last column is the dimensionality of the dynamical generator, which (for dense generators) asymptotically equals the space complexity, that is, the memory usage of the simulation. Here, d is the Hilbert space dimension.

3.2.2. Solver and integrator options

The behavior and the output of the Schrödinger equation solver, and of all other solvers, can be finely controlled by making use of the optional `options` argument. In contrast to earlier versions, the options in QuTiP v5 are now specified by a Python dictionary instead of a custom class. This change increases the flexibility for future extensions and allows different solvers to provide different sets of options more easily.

There is a large range of options available; a full list is provided in the online documentation for each solver. Frequently used options include `store_states`, determining whether the system state at each time in the provided `tlist` should be included in the output, and `store_final_state`, determining whether the final state of the evolution should be included. These states are then included in addition to the computed values of any requested observables, see below. Other frequently used options are `method`, specifying the ODE integration algorithm, and specific options for that algorithm such as the desired numerical precision and the maximum steps used in the solver:

```

options = {"store_states": True, "atol": 1e-12, "nsteps": 1000, "max_step": 0.1}
solver.options = options
result3 = solver.run(psi0, tlist) # Or: qt.sesolve(H, psi0, tlist, options=options)
print(result3)
<Result
Solver: sesolve
Solver stats:
    method: 'scipy zvode adams'
    init time: 3.5762786865234375e-05
    preparation time: 0.0001609325408935547
    run time: 0.007315397262573242
    solver: 'Schrodinger Evolution'
Time interval: [0.0, 40.0] (100 steps)
Number of e_ops: 0
States saved.
>

```

In addition to using the `store_states` option, we have here shown how to increase the precision (absolute tolerance) `atol`, the maximum number `nsteps` of integration steps between two points in the `tlist`, and the maximum allowed integration step `max_step` of the default Adams ODE integration method. The `max_step` option is often important in time-dependent problems with periods of idling interspersed with short pulses; without setting a maximum time step for the solver to take, these short pulses might be ignored when the ODE solver takes too large time steps.

3.2.3. `mesolve` part 1: A master equation solver for Lindblad dynamics and beyond

While the time-dependent Schrödinger equation in principle describes the dynamics of any quantum system, it is often impossible to solve in practice when the number of constituent systems and the dimensionality of the Hilbert space become too large (this issue is known as the exponential explosion problem of quantum mechanics). This is problematic when we want to consider how a given finite quantum system is influenced by a large, perhaps even continuous, environment. Decades of research have lead to tractable solutions to this problem in many parameter regimes, and in QuTiP we aim to make many of these solutions readily and easily available to all.

Of these solutions, master equations are by far the most common. The term master equation generally refers to a first-order linear differential equation for $\rho(t)$, the reduced density operator describing the state of the finite (open) quantum system. Such equations can be derived in numerous ways, for example from various approximation schemes to microscopic system-bath models, from Hamiltonian-level stochastic noise approaches, or from abstract mathematical considerations about the general properties of completely positive trace-preserving maps of quantum states. In the latter context, the master equations take on a particularly nice form and are called Lindblad (or Gorini–Kossakowski–Sudarshan–Lindblad) master equations. It is this form which is the most commonly used, and by default implemented in QuTiP's `mesolve()` function. However, `mesolve()` can also be applied to master equations that do not have the Lindblad form, as we will show later.

The Lindblad master equation is an equation of motion for the density operator $\rho(t)$ of the open quantum system. It has the following form:

$$\dot{\rho}(t) = -\frac{i}{\hbar}[H(t), \rho(t)] + \sum_n \frac{1}{2} [2C_n \rho(t) C_n^\dagger - \rho(t) C_n^\dagger C_n - C_n^\dagger C_n \rho(t)]. \quad (3)$$

The use of a density operator instead of a state vector allows us to consider classical uncertainty in the description of the system. The density operator

$$\rho = \sum_k p_k |\psi_k\rangle\langle\psi_k|, \quad (4)$$

where $\sum_k p_k = 1$, describes a classical mixture of possible quantum states $|\psi_k\rangle$ occurring with probabilities p_k . For example, this kind of state describes a system in thermal equilibrium, where $|\psi_k\rangle$ are then the

eigenstates of the system's Hamiltonian and $p_k = \exp[-\hbar\omega_k/(k_B T)]/Z$. Here, $k_B T$ is the thermal energy, $\hbar\omega_k$ the eigenenergy, and Z the partition function. This particular state often arises as the long-time steady-state of master equations obeying the detailed balance condition (see below).

In addition to the system Hamiltonian, we see that the evolution (3) depends on so-called collapse operators, or jump operators, $C_n = \sqrt{\gamma_n} A_n$. Here, γ_n can generally be understood as rates describing the frequency of transitions between the states connected by the operators A_n . A standard way of deriving the Lindblad equation (3) from a microscopic description is using the Born-Markov and secular approximations; in this case, A_n is related to the environment coupling operators of the microscopic description, and γ_n is related to the square of the coupling strength. However, as mentioned before, one can also arrive at the Lindblad equation by considering the most general possible time evolution for ρ that is completely positive and trace preserving. This requirement means roughly that ρ will represent a physical state at all times, having eigenvalues that are positive and sum to one (as needed for a physical probability distribution). From the latter, mathematical point of view, A_n could be arbitrary operators on the system Hilbert space and γ_n arbitrary positive rates.

In QuTiP, the master equation solver `mesolve()` follows mostly the same syntax as the Schrödinger equation solver discussed earlier. The main difference is that collapse operators C_n , defining the dissipation due to contact with an environment, can be specified. Using the same Hamiltonian, time steps, and initial condition as before, we can add dissipation on both qubits by defining the following list of collapse operators: $C_1 = \sqrt{\gamma} \sigma_-^{(1)}$ and $C_2 = \sqrt{\gamma} \sigma_-^{(2)}$, where $\sigma_-^{(i)}$ is an operator which takes the qubit (i) from its excited state to its ground state.

```
sm1 = qt.sigmax() & qt.queye(2)
sm2 = qt.queye(2) & qt.sigmax()
gam = 0.1 # dissipation rate
c_ops = [np.sqrt(gam) * sm1, np.sqrt(gam) * sm2] #list of collapse operators for solver
psi0 = qt.basis(2, 0) & qt.basis(2, 1) #initial state (qubit 1 excited)
tlist = np.linspace(0, 40, 100) #time steps

result_me = qt.mesolve(H, psi0, tlist, c_ops, e_ops=[sz1, sz2]) #run mesolve
```

We have also provided a list of expectation operators in the argument `e_ops`. This argument tells the solver that we want to evaluate and return the expectation values $\langle E_n \rangle = \text{tr}[E_n \rho(t)]$ for each observable E_n in the list `e_ops` at every time step in `tlist`. If the `e_ops` argument is provided, the list of system states will not be stored unless asked for by providing the flag `store_states` in `options`.

In this example, the collapse operators were introduced phenomenologically and act locally, or independently, on each qubit. However, different choices of collapse operators are possible; one can derive a variety of master equations from microscopic models under different types of approximations. For example, assuming that the qubits are interacting more strongly with each other than with the bath, one arrives at a so-called global master equation under the standard Born-Markov-secular approximations. The global master equation involves collapse operators that act like annihilation and creation operators on the total, coupled, eigenstates of the interacting two-qubit system,

$$A_{ij} = |\psi_i\rangle\langle\psi_j|, \quad (5)$$

and rates

$$\gamma_{ij} = |\langle\psi_i| d|\psi_j\rangle|^2 S(\Delta_{i,j}). \quad (6)$$

Here, the states $|\psi_i\rangle$ are the eigenstates of H and $\Delta_{i,j} = E_j - E_i$ are differences of eigenenergies. Furthermore, d is the coupling operator of the system to the environment. The power spectrum

$$S(\omega) = 2J(\omega)[n_{th}(\omega) + 1]\theta(\omega) + 2J(-\omega)[n_{th}(-\omega)]\theta(-\omega), \quad (7)$$

depends on details of the environment like its spectral density $J(\omega)$ and its temperature through the Bose-Einstein distribution $n_{th}(\omega)$. Here, θ is the Heaviside function. These rates now obey the detailed balance condition

$$\gamma_{ij}/\gamma_{ji} = \exp[\Delta_{i,j}/(k_B T)]. \quad (8)$$

Assuming just a flat spectral density $J(\omega) = \gamma/2$ and zero temperature gives $S(\omega) = \gamma\theta(\omega)$, i.e., only spontaneous decay can occur, no stimulated decay or excitation can happen. In the following example, we implement this zero-temperature environment manually for our two-qubit model using `mesolve()`. We see that the long-time evolution leads to a state that is close to the coupled ground state of the two-qubit Hamiltonian:

```
def power_spectrum(w):
    if w >= 0:
        return gam #flat power spectrum
    else:
        return 0 #Zero temperature: only has support on positive frequencies

all_energy, all_state = H.eigenstates() #get eigenstates (ES)
Nmax = len(all_state)
collapse_list = []
for i in range(Nmax):
    for j in range(Nmax):
        deE = (all_energy[j] - all_energy[i]) #energy splitting between ES
        rate = power_spectrum(deE) * (
            np.absolute(sx1.matrix_element(all_state[i].dag(), all_state[j])) ** 2
            + np.absolute(sx2.matrix_element(all_state[i].dag(), all_state[j])) ** 2
        ) #Rates for both qubits
        if rate > 0: #add to collapse operator list
            collapse_list.append(np.sqrt(rate) * all_state[i] * all_state[j].dag())

tlist_long = np.linspace(0, 1000, 100) #time steps
result_me_global = qt.mesolve(H, psi0, tlist_long, collapse_list) #run mesolve
fidelity = qt.fidelity(result_me_global.states[-1], all_state[0] @ all_state[0].dag())
print(f"Fidelity with ground-state: {fidelity:.6f}") #compare final state with ground state
| Fidelity with ground-state: 1.000000
```

In Fig. 2, we show the results of the local and dressed (global) Lindblad simulations and compare the results with the Bloch-Redfield solver. The Bloch-Redfield solver, which will be explained later, is designed to automatically construct a weak-coupling master equation from a given bath power spectrum, allowing for time-dependent system Hamiltonians and arbitrary degrees of the secular approximation. If the secular approximation is fully implemented, using the Bloch-Redfield solver is equivalent to the global master equation that we constructed above manually. We find that when the qubit-qubit coupling is small, the results from the local and global master equations both agree well with the Bloch-Redfield solver. For large qubit-qubit coupling, the local master equation produces deviating results from the global/Bloch-Redfield approach.

For advanced applications of QuTiP, it is important to understand how a Lindblad equation is constructed internally. In order to start a simulation, the user must first pass a Hamiltonian and collapse operators to QuTiP, which are operators, and an initial condition, which may be given as a vector (for a pure state) or an operator (for a mixed state). However, internally, QuTiP uses the superoperator representation to solve the Lindblad equation (3). In short, this means that the density matrix representing the system state is stacked, in column order, into a single large vector. The right-hand-side of the Lindblad equation, called the Liouvillian or the Lindbladian, is a superoperator acting on this new larger vector space. It can thus be represented as a large matrix, which multiplies the new larger state vector only from the left.

Formally, this representation takes advantage of the isomorphism between operators acting on a Hilbert space and vectors in an enlarged space consisting of two copies of the original Hilbert space, $\mathcal{L}(\mathcal{H}) \cong \mathcal{H} \otimes \mathcal{H}$. One can visualize this as mapping the component of a density operator corresponding to the basis projector $|\psi_i\rangle\langle\psi_j|$ to a component in a new vector on the double space corresponding to the basis vector $|\psi_j\rangle\otimes|\psi_i\rangle$. Then, operators acting on the original density operator element from the left and right, $A|\psi_i\rangle\langle\psi_j|B$, become operators acting just from the left as $(B^\dagger \otimes A)(|\psi_j\rangle\otimes|\psi_i\rangle)$.

This construction makes the construction of the actual matrix describing the Lindbladian formally very simple, and it is usually numerically more straightforward to solve an ODE in a matrix-vector format, as opposed to a matrix-matrix format. However, it comes with the downside that the memory cost increases

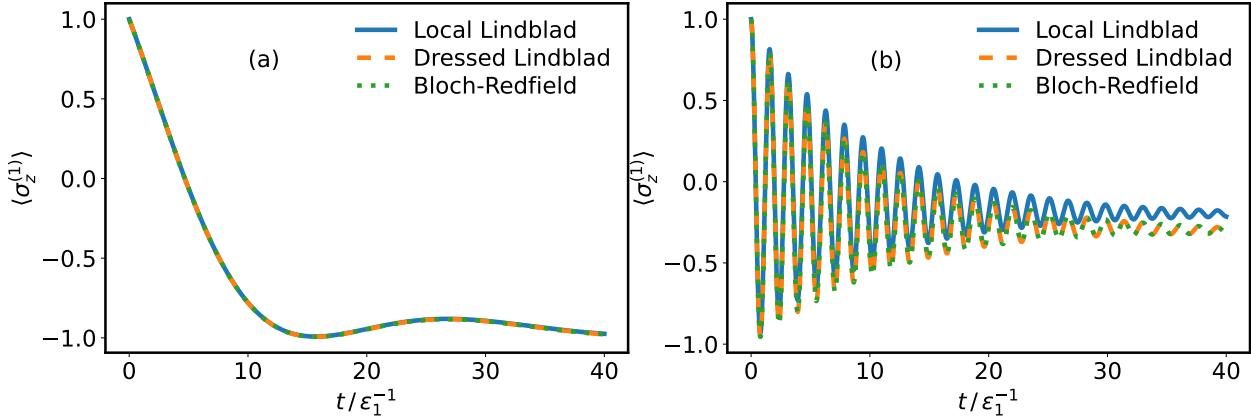


Figure 2: For the example problem of two interacting qubits, we compare the output of two `mesolve()` simulations. One uses local collapse operators acting on the bare states of each qubit (local Lindblad equation), and the other uses dressed collapse operators acting on the global eigenstates (global Lindblad equation). In addition, we include a simulation performed with the Bloch-Redfield solver (`brmesolve()`). Panel (a) shows weak coupling between the qubits ($g = 0.1\epsilon_1$), where the local Lindblad description is sufficient from the perspective of detailed balance. Panel (b) shows the dynamics with strong coupling ($g = 2\epsilon_1$), where local collapse operators predict a steady state that is not compatible with detailed balance. In both cases, the qubits are resonant $\epsilon_1 = \epsilon_2$.

quadratically due to the doubling of the Hilbert space. When this cost becomes prohibitive, one can consider employing the Monte-Carlo solver described in the next section. In future QuTiP versions, we plan to implement more memory-efficient alternative ways of storing the Lindbladian by avoiding vectorization (as done in the `dynamiqs` package [32]).

As noted before, QuTiP’s master equation solver is not restricted to just solving Lindblad equations like (3), but can also be applied to any other master equation. The Liouvillian superoperators for other master equations must be constructed by hand, for example using the built-in functions `spre()`, `spos()` and `sprepost()`, which convert operators on the original Hilbert space to operators in the double space discussed earlier (see Table A.12 for a list of more superoperator-related functions). These Liouvillians can be then passed directly to `mesolve()` in place of the system Hamiltonian. For example, the Lindbladian corresponding to the master equation of the earlier example can be constructed manually via:

```
lindbladian = -1.0j * (qt.spre(H) - qt.spost(H)) #Hamiltonian term
for c in c_ops:
    lindbladian += (
        qt.sprepost(c, c.dag()) - 0.5 * (qt.spre(c.dag() * c) + qt.spost(c.dag() * c))
    ) #Manual construction of Lindblad
```

In QuTiP v5, the master equation solver can be used with a new solver class interface, like we have seen earlier for the Schrödinger equation solver. It has also been augmented with new options for ODE integration methods: Verner’s “most efficient” Runge-Kutta methods of order 7 and 9 are available as `vern7` and `vern9`, and an approach based on diagonalization for time-independent systems is available as `diag`. Furthermore, the solver has been updated to take advantage of the new data layers. In section 3.2.5, we will demonstrate how to use the QuTiP-JAX data layer to take advantage of custom data layers to enhance `mesolve()` using GPUs.

3.2.4. `mesolve` part 2: Time-dependent systems

As mentioned in the earlier summary of the `QobjEvo` class, QuTiP and most of its solvers support time-dependent quantum objects. The user may specify the time dependence in a variety of ways: as a Python function, as a string (which will be compiled into machine code at the first usage), or as discrete time-dependent data which will be interpolated with cubic splines by QuTiP.

Usually, the user does not need to deal with `QobjEvo` objects themselves, and can just directly provide the time-dependence to the solver. Consider a standard example of a driven qubit with the time-dependent

Hamiltonian

$$H = \frac{\Delta}{2}\sigma_z + \frac{A}{2}\sin(\omega_d t)\sigma_x, \quad (9)$$

which experiences dissipation through contact with a zero-temperature bath with the coupling rate γ . We model the influence of the bath by a Lindblad equation with the collapse operator σ_- in the undriven basis. The assumption that the noise acts in the undriven basis is valid when the drive amplitude A is much smaller than the natural system frequency Δ .

If we assume in addition that the driving is close to resonant, $\omega_d \approx \Delta$, we can perform the rotating wave approximation to find, in a rotating frame,

$$H_{\text{RWA}} = \frac{\Delta - \omega_d}{2}\sigma_z + \frac{A}{4}\sigma_x, \quad (10)$$

which reduces the problem to an undriven one. This is a useful consistency check on the driven results for this simple example.

Defining the driven setup in QuTiP is easy; we first enter the driving field as a Python function:

```
def f(t):
    return np.sin(omega_d * t)
```

Then the undriven and driven parts of the Hamiltonian can be defined with:

```
H0 = Delta / 2.0 * qt.sigmax()
H1 = [A / 2.0 * qt.sigmax(), f]
H = [H0, H1]
```

This specification of the time-dependent Hamiltonian can be passed to `mesolve` as usual:

```
c_ops_me = [np.sqrt(gamma) * qt.sigmax()]
me_result = qt.mesolve(H, psi0, tlist, c_ops=c_ops_me, e_ops=e_ops)
```

Note that, as mentioned, the collapse operator is constant in time, i.e., it acts on the undriven basis. Alternatively, we could have entered the driving field as a string:

```
f = f"sin({omega_d} * t)"
```

or as time-dependent data:

```
f = np.sin(omega_d * tlist)
```

In the latter case, one has to take care that the spacing of the times in `tlist` is sufficiently smaller than the period length.

To check the validity of a given model, it is useful to compare to other solvers. In this case, we will compare to the Bloch-Redfield solver and to the solver for the hierarchical equations of motion (HEOM). More details on these will be given later.

For the Bloch-Redfield solver, we need to specify the bath's power spectrum, which captures how it affects the system at different frequencies. As in the previous example, we will use a white-noise bath at zero temperature. For the reader's convenience, the following code snippet, which defines the power spectrum and invokes the Bloch-Redfield solver, demonstrates also how a finite-temperature environment could be handled:

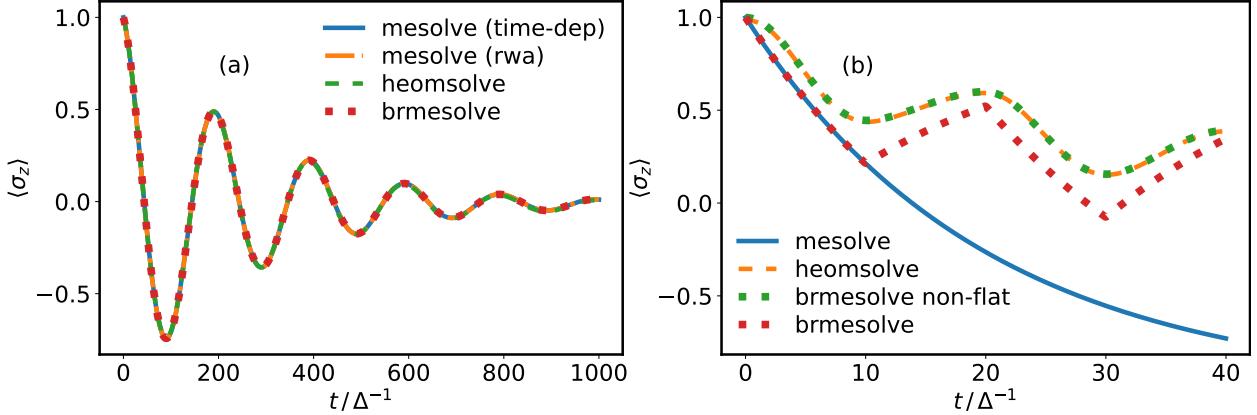


Figure 3: In panel (a), we show the dynamics of a resonantly driven qubit simulated with `mesolve`, `brmesolve` and the HEOM solver. We also compare the results to a simulation of the time-independent RWA Hamiltonian, Eq. (10). The bath is chosen to have zero temperature and a flat spectral density $J(\omega) = \gamma$ with a rate $\gamma = 0.005\Delta/(2\pi)$. The drive is assumed to be weak ($A = 0.01\Delta$) and on resonance ($\omega_d = \Delta$). In panel (b), we show the second example, where the energy of a qubit is adiabatically, sinusoidally modulated between positive and negative values with frequency $\omega_d = 0.05\Delta$ and amplitude $A = \Delta$. The qubit is in contact with a flat zero-temperature bath with rate $\gamma = 0.05\Delta/(2\pi)$. The naive `mesolve` implementation with a single collapse operator that is not sensitive to the system's energy predicts just exponential decay. The Bloch-Redfield and HEOM solvers predict reversal of the decay when the energy changes sign, as a true zero-temperature bath can only remove energy from the system. The `heomsolve` result differs slightly from the `brmesolve` result since the spectral density used in `heomsolve` is not flat. When the same spectral density used in the HEOM method is used explicitly in the Bloch-Redfield solver both approaches agree. This is shown with the green dotted line which we called `brmesolve non-flat`, which uses the same spectral density as HEOM.

```

def nth(w): # Bose einstein distribution
    if temp > 0:
        return 1 / (np.exp(w / temp) - 1)
    else:
        return 0

def power_spectrum(w):# Power spectrum
    if w > 0: #Influences emission
        return gamma * (nth(w) + 1)
    elif w == 0:
        return 0
    else: #Influences absorption
        return gamma * nth(-w)

a_ops = [[qt.sigmax(), power_spectrum]] #system operator which couples to bath, and
                                         #bath power spectrum are needed
brme_result = qt.brmesolve(H, psi0, tlist, a_ops=a_ops, e_ops=e_ops, sec_cutoff=-1)

```

Here, H is the same time-dependent system Hamiltonian that was defined before.

For the HEOM solver, the process is much more involved; we need to specify a multi-exponential decomposition of the correlation functions of the bath. However, the correlation functions are non-exponential at zero temperature, requiring us to apply a multi-exponential fit in order to use the HEOM approach. A full example of this process is given in the complete code example available on GitHub [3]. In Fig. 3a, we show a comparison of the resonantly driven qubit simulated with all three time-dependent solvers as well as the time-independent rotating wave approximation. As expected, all results agree with each other very well.

A second example, to illustrate where using naive local-basis collapse operators can fail, is that of a single qubit whose energies are adiabatically switched between positive and negative values,

$$H = \frac{\Delta}{2} \sin(\omega_d t) \sigma_z. \quad (11)$$

When the drive is slow, we expect the bath to be able to respond to this change. It should therefore always induce transitions from the higher-energy state to the lower-energy one, extracting energy from the system. As we see in Fig. 3b, a feature of the Bloch-Redfield solver and the HEOM method is that they can capture this switching effect automatically. The naive approach, however, using a single constant collapse operator σ_- fails and is insensitive to the drive. This latter approach could be easily improved by defining more correct collapse operators, but we included the most simple choice of collapse operator here in order to demonstrate a potential pitfall of using phenomenological Lindblad equations, and to demonstrate the utility of QuTiP's more advanced solvers in validating or invalidating one's choice of approximate master equation.

3.2.5. `mesolve` part 3: JAX and GPU acceleration with `DiffraX`

The use of graphical processing units (GPUs) to accelerate the solution of numerical tasks has become ubiquitous in the last 10 years, with applications ranging from mining cryptocurrency to training large language models. Historically, using GPUs for science required custom algorithm implementations in Cuda or OpenCL and was labour intensive, limiting their use to only the most demanding of applications. However, in recent years, the use of GPUs also for scientific tasks has greatly increased due to the availability of off-the-shelf packages like TensorFlow, CuPy, or JAX. With the flexible data layer in QuTiP, these packages can be slotted in and used with minimal overhead for the user.

After some initial experimentation with CuPy [28] and TensorFlow [29], development has focused on a QuTiP-JAX [26] data layer because of its powerful auto-differentiation capabilities and its mass adoption by the broader machine learning community. We will discuss the auto-differentiation aspects later, and in this section just provide an example of how QuTiP-JAX can be used for solving open quantum system dynamics on a GPU.

Before continuing, it is important to consider in which circumstances GPUs would actually accelerate such a task. GPUs tend to shine when evaluating many small matrix-vector problems in parallel, so one expects it to help substantially when integrating the behavior of a small system for many different parameters, such as for qiskit-dynamics [14], or repeated multiplication of small matrices to the subspace of a large quantum state, as demonstrated in various quantum circuit simulators such as `yao.jl` [33]. However, when solving a single ODE of a large system, whose integration involves repeated matrix-vector products with very large matrices, the potential advantage is less clear. The sequential nature of an ODE integrator makes it hard to parallelize. Although we expect that, for large matrices, it could be beneficial to parallelize individual matrix-vector products themselves at the level of column-row products, this optimization would require substantial modification of lower-level libraries. Here, we will show, with a practical example, that there is a cross-over in the system size where it becomes advantageous to use a GPU in QuTiP nevertheless.

In order to use QuTiP-JAX, one must install it as well as the JAX package in addition to the core QuTiP package. It can then be imported as follows:

```
import jax
import jax.numpy as jnp
import qutip_jax # noqa: F401
```

Using `import jax.numpy as jnp` is convenient as JAX mirrors all of the functionality of NumPy with equivalent functions that can be used with JAX (for the purposes of auto-differentiation, for example). The import of the `qutip_jax` module enables, as a side effect, the data layer formats '`jax`' and '`jaxdia`'. The former is a dense format, while the latter is a custom sparse diagonal format. An experimental CSR format is available within JAX, but currently not yet supported by QuTiP.

```
print(qt.qeye(3, dtype='jax').dtype.__name__)
| JaxArray

print(qt.qeye(3, dtype='jaxdia').dtype.__name__)
| JaxDia
```

To use the JAX data-layer within the master equation solver we need to use the `diffraX` ODE integrator, which we select by passing the option "`method`":`"diffraX"` to the solver. A short-cut to this can be done

with the new `qutip_jax.set_as_default()` functionality, which enables us to switch all QuTiP objects' data types to JAX and change the solver method to `diffra`x in a single step:

```
# Use JAX as the backend
qutip_jax.set_as_default()
```

The following code can be used to revert this change:

```
qutip_jax.set_as_default(revert=True)
```

Apart from this, manipulating QuTiP-JAX objects and solving problems with the master equation solver proceeds as with any other data format. To give a concrete example, and demonstrate an advantage of using GPUs, we consider a 1D Ising spin chain model:

$$H = \sum_{i=1}^N g_0 \sigma_z^{(n)} - \sum_{n=1}^{N-1} J_0 \sigma_x^{(n)} \sigma_x^{(n+1)}. \quad (12)$$

Here, N is the number of spins, g_0 the level splitting and J_0 a coupling constant. The end of the spin chain is in contact with an environment, modeled as a Lindblad dissipator with the collapse operator $\sigma_x^{(N-1)}$ and the coupling rate γ . This model can be defined and solved in QuTiP as follows:

```
def Ising_solve(N, g0, J0, gamma, tlist, options, data_type='CSR'):
    # N : number of spins
    # g0 : splitting
    # J0 : couplings
    with qt.CoreOptions(default_dtype=data_type):

        # Setup operators for individual qubits
        sx_list, sy_list, sz_list = [], [], []
        for i in range(N):
            op_list = [qt.qeye(2)] * N
            op_list[i] = qt.sigmax()
            sx_list.append(qt.tensor(op_list))
            op_list[i] = qt.sigmay()
            sy_list.append(qt.tensor(op_list))
            op_list[i] = qt.sigmaz()
            sz_list.append(qt.tensor(op_list))

        # Hamiltonian - Energy splitting terms
        H = 0.
        for i in range(N):
            H += g0 * sz_list[i]

        # Interaction terms
        for n in range(N - 1):
            H += -J0 * sx_list[n] * sx_list[n + 1]

        # Collapse operator acting locally on single spin
        c_ops = [gamma * sx_list[N-1]]

        # Initial state
        state_list = [qt.basis(2, 1)] * (N-1)
        state_list.append(qt.basis(2, 0))
        psi0 = qt.tensor(state_list)

        result = qt.mesolve(H, psi0, tlist, c_ops, e_ops=sz_list, options=options)
    return result, result.expect[-1]
```

Note that we used the `qt.CoreOptions(default_dtype=data_type)` context manager so that objects created with most internal functions use the same data format (overriding their default behavior). Note however that this currently does not override the behavior of every function which can create a Qobj in QuTiP (e.g., PIQs and HEOMSolver do not support this option).

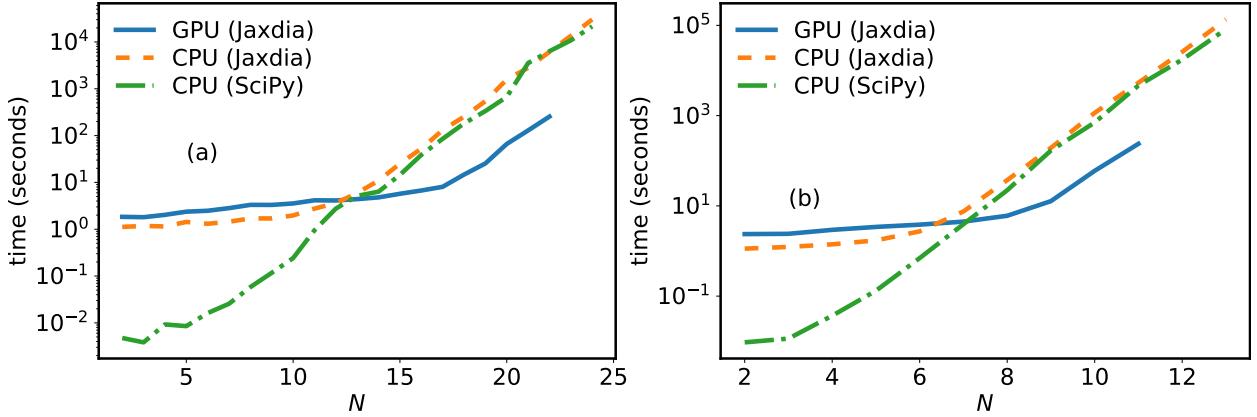


Figure 4: Benchmark of the time required to solve the dynamics of an Ising spin chain as a function of the number of spins N using the JAX data layer. The simulation was performed on an NVIDIA A100 GPU with 80 GB of RAM. The CPU used for comparison was an AMD EPYC 7713 (64 cores). Panel (a) shows a noiseless example for up to 22 spins (24 on CPU) using `sesolve`. Panel (b) shows the same problem in the presence of noise for up to 11 spins (12 on CPU) using `mesolve`. The Schrödinger equation solver is able to integrate more spins because of the memory cost of the superoperator constructed by `mesolve`. In both cases, we see a crossover at a certain system size where the GPU solver becomes more performant, but the GPU memory limit is reached shortly after. For the examples run on CPU we also differentiate between the default SciPy based solver and using JAX in CPU mode. In all JAX examples we used the `jaxdia` data-layer. The dense Jax format runs out of memory much more quickly (not shown).

With this definition, we can now compare solving the same problem with the default (sparse) data format on the CPU and with JAX on the GPU. For example, to run the solver on the GPU one can do the following:

```
from diffraax import PIDController, Tsit5

with jax.default_device(jax.devices("gpu")[0]):
    # System parameters
    N = 4 #number of spins
    g0 = 1 #energy splitting of spins
    J0 = 1.4 #spin-spin coupling
    gamma = 0.1 #dissipation rate

    # Simulation parameters
    tlist = jnp.linspace(0, 5, 100) #times steps
    options = {
        "normalize_output": False,
        "store_states": True,
        "method": "diffraax",
        "stepsize_controller": PIDController(rtol=qt.settings.core['rtol'],
                                             atol=qt.settings.core['atol']),
        "solver": Tsit5()
    }

    result_ising, sz1 = Ising_solve(N, g0, J0, gamma, tlist, options, data_type='jaxdia')
```

By comparing the performance of simulating the dynamics of the Ising spin chain with the standard QuTiP CPU-bound method (`adams`) and the equivalent `diffraax` method on a GPU, we observe a threshold in system size where the GPU outperforms the CPU calculation by up to two orders of magnitude. Simulating systems of this size requires the use of the `jaxdia` format to avoid memory limitations of the dense `jax` format. However, even with the `jaxdia` format and using a state-of-the-art graphics card with 80 gigabytes of RAM, the memory limit is reached already at 11 spins. For the same model without dissipation, which can be solved with the Schrödinger equation solver, 22 spins are possible. To go beyond these limits and harness large-scale high-performance computing requires us to distribute the ODE integration across multiple GPU nodes. This is a challenging task that we plan to explore in future evolutions of QuTiP-JAX.

3.2.6. `steadystate`: A steady-state solver for master equations

When solving the dynamics of a closed Hamiltonian system with an initial condition that is not an eigenstate of the system, one will typically observe persistent oscillations that never decay. However, when considering an open system described with a master equation $\dot{\rho}(t) = \mathcal{L}\rho(t)$, where \mathcal{L} is the Liouvillian superoperator, the dissipation terms will gradually suppress coherences (in certain bases) and push the system in the long time limit into a steady state. The steady state satisfies the equation

$$\frac{d\rho(t \rightarrow \infty)}{dt} = \mathcal{L}\rho(t \rightarrow \infty) = 0, \quad (13)$$

i.e., it does no longer change with time. The steady state is usually unique, but there can be multiple possible steady states under certain conditions (dark states which are not affected by the dissipation, or non-connected subspaces). We will here focus on the case where a unique steady state exists and mention some caveats about the degenerate case at the end of this section. One should also note that the discussion above applies only to time-independent systems. Systems with time-dependent driving typically have persistent oscillations around some fixed point in the long-time regime, which can be found either by explicitly solving the dynamics of the master equation or by using QuTiP's `steadystate_floquet()` function. This function will not be discussed further here, and we refer to the online documentation for more information.

For time-independent problems described by Lindblad or other master equations, QuTiP provides the `steadystate()` function which supports various methods to solve the steady-state condition (13). Like `mesolve()`, `steadystate()` takes as inputs either a Hamiltonian and collapse operators, or a Liouvillian superoperator constructed manually by the user.

The default solution method is called `direct` and directly solves (13) as a linear equation, using normalization as an extra condition to obtain a uniquely solvable set of linear equations. Using the function's `solver` parameter, the user can select from a variety of linear equation solvers. For dense Liouvillians, there are the solvers `solve` and `lstsq` based on NumPy functions, and for sparse problems, many SciPy sparse linear equation solvers can be used: `spsolve`, `gmres`, `lgmres`, and `bicgstab`. Some of these use iterative algorithms that become useful when the memory cost of finding the full exact solution is too high. Finally, there is an `mkl_spsolve` sparse solver which uses Intel's Math Kernel Library. This can offer substantial performance benefits, particularly on Intel CPUs, but requires that the necessary libraries are installed on the system.

The alternative method `power` also solves the linear equation (13), but it uses an additional iterative inverse power step [34] which starts by assuming $\rho_{ss}^{(0)} = 1$ and then solves $\mathcal{L}\rho_{ss}^{(n)} = \rho_{ss}^{n-1}$ until $\mathcal{L}\rho_{ss}^{(n)} < \varepsilon_{tol}$ for some small tolerance ε_{tol} close to zero. With this method, the same solvers can be used as with the `direct` method discussed previously. Two further methods are available: `eigenvalue`, which finds the zero eigenvector of \mathcal{L} iteratively, and `svd`, which use a dense singular-value decomposition of the Liouvillian.

Generally speaking, the direct method with either an exact linear equation solver or an iterative one (for large system sizes) are the most commonly employed methods. The optimal choice for a given situation tends to be problem specific. For more details, a comprehensive analysis and benchmark of some of these methods for common quantum problems is provided in [34], alongside a deeper explanation of the origin of the different approaches. Finally, in the situation where multiple steady states exist, the different solvers can often produce very different results; some may fail, others may produce linear combinations of possibilities, and so on. So far, QuTiP does not support an automated approach to dealing with this issue; therefore, the onus is on the user to understand the connectivity and properties of their model, or to check their results using long-time propagation of the dynamics with `mesolve()`.

3.2.7. `mcsolve`: A Monte Carlo solver for quantum trajectories

The Lindblad master equation described in (3) describes the ensemble-averaged dynamics of a quantum system in contact with an environment. In other words, it describes the expected results averaged over many repetitions of the same “experiment”. Interestingly, this equation can be unravelled in terms of trajectories of possible outcomes of single experiments. Each trajectory is assigned a classical probability, which is related to the likelihood of the environmental behavior that is necessary to produce that trajectory.

The trajectory unravelling thus provides a theoretical way of describing the fluctuation of results between multiple realizations of the experiment, and it sheds light on the physical interpretation of the Lindblad master equation. Further, it is also interesting from the point of view of numerical efficiency, because we can sometimes unravel the master equation in such a way that the density operator is written as an ensemble average of pure states which follow a non-linear stochastic Schrödinger equation [35]. Therefore, it is not necessary to store the full super-operator form of the Lindbladian generator in memory, and one can work with regular states instead of density operators. For large systems, this approach can save both computing time and memory, in particular if many trajectories can be simulated in parallel.

The Monte Carlo solver in QuTiP is based on the Monte Carlo wave function (MCWF) technique, which is an unravelling of the Lindblad equation in terms of pure states following quantum jump trajectories. These trajectories consist of periods of coherent, deterministic time evolution described by the effective non-Hermitian Hamiltonian

$$H_{\text{eff}} = H - \frac{i}{2} \sum_n C_n^\dagger C_n \quad (14)$$

interspersed with discrete quantum jumps. Here, H and C_n are the Hamiltonian and the collapse operators from the Lindblad equation (3). The non-Hermitian evolution does not conserve the norm of the wave function, but causes it to continuously decrease towards zero. The norm represents the survival probability, that is, the probability for the system to still evolve along that trajectory without having undergone a quantum jump (some new insights on this unraveling can be found in [36, 37]). The decrease of the norm of the state within a time step hence represents the probability of the system undergoing a quantum jump in that time step.

At a certain time, randomly chosen by considering the survival probability, a quantum jump occurs and the system undergoes a transition described by one of the collapse operators C_n . The probability that the jump at time t is described by the n -th collapse operator is proportional to $\langle \psi(t) | C_n^\dagger C_n | \psi(t) \rangle$, where $|\psi(t)\rangle$ is the trajectory state before that jump. This process can be interpreted as the environment “measuring” the system, such that the system is projected into the “target state” associated with the jump, and then re-normalized:

$$|\psi(t)\rangle \rightarrow (\langle \psi(t) | C_n^\dagger C_n | \psi(t) \rangle)^{-1/2} C_n |\psi(t)\rangle. \quad (15)$$

The record of these jumps can in principle be obtained by an experimenter monitoring the environment and marking when they see the transition described by the collapse operator. For example, this transition might be the system emitting a photon into the environment, or charge moving between discrete states.

Basic example. — The actual implementation of this method within QuTiP follows the algorithm outlined in [35, 38, 39] and is described in the documentation. It is invoked with the function `mcsolve()`, which can be used exactly like `mesolve()` and returns an average over 500 trajectories by default. The number of trajectories can be adjusted using the parameter `ntraj`.

```
result_mc = qt.mcsolve(H, psi0, tlist, c_ops=[sz1, sz2], ntraj=ntraj,
                      options={"map": "parallel", "keep_runs_results": True})
```

We used the option “`map`”:“`parallel`” to automatically take advantage of multiple CPU cores to simulate different trajectories in parallel, and the option “`keep_runs_results`”:`True`, which stores the results of the individual trajectories in addition to the ensemble average. When the latter option is enabled, the `runs_expect` property of the result object returns, for each expectation operator, a list of trajectories. Each of these trajectories is given as a time series for the expectation value of that operator, with the entries of the time series corresponding to the times in the `tlist` given to the solver. The average of the expectation values over all generated trajectories are returned in the `average_expect` property, and their standard deviation is in `std_expect`. These two properties are available also if the `keep_runs_results` option is not enabled.

Furthermore, if the option “`store_states`”:`True` is used, the `runs_states` property returns the states for all trajectories (if requested), while `average_states` contains their average. Given a sufficiently large number of trajectories, the states in the `average_states` property of the Monte Carlo result will approximate the states returned by the regular master equation solver.

Changes in v5. — Finally, in QuTiP v5, the result object gained a new `photocurrent` property, which saves the expectation value of the photocurrent. This property replaces the `photocurrent_sesolve()` stochastic methods from earlier QuTiP releases.

The finite sampling used by the Monte Carlo solver means that sometimes care must be taken to obtain a result close to the true ensemble average. The solver provides several methods to decide at which point to stop generating more trajectory samples. As mentioned previously, the option `ntraj` controls the maximum number of trajectories; calling `mcsolve` with `ntraj=1000` and no further options will thus use 1000 trajectories instead of the default 500. In addition, a maximum computation time can be specified; for example, using the option `timeout=60` will stop the solver after 60 seconds even if the requested number of trajectories has not been reached yet. Finally, the `target_tol` option interrupts the computation when the statistical error of the results reaches the given target value. For more fine-tuning, target values can be given for both the absolute and the relative error, and it is also possible to use different target values for the different expectation operators. How to specify these tolerances is explained in detail in the documentation.

In QuTiP v5, a class interface has been added to the Monte Carlo solver in analogy to the class interfaces for the other solvers discussed above. Another addition in version 5 is an improved sampling algorithm, which can be enabled in the options with "improved_sampling":True. This algorithm is particularly useful when the dissipation rates are very small. In this case, with the default algorithm, many of the simulated trajectories may end up containing no jumps and only deterministic evolution; all of these trajectories are thus identical. To avoid these redundant computations, the improved sampling algorithm only runs the no-jump trajectory once and includes it in the final statistics with an appropriate weighting.

Another improvement in the Monte Carlo solver in QuTiP v5 is the support for mixed initial states. Given a mixed initial state, the solver runs trajectories for each pure initial state in the mixture, and correctly weights the results in the final averaging. Users have the option to manually control the number of trajectories used for each initial state, or to select this number automatically based on the requested total number of trajectories. When combined with the improved sampling option, one no-jump trajectory will be generated for each initial state. Future improvements to the solver may incorporate recent advances in optimizing the Monte Carlo method based around the waiting-time distribution [40].

3.2.8. `nm_mcsolve`: A Monte Carlo solver for non-Markovian baths

If the time evolution of an open quantum system exhibits non-Markovian effects, the dynamics of its reduced density matrix cannot be described by a Lindblad master equation. However, by applying the time-convolutionless (TCL) projection operator technique, it is still possible to write the dynamics in the time-local form [16, 41–43]

$$\begin{aligned} \dot{\rho}(t) &= -\frac{i}{\hbar}[H(t), \rho(t)] + \sum_n \gamma_n(t) \mathcal{D}_n[\rho(t)] \quad \text{with} \\ \mathcal{D}_n[\rho(t)] &= A_n \rho(t) A_n^\dagger - \frac{1}{2} [A_n^\dagger A_n \rho(t) + \rho(t) A_n^\dagger A_n]. \end{aligned} \quad (16)$$

Here, $H(t)$ is a system Hamiltonian and A_n are jump operators but, in contrast to a Lindblad equation, the coupling rates $\gamma_n(t)$ may be negative at some or all times.

In the Monte Carlo wave function (MCWF) method implemented in QuTiP's `mcsolve`, quantum jumps occur with probabilities proportional to the coupling rates. In the present case, these probabilities would be negative; thus, the MCWF cannot be immediately applied to master equations like (16). This problem can be navigated by mapping the dynamics to an equivalent Lindblad master equation and applying the MCWF method to that Lindblad equation. For example, it is known in general that time-local quantum master equations can always be mapped to Lindblad equations on the double Hilbert space [44–46].

The non-Markovian Monte Carlo solver `nm_mcsolve()`, which was added to QuTiP in version 5, follows a similar approach. By introducing a trajectory weighting called the “influence martingale”, a master equation of the type (16) can be mapped to a Lindblad equation on the same Hilbert space, as shown in Refs. [43, 47, 48]. The formalism requires that the jump operators satisfy a completeness relation of the form $\sum_n A_n^\dagger A_n = \alpha \mathbb{1}$ for some scalar $\alpha > 0$. The function `nm_mcsolve()` automatically ensures that this

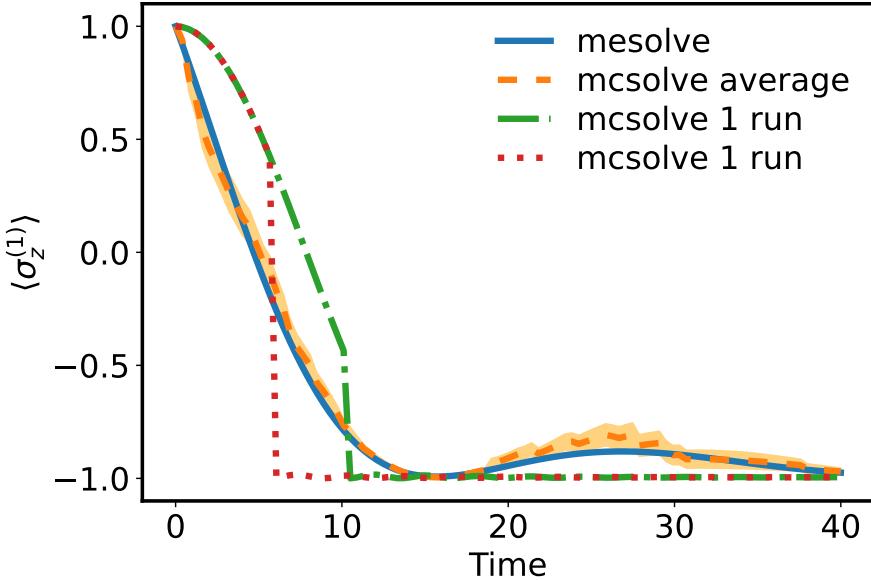


Figure 5: Here we show the same example as Fig. 2 evaluated with the Monte-Carlo solver `mcsolve`. The average behavior is compared to the result from `mesolve`, alongside two example trajectories showing discrete quantum jumps. The yellow shaded region indicates the convergence error based on the standard deviation of the trajectories and the number of trajectories, $\sigma_{\text{err}} = \sigma/\sqrt{N_{\text{traj}}}$. Here, we use a smaller number of trajectories $N_{\text{traj}} = 100$ than default, to amplify the error and make it more visible in this plot.

relation is satisfied by adding, if necessary, an additional jump operator with zero coupling rate. It then calculates the shift function

$$s(t) = 2 |\min\{0, \gamma_1(t), \gamma_2(t), \dots\}| , \quad (17)$$

which ensures that the shifted rates $\Gamma_n(t) = \gamma_n(t) + s(t)$ are non-negative. Finally, it uses the regular MCWF method to generate trajectories $|\psi(t)\rangle$ for the completely positive Lindblad equation

$$\dot{\rho}'(t) = -\frac{i}{\hbar} [H(t), \rho'(t)] + \sum_n \Gamma_n(t) \mathcal{D}_n[\rho'(t)] , \quad (18)$$

such that $\rho'(t) = \mathbb{E}\{|\psi(t)\rangle\langle\psi(t)|\}$, where \mathbb{E} denotes averaging over the trajectory ensemble. The original state can then be reconstructed through the average $\rho(t) = \mathbb{E}\{\mu(t)|\psi(t)\rangle\langle\psi(t)|\}$, where

$$\mu(t) = \exp\left[\alpha \int_0^t s(\tau) d\tau\right] \prod_k \frac{\gamma_{n_k}(t_k)}{\Gamma_{n_k}(t_k)} \quad (19)$$

is the influence martingale. The product runs over all jumps on the trajectory with jump channels n_k and jump times $t_k < t$.

We note that the technique described above, and the non-Markovian Monte Carlo solver, are not limited to completely positive dynamics and apply to any master equation of the form (16). Examples for non-positive dynamics described in this form can be found, for example, in the study of Redfield equations [49] or of unitary evolution subject to classical noise [50, 51].

For an illustrating example, we consider the damped Jaynes-Cummings model, which describes a two-level atom coupled to a damped cavity mode. The cavity mode can be mathematically eliminated, leaving us with the two-level atom coupled to an effective environment with the power spectrum [16]

$$S(\omega) = \frac{\lambda\Gamma^2}{(\omega_0 - \Delta - \omega)^2 + \Gamma^2} . \quad (20)$$

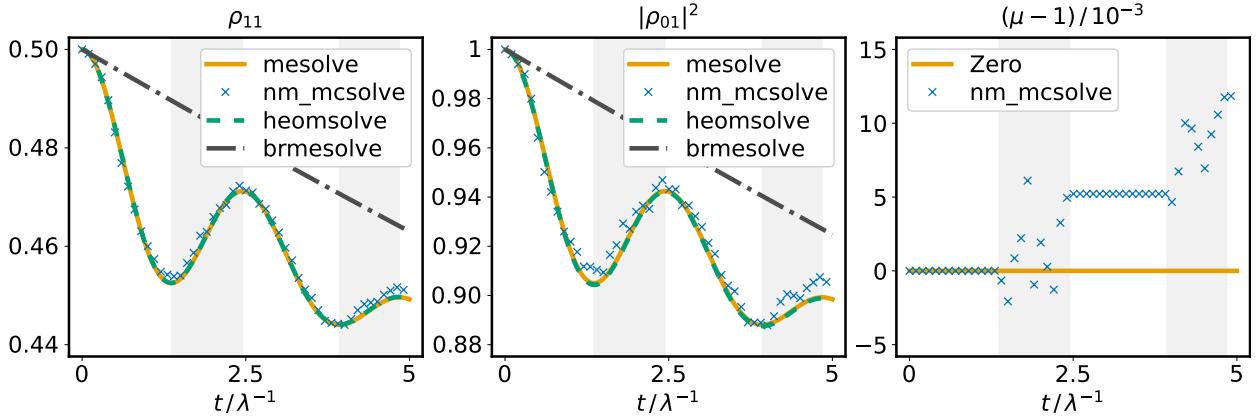


Figure 6: Damped Jaynes-Cummings Model. The figure shows results from the simulations described in Sec. 3.2.8, obtained with four different solvers available in QuTiP, including the non-Markovian Monte Carlo solver. In all panels, the gray background indicates times where $\gamma(t)$ is negative. We used $\Gamma = 0.3\lambda$, $\Delta = 8\Gamma$, and $\omega_0 = 100\lambda + \Delta$.

Here, λ is the atom-cavity coupling strength, ω_0 the atom transition frequency, Δ the cavity detuning and Γ the spectral width. Under the rotating wave approximation and at zero temperature, the dynamics of the two-level atom can be shown to follow the exact master equation [16]

$$\dot{\rho}(t) = \frac{A(t)}{2i\hbar} [\sigma_+ \sigma_-, \rho(t)] + \gamma(t) \mathcal{D}_-[\rho(t)], \quad (21)$$

where \mathcal{D}_- is the dissipator for the Lindblad operator σ_- . Further, σ_\pm are the ladder operators for the atom, $\rho(t)$ is the atom state in the interaction picture, and $\gamma(t)$ and $A(t)$ are the real and imaginary parts of the expression

$$\gamma(t) + iA(t) = \frac{2\lambda\Gamma \sinh(\delta t/2)}{\delta \cosh(\delta t/2) + (\Gamma - i\Delta) \sinh(\delta t/2)}, \quad (22)$$

with

$$\delta = [(\Gamma - i\Delta)^2 - 2\lambda\Gamma]^{1/2}. \quad (23)$$

Depending on the choice of system parameters, the coupling rate $\gamma(t)$ may become negative at some times.

We now apply a variety of QuTiP's solvers, including `nm_mcsolve()`, to this example. We first define the system Hamiltonian and initial state,

```
H = qt.sigmap() * qt.sigmam() / 2 #Hamiltonian
initial_state = (qt.basis(2, 0) + qt.basis(2, 1)).unit() #initial state
tlist = np.linspace(0, 5, 500) #time steps
```

and the functions $\gamma(t)$ and $A(t)$ as Python functions `gamma` and `A`. Despite $\gamma(t)$ being negative at some times, the exact master equation (21) can be integrated using `mesolve()`. However, the Liouvillian superoperator must be constructed by hand, since, by definition, dissipation terms added through the `c_ops` parameters cannot have negative rates:

```
unitary_gen = qt.liouvillian(H) #Hamiltonian term
dissipator = qt.lindblad_dissipator(qt.sigmam()) #Lindblad term (no rate included)
me_solution = qt.mesolve([[unitary_gen, A], [dissipator, gamma]], initial_state, tlist)
```

We can now easily run the non-Markovian Monte Carlo simulation. Instead of a list of collapse operators `c_ops`, this solver takes a list of pairs of jump operators and corresponding rates, which makes negative rates possible. Otherwise, it accepts mostly the same parameters as `mcsolve()`:

```

mc_solution = qt.nm_mcsolve([[H, A]], initial_state, tlist,
                           ops_and_rates=[(qt.sigmap(), gamma)],
                           ntraj=1_000, options={'map': 'parallel'})

```

The solvers used so far were based on the exact master equation (21). Additionally, we consider methods that will be discussed below, that apply directly to a spin-boson model with the given power spectrum (20) and corresponding free reservoir auto-correlation function

$$C(t) = \frac{\lambda\Gamma}{2} \exp[-i(\omega_0 - \Delta)t - \lambda|t|]. \quad (24)$$

We define the system Hamiltonian and system coupling operator $Q = \sigma_+ + \sigma_-$ in the Schrödinger picture,

```

H = omega_0 * qt.sigmap() * qt.sigmap() #Hamiltonian
Q = qt.sigmap() + qt.sigmap() #system operator which couples to the bath

```

where ω_0 was chosen much larger than Δ to ensure validity of the rotating wave approximation. The HEOM solver can be applied after decomposing the correlation function (24) into its real and imaginary parts:

```

ck_real = [gamma0 * lamb / 4] * 2 #Real term prefactors
vk_real = [lamb - 1j * omega_c, lamb + 1j * omega_c] #Real term exponents
ck_imag = np.array([1j, -1j]) * gamma0 * lamb / 4 #Imag term prefactors
vk_imag = vk_real #Imag term exponents

heom_bath = qt.heom.BosonicBath(Q, ck_real, vk_real, ck_imag, vk_imag) #create bath
heom_solution = qt.heom.heomsolve(H, heom_bath, 10, qt.ket2dm(initial_state), tlist)

```

Note that the parameter `omega_c` appearing here is $\omega_c = \omega_0 - \Delta$.

Finally, we compare with the Bloch-Redfield solver, which takes as its input the power spectrum (20):

```

def power_spectrum(w):
    return gamma0 * lamb**2 / ((omega_c - w)**2 + lamb**2)

br_solution = qt.brmesolve(H, initial_state, tlist, a_ops=[(qt.sigmax(), power_spectrum)])

```

For comparison, the results obtained with the HEOM solver and the Bloch-Redfield solver must be transformed into the interaction picture:

```

Us = [(-1j * H * t).expm() for t in tlist] #Transformation operator
heom_states = [U * state * U.dag() for (U, state) in zip(Us, heom_solution.states)]
br_states = [U * state * U.dag() for (U, state) in zip(Us, br_solution.states)]

```

The results of these simulations are shown in Fig. 6. Using only 1000 trajectories, the MCWF simulation reproduces the exact solutions obtained with `mesolve()` and `heomsolve()` well. The Bloch-Redfield equation produces a very different picture at the short time-scale considered here, showing that we are deep in the non-Markovian regime. The small deviations between `mesolve()` and `heomsolve()` stem from the rotating wave approximation.

Figure 6 shows that whenever $\gamma(t)$ is negative, coherence is restored in the atom state. It also shows the average value of the influence martingale (19), which is stored in the `mc_solution.trace` field. The average influence martingale is an estimator for $\text{tr } \rho(t) = \mathbb{E}\{\mu(t)\}$. We see that it is constant when $\gamma(t)$ is positive but fluctuates otherwise. Its deviation from the exact value $\text{tr } \rho(t) = 1$ can be used as an indicator for how well the Monte Carlo simulation has converged.

3.2.9. `brmesolve`: Bloch-Redfield master equation solver

Earlier, in Section 3.2.3, we discussed a master equation in Lindblad form, which described transitions between system eigenstates with rates proportional to the power spectrum of an environment. Such master equations can be derived microscopically from a system-bath model, where the power spectrum encodes the frequency-dependent coupling strength and the temperature of the environment. The approximations used to derive the master equation in this form are called the Born-Markov-Secular approximations. The Born

and Markov approximations capture the weak-coupling and memoryless nature of the environment, and the secular approximation assumes that certain high frequency terms can be discarded.

A master equation derived through these approximations is called a Bloch-Redfield master equation, and it can be conveniently constructed in QuTiP using the `brmesolve()` solver. A useful feature of this solver is that the secular approximation can be relaxed to a specified degree, leading to a so-called non-secular and non-Lindblad equation of motion. Despite not having strict Lindblad form, this generalized master equation is still capable of describing the reduced state of the system in certain parameter regimes. Being able to soften the secular approximation can be important if the bath has a certain structure, or if the dissipation rates or the temperature of the bath are large.

A full derivation can be found in the literature [16], and a short discussion of its most important steps is included in the QuTiP documentation. The final equation implemented in QuTiP can be written as

$$\frac{d}{dt}\rho_{ab}(t) = -i\omega_{ab}\rho_{ab}(t) + \sum_{c,d}^{\text{sec}} \frac{1}{2}R_{abcd}\rho_{cd}(t). \quad (25)$$

The indices a, b refer to matrix-elements of operators in the eigenbasis of the system Hamiltonian H_{sys} with eigenenergies ω_m and $\omega_{ab} = \omega_a - \omega_b$. The system is coupled to one or more baths labelled by the index α through operators A^α , with matrix elements A_{ab}^α in the Hamiltonian eigenbasis. The bath is fully described by its power spectrum $S_\alpha(\omega)$, which we defined in (7).

Given these definitions, the Bloch-Redfield tensor can be written as

$$R_{abcd} = - \sum_{\alpha,\beta} \left\{ \delta_{bd} \sum_n A_{an}^\alpha A_{nc}^\beta S_{\alpha\beta}(\omega_{cn}) - A_{ac}^\beta A_{db}^\alpha S_{\alpha\beta}(\omega_{ca}) \right. \\ \left. + \delta_{ac} \sum_n A_{dn}^\alpha A_{nb}^\beta S_{\alpha\beta}(\omega_{dn}) - A_{ac}^\beta A_{db}^\alpha S_{\alpha\beta}(\omega_{db}) \right\}, \quad (26)$$

The sum in (25) refers to the degree to which the secular approximation is applied in the derivation of (26). Making the secular approximation corresponds to removing certain terms in the sums over c and d (which arise from fast oscillating terms in the interaction picture).

In QuTiP, the Bloch-Redfield tensor can be calculated with the function `bloch_redfield_tensor()` and used directly in `mesolve()`, akin to the manual Lindblad construction we showed earlier. These two steps can be combined by using the `brmesolve()` solver interface directly. The secular approximation can be controlled using the `sec_cutoff` parameter, with `sec_cutoff=-1` implementing the complete non-secular tensor, and any positive float giving a partial secular approximation (neglecting terms oscillating with frequencies $|\omega_{ab} - \omega_{cd}|$ larger than this value). The default value of 0.1 will give rise to a secular equation of motion in most cases.

The other powerful and important feature of this solver is its support for time-dependent Hamiltonians, which we demonstrated earlier in Sec. 3.2.4. The assumption used here is that the environment always sees the system in its ‘instantaneous eigenbasis’ at any given time, so the above equation of motion applies, but with the eigenbasis used to construct the Redfield tensor constantly changing. This feature of the time-dependent Bloch-Redfield equation is very challenging numerically because the Hamiltonian must be diagonalized at every instance in time. The solver `brmesolve()` is optimized for this diagonalization, but it remains a challenging numerical problem.

An example of using `brmesolve()` is shown in Fig. 2 (see also Fig. 3 and Fig. 6), alongside the results from local and global Lindblad equations solved with `mesolve()`. To define a problem for the `brmesolve()` solver, we must provide a list of system operators that couple to an environment, and the power spectrum of that environment. The example in Fig. 2 is constructed with:

```

def power_spectrum(w):
    if w >= 0: #Zero-temperature, only has positive-frequency support
        return gam
    else:
        return 0

result_BR = qt.brmesolve(H, psi0, tlist, e_ops=[sz1, sz2],
                         a_ops=[[sx1, power_spectrum], [sx2, power_spectrum]])

```

The example for a driven problem, in Sec. 3.2.4, was defined there already, and demonstrated in Fig. 3.

Finally, we note that the definition in Eq. (26) allows for non-Hermitian coupling operators, as required by interacting Fermionic systems which might obey an interaction Hamiltonian in the form $H_I = \sum_\alpha A_\alpha^\dagger \otimes B_\alpha + A_\alpha \otimes B_\alpha^\dagger$. In QuTiP 5 this is now supported through the use of a `FermionicEnvironment` to describe the bath, or `brcrosssterm()` to manually build custom interactions. An explicit example of how to use the Bloch-Redfield solver with the environment class is described in 3.2.12.

3.2.10. Floquet methods: the Floquet basis

In previous sections we have seen different methods to solve time-dependent Hamiltonians, each of them suitable for different scenarios. Here, we discuss Hamiltonians with a periodic time-dependence. In such cases, a natural approach is using the Floquet theorem to tackle the problem, similar to how using the Bloch theorem (a particular application of the Floquet theorem) greatly simplifies problems with spatial periodicity.

Literature on Floquet theory [52] and its application to quantum systems is extensive [53–55]. Here, for completeness, we will only introduce the main results when applied to time-periodic Hamiltonians. Let H be a Hamiltonian periodic in time with period T , such that

$$H(t) = H(t + T). \quad (27)$$

The system state follows the time-dependent Schrödinger equation $i\hbar \frac{\partial}{\partial t} |\psi(t)\rangle = H(t) |\psi(t)\rangle$. The Floquet theorem states that there exist state solutions defined as

$$|\psi_\alpha(t)\rangle = \exp(-i\epsilon_\alpha t/\hbar) |\Phi_\alpha(t)\rangle, \quad (28)$$

where ϵ_α are the quasi-energies and $|\Phi_\alpha(t)\rangle = |\Phi_\alpha(t+T)\rangle$ the Floquet modes. Then, any solution $|\psi(t)\rangle$ of the time-dependent Schrödinger equation can be written as a linear combination of the Floquet states such that

$$|\psi(t)\rangle = \sum_\alpha c_\alpha |\psi_\alpha(t)\rangle, \quad (29)$$

where the constants c_α are determined by the initial conditions.

By inserting (28) into the Schrödinger equation, one can define the Floquet Hamiltonian,

$$H_F(t) \equiv H(t) - i\hbar \frac{\partial}{\partial t}, \quad (30)$$

which converts the time-dependent Schrödinger equation into a time-independent problem,

$$H_F(t) |\Phi_\alpha(t)\rangle = \epsilon_\alpha |\Phi_\alpha(t)\rangle. \quad (31)$$

Once we obtain the Floquet modes at $t \in [0, T]$, using Eq. (28) we immediately know $|\psi(t)\rangle$ at any large t .

We now turn our attention to the application of Floquet theory in QuTiP. We first analyse the computational performance of using the Floquet basis compared to the Schrödinger equation solver `sesolve()`. For this example, we use the Hamiltonian of a two-level system driven by a periodic function,

$$H = -\frac{\epsilon}{2}\sigma_z - \frac{\Delta}{2}\sigma_x + \frac{A}{2}\sin(\omega_d t)\sigma_x, \quad (32)$$

where ϵ is the energy-splitting, Δ the coupling strength and A the drive amplitude. As discussed earlier, this Hamiltonian can be defined in QuTiP as follows.

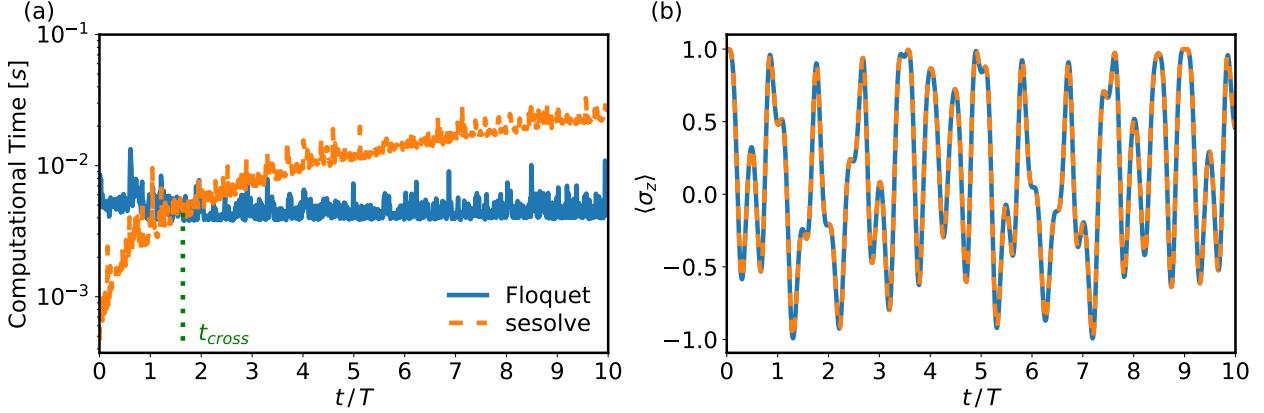


Figure 7: Two-level system driven periodically (Eq. (32)) with energy splitting $\epsilon = 2\pi$, coupling strength $\Delta = 0.2 \times 2\pi$, drive amplitude $A = 2.5 \times 2\pi$ and drive frequency $\omega_d = 2\pi$. (a) Time to evolve the state from the initial state until the given time, using the Floquet method (solid blue line) and `sesolve()` (orange dashed line). The dotted green line indicates the normalised time, t_{cross} , at which Floquet basis and `sesolve()` take the same computational time to evolve the state. For times beyond this, direct integration with `sesolve()` is less efficient than using the Floquet basis. Note that intermediate results such as the computed Floquet basis or the evolved state are not reused between data points. (b) Expected value of σ_z as a function of time, computed using the Floquet method and `sesolve()`.

```
H0 = -1 / 2 * (epsilon * qt.sigmax() + delta * qt.sigmax()) #Hamiltonian (static)
H1 = A / 2 * qt.sigmax() #Hamiltonian (driven part)
args = {'w': omega}
H = [H0, [H1, lambda t, w: np.sin(w * t)]] #Creates a QobjEvo Hamiltonian
```

Figure 7(a) shows the speed comparison between the Floquet approach and direct integration of the Schrödinger equation. One can see that the computational time needed using the Floquet method is, on average, independent of the time until which we want to evolve our state. Because we always need to compute at least one full period to use the Floquet basis, the overhead of the Floquet method is disadvantageous compared to `sesolve()` at short times. However, at large stroboscopic times it becomes more efficient to use the Floquet basis, since we take advantage of the system periodicity in that way. In QuTiP we could implement this using the `fsesolve()` solver, or manually as follows:

```
floquetbasis = qt.FloquetBasis(H, T, args)
# Decomposing initial state into Floquet modes
f_coeff = floquetbasis.to_floquet_basis(psi0)
# Obtain evolved state in the original basis
psi_t = floquetbasis.from_floquet_basis(f_coeff, t)
```

Figure 7(b) shows the agreement between the two methods at arbitrary stroboscopic times.

A good example to analyze the dimensional scaling of the Floquet implementation is to study a one-dimensional Ising spin chain under a periodic drive. The Hamiltonian of this system can be expressed as

$$H = g_0 \sum_{n=1}^N \sigma_z^{(n)} - J_0 \sum_{n=1}^{N-1} \sigma_x^{(n)} \sigma_x^{(n+1)} + A \sin(\omega_d t) \sum_{n=1}^N \sigma_x^{(n)}, \quad (33)$$

where g_0 is the level splitting, J_0 is the coupling constant between nearest-neighbour spins and A is the drive strength. In QuTiP, this Hamiltonian can be implemented similarly to the Ising Hamiltonian without drive used in Sec. 3.2.5, except that here we additionally include the driving term.

In Fig. 8, we compare the performance of `FloquetBasis()` and `sesolve()` when studying the Ising spin chain under a periodic drive, Eq. (33). Particularly, we are interested in comparing these two methods depending on the number of particles in the system N . Figure 8(b) shows, depending on the number of particles N , the crossover time t_{cross} at which the Floquet method becomes more performant than the

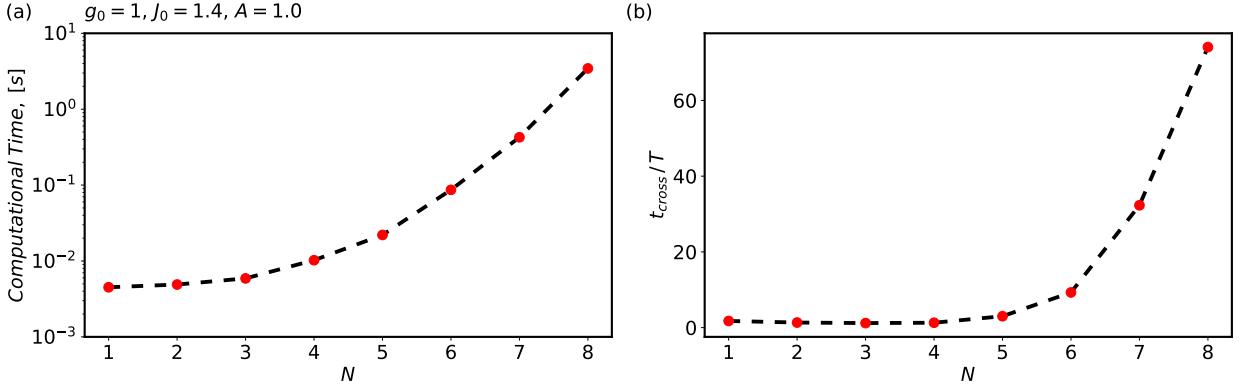


Figure 8: Numerical study of the performance of Floquet basis and `sesolve` depending on the dimension of the system N . (a) Computational time needed to reach the t_{cross} depending on N . (b) Normalised time t_{cross}/T at which Floquet basis and `sesolve` require similar computational time versus N . Discrete data points are shown as red dots, while dashed lines are included to facilitate visualization of the overall trend.

direct `sesolve()` method. Figure 8(a) shows the computational time required to evolve the system until that crossover time.

In addition to the Floquet basis transformation and the `fsesolve()` solver mentioned before, QuTiP includes the solver `fmesolve()` which is capable of analyzing time-periodic Hamiltonians which are affected by a dissipative bath, i.e., quasi-time-periodic Hamiltonians. We refer the reader to the QuTiP documentation for examples using this solver. When applying it, the user must proceed with caution since a generalized Floquet method can give unphysical results if the dissipation rate is too large compared to the pure eigenenergies of the periodic Hamiltonian. In a future work, we plan to discuss the intricacies of generalized Floquet methods and this solver in particular. Moreover, another solver `flimesolve()`, which also uses Floquet theory to approach open quantum systems, is currently being developed [56].

3.2.11. `somesolve`: Stochastic master equation solver

When modelling an open quantum system, classical stochastic noise can be used to simulate a large range of phenomena. For example, classical noise can be used as a random term in the Hamiltonian as a means to simulate a classical environment randomly changing some system property in each run of an experiment. Another example is the Monte Carlo solver discussed earlier, where classical randomness is used to simulate the random chance of a quantum jump occurring in a dissipative Lindblad process.

In the `somesolve()` solver that we will discuss in this section, noise appears because of a continuous measurement. The solver allows us to generate the trajectory evolution of a quantum system conditioned on a noisy measurement record. Historically, this type of solver was used by the quantum optics community to model homodyne (single quadrature) and heterodyne (two-quadrature) detection of light emitted from a cavity. However, the solver is quite general, and can in principle be also applied to other types of problems.

To demonstrate its use we will focus on the standard example of a stochastic master equation describing an optical cavity whose output is subject to homodyne detection. The cavity obeys the general stochastic master equation,

$$d\rho(t) = -i[H, \rho(t)] dt + \mathcal{D}[a]\rho(t) dt + \mathcal{H}[a]\rho dW(t) \quad (34)$$

with $\mathcal{D}[a]\rho = a\rho a^\dagger - \frac{1}{2}a^\dagger a\rho - \frac{1}{2}\rho a^\dagger a$ being the Lindblad dissipator and $H = \Delta a^\dagger a$ the Hamiltonian, which together capture the deterministic part of the system's evolution. The term $\mathcal{H}[a]\rho = a\rho + \rho a^\dagger - \text{tr}[\rho a^\dagger a]$ represents the stochastic part which captures the conditioning of a trajectory through continuous monitoring of the operator a . Here, $dW(t)$ is the increment of a Wiener process obeying $\mathbb{E}[dW] = 0$ and $\mathbb{E}[dW^2] = dt$.

This equation can be easily implemented in `somesolve()` with

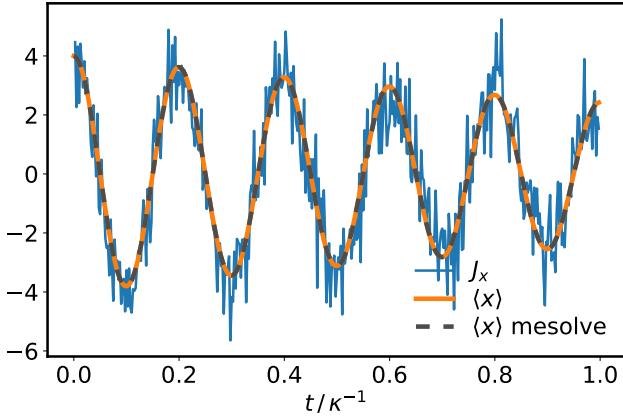


Figure 9: An example of solving the stochastic master equation for a dissipative cavity, with decay rate κ and Hamiltonian $H = \Delta a^\dagger a$, undergoing homodyne monitoring of the output field. Here we plot the averaged homodyne current $J_x = \langle x \rangle + dW/dt$, the average system behavior $\langle x \rangle$ for 50 trajectories, and the average result for $\langle x \rangle$ predicted by `mesolve` for the same model (without resolving conditioned trajectories). In this example the conditioned trajectories are only weakly affected by the noise, while the homodyne current remains noisy. Parameters used were $\Delta = 10\pi\kappa$, and initial condition of a coherent state with displacement $\alpha = 2$.

```
stoc_solution = qt.smesolve(
    H, rho_0, times, c_ops=[], sc_ops=[np.sqrt(kappa) * a], e_ops=[x],
    ntraj=num_traj, options={"dt": 0.00125, "store_measurement": True}
)
```

We have chosen here an initial coherent state, and collect expectation values of the operator $x = a + a^\dagger$. The parameter `sc_ops` indicates which operators are being monitored, and `stoc_solution.expect` returns a list of the averaged results, while `stoc_solution.measurement` returns a list of lists of individual measurement results of the operator $J_x(t) = \langle x \rangle + dW/dt$. The optional `c_ops` can be a standard list of collapse operators describing additional, unmonitored, baths. The results of this simulation are shown in Fig. 9. Interestingly, in this example the conditioned system trajectories are only weakly affected by the noise, making it a useful example for tests of this solver.

3.2.12. HEOMSolver: Hierarchical Equations of Motion

While the other solvers in QuTiP largely rely on either perturbative approximations and/or assumptions about the Markovianity of the environment a system is coupled to, the hierarchical equations of motion (HEOM) are a numerically exact method [22, 57, 58] to solve the dynamics of an open quantum system, under a minimal set of assumptions. It originated in the field of physical chemistry, where it was used to solve problems related to electronic energy transport in photosynthetic light-harvesting. It has recently found utility in a broad range of other fields, from quantum information to quantum electronics, and it is now used as a benchmark for developing other methods [59]. In QuTiP we provide a solver for both bosonic and fermionic environments, which in an upcoming release will support arbitrary spectral densities and correlation functions via a built-in fitting procedure.

The minimal assumptions that the HEOM method relies on are that the bath is Gaussian, initially in an equilibrium thermal state, and that the bath operator which couples to the system is linear. Using the Feynman-Vernon influence functional, one can show that, under these assumptions, the influence of the environment is fully characterized by its second order correlation function. For a bosonic environment, this correlation function can be expressed as (see [60] for an explanation of the fermionic bath case, and [22, 61] for applications)

$$C(t) = \int_0^\infty d\omega \frac{J(\omega)}{\pi} \left(\coth\left(\frac{\beta\omega}{2}\right) \cos(\omega t) - i \sin(\omega t) \right). \quad (35)$$

The derivation of the HEOM is also based on the Feynman-Vernon influence functional. The HEOM relies on the assumption that the correlation function can be written as a sum of decaying exponentials like

$$C(t) = C_R(t) + C_I(t) \quad \text{with} \quad (36a)$$

$$C_R(t) = \sum_{k=1}^{N_R} c_k^R \exp(-\gamma_k^R t) \quad \text{and} \quad (36b)$$

$$C_I(t) = \sum_{k=1}^{N_I} c_k^I \exp(-\gamma_k^I t) \quad (36c)$$

(though some variants of the technique have generalized this assumption [62]). By taking repeated derivatives of the influence functional, in conjunction with the assumption (36), one arrives at the coupled HEOM differential equations

$$\begin{aligned} \dot{\rho}^n(t) = & -i[H_S, \rho^n(t)] - \sum_{j=R,I} \sum_{k=1}^{N_j} n_{jk} \gamma_k^j \rho^n(t) - i \sum_{k=1}^{N_R} c_k^R n_{Rk} \{Q, \rho^{n_{Rk}}(t)\} \\ & + \sum_{k=1}^{N_I} c_k^I n_{Ik} \{Q, \rho^{n_{Ik}}(t)\} - i \sum_{j=R,I} \sum_{k=1}^{N_j} [Q, \rho^{n_{jk}^+}(t)], \end{aligned} \quad (37)$$

where $n = (n_{R1}, n_{R2}, \dots, n_{RN_R}, n_{I1}, n_{I2}, \dots, n_{IN_I})$ is a multi-index label of non-negative integers n_{jk} , and n_{jk}^- (n_{jk}^+) denotes the multi-index with the selected entry reduced (increased) by one. Further, Q is the generic system operator which couples to the bath. In practice, the *a priori* infinite hierarchy is truncated to $n_{jk} \leq N_c$, for a suitably chosen cutoff N_c . While the label $n = (0, 0, \dots, 0)$ corresponds to the system density matrix, operators $\rho^n(t)$ with $n \neq (0, 0, \dots, 0)$ are referred to as auxiliary density operators (ADOs), and encode correlations between system and bath.

To choose the value of the cutoff N_c , one typically starts with a small value which is then increased step by step until convergence is found. Heuristic arguments indicate that a lower bound is given by [63]

$$N_c \gtrsim \frac{\omega_S}{\min_{k,j} \operatorname{Re}[\gamma_k^j]}, \quad (38)$$

where ω_S is the largest system frequency and Re denotes the real part. In future versions of QuTiP, we are planning to implement more efficient cutoff mechanisms, where ADOs are kept or discarded according to an importance criterion [64–66].

The implementation of the HEOM in QuTiP is explained in greater detail in [22]. Following the release of QuTiP v5, it is currently being enhanced to be more compatible with the other solvers (`mesolve()` and `brmesolve()`) with a generic environment class that allows us to quickly compute the power spectrum. The logic of this new environment class, and its functionality, are described in Fig. (11) and Tables (7), (8) and (9).

Basic example. — To demonstrate how to use the HEOM solver in practice, let us consider the evolution of a qubit in a thermal bosonic environment with the Hamiltonian

$$H = \frac{\omega_0}{2} \sigma_z + \frac{\Delta}{2} \sigma_x + \sum_k w_k a_k^\dagger a_k + \sum_k g_k \sigma_z (a_k + a_k^\dagger). \quad (39)$$

In the continuum limit, one can describe the couplings through the spectral density

$$J(\omega) = \pi \sum_k |g_k|^2 \delta(\omega - \omega_k). \quad (40)$$

Typically, bosonic HEOM solvers use either the overdamped Drude-Lorentz spectral density or the underdamped Brownian motion spectral density. Let us for example consider the underdamped spectral density

$$J(\omega) = \frac{\lambda^2 \Gamma \omega}{(\omega_c^2 - \omega^2)^2 + \Gamma^2 \omega^2}. \quad (41)$$

We begin by initializing a bath, based upon the parameters in the spectral density:

```
env = UnderDampedEnvironment(lam=lam, gamma=gamma, T=T, w0=w0) #Create environment
bath = env.approximate("matsubara", Nk=5) #Approximate environment with Matsubara series
```

The `env` object contains the exact information about the bosonic bath, while the `bath` object is an approximated version of `env` that is specifically designed for the `HEOMSolver`, which requires as input an exponential decomposition of the bath correlation functions as described above. For this particular spectral density, Eq. 36 can be realized via the Matsubara or Padé decompositions. These decompositions rely on a truncation parameter N_k that introduces an approximation to the true bath dynamics by truncating the Matsubara or Padé decompositions into a finite number of exponentials. The QuTiP implementation makes it easy to see the impact of the approximation, as it is straightforward to compute the exact and approximated quantities that describe the bath (correlation function, spectral density and power spectrum)

```
env = UnderDampedEnvironment(lam=lam, gamma=gamma, T=T, w0=w0) #Exact environment
bath = env.approximate("matsubara", Nk=5) #Approximate environment
C = env.correlation_function(t) #Exact environment correlation functions
C2 = bath.correlation_function(t) #Approximate environment correlation functions
```

Similar notation is used for both the power spectrum and spectral density. The power spectrum can be used as quick gateway to compare with the other available solvers. For example, we can solve the HEOM equations by using the bath object (in a tuple, alongside which system operator `Q` it couples to), the system Hamiltonian and the `max_depth` parameter that specifies the cutoff N_c of the hierarchy equations, with the `HEOMSolver` as follows:

```
# -- HEOM --
solver = HEOMSolver(Hsys, (bath, Q), max_depth=9)
result_h = solver.run(rho0, t)
```

Simultaneously, we can compare this to a Bloch-Redfield solution also using the bath object properties,

```
# -- BLOCH-REDFIELD --
a_ops = [[Q, env]]
resultBR = brmesolve(Hsys, rho0, t, a_ops=a_ops, sec_cutoff=-1)
```

See Fig. 10 for a comparison of the results.

Ohmic bath with exponential cutoff. — The QuTiP implementation of the HEOM allows for the simulation of more general spectral densities: the user can create an arbitrary `BosonicEnvironment` from the spectral density, correlation function or power spectrum. As the HEOM requires a decaying exponential representation of the correlation functions, we can either fit the spectral density with one with a known analytical decomposition of its correlation function, or the correlation functions themselves, as explained in [22, 67]. Let us consider the simulation of a Ohmic bath, whose spectral density is given by

$$J(\omega) = \alpha \omega \exp(-|\omega|/\omega_c). \quad (42)$$

Because this spectral density is frequently used in the literature, a special class, `OhmicEnvironment`, has been added for convenience. As mentioned, in order to use the HEOM solver, we must first choose between fitting either the correlation function or the spectral density. The code snippet below shows how easy it is to set up simulations using these different approaches:

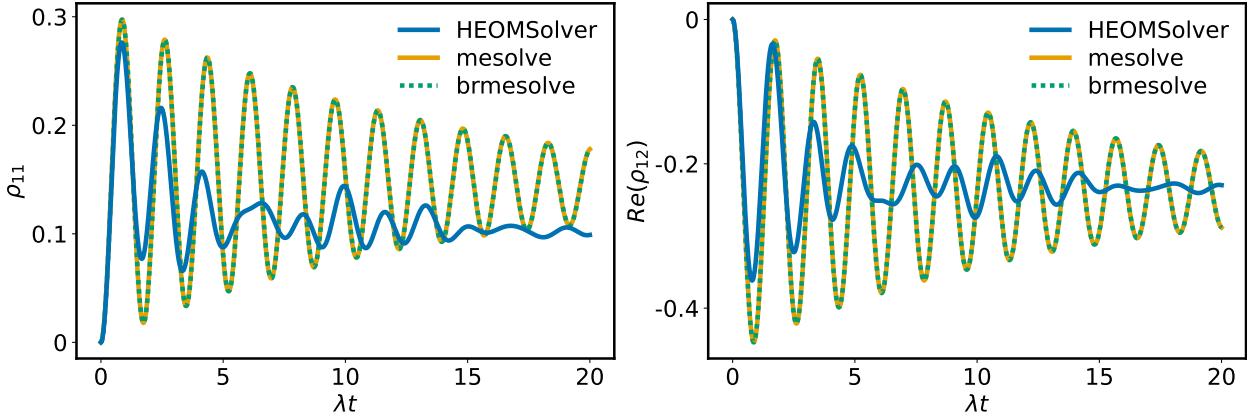


Figure 10: For the example of a standard spin-boson problem, we compare the output of the different master equation solvers available in QuTiP, namely, the Lindblad master equation solver (`mesolve()`), the Bloch-Redfield equation solver (`brmesolve()`) and the hierarchical equations of motion solver (`HEOMSolver`) for a spin coupled to a bosonic bath described by an underdamped Brownian motion spectral density. Left, (a), shows the evolution of the qubit population for the different approaches. Notice that both `brmesolve` and `mesolve` coincide, but they differ substantially from the HEOM result in this deeply non-Markovian regime. The right figure, (b), shows the dynamics of the coherence values. The parameters used to generate these figures are: $\lambda = 0.5 \Delta$, $\Gamma = 0.1 \Delta$, $T = 0.5 \Delta$, $N_k = 5$, $N_c = 6$ and $\omega_0 = \frac{3\Delta}{2}$.

```
# -- FITTING PREDEFINED OHMIC CLASS --
w = np.linspace(0, 100, 2000) #Fitting range in frequency-domain
env_fs, _ = oh.approximate("sd", wlist=w, Nk=3, Nmax=3) #Fit spectral density

t = np.linspace(0, 10, 1000) #Fitting range in time-domain
env_fc, _ = oh.approximate("cf", tlist=t, Ni_max=5, Nr_max=4,
                           target_rmse=None, maxfev=int(1e9)) #Fit correlation functions
```

The first output of the approximation by fitting is a `BosonicEnvironment` object, while the second output provides the fit information. This is particularly useful when dealing with non-standard baths, where one often needs more exponents to accurately describe the bath even when dealing with high temperatures, as the fit information helps one decide how many exponents to take into consideration.

The bath obtained from the fitting can be passed to the solver to quickly obtain its dynamics

```
# -- SOLVING DYNAMICS --
tlist = np.linspace(0, 10, 1000)
HEOM_corr_fit = HEOMSolver(Hsys, (env_fc, Q), max_depth=5)
```

Figure 12 shows an example of fitting the Ohmic spectral density with exponential cut-off with a set of underdamped Brownian motion spectral densities (sometimes called the Meier-Tannor fitting approach).

Zero temperature and the localization-delocalization phase transition. — One of the benefits of including these fitting routines is to be able to simulate situations where the structure of the spectral density has non-trivial effects. An example of such a situation is the localization-delocalization transition in the spin-boson model [68–71]. When the temperature of the bath is $T = 0$, one expects the steady state of the system to be delocalized $\langle \sigma_z(t \rightarrow \infty) \rangle = 0$. However, when the coupling to the bath is increased the steady state goes from a completely delocalized state to what appears to be a localized state for long-times $\langle \sigma_z(t \rightarrow \infty) \rangle \neq 0$. To demonstrate this, we follow [70] and choose an Ohmic spectral density with a polynomial cutoff of the form

$$J(\omega) = \frac{\pi\alpha\omega}{2(1 + (\frac{\omega}{\omega_c})^2)} . \quad (43)$$

Environment Class Overview

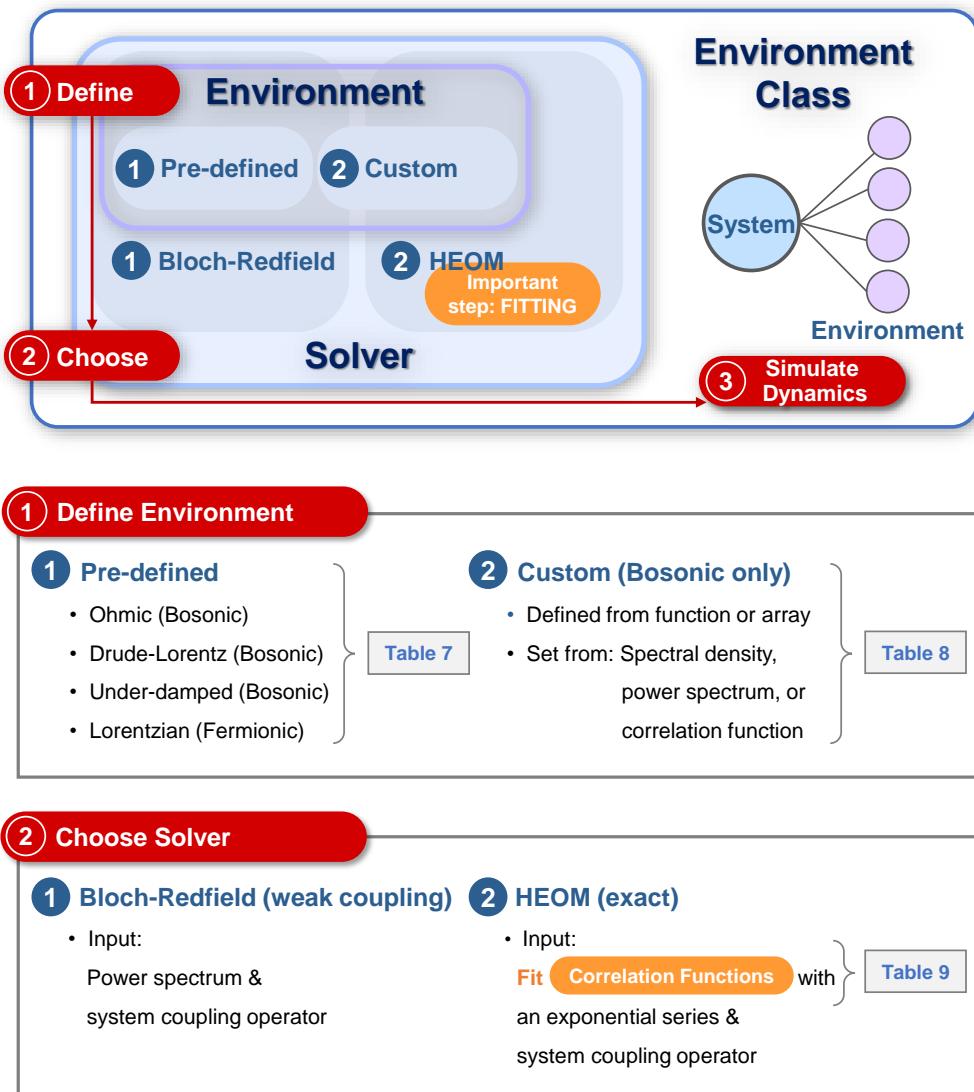


Figure 11: An overview of the environment class, which is now supported by the HEOM and Bloch-Redfield solvers. Details on how predefined and custom environments can be made is provided in tables (7) and (8), and the fitting methods that can be used to approximate the bath correlation function for use in the HEOM solver are described in table (9)

Predefined environment type	Spectral density Function	Python Function	Padé or Matsubara Available	Bosonic or Fermionic
Ohmic	$J(\omega) = \alpha(\omega^s/\omega_c^{s-1}) \exp[-\omega/\omega_c]$	OhmicEnvironment(T, alpha, wc, s)	✗	Bosonic
Drude-Lorentz	$J(\omega) = 2\lambda\gamma\omega/(\gamma^2 + \omega^2)$	DrudeLorentzEnvironment(T, lam, gamma)	✓	Bosonic
Under-Damped	$J(\omega) = \lambda^2\Gamma\omega/[(\omega_0^2 - \omega^2)^2 + \Gamma^2\omega^2]$	UnderDampedEnvironment(T, lam, Gamma, w0)	✓	Bosonic
Lorentzian	$J(\omega) = \gamma W^2/[(\omega - \omega_0)^2 + W^2]$	LorentzianEnvironment(T, mu, gamma, W, w0)	✓	Fermionic

Table 7: Predefined environments: Table of predefined environments as given by their spectral density, and the internal QuTiP function to create them. The fourth column indicates whether or not a Padé or Matsubara series expansion of the bath correlation functions is available. All baths require an initial temperature (which can be zero, though this invalidates the series expansion for the correlation functions, and fitting should be used instead). For the Fermionic Lorentzian bath we also need to provide a chemical potential (defined via the `mu` parameter).

Due to this effect being present at $T = 0$, we fit the correlation function rather than the spectral density. Obtaining the correlation function is straightforward once we create a `BosonicEnvironment`.

```
def J(w, alpha=1):
    """ The Ohmic bath spectral density as a function
    of w (and the bath parameters). """
    return (np.pi/2) * w * alpha * 1 / (1+(w/wc)**2)**2

env = BosonicEnvironment.from_spectral_density(lambda w: J(w, alpha), wMax=50, T=T)
```

For this example, let us consider the same system Hamiltonian as before, with $\omega_0 = 0$. Figure 13 shows how by increasing the coupling to the bath, the system goes from a localized state to a delocalized state. This is the so called spin-boson localization-delocalization phase transition [68].

Apart from the method used above, the QuTiP environment class provides several different ways to decompose arbitrary environments into a decaying exponential form as in Eq. (36), as needed for simulation with the HEOMSolver. Table 9 provides a summary of the methods available. For details about the methods, we refer to the documentation or Refs. [72, 73]. At present, most of these methods only work for bosonic environments; however, fermionic environments will be supported in the near future.

3.2.13. Visualization of solver results

QuTiP comes with a range of functions to visualize the results returned by its solvers. In addition to automated functions for plotting expectation values with Matplotlib, QuTiP provides utility functions to calculate important and commonly used representations of quantum states, including the Bloch-sphere for two-level systems and pseudo-probability functions, such as the Wigner and Husimi functions, for harmonic systems like cavities.

In this section, we present a few examples to illustrate the utility and features of these plotting functions. Many more examples can be found in the tutorial notebooks described in Appendix B.

For both learners and researchers, using the Bloch sphere to visualize the overlap of a qubit state with the Pauli matrices in three-dimensional space can be useful to help understanding how the state evolves, particularly when combined with QuTiP's animation features. In QuTiP, a Bloch sphere is instantiated with `b = Bloch()`. Points can be added to it using coordinates, as in `b.add_points([1/np.sqrt(3), 1/np.sqrt(3), 1/np.sqrt(3)])`, and the sphere plotted with `b.render()`. Vectors can be added in a similar way, using `b.add_vectors([0, 1, 0])`.

More commonly, one wants to visualize the output of solvers, which is provided in the form of state vectors or density operators. These objects can be added to a sphere using the method `b.add_states(state)`. The tutorial notebook `0004_qubit-dynamics`, briefly summarized in Appendix B, demonstrates this feature using the example of a driven qubit, both with and without noise, and the result is shown in Fig. 14.

Defined from:	Python Function	Important optional arguments
Spectral density	BosonicEnvironment .from_spectral_density(func)	T: temperature of the bath. wlist: array of frequencies (if array-based). wMax: maximum frequency used in Fourier-transform conversions.
Power Spectrum	BosonicEnvironment .from_power_spectrum(func)	As above
Correlation function	BosonicEnvironment .from_correlation_function(func)	T: temperature of the bath. tlist: array of times (if array-based). tMax: maximum time used in Fourier-transform conversions.
Correlation function from exponential series (bosonic or fermionic)	ExponentialBosonicEnvironment (ck_real, ck_imag, vk_real, vk_imag) ExponentialFermionicEnvironment (ck_plus, vk_plus, ck_minus, vk_minus)	T: temperature of the bath. combine: Boolean, indicates whether common-frequency com- pression is used (bosonic case only). mu: Chemical potential (fermionic case only).

Table 8: Custom environments: Table of functions for defining custom environments from either their spectral density, power spectrum or correlation functions (only custom bosonic environments are currently supported in this way, though fermionic ones can be done manually as shown in the final row in the table). The parameter `func` can be a Python function or a list/array of points (requires a corresponding `tlist` or `wlist` optional parameter to be provided). Both `wMax` and `tMax` should be chosen such that the function or array used is negligible for higher values of t or ω .

Pseudo-probability functions are also commonly used to visualize data arising from continuous variable systems. In QuTiP, such systems are truncated on finite dimensional Fock spaces, but these pseudo-probability functions can still be used, given sufficient truncation. The Wigner function, for example, helps visualize the probability of the position and momentum quadratures of cavities, and famously contains negative probabilities for non-classical states.

We can demonstrate the visualization of Wigner functions straightforwardly with another common example; a cavity prepared in a “Schrödinger cat” state, i.e., a superposition $\psi = \frac{1}{N}(|\alpha_1\rangle + |\alpha_2\rangle)$ of two coherent states (where N is a normalization factor). The Wigner function, shown in Fig. 15, clearly shows the expected negative values for such a highly non-classical state. The non-classicality is less apparent in the Husimi-Q function, also shown in Fig. 15, which is non-negative by definition.

New in QuTiP v5 is a suite of tools to automate the animation of many of these commonly used functions. These can be explored in the tutorial [74], and they include customized methods for the:

- Wigner function (`anim_wigner` and `anim_wigner_sphere`),
- Hinton plots (`anim_hinton`),
- sphere plots (`anim_sphereplot`),
- histograms (`anim_matrix_histogram`),
- Fock state distributions (`anim_fock_distribution`),
- spin distributions (`anim_spin_distribution`),
- Qubism plots (for plotting the states of many qudits, `anim_qubism`), and
- Schmidt plots (for plotting matrix elements of a quantum state, `anim_schmidt`).

In addition, there is a new option (`qutip.settings.colorblind_safe`) to choose plotting colors from a palette of colorblind safe colors.

Method	Arbitrary Functions	Allows Constraints	No need of Extra Input	Optimization Convergence	Sampling Insensitive	Arbitrary Temperatures	Works on Noisy data	Recommended when...
NLSQ (ps, sd, cf)	✓	✓	✗	✗	✗	✓	✓	You have an idea about which exponents should be included [67].
AAA	✓	✗	✓	✓	✗	✓	✓	You need high-accuracy in the steady-state and the spectral density is not too structured.
Prony	✓	✗	✓	✓	✗	✓	✗	The correlation function is noiseless and long lived.
Matsubara	✗	✗	✓	✓	✓	✗	✗	Doing high temperature simulations using the specific spectral densities it is available for.
Pade	✗	✗	✓	✓	✓	✗	✗	The same cases as the Matsubara method. This is recommended over Matsubara whenever available.
ESPIRA	✓	✗	✓	✗	✗	✓	✓	Looking for a general-purpose method.
ESPRIT	✓	✗	✓	✓	✗	✓	✓	The correlation function is long lived.

Table 9: Once an environment is created (see Tables (7) and (8)), its correlations can then be approximated as an exponential series in a variety of ways. This is done by the class method `.approximate("type")` where "type" is a string defining the method desired, as listed in the first column of this table. They take a variety of optional arguments to define fitting range or bounds, depending on the method used. Non-linear least squares fitting (NLSQ), the method used in this section, might be performed on the spectral density, the power spectrum, or the correlation function, as specified by the type strings "ps", "sd", and "cf". "Arbitrary Functions" stands for the ability of approximating an arbitrary user-defined environment. "Allows Constraints" stands for the ability to limit the range of values the parameters in Eq. 36, which is often useful when working with HEOM as it may improve convergence. "No need of Extra Input" refers to the input that the algorithm needs to work properly: we consider a set of sampling points as basic input, and other pieces of information required as extra input. "Optimization Convergence" refers to the fact that the approximation does not get stuck in local minima preventing convergence, in some cases failing to provide any approximation. "Sampling Insensitive" refers to the fact that changing the sampling points does not change the approximation. "Arbitrary Temperatures" refers to the method converging at arbitrarily low temperatures. "Works on Noisy data" applies when the user defined spectral density comes from data that is noisy.

3.3. Additional features in QuTiP v5

In addition to the new data layer and solver features described earlier, there are other new features in QuTiP v5, and some of the previously existing features have received important updates. Below, we describe in detail three such new or updated features: excitation number restricted states, which are crucial for the simulation of large composite systems, and now have improved back-end support through a new dimensions class, the option of using the JAX auto-differentiation functionality with the new JAX data layer, and support for the Message Passing Interface (MPI) in the parallelization of various solvers, which enables the easy use of super-computing resources.

3.3.1. Excitation number restricted states

When modeling many interacting quantum systems, QuTiP does not by default apply any approximation apart from truncating the individual systems' Hilbert spaces. For example, when modelling the discrete Fock-space representation of a quantum harmonic oscillator, we normally truncate the states at some finite number of excitations. This is clearly demonstrated when defining the commonly used Jaynes-Cummings

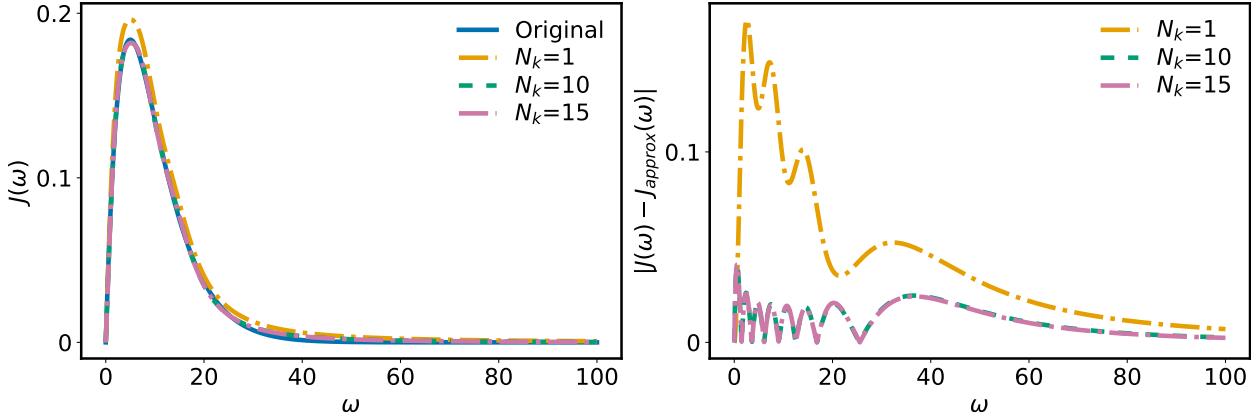


Figure 12: Approximation of the spectral density of an Ohmic Bath via fitting with three underdamped Brownian motion spectral density functions. Importantly, when fitting spectral densities for the HEOM method, we must still expand each spectral density's contribution to the total correlation functions in terms of exponentials. To gain insight of the effectiveness of both the fit and the correlation function expansion, we can then use that expansion, along with the temperature information of the bath, to reconstruct a total effective spectral density, which is what we show in this plot. In the figure, we can see that the number N_k of exponents kept per underdamped mode has an effect on the approximation of the resulting effective bath spectral density. Ultimately one should aim to find the minimum number of exponents that make up a good approximation of the spectral density. The parameters of the original Ohmic bath are $\lambda = 0.1$, $\omega_c = 5$, and $T = 1$.

model from quantum optics, which describes a single two-level atom interacting with a single-mode cavity under the rotating wave approximation:

```

N_cut = 2 #Number of Fock states to keep

psi0 = qt.basis(2, 0) & qt.basis(N_cut, 0) #initial state (qubit excited)
sz = qt.sigmax() & qt.qeye(N_cut)
sm = qt.sigmay() & qt.qeye(N_cut)
a = qt.qeye(2) & qt.destroy(N_cut)

H_JC = (
    0.5 * eps * sz + omega_c * a.dag() * a +
    g * (a * sm.dag() + a.dag() * sm)
) #Jaynes-Cummings Hamiltonian

```

In this case the Hilbert space of the cavity is truncated at two Fock states, so the total number of states is four. However, the Hamiltonian conserves the total number of excitations in the coupled system. With an initial condition containing at most a single excitation, the double-excitation state, where the atom is excited and a photon is in the cavity, is therefore decoupled from the dynamics. Using excitation number restricted (ENR) states, we can truncate the total Hilbert space to exclude the double-excitation state:

```

N_exc = 1 #Number of total excitations in truncated system
dims = [2, N_cut] #Original system dimensions (before ENR truncation)

psi0 = qt.enr_fock(dims, N_exc, [0, 0]) #Initial state in ENR space
sm, a = qt.enr_destroy(dims, N_exc) #operators in ENR space
sz = 2 * sm.dag() * sm - 1

H_enr = (
    0.5 * eps * sz + omega_c * a.dag() * a +
    g * (sm.dag() * a + a.dag() * sm)
) #Jaynes-Cummings Hamiltonian

```

Here, $N_{\text{exc}}=1$ is the maximum number of excitations we wish to consider across the whole Hilbert space. The function `enr_destroy(dims, N_exc)` returns a list of annihilation operators for each subsystem in `dims` which only act on a reduced space including states with up to that total excitation number. In this example,

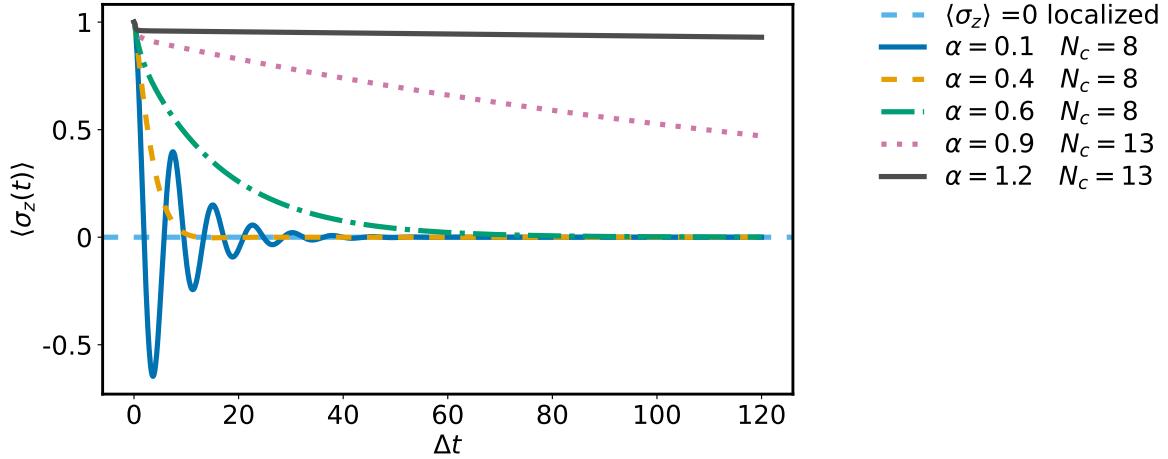


Figure 13: This figure shows the spin-boson localization-delocalization phase transition. For the simulation, we used $\omega_0 = 0$, $\omega_c = 10\Delta$ and $T = 0$. The correlation function was fit with two exponents for the real part and one for the imaginary part.

the restricted Hilbert space is spanned by $|0,0\rangle$, $|0,1\rangle$ and $|1,0\rangle$. The first annihilation operator can be thought of as the operator $|0,0\rangle\langle 1,0|$, and the second one as $|0,0\rangle\langle 0,1|$. With these constructions, we can then recreate the full Hamiltonian and dynamics of the Jaynes-Cummings model, omitting the unimportant double occupation state.

The power of this approach lies in situations with many subsystems, where one only needs to consider a limited number of excitations. One of the core QuTiP notebooks demonstrates this well, with a large chain of coupled Jaynes-Cummings models [75]. But it can also be used as a powerful truncation tool in situations where the Hamiltonian is not necessarily excitation-number conserving (see [76]).

It is important to note that since ENR states essentially compress the normal tensor structure of states and operators onto one single reduced Hilbert space, annihilation and creation operators of different subsystems no longer commute. Hence, care must be taken when representing operators on the ENR state space. Typically, when constructing Hamiltonians, one should order annihilation operators to the right and creation operators to the left. In addition, ENR states require the use of a range of custom functions, like `enr_fock()`, and many standard utility functions in QuTiP will fail when used with them. In v5 the addition of unique dimension objects for ENR states potentially allows these issues to be resolved, but general compatibility is still ongoing work.

To demonstrate the utility of ENR states in a complex problem, we now consider an important example from the literature; that of a qubit interacting with a one-dimensional waveguide [77] truncated by a mirror. Generally, this type of problem involving time-delayed feedback is difficult to model numerically [78]. One common method to capture the finite time delay of photons reaching the mirror and returning to the system is discretization of modes in the waveguide. This discretization can be performed in multiple ways (see [77] and [79]), but the approach taken in [80, 81] is particularly amenable to using ENR states. In this approach, spatial discretization and temporal discretization of the waveguide are done hand-in-hand, and open ends of the wave guide are truncated using a Monte Carlo-like measurement step. This procedure still requires modelling a large number of waveguide modes, or ‘‘boxes’’, which the authors of [80] were able to achieve using an ENR-like truncation of their basis states (albeit done manually, not using QuTiP).

It is relatively straightforward to implement this procedure in QuTiP. However, following [80], we implement the time evolution manually using a product of propagators for small discrete time steps rather than using one of the standard QuTiP solvers directly. We refer readers to [80] for a complete description, but

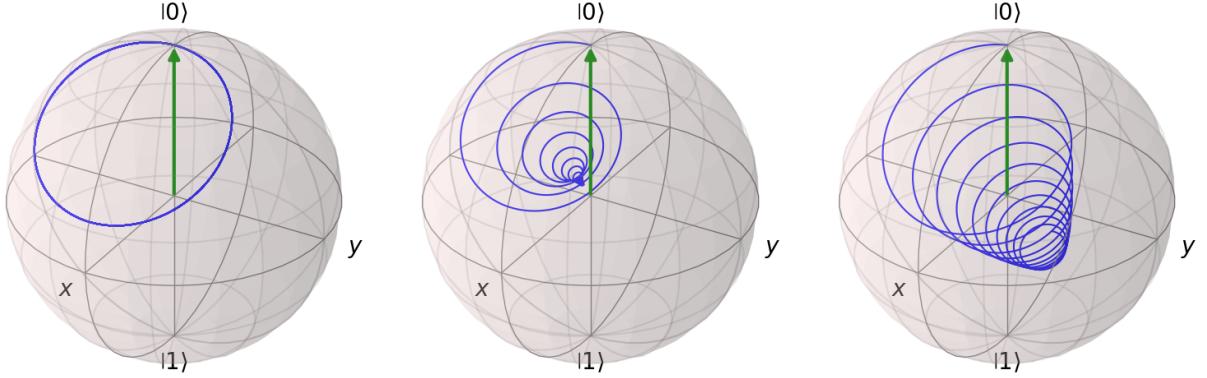


Figure 14: The left figure shows the Bloch sphere representation of the dynamics of a qubit undergoing unitary evolution under the Hamiltonian $H = \Delta[\cos(\theta)\sigma_z + \sin(\theta)\sigma_x]$ with $\theta = 0.1\pi$ for an initially excited qubit. The middle figure shows the dynamics of the same system with added dephasing, with a rate $\gamma_p = 0.5\Delta/(2\pi)$. The right figure shows the same model but with added relaxation at the rate $\gamma_r = 0.5\Delta/(2\pi)$.

succinctly, one starts with the Hamiltonian for the system coupled to a discrete-frequency waveguide:

$$H = \frac{\epsilon}{2}\sigma_z + \sum_{\alpha \in \{L,R\}} \sum_{k=0}^{N-1} \omega_k b_{k,\alpha}^\dagger b_{k,\alpha} + \sqrt{\frac{2\pi}{L_0}} \sum_{\alpha \in \{L,R\}} \sum_{k=0}^{N-1} \kappa_\alpha(\omega_k) [\sigma_+ b_{k,\alpha} + \text{H.c.}] . \quad (44)$$

Here, it was assumed that a mirror truncates the left side of the waveguide at some finite distance $L_0/2$, and the sums run over left- and right-moving modes and over their discretized frequencies ω_k . The coupling terms are $\kappa_L(\omega) = \sqrt{\gamma_L/(2\pi)}$ and $\kappa_R(\omega) = \sqrt{\gamma_R/(2\pi)}e^{i\phi}e^{i\omega\tau}$, where γ_α are coupling constants, $\tau = L_0/c$ is the total travel time of photons to the mirror and back (with c being the speed of light in the waveguide), and ϕ is an additional phase change incurred from the reflection at the mirror. In other words, a photon can propagate to the left, hit the mirror, and then return as a right propagating photon that then interacts with the system with an accumulated phase $(\omega\tau + \phi)$.

The key step is to transform the discrete frequencies into spatially discretized modes with the discrete Fourier transform

$$B_{n,\alpha} = (1/\sqrt{N}) \sum_{k=0}^{N-1} b_{k,\alpha} \exp[(\pm)_\alpha i\omega_k n \Delta t], \quad (45)$$

where $\Delta t = L_0/N$ is the time-domain sampling corresponding to the spatial discretization of the total travel length L_0 . The number of modes is N and, assuming linear dispersion, their frequencies are $\omega_k = 2\pi k/L_0$ (setting now $c = 1$). Finally, the sign in the exponent is “+” for right-moving and “−” for left-moving modes.

Under this transformation the model becomes one where photons emitted in a time interval Δt are then moved, conveyor-belt-like, through these discrete modes (“boxes”) until they hit the mirror at time $\tau/2 \equiv N\Delta t/2$, and then return in right-moving boxes until they again interact with the system at time τ . Photons emitted into the right side of the waveguide never return, and can be accommodated by projecting, at each time step, the system onto the appropriate state depending on whether a photon is observed in the right-most-box or not. As previously mentioned, this projection technique has formal similarity with the Monte-Carlo wavefunction method.

Because the interaction between the qubit and the waveguide is assumed to be weak, and in a rotating wave approximation, we can model the whole setup including the qubit and N modes efficiently using ENR states. In Fig. 16 we show, reproducing [80], the dynamics of the excited state population of the qubit with two different choices of phase, $\phi = 0$ and $\phi = \pi$, demonstrating the effect of interference with the returning photons. Here we used $N = 21$, and only a single excitation, which with ENR states can be presented with

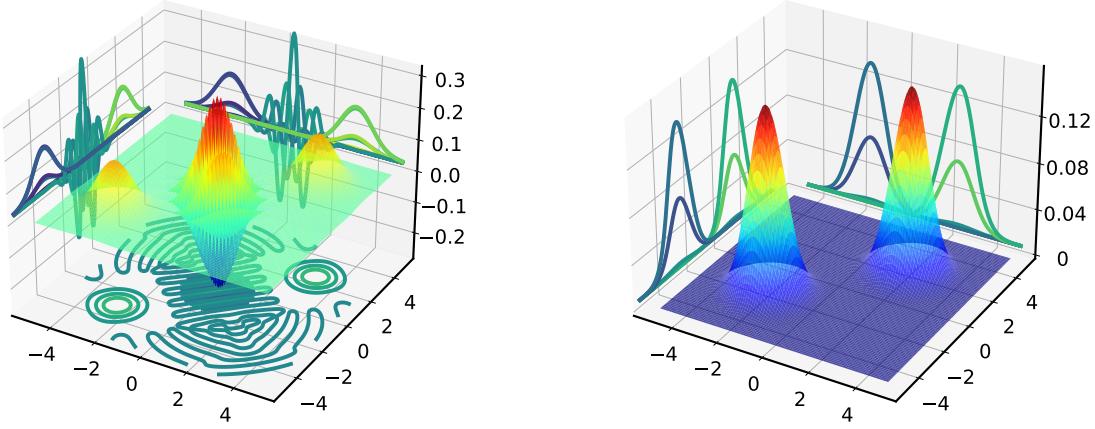


Figure 15: Left figure shows the Wigner function, a pseudo-probability distribution, of a cavity prepared in a Schrödinger cat state $\psi = \frac{1}{\sqrt{2}}(|\alpha_1\rangle + |\alpha_2\rangle)$, with $\alpha_1 = -2.0 - 2j$ and $\alpha_2 = 2.0 + 2j$. The right figure shows instead the Husimi-Q function for the same state.

a very small state space of just 23 states (to be compared to 2^{22} states needed for a brute force calculation with a Fock space truncation of each mode of just 2, without ENR states). Figure 17 shows the occupation of the waveguide modes as a function of time, illustrating the linear transportation of the excitation through the waveguide and the effect of the reflection phase ϕ on the waveguide populations.

3.3.2. Automatic Differentiation: JAX

In an earlier section, we showed how QuTiP’s new JAX data layer can be used to run calculations on the GPU. An additional feature of using the JAX data layer is access to automatic differentiation, or auto-differentiation. In problems where derivatives are important, like optimization, we must often resort to numerical approximations, e.g., finite difference, to evaluate them. When higher order derivatives are required, such approximations can be numerically costly and inaccurate. Auto-differentiation relies on the concept that any numerical function is, at a low-level, expressible in elementary analytical functions and operations. This can be exploited, via the chain rule, to give access to the derivative of almost any higher-level function.

The JAX library makes it possible to conveniently and easily use auto-differentiation for a variety of applications. For example, in the QuTiP-QOC library we take advantage of this feature to find derivatives of a control objective with respect to the control parameters in order to find the optimal pulse shape implementing a complex operation.

A full explanation and set of examples of automatic differentiation is beyond the scope of this work, but we will showcase two basic examples here: one arising from the field of counting statistics, and the other relevant to Hamiltonian control. In addition, the auto-differentiation capabilities of QuTiP-JAX will also be used in the section on the optimal control package QuTiP-QOC.

Counting statistics of an open quantum system.. – In the first example, we have an open quantum system in contact with an environment, and there is a measurement device which keeps track of excitations flowing between system and environment.

The measurement results of this process can be expressed in a variety of ways, but considerable insight can be gained by thinking about the statistics of such events; the mean, variance, skewness, and so on, of the probability distribution describing the number of excitations n that have been exchanged by a certain time t . This distribution $P_n(t)$ is called the full counting statistics, and many common experimental observables such as current or shot noise can be extracted from its properties.

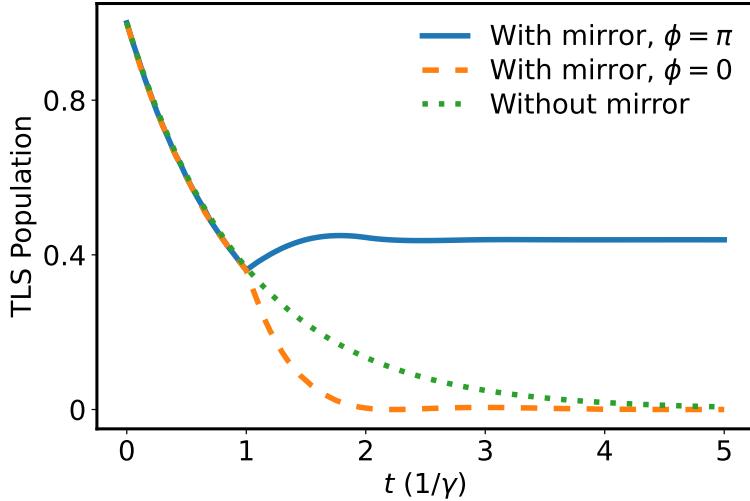


Figure 16: The dynamics of a two-level system, or qubit, interacting with a waveguide truncated at one end by a mirror. Here, we chose $\epsilon = 0$ and used $N = 21$ discrete waveguide modes, $\Delta t = L_0/N$, with $L_0 = \gamma c = 1$, with $\gamma = \gamma_L + \gamma_R$ so that the roundtrip time $\tau = 1/\gamma$. The orange curve shows the case with a mirror reflection with phase $\phi = \pi$, and the green dashed curve shows mirror reflection with phase $\phi = 0$, both averaged over 4000 trajectories. The red dotted curve shows the exponential decay expected from an open waveguide without mirror, and the blue dashed line shows the expected population at the time the first emitted photon returns and interacts with the qubit again.

When the interaction between a system and its environment is described by a Lindblad master equation, we can obtain this distribution from a slightly modified definition of the density operator and said master equation. Succinctly, one introduces the “tilted” density operator $G(z, t) = \sum_n e^{z n} \rho^n(t)$, where $\rho^n(t)$ is the density operator of the system conditioned on n jumps, or exchanges, occurring by time t and $\text{Tr}[\rho^n(t)] = P_n(t)$. It obeys the tilted equation of motion, which reads (in the case where the contact with the environment is just through a single jump operator C)

$$\dot{G}(z, t) = -\frac{i}{\hbar} [H(t), G(z, t)] + \frac{1}{2} [2e^z C \rho(t) C^\dagger - \rho(t) C^\dagger C - C^\dagger C \rho(t)] . \quad (46)$$

The dummy variable z introduced here is conventionally called a counting field. For $z = 0$, we obtain $G(0, t) = \rho(t)$, and Eq. (46) becomes the Lindblad equation (3). However, having access to $G(z, t)$ gives us access to the moments of P_n through derivatives

$$\langle n^m \rangle(t) = \sum_n n^m \text{Tr}[\rho^n(t)] = \frac{d^m}{dz^m} \text{Tr}[G(z, t)]|_{z=0} . \quad (47)$$

Normally, obtaining these derivatives would involve taking finite differences of Eq. (46), or writing explicit dynamic equations of motion for the moments themselves and solving them simultaneously. With JAX and auto-differentiation we can obtain them explicitly, as shown in the following example. We model a system with two levels, described by the annihilation operator d . This system is coupled to two reservoirs via the rates Γ_L and Γ_R . One of the couplings (governed by the rate Γ_R) is modified with a counting field as shown in Eq. (46); this is the channel whose counting statistics we monitor. This simple model is often used to study the basic dynamics of charge being transported through a single quantum dot.

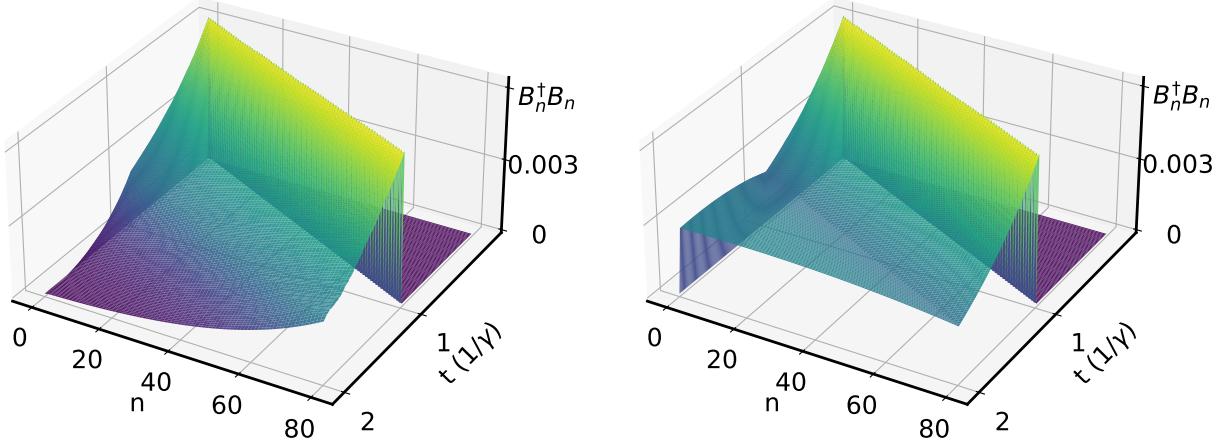


Figure 17: Occupation of the waveguide modes $B_n^\dagger B_n$, where we have increased the number of modes to 80. The time steps only extend to $\gamma t = 2$, so that in the left figure, for $\phi = 0$, we can see the overall loss of population in the waveguide modes after around $\gamma t = 1$, the round-trip time, while in the right figure, for $\phi = \pi$, we see the saturation of the occupation.

```

ed = 1 #Quantum dot energy
GammaL = 1 #Transport rate from left reservoir
GammaR = 1 #Transport rate to right reservoir

with jax.default_device(jax.devices("gpu")[0]):
    with qt.CoreOptions(default_dtype="jaxdia"):
        d = qt.destroy(2)
        H = ed * d.dag() * d #Dot Hamiltonian
        c_op_L = jnp.sqrt(GammaL) * d.dag() #collapse operator for (input) left reservoir
        c_op_R = jnp.sqrt(GammaR) * d #collapse operator for (output) right reservoir

        L0 = (
            qt.liouvillian(H) + qt.lindblad_dissipator(c_op_L)
            - 0.5 * qt.spree(c_op_R.dag() * c_op_R)
            - 0.5 * qt.spost(c_op_R.dag() * c_op_R)
        ) #Liouvillian without Jump term to right reservoir
        #Jump term to right reservoir:
        L1 = qt.sprepost(c_op_R, c_op_R.dag())
        #Find steady state:
        rho0 = qt.steadystate(L0 + L1)

    def rhoz(t, z): #returns final system state (counting field z)
        L = L0 + jnp.exp(z) * L1 # jump term with counting-field
        tlist = jnp.linspace(0, t, 50)
        result = qt.mesolve(L, rho0, tlist, options=options)
        return result.final_state.tr()

    # first derivative
    drhozdz = jax.jacrev(rhoz, argnums=1)
    # second derivative
    d2rhozdz = jax.jacfwd(drhozdz, argnums=1)

```

The code above manually constructs the Lindbladian, with the counting field on the appropriate jump term, and then solves the master equation using `mesolve()`. The solution is a function of two parameters, time and the counting field. We can then find the first and second counting-field derivatives of the state

at any time. In the long-time limit, analytical expressions for the current $I = \langle n(t) \rangle / t$ and the shot noise (variance) $S = (\langle n^2(t) \rangle - \langle n(t) \rangle^2) / t$ are well known; we compare them to the output of our numerical functions below. Note that JAX's auto-differentiation capabilities function both on CPU and GPU; we used the command `jax.default_device` to run our calculations on a GPU with no further modifications to the code required.

```
tf = 100
Itest = GammaL * GammaR / (GammaL + GammaR)
print("Analytic current", Itest)
print("Numerical current", drhozdz(tf, 0.) / tf)
print("Analytical shot noise (2nd cumulant)",
      Itest * (1 - 2 * GammaL * GammaR / (GammaL + GammaR)**2))
print("Numerical shot noise (2nd cumulant)",
      (d2rhozdz(tf, 0.) - drhozdz(tf, 0.)**2) / tf)

Analytic current 0.5
Numerical current 0.4999
Analytical shot noise (2nd cumulant) 0.25
Numerical shot noise (2nd cumulant) 0.25125
```

Derivatives of a driven Rabi model. — To further demonstrate the utility of automatic differentiation, we show in the next example how to take derivatives with respect to Hamiltonian parameters. We consider the driven Rabi model, which describes the interaction of a two-level quantum system (qubit) with an external driving field. The system is described by the time-dependent Hamiltonian

$$H(t) = \frac{\hbar\omega_0}{2}\sigma_z + \frac{\hbar\Omega}{2}\cos(\omega t)\sigma_x, \quad (48)$$

where ω_0 is the qubit energy splitting, Ω is the Rabi frequency, ω is the driving frequency, and $\sigma_{x/z}$ are Pauli matrices. In the presence of dissipation, the system's evolution is governed by the Lindblad master equation, which includes energy relaxation via collapse operators. We include the dissipation rate γ with the collapse operator $C = \sqrt{\gamma}\sigma_-$.

As we will discuss in the section on QuTiP-QOC, in quantum control and optimization tasks it is crucial to understand how physical observables depend on system parameters. Automatic differentiation provides an efficient and accurate way to compute gradients with respect to parameters such as the driving frequency ω . This allows for gradient-based optimization techniques to be applied, bypassing the inefficiencies and inaccuracies of finite differences.

In this example, we compute the gradient of the population of the excited state $P_e(t) = \langle e | \rho(t) | e \rangle$ with respect to ω . By using JAX's auto-differentiation tools in combination with QuTiP's Monte Carlo solver `mcsolve()`, we can compute the exact gradient of the final excited-state population with respect to the driving frequency. The example only showcases the ability to compute the derivative; this ability could then be employed to aid with optimization tasks, such as achieving maximum excitation of the qubit.

Below is the implementation in Python using QuTiP and JAX:

```

import jax
import jax.numpy as jnp
import qutip
import qutip_jax

# Set JAX backend for QuTiP
qutip_jax.set_as_default()

# Define time-dependent driving function
@jax.jit
def driving_coeff(t, omega):
    return jnp.cos(omega * t)

# Define the system Hamiltonian
def setup_system(omega):
    H_0 = qutip.sigmax() #Hamiltonian (time-independent)
    H_1 = qutip.sigmax() #Hamiltonian (driven part, time-dependent)
    H = [H_0, [H_1, driving_coeff]] #Total Hamiltonian (QobjEvo)
    return H

gamma = 0.1 # Dissipation rate
c_ops = [jnp.sqrt(gamma) * qutip.sigmax()]
psi0 = qutip.basis(2, 0) #Initial state

tlist = jnp.linspace(0.0, 10.0, 100) #time steps

e_ops = [qutip.projection(2, 1, 1)]

# Objective function: simulate and return the population of the excited state at final time
def f(omega):
    H = setup_system(omega)
    result = qutip.mcsolve(H, psi0, tlist, c_ops, e_ops, ntraj=100, args = {"omega": omega})
    return result.expect[0][-1]

# Compute gradient of the excited-state population w.r.t. omega
grad_f = jax.grad(f)(2.0)

```

3.3.3. MPI support for high performance computing

It is in the nature of the trajectory solvers `mcsolve()`, `nm_mcsolve()` and `smesolve()` that their simulations can be easily parallelized. Previously, these functions accepted a `map_func` argument which could be set to either `serial_map` or `parallel_map` to either simulate only one trajectory at a time, or multiple trajectories in parallel. The function `parallel_map` utilizes Python's multiprocessing module, which runs multiple processes on the same computer.

In QuTiP version 5, the `map_func` argument has given way to the "map" option, which may be set either to "serial" or "parallel" in order to invoke `serial_map()` or `parallel_map()`, or to one of the new options "loky" or "mpi". The option "loky" is mostly equivalent to "parallel" (but may be more performant in some situations). We will in the following discuss the remaining option, "mpi".

The Message Passing Interface (MPI) is a standardized API facilitating parallel computations on multiple nodes of parallel computing architectures such as high performance computing clusters. In Python, this API can be conveniently accessed through the *MPI for Python* package [82–85]. When the option "mpi" is passed to one of QuTiP's trajectory solvers, an instance of this package's `MPIPoolExecutor` class is created that QuTiP will rely on for the parallelization. It is strongly recommended to also pass the option "num_cpus" which determines the number of worker processes to use. The environment must therefore be configured to allow the application to use at least this number of processes, plus one (for the parent process). The following code snippet demonstrates the use of this option with the Monte Carlo solver:

```

result = qt.mcsolve(
    H, initial_state, tlist, collapse, ntraj=NUM_TRAJECTORIES,
    options={"progress_bar": False, "map": "mpi", "num_cpus": NUM_WORKER_PROCESSES}
)

```

The exact procedure how to set up an environment in which *MPI for Python* can successfully interact with an MPI implementation depends strongly on which versions of what MPI implementations and of what job schedulers are available on the cluster. Providing a guide for this task goes beyond the scope of this text, and of the QuTiP project. QuTiP simply assumes that the environment is set up correctly, and provides the additional option "mpi_options" through which users may provide a dictionary with configuration options that will be passed directly to the constructor of the `MPIPoolExecutor`.

We note that the `MPIPoolExecutor` class is based on the `MPI_Comm_spawn` routine, which may not be available in all environments. For environments where it is not, *MPI for Python* provides a workaround where one replaces the normal script invocation "`python <FILE>`" with

```
mpiexec -n <N> python -m mpi4py.futures <FILE>
```

The `mpiexec` command will then immediately spawn the requested number N of processes. *MPI for Python* handles the management of these processes and behaves as if the `MPIPoolExecutor` had spawned the worker processes. The use of this workaround should not necessarily harm the performance, but it means that the full number of worker processes will live during the entire runtime of the application.

4. QuTiP's other main packages

In QuTiP v5, the structure of the QuTiP library has changed, as we have chosen to move large, independent, features into distinct sub-packages. Primarily, this choice was made to minimize the number of external dependencies of the core of QuTiP, increasing its maintainability, but it also enables the development of data layer “plug-ins” like QuTiP-JAX and other experimental ones. In this section, we will review the main sub-packages, QuTiP-QOC and QuTiP-QIP.

4.1. Optimal control: QuTiP-QOC

Quantum systems are generally sensitive to external perturbations. On the one hand, this sensitivity can be used to perform precise measurements or operations, but on the other hand, it makes the implementation of quantum devices difficult by introducing noise and errors. Therefore, finding the optimal control fields that achieve a desired quantum operation under various objectives (e.g., minimum energy consumption or maximum robustness to noise) is a challenging and important problem. In practice, there are often constraints on the control fields, such as bounds on their bandwidth, amplitude or duration. These factors make quantum optimal control a complex and rich field of research, with diverse methods and applications.

Among the most frequently used techniques for finding optimal control functions are the GRAdient Ascent Pulse Engineering (GRAPE) and Chopped RAndom Basis (CRAB) methods, which are both supported by the quantum optimal control package of QuTiP v4, QuTiP-QTRL [86–88]. Along with the QuTiP v5 release comes the new family package QuTiP-QOC [89], which includes both of these methods as well as two new ones. Further, it introduces a general control framework to address pulse optimization in a customizable manner by providing keyword argument access to the underlying QuTiP and SciPy functions. The new optimization techniques are the Gradient Optimization of Analytic conTrols (GOAT) algorithm [90] and the JAX OPTimization (JOPT), which is based on the auto-differentiation [91] capabilities of JAX and seamlessly integrates with QuTiP-JAX.

Both new techniques, GOAT and JOPT, work with analytic control functions and offer the possibility to treat the system evolution time as a variable optimization parameter. Additionally, all pulse optimization routines now come with a global and local parameter search, making it easier to escape local minima. See Table 10 for an overview of all supported features.

	analytic	local	global	time	multi-objective
GRAPE	no	v4+	v5	no	v5
CRAB	v4+	v4+	v5	no	v5
GOAT	v5	v5	v5	v5	v5
JOPT	v5	v5	v5	v5	v5

Table 10: Support for parameterized analytic control functions, local and global parameter search, variable evolution time and multi-objective optimization in the QuTiP v4 (QuTiP-QTRL) and v5 (QuTiP-QOC) packages.

Basic example of optimal control: optimizing the Hadamard gate. — The following example shows how to use all of the above-mentioned methods to find optimal control parameters to implement a Hadamard gate on a single qubit. In general, the qubit might be subject to dissipation, captured in the Lindbladian formulation with the jump operator σ_- . For simplicity we assume parameterized σ_x, σ_y and σ_z rotations for the control Hamiltonian

$$H_c(t) = c_x(t)\sigma_x + c_y(t)\sigma_y + c_z(t)\sigma_z, \quad (49)$$

where $c_x(t)$, $c_y(t)$ and $c_z(t)$ are independent control functions. Furthermore, we model a constant drift Hamiltonian

$$H_d = \frac{1}{2}(\omega\sigma_z + \delta\sigma_x) \quad (50)$$

with associated energy splitting ω and tunneling rate δ . The amplitude damping rate for the collapse operator $C = \sqrt{\gamma}\sigma_-$ is denoted γ . The total time evolution of the qubit is therefore assumed to have the form

$$\dot{\rho}(t) = -\frac{i}{\hbar}[H_d + H_c(t), \rho(t)] + \frac{1}{2} [2C\rho(t)C^\dagger - \rho(t)C^\dagger C - C^\dagger C\rho(t)]. \quad (51)$$

By default, in the case of open system state transfer or map synthesis (as in this example), the optimization will minimize the trace distance to the target state or channel. In the case of closed system state transfer or gate synthesis objectives, it will minimize the overlap with the target vector or gate. Even though not explicitly shown in the following example, all algorithms can be run with multiple objectives (list of `qoc.Objective` instances), where each objective can be supplied with an additional weight parameter.

```
import qutip_qoc as qoc

# objective
initial = qt.qeye(2) # identity
target = qt.gates.hadamard_transform()

# energy splitting, tunneling, amplitude damping
omega, delta, gamma = 0.1, 1.0, 0.1
sx, sy, sz = qt.sigmax(), qt.sigmay(), qt.sigmaz()

Hc = [sx, sy, sz] # control operator
Hc = [qt.liouvillian(H) for H in Hc]

Hd = 1 / 2 * (omega * sz + delta * sx) # drift term
Hd = qt.liouvillian(H=Hd, c_ops=[np.sqrt(gamma) * qt.sigmax()])
```

4.1.1. The GRAPE algorithm

The GRAPE algorithm, initially designed for nuclear magnetic resonance (NMR) pulse sequences, has applications in various physical systems, including superconducting qubits, and can be used to optimize noisy quantum devices in QuTiP [21, 87, 92]. By minimizing an infidelity loss function that measures how close the final state or unitary transformation is to the desired target, the algorithm optimizes evenly spaced piecewise constant pulse amplitudes. The random or educated initial guess control pulse is updated iteratively according to the derivative of the loss function.

```

# combined operator list
H = [Hd, Hc[0], Hc[1], Hc[2]]

# pulse time interval
times = np.linspace(0, np.pi/2, 100)

# run the optimization
res_grape = qoc.optimize_pulses(
    objectives=qoc.Objective(initial, H, target),
    control_parameters={
        "ctrl_x": {
            "guess": np.sin(times),
            "bounds": [-1, 1]
        },
        "ctrl_y": {
            "guess": np.cos(times),
            "bounds": [-1, 1]
        },
        "ctrl_z": {
            "guess": np.tanh(times),
            "bounds": [-1, 1]
        }
    },
    tlist=times,
    algorithm_kwargs={
        "alg": "GRAPE",
        "fid_err_targ": 0.01
    }
)

```

GRAPE concurrently updates the pulse amplitudes by calculating the derivatives using the discretized unitary forward and backward evolution operators. This approach bears similarities with the Krotov method that is also available as a QuTiP affiliated package, but instead updates the control intervals sequentially [93]. The QuTiP native Hamiltonian formulation makes it easy to define even multiple objectives in a common fashion for low effort comparisons with the Krotov library [93, 94]. The GRAPE algorithm is capable of accommodating various constraints on control fields, such as amplitude limits and offsets, for a range of default initial pulses (e.g., Gaussian, square, etc.), while striving to achieve the desired target infidelity, specified by the `fid_err_targ` keyword.

4.1.2. The CRAB algorithm

The CRAB algorithm has been applied to a range of challenging problems, like phase transitions in many-body systems, implementations of gates on transmon qubits, and entanglement generation for communication [95, 96]. It is based on the idea of expanding the control fields in a random basis and optimizing the expansion coefficients $\vec{\alpha}$. This has the advantage of using analytical control functions $c(\vec{\alpha}, t)$ on a continuous time interval, and is by default a Fourier expansion. Instead of calculating the gradient with respect to individual time slots, the search space is reduced to the function parameters. Typically, these parameters have one order of magnitude fewer dimensions and can efficiently be calculated through direct search algorithms (like Nelder-Mead). The basis function is only expanded for some finite number of summands and the initial basis coefficients are usually picked at random.

The implementation in QuTiP-QOC also provides the possibility to balance multiple objectives in order to account for, e.g., variations in the control fields. Even though the optimization is performed over the function basis parameters, it is possible to initialize the overall pulse shape in the same way as with the GRAPE algorithm. On top of the original QuTiP-QTRL implementation, the method can now also exploit domain knowledge by providing an initial frequency (enabled through the `fix_frequency` keyword) and amplitudes for the Fourier expansion.

```

# c0 * sin(c2*t) + c1 * cos(c2*t) + ...
n_params = 3 # adjust in steps of 3

# run the optimization
res_crab = qoc.optimize_pulses(
    objectives=qoc.Objective(initial, H, target),
    control_parameters={
        "ctrl_x": {
            "guess" : [1 for _ in range(n_params)],
            "bounds": [(-1, 1)] * n_params
        },
        "ctrl_y": {
            "guess" : [1 for _ in range(n_params)],
            "bounds": [(-1, 1)] * n_params
        },
        "ctrl_z": {
            "guess" : [1 for _ in range(n_params)],
            "bounds": [(-1, 1)] * n_params
        }
    },
    tlist=times,
    algorithm_kwargs={
        "alg": "CRAB",
        "fid_err_targ": 0.01,
        "fix_frequency": False
    }
)

```

4.1.3. The GOAT algorithm

Similar to CRAB, this method also works with analytical control functions. By constructing a coupled system of equations of motion, the derivative of the (time ordered) evolution operator with respect to the control parameters can be calculated after numerical forward integration. In unconstrained settings, GOAT was found to outperform the previously described methods in terms of convergence and fidelity achievement [97]. Our QuTiP implementation allows for arbitrary control functions provided together with their respective derivatives in a common Python manner.

```

def sin(t, c):
    return c[0] * np.sin(c[1] * t)

def grad_sin(t, c, idx):
    if idx == 0: # w.r.t. c0
        return np.sin(c[1] * t)
    if idx == 1: # w.r.t. c1
        return c[0] * np.cos(c[1] * t) * t
    if idx == 2: # w.r.t. time
        return c[0] * np.cos(c[1] * t) * c[1]

```

For even faster convergence, it extends the original algorithm with the option to optimize controls with respect to the overall time evolution, which can be enabled by specifying the additional time keyword argument.

```

# similar to qutip.QobjEvo
H = [Hd] + [[hc, sin, {"grad": grad_sin}] for hc in Hc]

ctrl_parameters = {
    id: {
        "guess": [1, 0], # c0 and c1
        "bounds": [(-1, 1), (0, 2*np.pi)]
    } for id in ['x', 'y', 'z']
}

# magic kwd to treat time as optimization variable
ctrl_parameters["__time__"] = [
    "guess": times[len(times) // 2],
    "bounds": [times[0], times[-1]],
]

# run the optimization
res_goat = qoc.optimize_pulses(
    objectives=qoc.Objective(initial, H, target),
    control_parameters=ctrl_parameters,
    tlist=times,
    algorithm_kwargs={
        "alg": "GOAT",
        "fid_err_targ": 0.01,
    }
)

```

4.1.4. Integration with QuTiP-JAX

As discussed earlier, QuTiP’s new JAX backend provides state-of-the-art automatic differentiation capabilities. Using the chain rule of differentiation for elementary computer operations, automatic differentiation is almost as exact as symbolic differentiation and is most commonly used in machine learning [98]. Recently it has also found its way into quantum optimal control [99]. Through the JAX backend, these capabilities can be used with the new control framework. As with QuTiP’s GOAT implementation, any analytically defined control function can be handed to the algorithm. However, in this method, JAX automatic differentiation abilities take care of calculating the derivative throughout the whole system evolution. Therefore the user does not have to provide any derivatives manually. In the previous example, this simply means to swap the control functions with their just-in-time compiled version:

```

import jax

@jax.jit
def sin_x(t, c, **kwargs):
    return c[0] * jax.numpy.sin(c[1] * t)

# same for sin_y and sin_z ...

H = [Hd] + [[Hc[0], sin_x], [Hc[1], sin_y], [Hc[2], sin_z]]

res_jopt = qoc.optimize_pulses(
    objectives=qoc.Objective(initial, H, target),
    control_parameters=ctrl_parameters,
    tlist=times,
    algorithm_kwargs={
        "alg": "JOPT",
        "fid_err_targ": 0.01,
    }
)

```

After running the global and local optimization, one can compare the results obtained by the various algorithms through a `qoc.Result` object, which provides common optimization metrics along with the `optimized_controls` (see Fig. 18).

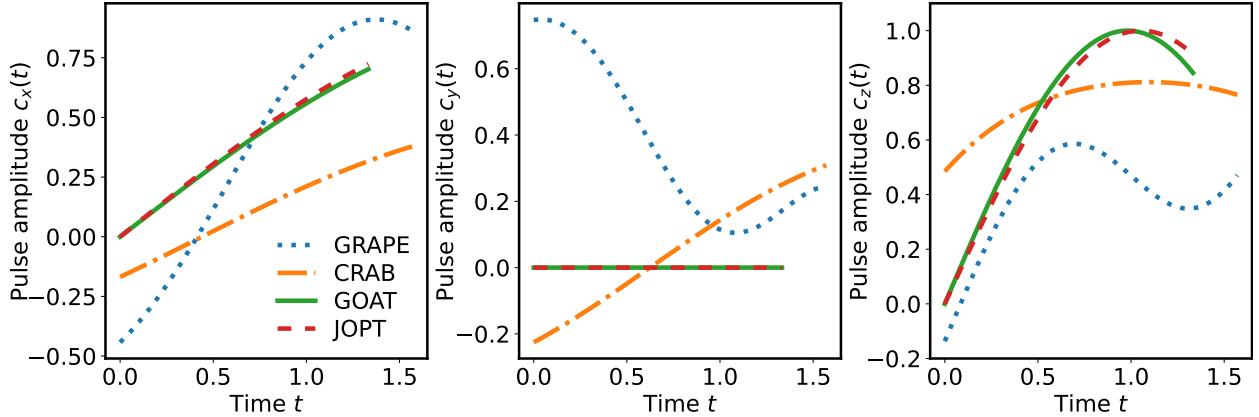


Figure 18: Optimized pulse amplitudes implementing the Hadamard operator for a single qubit system with the control Hamiltonian (49). The amplitudes were obtained through the various algorithms provided by the QuTiP-QOC package. For this simple example, all algorithms quickly find pulse shapes with the requested fidelity (greater than 0.99). Note that the GOAT and JOPT curves end early because optimization was also performed over the final time.

4.2. Quantum circuits: QuTiP-QIP

Quantum circuits are a common conceptual and visual tool to represent and manipulate algorithms running on quantum computers. QuTiP supports this kind of language with the QuTiP-QIP [100] family package, which offers several unique features. First, like the rest of QuTiP, it is one of the most popular fully independent and academically supported packages for its task. Second, it allows seamless integration of the unitaries representing a given circuit with the rest of QuTiP via the `Qobj` class. Finally, it integrates with both QuTiP-QTRL (soon QuTiP-QOC) and the various QuTiP open-system solvers to allow for pulse-level simulation of quantum circuits, including realistic noise.

The full scope of this package was recently demonstrated in [21]. In this section, we first show the newly added circuit visualization feature, then walk through a new example to demonstrate some of the unique features of QuTiP-QIP. To tie into the earlier examples in this paper, and the general theme of QuTiP of modeling open quantum system dynamics, our aim is to show: (i) How to construct a simple quantum circuit which simulates the dynamics of a given Hamiltonian (in this case, Eq. (1)). (ii) How to simulate the dynamics of an open system, i.e., a Lindblad equation, using ancillas to induce the correct noisy dynamics, Eq. (3). (iii) How to then run both simulations on a hardware backend, termed a processor, that simulates itself the intrinsic noisy dynamics of a given quantum hardware implementation.

4.2.1. Circuit Visualization

In parallel to the QuTiP v5 release, a recent release of QuTiP-QIP introduced enhanced circuit visualization capabilities by extending the list of available renderers to include also Matplotlib-based and text-based renderers in addition to the previously available LaTeX renderer. This update makes circuit visualization more accessible and reduces (heavy and complex) reliance on external LaTeX dependencies. The Matplotlib renderer is particularly useful for generating visually appealing and highly customizable quantum circuits, while the text-based renderer is designed for quick, lightweight checks, ideal for development in command line interfaces or in environments with limited dependencies. Support for the LaTeX renderer remains available.

Additionally, the visualization of quantum circuits has been significantly streamlined through the use of the `draw` API. For instance, a quantum circuit can be defined as demonstrated below.

```

qc = QubitCircuit(2, num_cbits=1) #Circuit consists of two qubits + one classical register
qc.add_gate("H", 0) #Add a Hadamard on qubit '0'
qc.add_gate("H", 1) #Add a Hadamard on qubit '1'
qc.add_gate("CNOT", 1, 0) #Add a CNOT between 0 and 1
qc.add_measurement("M", targets=[0], classical_store=0) #Add a measurement, store result in
#                                         classical register

```

Once defined, the circuit can be rendered using one of the following commands, depending on the desired output format as illustrated in Fig. 19.

```

qc.draw("latex")
qc.draw("text")
qc.draw("matplotlib")

```

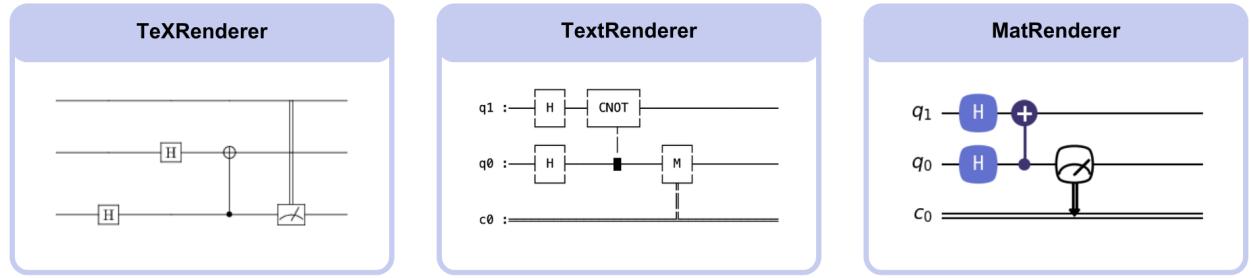


Figure 19: The figure illustrates the visual output of three different quantum circuit renderers available in QuTiP-QIP: TeXRenderer (utilizes LaTeX for rendering), TextRenderer (utilizes ASCII characters for lightweight visualization), and MatRenderer (leverages Matplotlib for customizable graphics).

Both new renderers utilize a layer-based system for gate placement, optimizing circuit compactness and enhancing the clarity of parallel computation steps during simulation. Customization options range from basic changes such as renaming gates or adjusting colors and titles to advanced features like modifying gate shapes, alignment, and spacing. The MatRenderer provides customization flexibility at both the gate and circuit levels, enabling users to tailor circuit representations to meet specific requirements.

4.2.2. Simulating Hamiltonian dynamics

In quantum simulation, one standard approach to simulating the dynamics of a quantum system is reducing the propagation of the Schrödinger equation into a discrete set of short time steps [101, 102] (though analog simulations are also important in certain situations [103]). The propagator in one time step is approximated by “Trotterization”,

$$\psi(t_f) = e^{-i(H_A+H_B)t_f} \psi(0) \approx [e^{-iH_A dt} e^{-iH_B dt}]^d \psi(0), \quad (52)$$

for sufficiently small time steps $dt = t_f/d$, which are repeated d times (higher-order approximations can also be performed, which allows for larger time steps).

The Hamiltonians H_A and H_B here should be chosen such that the individual unitaries $e^{-iH_{A,B}dt}$ can be easily mapped to basic quantum gates in the circuit model. When the simulated system can be represented as spin operators with some limited range of interactions, this mapping to qubits is straightforward [104, 105], as with our simple example below. Alternatively, in a more general context, if the problem is represented as an abstract set of N states, and $N \times N$ operators, various mappings are possible [106–109]. A common approach is to map the N states to the states described by $n = \log_2(N)$ qubits. The $N \times N$ operators can then be mapped to strings of Pauli operators, for which efficient gate-mappings are known [104]. However, the number of strings required will be exponential in n for dense problems, so some degree of sparsity or structure, and efficient access to matrix elements, is needed.

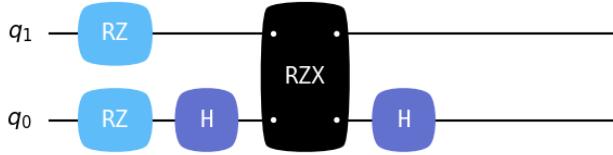


Figure 20: Circuit model of a single Trotterization step of (53) and (54).

The example of two interacting qubits defined in Eq. (1) can be expressed in the above formula, Eq. (52), with

$$H_A = \frac{\epsilon_1}{2} \sigma_z^{(1)} + \frac{\epsilon_2}{2} \sigma_z^{(2)} \quad \text{and} \quad (53)$$

$$H_B = g \sigma_x^{(1)} \sigma_x^{(2)}. \quad (54)$$

In this case there is no difficulty with mapping the Hilbert space of the problem (two-level systems) being simulated to the circuit model (qubits). In more complex cases, various mappings exist for e.g. fermions, bosons or large spins, but they are not yet natively supported in QuTiP-QIP.

To implement (52), we simply need to construct a circuit consisting of two qubits and a set of gates $A_1 = \exp[-i\epsilon_1 \sigma_z^{(1)} dt/2]$, $A_2 = \exp[-i\epsilon_2 \sigma_z^{(2)} dt/2]$ and $B = \exp[-ig \sigma_x^{(1)} \sigma_x^{(2)} dt]$ acting repeatedly (d times) on a given initial condition. The gates one can use to represent these operations depend on the operations available for the underlying hardware. We use the predefined gates RZ defining a rotation around the Z axis, and a combination of Hadamard gate and ZX rotation RZX gates to implement the XX interaction unitary, as these gates are native operations on the superconducting qubits processor SCQubits.

To define this circuit and its constituent gates we use the `QubitCircuit` class:

```
trotter_simulation = QubitCircuit(2) #Create circuit consisting of 2 qubits
```

Here we initialize a circuit with just two qubits. We can also choose to have classical bits as well, which can be used to store measurements and implement feedback. These are not needed here, however. We now add the gates which implement the Trotterization of (53) for a small time step $dt = t_f/d$

```
trotter_simulation.add_gate("RZ", targets=[0], arg_value=(epsilon1 * dt))
trotter_simulation.add_gate("RZ", targets=[1], arg_value=(epsilon2 * dt))
```

These RZ gates rotate both qubits around the Z axis by an angle given by `arg_value`. The interaction between the two qubits, (54), can then be implemented with the gates:

```
trotter_simulation.add_gate("H", targets=[0]) #Hadamard on qubit 0
trotter_simulation.add_gate("RZX", targets=[0, 1], arg_value=g * dt * 2) #RZX gate
trotter_simulation.add_gate("H", targets=[0]) #Hadamard on qubit 0
```

A full list of available gates can be found in the official documentation for QuTiP-QIP. In addition, custom gates can be easily implemented. One important point however is that, currently, not all gates can be decomposed into a set of native gates for a given hardware processor, as we will show below. Work is ongoing to have a gate decomposition graph for all predefined gates. In addition, a feature we will not demonstrate here is the simulation of measurements. Instead, we will just take the final output of the circuit and manipulate it as a standard quantum object.

We can directly visualize the circuit we just constructed, see Fig. 20. To construct a larger circuit to simulate more time steps we can keep appending the same circuit we defined above, d times, until we simulate the full integration time t_f . Alternatively, we can repeatedly run the above circuit, saving the output and using it as the input for the same circuit in the next iteration. For our purposes, the latter option is easier, as it allows us to save the state at all time steps:

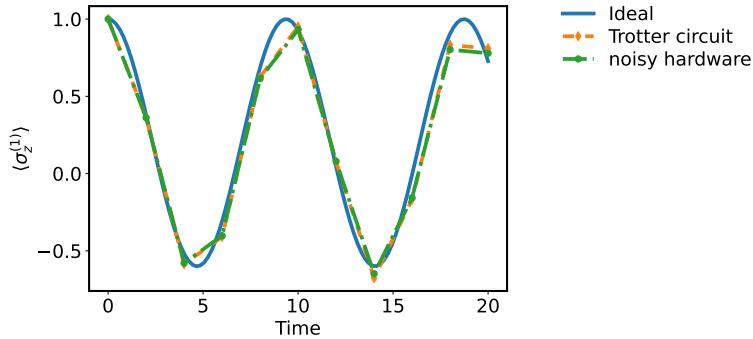


Figure 21: Results for the simulation of the closed system dynamics of the system described with the Hamiltonian (1) for $g = 0.3\epsilon_2$, $\epsilon_1 = 0.7\epsilon_2$. The circuit result, in blue, is the result of repeating the circuit described in Fig. 20 $d = t_f/dt$ times, with $t_f = 20\epsilon_2^{-1}$, $dt = 2\epsilon_2^{-1}$ and $d = 10$. In the processor simulation using a superconducting qubit backend, we set $T_1 = T_2 = 2 \times 10^5 \epsilon_2^{-1}$ to see the influence of finite hardware noise on the simulation while still obtaining a reasonable approximation to the full simulation.

```
# Evaluate multiple iteration of a circuit
result_circ = init_state
state_trotter_circ = [init_state]

for dd in range(num_steps):
    result_circ = trotter_simulation.run(state=result_circ) #Use previous result as start
                                                               of next step
    state_trotter_circ.append(result_circ)
```

The result of repeating this procedure d times (equivalent to the `numsteps` parameter in the code example) is shown in Fig. 21. We compare the result to the exact integration solution and to the result from using a processor backend, which includes noise and pulse-level simulation. Note that we chose the two qubits to be off-resonant to demonstrate the effect of finite Trotterization error (which would be negligible on resonance). We can thus see the trade-off between the Trotterization error and hardware errors: longer time steps in the simulation increase the Trotterization error but reduce the number of gates and, hence, the influence of hardware errors.

To run a given quantum circuit on a processor, we simply initialize the desired processor and then load the circuit into it. For example, here we use the superconducting circuit processor:

```
processor = SCQubits(num_qubits=2, t1=2e5, t2=2e5) #Superconducting circuits processor
processor.load_circuit(trotter_simulation) #Load circuit into processor
# Since SCQubit are modelled as qutrit, we need three-level systems here for initial
# condition:
init_state = tensor(basis(3, 0), basis(3, 1))
```

Similar to the direct simulation of the circuit, we define an initial condition and then run the processor with `processor.run_state(init_state)`. The output is now a `Result` object from the QuTiP solver being used. In this case, the solver is `mesolve()` as we specified finite T_1 and T_2 times when initiating the processor. The processor itself is defined internally by a Hamiltonian, available control operations and pulse shapes, T_1 and T_2 times and so on. It assumes a set of default parameters, but these can be overwritten upon initialization (as with T_1 and T_2 above). We can see the pulse shapes used in the solver by calling `processor.plot_pulses()`. An example of the compiled control pulse of the circuit in Fig. 20 is given in Fig. 22. A discussion of the full potential of processors, and how to implement custom ones, can be found in [21] and the package documentation.

4.2.3. Simulating master equation dynamics

We now demonstrate a more complex example where we can take advantage of QuTiP-QIP's easy integration with standard `Qobj` objects. This example shows that, interestingly, noise-free unitary quantum

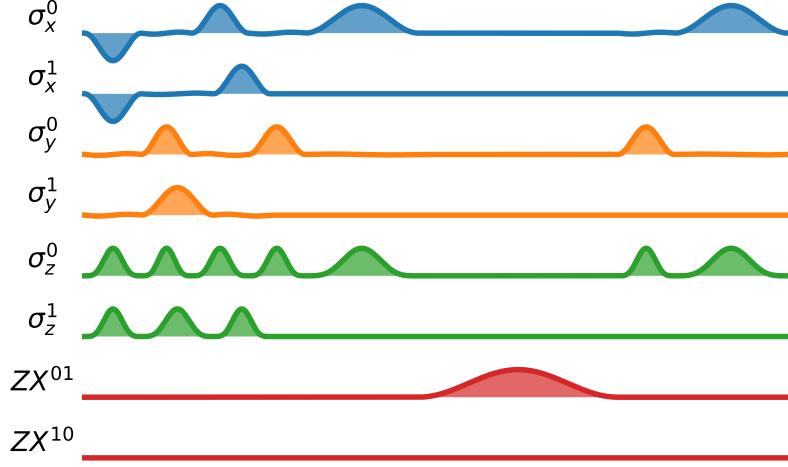


Figure 22: Pulse-level decomposition of the circuit in Fig. 20 as used by the superconducting qubits processor.

circuits can be used to simulate open noisy quantum systems. This type of simulation has a range of interesting applications, like dissipative state engineering [110–114], dissipative error correction [115], and modelling many-body dissipative systems.

To realize the Lindblad master equation described in (3), we implement a recent proposal [116, 117], which employs a single ancilla, and measurements/resets, for each Lindblad collapse operator. At $t = 0$, we prepare the dilated state

$$|\psi_D(t=0)\rangle = |\psi(t=0)\rangle \otimes |0\rangle^{\otimes K} \quad (55)$$

for the K total ancillas. At every time step dt , the system interacts with the ancillas for a time \sqrt{dt} , after which the ancillas are reset again to their ground-state. The unitary operation part of the interaction between system (i) and its associated ancilla for collapse operator (k) is given by

$$U(\sqrt{dt}) = \exp\left[-i\sqrt{\gamma_k}(\sigma_-^{(i)}\sigma_+^{(k)} + \sigma_+^{(i)}\sigma_-^{(k)})\sqrt{dt}\right]. \quad (56)$$

The unitary $U(\sqrt{dt})$ has a rather complex decomposition in terms of native gates for the backends used here, so instead we impose another Trotter approximation and write

$$\begin{aligned} U(\sqrt{dt}) &= \exp\left[-\frac{i}{2}(\sigma_x^{(i)}\sigma_x^{(k)} + \sigma_y^{(i)}\sigma_y^{(k)})\sqrt{\gamma_k dt}\right] \\ &\approx \exp\left[-\frac{i}{2}\sigma_x^{(i)}\sigma_x^{(k)}\sqrt{\gamma_k dt}\right] \exp\left[-\frac{i}{2}\sigma_y^{(i)}\sigma_y^{(k)}\sqrt{\gamma_k dt}\right], \end{aligned} \quad (57)$$

and then we decompose

$$\exp\left[-\frac{i}{2}\sigma_x^{(i)}\sigma_x^{(k)}\sqrt{\gamma_k dt}\right] = H^{(i)} U_{RZX}(\sqrt{\gamma_k dt}) H^{(i)} \quad (58)$$

as before, and

$$\exp\left[-\frac{i}{2}\sigma_y^{(i)}\sigma_y^{(k)}\sqrt{\gamma_k dt}\right] = U_{RX}^{(i)}\left(\frac{\pi}{2}\right)^{(k)} U_{RZ}^{(k)}\left(-\frac{\pi}{2}\right) U_{RZX}(\sqrt{\gamma_k dt}) U_{RX}\left(-\frac{\pi}{2}\right)^{(i)} U_{RZ}^{(k)}\left(\frac{\pi}{2}\right). \quad (59)$$

After these operations have been applied, we trace out the ancillas, project them to their ground states, and use the new state

$$|\psi(dt)\rangle = \text{Tr}_{\otimes K} [|\psi_D(dt)\rangle \langle \psi_D(dt)|] \otimes |0\rangle \langle 0|^{\otimes K} \quad (60)$$

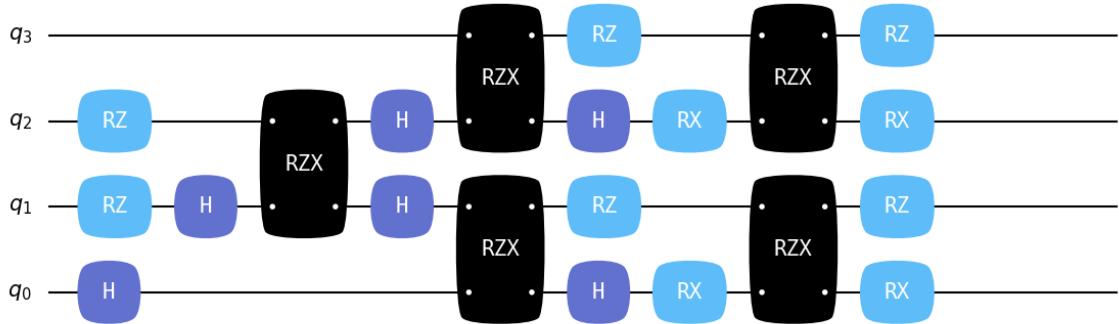


Figure 23: Circuit model of a single Trotterization step of (53) and (54), followed by the interaction with two ancillas implementing (56).

for the next time step. We show the circuit which implements a single time step in Fig. 23. Note that to avoid unnecessary two-qubit operations between non-neighbouring states, we use the middle two qubits as the system qubits and the first and last one as ancillas. The compiled pulse is shown in Fig. 24, which is much longer than the first one. Therefore, the evolution will suffer more from the decoherence of the physical qubits. In Fig. 25, we show the results of a standard `mesolve()` simulation, the Trotterized circuit simulation with ancillas, and the circuit simulation running on the same spin-chain backend as before.

5. QuTiP’s community

QuTiP is an open source package that operates like an open-door community, with an active mailing list giving feedback on common problems and applications. Community contributions to the QuTiP package are welcome, and only limited by the free time of the largely voluntary development and administration team.

5.1. The admin team and governance

The administration team, which reviews code, makes contributions, and decides on current and future development plans, is made up of members from across the globe. Regular volunteers or students who show a perspicacity are nominated by current administrators to join the team, which gives them authority to join monthly administration meetings and approve pull requests. Other administration members are sometimes technical staff, directly hired by one of the academic institutions supporting QuTiP. Particular note should be given to support from JST Moonshot program, which supported several staff members in RIKEN, and Institute Quantique, which supported Eric Giguère in Sherbrooke.

The admin team relies on a governance body with five board members whose role is to resolve any serious conflicts and to give objective feedback on any large-scale decisions about QuTiP’s future. Finally, we note that two members of the current administration team are students who graduated from Google Summer of Code’s projects and decided to continue contributing to QuTiP voluntarily. Such students have been a vital part of the development of QuTiP v5, keeping QuTiP alive and flourishing.

5.2. RIKEN

As described in the introduction, QuTiP was originally developed in the group of Franco Nori in RIKEN. While the development team has since become global, RIKEN still serves as a focus for continued development by research scientists, technical staff, and intern students. Several recent important contributions of intern students include expansion of the HEOM method, the new QuTiP-QOC library, a general environment class, and an experimental tensor network data layer. In addition, members of the RIKEN team serve as mentors on GSOC projects and assist in the development and management of new features across QuTiP.

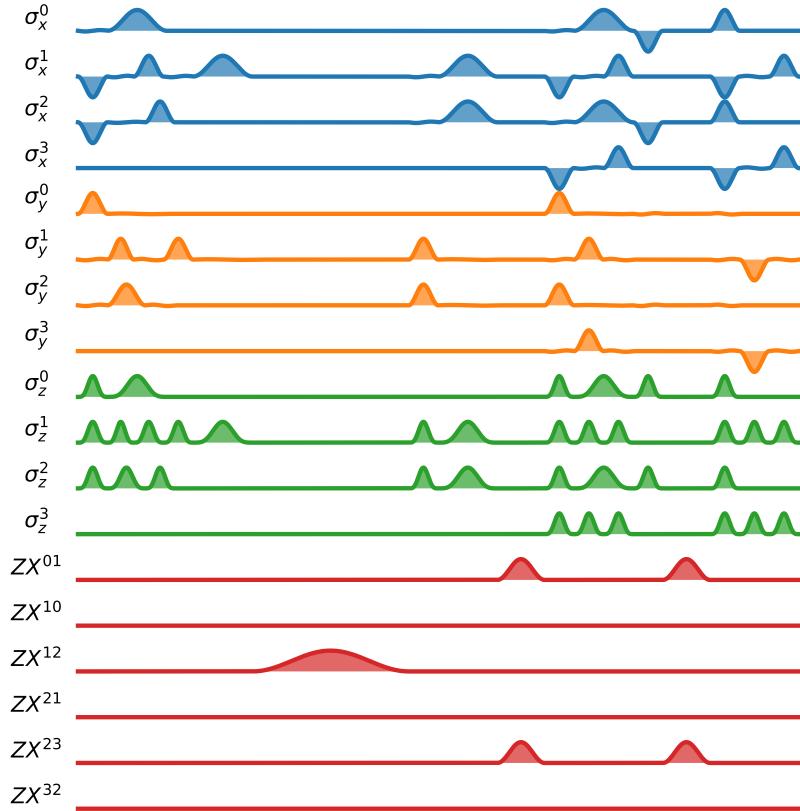


Figure 24: Pulse-level decomposition of the circuit in Fig. 23 as used by the spin-chain processor.

5.3. NumFOCUS and Google Summer of Code

QuTiP is affiliated to two non-profit organizations that provide support for development in various ways: NumFOCUS and the Unitary Foundation (the latter will be discussed in a later section). NumFOCUS has a long history of supporting open source projects, and primarily acts as an umbrella organization which enables QuTiP to apply to Google Summer of Code as a mentoring organization. It also supports development with periodic mini-grants. Their support for Google Summer of Code has been very useful, and enabled the QuTiP developers to mentor 2-3 students every year for the last few years.

5.4. The Unitary Foundation

QuTiP is also affiliated with the Unitary Foundation, which focuses on the support and development of open source software for quantum computing applications. The Unitary Foundation has supported QuTiP development through micro-grants and Hackathon bounties, both of which have helped nascent volunteers to contribute to QuTiP and substantially raised the visibility of QuTiP in this community. In addition, they operate a large Discord server through which people can interface with the QuTiP admin team.

5.5. Packages that use QuTiP

One of the most successful ingredients of QuTiP has been the flexibility and intuitive features of the `qutip.Qobj` class. Based on its use for the representation of quantum operators and quantum states, several other packages have formed a constellation of libraries. To date, 733 GitHub repositories and 83 released packages depend on QuTiP. A small selection of popular examples are:

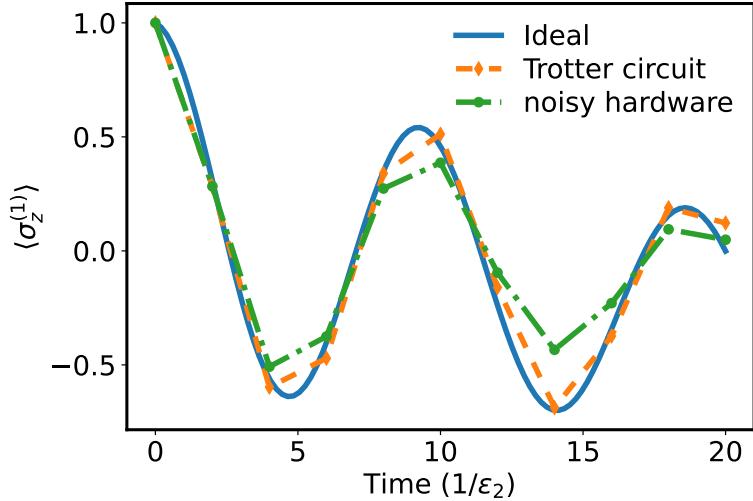


Figure 25: Results for the quantum simulation of open-system dynamics defined by the Hamiltonian (1). The Hamiltonian is the same as used in Fig. 21. The Lindblad noise, acting on both qubits with rate $\gamma = 0.03 \epsilon_2$, can be simulated by an additional interaction with two ancillas. The magnitude of this interaction corresponds to simulating a time step \sqrt{dt} for every full “coherent” time step dt (see [116, 117]), then tracing out the ancillas and resetting them to their ground states. The Trotter circuit (orange) reproduces quite well the results of `mesolve` (blue). The green curve, showing the circuit run on a superconducting qubits processor, exhibits significantly more error than in Fig. 21 because of the much longer circuit depth.

- Krotov [93]: As mentioned in the optimal control section, this QuTiP-affiliated library provides access to the Krotov method for optimizing control of quantum systems.
- SCqubits [118]: A powerful and popular library for computing the energy spectra of common superconducting qubit designs, that interfaces with QuTiP to perform time-evolution simulations.
- Sqcircuit [119]: An alternative to SCqubits that also models superconducting circuits, but has the added feature of using an efficient choice of basis to perform diagonalization of complex circuits.
- Dynamiqs [32]: An open systems solver deeply integrated with JAX. While lacking some of QuTiP’s features, it has a robust and focused integration with the JAX library, and is compatible with QuTiP `Qobj`.
- Qibo [120], a hardware focused middleware API that supports a QuTiP backend.

Several commercially developed packages use QuTiP as well, including:

- NVIDIA CUDA-Q [121], which, for example, uses it for Bloch sphere visualization.
- Qiskit [14], which also uses QuTiP’s Bloch-sphere visualization and its parallel map function.
- Pasqal’s Pulser package [122] which supports a QuTiP backend.

6. Future development

Historically, QuTiP has been mostly used as an academic tool for research and education. It allows quick implementation and experimentation with theoretical models of closed and open quantum systems, and has made research both faster and more reproducible for thousands of scientists around the world. With QuTiP v5, the focus has been on maintaining that utility while also allowing for future expansion in terms of state-of-the-art performance and use of high-performance distributed computing. Future development plans will continue in this regard.

In addition, recent developments in the field of quantum simulation software suggest another goal for QuTiP’s future direction. QuTiP, and in particular QuTiP-QIP, remain one of the few academically independent platforms for simulating quantum circuits and quantum computers in general. Other powerful platforms with comparable feature sets have arisen from industrial efforts. Nevertheless, there is still an important role for QuTiP with its promise to be continually maintained and developed, and fully open-source.

In this regard, there is a strong incentive to improve QuTiP’s support both for simulating quantum computing systems and for supporting cloud computing services as backends. In the future, QuTiP could serve as a system-agnostic and independent platform for development, with which one could access and use many hardware platforms (both industrial and academic). Initial efforts to this end came from backend support added by IonQ developers for their hardware, and we wish to continue in this direction by supporting as many hardware platforms as possible. On the question of academic independence, we note that some members of the administration team, present and alumni, have moved into industrial positions with companies like IBM, Zurich Instruments, and others. However, a robust board of mentors is in place to prevent bias.

In addition to these two goals, state-of-the-art performance and cloud-hardware support, we must also consider issues of maintainability. In that aspect, our strategy of QuTiP-family subpackages is working well, and we will continue with it in the future. However, there is an inevitability that some features that are not part of QuTiP-core may become abandoned or not well maintained, simply due to administrators or developers moving onto new tasks and fluctuations in academic funding support. Finding solutions to this ongoing problem, common to many open-source software packages, is one of the important future goals of the QuTiP administration team.

In addition to these broad strategic goals, in our recent workshop the following features were discussed for future releases:

6.1. *QuTiP’s role in the quantum computing revolution*

Hamiltonian library. — A library of common models (Hamiltonians, master equations, and so on) for common physical systems people encounter would be useful both for academics and educators. At first, we plan to include them within the QuTiP core as Python functions and then consider later how they might be exported in a common format for use in other libraries.

Simulating QPUs. — There are broad plans to extend the list of “processors”, or quantum processing units, available to QuTiP-QIP to include a larger range of physical systems [123–126]. This extension would tie in with the Hamiltonian library specified above, and the longer-term goal to support physical hardware QPUs through cloud API services.

QIP Tutorials. — In addition to predefined processors or QPUs, we plan to build a set of tutorials for QuTiP-QIP, showing explicitly how to create models of different types of quantum processors.

6.2. *QuTiP’s role in fundamental scientific research*

More and faster solvers. — We want to always support the state of the art in open quantum systems, identifying and incorporating methods which provide utility for a broad range of problems, and overall continually work to make all solvers faster and more efficient. New solvers currently being developed or considered are: methods to iteratively construct equations of motions of operators (sometimes called a “cumulant expansion”), and a non-Markovian master equation expansion based on time-convolutionless truncation of bath properties [51] (confusingly, sometimes also called a “cumulant expansion”). Additionally, currently under active development is a new Floquet master equation solver which features flexible levels of truncation of Floquet frequencies, a Dyson-expansion based solver for efficient simulation of quickly driven systems [127], and new Krylov-space solvers.

GPU support, high-performance computing, and qutip-cuquantum. — Multiprocessing in QuTiP currently takes three forms:

- Support for GPU through custom data layers like JAX and the in-development cuquantum data layer (see below),
- parallel use of multiple cores on a single CPU or node, instantiated either by explicitly using functions like `parallel_map()` or implicitly in natural multi-task problems like the stochastic and Monte Carlo solvers, and
- the use of multiple cores by underlying libraries and methods such as Intel-MKL, which can help speed up ODE solving and finding steady states.

Historically, we also supported OpenMP, but this support was removed with version 5 due to lack of use and heavy maintenance. Recently, support for MPI was added, extending point (ii) above to multiple nodes.

A very recent project in development (in collaboration with NVIDIA) is support for multiple GPUs with the `qutip-cuquantum` data layer [128], which takes advantage of the powerful cuquantum library from NVIDIA themselves [129]. By incorporating a symbolic representation of the problem being modeled, dynamics simulations using both `mesolve` and `sesolve` can be distributed across multiple GPUs, allowing for both speed-ups and larger scale simulations over that available via single-core CPU and single-GPU simulations. Expanding support for such symbolic representations across more parts of QuTiP is a planned core feature of our next release, QuTiP v6.

Tensor-network data layer. — Approximate or exact truncation methods (e.g., matrix product states (MPS), matrix product operators (MPOs), and tensor networks (TN)) that take advantage of the algebraic structure of quantum operators for specific problems (like one-dimensional spin chains) have proven very successfully in more efficiently analysing and simulating certain many-body problems (both closed and open) numerically. Motivated by this progress in the community, early in the development of version 5, we began exploring a tensor-network data layer, which exists currently in a very early alpha form. However, QuTiP has historically focused on providing solvers for arbitrary systems and its low-level data layer operates on two dimensional matrices. The data layer is therefore not the correct level for describing these algebraic structures. Work is underway to extend QuTiP with a symbolic operator description (akin to the early version being used in `qutip-cuquantum`), which will allow these richer structures to be captured and for solvers to use this to efficiently compute solutions for operators which have additional structure. Importantly, it will also enable easier integration with existing packages for these methods, as well as developing our own implementations. We expect this to be a useful extension not only for TN, MPS and MPOs but also for structuring problems for GPUs and HPCs where matching the problem description to the hardware provides significant performance benefits.

QuTiP beyond Python. — QuTiP is, and always will be, a Python orientated package. However, the popularity of QuTiP means that it has led to the creation of QuTiP-like packages in other languages. One of these, written in Julia, `QuantumToolbox.jl`, was recently incorporated into the QuTiP organization (and includes a recent Julia version of the HEOM solver [66]). Being developed by Alberto Mercurio, Yi-Te Huang, and others, it aims to offer syntax compatibility with QuTiP, and support the powerful distributed computing capabilities Julia is offering. In addition, an older and alternative Julia package inspired by QuTiP is `QuantumOptics.jl` [130], which complements the efforts being made in `QuantumToolbox.jl`.

7. Conclusion

QuTiP remains to be one of the most popular academically independent and fully open-source toolkits for simulating open quantum systems. Its wide adoption by industry, education, and research has also helped support its continued development, by attracting young and enthusiastic contributors. The release of v5 of QuTiP, particularly the substantial changes to the data layer, enables QuTiP to remain relevant

as the open-source quantum software community continues to grow. The examples we provided in this work, demonstrating unique applications of both old and new solvers and features of QuTiP, also serve an important purpose in exemplifying unique and new ways QuTiP can be used by the community.

8. Acknowledgements

Over the past five years, the development of QuTiP has greatly benefited from the generous support of the Japan Science and Technology Agency (JST) Moonshot R&D Grant Number JPMJMS2061, and we thank them for their kind support. We wish to thank a large range of past contributors to QuTiP, particularly Arne Grismo, Cassandra Granade, Michael H. Goerz, Anubhav Vardhan, Saumya Biswas, Sidhant Saraogi, Asad Raza, Felipe Bivort Haiek, Purva Thakre, Christian Staufenbiel, Xavier Spronken, Shreyas Pradhan, Trent Fridley, Yuji Tamakoshi and Alessia Parato. We also thank Mana Lambert for designing and contributing Fig. 1, Gavin Crowder for feedback on the waveguide example in section (3.3.1), the NVIDIA cuquantum team for support in the development of the qutip-cuquantum data layer, and Maximilian Meyer-Möllerlinghof for assistance in adapting the code examples into tutorial notebooks. F.N. is also 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 Office of Naval Research (ONR) Global (via Grant No. N62909-23-1-2074). N. L. is supported by the RIKEN Incentive Research Program and by MEXT KAKENHI Grant Numbers JP24H00816, JP24H00820. P. M. performed this work as an International Research Fellow of the Japan Society for the Promotion of Science (JSPS). M.G. is supported by the New Energy and Industrial Technology Development Organization (NEDO), project code JPNP16007. E.G. is supported by the Ministère de l’Économie, de l’Innovation et de l’Énergie du Québec. We also acknowledge the Information Systems Division, RIKEN, for the use of their facilities.

Appendix A. Tables

Here we provide tables summarizing various useful methods, functions, and concepts in QuTiP.

Method / Attribute	Description
<code>copy()</code>	Create a copy of the Qobj.
<code>conj()</code>	Complex conjugate.
<code>contract()</code>	Contract subspaces of the tensor structure that are 1D.
<code>cosm()</code>	Matrix cosine of the Qobj.
<code>dag()</code>	Adjoint (Hermitian conjugate) of the Qobj.
<code>data_as(format, copy)</code>	Retrieve the data in the desired format.
<code>diag()</code>	Diagonal elements of the Qobj.
<code>dnorm()</code>	Diamond norm of the Qobj.
<code>dual_chan()</code>	Obtain the dual channel of the Qobj.
<code>eigenenergies()</code>	Eigenvalues of the Qobj.
<code>eigenstates()</code>	Eigenvalues and eigenstates of the Qobj.
<code>groundstate()</code>	Ground state eigenvalue and eigenvector.
<code>expm()</code>	Matrix exponential of the Qobj.
<code>full()</code>	Dense array representation of the Qobj data.
<code>inv()</code>	Matrix inverse of the Qobj.
<code>logm()</code>	Matrix logarithm of the Qobj.
<code>matrix_element(bra, ket)</code>	Matrix element between the specified bra and ket vectors.
<code>norm()</code>	Norm of the Qobj.
<code>overlap(other)</code>	Overlap between two Qobjs.
<code>permute(order)</code>	Reorder the tensor structure of a composite Qobj.
<code>proj()</code>	Projector for a ket or bra vector.
<code>ptrace(sel)</code>	Partial trace over the specified subsystems.
<code>purity()</code>	Purity of the Qobj.
<code>sinm()</code>	Matrix sine of the Qobj.
<code>sqrtm()</code>	Matrix square root of the Qobj.
<code>tidyup(atol)</code>	Remove small elements (with tolerance atol) from the Qobj.
<code>tr()</code>	Trace of the Qobj.
<code>trans()</code>	Transpose of the Qobj.
<code>transform(input)</code>	Perform basis transformation defined by the input matrix.
<code>trunc_neg()</code>	Remove negative eigenvalues.
<code>unit()</code>	Normalizes the Qobj.
<code>data</code>	The QuTiP-internal data-layer object storing the data.
<code>dtype</code>	The data-layer type used for storing the data.
<code>dims, shape, type</code>	Basic information about the Qobj, explained in the main text.
<code>isherm, isunitary</code>	Indicates if the Qobj is a Hermitian / unitary operator.
<code>isket, isbra, isoper,</code>	Indicates if the Qobj has the respective type.
<code>issuper, isoperket, isoperbra</code>	
<code>iscp, ishp, istp, iscptp</code>	Indicates if the Qobj is a map that is completely positive (CP) / hermiticity-preserving (HP) / trace-preserving(TP) / CP and TP.

Table A.11: Methods of the `Qobj` Class in QuTiP. We have omitted optional parameters for some of the methods; they can be found in the full online reference together with more detailed explanations.

Function	Description
<code>lindblad_dissipator(a,b)</code>	Generates the Lindblad dissipator for the given Lindblad operators, $\mathcal{D}[a,b]\rho = a\rho b^\dagger - \frac{1}{2}a^\dagger b\rho - \frac{1}{2}\rho a^\dagger b$.
<code>liouvillian(H, c_ops)</code>	Generates the Liouvillian superoperator for a given Hamiltonian and list of collapse operators.
<code>spre(A)</code>	Generates the superoperator corresponding to left multiplication by the operator A .
<code>spost(A)</code>	Generates the superoperator corresponding to right multiplication by the operator A .
<code>sprepost(A,B)</code>	Generates the superoperator for left and right multiplication by operators A and B respectively.
<code>operator_to_vector(op)</code>	Vectorizes an operator.
<code>vector_to_operator(vec)</code>	Reshapes a vectorized operator back to its original matrix form.

Table A.12: Table of commonly used functions for creating and manipulating superoperators

Function	Description
<code>entropy_vn(rho, base)</code>	Computes the von Neumann entropy of a density matrix, defined as $S(\rho) = -\text{Tr}(\rho \log_b \rho),$ where b is the logarithmic base (default is e).
<code>entropy_linear(rho)</code>	Computes the linear entropy of a density matrix, given by $S_L(\rho) = 1 - \text{Tr}(\rho^2).$
<code>entropy_mutual(</code> <code>rho, selA, selB</code> <code>)</code>	Calculates the quantum mutual information for a bipartite density matrix ρ , given by $I(A : B) = S(\rho_A) + S(\rho_B) - S(\rho_{AB}),$ where ρ_A and ρ_B are the reduced density matrices. The parameters <code>selA</code> and <code>selB</code> specify the subspaces A and B .
<code>entropy_conditional(</code> <code>rho, selB</code> <code>)</code>	Computes the conditional entropy for a bipartite density matrix: $S(A B) = S(\rho_{AB}) - S(\rho_B),$ where ρ_{AB} is the bipartite density matrix, and ρ_B is the reduced density matrix of B . The parameter <code>selB</code> specifies the subspace B .
<code>entropy_relative(</code> <code>rho, sigma, base</code> <code>)</code>	Calculates the relative entropy between two density matrices ρ and σ , defined as $S(\rho \sigma) = \text{Tr}(\rho \log_b \rho) - \text{Tr}(\rho \log_b \sigma),$ where b is the logarithmic base (default is e). This quantifies the "distance" between the quantum states ρ and σ .
<code>concurrence(rho)</code>	Calculates the concurrence of a two-qubit density matrix ρ , an entanglement measure: $C(\rho) = \max \left(0, \sqrt{\lambda_1} - \sqrt{\lambda_2} - \sqrt{\lambda_3} - \sqrt{\lambda_4} \right),$ where λ_i are the eigenvalues of $R = \rho(\sigma_y \otimes \sigma_y)\rho^*(\sigma_y \otimes \sigma_y)$.
<code>negativity(rho, subsys)</code>	Computes the negativity of a bipartite quantum state ρ , defined as $\mathcal{N}(\rho) = \frac{\ \rho^{T_B}\ _1 - 1}{2},$ where ρ^{T_B} is the partial transpose and $\ \cdot\ _1$ the trace norm. The parameter <code>subsys</code> specifies for which subsystem to compute the negativity.

Table A.13: Summary of entropy and entanglement measure functions in QuTiP

Function	Description
<code>fidelity(rho, sigma)</code>	Computes the fidelity between two quantum states ρ and σ , defined as $F(\rho, \sigma) = \text{Tr} \left[\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}} \right].$ Fidelity measures how close two quantum states are to each other.
<code>tracedist(rho, sigma)</code>	Computes the trace distance between two density matrices ρ and σ : $D_{\text{tr}}(\rho, \sigma) = \frac{1}{2} \ \rho - \sigma\ _1,$ where $\ \cdot\ _1$ denotes the trace norm (sum of singular values).
<code>hilbert_dist(A, B)</code>	Computes the Hilbert-Schmidt distance between two operators A and B : $D_{\text{HS}}(A, B) = \sqrt{\text{Tr} [(A - B)^\dagger (A - B)]}.$ This measures the “distance” between operators in Hilbert space.
<code>bures_dist(rho, sigma)</code>	Calculates the Bures distance between two quantum states ρ and σ : $D_{\text{Bures}}(\rho, \sigma) = \sqrt{2(1 - F(\rho, \sigma))},$ where $F(\rho, \sigma)$ is the fidelity between ρ and σ .
<code>bures_angle(rho, sigma)</code>	Computes the Bures angle between two quantum states ρ and σ : $\theta_{\text{Bures}}(\rho, \sigma) = \arccos(F(\rho, \sigma)).$ This provides an angular measure of the distance between states.
<code>average_gate_fidelity(E, F)</code>	Computes the average gate fidelity between two quantum channels \mathcal{E} and \mathcal{F} , using the definition from [131].
<code>process_fidelity(E, F)</code>	Calculates the process fidelity between two quantum channels \mathcal{E} and \mathcal{F} , via their Choi matrices $\chi_{\mathcal{E}}$ and $\chi_{\mathcal{F}}$. Since QuTiP v5, this function uses the definition from [131].
<code>dnorm(E, F)</code>	Computes the diamond norm distance between two channels \mathcal{E} and \mathcal{F} : $\ \mathcal{E} - \mathcal{F}\ _\diamond = \sup_{\rho} \ (\mathcal{E} \otimes \mathbb{I})(\rho) - (\mathcal{F} \otimes \mathbb{I})(\rho)\ _1,$ where the supremum is over all density matrices ρ on an extended Hilbert space, and \mathbb{I} is the identity map.

Table A.14: Summary of commonly used metric functions in QuTiP

Appendix B. Summary of tutorials and example notebooks

QuTiP has historically provided a large array of tutorials and example notebooks [4, 132]. With v5, initiated through the GSoC project of Christian Staufenbiel, these notebooks are going through an overhaul, to unify and modernize them, make them more compact with a switch to markdown, and include rudimentary tests of their functionality (which also acts as an additional check of QuTiP itself).

Here we provide a detailed summary of some of the updated notebooks for v5, including an abridged version of their content which is useful and complementary to the main text.

Appendix B.1. Time-evolution tutorials

This set of tutorials can be seen as the core demonstration of QuTiP's functionality and features [4].

Appendix B.1.1. 0001_qobjevo

In this tutorial we demonstrate the flexibility and power of the `Qobjevo` class for representing time-dependent objects. This has largely been covered earlier in this work, but it is interesting to show some examples from that notebook. To recap, when solvers in QuTiP are provided with a time dependent operator, typically in terms of a tuple containing the quantum object and a representation of the time-dependence as a function, string, or array, internally this is converted into `Qobjevo`, which compiles and optimizes the representation.

These objects can also be created and compiled manually with lists of objects and time-dependent coefficients $[A_0, [A_1, f_1], [A_2, f_2], \dots]$ where A_k are quantum objects and f_k either functions, strings, or arrays. Another flexible option is to use the `coefficient` function to wrap the time-dependent function, string, or array, and then multiply it with a `Qobj` as needed.

```
def cos_t(t):
    return np.cos(t)
function_form = (n + (a + ad) * qutip.coefficient(cos_t))
```

String format representations should represent a valid Python function or expression that returns a complex number. So for the above example, instead one could call `string_form = QuTiP.QobjEvo([n, [a + ad, "cos(t)"]])`. Any NumPy or `scipy.special` functions can be referred to with `np` or `spe`.

Finally, if one is using data to represent the time dependence of a given object one may pass an array of data points, alongside a list of time steps at which those points are specified. Intermediate times will be interpolated with cubic splines.

```
tlist = np.linspace(0, 10, 101)
values = np.cos(tlist)

array_form = (n + (a + ad) * qutip.coefficient(values, tlist=tlist))
```

Once defined, `Qobjevo` functions can be called to return the value of the `Qobj` at that time, as well as manipulated with the same mathematical rules as a `Qobj` and mixed with scalars, `Qobj` and other `QobjEvo` objects.

Even more powerfully, `QobjEvo` functions can take arguments that can be quite complex, including objects or values that are derived from the solver they are used in. This allows one to directly implement both state and expectation-value based feedback. More details are provided in the tutorial, but, for example, an argument that will capture the state of the system during the evolution with `mesolve` can be instantiated with `StateFeedback()`, and that state can then be used to inform the behavior of a function that is used in `mesolve` itself:

```
args = {"state": qutip.MESolver.StateFeedback(default=qutip.fock_dm(4, 2))}
def print_args(t, state):
    print(f"state:{\n{state}\n}")
    return t + state.norm()
td_args = qutip.QobjEvo([Id, print_args], args=args)
```

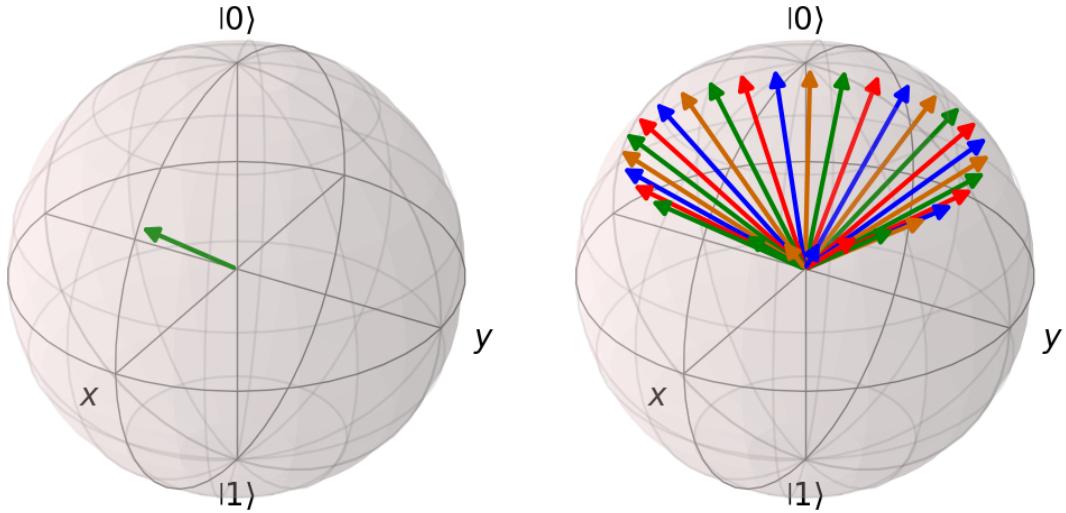


Figure B.26: Left shows the Bloch sphere representation of the state `psi = (2.0 * basis(2, 0) + basis(2, 1)).unit()`, while right shows the state at various times while precessing under a constant magnetic field.

Appendix B.1.2. 0002_larmor-precession

This tutorial demonstrates basic usage of the `sesolve()` function with an example of Larmor precession. It demonstrates how to use in-built plotting functions to show the state of a spin on a Bloch sphere Fig. B.26. It then demonstrates how to solve the dynamics of such a spin in the presence of a magnetic field in the Z direction, and obtain expectation value of observables, with `sesolve`, and plot the state on the Bloch sphere.

The tutorial then demonstrates how to use `Qobjevo` to represent time-dependent magnetic fields, and use those with `sesolve`.

Appendix B.1.3. 0003_qubit-dynamics

This tutorial shows how to model the dynamics of a qubit under the influence of noise using `mesolve`. Like the previous example, it also demonstrates how to visualize these dynamics with a Bloch sphere. This is shown in Fig. 14 where unitary, dephasing, and relaxation evolution are shown. For dephasing, the dynamics moves towards the center of the Bloch sphere, while for relaxation, it moves to a point within the volume.

Appendix B.1.4. 0004_rabi-oscillations

This tutorial shows how to simulate the dynamics of the Rabi model, one of the most fundamental models of quantum optics, which describes a single two-level atom interacting with a single cavity mode of light. The Hamiltonian for this system is,

$$H = \hbar\omega_c a^\dagger a + \frac{1}{2}\hbar\omega_a \sigma_z + \hbar g(a^\dagger + a)(\sigma_- + \sigma_+) \quad (\text{B.1})$$

which can be simplified, for weak coupling, with the rotating-wave approximation:

$$H_{\text{RWA}} = \hbar\omega_c a^\dagger a + \frac{1}{2}\hbar\omega_a \sigma_z + \hbar g(a^\dagger \sigma_- + a \sigma_+), \quad (\text{B.2})$$

here ω_c and ω_a are the frequencies of the cavity and atom, while g is the interaction strength between light and matter. This tutorial shows how to solve the dynamics of this model using `mesolve` when both the light and matter are in contact with a noisy environment. The steps needed to do so are similar to other examples

we have already described in this paper, so we will not linger on this example, but just mention that the example in this tutorial is only valid for weak-intermediate light-matter coupling, and in the ultra-strong light matter regime, when g approaches ω_c and ω_a , one should instead employ `brmesolve`, as described earlier for the interacting qubit example. This is explained in more detail in tutorial “0009_brmesolve_cavity-QED”.

Appendix B.1.5. 0005_spin-chain

This tutorial explains how to solve the dynamics of a Heisenberg spin-chain using `mesolve`. In construction it is similar to the example for the 1D Ising model we used as a bench-mark in the main text, but the Hamiltonian for the spin chain instead takes the form,

$$H = -\frac{1}{2} \sum_n^N h_n \sigma_z(n) - \frac{1}{2} \sum_n^{N-1} [J_x^{(n)} \sigma_x(n) \sigma_x(n+1) + J_y^{(n)} \sigma_y(n) \sigma_y(n+1) + J_z^{(n)} \sigma_z(n) \sigma_z(n+1)], \quad (\text{B.3})$$

where h_n is the magnetic field felt by spin n while $J_k^{(n)}$ describes the nearest neighbour interaction strength in direction k for spin n . The example also includes dephasing on these spins, and demonstrates how to visualize dynamics of such a complex system.

Appendix B.1.6. 0006_photon_birth_death

This tutorial is an example of how to use `mcsolve`, the Monte-Carlo solver, to simulate experimental results that appeared in [133]. The tutorial demonstrates how to reproduce figure 3 from this work, by simulating the creation and annihilation of photons inside an cavity, due to a thermal environment, when the cavity is initially prepared in the single-photon Fock state.

The model is very simple, consisting of the Hamiltonian $H = a^\dagger a$, the aforementioned initial Fock state, and collapse operators describing the creation/annihilation of photons are appropriately weighted rates: $C_1 = \sqrt{\kappa(1 + \langle n \rangle)} a$ and $C_2 = \sqrt{\kappa \langle n \rangle} a^\dagger$, where κ is the bare rate and n is the thermal Bose-Einstein factor which depends on the temperature and the cavity frequency. The tutorial then demonstrates how to solve this example using `mcsolve` and look at the dynamics for different numbers of trajectories.

Appendix B.1.7. 0007_brmesolve_tls

This tutorial demonstrates the basic usage of the Bloch-Redfield solver `brmesolve` for a single two-level system (TLS). This covers the same ground as our earlier explanation, but does mention one important feature we did not cover in the main paper; the calculation of steady states. The function `R, ekets = bloch_redfield_tensor(H, [a_op])` returns the Bloch-Redfield tensor, i.e., the right-hand-side of the equation of motion in superoperator form. This can then be used with the `steadystate(R)` function to find directly the steady-state solution of the Bloch-Redfield equation one is simulating.

Appendix B.1.8. 0008_brmesolve_time_dependence

This tutorial explains how to use both time-dependent Hamiltonians and time-dependent dissipation with the Bloch-Redfield solver. The functionality is equivalent to using time-dependence in other solvers. One important point this tutorial covers is that with the Bloch-Redfield solver, the coupling to the environment must be a Hermitian operator. Complex time-dependence can be included by splitting the operator into two parts, so that something like

$$A = f(t)a + f(t)^*a^\dagger \quad (\text{B.4})$$

can be implemented with `a_ops = [[[a, 'exp(1j*t)'], [a.dag(), 'exp(-1j*t)']], f'kappa * (w >= 0)]]`, under the restriction that the second function is the complex conjugate of the first one, and the second operator is the Hermitian conjugate of the first operator.

Appendix B.1.9. 0009_brmesolve_cavity-QED

In this tutorial, we present another example of when the Bloch-Redfield solver can be useful for constructing a more physically accurate master equation than relying on local dissipation alone. The logic is similar to that in the main text of this work for the two-interacting qubit example, but instead, in this tutorial, concentrates on the Rabi model from quantum optics (described already in 0004_rabi-oscillations).

The tutorial demonstrates that for weak coupling between light and matter, including cavity dissipation using `brmesolve`, or including local dissipation on the cavity alone via `mesolve`, produce similar results. However, for strong coupling, the local Lindblad model can fail to produce physical results (in essence, we can see continuous emission out of the cavity without any input; a perpetuum mobile).

Appendix B.1.10. 0010_brmesolve_phonon_interaction

This tutorial, contributed by K. A. Fischer from Stanford University, is a complex example of how to use `brmesolve` to simulate the phonon-assisted initialization of a quantum dot, reproducing the results of an existing article [134].

Appendix B.1.11. 0011_floquet_solver

This tutorial describes the basic use of the Floquet Schrodinger and Master equation solvers, with an example akin to the description we used in the main text.

Appendix B.1.12. 0012_floquet_formalism

This tutorial describes some of the underpinnings of the Schrodinger and Master equation Floquet solvers in terms of the `FloquetBasis` and quasi-energies. It is a very useful starting point for users who wish to use these objects for more niche applications.

Appendix B.1.13. 0013_nonmarkovian_monte_carlo

This tutorial expands upon the use of the `nm_mcsolve()` method for unravelling, in terms of Monte-Carlo trajectories, a master equation with non-Markovian rates (i.e., rates which are time-dependent and sometimes negative).

As explained in the section on this solver, this is quite a nuanced method, and this tutorial provides several practical examples to help users get accustomed to it: (1) a two-level atom in a photonic band-gap and (2) a two-qubit Redfield equation derived from when two qubits interact with a common bath. It also describes how to employ the MPI feature in qutip to solve these trajectories in parallel on a super-computing cluster.

Appendix B.1.14. 0015_smesolve-heterodyne

This example shows how to use `smesolve` to simulate heterodyne measurement of a cavity. In the earlier section on `smesolve` a similar example for homodyne measurement was given. The key difference here is the measurement gives information on both quadratures of the system being measured. The tutorial presents two approaches to defining and solving this type of problem.

Appendix B.1.15. 0016_smesolve-inefficient-detection

This example demonstrates further utility of `smesolve` for modelling an important example from Wiseman and Milburn, Quantum measurement and control, section. 4.8.1. In this example, a lossy cavity is monitored with an inefficient photodetector. This means the system undergoes evolution with two noise channels, one where photons are successfully detected, and one where they are not. This is easily formulated as a Monte-Carlo master equation for use with `mcsolve`. Furthermore, if this inefficient detection is used as part of a homodyne detection protocol, this can also be formulated as a stochastic master equation for use with `smesolve`.

Appendix B.1.16. 0016_smesolve-jc-photocurrent

This example generalizes the previous so that the system being monitored includes an atom interacting

with a cavity, as described by the Jaynes-Cummings model. The output of the cavity is imperfectly detecting with an inefficient photodetector, which is again simulated with `mcsolve`.

Appendix B.1.17. 0018_measures-trajectories-cats-kerr

In this example, contributed by Fabrizio Minganti, reproduces published results in [135–137]. It is an extremely detailed example of modelling a nonlinear Kerr resonator undergoing parametric two-photon driving. It demonstrates that how the cavity is measured, either via photon counting (with `mcsolve`) or homodyne detection (with `sme solve`), effects ones ability to understand the nature of the cavity state.

Appendix B.1.18. 0019_optomechanical-steadystate

This tutorial serves to demonstrate the varying steady-state solvers in QuTiP using an archtypical example from optomechanics. As well as showing the physics of optomechanics systems, and how the steady-state of an oscillator can be inspected for unphysical results, it demonstrates how the different methods and solvers in `steadystate()` can be invoked.

Appendix B.1.19. 0020_homodyned-Jaynes-Cummings-emission

This example, contributed by K. A. Fisher and A. V. Domingues (reproducing results from [138]), demonstrates how to obtain photonic correlation functions of a Jaynes-Cummings systems, with the purpose of probing the lowest-lying states as an effective two-level system. It also shows how to obtain second-order correlation functions from `mesolve()` manually (instead of using utility functions).

Appendix B.1.20. 0021_quasi-steadystate-driven-system

This notebook demonstrates the different ways steady-state information of periodically driven dissipative quantum systems can be calculated. In particular, it demonstrates how properties of the periodically oscillating steady-state, averaged over one period, can be obtained with either `propagator_steadystate()` or `steadystate_floquet()`.

Appendix B.2. Lectures

The set of tutorials are a series of Lectures from Robert Johansson detailing use of QuTiP, from basics to complex applications. These were originally written for a taught invited course at Chalmers University in Sweden for an early version of QuTiP. They have been updated and fixed to function with both QuTiP v4 and v5, and have proven to be a valuable for students and researchers everywhere.

Appendix B.3. Quantum circuits and Pulse-level-circuit-simulation

These two sets of tutorials detail use of QuTiP-QIP, from basic circuit simulations to complex pulse-level noise models. They include basic examples, like a simple circuit implementing CNOT and Toffoli gates and a tutorial on how to export and import `openqasm` circuits, and more complex examples of quantum algorithms like the Quantum Fourier transform and the Deustch-Josza algorithm.

Appendix B.4. Visualization

QuTiP includes many visualization tools for common ways to present a quantum state, like the Bloch sphere and the Wigner Function. This set of tutorials demonstrates much of this functionality, alongside demonstrations of built-in animation functions, process tomography functions, and more.

Appendix B.5. HEOM: Hierarchical Equations of Motion

The HEOM tutorials demonstrate how to use QuTiP's hierarchical equations of motion solver for both fermionic and bosonic baths. These examples have been explained in detail in our recent publication [22].

Appendix B.6. Miscellaneous

This repository is a place to include demonstrations of new QuTiP functionality, like the JAX backend, and more complex physical examples.

Appendix B.7. QuTiP-notebooks

This older repository [132] contains many more examples and tutorials that have not yet been ported to v5. However, they remain an important resource, and overtime we hope to have all relevant notebooks updated and included in the official qutip-tutorials repository.

References

- [1] J. R. Johansson, P. D. Nation, and F. Nori, QuTiP: An open-source Python framework for the dynamics of open quantum systems, *Comput. Phys. Commun.* **183**, 1760 (2012).
- [2] J. R. Johansson, P. D. Nation, and F. Nori, QuTiP 2: A Python framework for the dynamics of open quantum systems, *Comput. Phys. Commun.* **184**, 1234 (2013).
- [3] <https://github.com/qutip/qutip-paper-v5-examples>.
- [4] <https://github.com/qutip/qutip-tutorials>.
- [5] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, Array programming with NumPy, *Nature* **585**, 357 (2020).
- [6] C. W. Groth, M. Wimmer, A. R. Akhmerov, and X. Waintal, Kwant: a software package for quantum transport, *New Journal of Physics* **16**, 063065 (2014).
- [7] P. Giannozzi *et al.*, QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials, *Journal of Physics: Condensed Matter* **21**, 395502 (2009).
- [8] R. T. McGibbon, K. A. Beauchamp, M. P. Harrigan, C. Klein, J. M. Swails, C. X. Hernández, C. R. Schwantes, L.-P. Wang, T. J. Lane, and V. S. Pande, MDTraj: A modern open library for the analysis of molecular dynamics trajectories, *Biophysical Journal* **109**, 1528 (2015).
- [9] The Astropy Collaboration *et al.*, The Astropy Project: Sustaining and growing a community-oriented open-source project and the latest major release (v5.0) of the core package, *The Astrophysical Journal* **935**, 167 (2022).
- [10] W. Zeng, B. Johnson, R. Smith, N. Rubin, M. Reagor, C. Ryan, and C. Rigetti, First quantum computers need smart software, *Nature News* **549**, 149 (2017).
- [11] J. Dargan, *The quantum insider: Top 35 open source quantum computing tools* (2024).
- [12] D. S. Steiger, T. Häner, and M. Troyer, ProjectQ: an open source software framework for quantum computing, *Quantum* **2**, 49 (2018).
- [13] J. R. McClean *et al.*, OpenFermion: the electronic structure package for quantum computers, *Quantum Science and Technology* **5**, 034014 (2020).
- [14] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, and J. M. Gambetta, Quantum computing with Qiskit, [arXiv:2405.08810 \[quant-ph\]](https://arxiv.org/abs/2405.08810) (2024).
- [15] S. M. Tan, *The quantum optics toolbox for MATLAB* (2002).
- [16] H.-P. Breuer and F. Petruccione, *The theory of open quantum systems* (Oxford University Press, 2002).
- [17] D. A. Lidar, Lecture Notes on the Theory of Open Quantum Systems, [arXiv:1902.00967 \[quant-ph\]](https://arxiv.org/abs/1902.00967) (2019).

- [18] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, *et al.*, SciPy 1.0: fundamental algorithms for scientific computing in Python, *Nat. Methods* **17**, 261 (2020).
- [19] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, Cython: The best of both worlds, *Computing in Science & Engineering* **13**, 31 (2011).
- [20] J. D. Hunter, Matplotlib: A 2D graphics environment, *Comput. Sci. Eng.* **9**, 90 (2007).
- [21] B. Li, S. Ahmed, S. Saraogi, N. Lambert, F. Nori, A. Pitchford, and N. Shammah, Pulse-level noisy quantum circuits with qutip, *Quantum* **6**, 630 (2022).
- [22] N. Lambert, T. Raheja, S. Cross, P. Menczel, S. Ahmed, A. Pitchford, D. Burgarth, and F. 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** (2023).
- [23] N. Shammah, S. Ahmed, N. Lambert, S. De Liberato, and F. Nori, Open quantum systems with local and collective incoherent processes: Efficient numerical simulations using permutational invariance, *Phys. Rev. A* **98**, 063815 (2018).
- [24] Google Summer of Code, <https://summerofcode.withgoogle.com/>.
- [25] <https://github.com/qgrad/qgrad>.
- [26] <https://github.com/qutip/qutip-jax>.
- [27] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, **JAX: composable transformations of Python+NumPy programs** (2018).
- [28] <https://github.com/qutip/qutip-cupy>.
- [29] <https://github.com/qutip/qutip-tensorflow>.
- [30] <https://github.com/qutip/qutip-tensornetwork>.
- [31] <https://data-apis.org/array-api/latest>.
- [32] P. Guilmartin, R. Gautier, A. Bocquet, and É. Genois, Dynamiqs: an open-source Python library for GPU-accelerated and differentiable simulation of quantum systems (2024).
- [33] X.-Z. Luo, J.-G. Liu, P. Zhang, and L. Wang, Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design, *Quantum* **4**, 341 (2020).
- [34] P. D. Nation, Steady-state solution methods for open quantum optical systems, [arXiv:1504.06768 \[quant-ph\]](https://arxiv.org/abs/1504.06768) (2015).
- [35] J. Dalibard, Y. Castin, and K. Mølmer, Wave-function approach to dissipative processes in quantum optics, *Phys. Rev. Lett.* **68**, 580 (1992).
- [36] C. Gneiting, A. Koottandavida, A. V. Rozhkov, and F. Nori, Unraveling the topology of dissipative quantum systems, *Phys. Rev. Res.* **4**, 023036 (2022).
- [37] C. Gneiting, A. V. Rozhkov, and F. Nori, Jump-time unraveling of Markovian open quantum systems, *Phys. Rev. A* **104**, 062212 (2021).
- [38] R. Dum, P. Zoller, and H. Ritsch, Monte carlo simulation of the atomic master equation for spontaneous emission, *Phys. Rev. A* **45**, 4879 (1992).

- [39] K. Mølmer, Y. Castin, and J. Dalibard, Monte carlo wave-function method in quantum optics, *J. Opt. Soc. Am. B* **10**, 524 (1993).
- [40] M. Radaelli, G. T. Landi, and F. C. Binder, Gillespie algorithm for quantum jump trajectories, *Phys. Rev. A* **110**, 062212 (2024).
- [41] H.-P. Breuer, Genuine quantum trajectories for non-Markovian processes, *Phys. Rev. A* **70**, 012106 (2004).
- [42] J. Piilo, S. Maniscalco, K. Häkkinen, and K.-A. Suominen, Non-Markovian Quantum Jumps, *Phys. Rev. Lett.* **100**, 180402 (2008).
- [43] B. I. C. Donvil and P. Muratore-Ginanneschi, On the Unraveling of Open Quantum Dynamics, *Open Syst. Inf. Dyn.* (2023).
- [44] H.-P. Breuer, B. Kappler, and F. Petruccione, Stochastic wave-function method for non-Markovian quantum master equations, *Phys. Rev. A* **59**, 1633 (1999).
- [45] M. R. Hush, I. Lesanovsky, and J. P. Garrahan, Generic map from non-Lindblad to Lindblad master equations, *Phys. Rev. A* **91**, 032113 (2015).
- [46] P. Menczel, K. Funo, M. Cirio, N. Lambert, and F. Nori, Non-hermitian pseudomodes for strongly coupled open quantum systems: Unravelings, correlations, and thermodynamics, *Phys. Rev. Res.* **6**, 033237 (2024).
- [47] B. Donvil and P. Muratore-Ginanneschi, Quantum trajectory framework for general time-local master equations, *Nat Commun* **13**, 4140 (2022).
- [48] B. Donvil and P. Muratore-Ginanneschi, Unraveling-paired dynamical maps recover the input of quantum channels, *New J. Phys.* **25**, 053031 (2023).
- [49] D. Davidović, Completely Positive, Simple, and Possibly Highly Accurate Approximation of the Redfield Equation, *Quantum* **4**, 326 (2020).
- [50] C. Gneiting, Disorder-dressed quantum evolution, *Phys. Rev. B* **101**, 214203 (2020).
- [51] P. Groszkowski, A. Seif, J. Koch, and A. A. Clerk, Simple master equations for describing driven systems subject to classical non-Markovian noise, *Quantum* **7**, 972 (2023).
- [52] G. Floquet, Sur les équations différentielles linéaires à coefficients périodiques, *Annales scientifiques de l'École Normale Supérieure 2e série*, **12**, 47 (1883).
- [53] J. H. Shirley, Solution of the Schrödinger equation with a Hamiltonian periodic in time, *Phys. Rev.* **138**, B979 (1965).
- [54] M. Grifoni and P. Hänggi, Driven quantum tunneling, *Physics Reports* **304**, 229 (1998).
- [55] C. Creffield, Location of crossings in the Floquet spectrum of a driven two-level system, *Phys. Rev. B* **67**, 165301 (2003).
- [56] F. Clawson and E. B. Flagg, Floquet-Lindblad master equation approach to open quantum system dynamics, [arXiv:2410.18046 \[quant-ph\]](https://arxiv.org/abs/2410.18046) (2024).
- [57] Y. Tanimura, Numerically “exact” approach to open quantum dynamics: The hierarchical equations of motion (HEOM), *The Journal of Chemical Physics* **153**, 020901 (2020).
- [58] Y. Tanimura and R. 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 (1989).

- [59] M. Cygorek and E. M. Gauger, ACE: A general-purpose non-Markovian open quantum systems simulation toolkit based on process tensors, *The Journal of Chemical Physics* **161**, 074111 (2024).
- [60] M. Cirio, N. Lambert, P. Liang, P.-C. Kuo, Y.-N. Chen, P. Menczel, K. Funo, and F. Nori, Pseudofermion method for the exact description of fermionic environments: From single-molecule electronics to the Kondo resonance, *Phys. Rev. Res.* **5**, 033011 (2023).
- [61] P.-C. Kuo, N. Lambert, M. Cirio, Y.-T. Huang, F. Nori, and Y.-N. Chen, Kondo QED: The Kondo effect and photon trapping in a two-impurity Anderson model ultrastrongly coupled to light, *Phys. Rev. Res.* **5**, 043177 (2023).
- [62] H. Rahman and U. Kleinekathöfer, Chebyshev hierarchical equations of motion for systems with arbitrary spectral densities and temperatures, *The Journal of Chemical Physics* **150**, 244104 (2019).
- [63] A. Ishizaki and G. R. Fleming, Theoretical examination of quantum coherence in a photosynthetic system at physiological temperature, *PNAS* **106**, 17255 (2009).
- [64] R. Härtle, G. Cohen, D. R. Reichman, and A. J. Millis, Decoherence and lead-induced interdot coupling in nonequilibrium electron transport through interacting quantum dots: A hierarchical quantum master equation approach, *Phys. Rev. B* **88**, 235426 (2013).
- [65] S. Wenderoth, J. Bätge, and R. Härtle, Sharp peaks in the conductance of a double quantum dot and a quantum-dot spin valve at high temperatures: A hierarchical quantum master equation approach, *Phys. Rev. B* **94**, 121303 (2016).
- [66] Y.-T. Huang, P.-C. Kuo, N. Lambert, M. Cirio, S. Cross, S.-L. Yang, F. Nori, and Y.-N. Chen, An efficient Julia framework for hierarchical equations of motion in open quantum systems, *Communications Physics* **6**, 313 (2023).
- [67] N. Lambert, S. Ahmed, M. Cirio, and F. Nori, Modelling the ultra-strongly coupled spin-boson model with unphysical modes, *Nat. Commun.* **10**, 3721 (2019).
- [68] M. Xu, Y. Yan, Q. Shi, J. Ankerhold, and J. T. Stockburger, Taming Quantum Noise for Efficient Low Temperature Simulations of Open Quantum Systems, *Phys. Rev. Lett.* **129**, 230601 (2022).
- [69] S. Wenderoth, H.-P. Breuer, and M. Thoss, Non-markovian effects in the spin-boson model at zero temperature, *Physical Review A* **104**, 012213 (2021).
- [70] Y. Zhou and J. Shao, Solving the spin-boson model of strong dissipation with flexible random-deterministic scheme, *The Journal of Chemical Physics* **128**, 034106 (2008).
- [71] H. Wang and M. Thoss, From coherent motion to localization: dynamics of the spin-boson model at zero temperature, *New Journal of Physics* **10**, 115005 (2008).
- [72] H. Takahashi, S. Rudge, C. Kaspar, M. Thoss, and R. Borrelli, High accuracy exponential decomposition of bath correlation functions for arbitrary and structured spectral densities: Emerging methodologies and new approaches, *The Journal of Chemical Physics* **160**, 204105 (2024).
- [73] G. Suárez and M. Horodecki, Making non-markovian master equations accessible with approximate environments, [arXiv:2506.22346 \[quant-ph\]](https://arxiv.org/abs/2506.22346) (2025).
- [74] <https://github.com/qutip/qutip-tutorials/blob/main/tutorials-v5/visualization/animation-demo.md>.
- [75] <https://github.com/qutip/qutip-tutorials/blob/main/tutorials-v5/miscellaneous/excitation-number-restricted-states-jc-chain.md>.
- [76] M. Lednev, F. J. García-Vidal, and J. Feist, Lindblad master equation capable of describing hybrid quantum systems in the ultrastrong coupling regime, *Phys. Rev. Lett.* **132**, 106902 (2024).

- [77] J. Román-Roche, E. Sánchez-Burillo, and D. Zueco, Bound states in ultrastrong waveguide QED, *Phys. Rev. A* **102**, 023702 (2020).
- [78] J. Zhang, Y.-x. Liu, R.-B. Wu, K. Jacobs, and F. Nori, Quantum feedback: Theory, experiments, and applications, *Physics Reports* **679**, 1–60 (2017).
- [79] A. L. Grimsmo, Time-delayed quantum feedback control, *Phys. Rev. Lett.* **115**, 060402 (2015).
- [80] S. Arranz Regidor, G. Crowder, H. Carmichael, and S. Hughes, Modeling quantum light-matter interactions in waveguide QED with retardation, nonlinear interactions, and a time-delayed feedback: Matrix product states versus a space-discretized waveguide model, *Phys. Rev. Res.* **3**, 023030 (2021).
- [81] G. Crowder, L. Ramunno, and S. Hughes, Quantum trajectory theory and simulations of nonlinear spectra and multiphoton effects in waveguide-QED systems with a time-delayed coherent feedback, *Phys. Rev. A* **106**, 013714 (2022).
- [82] L. Dalcín, R. Paz, and M. Storti, MPI for Python, *J. Parallel Distrib. Comput.* **65**, 1108 (2005).
- [83] L. Dalcín, R. Paz, M. Storti, and J. D'Elía, MPI for Python: Performance improvements and MPI-2 extensions, *J. Parallel Distrib. Comput.* **68**, 655 (2008).
- [84] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, Parallel distributed computing using Python, *Adv. Water Resour. New Computational Methods and Software Tools*, **34**, 1124 (2011).
- [85] L. Dalcin and Y.-L. L. Fang, Mpi4py: Status Update After 12 Years of Development, *Comput. Sci. Eng.* **23**, 47 (2021).
- [86] T. Caneva, T. Calarco, and S. Montangero, Chopped random-basis quantum optimization, *Phys. Rev. A* **84**, 022326 (2011).
- [87] N. Khaneja, T. Reiss, C. Kehlet, T. Schulte-Herbrüggen, and S. J. Glaser, Optimal control of coupled spin dynamics: design of NMR pulse sequences by gradient ascent algorithms, *Journal of Magnetic Resonance* **172**, 296 (2005).
- [88] <https://github.com/qutip/qutip-qtrl>.
- [89] <https://github.com/qutip/qutip-qoc>.
- [90] S. Machnes, E. Assémat, D. Tannor, and F. K. Wilhelm, Tunable, flexible, and efficient optimization of control pulses for practical qubits, *Phys. Rev. Lett.* **120**, 150401 (2018).
- [91] M. H. Goerz, S. C. Carrasco, and V. S. Malinovsky, Quantum optimal control via semi-automatic differentiation, *Quantum* **6**, 871 (2022).
- [92] Z. Zong, Z. Sun, Z. Dong, C. Run, L. Xiang, Z. Zhan, Q. Wang, Y. Fei, Y. Wu, W. Jin, C. Xiao, Z. Jia, P. Duan, J. Wu, Y. Yin, and G. Guo, Optimization of a controlled- z gate with data-driven gradient-ascent pulse engineering in a superconducting-qubit system, *Phys. Rev. Appl.* **15**, 064005 (2021).
- [93] M. H. Goerz, D. Basilewitsch, F. Gago-Encinas, M. G. Krauss, K. P. Horn, D. M. Reich, and C. P. Koch, Krotov: A Python implementation of Krotov's method for quantum optimal control, *SciPost Phys.* **7**, 80 (2019).
- [94] T. Araki, F. Nori, and C. Gneiting, Robust quantum control with disorder-dressed evolution, *Phys. Rev. A* **107**, 032609 (2023).
- [95] M. M. Müller, R. S. Said, F. Jelezko, T. Calarco, and S. Montangero, One decade of quantum optimal control in the chopped random basis, *Reports on Progress in Physics* **85**, 076001 (2022).

- [96] H. A. Corti, L. Banchi, and A. Cidronali, Robustness of a universal gate set implementation in transmon systems via Chopped Random Basis optimal control, *Physics Letters A* **438**, 128119 (2022).
- [97] B. Riaz, C. Shuang, and S. Qamar, Optimal control methods for quantum gate preparation: a comparative study, *Quantum Information Processing* **18**, 100 (2019).
- [98] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, Automatic differentiation in machine learning: a survey, *Journal of Machine Learning Research* **18**, 1 (2018).
- [99] Y. Song, J. Li, Y.-J. Hai, Q. Guo, and X.-H. Deng, Optimizing quantum control pulses with complex constraints and few variables through autodifferentiation, *Phys. Rev. A* **105**, 012616 (2022).
- [100] <https://github.com/qutip/qutip-qip>.
- [101] I. Buluta and F. Nori, Quantum simulators, *Science* **326**, 108–111 (2009).
- [102] I. M. Georgescu, S. Ashhab, and F. Nori, Quantum simulation, *Reviews of Modern Physics* **86**, 153–185 (2014).
- [103] P. D. Nation, J. R. Johansson, M. P. Blencowe, and F. Nori, Colloquium: Stimulating uncertainty: Amplifying the quantum vacuum with superconducting circuits, *Rev. Mod. Phys.* **84**, 1 (2012).
- [104] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, 2000).
- [105] S. Lloyd, Universal quantum simulators, *Science* **273**, 1073 (1996).
- [106] D. Aharonov and A. Ta-Shma, Adiabatic quantum state generation and statistical zero knowledge, in *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03 (Association for Computing Machinery, New York, NY, USA, 2003) p. 20–29.
- [107] A. M. Childs and N. Wiebe, Hamiltonian simulation using linear combinations of unitary operations, *Quantum Info. Comput.* **12**, 901–924 (2012).
- [108] D. W. Berry, A. M. Childs, and R. Kothari, Hamiltonian simulation with nearly optimal dependence on all parameters, in *2015 IEEE 56th Annual Symposium on Foundations of Computer Science* (2015) pp. 792–809.
- [109] G. H. Low and I. L. Chuang, Hamiltonian simulation by qubitization, *Quantum* **3**, 163 (2019).
- [110] N. Lambert, M. Cirio, J.-D. Lin, P. Menczel, P. Liang, and F. Nori, Fixing detailed balance in ancilla-based dissipative state engineering, *Phys. Rev. Res.* **6**, 043229 (2024).
- [111] X. M. *et al.*, Stable quantum-correlated many-body states through engineered dissipation, *Science* **383**, 1332–1337 (2024).
- [112] T. S. Cubitt, Dissipative ground state preparation and the dissipative quantum eigensolver, [arXiv:2303.11962 \[quant-ph\]](https://arxiv.org/abs/2303.11962) (2023).
- [113] M. Raghuandan, F. Wolf, C. Ospelkaus, P. O. Schmidt, and H. Weimer, Initialization of quantum simulators by sympathetic cooling, *Science Advances* **6**, eaaw9268 (2020).
- [114] S. Polla, Y. Herasymenko, and T. E. O'Brien, Quantum digital cooling, *Phys. Rev. A* **104**, 012414 (2021).
- [115] Y. Zeng, Z.-Y. Zhou, E. Rinaldi, C. Gneiting, and F. Nori, Approximate autonomous quantum error correction with reinforcement learning, *Phys. Rev. Lett.* **131**, 050601 (2023).

- [116] Z. Ding, X. Li, and L. Lin, Simulating open quantum systems using Hamiltonian simulations, *PRX Quantum* **5** (2024), arXiv:2311.15533 [quant-ph].
- [117] R. Cleve and C. Wang, Efficient quantum algorithms for simulating Lindblad evolution, arXiv:1612.09512 [quant-ph] (2017).
- [118] P. Groszkowski and J. Koch, Squbits: a python package for superconducting qubits, *Quantum* **5**, 583 (2021).
- [119] T. Rajabzadeh, Z. Wang, N. Lee, T. Makihara, Y. Guo, and A. H. Safavi-Naeini, Analysis of arbitrary superconducting quantum circuits accompanied by a python package: SQcircuit, *Quantum* **7**, 1118 (2023).
- [120] S. Efthymiou, S. Ramos-Calderer, C. Bravo-Prieto, A. Pérez-Salinas, D. García-Martín, A. García-Saez, J. I. Latorre, and S. Carrazza, Qibo: a framework for quantum simulation with hardware acceleration, *Quantum Science and Technology* **7**, 015018 (2021).
- [121] <https://developer.nvidia.com/cuda-q>.
- [122] H. Silvério, S. Grijalva, A. Cornillot, L. Henriet, L. Ajdnik, P. Karalekas, L. Leclerc, C. de Terrasson, L. Vignoli, Jbrem, M. D'Arcangelo, C. Dalyac, Louis-PaulHenry, A. Wennersteen, D. Gessa, L. Emmanuel, R. Dutta, N. Shammah, Codoscope, MatthieuMoreau, R. Tsai, Y. Gondhalekar, A. B. Rava, A. Panigrahi, Harold, Julius, L.-J. Tallot, Oliver, Slimane33, and WingCode, pasqal-io/pulser: v1.1.0 (2024).
- [123] I. Buluta, S. Ashhab, and F. Nori, Natural and artificial atoms for quantum computation, *Rep. Prog. Phys.* **74**, 104401 (2011).
- [124] X. Gu, A. F. Kockum, A. Miranowicz, Y.-x. Liu, and F. Nori, Microwave photonics with superconducting quantum circuits, *Physics Reports* **718–719**, 1–102 (2017).
- [125] A. F. Kockum and F. Nori, Quantum bits with Josephson Junctions, in *Fundamentals and Frontiers of the Josephson Effect* (Springer International Publishing, 2019) p. 703–741.
- [126] B. Cheng, X.-H. Deng, X. Gu, Y. He, G. Hu, P. Huang, J. Li, B.-C. Lin, D. Lu, Y. Lu, C. Qiu, H. Wang, T. Xin, S. Yu, M.-H. Yung, J. Zeng, S. Zhang, Y. Zhong, X. Peng, F. Nori, and D. Yu, Noisy intermediate-scale quantum computers, *Frontiers of Physics* **18**, 21308 (2023).
- [127] R. Shillito, J. A. Gross, A. Di Paolo, E. Genois, and A. Blais, Fast and differentiable simulation of driven quantum systems, *Phys. Rev. Res.* **3**, 033266 (2021).
- [128] <https://github.com/qutip/qutip-cuquantum>.
- [129] H. Bayraktar, A. Charara, D. Clark, S. Cohen, T. Costa, Y.-L. L. Fang, Y. Gao, J. Guan, J. Gunnels, A. Haidar, A. Hehn, M. Hohnerbach, M. Jones, T. Lubowe, D. Lyakh, S. Morino, P. Springer, S. Stanwyck, I. Terentyev, S. Varadhan, J. Wong, and T. Yamaguchi, cuquantum sdk: A high-performance library for accelerating quantum science, in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Vol. 01 (2023) pp. 1050–1061.
- [130] S. Krämer, D. Plankensteiner, L. Ostermann, and H. Ritsch, QuantumOptics.jl: A Julia framework for simulating open quantum systems, *Computer Physics Communications* **227**, 109 (2018).
- [131] A. Gilchrist, N. K. Langford, and M. A. Nielsen, Distance measures to compare real and ideal quantum processes, *Phys. Rev. A* **71**, 062310 (2005).
- [132] <https://github.com/qutip/qutip-notebooks>.

- [133] S. Gleyzes, S. Kuhr, C. Guerlin, J. Bernu, S. Deléglise, U. Busk Hoff, M. Brune, J.-M. Raimond, and S. Haroche, Quantum jumps of light recording the birth and death of a photon in a cavity, *Nature* **446**, 297–300 (2007).
- [134] P.-L. Ardel, L. Hanschke, K. A. Fischer, K. Müller, A. Kleinkauf, M. Koller, A. Bechtold, T. Simmet, J. Wierzbowski, H. Riedl, G. Abstreiter, and J. J. Finley, Dissipative preparation of the exciton and biexciton in self-assembled quantum dots on picosecond time scales, *Phys. Rev. B* **90**, 241404 (2014).
- [135] N. Bartolo, F. Minganti, J. Lolli, and C. Ciuti, Homodyne versus photon-counting quantum trajectories for dissipative Kerr resonators with two-photon driving, *The European Physical Journal Special Topics* **226**, 2705–2713 (2017).
- [136] F. Minganti, N. Bartolo, J. Lolli, W. Casteels, and C. Ciuti, Exact results for schrödinger cats in driven-dissipative systems and their feedback control, *Scientific Reports* **6** (2016).
- [137] N. Bartolo, F. Minganti, W. Casteels, and C. Ciuti, Exact steady state of a kerr resonator with one- and two-photon driving and dissipation: Controllable wigner-function multimodality and dissipative phase transitions, *Phys. Rev. A* **94**, 033841 (2016).
- [138] K. A. Fischer, Y. A. Kelaita, N. V. Sapra, C. Dory, K. G. Lagoudakis, K. Müller, and J. Vučković, On-chip architecture for self-homodyned nonclassical light, *Phys. Rev. Appl.* **7**, 044002 (2017).