Numerical Methods for Quantum Optics and Open Quantum Systems

Alberto Mercurio *[1] and Daniele De Bernardis †[2]

[1]École Polytechnique Fédérale de Lausanne (EPFL), Rte Cantonale , Lausanne , Switzerland , 1015
[2]Istituto Nazionale di Ottica (INO-CNR), Largo Enrico Fermi, 6 , Firenze , Italia , 50125

17-06-2025

*alberto.mercurio@epfl.ch
†daniele.debernardis@cnr.it

Table of contents

1. Home Page

Numerical Methods for Quantum Optics and Open Quantum Systems is a hands-on course that shows you how to model and simulate open quantum systems in quantum optics with Python and QuTiP. The notes mix concise explanations, essential equations, and runnable code cells that work both on your computer and in Google Colab. Everything lives in a Quarto project on GitHub and is published in HTML and PDF for easy reading and collaboration. By the end, you will be able to set up and explore standard problems—such as photon cavities, two-level atoms, and open-system dynamics—using tools you can reuse in research and projects.

2. Introduction to quantum mechanics

Quantum mechanics (QM) is the general theory to describe any microscopic phenomenon (until now), at low and high energies, for a few and infinitely many particles.

Its structure can be broken down into three main pieces that can be considered logically independent:

1. Classical mechanics $F = ma$. As remarked in the third book by Landau (Landau and Lifshitz 1981), electrons, atoms, photons, and even subatomic particles are still described in terms of positions and velocities, following the algebraic relations of classical mechanics (non-relativistic or relativistic, depending). Somehow, this is the only way we have to build a measurable representation of nature. A physical system is thus defined by a set of degrees of freedom, which are quantified by vectors (non-relativistic, or 4-vectors relativistic) representing the canonical variables q, P of a Hamiltonian system. The Hamiltonian function $H(q, P)$ generates the time evolution through the Hamilton equation (equivalent to the Euler-Lagrange equation and thus to the Newton equation).
2. Non-commuting variables. The canonical variables are non-commuting $[q, P] \neq 0$, and thus, on top of their vectorial character, they contain more information than purely classical ones. While transforming like classical vectors, quantum variables are not vectors representable with three real numbers, but rather matrices with complex entries. To preserve the Hamiltonian structure of classical mechanics, they require precise commutation relations, which are postulated to ensure consistency. More precisely, quantum variables are linear operators over a Hilbert space. Vectors of the Hilbert space represent the state of the system. A system composed by many subsystem has the structure of a tensor product of Hilbert spaces. Solving a quantum system means solving a multi-linear algebra problem.
3. Probabilistic interpretation, Born rule, and measurement postulate. Experiments on microscopic systems show random outcomes on single realizations. Quantum mechanics describes thus only the probability of finding a certain number in a measurement. This probability distribution is given by the state just before the measure, more precisely by its modulus square. After the measurement the state is randomly projected in an eigenstates of the measured quantity, where its eigenvalue is the measurement output. We postulate this is the theory, because none knows what happens during the measurement, but we know that is consistent with the experiments. While we cannot include the measurement in the theory, we can include its effect as a random projection.

Interestingly, it is very important to know well classical mechanics, and its Hamiltonian formulation in particular, for the correct formulation of the problem. All the equations are the same, and the basics rules for the approximations (small amplitude expansions, decoupling of time scales, adiabatic approximations...etc) are still valid (since they are very constrained by the algebric relations between the variables). The basics intuition about which degrees of freedom are most important, and who contributes more or less to the dynamics are still valid. A good knowledge of classical mechanics provides the basis to set our problem right!

Classical equations are not enough, and to solve them we need multi-linear algebra. Instead of looking for solving sets of ordinary differential equations (ODE) like in classical mechanics, we are left we an infinite linear system of differential equations. We loose complexity from the linearity, but we regain it from the infinite dimension of the Hilbert space! Here we need to introduce truncations to finite dimensions and linear algebra methods to actually solve the equations.

The result of our calculations will be probability amplitudes and expectation values, all to be interpreted in a probabilistic way. We will see that this probabilistic interpretation can also enter in the equation by using stochastic calculus and so-called Monte Carlo methods.

Notice that here we did not talk about wave equation and wave-particle duality. These concepts are still here, but hidden in the non-commutativity of the variables. Typically quantum theory is exposed starting indeed from the Schrödinger equation and with the De-Broglie wave-particle duality, but for our purposes is actually nicer to start from this other formulation. This way of thinking about quantum mechanics was originally introduced by Heisenberg (W. Heisenberg 1925, 1927; Born, Heisenberg, and Jordan 1926).

2.1. Short digression: Heisenberg's matrix mechanics

While we often study QM from the wave-particle dualism and its Schrödinger formulation in terms of wave equation, it is worth noticing that the original Heisenberg's formulation (+ Born and Jordan) had exactly this form, with a specific focus on the multi-linear algebra element. It is called *matrix mechanics* and, while Schrödinger formalism is quite handy for analytical solutions, matrix mechanics is essentially the most natural way to perform numerical computations. After 100 years, we go back to the origin!

Heisenberg took for granted that the microscopic world is still represented in term of canonical variables $q, P, H(q, P)$ and Hamitonian equations $\dot{q} = \partial_P H$, $\dot{P} = -\partial_q H$. However he noticed that we never observe q, P of [e.g.] an atom or an electron, but rather the light emitted/scattered by these microscopic particles.

Figure 2.1.: A man walking at night under the street lights of the Munich's Englischer Garten. Werner could determine his position only when he's under the spot light.

In particular he knew the Balmer-Lyman-Paschen-Brackett series of the lines of the hydrogen atom. These series say that the frequency of each emission/absorption line of the hydrogen atom is given by the formula

$$\omega = \frac{R_Y}{\hbar}\left(\frac{1}{n^2} - \frac{1}{m^2}\right),$$

where $n, m \in \mathbb{N}$, R_Y is the Rydberg constant and $\hbar$ is the reduced Planck constant. Each observable frequency is determined by two integer numbers ω_{nm}, and it is thus natural to place it on a two-by-two table

$$\begin{pmatrix} 0 & \omega_{01} & \omega_{02} & \omega_{03} & \dots \\ \omega_{10} & 0 & \omega_{12} & \omega_{13} & \dots \\ \omega_{20} & \omega_{21} & 0 & \omega_{23} & \dots \\ \omega_{30} & \omega_{31} & \omega_{32} & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Each entry represents (according to Bohr) a jump between two stable orbits, or levels. However we only observe "transitions" between these orbits, and each single orbit is not observable even in principle. The position, velocity and everything measurable of a microscopic particle is then only defined during these jumps: it's like watching someone at night under the street lights...(Werner Heisenberg 1958).

So Heisenberg postulates that all the quantities associated to a microscopic particle must be also given in two-by-two tables, where [e.g.] one can see the position or momentum of the hydrogen's electron only during one of these transitions (after all we can just see the emitted light and its frequency).

$$q \longmapsto \begin{pmatrix} 0 & q_{01} & q_{02} & q_{03} & \cdots \\ q_{10} & 0 & q_{12} & q_{13} & \cdots \\ q_{20} & q_{21} & 0 & q_{23} & \cdots \\ q_{30} & q_{31} & q_{32} & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

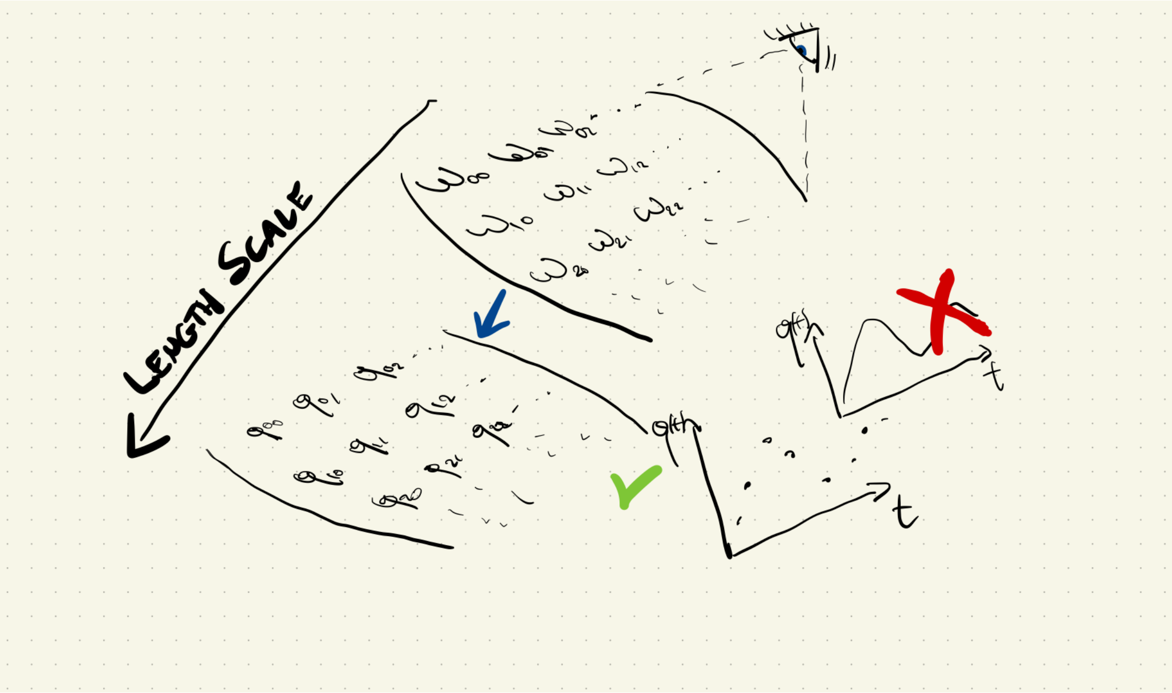


Figure 2.2.: What we see is only tables of frequencies. If we postulate that a classical mechanics trajectory exists it must be linked to these tables, since they are the only possible observable. The trajectory must be expressed as a table as well.

These tables are not only a convenient way to visualize the observations, but they are the physical variables of the theory. We can interpret $\{q_{nm}\}$ as a some sort of discretized version of the classical trajectory.

How do we use these tables to make theoretical predictions? A central problem for Heisenberg was how to sum and multiply these quantum variables, which are not numbers, but tables. The sum can be naturally taken as element-wise, but what about the multiplication? Heisenberg first noticed that the amplitude of emitted light also has a table $\{a_{nm}\}$, and it must follow the time evolution $a_{nm} \sim e^{-i\omega_{nm}t}$. Suppose we multiply two of these amplitudes, representing transitions sharing an equal level. In that case, we get another element that oscillates with a frequency of the table, $a_{nk}a^{-i\omega_{nk}t}\,a_{km}e^{-i\omega_{km}t} = a_{nk}a_{km}e^{-i\omega_{nm}t}$, because $\omega_{nk} + \omega_{km} = \omega_n - \omega_k + \omega_k - \omega_m = \omega_{nm}$ by definition. But the same is true if we sum over all possible k. Here it is not so easy to understand how Heisenberg went on (perhaps who know german very well could read Refs. (W. Heisenberg 1925, 1927; Born, Heisenberg, and Jordan 1926) and figure it out directly at the source), but, under the suggestion of Born and Jordan, they recognized that multiplication between physical variable represented as tables can be obtained using the matrix multiplication rule (which at the time was mostly unknown), $q^2_{nm} = \sum_k q_{nk}q_{km}$.

Interestingly, they immediately noticed that this implies the non-commutativity $[\hat{q}, \hat{P}] \neq 0$, which in principle seems not to be fixed (from here on we use that hat-notation $\hat{q}$ to indicate the quantum variable as a table, or in a more modern language, operator). However, they noticed that for consistency between the "oscillator" character of the tables (each element a_{nm} must evolve as an harmonic oscillator at the given frequency ω_{nm}), and the equation of the Hamiltonian mechanics there must be a precise commutation relation between q, P. Let's indeed consider that, as harmonically oscillating variables, each table element must follow

$$\partial_t q_{nm} = -i\omega_{nm}q_{nm} = -\frac{i}{\hbar}[H, q]_{nm} \qquad \partial_t P_{nm} = -i\omega_{nm}P_{nm} = -\frac{i}{\hbar}[H, P]_{nm}.$$

The use of the commutator and the Hamiltonian function here is justified by Bohr's theory, where the frequencies are given by energy levels $\omega_{nm} = (E_n - E_m)/\hbar$, and the energy is given by the Hamiltonian, which must be a diagonal table. Assuming $H = P^2/(2m) + V(q)$, if we compare these equations with Hamilton's

$$\partial_t \hat{q} = \hat{P}/m \qquad \partial_t \hat{P} = -\partial_{\hat{q}} V(\hat{q}),$$

we have that

$$P_{nm} = -im\,\omega_{nm}q_{nm} \qquad [\partial_q V(q)]_{nm} = \frac{i}{\hbar}[H, P]_{nm}.$$

Using $V(q) = -q$ we have that

$$[\hat{q}, \hat{P}] = i\hbar\hat{\mathbb{1}}.$$

Interestingly, the consistency between the existence of Bohr's levels, the oscillating origin of microscopic physical variables and Hamilton's equations implies a linear algebra structure that must be defined on an infinite dimensional vector space. A hint for this surprising consequence immediately comes by noticing

$$\mathrm{Tr}(\hat{q}\hat{P} - \hat{P}\hat{q}) = \mathrm{Tr}(\hat{q}\hat{P}) - \mathrm{Tr}(\hat{P}\hat{q}) = \mathrm{Tr}(\hat{q}\hat{P}) - \mathrm{Tr}(\hat{q}\hat{P}) = 0 \neq i\hbar\mathrm{Tr}(\hat{\mathbb{1}}) = \infty.$$

In any finite-dimensional space, this commutator cannot be proportional to the identity!

2.2. Exercise: the harmonic oscillator

Let's consider the example reported in the excellent book of Max Born (Born, Blin-Stoyle, and Radcliffe 1989): the harmonic oscillator.

$$\hat{H} = \frac{\hat{P}^2}{2m} + \frac{m\omega^2}{2}\hat{q}^2,$$

whose Hamilton equations are given by

$$\partial_t \hat{P} = -m\omega^2 \hat{q} \qquad \partial_t \hat{q} = \frac{\hat{P}}{m},$$

giving

$$\partial_t^2 \hat{q} = -\omega^2 \hat{q}.$$

Following Heisenberg, this must be true for each element of the corresponding quantum table [matrix, or operator], which, at the same time, must follow an oscillatory dynamics

$$q_{nm}(t) = q_{nm}(0)e^{-i\omega_{nm}t}.$$

Putting them together we find that $(\omega_{nm}^2 - \omega)q_{nm} = 0$, implying that $\omega_{nm} = \pm\omega$ and necessarily

$$q_{nm}(0) = 0 \quad if \quad m \neq n+1 \qquad q_{nm}(0) \neq 0 \quad if \quad m = n+1.$$

We thus find the structure

$$\hat{q} = \begin{pmatrix} 0 & q_{01} & 0 & 0 & \dots \\ q_{10} & 0 & q_{12} & 0 & \dots \\ 0 & q_{21} & 0 & q_{23} & \dots \\ 0 & 0 & q_{32} & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

and from $P_{nm} = im\omega_{nm}q_{nm}$,

$$\hat{P} = im\omega \begin{pmatrix} 0 & -q_{01} & 0 & 0 & ... \\ q_{10} & 0 & -q_{12} & 0 & ... \\ 0 & q_{21} & 0 & -q_{23} & ... \\ 0 & 0 & q_{32} & 0 & ... \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Using the commutation relation $[\hat{q}, \hat{P}] = i\hbar\hat{\mathbb{1}}$ we have that

$$-2im\omega \begin{pmatrix} q_{01}q_{10} & 0 & 0 & 0 & ... \\ 0 & q_{12}q_{21} - q_{01}q_{10} & 0 & 0 & ... \\ 0 & 0 & q_{23}q_{32} - q_{12}q_{21} & 0 & ... \\ 0 & 0 & 0 & q_{34}q_{43} - q_{23}q_{32} & ... \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{2.1}$$

$$= i\hbar \begin{pmatrix} 1 & 0 & 0 & 0 & ... \\ 0 & 1 & 0 & 0 & ... \\ 0 & 0 & 1 & 0 & ... \\ 0 & 0 & 0 & 1 & ... \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{2.2}$$

from which we derive the recurrence formula

$$q_{n\,n+1}q_{n+1\,n} = |q_{n\,n+1}|^2 = (n+1)\frac{\hbar}{2m\omega},$$

and

$$H_{nn} = m\omega^2 \left(|q_{n\,n+1}|^2 + |q_{n\,n-1}|^2\right) = \frac{\hbar\omega}{2}\,(2n+1)\,.$$

2.3. Being practical

After Dirac and Von Neumann we now know that there is more in quantum mechanics than ”tables”, commutator and oscillating elements. Indeed, having an Hilbert space for the linear operators directly implies the existence of states of the Hilbert space that constitute the domains of such operators. Moreover the linear algebra structure is also not sufficient, since for many particles we need a tensor product, and so we rather deal with multi-linear algebra. But the core of Heisenberg’s matrix mechanics and his perspective on quantum theory are still very central, as it will be clear in the rest of the course. What we do is in practice very similar in its logic:

- find all the degrees of freedom describing your problem as you would do in classical physics [e.g. define particle positions x_1, x_2, x_3 ..., or any generalized coordinate system q, or voltages and magnetic flux in circuit V, Φ, or even fields like $\mathbf{E}(\mathbf{r}), \mathbf{B}(\mathbf{r})$...].
- use the equation of motions from classical mechanics to derive a Lagrangian and then an Hamiltonian with canonical variables generically labelled as q, P. Here you could eventually employ approximations based on energy-time scales considerations (small amplitude oscillations, adiabatic elimination of fast variables etc...). This last sentence is not rigorous, but most of the time it works in the spirit of this quantum to classical relation.
- impose the canonical commutation relations $[\hat{q}, \hat{P}] = i\hbar$. Now the canonical variables (whatever they are) are interpreted as linear operators on an infinite dimensional Hilbert space $\mathcal{H}$ (Heisenberg's tables). This procedure preserves the algebraic relations between canonical variables given by Hamiltonian mechanics and, as Dirac showed, is equivalent to replacing Poisson brackets with commutator.
- find a good basis $\{|n\rangle\}$ for $\mathcal{H}$ and represent all the operators as infinite matrices by computing their matrix elements $\langle n|\hat{A}|m\rangle$ ($\hat{A}$ is an arbitrary operator of the considered problem).
- truncate the Hilbert space and make the matrices finite.
- finite dimensional matrices are typical objects well suitable for digital processors, so make numerics on a computer.
- interpret the results probabilistically using the Born rule.

3. About the course

3.1. Why simulate open quantum systems?

The experimental frontier of quantum optics increasingly targets systems that cannot be described by perfectly isolated, unitary dynamics. Photons leak from cavities, solid-state qubits couple to phonons, and measurement back-action reshapes quantum states in real time. In these scenarios the *open* character of the system—the interplay between coherent evolution and irreversible processes—becomes the defining feature, not a perturbation. Analytical solutions exist only for a handful of toy models; to design devices, interpret data, and test conceptual ideas we therefore rely on *numerical simulation* of open quantum dynamics.

Numerical methods allow us to:

- **Predict observables** such as spectra, correlation functions, or entanglement measures before running an experiment.
- **Prototype control protocols** (e.g., pulse shaping or feedback) that can stabilize fragile quantum states.
- **Explore parameter regimes** that are inaccessible analytically, revealing new phenomena like dissipative phase transitions or non-Markovian memory effects.

3.2. Why Python?

Python is *not* the fastest language for floating-point arithmetic—compiled languages like C or Fortran still win raw speed benchmarks—but it has become the lingua franca of modern scientific computing. Three qualities make it particularly compelling for our purposes:

1. **Expressiveness** – A succinct, readable syntax lowers cognitive overhead and lets us translate mathematical ideas into code quickly.
2. **Rich ecosystem** – Numpy, SciPy, Jupyter, Matplotlib, and data-analysis libraries coexist seamlessly, providing everything from linear algebra kernels to publication-quality plots.
3. **Community & portability** – Tutorials, StackOverflow answers, CI pipelines, and cloud platforms such as Google Colab enable beginners to run the same notebooks locally or on GPUs in the cloud with negligible setup.

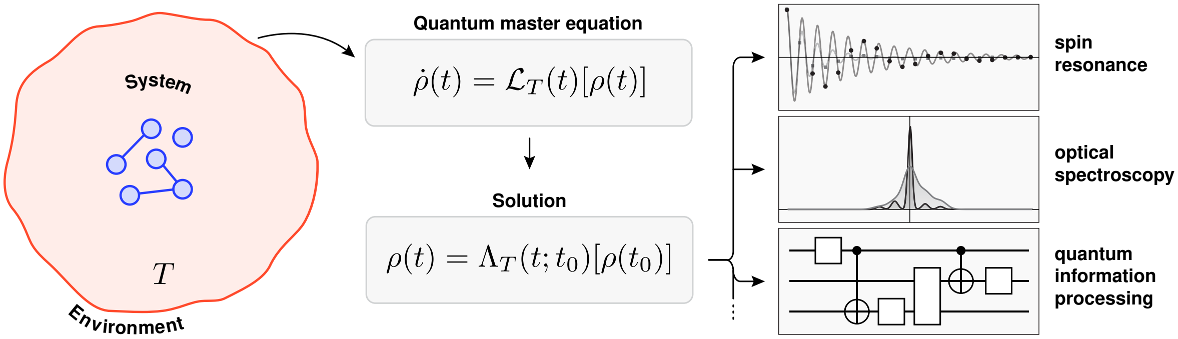


Figure 3.1.: Description of an open quantum system and its practical applications. A quantum system interacts with a macroscopic environment, leading to decoherence and dissipation. The evolution of the system is described the master equation $\dot{\hat{\rho}} = \mathcal{L}_T(t)[\hat{\rho}]$, where $\hat{\rho}$ is the density matrix and $\mathcal{L}_T(t)$ is the Liouville superoperator. The solution can be used to study the steady state and non-equilibrium properties of the system. The theoretical study of open quantum systems offers several tools for modeling spin resonance, optical spectra, and quantum information processing, and their use is certainly not limited to these fields and applications. Reproduced from (Campaioli, Cole, and Hapuarachchi 2024) under a CC BY 4.0 license.

Most importantly, Python hosts **QuTiP (Quantum Toolbox in Python)**(Johansson, Nation, and Nori 2012; Lambert et al. 2024) the de-facto standard library for simulating open quantum systems. QuTiP wraps efficient C and Fortran back-ends behind a high-level interface: you manipulate `Qobj` instances instead of raw matrices, and you call solvers such as `mesolve` or `mcsolve` for Lindblad-master equations and quantum trajectory simulations, respectively. The package is actively maintained, well documented, and battle-tested across thousands of research papers.

3.3. How does Python differ from other mainstream languages?

Language	Paradigm	Typical strength	Typical weakness
C / C++	Compiled, low-level	Maximal performance, fine-grained memory control	Verbose, higher barrier to entry, manual parallelization
Fortran	Compiled, array-oriented	Legacy HPC codes, excellent BLAS/LAPACK bindings	Limited modern features, smaller community

Language	Paradigm	Typical strength	Typical weakness
MATLAB	Proprietary, array-oriented	Integrated IDE, built-in plotting, domain-specific toolboxes	License cost, closed ecosystem
Python	Interpreted, multi-paradigm	Readability, vast open-source libraries, rapid prototyping	Overhead of interpreter, GIL limits naive multithreading

Python balances high-level productivity with the option to call compiled extensions (via Cython, Numba, or Rust bindings) whenever performance matters.

3.4. A glance at Julia and *QuantumToolbox.jl*

While Python dominates current scientific computing, it is not the only contender. In recent years, researchers and engineers have been exploring the need for a new programming language—one that combines the performance of compiled languages like C or Fortran with the ease of use and readability of scripting languages like Python or MATLAB. This is the motivation behind Julia.

Julia promises "*C-like speed with Python-like syntax*" by using just-in-time (JIT) compilation and a multiple-dispatch programming model. Within this language, the package *QuantumToolbox.jl*(Mercurio et al. 2025) has emerged as a high-performance analog to QuTiP. It mirrors QuTiP's API but benefits from Julia's performance model and native automatic differentiation. Benchmarks already demonstrate significant speed-ups, especially for large Hilbert spaces and GPU-accelerated workloads.

Nevertheless, Julia's ecosystem is still maturing. Its tooling, package stability, and IDE support are evolving rapidly but are not yet as robust as Python's. Similarly, QuantumToolbox.jl, while powerful, has a smaller user base and fewer educational resources compared to QuTiP. For a course focused on accessibility and broad applicability, we therefore choose to prioritize Python and QuTiP as the more mature and stable learning platform.

3.5. Course scope

In this course we therefore focus on **Python + QuTiP**. You will learn to:

- Build Hamiltonians and collapse operators in a composable way.
- Integrate master equations and unravel them into quantum trajectories.
- Compute expectation values, spectra, and correlation functions.

- Couple simulations to optimisation or machine-learning workflows within the wider Python ecosystem.

Where Julia can offer useful perspective we will point out parallels, but all hands-on examples will run in Python notebooks that you can execute locally or on Colab.

> **Take-away:** Numerical simulation is the microscope of modern quantum optics. Python and QuTiP give us a practical, accessible, and well-supported platform for that microscope—letting us peer into the dynamics of open quantum systems without getting lost in low-level details.

3.6. First steps in Python: lists, loops, and functions

3.6.1. Creating and using lists

Before diving into numerical simulations, it's useful to get acquainted with the basic syntax and features of Python. One of the simplest and most commonly used data structures is the **list**, which stores a sequence of elements. Lists are flexible—they can contain numbers, strings, or even other lists.

Here's how to create and access elements in a list:

```python
fruits = ['apple', 'banana', 'cherry']
print(f'First fruit: {fruits[0]}')
```

```
First fruit: apple
```

3.6.2. For loops

A `for` loop allows us to *iterate* through each item in a collection and execute the same block of code for every element. You will use loops constantly—whether you are sweeping parameter values, accumulating results, or analysing datasets—so it is worth seeing the syntax early.

```python
for fruit in fruits:
    print(f'I like {fruit}')
```

```
I like apple
I like banana
I like cherry
```

3.6.3. Defining functions

Functions bundle reusable logic behind a descriptive name. In quantum-optics simulations, well-structured functions help keep notebooks tidy—for instance, collecting the code that builds a Hamiltonian or evaluates an observable in one place. Below is a minimal example that squares a number.

```python
def square(x):
    return x * x

print(square(5))
```

```
25
```

3.6.4. Lambda (anonymous) functions

Occasionally we only need a *small, throw-away* function—say, as a callback or key in a sort operation. Python's `lambda` syntax lets us declare such anonymous functions in a single line, without the ceremony of `def`.

```python
square_lambda = lambda x: x * x
print(square_lambda(5))
```

```
25
```

3.6.5. Complex numbers

Python has built-in support for complex numbers, which are represented as `a + bj`, where `a` is the real part and `b` is the imaginary part. This is particularly useful in quantum mechanics, where complex numbers are ubiquitous.

```python
z = 1 + 2j
print(f'Complex number: {z}')
print(f'Real part: {z.real}')
print(f'Magnitude: {abs(z)}')
```

```
Complex number: (1+2j)
Real part: 1.0
Magnitude: 2.23606797749979
```

3.6.6. Why plain Python lists can be slow

Python lists store **references** to arbitrary Python objects. Each element carries its own type information and reference count. When you perform arithmetic on list elements, the interpreter must

1. Look up the byte-code for each operation.
2. Resolve types at runtime.
3. Dispatch to the correct C implementation.

This per-element overhead dominates runtime in numerical workloads.

3.6.7. Enter `numpy`

To overcome the performance limits of pure-Python lists, we turn to **NumPy**, which stores data in contiguous, fixed-type arrays and dispatches mathematical operations to highly-optimised C (and often SIMD/GPU) kernels. The example below shows how you can express a million-element computation in just two vectorised lines.

`numpy` provides fixed-type, contiguous arrays backed by efficient C (or SIMD/GPU) loops. Operations are dispatched **once** for the whole array, eliminating Python-level overhead and unlocking BLAS/LAPACK acceleration.

As an example, we can compute the sum of all the elements of a python list, comparing the performance with a numpy array.

```python
import numpy as np
import time # Only for benchmarking

my_list = [i / 1_000_000 for i in range(1_000_000)]

start = time.time() # start timer
sum_list = sum(my_list)  # sum using Python list
end = time.time()  # end timer
print(f'Sum using list: {sum_list}, '
      f'Time taken: {1e3*(end - start):.4f} milliseconds')

my_list_numpy = np.array(my_list)
start = time.time()  # start timer
sum_numpy = np.sum(my_list_numpy)  # sum using numpy array
end = time.time()  # end timer
print(f'Sum using numpy: {sum_numpy}, '
      f'Time taken: {1e3*(end - start):.4f} milliseconds')
```

```
Sum using list: 499999.5, Time taken: 8.0342 milliseconds
Sum using numpy: 499999.5, Time taken: 0.3781 milliseconds
```

NumPy is also able to perform vectorized operations, which let us express complex computations in a few lines of code. For example, we can compute a function of all elements in an array without writing explicit loops. This is not only more readable but also significantly faster, as the underlying C code can be optimised for performance.

```python
# Vectorized array operations
x = np.linspace(0, 100, 1_000_000)
y = np.sin(x) + 0.5 * x**2
print(y[:5])  # show first five results
```

```
[0.         0.00010001 0.00020002 0.00030005 0.00040008]
```

One line performs a million floating-point operations in compiled code—often orders of magnitude faster than an explicit Python loop.

4. Linear Algebra with NumPy and SciPy

Quantum systems are described by vectors and operators in complex Hilbert spaces. States $|\psi\rangle$ correspond to column vectors, and observables—like the Hamiltonian $\hat{H}$ or spin operators—are represented by matrices. Tasks such as finding energy spectra via eigenvalue decompositions, simulating time evolution through operator exponentials, and building composite systems with tensor (Kronecker) products all reduce to core linear-algebra operations.

In this chapter, we will leverage NumPy's and SciPy's routines (backed by optimized BLAS/LAPACK) to perform matrix–matrix products, eigen-decompositions, vector norms, and more. When system size grows, SciPy's sparse data structures and Krylov-subspace solvers will let us handle very large, structured operators efficiently.

By blending physical intuition (Schrödinger's equation, expectation values, operator algebra) with hands-on Python code, you'll see how powerful and intuitive modern linear-algebra libraries can be for quantum-mechanics simulations. Let's get started!

4.1. NumPy: The Foundation of Dense Linear Algebra

NumPy provides the `ndarray` type, an efficient, N-dimensional array stored in contiguous memory. This layout makes vectorized operations and low-level BLAS calls blazing fast. At its simplest, a 2D `ndarray` represents a matrix:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix},$$

and a 1D `ndarray` represents a column vector:

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}.$$

NumPy's dense arrays form the backbone of many quantum-simulation tasks—building Hamiltonians, computing overlaps, and propagating states all reduce to these core operations. Having a quick reference for them can speed up both writing and reading simulation code.

4.1.1. Summary of Core Functions

Operation	Equation	NumPy call
Matrix–matrix product	$C = AB$	`C = A.dot(B)` or `A @ B`
Matrix–vector product	$\mathbf{w} = A\mathbf{v}$	`w = A.dot(v)`
Eigenvalues and eigenvectors	$A\mathbf{x} = \lambda\mathbf{x}$	`w, v = np.linalg.eig(A)`
Determinant	$\det(A)$	`np.linalg.det(A)`
Inverse	A^{-1}	`np.linalg.inv(A)`
Norm (Frobenius)	$\|A\|_F = \sqrt{\sum_{ij} \|a_{ij}\|^2}$	`np.linalg.norm(A)`
Kronecker product	$A \otimes B$	`np.kron(A, B)`

In the table above, each abstract operation is paired with its NumPy call. Notice how intuitive the syntax is: the `@` operator reads like the usual linear-algebra notation.

4.1.2. Matrix–Matrix and Matrix–Vector Multiplication

Let's consider a simple example of a 2×2 matrix A and a 2-vector $\mathbf{v}$. This captures key ideas: operator composition via matrix–matrix products and state evolution via matrix–vector products. Indeed, in quantum mechanics, applying one operator after another corresponds to a matrix–matrix product, while acting on a quantum state uses a matrix–vector product. Consider the following:

```python
import numpy as np

# Define a 2×2 matrix and a 2-vector
A = np.array([[1, 2], [3, 4]])
v = np.array([5, 6])

# Matrix-matrix product
c = A @ A  # same as A.dot(A)
display("A @ A =", c)

# Matrix-vector product
w = A @ v  # same as A.dot(v)
display("A @ v =", w)
```

```
'A @ A ='
```

```
array([[ 7, 10],
       [15, 22]])
```

```
'A @ v ='
```

```
array([17, 39])
```

Here, `A @ A` computes A^2, and `A @ v` computes $A\mathbf{v}$.

4.1.3. Diagonalization

The eigenvalue problem is one of the cornerstones of both applied mathematics and quantum mechanics. Given a square matrix $A \in \mathbb{C}^{n\times n}$, we seek scalars $\lambda \in \mathbb{C}$ (eigenvalues) and nonzero vectors $\mathbf{x} \in \mathbb{C}^n$ (eigenvectors) such that

$$A\,\mathbf{x} = \lambda\,\mathbf{x}.$$

Physically, in quantum mechanics, A might be the Hamiltonian operator $\hat{H}$, its eigenvalues λ correspond to allowed energy levels, and the eigenvectors $\mathbf{x}$ represent stationary states. Mathematically, diagonalizing A transforms it into a simple form

$$A = V\,\Lambda\,V^{-1},$$

where Λ is the diagonal matrix of eigenvalues and the columns of V are the corresponding eigenvectors. Once in diagonal form, many operations—such as computing matrix exponentials for time evolution, powers of A, or resolving a system of differential equations—become trivial:

$$f(A) = V\,f(\Lambda)\,V^{-1}, \quad f(\Lambda) = \mathrm{diag}(f(\lambda_1), \dots, f(\lambda_n)).$$

In practice, NumPy's `np.linalg.eig` calls optimized LAPACK routines to compute all eigenpairs of a dense matrix:

```
w, v = np.linalg.eig(A)
display("Eigenvalues:", w)
display("Eigenvectors (as columns):\n", v)
```

```
'Eigenvalues:'
```

```
array([-0.37228132,  5.37228132])
```

```
'Eigenvectors (as columns):\n'
```

```
array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```

Under the hood, NumPy calls optimized LAPACK routines to diagonalize dense matrices.

4.1.4. Kronecker Product

In quantum mechanics, the state space of a composite system is the tensor product of the state spaces of its subsystems. If system 1 has Hilbert space $\mathcal{H}_A$ of dimension m and system 2 has $\mathcal{H}_B$ of dimension p, then the joint space is $\mathcal{H}_A \otimes \mathcal{H}_B$ of dimension mp. Operators on the composite system factorize as tensor (Kronecker) products of subsystem operators. For example, if A acts on system 1 and B on system 2, then

$$A \otimes B \; : \; \mathcal{H}_A \otimes \mathcal{H}_B \to \mathcal{H}_A \otimes \mathcal{H}_B$$

has matrix elements

$$(A \otimes B)_{(i\,,\,\alpha),(j\,,\,\beta)} = A_{ij}\, B_{\alpha\beta},$$

and in block form

$$A \otimes B = \begin{pmatrix} a_{11}\, B & a_{12}\, B & \cdots & a_{1n}\, B \\ \vdots & & & \vdots \\ a_{m1}\, B & a_{m2}\, B & \cdots & a_{mn}\, B \end{pmatrix},$$

yielding an $mp \times nq$ matrix when $A \in \mathbb{C}^{m\times n}$ and $B \in \mathbb{C}^{p\times q}$.

Why is this useful? In later chapters we will build multi-qubit gates (e.g. CNOT, controlled-phase), couple different oscillators, and assemble large Hamiltonians by taking tensor products of single-mode operators. The Kronecker product lets us lift any local operator into the full, composite Hilbert space.

In NumPy, the Kronecker product is computed with `np.kron`:

```python
B = np.array([[0, 1], [1, 0]])  # Pauli-X matrix
kron = np.kron(A, B)
display("A   B =", kron)
```

```
'A   B ='
```

```
array([[0, 1, 0, 2],
       [1, 0, 2, 0],
       [0, 3, 0, 4],
       [3, 0, 4, 0]])
```

Kronecker products build composite quantum-system operators from single-subsystem operators.

4.2. SciPy: Advanced Algorithms and Sparse Data

While NumPy covers dense linear algebra, SciPy complements it with:

Module	Purpose
`scipy.linalg`	Alternative LAPACK-based routines for dense ops
`scipy.sparse`	Data structures (COO, CSR, CSC) for sparse matrices
`scipy.sparse.linalg`	Iterative solvers (e.g. Arnoldi, Lanczos)
`scipy.integrate`	ODE and quadrature routines
`scipy.optimize`	Root-finding and minimization
`scipy.special`	Special mathematical functions

Compared to NumPy, SciPy's routines often expose extra options (e.g. choosing solvers) and can handle very large, sparse systems efficiently.

4.3. Some Useful Functions

Below are a few handy SciPy routines:

- Determinant: `scipy.linalg.det`
- Inverse: `scipy.linalg.inv`
- Frobenius norm: `scipy.linalg.norm`

```
import scipy.linalg as la

det = la.det(A)
inv = la.inv(A)
norm_f = la.norm(A)
display(det, inv, norm_f)
```

```
np.float64(-2.0)
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

```
np.float64(5.477225575051661)
```

4.4. Solving Linear Systems

A linear system has the form

$$A\mathbf{x} = \mathbf{b},$$

where $A \in \mathbb{R}^{n\times n}$ and $\mathbf{b} \in \mathbb{R}^n$ is known. For small n you can even solve by hand. For example, consider the 2×2 system

$$\begin{cases} x_1 + 2x_2 = 5, \\ 3x_1 + 4x_2 = 11. \end{cases} \qquad \Longrightarrow \quad A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \; \mathbf{b} = \begin{pmatrix} 5 \\ 11 \end{pmatrix}.$$

We can reproduce this with NumPy:

```
A = np.array([[1, 2], [3, 4]])
b = np.array([5, 11])
x = np.linalg.solve(A, b)
display("Solution x=", x)
```

```
'Solution x='
```

```
array([1., 2.])
```

SciPy's sparse module also offers `scipy.sparse.linalg.spsolve` for large, sparse A.

4.5. Sparse Matrices

As quantum systems scale to many degrees of freedom, the underlying operators—such as Hamiltonians or Liouvillian superoperators—grow exponentially in dimension but often remain highly structured and sparse. Instead of storing dense arrays with mostly zeros, sparse-matrix formats only record nonzero entries and their indices, dramatically reducing memory requirements. Common physical models, like spin chains with nearest-neighbor couplings or lattice Hamiltonians, have only $\mathcal{O}(N)$ or $\mathcal{O}(N \log N)$ nonzero elements, making sparse representations essential for large-scale simulations.

In the following sections, we will:

- Construct sparse matrices in COO formats with SciPy.
- Illustrate basic sparse-matrix operations (matrix–vector products, format conversions).
- Use `scipy.sparse.linalg.eigs` (Arnoldi) to compute a few eigenvalues of a sparse Hamiltonian.

The Coordinate (COO) format is a simple way to store sparse matrices. Instead of storing all entries, the COO format only keeps nonzero entries of the form (i, j, a_{ij}), which saves memory and speeds up computations. Graphically, a 5×5 example with 4 nonzeros might look like:

$$A = \begin{pmatrix} 7 & \cdot & \cdot & \cdot & 1 \\ \cdot & \cdot & 2 & \cdot & \cdot \\ \cdot & 3 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ 4 & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Here each number shows a location and its value. COO is very simple and intuitive, but not the most efficient. For larger matrices, we can use the Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) formats, which store the nonzero entries in a more compact way. The CSR format is very efficient for matrix–vector products.

Such matrix can be created in SciPy using the `coo_matrix` class:

```python
from scipy import sparse

# Create a sparse COO matrix
i = [0, 0, 1, 2, 4] # row indices
j = [0, 4, 2, 1, 0] # column indices
data = [7, 1, 2, 3, 4] # nonzero values
coo = sparse.coo_matrix((data, (i, j)), shape=(5, 5))
coo
```

```
<COOrdinate sparse matrix of dtype 'int64'
    with 5 stored elements and shape (5, 5)>
```

It is also possible to convert between different sparse formats. For example, to convert a COO matrix to CSR format, you can use the `tocsc()` method:

```
# Convert COO to CSR format
csr = coo.tocsr()
csr
```

```
<Compressed Sparse Row sparse matrix of dtype 'int64'
    with 5 stored elements and shape (5, 5)>
```

And the matrix–vector product is as simple as:

```
# Matrix-vector product
v = np.array([1, 2, 3, 4, 5])
w = coo @ v  # same as coo.dot(v)
w
```

```
array([12,  6,  6,  0,  4])
```

4.5.1. Eigenvalues of Sparse Matrices

Even with sparse storage, direct methods (dense diagonalization or full factorization) become intractable when the matrix dimension exceeds millions. To extract a few extremal eigenvalues or approximate time evolution, Krylov-subspace approaches (like the Arnoldi algorithm) build a low-dimensional orthonormal basis that captures the action of the operator on a subspace. By repeatedly applying the sparse matrix to basis vectors and orthogonalizing, Arnoldi produces a small Hessenberg matrix whose eigenpairs approximate those of the full operator. This hybrid strategy leverages both memory-efficient storage and iterative linear algebra to access spectral properties of huge quantum systems.

To approximate a few eigenvalues of a large, sparse matrix A, SciPy's `eigs` implements the Arnoldi algorithm. Under the hood it builds an m-dimensional Krylov basis. More precisely, given a starting vector v_1 with $\|v_1\|_2 = 1$, the m-dimensional Krylov subspace is

$$\mathcal{K}_m(A, v_1) = \operatorname{span}\{v_1, Av_1, A^2 v_1, \dots, A^{m-1} v_1\}.$$

The Arnoldi iteration produces the decomposition

$$AV_m = V_m H_m + h_{m+1,m}\, v_{m+1} e_m^\top,$$

where

- $V_m = [v_1, \dots, v_m]$ has orthonormal columns,
- H_m is an $m \times m$ upper-Hessenberg matrix,
- e_m is the m-th canonical basis vector.

The eigenvalues of H_m are called **Ritz values**; they approximate eigenvalues of A. As m grows, the approximation improves. In practice we combine Arnoldi with a **restart** strategy (after reaching a given m we keep the most accurate Ritz vectors and build a fresh Krylov basis). SciPy's `scipy.sparse.linalg.eigs` wrapper uses the implicitly restarted Arnoldi method from ARPACK.

As a pseudo-code, the Arnoldi algorithm can be summarized as follows:

1. Pick a random vector v and **normalize** it.
2. For $j = 1, \dots, m$
    1. $w = Av_j$
    2. **Orthogonalize**:

       $$h_{i,j} = v_i^\dagger w, \quad w \leftarrow w - h_{i,j} v_i \quad (i = 1, \dots, j)$$
    3. $h_{j+1,j} = \|w\|_2$.
    4. If $h_{j+1,j} = 0$, stop (the Krylov subspace is invariant).
    5. $v_{j+1} = w / h_{j+1,j}$.

The cost is m sparse matrix–vector products and $\mathcal{O}(m^2 n)$ scalar operations for orthogonalization (which stays moderate when $m \ll n$).

Here's a concrete example:

```python
from scipy.sparse.linalg import eigs

# Compute the 2 largest-magnitude eigenvalues of coo
vals, vecs = eigs(coo, k=2)
display("Sparse eigenvalues:", vals)
```

```
'Sparse eigenvalues:'
```

```
array([7.53112887+0.j, 2.44948974+0.j])
```

5. Speeding up Python for Linear Algebra Tasks

Python is easy to read, but pure-Python loops can be slow if you do not leverage optimized libraries (BLAS, LAPACK). Here we explore two tools—Numba and JAX—to accelerate common linear algebra operations.

5.1. Numba: Just-In-Time Compilation

Numba uses LLVM to compile Python functions to machine code at runtime. Key points:

- **Decorators**: Use `@njit` (nopython mode) for best speed.
- **Type inference**: Numba infers types on first run, then compiles specialized code.
- **Compilation overhead**: The first call incurs compilation time; subsequent calls are fast.
- **Object mode vs nopython mode**: Always aim for nopython mode to avoid Python object overhead.

  JIT Workflow 1. Call function → type inference → LLVM IR generation.
  2. LLVM IR → machine code (cached).
  3. Subsequent calls use cached machine code.

Example: Matrix–Vector Multiplication

```
from numba import njit
import numpy as np
import time # for timing

@njit
def matvec(A, x):
    m, n = A.shape
    y = np.zeros(m)
```

```
    for i in range(m):
        temp = 0.0
        for j in range(n):
            temp += A[i, j] * x[j]
        y[i] = temp
    return y

# Prepare data
dim = 500
A = np.random.rand(dim, dim)
x = np.random.rand(dim)

# Using NumPy's dot product
start = time.time()
y0 = A @ x
end = time.time()
print("NumPy time (ms): ", 1e3*(end - start))

# Using Numba's compiled function
y0 = matvec(A, x) # First call for compilation

start = time.time()
y1 = matvec(A, x)
end = time.time()
print("Numba time (ms): ", 1e3*(end - start))
```

```
NumPy time (ms):  0.3027915954589844
Numba time (ms):  0.29087066650390625
```

In practice, Numba can speed up this looped version by 10×–100× compared to pure Python, approaching the speed of NumPy's optimized routines. The reader is encouraged to try the code without the `@njit` decorator to see the difference in performance.

5.2. JAX: XLA Compilation and Automatic Differentiation

JAX is a high-performance library from Google Research that extends NumPy with just-in-time compilation and automatic differentiation. It - Compiles array operations via XLA, fusing kernels and reducing Python overhead. - Supports GPU and TPU backends with minimal code changes. - Provides `grad` for gradients of scalar functions, enabling optimisation and machine-learning tasks. - Offers advanced transformations like `vmap` (vectorisation) and `pmap` (parallelism on multiple devices).

JAX is widely used in deep learning frameworks (e.g. Flax, Haiku), reinforcement learning, and scientific research (including physics simulations), thanks to its blend of speed and flexibility.

5.2.1. A Quick Overview of Automatic Differentiation

Automatic differentiation (AD) is a family of techniques to compute exact derivatives of functions defined by computer programs. Unlike symbolic differentiation (which can lead to expression swell) or numerical finite-difference (which suffers from truncation and round-off error), AD exploits the fact that any complex function is ultimately composed of a finite set of elementary operations (addition, multiplication, sin, exp, …) whose derivatives are known exactly.

5.2.1.1. Limitations of Finite Differences

A common finite-difference formula for a scalar function $f(x)$ is the central difference

$$\frac{df}{dx}(x) \approx \frac{f(x+h) - f(x-h)}{2h},$$

with local truncation error $\mathcal{O}(h^2)$. However, this approach has important limitations:

1. **Truncation vs. round-off**: If h is too large, the $\mathcal{O}(h^2)$ term dominates. If h is too small, floating-point cancellation makes the numerator $f(x+h) - f(x-h)$ inaccurate.
2. **Cost with many parameters**: For $f : \mathbb{R}^n \to \mathbb{R}$, the gradient component i is

   $$\frac{\partial f}{\partial x_i}(\mathbf{x}) \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h}.$$

   Computing all n components requires $2n$ evaluations of f, so the cost scales as $\mathcal{O}(n)$ in f-calls. For large n (many parameters), this becomes prohibitive.
3. **Non-smooth or branching code**: When f contains control flow or non-differentiable operations, finite differences may give misleading or undefined results.

5.2.1.2. Automatic Differentiation and the Chain Rule

Automatic differentiation (AD) applies the chain rule to each elementary operation in code (addition, multiplication, sin, exp, etc.), yielding exact derivatives up to floating-point precision. For a composition

$$u = g(x), \quad y = f(u),$$

AD uses the chain rule:

$$\frac{dy}{dx} = \frac{df}{du}\frac{dg}{dx}.$$

In more complex nests, e.g.

$$v = h(u), \quad u = g(x), \quad y = f(v),$$

we get

$$\frac{dy}{dx} = \frac{df}{dv}\frac{dh}{du}\frac{dg}{dx}.$$

AD comes in two modes:

- **Forward mode** (propagate derivatives from inputs to outputs).
- **Reverse mode** (propagate sensitivities from outputs back to inputs).

JAX implements both and selects the most efficient strategy automatically.

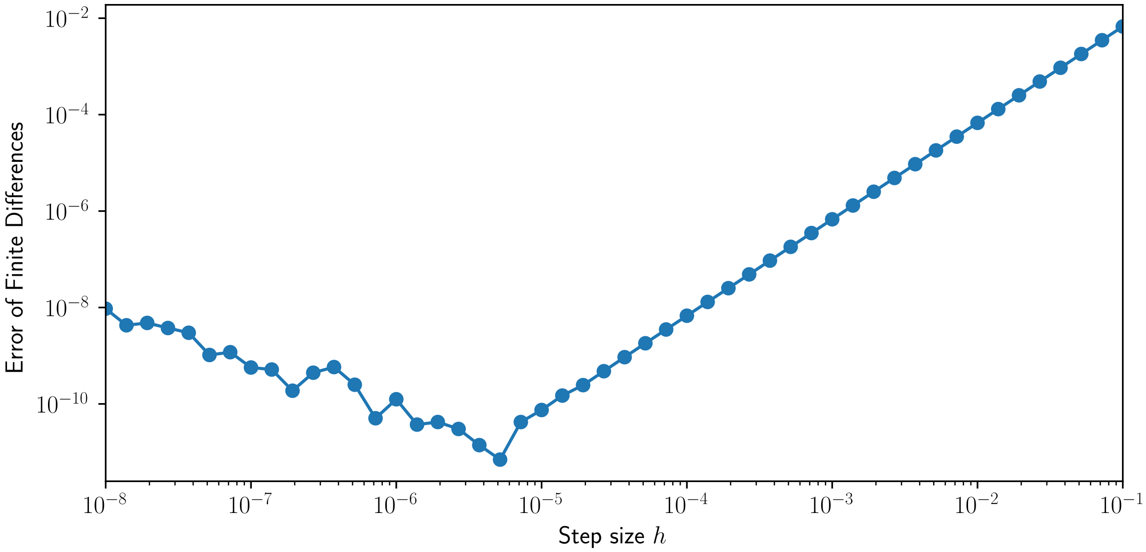
5.2.1.3. Comparing Accuracy: AD vs Finite Differences

Below is a Quarto code cell that plots the error of finite differences (varying step size h) and automatic differentiation against the true derivative of $f(x) = e^{\sin(x)}$ at $x = 1.0$.

```python
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt # for plotting

# Set JAX to use 64-bit floats
jax.config.update("jax_enable_x64", True)
```

```python
# Define function and true derivative
def f_np(x):
    return np.exp(np.sin(x))

def df_true(x):
    return np.cos(x) * np.exp(np.sin(x))

# Point of evaluation
x0 = 1.0

# Finite-difference errors for varying h
hs = np.logspace(-8, -1, 50)
errors_fd = []
for h in hs:
    df_fd = (f_np(x0 + h) - f_np(x0 - h)) / (2 * h)
    errors_fd.append(abs(df_fd - df_true(x0)))

# Automatic differentiation error (constant)
df_ad = jax.grad(lambda x: jnp.exp(jnp.sin(x)))(x0)
error_ad = abs(np.array(df_ad) - df_true(x0))

print(f"AD error: {error_ad}")
print(f"FD minimum error: {min(errors_fd)}")

# Plot
fig, ax = plt.subplots()
ax.loglog(hs, errors_fd, marker="o")
ax.set_xlabel("Step size $h$")
ax.set_ylabel("Error of Finite Differences")

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```

```
AD error: 0.0
FD minimum error: 7.006839553014288e-12
```



This plot illustrates that finite differences achieve minimal error at an optimal h, but degrade for too large or too small h, while AD remains accurate to machine precision regardless of step size.

5.3. Why Computing Gradients Is Important in Quantum Physics

In quantum physics, many problems reduce to optimizing parameters in a model or a control protocol. Computing gradients of a cost function with respect to these parameters is essential for efficient and reliable optimization.

1. **Variational quantum algorithms**: In methods like the variational quantum eigensolver (VQE)(Peruzzo et al. 2014), a parametrised quantum state $|\psi(\theta)\rangle$ depends on parameters $\theta = (\theta_1, \dots, \theta_n)$. One minimises the expectation

   $$E(\theta) = \langle \psi(\theta) | \hat{H} | \psi(\theta) \rangle.$$

   Gradient-based methods require

   $$\frac{\partial E}{\partial \theta_i} = \frac{\partial}{\partial \theta_i} \langle \psi(\theta) | \hat{H} | \psi(\theta) \rangle.$$

   AD enables exact evaluation of these derivatives through the quantum circuit parameters, improving convergence compared to gradient-free methods.

2. **Quantum optimal control**(D'Alessandro 2021; Khaneja et al. 2005): One shapes control fields $u(t)$ in the Hamiltonian

$$\hat{H}(t;u) = \hat{H}_0 + \sum_i u_i(t)\hat{H}_i$$

   to drive the system from an initial state $|\psi_0\rangle$ to a target $|\psi_T\rangle$. A typical cost function is

$$J[u] = 1 - |\langle\psi_T|\mathcal{U}_T[u]|\psi_0\rangle|^2,$$

   where $\mathcal{U}_T[u]$ is the time-ordered evolution. Computing gradients $\delta J/\delta u_i(t)$ is needed for gradient-ascent pulse engineering (GRAPE) algorithms. AD can differentiate through time-discretised propagators and ODE solvers, automating derivation of $\delta J/\delta u_i(t)$ and providing machine-precision gradients for faster convergence.

3. **Parameter estimation and tomography**(Lvovsky and Raymer 2009): Maximum-likelihood estimation for quantum states or processes often involves maximising a log-likelihood $L(\theta)$. Gradients speed up estimation and enable standard optimisers (e.g. L-BFGS).

By providing exact, efficient gradients even through complex quantum simulations (time evolution, measurement models, noise), automatic differentiation (via JAX or similar frameworks) has become a key tool in modern quantum physics research.

5.4. Summary

- **Numba**: Best for speeding up existing NumPy loops with minimal code changes. Ideal when you do not need gradients or accelerators.
- **JAX**: Ideal for optimisation tasks requiring gradients, large-scale batch operations, or GPU/TPU acceleration. The XLA compiler often outperforms loop-based JIT for fused kernels.

6. Ordinary Differential Equations

An ordinary differential equation (ODE) is an equation involving functions of one independent variable (for instance, time) and its derivatives. In the simplest scenario, suppose we have an unknown function $y(t)$. A first-order ODE can be written as:

$$\frac{dy(t)}{dt} = f(y(t), t),$$

where f is a known function, and $y(t)$ is the unknown to be determined. Higher-order ODEs can often be recast as systems of first-order ODEs by introducing additional variables for the higher derivatives.

6.1. General Definition and Examples

To see how ODEs arise in physical scenarios, consider Newton's second law, $m\,\frac{d^2x}{dt^2} = F(x,t)$. This second-order ODE can be reduced to a system of two first-order ODEs by introducing an auxiliary variable for velocity $v(t) = \frac{dx}{dt}$. Then we have:

$$\begin{cases} \frac{dx}{dt} = v, \\ \frac{dv}{dt} = \frac{F(x,t)}{m}. \end{cases}$$

In quantum mechanics, the time-dependent Schrödinger equation

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

can be viewed as a first-order ODE in the Hilbert space: the role of $|\psi(t)\rangle$ is analogous to $y(t)$, and $-\frac{i}{\hbar}\,\hat{H}$ plays the role of $f(\,\cdot\,,t)$ (assuming a time-independent $\hat{H}$). This analogy suggests that the Schrödinger equation can be treated using standard ODE solution techniques or, in more complicated cases, numerical integration.

A **linear** ODE has the form: $\frac{d\mathbf{y}(t)}{dt} = A\,\mathbf{y}(t) + \mathbf{b}(t)$, where $\mathbf{y}(t)$ is a vector function of time, A is a constant (or possibly time-dependent) matrix, and $\mathbf{b}(t)$ is a known inhomogeneous term. If $\mathbf{b}(t) = \mathbf{0}$, the equation is said to be homogeneous.

6.2. Solving Linear ODEs by Diagonalizing the System Matrix

A common case in quantum mechanics and in classical physics is the linear homogeneous system:

$$\frac{d\mathbf{y}(t)}{dt} = A\,\mathbf{y}(t), \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{6.1}$$

where A is a constant $n \times n$ matrix, and $\mathbf{y}_0$ is the initial condition.

6.2.1. Eigenvalue Decomposition

If A is diagonalizable, we can write:

$$A = V\,D\,V^{-1},$$

where D is a diagonal matrix whose entries are the eigenvalues λ_i of A, and the columns of V are the corresponding eigenvectors. Define:

$$\mathbf{z}(t) = V^{-1}\,\mathbf{y}(t).$$

Then, plugging this into Equation 6.1, we get

$$\frac{d\mathbf{z}(t)}{dt} = V^{-1}\,\frac{d\mathbf{y}(t)}{dt} = V^{-1}\,A\,\mathbf{y}(t) = V^{-1}\,(V\,D\,V^{-1})\,\mathbf{y}(t) = D\,\mathbf{z}(t).$$

Hence, in the $\mathbf{z}$-coordinates, the system becomes a set of n uncoupled first-order ODEs:

$$\frac{dz_i}{dt} = \lambda_i\,z_i(t), \quad \text{for } i = 1, \ldots, n.$$

These have the well-known solutions:

$$z_i(t) = z_i(0)\,e^{\lambda_i t}.$$

To enforce the initial condition $\mathbf{y}(0) = \mathbf{y}_0$, we note that $\mathbf{z}(0) = V^{-1}\,\mathbf{y}_0$. Hence, transforming back, we get:

$$\mathbf{y}(t) = V\,\mathbf{z}(t) = V\begin{pmatrix} z_1(0)\,e^{\lambda_1 t} \\ z_2(0)\,e^{\lambda_2 t} \\ \vdots \\ z_n(0)\,e^{\lambda_n t} \end{pmatrix} = V\,e^{Dt}\,V^{-1}\,\mathbf{y}_0.$$

Therefore, we obtain the compact form:

$$\mathbf{y}(t) = e^{At}\,\mathbf{y}_0,$$

or, equivalently,

$$\mathbf{y}(t) = V \begin{pmatrix} e^{\lambda_1} & & \\ & \ddots & \\ & & e^{\lambda_n} \end{pmatrix} V^{-1}\mathbf{y}_0\,.$$

In the case of A Hermitian, the time evolution can be expanded as

$$\mathbf{y}(t) = \sum_i (\mathbf{v}_i^\dagger \cdot \mathbf{y}_0)\mathbf{v}_i\, e^{\lambda_i t}\,,$$

where $\mathbf{v}_i$ are the eigenvectors of the matrix.

6.2.2. Relation to the Schrödinger Equation

When dealing with the time-dependent Schrödinger equation for a *time-independent* Hamiltonian $\widehat{H}$, we can represent $|\psi(t)\rangle$ in a certain basis, turning the Schrödinger equation into:

$$i\hbar\,\frac{d}{dt}\mathbf{c}(t) = H\,\mathbf{c}(t),$$

or equivalently,

$$\frac{d\mathbf{c}(t)}{dt} = -\frac{i}{\hbar}\,H\,\mathbf{c}(t).$$

We can identify $A = -\frac{i}{\hbar}\,H$. If H is diagonalizable (e.g., Hermitian matrices always have a complete set of orthonormal eigenvectors), then the above solution technique via diagonalization applies. The resulting exponential solution corresponds to the usual $e^{-\frac{i}{\hbar}Ht}$ operator that defines unitary time evolution in quantum mechanics.

6.2.2.1. Example: Harmonic Oscillator

The harmonic oscillator is described by the second-order ODE:

$$\frac{d^2x}{dt^2} + \omega^2 x = 0,$$

which can be rewritten as a first-order system:

$$\begin{cases} \frac{dx}{dt} = v, \\ \frac{dv}{dt} = -\omega^2 x. \end{cases}$$

or, in matrix form:

$$\frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix}\begin{pmatrix} x \\ v \end{pmatrix}.$$

By diagonalizing the matrix, we can find the solution to this system.

```python
import numpy as np

omega = 2.0

# Define the 2x2 matrix A
A = np.array([[0.0,  1.0],
              [-omega**2, 0.0]])

# Initial condition: x(0) = 1, v(0) = 0.5
x0 = np.array([1.0, 0.5])

# Diagonalize A
eigs, V = np.linalg.eig(A)
V_inv = np.linalg.inv(V)

z0 = V_inv @ x0

# Define a time array
t_points = np.linspace(0, 2, 200)

X_t = []
for t in t_points:
```

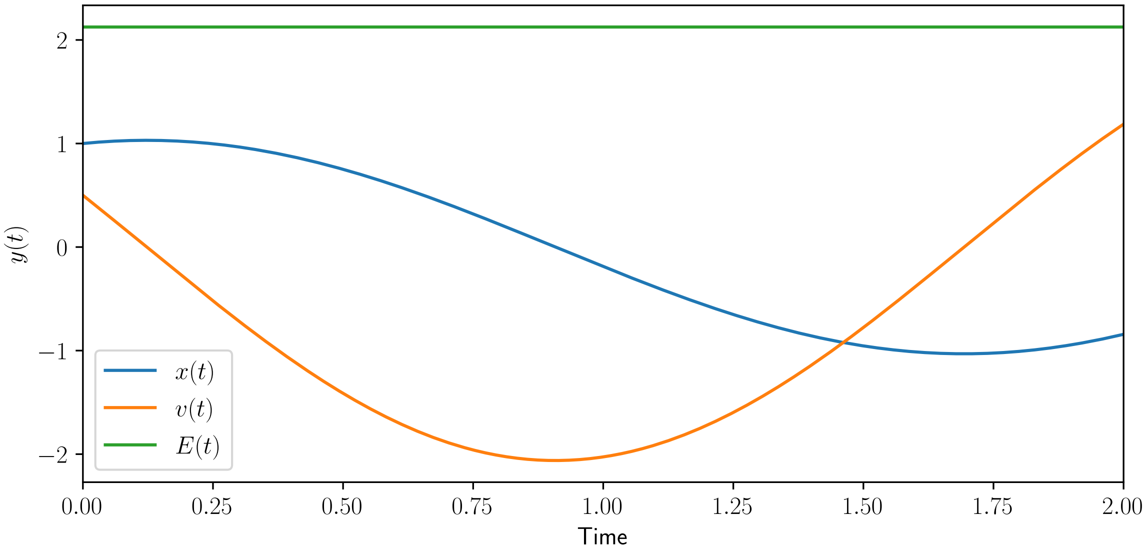
```
    # Compute the solution at time t
    z_t = np.diag(np.exp(eigs * t)) @ z0
    x_t = V @ z_t # Transform back to original coordinates
    X_t.append(x_t)

X_t = np.array(X_t).real

print("x(2) = ", X_t[-1])
```

```
x(2) =  [-0.84284424  1.18678318]
```

We have: - `A`: the system matrix. - `y0`: initial condition $\mathbf{y}(0)$. - We diagonalize A to find $A = VDV^{-1}$. - Then $\exp(At) = V\exp(Dt)V^{-1}$.

If you run the code, you'll see the final value of $\mathbf{y}(2)$.
We could also visualize the time evolution:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(t_points, X_t[:, 0], label="$x(t)$")
ax.plot(t_points, X_t[:, 1], label="$v(t)$")
ax.plot(t_points, 0.5 * omega**2 * X_t[:, 0]**2 + 0.5 * X_t[:, 1]**2,
        label="$E(t)$")
ax.set_xlabel("Time")
ax.set_ylabel("$y(t)$")
ax.legend()

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```



6.3. Numerical Solution via the Euler Method

In many realistic situations (e.g., time-dependent Hamiltonians, nonlinear effects, large dissipative systems described by master equations), finding an exact analytic solution can be very challenging or impossible. We then rely on *numerical methods* to solve ODEs.

6.3.1. Forward Euler Method

One of the simplest methods is the **forward Euler method**. Suppose we want to solve:

$$\frac{d\mathbf{y}(t)}{dt} = \mathbf{f}(\mathbf{y}(t), t), \quad \mathbf{y}(0) = \mathbf{y}_0.$$

We discretize time into steps $t_n = n\,h$ with step size h. The Euler method approximates the derivative at t_n by a difference quotient:

$$\frac{d\mathbf{y}(t_n)}{dt} \approx \frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{h}.$$

Hence, the system becomes the algebraic update:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\,\mathbf{f}(\mathbf{y}_n, t_n),$$

with $\mathbf{y}_0$ known. After iterating this rule for $n = 0, 1, 2, \ldots$, we obtain an approximate solution at discrete times t_n.

6.3.2. Stability Criterion for the Euler Method

While the Euler method is straightforward, it can be susceptible to numerical instability when the system has rapidly decaying or oscillatory modes. For example, consider the test equation $\frac{dy}{dt} = \lambda\, y$, where λ is a (possibly complex) constant. The exact solution is $y(t) = y(0)\, e^{\lambda t}$. In the Euler scheme, we get

$$y_{n+1} = y_n + h\,\lambda\, y_n = (1 + h\,\lambda)\, y_n.$$

Thus,

$$y_n = (1 + h\,\lambda)^n\, y_0.$$

For the method to be stable (i.e., for y_n to remain bounded in the limit $n \to \infty$ when the exact solution is stable), we require:

$$|1 + h\,\lambda| < 1,$$

when the real part of λ is negative (dissipative system). If this condition is not met, the numerical solution may diverge even though the true solution decays exponentially. In practice, one must choose the time step h small enough to satisfy such stability constraints.

6.3.2.1. Example: Harmonic Oscillator with Euler Method

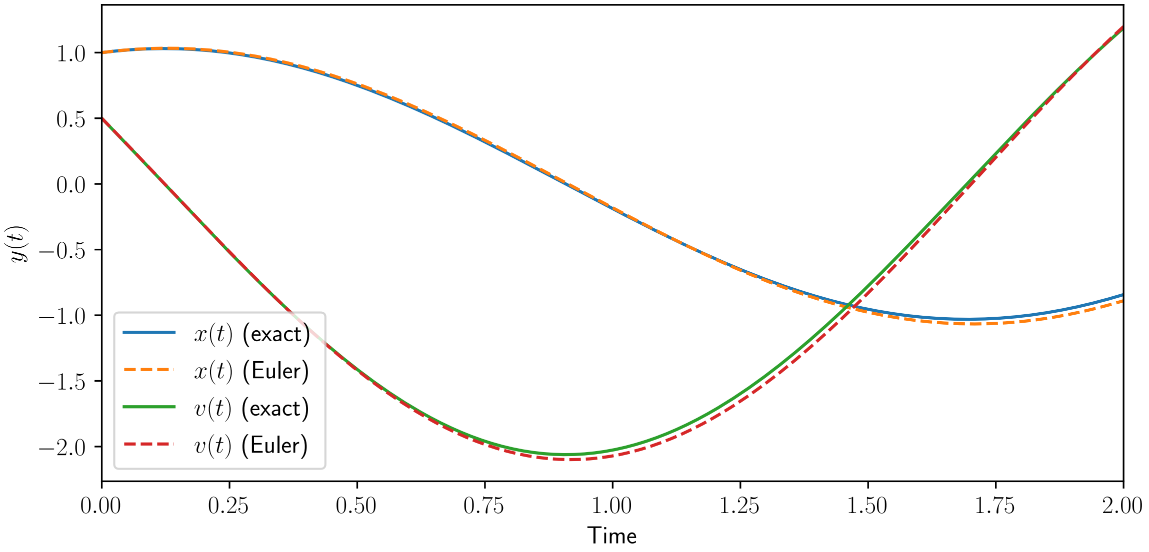
Let's now implement the **forward Euler** method for a simpler ODE. Consider the same harmonic oscillator, Euler's method approximates the evolution as:

$$\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} \simeq \begin{pmatrix} x_n \\ v_n \end{pmatrix} + h \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix} \begin{pmatrix} x_n \\ v_n \end{pmatrix},$$

where h is the time step.

```
h = 0.01

X_t_euler = np.zeros((len(t_points), 2))
X_t_euler[0] = x0
for n in range(len(t_points) - 1):
    X_t_euler[n+1] = X_t_euler[n] + h * A @ X_t_euler[n]

fig, ax = plt.subplots()
ax.plot(t_points, X_t[:, 0], label="$x(t)$ (exact)")
```

```
ax.plot(t_points, X_t_euler[:, 0], label="$x(t)$ (Euler)", linestyle='--')
ax.plot(t_points, X_t[:, 1], label="$v(t)$ (exact)")
ax.plot(t_points, X_t_euler[:, 1], label="$v(t)$ (Euler)", linestyle='--')
ax.set_xlabel("Time")
ax.set_ylabel("$y(t)$")
ax.legend()

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```



Here we see how the Euler solution compares to the exact solution obtained via diagonalization. Notice that using a large time step h can cause the Euler solution to deviate significantly from the exact decay (and may even diverge if $|1 - \lambda h| \geq 1$).

6.4. Applying These Methods to the Schrödinger Equation

Time-Independent Hamiltonian

For a time-independent Hamiltonian $\hat{H}$, the Schrödinger equation in vector form reads:

$$i\hbar \frac{d\mathbf{c}(t)}{dt} = H\,\mathbf{c}(t).$$

By setting $A = -\frac{i}{\hbar}H$, we recognize that this is a linear ODE. If H (or A) is diagonalizable, its eigen-decomposition yields an analytic solution. In quantum optics, these solutions describe unitary time evolution of a closed system, often expressed as:

$$\mathbf{c}(t) = e^{-\frac{i}{\hbar}Ht}\,\mathbf{c}(0).$$

6.4.1. Time-Dependent Hamiltonian

When $\widehat{H}(t)$ varies explicitly with time, one no longer has a simple exponential solution. Instead, one can divide the time interval of interest into many small sub-intervals and approximate $\widehat{H}(t)$ as constant in each interval. This procedure is related to the *time-ordered exponential*, but from a numerical perspective, we can simply implement a step-by-step integration (e.g., Euler, Runge–Kutta, or other higher-order methods) to construct $|\psi(t_{n+1})\rangle$ from $|\psi(t_n)\rangle$.

6.4.2. Open Quantum Systems

In open quantum systems, the evolution of the density matrix $\rho(t)$ is often governed by the master equation:

$$\frac{d\rho(t)}{dt} = \mathcal{L}[\rho(t)],$$

where $\mathcal{L}$ is the so-called Liouvillian superoperator, which could contain both Hamiltonian (coherent) parts and dissipative terms. Numerically, one can *vectorize* $\rho(t)$ (flattening the matrix into a vector) and represent $\mathcal{L}$ as a matrix $\mathcal{L}_{\text{mat}}$. Then, the equation again has the familiar linear form:

$$\frac{d\mathbf{r}(t)}{dt} = \mathcal{L}_{\text{mat}}\,\mathbf{r}(t).$$

Hence, the same techniques (matrix diagonalization for analytical solutions, or time stepping methods like Euler, Runge–Kutta, etc. for numerical solutions) remain valid.

6.5. Conclusion

In summary:

- An Ordinary Differential Equation (ODE) involves a function of one variable and its derivatives.

- When an ODE is linear and time-independent, one can analytically solve it by diagonalizing the system matrix.
- For more complicated (time-dependent or nonlinear) problems, numerical integration methods such as the Euler method can be applied.
- The Euler method is conceptually simple but demands careful choice of time step to ensure stability, particularly when the system matrix has eigenvalues with large negative real parts or when fast decaying/oscillatory modes are present.
- These ideas are directly applicable to quantum mechanical systems such as the Schrödinger equation or master equations for open systems. In the Schrödinger equation, diagonalization corresponds to finding energy eigenstates and frequencies, while in open quantum systems, vectorization plus diagonalization or numerical iteration handles both coherent and dissipative dynamics.

Throughout the course, we will leverage these fundamental methods—both analytical techniques (e.g., diagonalization) and numerical approaches (e.g., Euler and more sophisticated solvers)—to simulate quantum systems efficiently and accurately.

7. From Hamilton's equations to the Liouville equation in phase space

In the previous chapter we showed that many *linear* ordinary differential equations (ODEs) that appear in quantum mechanics can be solved elegantly by writing them in **matrix form** and diagonalising the matrix. In classical mechanics, however, the equations of motion

$$\begin{cases} \dot{x} &= \dfrac{p}{m}, \\ \dot{p} &= -\dfrac{\partial V(x)}{\partial x} \end{cases} \tag{7.1}$$

become *non-linear* as soon as the potential $V(x)$ is non-quadratic. Consequently the state vector $\mathbf{y} = (x, p)^{\mathsf{T}}$ no longer satisfies a linear system $\dot{\mathbf{y}} = A\,\mathbf{y}$. As an example, we will consider the Duffing-like oscillator with a quartic potential $V(x) = \frac{1}{2}kx^2 + gx^4$.

7.1. From Hamilton's equations to Liouville's continuity law

In Hamiltonian mechanics we usually track a *single* phase-space point $(x(t), p(t))$ by solving the Hamilton equations in Equation 7.1. Yet many physical questions are **statistical**:

- *Given ignorance about the exact initial state*, how does a whole *ensemble* of points evolve?
- *Which quantities remain constant* under the flow, and why?

Answering these requires an equation for a **phase-space density** $\rho(x(t), p(t), t)$, not individual trajectories. The Liouville equation supplies precisely that.

The **Liouville equation** describes how a classical probability density function in phase space evolves over time. It is a fundamental result in classical statistical mechanics and emerges directly from Hamilton's equations.

We aim to derive:

$$\frac{\partial \rho}{\partial t} + \{\rho, H\} = 0$$

where:

- $\rho(x, p, t)$ is the probability density in phase space,
- $H(x, p)$ is the Hamiltonian of the system,
- $\{f, g\} = \frac{\partial f}{\partial x}\frac{\partial g}{\partial p} - \frac{\partial f}{\partial p}\frac{\partial g}{\partial x}$ denotes the Poisson bracket.

We begin with the canonical equations of motion for a 1D system:

$$\begin{cases} \dot{x} = \frac{\partial H}{\partial p} \\ \dot{p} = -\frac{\partial H}{\partial x} \end{cases}$$

These equations describe the deterministic evolution of a point $(x(t), p(t))$ in phase space.

Let $\rho(x, p, t)$ be the density of an ensemble of classical systems in phase space. To study how this density evolves **along the flow** of the system, we compute the total derivative:

$$\frac{d}{dt}\rho(x(t), p(t), t) = \frac{\partial \rho}{\partial t} + \frac{\partial \rho}{\partial x}\frac{dx}{dt} + \frac{\partial \rho}{\partial p}\frac{dp}{dt}$$

Substituting Hamilton's equations:

$$\frac{d\rho}{dt} = \frac{\partial \rho}{\partial t} + \frac{\partial \rho}{\partial x}\frac{\partial H}{\partial p} - \frac{\partial \rho}{\partial p}\frac{\partial H}{\partial x} = \frac{\partial \rho}{\partial t} + \{\rho, H\}$$

In Hamiltonian mechanics, the phase space flow is **incompressible**: it preserves the volume element $dx \wedge dp$. This implies that the density ρ remains constant along each trajectory:

$$\frac{d}{dt}\rho(x(t), p(t), t) = 0$$

Hence, we obtain:

$$\frac{\partial \rho}{\partial t} + \{\rho, H\} = 0 \qquad \text{or} \qquad \frac{\partial \rho}{\partial t} = \{H, \rho\}$$

This is the **Liouville equation**.

7.2. Physical Interpretation

- The equation describes how a probability distribution in phase space flows under Hamiltonian evolution.
- The term $\{\rho, H\}$ encodes the flow of the distribution due to the system's dynamics.
- The total number of systems is conserved, and the phase-space density is **transported** without compression.

In short: **Liouville's theorem states that the probability density is constant along the trajectories of the system in phase space**.

7.3. Discretizing the Liouville Operator with Finite Differences

In classical statistical mechanics, the **Liouville equation** governs the time evolution of a probability density in phase space. To simulate this numerically, we can **discretize phase space** and rewrite the Liouville operator as a **sparse matrix**, using **finite difference approximations** for derivatives.

For a 1D system with Hamiltonian $H(x,p) = \frac{p^2}{2m} + V(x)$, we compute:

$$\{H, \rho\} = \frac{\partial H}{\partial x}\frac{\partial \rho}{\partial p} - \frac{\partial H}{\partial p}\frac{\partial \rho}{\partial x} = \frac{\partial V}{\partial x}\frac{\partial \rho}{\partial p} - \frac{p}{m}\frac{\partial \rho}{\partial x}$$

7.3.1. Discretizing Phase Space

We define a **uniform grid** of N_x points over x and N_p points over p:

- $x_i = x_0 + i \cdot \Delta x$, for $i = 0, \dots, N_x - 1$
- $p_j = p_0 + j \cdot \Delta p$, for $j = 0, \dots, N_p - 1$

The phase space density $\rho(x_i, p_j)$ is stored as a 2D array or flattened into a vector $\vec{\rho} \in \mathbb{R}^{N_x N_p}$.

We now define central difference matrices for the derivatives. Using second-order central differences:

$$\left.\frac{\partial \rho}{\partial x}\right|_{x_i} \approx \frac{\rho(x_{i+1}) - \rho(x_{i-1})}{2\Delta x}$$

This corresponds to a matrix $D_x \in \mathbb{R}^{N_x \times N_x}$ with the stencil:

$$D_x = \frac{1}{2\Delta x}\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ -1 & 0 & 1 & \cdots & 0 \\ 0 & -1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & 1 \\ 0 & 0 & \cdots & -1 & 0 \end{pmatrix} \tag{7.2}$$

Analogously:

$$D_p = \frac{1}{2\Delta p}\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ -1 & 0 & 1 & \cdots & 0 \\ 0 & -1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & 1 \\ 0 & 0 & \cdots & -1 & 0 \end{pmatrix} \tag{7.3}$$

Both matrices are **sparse**, **antisymmetric**, and can be constructed with sparse matrix tools of `scipy.sparse`.

7.3.2. Building the Liouville Matrix Operator

Once we flatten the 2D array $\rho(x_i, p_j)$ into a vector $\vec{\rho} \in \mathbb{R}^{N_x N_p}$, we define:

- $P = \mathrm{diag}(p_j/m)$ of shape $N_p \times N_p$
- $\partial_x V = \mathrm{diag}(\partial_x V(x_i))$ of shape $N_x \times N_x$

Then the full Liouville matrix L becomes:

$$L = (I_x \otimes D_p) \cdot (\partial_x V \otimes I_p) - (D_x \otimes I_p) \cdot (I_x \otimes P) \tag{7.4}$$

Here:

- I_x, I_p: identity matrices on position and momentum spaces
- $\otimes$: Kronecker product

This is a sparse matrix acting on $\vec{\rho}$, and encodes the total effect of the classical flow in phase space.

7.4. Time Evolution

We can evolve the discretized density using an ODE solver:

$$\frac{d\vec{\rho}}{dt} = L\vec{\rho} \tag{7.5}$$

Thus, we have reduced the problem to a linear ordinary differential equation (ODE) system, which can be solved using the standard tools discussed in Chapter 6.

Before concluding this section, let us summarize some important points:

- The Liouville operator can be expressed as a sparse matrix using finite differences.
- Position and momentum derivatives are replaced by central difference matrices.
- The discretized Liouville equation is a linear ODE system for the phase-space density vector.
- The phase space grid must be fine enough to resolve the flow.
- Boundary conditions (periodic, reflecting, absorbing) must be chosen according to the physics.
- This approach is analogous to how quantum Hamiltonians are discretized into matrices using finite differences.

7.5. Running example: a quartic non-linear oscillator

Let us keep the algebra to a minimum and pick the potential

$$V(x) = \tfrac{1}{2}kx^2 + gx^4,$$

with $k > 0$ (harmonic part) and $g > 0$ (hardening quartic term). Its Hamiltonian reads

$$H(x,p) = \frac{p^2}{2m} + V(x).$$

Although the equations of motion are non-linear, we could still integrate them numerically using `scipy.integrate.solve_ivp`. However, this goes out of the scope of this course, as we are interested in linear ordinary differential equations and their matrix equivalents. To restore linearity we have to take a step back and study **phase-space functions** rather than individual trajectories.

We can now construct the matrix operators for the Liouville equation following Equation 7.2, Equation 7.3, and Equation 7.4. We can take advantage of the tools provided by `scipy.sparse` to create the sparse matrices efficiently. We start by importing the necessary libraries

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
import scipy.sparse as sparse
import functools as ft

# Define a Gaussian function, useful for initial conditions
def gaussian(x, mu, sigma):
    """Generate a Gaussian function."""
    norm_factor = 1 / (sigma * np.sqrt(2 * np.pi))
    return np.exp(-0.5 * ((x - mu) / sigma) ** 2) * norm_factor
```

And then we define the grid and the operators for the phase space:

```python
N_x = 150 # Number of grid points in position space
N_px = 150 # Number of grid points in momentum space
x_bound = 5 # Position space boundary
px_bound = 5 # Momentum space boundary

# Identity matrices for the different dimensions
```

```
Ix = sparse.eye(N_x)
Ipx = sparse.eye(N_px)

x_list = np.linspace(-x_bound, x_bound, N_x)
px_list = np.linspace(-px_bound, px_bound, N_px)

dx = x_list[1] - x_list[0]
dpx = px_list[1] - px_list[0]

# Define the operators
x_op = sparse.diags(x_list)
px_op = sparse.diags(px_list)

# Use central differences for derivatives
d_x_op = sparse.diags([np.ones(N_x-1)/(2*dx),
                -np.ones(N_x-1)/(2*dx)], offsets=[1, -1])
d_px_op = sparse.diags([np.ones(N_px-1)/(2*dpx),
                -np.ones(N_px-1)/(2*dpx)], offsets=[1, -1])

# Create the full operator for the 4D phase space
x = ft.reduce(sparse.kron, [Ipx, x_op]).todia()
px = ft.reduce(sparse.kron, [px_op, Ix]).todia()

d_x = ft.reduce(sparse.kron, [Ipx, d_x_op]).todia()
d_px = ft.reduce(sparse.kron, [d_px_op, Ix]).todia()
```

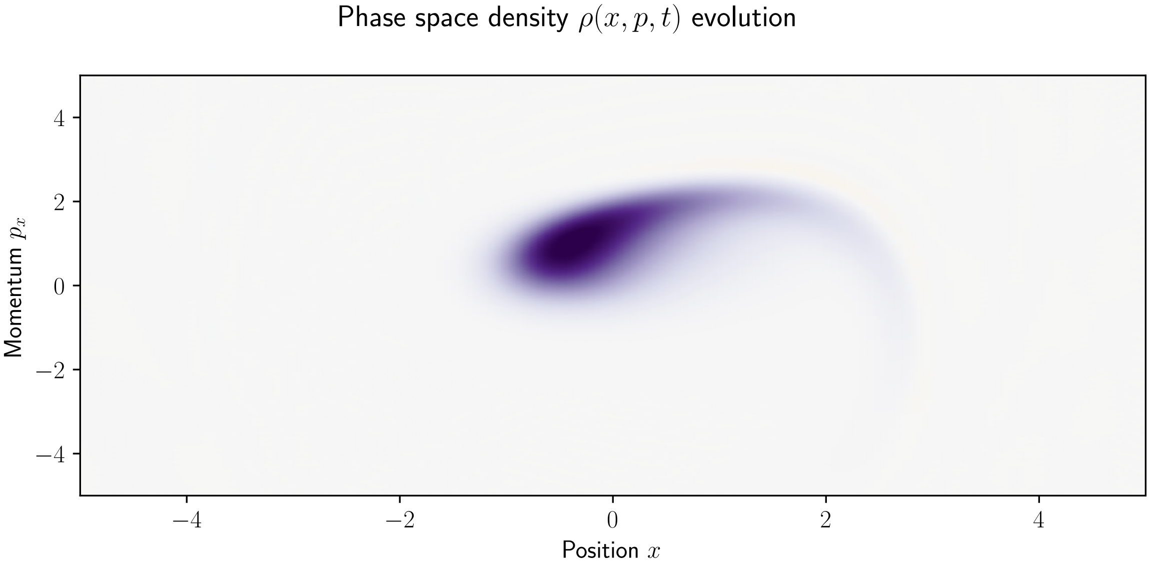
We can now compute the time evolution defined by Equation 7.5 by using the Euler method described in Section 6.3:

```
m = 0.5 # Mass of the particle
k = 2.0 # Spring constant
G = 0.15 # Nonlinear constant

dV_dx = k * x + 4 * G * x @ x @ x

# Liouville operator
L = dV_dx @ d_px - (px / m) @ d_x

t_list = np.linspace(0, 2, 100000)

# Initial state: we will use a Gaussian wave packet to avoid singularities
x_0 = gaussian(x_list, 1.0, np.sqrt(0.5))
```

```
p_0 = gaussian(px_list, 0.1, np.sqrt(0.5))
rho_0 = np.kron(p_0, x_0)

x_t = np.zeros(t_list.shape[0])
v_t = np.zeros(t_list.shape[0])
E_t = np.zeros(t_list.shape[0])

x_t[0] = rho_0.dot(x.dot(rho_0))
v_t[0] = rho_0.dot(px.dot(rho_0)) / m
E_t[0] = (0.5 * m * v_t[0]**2 + 0.5 * k * x_t[0]**2 + G * x_t[0]**4)

rho_t = [rho_0.copy()]
for i, t in enumerate(t_list[1:], 1):
    drho_dt = L @ rho_t[-1]

    # Simple Euler integration
    rho_t.append(rho_t[-1] + drho_dt * (t_list[1] - t_list[0]))

    x_t[i] = rho_t[-1].dot(x.dot(rho_t[-1]))
    v_t[i] = rho_t[-1].dot(px.dot(rho_t[-1])) / m
    E_t[i] = (0.5 * m * v_t[i]**2 + 0.5 * k * x_t[i]**2 + G * x_t[i]**4)
```

And we can visualize the final phase space density $\rho(x, p, t)$ as a 2D plot:

```
fig, ax = plt.subplots()

fig.suptitle(r"Phase space density $\rho(x, p, t)$ evolution")

img = ax.pcolormesh(x_list, px_list, rho_t[-1].reshape(N_x, N_px),
                    shading="gouraud", rasterized=True,
                    vmin=-0.3, vmax=0.3, cmap="PuOr")
ax.set_xlabel("Position $x$")
ax.set_ylabel("Momentum $p_x$")

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```



It is instructive to compare the results of this simulation with the quantum case, as shown in Appendix B. Notice how in the quantum case the Wigner (which is the quantum analogue of the phase space density) can take negative values, while in the classical case the phase space density is always non-negative.

8. Representing Quantum States and Operators with NumPy

In quantum mechanics, states and observables are represented using the algebra of Hilbert spaces. However, their infinite dimensions is incompatible with numerical simulations, that always requires finite elements. Hence, we truncate Hilbert spaces to a finite size, allowing us to run the quantum calculation on a computer. We can thus say that the whole problem of numerical quantum mechanics is then reduced to a problem of linear algebra. However, the intricated tensor structures of many-body Hilbert spaces requires also a powerful organization of the code and an easy way to access relevant information.

In the following we consider a system with a Hilbert space of dimension d. The set of basis states $\{|k\rangle : k = 1, \dots, d\}$ form an orthonormal basis, i.e., $\langle k \mid k'\rangle = \delta_{k,k'}$. In general there are systems with an infinite dimensional Hilbert space, or systems, where the dimension is too large to be tractable on a computer. In this case d denotes the number of truncated basis states, which is used in the numerical simulation. For a given choice of basis states we can express any state vector and any operator as

$$|\psi\rangle = \sum_{k=1}^{d} c_k |k\rangle, \quad \hat{A} = \sum_{k,l} A_{kl} |k\rangle\langle l|,$$

where $c_k = \langle k \mid \psi\rangle$ and $A_{kl} = \langle k|\hat{A}|l\rangle$. Therefore, in numerical simulations we represent states by vectors and operators by matrices according to the mapping

$$|\psi\rangle \mapsto \vec{\psi} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_d \end{pmatrix}, \quad \hat{A} \mapsto A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1d} \\ A_{21} & A_{22} & \dots & A_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ A_{d1} & A_{d2} & \dots & A_{dd} \end{pmatrix}.$$

The left and right operations of an operator on a vector then simply translate into matrix vector multiplications,

$$\hat{A}|\psi\rangle \mapsto \texttt{np.dot(A, psi)}, \quad \langle\psi|\hat{A} \mapsto \texttt{np.dot(np.conj(psi.T), A)},$$

where in Numpy `np.conj(psi.T)` is the hermitian transpose of a matrix or vector.

8.1. Pauli Operators

The Pauli operators are fundamental in quantum mechanics, especially in the context of qubits. They are represented as matrices in a two-dimensional Hilbert space, which is the simplest non-trivial quantum system.

$$\hat{\sigma}_x \mapsto \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \hat{\sigma}_y \mapsto \begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix}, \quad \hat{\sigma}_z \mapsto \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

In Numpy we simply define the corresponding matrices

```python
import numpy as np

sx = np.array([[0, 1], [1, 0]])
sy = np.array([[0, 1j], [-1j, 0]])
sz = np.array([[1, 0], [0, -1]])
```

8.2. Spin S Systems

A system with a fixed total spin S is described by the three collective spin operators S_x, S_y, S_z, which obey $\left[S_x, S_y\right] = iS_z$. The $d = 2S + 1$ basis states can be labeled as $|s, m\rangle$, where $m = S, S-1, \ldots, -S$. By introducing spin raising and lowering operators $S_\pm = S_x \pm iS_y$, where $S_-^\dagger = S_+$, all matrix elements can be obtained from the two relations

$$S_z|s,m\rangle = m|s,m\rangle, \quad S_-|s,m\rangle = \sqrt{S(S+1) - m(m-1)}|s, m-1\rangle =: s_-^m|s, m-1\rangle\,.$$

Explicitly,

$$\hat{S}_z \mapsto SZ = \begin{pmatrix} S & 0 & \ldots & 0 \\ 0 & S-1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & -S \end{pmatrix}, \quad \hat{S}_- \mapsto SM = \begin{pmatrix} 0 & 0 & \ldots & 0 \\ s_-^{S/2} & 0 & \ldots & 0 \\ 0 & s_-^{S/2-1} & \ddots & \vdots \\ \vdots & 0 & \ldots & 0 \end{pmatrix}.$$

All other operators can be obtained using $S_x = \left(S_+ + S_-\right)/2$ and $S_y = i\left(S_+ - S_-\right)/2$. Example for a spin $S = 1$ system

```
s = 1
d = int(2 * s + 1)

# Vector of the diagonal elements for the SZ operator
vecm = np.flip(np.arange(-s, s + 1))

SZ = np.diag(vecm, 0)

# Vector for SM operator (ladder down)
vec2 = vecm[:d-1]  # remove the last entry
vec3 = np.sqrt(s * (s + 1) - vec2 * (vec2 - 1))
SM = np.diag(vec3, k=-1)  # place vec3 in lower diagonal

# Construct SX and SY
SX = (SM + SM.T) / 2
SY = 1j * (SM.T - SM) / 2
```

Note that in Numpy the command `A * B` is a element-wise multiplication of two matrices, while `A @ B` would implement the usual matrix multiplication.

8.3. Harmonic Oscillator

In a Hilbert space of dimension N, quantum states can be represented as vectors, and operators as matrices. Here we demonstrate the destroy operator, a, which lowers the state by one quantum number. For a detailed discussion on the quantum harmonic oscillator and the bosonic annihilation operator, refer to Appendix A.

For a harmonic oscillator with number states $|n\rangle$ the only nonzero matrix elements of the annihilation operator $\hat{a}$ are given by $\langle n-1|\hat{a}|n\rangle = \sqrt{n}$

$$\hat{a} \mapsto A = \begin{pmatrix} 0 & 1 & \dots & \cdots & \dots & 0 \\ 0 & 0 & \sqrt{2} & \dots & \cdots & 0 \\ 0 & 0 & 0 & \sqrt{3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & 0 & \sqrt{d-1} \\ 0 & 0 & 0 & \cdots & \cdots & 0 \end{pmatrix}$$

This operator acts on Fock states to lower their quantum number by one, with a factor of $\sqrt{n}$, where n is the quantum number of the initial state. In other words, $\hat{a}|n\rangle = \sqrt{n}|n-1\rangle$.

In the following code, we define the destroy operator by using NumPy, and we also define some Fock states for demonstration.

In Numpy we use the command `np.diag(v, k=r)`, which creates a diagonal matrix with the elements of the vector `v` placed in the r-th diagonal ($r = 0, \pm 1, \pm 2, ...$).

```python
def destroy(d):
    # creates a vector of the d-1 off-diagonal elements
    v=np.sqrt( np.arange(d-1) )
    # matrix with the elements of vec placed in the upper diagonal
    a=np.diag(v,k=1)
    return a

# Define the fock states
def fock(d, i):
    res = np.zeros(d)
    res[i] = 1
    return res

d = 7
zero_state = fock(d, 0)
one_state = fock(d, 1)
two_state = fock(d, 2)
three_state = fock(d, 3)

destroy_operator = destroy(d)
destroy_operator
```

```
array([[0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        ],
       [0.        , 0.        , 1.        , 0.        , 0.        ,
        0.        , 0.        ],
       [0.        , 0.        , 0.        , 1.41421356, 0.        ,
        0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 1.73205081,
        0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ,
        2.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 2.23606798],
       [0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        ]])
```

Other operators (e.g., $\hat{a}^\dagger, \hat{a}^\dagger\hat{a}$) can be obtained by a hermitian transpose

$$\hat{a}^\dagger \mapsto \texttt{np.conj(a.T)}$$

and matrix multiplications

$$\hat{a}^\dagger\hat{a} \mapsto \texttt{np.matmul(np.conj(a.T) , a)}\,.$$

Note that in some cases this introduces truncation artifacts. For example, the matrix for the operator $M =$ `np.matmul(a, np.conj(a.T))` has a zero diagonal element $M[d, d] = 0$ inherited from the matrix `np.conj(a.T))`, while the same operator constructed in a different way, $M_2 =$ `np.conj(a.T) * a + np.eye(d)`, does not. This can be avoided by constructing this operator explicitly. Note that this type of truncation artifacts are related to the fact that in a infinite Hilbert space $\mathrm{Tr}([a, a^\dagger]) \neq 0$ (actually, striclty speaking, $= \infty$) as a consequence of the canonical commutation relation. On the contrary, in a finite Hilbert space for any two operators O_1, O_2, $\mathrm{Tr}([O_1, O_2]) = 0$. Taking a dimension d large enough allows to make these artifacts a negligible error in the whole computation.

8.3.1. Action of the Destroy Operator on a Fock State

The action of the destroy operator a on a Fock state $|n\rangle$ lowers the state by one quantum number, multiplied by a factor $\sqrt{n}$. For example, applying a to the state $|3\rangle$ yields:

$$\hat{a}|3\rangle = \sqrt{3}|2\rangle$$

This demonstrates the lowering action of the destroy operator with a specific factor, dependent on the quantum number of the state being acted upon.

```python
# Apply the destroy operator on the one state
result_state = np.dot(destroy_operator, three_state)

print("Resulting State:")
result_state
```

```
Resulting State:
```

```
array([0.        , 0.        , 1.41421356, 0.        , 0.        ,
       0.        , 0.        ])
```

8.4. Partial Trace

In Section 4.1.4, we have already discussed the concept of tensor products. Here we will introduce the **partial trace**, a crucial operation in quantum mechanics that allows us to focus on a subsystem of a larger composite system.

The **partial trace** over a subsystem, say B, of a composite system AB, mathematically expresses as "tracing out" B, leaving the reduced state of A. For a bipartite state ρ_{AB}, the partial trace over B is:

$$\mathrm{Tr}_B(\hat{\rho}_{AB}) = \sum_{i \in \mathcal{H}_B} \langle i|\hat{\rho}_{AB}|i\rangle$$

where $\{|i\rangle\}$ forms a complete basis for subsystem B.

Let's try it with an entangled Bell's state between two qubits:

$$|\phi^+\rangle = \frac{1}{\sqrt{2}}\left(|0,0\rangle + |1,1\rangle\right)$$

```
def ptrace(psi, subspace_to_keep, dim_subspace):
    dim1, dim2 = dim_subspace

    rho = np.outer(psi, psi.conj())

    # Reshape rho to separate the subsystems' degrees of freedom
    rho_reshaped = rho.reshape(dim1, dim2, dim1, dim2)

    if subspace_to_keep == 1:
        # Perform the trace over the second subsystem
        traced_out = np.trace(rho_reshaped, axis1=1, axis2=3)
    elif subspace_to_keep == 2:
        # Perform the trace over the first subsystem
        traced_out = np.trace(rho_reshaped, axis1=0, axis2=2)
    else:
        raise ValueError("subspace_to_keep must be either 1 or 2.")

    return traced_out

# Bell state between two qubits
phi_plus = ( np.kron(fock(2, 1), fock(2, 1)) + np.kron(fock(2, 0), fock(2, 0)) ) / np.sqrt(2

# Reduced density matrix of the first qubit
```

```
rho_1 = ptrace(phi_plus, 1, (2, 2))
rho_1
```

```
array([[0.5, 0. ],
       [0. , 0.5]])
```

8.5. Why QuTiP?

While NumPy and SciPy are powerful tools for numerical computations, they lack specific functionalities for efficiently handling complex quantum systems. QuTiP is designed to fill this gap, offering features such as:

- Easy manipulation and visualization of quantum objects.
- Support for operations on states and operators in different Hilbert spaces.
- Tools for dealing with composite systems, partial traces, and superoperators. It is like to have the book “Quantum noise” (by Gardiner and Zoller) already implemented in your laptop!

In the next chapters, we’ll explore how QuTiP simplifies these tasks, making it an invaluable tool for quantum optics simulations.

9. Introduction to QuTiP

The QuTiP package can be imported with

```
import qutip
import numpy as np
```

It can also be imported with the command `from qutip import *`, that automatically imports all the QuTiP functions. However, here we use the first method, in order to explicitly see the QuTiP functions.

```
qutip.about()
```

```
QuTiP: Quantum Toolbox in Python
================================
Copyright (c) QuTiP team 2011 and later.
Current admin team: Alexander Pitchford, Nathan Shammah, Shahnawaz Ahmed, Neill Lambert, Eri
Board members: Daniel Burgarth, Robert Johansson, Anton F. Kockum, Franco Nori and Will Zeng
Original developers: R. J. Johansson & P. D. Nation.
Previous lead developers: Chris Granade & A. Grimsmo.
Currently developed through wide collaboration. See https://github.com/qutip for details.

QuTiP Version:      5.1.1
Numpy Version:      2.2.6
Scipy Version:      1.15.3
Cython Version:     3.1.1
Matplotlib Version: 3.10.3
Python Version:     3.13.4
Number of CPUs:     4
BLAS Info:          Generic
INTEL MKL Ext:      None
Platform Info:      Linux (x86_64)
Installation path:  /opt/hostedtoolcache/Python/3.13.4/x64/lib/python3.13/site-packages/quti
```

```
Installed QuTiP family packages
-------------------------------

No QuTiP family packages installed.

==============================================================================
Please cite QuTiP in your publication.
==============================================================================
For your convenience a bibtex reference can be easily generated using `qutip.cite()`
```

9.1. Quantum Operators

Quantum operators play a crucial role in the formulation of quantum mechanics, representing physical observables and operations on quantum states. In QuTiP, operators are represented as `Qobj` instances, just like quantum states. This section introduces the creation and manipulation of quantum operators.

9.1.1. Creating Operators

Operators in quantum mechanics can represent measurements, such as position or momentum, and transformations, such as rotation. Let's see how we can define some common operators in QuTiP.

9.1.1.1. The Annihilation Operator of the Quantum Harmonic oscillator

The harmonic oscillator is a fundamental model in quantum mechanics for understanding various physical systems. Its quantization leads to the concept of creation and annihilation operators, which respectively increase and decrease the energy of the system by one quantum of energy.

The annihilation operator, often denoted by $\hat{a}$, acts on a quantum state to reduce its quantum number. The action of $\hat{a}$ on a state $|n\rangle$ is defined as:

$$\hat{a}|n\rangle = \sqrt{n}|n-1\rangle$$

Here, $|n\rangle$ represents a quantum state with n quanta of energy (also known as a Fock state), and $\sqrt{n}$ is the normalization factor. The matrix representation of the annihilation operator in an d-dimensional Hilbert space is given by an upper triangular matrix with the square roots of natural numbers as its off-diagonal elements.

```
# Define the annihilation operator for d-dimensional Hilbert space
d = 7

a = qutip.destroy(d)

print("Annihilation operator (a) for d=7:")
a
```

```
Annihilation operator (a) for d=7:
```

```
Quantum object: dims=[[7], [7]], shape=(7, 7), type='oper', dtype=Dia, isherm=False
Qobj data =
[[0.         1.         0.         0.         0.         0.
  0.        ]
 [0.         0.         1.41421356 0.         0.         0.
  0.        ]
 [0.         0.         0.         1.73205081 0.         0.
  0.        ]
 [0.         0.         0.         0.         2.         0.
  0.        ]
 [0.         0.         0.         0.         0.         2.23606798
  0.        ]
 [0.         0.         0.         0.         0.         0.
  2.44948974]
 [0.         0.         0.         0.         0.         0.
  0.        ]]
```

9.1.1.2. Pauli Matrices

The Pauli matrices are fundamental in the study of quantum mechanics, representing the spin operators for a spin-1/2 particle and quantum two-level systems.

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \; \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \; \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

We can define these matrices in QuTiP as follows:

```
sigma_x = qutip.sigmax()
sigma_y = qutip.sigmay()
sigma_z = qutip.sigmaz()

print("Sigma X:")
display(sigma_x)
print("\n")
print("Sigma Y:")
display(sigma_y)
print("\n")
print("Sigma Z:")
sigma_z
```

```
Sigma X:
```

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

```
Sigma Y:
```

```
Quantum object: dims=[[2], [2]], shape=(2, 2), type='oper', dtype=CSR, isherm=True
Qobj data =
[[0.+0.j 0.-1.j]
 [0.+1.j 0.+0.j]]
```

```
Sigma Z:
```

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

9.1.2. Operator Functions and Operations

QuTiP supports various operations on operators, including addition, multiplication (both scalar and matrix), and the commutator. These operations are essential for constructing Hamiltonians, calculating observables, and more.

9.1.2.1. Example: Commutator of Pauli Matrices

The commutator of two operators A and B is defined as $[A, B] = AB - BA$. Let's calculate the commutator of σ_x and σ_y.

```
commutator_xy = qutip.commutator(sigma_x, sigma_y)
print("Commutator of Sigma X and Sigma Y:")
display(commutator_xy)
commutator_xy == 2j * sigma_z
```

```
Commutator of Sigma X and Sigma Y:
```

```
Quantum object: dims=[[2], [2]], shape=(2, 2), type='oper', dtype=CSR, isherm=False
Qobj data =
[[0.+2.j 0.+0.j]
 [0.+0.j 0.-2.j]]
```

```
True
```

9.2. Quantum States

Quantum states describe the state of a quantum system. In QuTiP, states are represented again as `Qobj` instances. This section focuses on the representation and manipulation of quantum states.

9.2.1. Fock States

The most basic quantum states are the fock states, often denoted as $|n\rangle$ (with $n \in \mathbb{N}$). Let's see how we can create these in QuTiP.

9.2.2. Superposition States

Quantum mechanics allows particles to be in a superposition of states. Let's create a superposition state.

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)$$

```
fock_0 = qutip.fock(d, 0)  # Fock state |0>
fock_1 = qutip.fock(d, 1)  # Fock state |1>

# Creating a superposition state
superposition_state = (fock_0 + fock_1).unit()  # Normalize the state

print("Superposition state:")
superposition_state
```

```
Superposition state:


Quantum object: dims=[[7], [1]], shape=(7, 1), type='ket', dtype=Dense
Qobj data =
[[0.70710678]
 [0.70710678]
 [0.        ]
 [0.        ]
 [0.        ]
 [0.        ]
 [0.        ]]
```

9.2.3. Coherent States

Coherent states in QuTiP represent quantum states closest to classical waves, defined as

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

with minimal uncertainty.

The coherent state is an eigenstate of the annihilation operator

$$\hat{a}|\alpha\rangle = \alpha|\alpha\rangle$$

Warning!

Remember that every `Qobj` lives in a truncated Hilbert space. If the α value is too large, the state will become a non-physical state because it will touch the high energy levels of the truncated Hilbert space.

```
alpha = 0.8
coherent_state = qutip.coherent(d, alpha)

coherent_state
```

```
Quantum object: dims=[[7], [1]], shape=(7, 1), type='ket', dtype=Dense
Qobj data =
[[0.72614904]
 [0.58091919]
 [0.32861796]
 [0.1517784 ]
 [0.06073653]
 [0.02159193]
 [0.00767346]]
```

Let's compute the fidelity between $|\alpha\rangle$ and $\hat{a}|\alpha\rangle/\alpha$.

```
qutip.fidelity(a * coherent_state / alpha, coherent_state)
```

```
np.float64(0.9999661194274998)
```

9.2.4. Spin States

```
qutip.spin_state(0.5, -1)
```

```
Quantum object: dims=[[2], [1]], shape=(2, 1), type='ket', dtype=Dense
Qobj data =
[[0.]
 [1.]]
```

9.2.5. Density Matrices

Quantum states can also be represented using density matrices, which are useful for describing mixed states.

9.2.5.1. Creating a Density Matrix

Let’s convert our superposition state into a density matrix.

```python
# Creating a density matrix from a state
density_matrix = superposition_state * superposition_state.dag()  # Outer product

print("Density matrix of the superposition state:")
density_matrix
```

```
Density matrix of the superposition state:
```

```
Quantum object: dims=[[7], [7]], shape=(7, 7), type='oper', dtype=Dense, isherm=True
Qobj data =
[[0.5 0.5 0.  0.  0.  0.  0. ]
 [0.5 0.5 0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0. ]]
```

9.2.6. Partial Trace

The **partial trace** over a subsystem, say B, of a composite system AB, mathematically expresses as “tracing out” B, leaving the reduced state of A. For a bipartite state ρ_{AB}, the partial trace over B is:

$$\mathrm{Tr}_B(\hat{\rho}_{AB}) = \sum_{i\in\mathcal{H}_B} \langle i|\hat{\rho}_{AB}|i\rangle$$

where $\{|i\rangle\}$ forms a complete basis for subsystem B.

Let’s try it with an entangled Bell’s state between two qubits:

$$|\phi^+\rangle = \frac{1}{\sqrt{2}}\left(|0,0\rangle + |1,1\rangle\right)$$

```
# Bell state between two qubits
phi_plus = ( qutip.tensor(qutip.spin_state(1/2, -1), qutip.spin_state(1/2, -1)) + qutip.tens

# Reduced density matrix of the first qubit
rho_1 = qutip.ptrace(phi_plus, 1)
rho_1
```

```
Quantum object: dims=[[2], [2]], shape=(2, 2), type='oper', dtype=Dense, isherm=True
Qobj data =
[[0.5 0. ]
 [0.  0.5]]
```

We now apply the partial trace to a more complicated state, that is composed by two bosonic modes and two spins $|j_1, m_1\rangle$ and $|j_2, m_2\rangle$, with $j_1 = 1$ and $j_2 = \frac{1}{2}$, $m_1 = 0$, and $m_2 = 1$.

```
j1 = 1
j2 = 1/2
m1 = 0
m2 = 1



psi = qutip.tensor(qutip.fock(d, 3), qutip.fock(d, 1), qutip.spin_state(j1, 0), qutip.spin_s

# Trace only the second spin state
rho_0 = qutip.ptrace(psi, [0, 1, 2])
display(rho_0)

# Trace only the first bosonic mode and the second spin state
rho_1 = qutip.ptrace(psi, [1, 2])
display(rho_1)

# Trace all except the second bosonic mode
rho_2 = qutip.ptrace(psi, [1])
rho_2
```

```
Quantum object: dims=[[7, 7, 3], [7, 7, 3]], shape=(147, 147), type='oper', dtype=Dense, ish
Qobj data =
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]


Quantum object: dims=[[7, 3], [7, 3]], shape=(21, 21), type='oper', dtype=Dense, isherm=True
Qobj data =
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]


Quantum object: dims=[[7], [7]], shape=(7, 7), type='oper', dtype=Dense, isherm=True
Qobj data =
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
```

```
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
```

9.3. Eigenstates and Eigenvalues

The eigenstates and eigenvalues of a system or an operator provide crucial insights into its properties. Let's explore how to calculate these in QuTiP.

```
# Example: Eigenstates and eigenvalues of Pauli Z
eigenvalues, eigenstates = sigma_z.eigenstates()

print("Eigenvalues of Sigma Z:")
display(eigenvalues)
print("\n")
print("Eigenstates of Sigma Z:")
display(eigenstates)
```

```
Eigenvalues of Sigma Z:
```

```
array([-1.,  1.])
```

```
Eigenstates of Sigma Z:
```

```
array([Quantum object: dims=[[2], [1]], shape=(2, 1), type='ket', dtype=Dense
       Qobj data =
       [[ 0.]
        [-1.]]                                                                  ,
       Quantum object: dims=[[2], [1]], shape=(2, 1), type='ket', dtype=Dense
       Qobj data =
       [[-1.]
        [-0.]]                                                                  ],
      dtype=object)
```

9.4. Computing Expectation Values

The expectation value of an operator provides insight into the average outcome of a quantum measurement. For a quantum state $|\psi\rangle$ and an operator $\hat{O}$, the expectation value is given by:

$$\langle\hat{O}\rangle = \langle\psi|\hat{O}|\psi\rangle$$

Expectation values are crucial for predicting measurable quantities in quantum mechanics. Let's compute the expectation value of the number operator $\hat{n} = \hat{a}^\dagger\hat{a}$ for a coherent state, which represents a quantum state closest to a classical harmonic oscillator.

```
# Define the coherent state |psi> with alpha=2
alpha = 0.8
psi = qutip.coherent(d, alpha)

# Define the number operator n = a.dag() * a
n = a.dag() * a

# Compute the expectation value of n for the state |psi>
expectation_value_n = qutip.expect(n, psi)

print("Expectation value of the number operator for |psi>:")
display(expectation_value_n)
print("\n")
print("The squared modulus of alpha is:")
display(abs(alpha) ** 2)
```

```
Expectation value of the number operator for |psi>:


0.639996733025295




The squared modulus of alpha is:


0.6400000000000001
```

9.5. A complete example: the Quantum Harmonic Oscillator

In Appendix A we have already defined the quantum harmonic oscillator, which is a fundamental model in quantum mechanics. In the energy eigenbasis, the quantum harmonic oscillator is described by the Hamiltonian

$$\hat{H} = \omega \hat{a}^\dagger \hat{a} \,,$$

where ω is the resonance frequency and $\hat{a}$ is the bosonic annihilation operator. In this basis, the Hamiltonian is indeed diagonal, with eigenvalues $E_n = \omega n$, where $n \in \mathbb{N}$.

```
N = 120 # Hilbert space cutoff
w = 1 # Resonance frequency of the harmonic oscillator

a = qutip.destroy(N)

H = w * a.dag() * a

H
```

```
Quantum object: dims=[[120], [120]], shape=(120, 120), type='oper', dtype=Dia, isherm=True
Qobj data =
[[  0.   0.   0. ...   0.   0.   0.]
 [  0.   1.   0. ...   0.   0.   0.]
 [  0.   0.   2. ...   0.   0.   0.]
 ...
 [  0.   0.   0. ... 117.   0.   0.]
 [  0.   0.   0. ...   0. 118.   0.]
 [  0.   0.   0. ...   0.   0. 119.]]
```

9.6. Passing in the position basis

From the classical point of view, we are used to describe the harmonic oscillator in terms of position and momentum. In quantum mechanics, we can also express the system in terms of the position and momentum operators, which are related to the annihilation and creation operators as follows ($\hbar = 1$):

$$\hat{x} = \frac{1}{\sqrt{2m\omega}} \left(\hat{a} + \hat{a}^\dagger\right) ,$$
$$\hat{p} = i\sqrt{\frac{m\omega}{2}} \left(\hat{a}^\dagger - \hat{a}\right) .$$

We first check that

$$[\hat{x}, \hat{p}] = i$$

```
m = 0.5

x = (a + a.dag()) / np.sqrt(2 * m * w)
p = - 1j * (a - a.dag()) * np.sqrt(m * w / 2)

qutip.commutator(x, p)
```

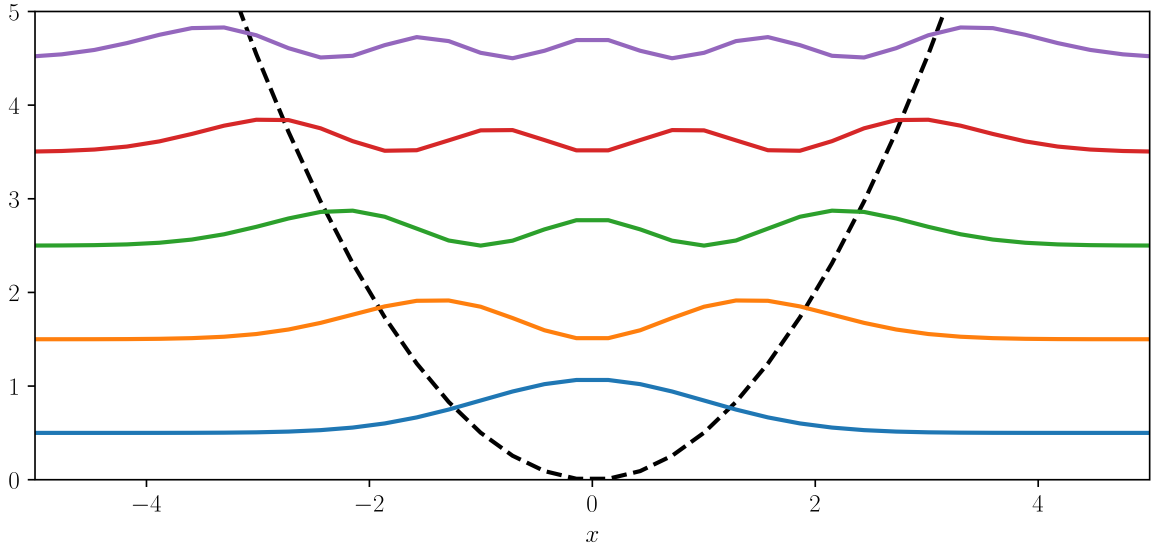
```
Quantum object: dims=[[120], [120]], shape=(120, 120), type='oper', dtype=Dia, isherm=False
Qobj data =
[[0.  +1.j 0.  +0.j 0.  +0.j ... 0.  +0.j 0.  +0.j 0.  +0.j]
 [0.  +0.j 0.  +1.j 0.  +0.j ... 0.  +0.j 0.  +0.j 0.  +0.j]
 [0.  +0.j 0.  +0.j 0.  +1.j ... 0.  +0.j 0.  +0.j 0.  +0.j]
 ...
 [0.  +0.j 0.  +0.j 0.  +0.j ... 0.  +1.j 0.  +0.j 0.  +0.j]
 [0.  +0.j 0.  +0.j 0.  +0.j ... 0.  +0.j 0.  +1.j 0.  +0.j]
 [0.  +0.j 0.  +0.j 0.  +0.j ... 0.  +0.j 0.  +0.j 0.-119.j]]
```

We now numerically diagonalize the position operator $\hat{x}$, such that $\hat{U}^\dagger \hat{x} \hat{U}$ is diagonal. Then we plot the eigenstates of the quantum harmonic oscillator in the new basis, obtained with

$$|\phi_n(x)\rangle = \hat{U}^\dagger |\psi_n\rangle$$

```
import matplotlib.pyplot as plt

E, T = x.eigenstates()

U = np.zeros((N, N)).astype(np.complex128)
for i in range(N):
    U[:,i] = T[i].full().flatten()
```

```
U = qutip.Qobj(U)

xlist = ( U.dag() * x * U ).diag()
dx = xlist[1] - xlist[0]

# Harmonic potential
V = w**2 / 2 * xlist**2

fig, ax = plt.subplots()

ax.plot(xlist, V, color="black", ls="--", lw=2)
for i in range(5):
    factor = 5 # The purpose of this factor is to only make more visible the states
    ax.plot(xlist, factor * np.abs( (U.dag() * qutip.fock(N, i)).full() )**2 + i * w + w/2,

ax.set_xlabel(r"$x$")
ax.set_xlim(-5, 5)
ax.set_ylim(0, 5)

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```



10. Closed Quantum Systems in QuTiP: The Schrödinger equation

```
import numpy as np
import matplotlib.pyplot as plt
from qutip import *
```

10.1. From wavefunctions to the Schrödinger equation in configuration space

In Chapter 7 we moved from **Hamilton's equations** for a single point to **Liouville's equation** for an ensemble. Quantum mechanics follows a similar path, replacing phase-space densities with **wavefunctions** (or density operators). We now show this parallelism in detail.

10.1.1. From a single wavepacket to the quantum continuity law

A non-relativistic quantum system in one dimension is described by a complex wavefunction $\psi(x,t)$. Its **probability density** and **current** are

$$\rho(x,t) \;=\; |\psi(x,t)|^2, \qquad j(x,t) \;=\; \frac{\hbar}{m}\,\Im[\psi^*(x,t)\,\partial_x\psi(x,t)].$$

Starting from the **time-dependent Schrödinger equation**

$$i\hbar\,\partial_t\psi(x,t) \;=\; \hat{H}\,\psi(x,t) \quad \text{with} \quad \hat{H} \;=\; -\frac{\hbar^2}{2m}\,\partial_x^2 + V(x) \tag{10.1}$$

and its complex conjugate, a short calculation yields the **continuity equation**

$$\partial_t\rho + \partial_x j \;=\; 0, \tag{10.2}$$

which plays the same role as Liouville's incompressibility condition: probability is **transported** in configuration space without being created or destroyed.

10.1.2. The Schrödinger equation as a linear ODE in Hilbert space

While Equation 10.1 is a partial differential equation, it is *linear* in ψ, exactly like the matrix ODE $\dot{\mathbf{y}} = A\mathbf{y}$ used for classical linear systems. Written in Dirac notation

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

the resemblance becomes explicit: **replace** the classical matrix A by the quantum Hamiltonian operator $\hat{H}$ and the state vector $\mathbf{y}$ by a ket $|\psi\rangle$.

For statistical mixtures one introduces the **density operator** $\hat{\rho}$ and obtains the **von Neumann equation**

$$i\hbar\, \partial_t \hat{\rho} \;=\; [\hat{H},\, \hat{\rho}], \tag{10.3}$$

a direct quantum counterpart of the Liouville equation in Equation 7.5.

With these ingredients we now possess a one-to-one map between the **classical Liouville formulation** and the **quantum Schrödinger formulation**, both expressible as sparse linear ODEs ready for numerical treatment. Let's consider a simple example of a harmonic oscillator, which is described by the Hamiltonian

$$\hat{H} = \hbar\omega \hat{a}^\dagger \hat{a}\,,$$

where $\hat{a}$ and $\hat{a}^\dagger$ are the annihilation and creation operators, respectively, and ω is the angular frequency of the oscillator.

```
N = 120 # Number of Fock states
w = 1 # Angular frequency of the oscillator

a = destroy(N) # Annihilation operator
H = w * a.dag() * a # Hamiltonian of the harmonic oscillator

H
```

```
Quantum object: dims=[[120], [120]], shape=(120, 120), type='oper', dtype=Dia, isherm=True
Qobj data =
[[  0.   0.   0. ...   0.   0.   0.]
 [  0.   1.   0. ...   0.   0.   0.]
 [  0.   0.   2. ...   0.   0.   0.]
```

```
 ...
 [  0.    0.    0. ... 117.    0.    0.]
 [  0.    0.    0. ...   0. 118.    0.]
 [  0.    0.    0. ...   0.    0. 119.]]
```

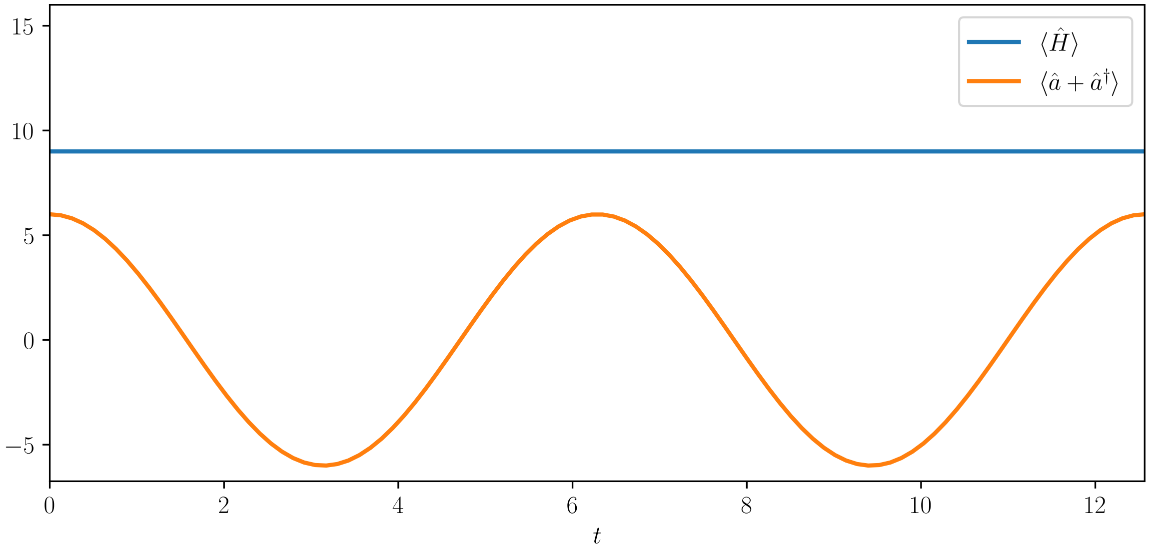
In QuTip, the Schrödinger equation is solved with the function `sesolve`, which stands for "Schrödinger equation solver".

```
alpha = 3 # Coherence of the initial state

# We start from a coherent state the most classic-like state
psi0 = coherent(N, alpha)

# List of the times for the time evolution
tlist = np.linspace(0, 2 * 2*np.pi/w, 100)

e_ops = [H, a + a.dag()]

sol = sesolve(H, psi0, tlist, e_ops=e_ops)
```

10.2. Plot the expectation values

We can access to the expectation values with the command `sol.expect[i]`, where `i` is the index of the i-th operator for which we want to calculate te expectation value as a function of time.

```
fig, ax = plt.subplots()

ax.plot(tlist, sol.expect[0], label=r"$\langle \hat{H} \rangle$", lw=2)
ax.plot(tlist, sol.expect[1], label=r"$\langle \hat{a} + \hat{a}^\dagger \rangle$", lw=2)
ax.legend()
ax.set_xlabel(r"$t$")
ax.set_xlim(tlist[0], tlist[-1])
ax.set_ylim(None, 16)

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```



10.3. Access directly to the evolution of the state

We can also access directly to the wavefunction at each `tlist`. This can be done by simply calling `sol.states` to the solver without the `e_ops` operators.

To check this feature, let's see if after 10 cycles we get still the initial state, by calculating the fidelity

$$\langle \psi \left(t = 10T \right) | \psi \left(t = 0 \right) \rangle$$

where $T = \frac{2\pi}{\omega}$.

```
sol = sesolve(H, psi0, tlist)

sol.states[0].dag() * sol.states[-1]
```

```
(0.999999999991156-6.853062359472892e-07j)
```

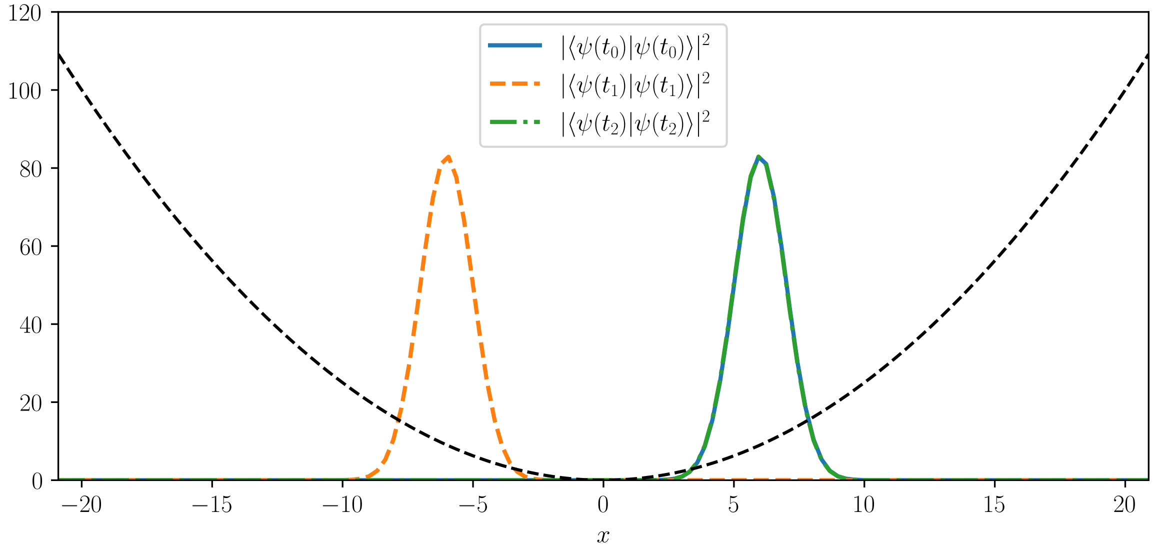
We then switch to the position eigenbasis. Thus, we need to diagonalize the position operator. This in general involves the *Laguerre* functions, but here we limit ourselves to numerically diagonalize the position operator.

We now show the wavefunctions in the position basis at three different times: t_0, t_1 and t_2, with

$$t_0 = 0 \, , \quad t_1 = \frac{\pi}{\omega} \, , \quad t_2 = \frac{2\pi}{\omega} \, ,$$

showing the exact periodicity of this system.

```python
idx_t0 = 0
idx_t1 = np.where(tlist >= np.pi/w)[0][0]
idx_t2 = np.where(tlist >= 2*np.pi/w)[0][0]

psi0_x = U.dag() * sol.states[idx_t0]
psi1_x = U.dag() * sol.states[idx_t1]
psi2_x = U.dag() * sol.states[idx_t2]

# We define the potential for the harmonic oscillator
V = 0.5 * m * w**2 * xlist**2

fig, ax = plt.subplots()

ax.plot(xlist, 700 * np.abs(psi0_x.full())**2,
        label=r"$\vert \langle \psi (t_0) \vert \psi (t_0) \rangle \vert^2$", lw=2)
ax.plot(xlist, 700 * np.abs(psi1_x.full())**2,
        label=r"$\vert \langle \psi (t_1) \vert \psi (t_1) \rangle \vert^2$", ls="--", lw=2)
ax.plot(xlist, 700 * np.abs(psi2_x.full())**2,
        label=r"$\vert \langle \psi (t_2) \vert \psi (t_2) \rangle \vert^2$", ls="-.", lw=2)
ax.plot(xlist, V, color="black", ls="--")
ax.legend()
ax.set_xlabel(r"$x$")
ax.set_xlim(xlist[0], xlist[-1])
ax.set_ylim(0, 120)

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```



We can also export an animation, showing more easily the time evolution of the state

```
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

plt.rcParams.update({'font.size': 8})
fig, ax = plt.subplots(figsize=(4.6, 2.8))

line, = ax.plot(xlist, 700 * np.abs(psi0_x.full())**2, lw=2)
ax.plot(xlist, V, color="black", ls="--")
ax.set_xlabel(r"$x$")
ax.set_xlim(xlist[0], xlist[-1])
ax.set_ylim(0, 120)

plt.close(fig) # Otherwise the static figure also appears

def update(frame):
    psi_t = U.dag() * sol.states[frame]
    line.set_ydata( 700 * np.abs(psi_t.full())**2 )
    return line,

fps = 25
ani = FuncAnimation(fig, update, frames=len(tlist), blit=True, interval=1000 / fps)

HTML(ani.to_jshtml())
```

```
Unable to display output for mime type(s): text/html
```

11. Open Dynamics in QuTiP: The Master Equation

In Chapter 10, we discussed the evolution of closed quantum systems, where the dynamics is governed by the Schrödinger equation. The evolution of the state vector $|\psi(t)\rangle$ is given by:

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

where $\hat{H}$ is the Hamiltonian operator of the system. The solution to this equation leads to unitary evolution. In contrast to the closed case, open quantum systems interact with their environment, leading to non-unitary evolution described by the Master equation:

$$\frac{d\hat{\rho}}{dt} = -\frac{i}{\hbar}[\hat{H},\hat{\rho}] + \sum_k \left(\hat{L}_k\hat{\rho}\hat{L}_k^\dagger - \frac{1}{2}\{\hat{L}_k^\dagger\hat{L}_k,\hat{\rho}\}\right)$$

Here, $\hat{\rho}$ represents the density matrix of the system, $\hat{L}_k$ are the Lindblad operators representing different dissipation processes. and $\{\hat{A},\hat{B}\} = \hat{A}\hat{B} + \hat{B}\hat{A}$ is the anti-commutator between the operators $\hat{A}$ and $\hat{B}$.

11.1. Example: the damped harmonic oscillator

We can use the Master equation to study the dynamics of a damped harmonic oscillator. The Hamiltonian for a harmonic oscillator is given by:

$$\hat{H} = \omega_0\hat{a}^\dagger\hat{a}$$

where $\hat{a}$ and $\hat{a}^\dagger$ are the annihilation and creation operators, respectively, and ω_0 is the angular frequency of the oscillator.

```
import numpy as np
from qutip import *

# Parameters
N = 30  # Number of Fock states
omega_0 = 1.0  # Angular frequency of the oscillator

a = destroy(N)  # Annihilation operator

# Hamiltonian
H = omega_0 * a.dag() * a
```

To model the damping, we can introduce a Lindblad operator that represents the interaction with the environment. In the case of the interaction with a zero-temperature bath, the Lindblad operator can be defined as:

$$\hat{L} = \sqrt{\gamma}\hat{a}$$

where γ is the damping rate. Let's now initialize the system in a coherent state and evolve it using the Master equation. We will also visualize the evolution of the position and energy expectation values over time.

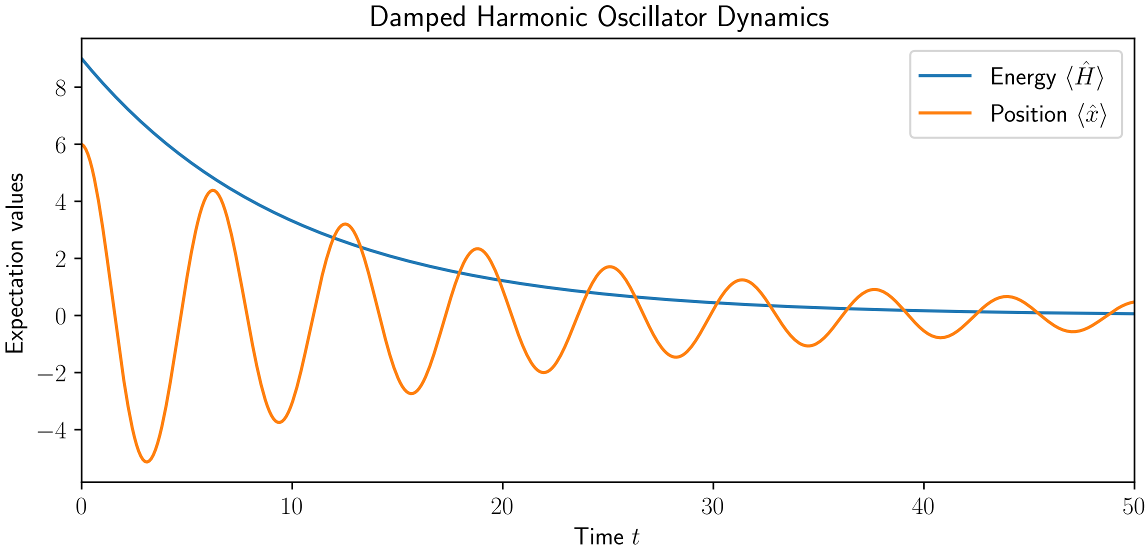
```
# Damping rate
gamma = 0.1

# Lindblad operator
L = np.sqrt(gamma) * a

# Initial state: coherent state
alpha = 3.0
psi_0 = coherent(N, alpha)

tlist = np.linspace(0, 50, 500)

# Solve the Master equation
result = mesolve(H, psi_0, tlist, [L], e_ops=[a.dag() * a, a + a.dag()])
result
```

```
<Result
  Solver: mesolve
```

```
  Solver stats:
    method: 'scipy zvode adams'
    init time: 0.00017738342285156250
    preparation time: 0.0002970695495605469
    run time: 0.26758480072021484
    solver: 'Master Equation Evolution'
    num_collapse: 1
  Time interval: [0.0, 50.0] (500 steps)
  Number of e_ops: 2
  State not saved.
>
```

11.1.1. Plotting the results

```python
import matplotlib.pyplot as plt

# Plot the expectation values
fig, ax = plt.subplots()

ax.plot(tlist, result.expect[0], label=r"Energy $\langle \hat{H} \rangle$")
ax.plot(tlist, result.expect[1].real, label=r"Position $\langle \hat{x} \rangle$")
ax.set_xlabel(r"Time $t$")
ax.set_ylabel("Expectation values")
ax.legend()
ax.set_title("Damped Harmonic Oscillator Dynamics")

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close()
SVG("_tmp_fig.svg")
```



11.2. Time-dependent parameters: the case of logical quantum gates

In classical computing, logical gates such as AND, OR, and NOT process bits (0 or 1) to perform basic operations. For example:

- AND outputs 1 only if both inputs are 1.
- OR outputs 1 if at least one input is 1.
- NOT inverts its input bit.

These gates form the building blocks of all classical algorithms and digital circuits.

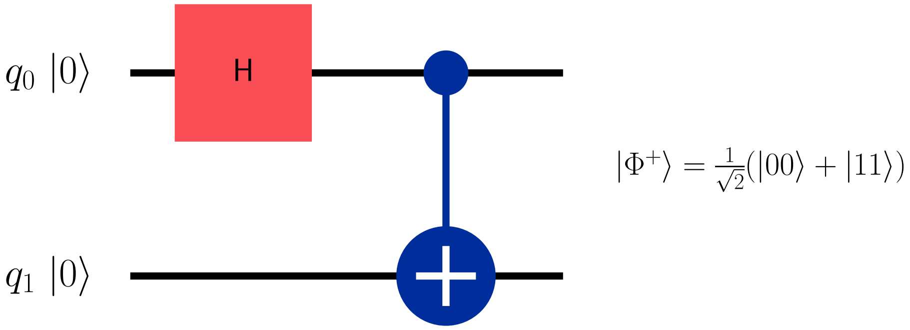
Quantum computing generalizes this idea by using qubits, which can exist in superpositions of 0 and 1. Quantum gates act on these superposed states via time-dependent Hamiltonians, enabling phenomena like entanglement and interference. This richer behaviour unlocks powerful algorithms (e.g., Shor's factoring, Grover's search) that have no efficient classical equivalent.

As an example, we consider a combination of gates that generates a bell state

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \tag{11.1}$$

To achieve this, we can use a sequence of quantum gates:

1. **Hadamard gate**: Applies a Hadamard gate to the first qubit, transforming $|0\rangle$ into $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.
2. **CNOT gate**: Applies a controlled-NOT gate, where the first qubit controls the second. This flips the second qubit if the first is in state $|1\rangle$.



Each gate can be represented by a time-dependent Hamiltonian. For example, the Hadamard gate can be implemented using a Hamiltonian

$$\hat{H}_{\mathrm{H}}^{(1)}(t_0, t) = \Theta(t - t_0)\Theta(t_0 + \pi/2 - t)\frac{1}{\sqrt{2}}\left(\hat{\sigma}_x^{(1)} - \hat{\sigma}_z^{(1)}\right)$$

where $\Theta(t)$ is the Heaviside step function, and $\hat{\sigma}_x^{(1)}$ and $\hat{\sigma}_z^{(1)}$ are the Pauli operators acting on the first qubit. The Heaviside function ensures that the Hamiltonian is non-zero only during the time interval $[t_0, t_0 + \pi/2]$, where t_0 is the time at which the gate is applied.

The CNOT gate can be implemented using a Hamiltonian that couples the two qubits

$$\hat{H}_{\mathrm{CNOT}}(t_0, t) = \Theta(t - t_0)\Theta(t_0 + \pi/2 - t)\ \left(\hat{\mathbb{1}} + \hat{\sigma}_z^{(1)}\right) \otimes \left(\hat{\mathbb{1}} - \hat{\sigma}_x^{(2)}\right)$$

where $\hat{\sigma}_x^{(2)}$ acts on the second qubit, and $\hat{\mathbb{1}}$ is the identity operator. The Heaviside function again ensures that the Hamiltonian is non-zero only during the time interval $[t_0, t_0 + \pi/2]$.

In absence of losses, the output state after applying these gates is exactly the Bell state $|\Phi^+\rangle$ defined in Equation 11.1. However, in a realistic scenario, we need to account for the effects of decoherence and dissipation, which can be modeled using the Master equation. This poses a challenge for quantum computing, as the coherence of the qubits must be maintained during the gate operations to ensure the correct output state.

To simulate the effects of decoherence, we consider the case where the qubits are in interaction with a finite-temperature environment, which can lead to energy dissipation and dephasing. In this case, the Master equation can be used to describe the dynamics of the system. We can introduce a Lindblad operator that represents the interaction with the environment. For example, we can use a Lindblad operator that describes the decay of the qubits:

$$\begin{aligned}\hat{L}_1 &= \sqrt{\gamma(n_{\text{th}}+1)}\hat{\sigma}_-^{(1)}, \quad \hat{L}_2 = \sqrt{\gamma(n_{\text{th}}+1)}\hat{\sigma}_-^{(2)}\\ \hat{L}_3 &= \sqrt{\gamma n_{\text{th}}}\hat{\sigma}_+^{(1)}, \quad \hat{L}_4 = \sqrt{\gamma n_{\text{th}}}\hat{\sigma}_+^{(2)}\end{aligned}$$

where $\hat{\sigma}_+^{(i)}$ and $\hat{\sigma}_-^{(i)}$ are the raising and lowering operators for the i-th qubit, respectively, n_{th} is the average number of thermal excitations in the environment, and γ is the decay rate. The Lindblad operators describe both energy dissipation (via $\hat{\sigma}_-$) and thermal excitation (via $\hat{\sigma}_+$) of the qubits.

We can use the ability of QuTiP to define time-dependent Hamiltonians to simulate the evolution of the system under the influence of these gates and the Lindblad operators.

```python
def heaviside(t, t0):
    return t >= t0

def hadamard_coeff(t):
    t0 = 0  # Start time of the Hadamard gate
    return heaviside(t, t0) * heaviside(t0 + np.pi / 2, t)

def cnot_coeff(t):
    t0 = np.pi / 2  # Start time of the CNOT gate
    return heaviside(t, t0) * heaviside(t0 + np.pi / 2, t)

sm1 = tensor(sigmam(), qeye(2))  # Lowering operator for qubit 1
sm2 = tensor(qeye(2), sigmam())  # Lowering operator for qubit 2
sx1 = tensor(sigmax(), qeye(2))  # Pauli X for qubit 1
sz1 = tensor(sigmaz(), qeye(2))  # Pauli Z for qubit 1
sx2 = tensor(qeye(2), sigmax())  # Pauli X for qubit 2

# Parameters
gamma = 0.1  # Decay rate
T = 0.1  # Temperature (arbitrary units)
n_th = 1 / (np.exp(1 / T) - 1)  # Average number of thermal excitations

# Hadamard gate Hamiltonian
H_hadamard = (sx1 - sz1) / np.sqrt(2)
```

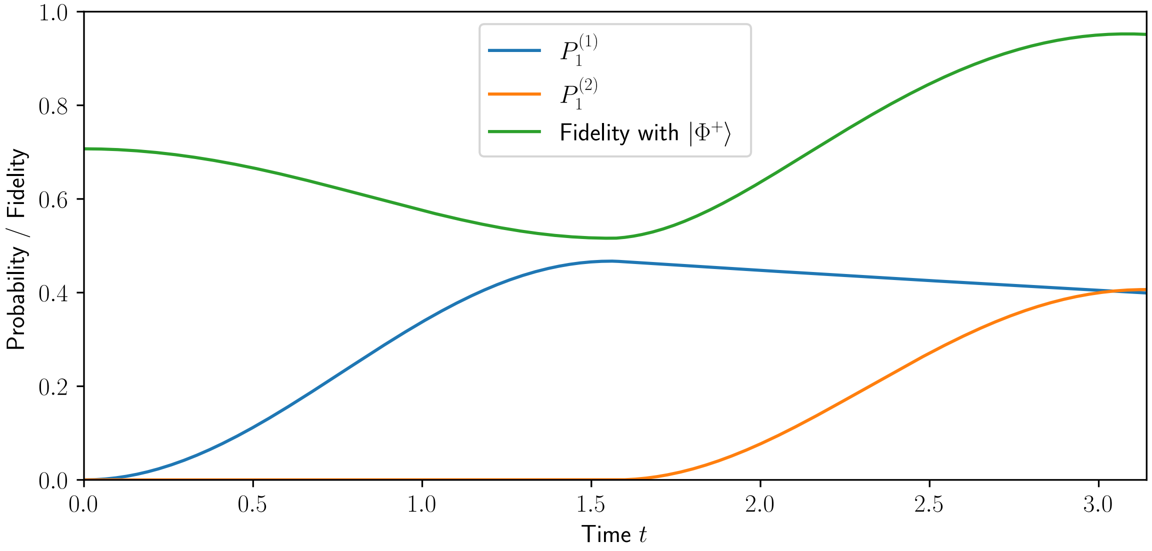
```
# CNOT gate Hamiltonian
H_cnot = (1 + sz1) * (1 - sx2) / 2

# Time-dependent Hamiltonian
H = [[H_hadamard, hadamard_coeff],
     [H_cnot, cnot_coeff]]

# Lindblad dissipation operators
L1 = np.sqrt(gamma * (n_th + 1)) * sm1
L2 = np.sqrt(gamma * (n_th + 1)) * sm2
L3 = np.sqrt(gamma * n_th) * sm1.dag()
L4 = np.sqrt(gamma * n_th) * sm2.dag()

c_ops = [L1, L2, L3, L4]

# Initial state: |00>
psi_0 = tensor(basis(2, 1), basis(2, 1))

# Time list
tlist = np.linspace(0, np.pi, 500)

# Solve the Master equation
result = mesolve(H, psi_0, tlist, c_ops)
result
```

```
<Result
  Solver: mesolve
  Solver stats:
    method: 'scipy zvode adams'
    init time: 7.176399230957031e-05
    preparation time: 0.0002262592315673828
    run time: 0.028150558471679688
    solver: 'Master Equation Evolution'
    num_collapse: 4
  Time interval: [0.0, 3.141592653589793] (500 steps)
  Number of e_ops: 0
  States saved.
>
```

We now plot the following quantities as a function of time:

- The probability of finding the first qubit in state $|1\rangle$.
- The probability of finding the second qubit in state $|1\rangle$.
- The fidelity with the target Bell state $|\Phi^+\rangle$.

```
# Bell state
bell_state = (tensor(basis(2, 0), basis(2, 0)) + tensor(basis(2, 1), basis(2, 1))).unit()

P0_1 = expect(sm1.dag() * sm1, result.states)
P0_2 = expect(sm2.dag() * sm2, result.states)
fid = [fidelity(bell_state, s) for s in result.states]

fig, ax = plt.subplots()

ax.plot(tlist, P0_1, label=r"$P_1^{(1)}$")
ax.plot(tlist, P0_2, label=r"$P_1^{(2)}$")
ax.plot(tlist, fid, label=r"Fidelity with $|\Phi^+\rangle$")
ax.set_xlabel(r"Time $t$")
ax.set_ylabel("Probability / Fidelity")
ax.set_ylim(0, 1)
ax.legend()

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close()
SVG("_tmp_fig.svg")
```



12. The Jaynes–Cummings model: The Prove of Field Quantization

At the crossroads of **quantum optics** and **cavity QED**, the **Jaynes–Cummings (JC) model** stands out as the simplest non-trivial arena in which a *single* photon mode interacts with a *single* two-level atom. Despite – or perhaps because of – its austerity, it captures phenomena that range from vacuum Rabi oscillations to Schrödinger-cat states. In what follows I weave together the *story* of the model, its mathematics, and its experimental vindication, culminating in Serge Haroche's celebrated observation of field quantisation.

The Jaynes–Cummings Hamiltonian: more than a toy

The JC Hamiltonian was born amid a heated 1960s debate over whether maser radiation had to be quantised. Edward Jaynes and Frederick Cummings answered with a fully quantum description whose predictions flatly contradicted semi-classical intuition. In the rotating-wave approximation it reads

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

where $\hat{a}$ is the cavity field's annihilation operator, $\hat{\sigma}_z$ is the atom's Pauli operator, and g is the coupling strength. The first two terms describe the free evolution of cavity and atom, while the last term couples them.

The JC model conserves the total excitation number $N = \hat{a}^\dagger \hat{a} + \hat{\sigma}_+ \hat{\sigma}_-$. This means that the Hilbert space decomposes into sectors of fixed N, each spanned by the states

$$|\psi_{n,1}\rangle = |n, e\rangle \quad \text{and} \quad |\psi_{n,2}\rangle = |n+1, g\rangle$$

The JC Hamiltonian acts within each sector as a 2×2 matrix

$$\hat{H}_n = \begin{pmatrix} \omega_c n + \frac{\omega_q}{2} & g\sqrt{n+1} \\ g\sqrt{n+1} & \omega_c(n+1) - \frac{\omega_q}{2} \end{pmatrix}.$$

The eigenvalues of this matrix are

$$E_{n,\pm} = (n + \frac{1}{2})\omega_c \pm \frac{1}{2}\Omega_n \tag{12.1}$$

where

$$\Omega_n = \sqrt{(\omega_c - \omega_q)^2 + 4g^2(n+1)} \tag{12.2}$$

is the Rabi frequency.

Collapse and revival in plain words

When the cavity field is prepared in a coherent state $|\alpha\rangle$ and the atom in its ground state $|g\rangle$, the atomic inversion can be written as a superposition of Rabi oscillations at frequencies Ω_n, each weighted by the Poisson probability $P_n = e^{-|\alpha|^2}|\alpha|^{2n}/n!$. The resulting interference leads to an initial collapse of the oscillations, followed by a revival. The time evolution of the atomic inversion is given by

$$\langle \hat{\sigma}_z(t) \rangle = \sum_{n=0}^{\infty} P_n \, \cos(2\Omega_n \, t) \, .$$

The open-system perspective

Real cavities leak, as well as electrons in the atom decay. To account for this, we can extend the JC Hamiltonian with a Lindblad term that describes the interaction with the environment. The master equation for the density operator ρ reads

$$\dot{\rho} = -\tfrac{i}{\hbar}[\hat{H}, \rho] + \kappa \, \mathcal{D}[\hat{a}]\rho + \gamma \, \mathcal{D}[\hat{\sigma}_-]\rho,$$

with $\mathcal{D}[\hat{O}]\rho = \hat{O}\rho\hat{O}^\dagger - \frac{1}{2}\hat{O}^\dagger\hat{O}, \rho$ being the Lindblad dissipator. When κ and γ are small compared to the Rabi frequency, the JC model is in the strong-coupling regime, where the coherent oscillations are visible despite the dissipation. On the contrary, if κ or γ are large, the oscillations are damped and eventually disappear, which corresponds to the weak-coupling regime.

12.1. Simulating the JC model with QuTiP

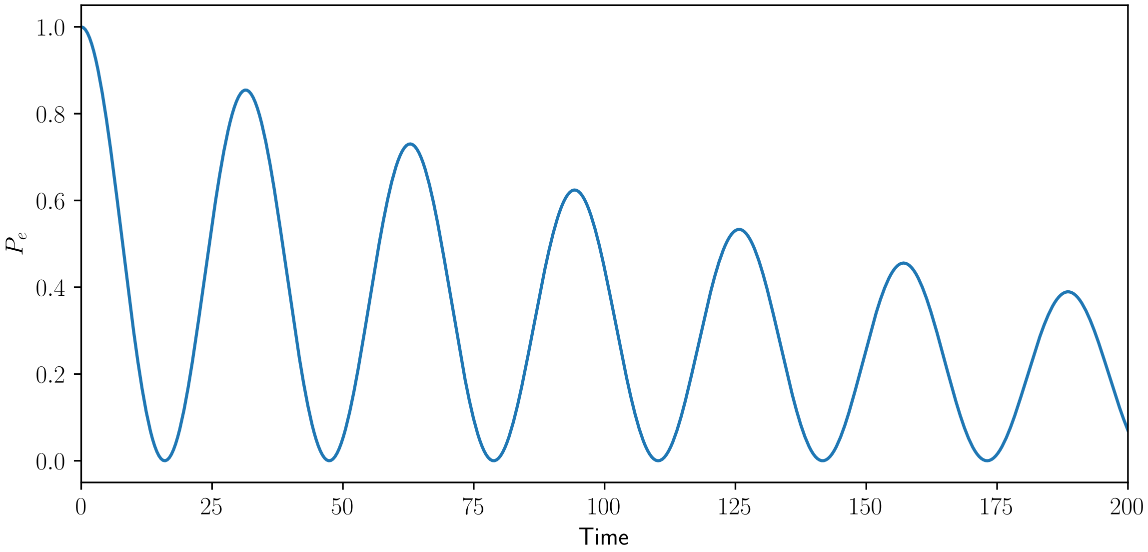
```
import numpy as np
import matplotlib.pyplot as plt
from qutip import *

# Parameters
wc    = 5.0   # cavity
wq    = 5.0   # atom (on resonance)
g     = 0.1   # coupling
kappa = 0.01  # cavity decay
gamma = 0.00  # atomic T1 (suppressed)
N     = 20    # Fock cutoff

# Operators
a  = tensor(destroy(N), qeye(2))
sp = tensor(qeye(N), sigmap())
sm = tensor(qeye(N), sigmam())
sz = tensor(qeye(N), sigmaz())

H = wc * a.dag() * a + 0.5*wq * sz + g * (a.dag() * sm + a * sp)

# Dissipators
c_ops = [np.sqrt(kappa) * a, np.sqrt(gamma) * sm]
```

Damped vacuum Rabi oscillations: The *hello-world* of cavity QED

We now simulate the open system dynamics of the JC model, starting from the $|\psi(0)\rangle = |0, e\rangle$ state and watch excitation watch the atomic population in time. Theory predicts a cosine at frequency $2g$, blurred by an exponential envelope $e^{-\kappa t/2}$.

```
 0 = tensor(basis(N, 0), basis(2, 0))
tlist  = np.linspace(0, 200, 500)

result = mesolve(H,  0, tlist, c_ops, e_ops=[sp*sm])
P_e = result.expect[0]

fig, ax = plt.subplots()
ax.plot(tlist, P_e)
ax.set_xlabel("Time")
ax.set_ylabel(r"$P_{e}$")

# Show in Quarto
plt.savefig("_tmp_fig.svg")
```

```
plt.close(fig)
SVG("_tmp_fig.svg")
```



12.2. Demonstration of Electromagnetic Field Quantization: The Haroche Experiment

A landmark experiment that unambiguously demonstrated the quantization of the electromagnetic field was conducted by **Serge Haroche's group** in the 1990s (Brune et al. 1996). The experiment used high-**Q** superconducting microwave cavities prepared in a **coherent state** $|\alpha\rangle$. Rydberg atoms were sent one at a time through the cavity, interacting dispersively with the quantized field. The population of the atomic excited state was measured as a function of the atom-cavity interaction time.

12.2.1. Key Observations:

- The atomic population exhibited **collapse and revival** dynamics.
- These revivals correspond to the quantum interference between Rabi oscillations at different frequencies $\Omega_n = \Omega_0\sqrt{n+1}$, each associated with a Fock state component $|n\rangle$ in the coherent state.
- A **Fourier transform** of the signal revealed multiple peaks, each corresponding to a discrete photon number.

This experiment **cannot be explained** using a classical field description. The observed dynamics and frequency components are direct evidence of the **discrete (quantized) nature** of the electromagnetic field. It remains one of the most compelling demonstrations of field quantization in quantum optics.

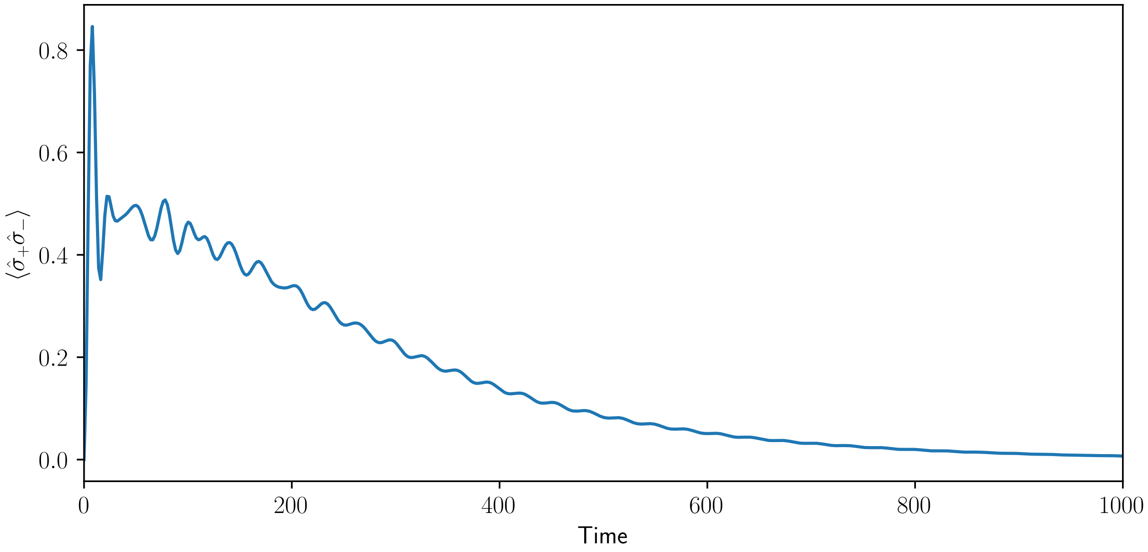
Preparing the coherent field

To simulate Haroche's experiment, we need to prepare a coherent state $|\alpha\rangle$ in the cavity and a two-level atom in its ground state.

```
 = 2.0
_fld = coherent(N,  )
_atm = basis(2, 1)
0 = tensor( _fld,  _atm)
```

We can now use the `mesolve` function to evolve the system in time, while measuring the atomic population.

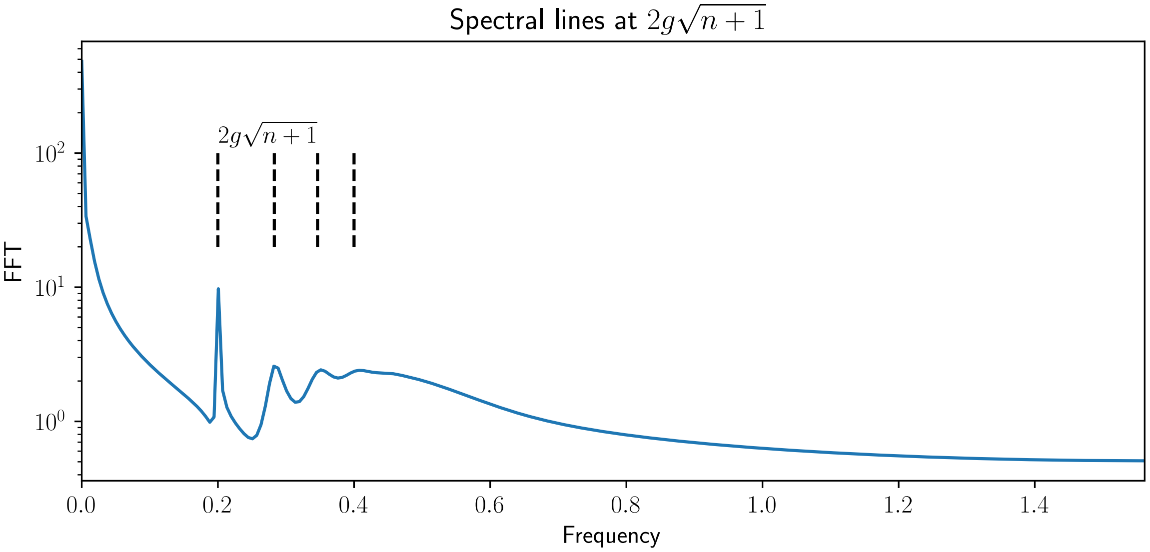
```
tlist  = np.linspace(0, 1000, 500)

out = mesolve(H,  0, tlist, c_ops, e_ops=[sp*sm])
pop = out.expect[0]

fig, ax = plt.subplots()
ax.plot(tlist, pop)
ax.set_xlabel("Time")
ax.set_ylabel(r"$\langle\hat\sigma_+ \hat\sigma_-\rangle$")

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```



Fourier spectroscopy: The signature of Fock states

Each Fock state $|n\rangle$ drives the atom at $\Omega_n = 2g\sqrt{n+1}$, so the Fourier transform of $\langle \hat{\sigma}_z(t) \rangle$ must exhibit a comb at precisely those spacings.

```
from numpy.fft import fft, fftfreq

# We remove the exponential decay factor to make the peaks more visible
exp_factor = np.exp(-kappa * tlist / 2)
fft_sig = fft(pop / exp_factor)[0:len(pop)//2]

freq = fftfreq(len(pop), (tlist[1]-tlist[0]))[:len(pop)//2] * 2 * np.pi

fig, ax = plt.subplots()
ax.plot(freq, np.abs(fft_sig))
ax.vlines(2*g*np.sqrt(np.arange(1, 5)), 20, 100, color="black", linestyles="--")
ax.text(2*g*np.sqrt(1), 110, r"$2g\sqrt{n+1}$", fontsize=12, ha="left", va="bottom")
ax.set_yscale("log")
ax.set_xlabel("Frequency")
ax.set_ylabel("FFT")
ax.set_title("Spectral lines at $2g\sqrt{n+1}$")

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```



This plot worths the Nobel prize in 2012, as it shows the quantization of the electromagnetic field. Indeed, the peaks at $2g\sqrt{n+1}$ are a direct signature of the Fock states $|n\rangle$ in the coherent state $|\alpha\rangle$. Each peak corresponds to a different photon number, and their spacing reflects the quantized nature of the electromagnetic field.

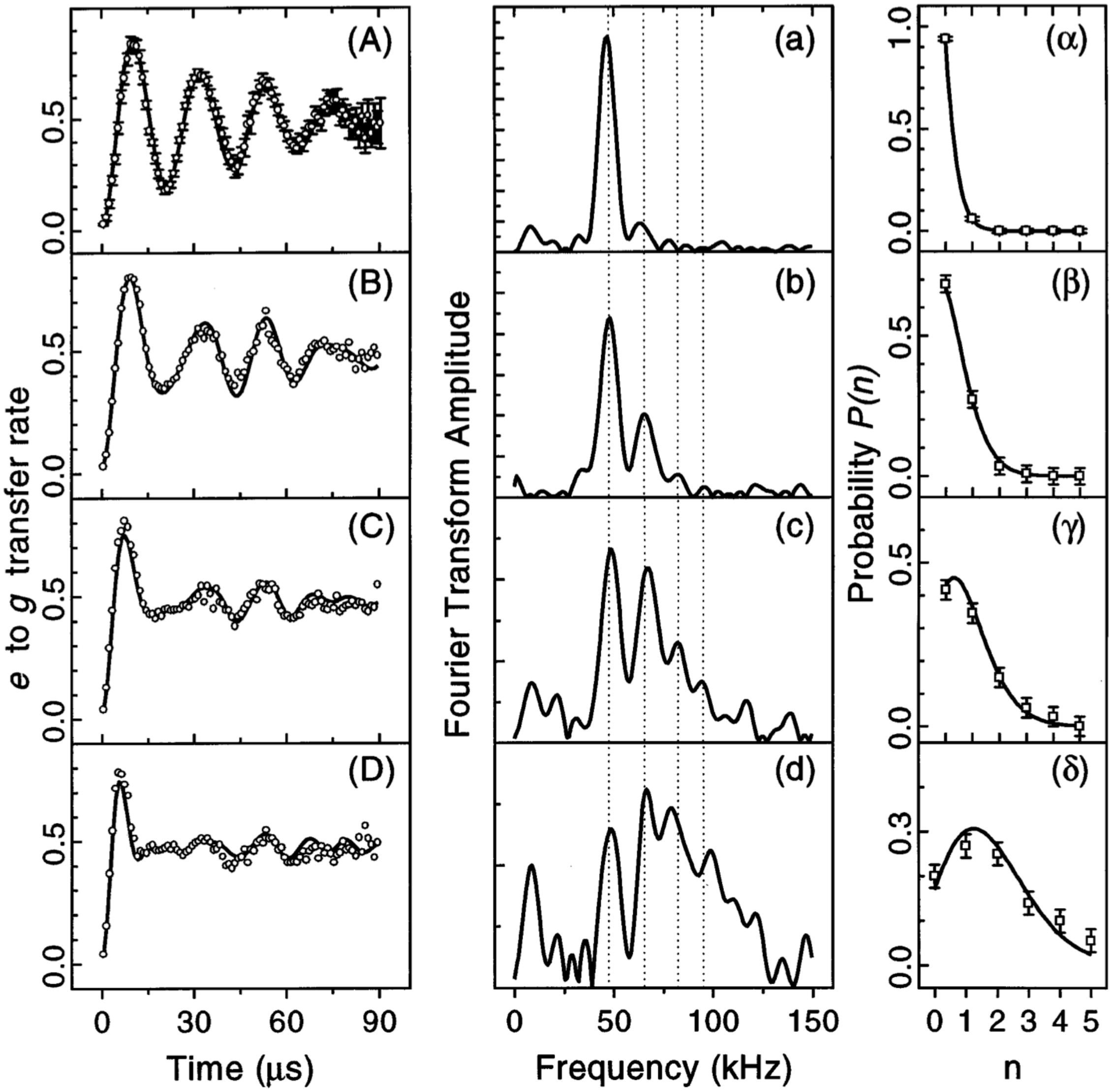


Figure 12.1.: Experimental data from Haroche's group showing the quantization of the electromagnetic field at different initial coherent state amplitudes (Brune et al. 1996).

13. Resonance Fluorescence and the Mollow Triplet

When a *resonant* laser drives a two-level atom strongly, its resonance-fluorescence spectrum splits into **three** Lorentzian peaks: a central line at the laser frequency and two symmetric sidebands. Predicted by B.R. Mollow in 1969 (Mollow (1969)) and first observed soon after, this *Mollow triplet* is a hallmark of light–matter interaction in the strong-drive (dressed-state) regime.

13.1. Physical picture

Consider a two-level atom with energy separation ω_0 driven by a coherent laser at frequency ω_L.

$$H = \frac{\omega_0}{2}\,\hat{\sigma}_z + \frac{\Omega}{2}\,(\hat{\sigma}_+ e^{-i\omega_L t} + \hat{\sigma}_- e^{+i\omega_L t}).$$

Transforming to the laser rotating frame with $U(t) = \exp\left[-i\frac{\omega_L t}{2}\hat{\sigma}_z\right]$ removes the explicit time dependence and shifts the zero of energy, yielding the textbook Hamiltonian used below

$$H = \frac{\Delta}{2}\,\hat{\sigma}_z + \frac{\Omega}{2}\,(\hat{\sigma}_+ + \hat{\sigma}_-),$$

with detuning $\Delta = \omega_0 - \omega_L$ and on-resonance Rabi frequency $\Omega = \mu E_0/\hbar$. This semiclassical model, combined with a Lindblad dissipator for spontaneous emission at rate γ, fully captures the triplet.

13.2. Analytic spectrum

In order to analyse the properties of the emitted light, we can compute the **power spectrum** of the scattered photons, defined as the Fourier transform of the correlation function $\langle \hat{E}^{(-)}(t)\hat{E}^{(+)}(0)\rangle$, where $\hat{E}^{(-)}$ and $\hat{E}^{(+)}$ are the negative and positive frequency parts of the electric field operator. In the rotating frame, these operators are related to the atom's raising and lowering operators $\hat{\sigma}_+$ and $\hat{\sigma}_-$ as (Walls and Milburn 2008)

$$\hat{E}^{(-)}(t) \propto \hat{\sigma}_+(t) \quad \text{and} \quad \hat{E}^{(+)}(t) \propto \hat{\sigma}_-(t)\,.$$

The power spectrum is then given by

$$S(\omega) = \int_{-\infty}^{\infty} e^{i\omega t} \langle \hat{E}^{(-)}(t) \hat{E}^{(+)}(0) \rangle dt\,. \tag{13.1}$$

Treating spontaneous emission at rate γ with a Lindblad term $\hat{L} = \sqrt{\gamma}, \hat{\sigma}_-$, the power spectrum of scattered photons is (Walls and Milburn 2008)

$$S(\omega) \propto \frac{\frac{\gamma}{2}}{(\omega - \omega_L)^2 + (\frac{\gamma}{2})^2} + \frac{3\gamma/8}{(\omega - \omega_L - \Omega_R)^2 + (3\gamma/4)^2} + \frac{3\gamma/8}{(\omega - \omega_L + \Omega_R)^2 + (3\gamma/4)^2}.$$

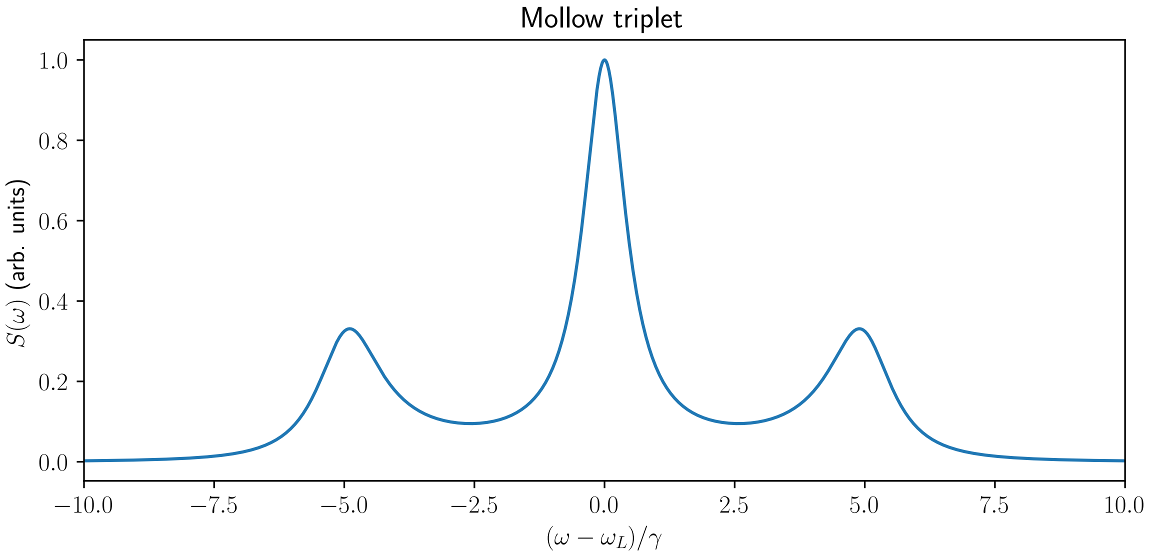
with a central peak at ω_L and two sidebands at $\omega_L \pm \Omega_R$, where $\Omega_R = \sqrt{\Delta^2 + \Omega^2}$ is the Rabi frequency in the rotating frame.

13.3. Numerical spectrum in QuTiP

Below is a minimal QuTiP script that reproduces the triplet for a resonantly driven atom ($\Delta = 0$). The code computes the emission spectrum $S(\omega)$ in Equation 13.1 by using the `spectrum` function to compute the Fourier transform of the correlation function of the emission operators.

```
import numpy as np
import matplotlib.pyplot as plt
from qutip import *

gamma = 1.0          # spontaneous emission rate
Omega = 5.0 * gamma   # Rabi frequency (drive strength)
Delta = 0.0           # laser detuning

sm = sigmam()
sp = sigmap()
sz = sigmaz()

H = 0.5 * Delta * sz + 0.5 * Omega * (sp + sm)
```

```
c_ops = [np.sqrt(gamma) * sm]

rho_ss = steadystate(H, c_ops)

wlist = np.linspace(-10 * gamma, 10 * gamma, 2000)
S = spectrum(H, wlist, c_ops, sp, sm)
S /= np.max(S)

plt.plot(wlist / gamma, S)
plt.xlabel(r"$(\omega - \omega_L)/\gamma$")
plt.ylabel(r"$S(\omega)$ (arb. units)")
plt.title("Mollow triplet")
plt.xlim(-10, 10)

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close()
SVG("_tmp_fig.svg")
```



Running the code with $\Omega = 5, \gamma$ reproduces the canonical spectrum: a narrow central line and two broader sidebands at $\pm\Omega$.

13.4. Photon statistics and antibunching

Given the emission spectrum, we may ask if the emitted light is *classical* or *quantum*. The answer can be found in the **photon statistics** of the resonance fluorescence, which can be probed by measuring the **second-order correlation function**

$$g^{(2)}(\tau) = \frac{\langle \hat{E}^{(-)}(0)\,\hat{E}^{(-)}(\tau)\,\hat{E}^{(+)}(\tau)\,\hat{E}^{(+)}(0)\rangle}{\langle \hat{E}^{(-)}\hat{E}^{(+)}\rangle^2} \propto \frac{\langle \sigma^\dagger(0)\,\sigma^\dagger(\tau)\,\sigma^-(\tau)\,\sigma^-(0)\rangle}{\langle \sigma^\dagger\sigma^-\rangle^2}. \tag{13.2}$$

where $\hat{E}^{(+)}$ and $\hat{E}^{(-)}$ are the positive and negative frequency parts of the electric field operator, and $\sigma^\dagger$, σ^- are the raising and lowering operators of the two-level atom. The quantity $g^{(2)}(\tau)$ measures the probability of detecting one photon at time 0 and another at time τ, normalised by the square of the average number of photons emitted.

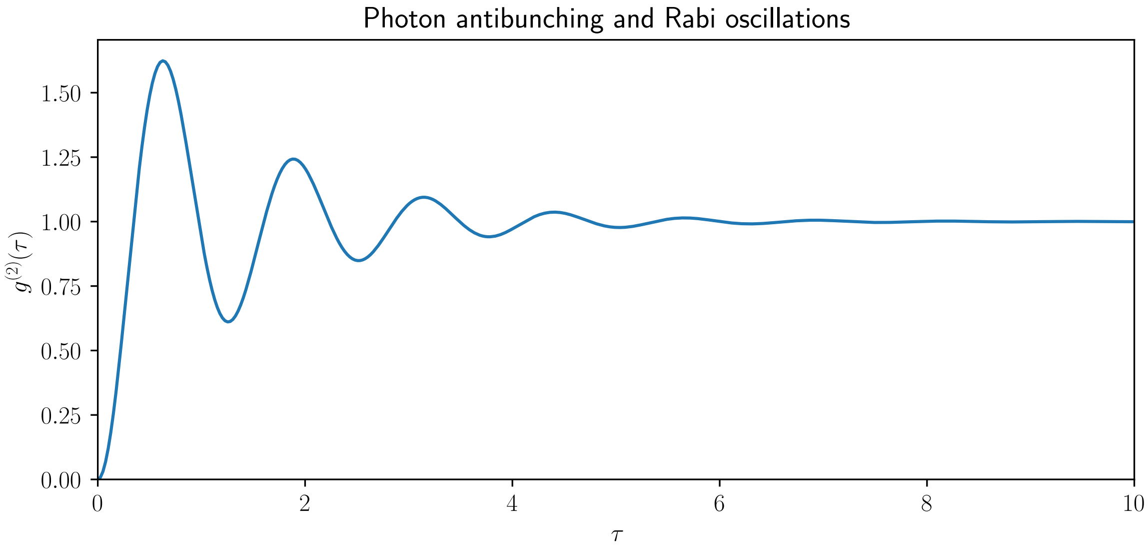
For *classical* light $g^{(2)}(0) \geq 1$ (photon bunching), whereas a single quantum emitter produces **antibunching** with $g^{(2)}(0) = 0$: once a photon is emitted, the atom is in its ground state and cannot emit another immediately, so the probability of detecting two photons with zero delay vanishes.

Under strong driving $g^{(2)}(\tau)$ also displays damped Rabi oscillations at Ω_R—a direct time-domain analogue of the sidebands.

13.4.1. QuTiP example

```
tau_list = np.linspace(0, 10 / gamma, 400)  # time delays

rho_ss = steadystate(H, c_ops)

corr = correlation_3op_1t(H, None, tau_list, c_ops, sp, sp*sm, sm)

n_ss = expect(sp * sm, rho_ss) # steady-state population
g2 = np.real(corr) / (n_ss ** 2) # normalized

plt.plot(tau_list, g2)
plt.xlabel(r"$\tau$")
plt.ylabel(r"$g^{(2)}(\tau)$")
plt.title("Photon antibunching and Rabi oscillations")
plt.ylim(0, None)

# Show in Quarto
plt.savefig("_tmp_fig.svg")
```

```
plt.close()
SVG("_tmp_fig.svg")
```



The plot shows $g^{(2)}(0) \approx 0$ (perfect antibunching in the ideal model). As τ increases the function overshoots above 1 and undergoes damped oscillations at the Rabi frequency before relaxing to the Poissonian value $g^{(2)}(\infty) = 1$.

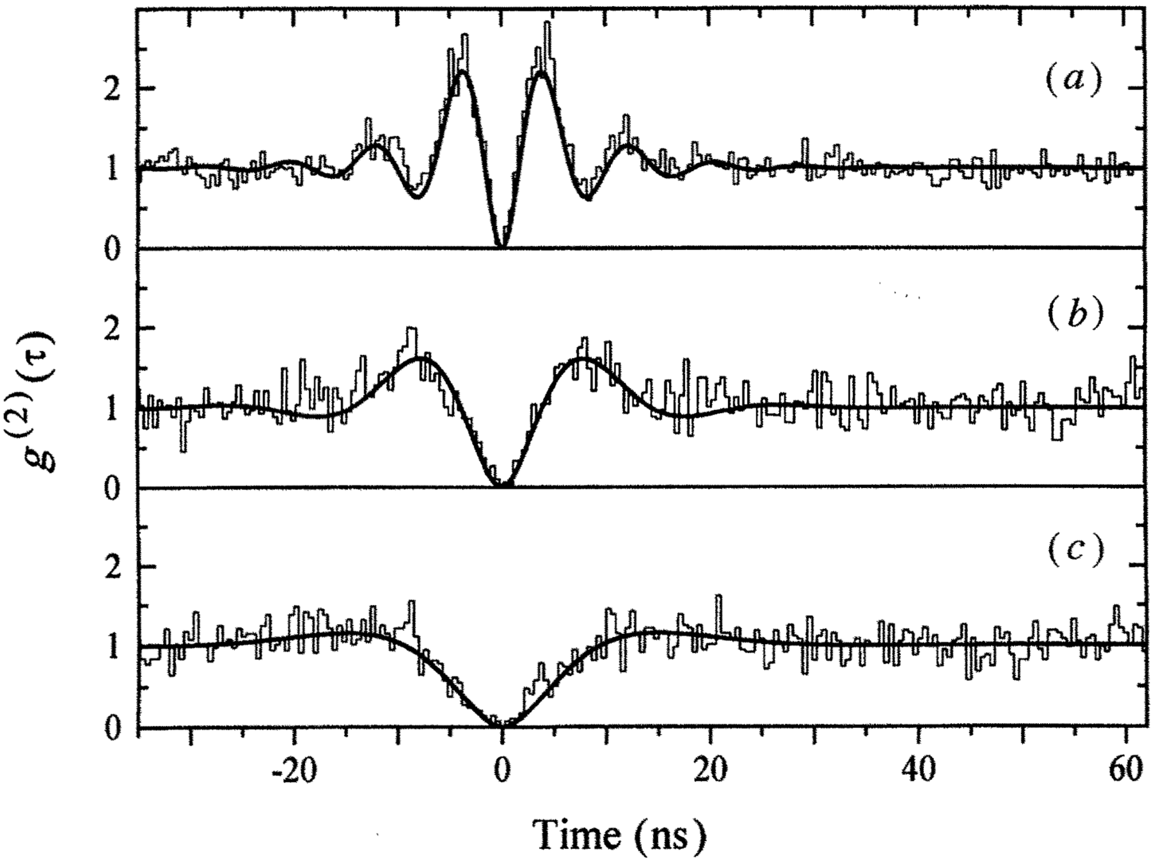


Figure 13.1.: The second order correlation function of the fluorescent light form a single mercury ion in a trap versus delay time τ. The antibunching at $\tau = 0$ is clearly visible, as well as the Rabi oscillations at longer delays. Figure taken from (Walther 1998).

13.5. Discussion

- **Antibunching** unambiguously indicates a single quantum emitter or sub-Poissonian light.
- **Rabi oscillations in $g^{(2)}$** time-domain fingerprint of the dressed-state splitting that generates the Mollow sidebands.
- **Technological relevance** Resonance-fluorescence photons combine single-photon purity (antibunching) with high brightness and tunable frequency via the drive laser.

14. The Emission Spectrum of the Jaynes–Cummings Model

In Chapter 12, we introduced the Jaynes–Cummings (JC) model as a cornerstone of quantum optics and cavity QED. Here we focus on the emission spectrum of the JC system, exploring how the interaction between a single atom and a single cavity mode manifests in the frequency domain. We will see how the spectral lines reflect the dressed–state structure of the system and how dissipation modifies this picture.

14.1. Analytic Analysis

Under the condition of zero detuning $\Delta = \omega_c - \omega_q = 0$, the Jaynes–Cummings dressed-state energies simplify. For the n-th excitation manifold, the two eigenvalues become:

$$E_{n,\pm} = n\,\omega_c \pm g\sqrt{n+1}\,.$$

Here:

- ω_c is the cavity frequency.
- g is the single-photon coupling strength.
- $n = 0, 1, 2, ...$ counts the total number of excitations shared between atom and field.

Whenever the system loses a photon, it jumps from the n-th manifold down to the $(n-1)$-th. Each of the two states at level n ($E_{n,+}$ or $E_{n,-}$) can decay into either of the two states at level $n-1$. Concretely, the four allowed decay channels are:

- $E_{n,+} \to E_{n-1,+}$
- $E_{n,+} \to E_{n-1,-}$
- $E_{n,-} \to E_{n-1,+}$
- $E_{n,-} \to E_{n-1,-}$

Each channel corresponds to a distinct emission line in the spectrum, with its frequency given by the energy difference between initial and final states.

14.1.1. Case of Small Population

When the system is weakly excited, it rarely climbs above the first excitation manifold. Practically, only $n = 0$ (the vacuum) and $n = 1$ are occupied. In this regime:

1. **Manifolds populated:** Only $n = 0$ and $n = 1$.
2. **Transitions:** From $E_{1,+}$ and $E_{1,-}$ down to the vacuum at $E_0 = 0$.
3. **Observed lines:** Exactly two, located at:

$$\omega = \omega_c \pm g$$

These two peaks form the well-known **vacuum Rabi splitting**. As soon as you start populating higher manifolds, the spectrum becomes richer, with more lines appearing at frequencies $\omega_c \pm g(\sqrt{n+1} \pm \sqrt{n})$ for $n = 0, 1, 2, \ldots$.

14.2. Numerical simulation with QuTiP

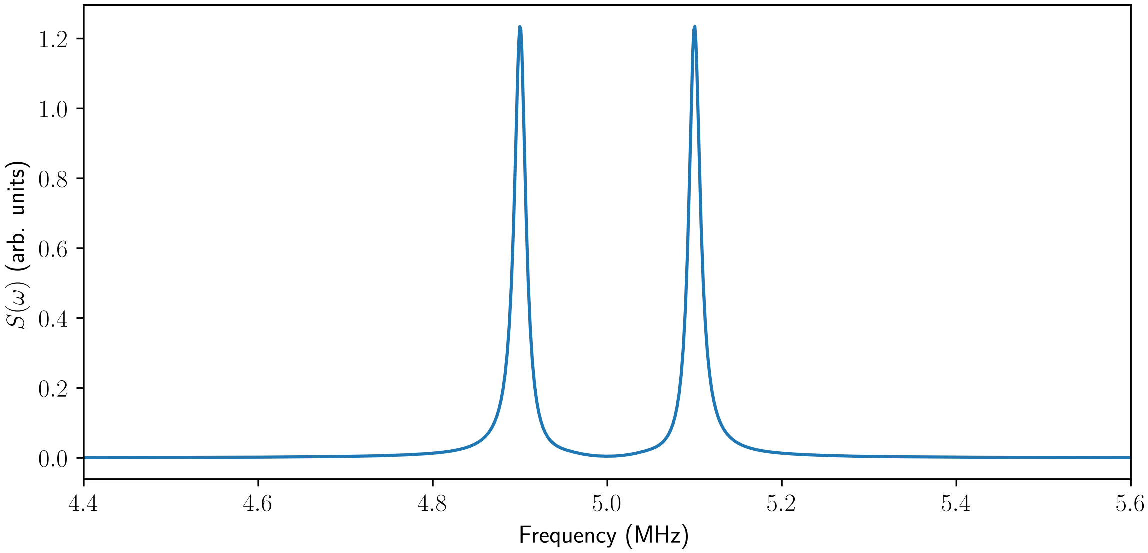
We now simulate the emission spectrum of the JC model using QuTiP, focusing on the weak and strong coupling regimes. We will compute the emission spectrum $S(\omega)$ of the cavity field using the `spectrum` function. We first define the system and its parameters.

```python
import numpy as np
import matplotlib.pyplot as plt
from qutip import *

# Parameters
wc    = 5.0   # cavity
wq    = 5.0   # atom (on resonance)
g     = 0.1   # coupling
kappa = 0.03  # cavity decay
gamma = 0.00  # atomic T1 (suppressed)
n_th  = 0.01  # thermal photons of the environment
N     = 20    # Fock cutoff

# Operators
a  = tensor(destroy(N), qeye(2))
sp = tensor(qeye(N), sigmap())
sm = tensor(qeye(N), sigmam())
sz = tensor(qeye(N), sigmaz())
```

```
H = wc * a.dag() * a + 0.5*wq * sz + g * (a.dag() * sm + a * sp)

# Dissipators
c_ops = [np.sqrt(kappa * (n_th + 1)) * a, np.sqrt(gamma * (n_th + 1)) * sm,
        np.sqrt(kappa * n_th) * a.dag(), np.sqrt(gamma * n_th) * sp]
```

And we finally compute the emission spectrum.

```
w_list = np.linspace(wc-6*g, wc+6*g, 1000)

spec = spectrum(H, w_list, c_ops, a.dag(), a)

fig, ax = plt.subplots()

ax.plot(w_list, spec.real)
ax.set_xlabel("Frequency (MHz)")
ax.set_ylabel(r"$S(\omega)$ (arb. units)")

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```



Bibliography

Born, M., R. J. Blin-Stoyle, and J. M. Radcliffe. 1989. *Atomic Physics.* Dover Books on Physics and Chemistry. Dover Publications. https://books.google.it/books?id=3eDcCgAAQBAJ.

Born, M., W. Heisenberg, and P. Jordan. 1926. "Zur Quantenmechanik. II." *Zeitschrift für Physik* 35 (8): 557–615. https://doi.org/10.1007/BF01379806.

Brune, M., F. Schmidt-Kaler, A. Maali, J. Dreyer, E. Hagley, J. M. Raimond, and S. Haroche. 1996. "Quantum Rabi Oscillation: A Direct Test of Field Quantization in a Cavity." *Phys. Rev. Lett.* 76 (March): 1800–1803. https://doi.org/10.1103/PhysRevLett.76.1800.

Campaioli, Francesco, Jared H. Cole, and Harini Hapuarachchi. 2024. "Quantum Master Equations: Tips and Tricks for Quantum Optics, Quantum Computing, and Beyond." *PRX Quantum* 5 (June): 020202. https://doi.org/10.1103/PRXQuantum.5.020202.

D'Alessandro, Domenico. 2021. *Introduction to Quantum Control and Dynamics.* Chapman; Hall/CRC. https://doi.org/10.1201/9781003051268.

Heisenberg, W. 1925. "Über Quantentheoretische Umdeutung Kinematischer Und Mechanischer Beziehungen." *Zeitschrift für Physik* 33 (1): 879–93. https://doi.org/10.1007/BF01328377.

———. 1927. "Über Den Anschaulichen Inhalt Der Quantentheoretischen Kinematik Und Mechanik." *Zeitschrift für Physik* 43 (3): 172–98. https://doi.org/10.1007/BF01397280.

Heisenberg, Werner. 1958. *Physics and Philosophy: The Revolution in Modern Science.* Edited by Ruth Nanda Anshen. Amherst, N.Y.: Prometheus Books.

Johansson, J. R., P. D. Nation, and Franco Nori. 2012. "QuTiP: An open-source Python framework for the dynamics of open quantum systems." *Computer Physics Communications* 183 (8): 1760–72. https://doi.org/10.1016/j.cpc.2012.02.021.

Khaneja, Navin, Timo Reiss, Cindie Kehlet, Thomas Schulte-Herbrüggen, and Steffen J. Glaser. 2005. "Optimal Control of Coupled Spin Dynamics: Design of NMR Pulse Sequences by Gradient Ascent Algorithms." *Journal of Magnetic Resonance* 172 (2): 296–305. https://doi.org/10.1016/j.jmr.2004.11.004.

Lambert, Neill, Eric Giguère, Paul Menczel, Boxi Li, Patrick Hopf, Gerardo Suárez, Marc Gali, et al. 2024. "QuTiP 5: The Quantum Toolbox in Python." *arXiv:2412.04705.* https://arxiv.org/abs/2412.04705.

Landau, L. D., and E. M. Lifshitz. 1981. *Quantum Mechanics: Non-Relativistic Theory.* Course of Theoretical Physics. Butterworth-Heinemann. https://books.google.it/books?id=SvdoN3k8EysC.

Lvovsky, A. I., and M. G. Raymer. 2009. "Continuous-Variable Optical Quantum-State

Tomography." *Rev. Mod. Phys.* 81 (March): 299–332. https://doi.org/10.1103/RevModPhys.81.299.

Mercurio, Alberto, Yi-Te Huang, Li-Xun Cai, Yueh-Nan Chen, Vincenzo Savona, and Franco Nori. 2025. "QuantumToolbox.jl: An Efficient Julia Framework for Simulating Open Quantum Systems." *arXiv Preprint arXiv:2504.21440*. https://doi.org/10.48550/arXiv.2504.21440.

Mollow, B. R. 1969. "Power Spectrum of Light Scattered by Two-Level Systems." *Phys. Rev.* 188 (December): 1969–75. https://doi.org/10.1103/PhysRev.188.1969.

Peruzzo, Alberto, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. 2014. "A Variational Eigenvalue Solver on a Photonic Quantum Processor." *Nature Communications* 5 (1). https://doi.org/10.1038/ncomms5213.

Walls, DF, and Gerard J Milburn. 2008. *Quantum Optics*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-28574-8.

Walther, H. 1998. "Single atom experiments in cavities and traps." *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 454 (1969): 431–45. https://doi.org/10.1098/rspa.1998.0169.

A. The quantum harmonic oscillator

As an example, here we consider the quantum harmonic oscillator. Classically, the harmonic oscillator is defined as a system subject to the force $\mathbf{F} = -k\mathbf{r}$, where k is the elastic constant. In other words, the force is proportional to the displacement from a stable point (in this case the origin).

Following the relation $\mathbf{F} = -\nabla V(\mathbf{r})$, we can say that the corresponding potential is $V(\mathbf{r}) = k/2\ \mathbf{r}^2$. The solution of the Schrodinger equation

$$i\hbar \frac{\partial}{\partial t}\Psi(\mathbf{r},t) = -\frac{\hbar^2}{2m}\nabla^2\Psi(\mathbf{r},t) + V(\mathbf{r})\Psi(\mathbf{r},t)\,,$$

where $\hbar$ is the reduced Planck constant, m is the mass of the particle, and ∇^2 is the Laplacian operator, gives us the eigenstates of the system. Considering only the one-dimensional case, we obtain the following eigenstates for the quantum harmonic oscillator:

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}}\left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega x^2}{2\hbar}} H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right)\,, \tag{A.1}$$

where $\omega = \sqrt{k/m}$ is the resonance frequency of the oscillator and H_n is the n-th Hermite polynomial.

A useful way to describe the quantum harmonic oscillator is by using the ladder operators

$$\hat{a} = \sqrt{\frac{m\omega}{2\hbar}}\left(\hat{x} + i\frac{1}{m\omega}\hat{p}\right) \tag{A.2}$$

$$\hat{a}^\dagger = \sqrt{\frac{m\omega}{2\hbar}}\left(\hat{x} - i\frac{1}{m\omega}\hat{p}\right)\,, \tag{A.3}$$

note that the position $\hat{x}$ and conjugate momentum $\hat{p}$ are operators too. If we now write the eigenstates in Equation 6.1 in the bra-ket notation ($\psi_n \to |n\rangle$), the ladder operators allow us to move from one eigenstate to the next or previous one:

$$\hat{a}\,|n\rangle = \sqrt{n}\,|n-1\rangle \tag{A.4}$$

$$\hat{a}^{\dagger}\,|n\rangle = \sqrt{n+1}\,|n+1\rangle\ , \tag{A.5}$$

and it is straightforward to recognize the creation ($\hat{a}^{\dagger}$) and annihilation ($\hat{a}$) operators. In this framework the system Hamiltonian of the quantum harmonic oscillator becomes

$$\hat{H} = \hbar\omega\left(\hat{a}^{\dagger}\hat{a} + \frac{1}{2}\right)\ .$$

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import eval_hermite, factorial

# Physical parameters
m = 1.0
k = 1.0
w = np.sqrt(k/m)
alpha = -np.sqrt(2)   # coherent-state parameter

# Grid
bounds = 6.0
x = np.linspace(-bounds, bounds, 1000)

# n-th eigenfunction of the HO (hbar=1)
def psi(n, x):
    Hn = eval_hermite(n, np.sqrt(m*w) * x)
    norm = (m*w/np.pi)**0.25 / np.sqrt(2**n * factorial(n))
    return norm * Hn * np.exp(-m*w*x**2/2)

# Build the first six eigenstates and energies
psi_n = [psi(n, x) for n in range(6)]
E_n   = [(n + 0.5) * w for n in range(6)]

# Coherent-state wavefunction (real alpha   no overall phase)
psi_coh = (m*w/np.pi)**0.25 * np.exp(- (x - np.sqrt(2)*alpha)**2 / 2)

# Plotting
fig, ax = plt.subplots()

# 1) potential
ax.plot(x, 0.5*k*x**2, 'k--', lw=2, label=r'$V(x)=\tfrac12 k x^2$')
```

```
# 2) coherent state
ax.fill_between(x, psi_coh, color='gray', alpha=0.5)
ax.plot(x, psi_coh, color='gray', lw=2, label='Coherent state')

# 3) eigenstates offset by E_n
lines = []
for n in range(6):
    y = psi_n[n] + E_n[n]
    line, = ax.plot(x, y, lw=2, label=fr'$|{n}\rangle$')
    lines.append(line)

# Cosmetics
ax.set_ylim(0, 7)
ax.set_xlabel(r'$x$')

# State labels on the right
for n, line in enumerate(lines):
    ax.text(4.5, E_n[n] + 0.2, rf'$|{n}\rangle$', color=line.get_color())

ax.text(-3.5, 6.5, r"$V(x)$")
ax.text(-3, 0.9, r"$\ket{\alpha}$", color="grey")
ax.annotate("", xy=(-5.5,3.5), xytext=(-5.5,2.5), arrowprops=dict(arrowstyle="<->"))
ax.text(-5.4, 3, r"$\hbar \omega$", ha="left", va="center")

plt.show()
```

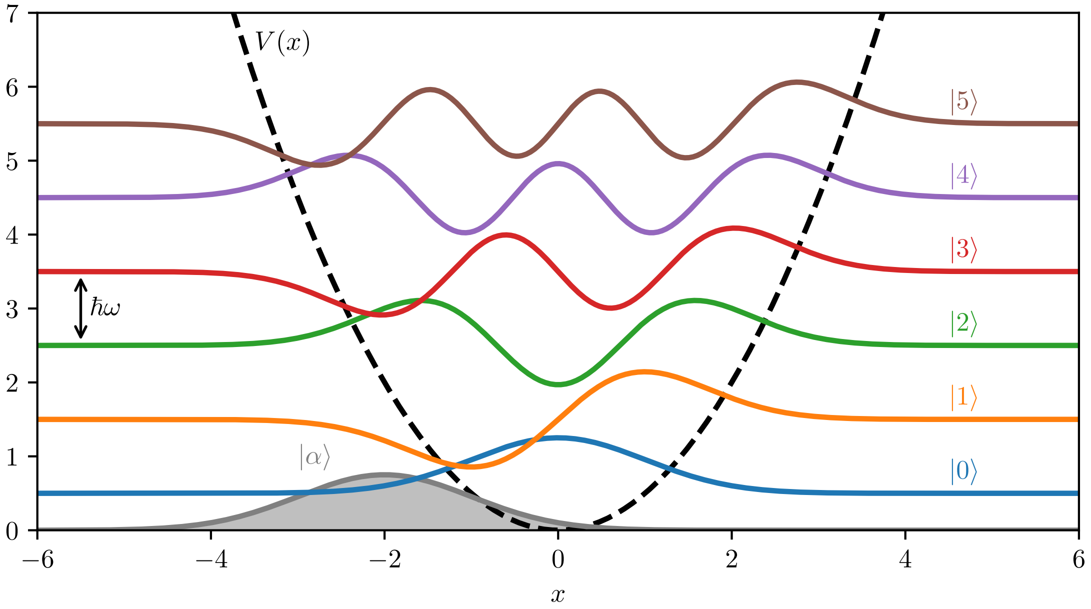


Figure A.1.: First eigenstates of one-dimensional the quantum harmonic oscillator, each of them vertically shifted by the corresponding eigenvalue. The grey-filled curve corresponds to a coherent state with $\alpha = -\sqrt{2}$. The used parameter are $m = 1$, $\omega = 1$, and $\hbar = 1$.

It is worth introducing the coherent state $|\alpha\rangle$ of the harmonic oscillator, defined as the eigenstate of the destroy operator, with eigenvalue α, in other words, $\hat{a}\,|\alpha\rangle = \alpha\,|\alpha\rangle$. It can be expressed analytically in terms of the eigenstates of the quantum harmonic oscillator

$$|\alpha\rangle = e^{-\frac{1}{2}|\alpha|^2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}}\,|n\rangle \ ,$$

and it can be seen as the most classic-like state since it has the minimum uncertainty $\Delta x \Delta p = \hbar/2$.

Figure A.1 shows the first eigenstates of the quantum harmonic oscillator, each of them vertically shifted by the respective energy, while the grey-filled curve is a coherent state with $\alpha = -\sqrt{2}$. The black dashed curve is the potential, choosing $k = 1$, $m = 1$, and $\hbar = 1$. It is worth noting that also the groundstate $|0\rangle$ has a nonzero energy ($E_0 = \hbar\omega/2$).

B. From classical densities to the Wigner function

So far we have:

- a **probability density** $\rho(x,p,t)$ for *classical* ensembles, obeying Liouville's equation;
- a **wavefunction** $\psi(x,t)$ or a **density operator** $\hat{\rho}$ for *quantum* systems, obeying the Schrödinger/von Neumann equations.

The Wigner function builds a **bridge** between these two pictures.
It lives in phase space like $\rho(x,p)$, yet it is derived from $\hat{\rho}$ and keeps all quantum information.

B.1. Definition

For a one–dimensional system the **Wigner function** is

$$W(x,p,t) \;=\; \frac{1}{2\pi\hbar}\int_{-\infty}^{+\infty} dy\; e^{-ipy/\hbar}\,\big\langle x+\tfrac{y}{2}\big|\hat{\rho}(t)\big|x-\tfrac{y}{2}\big\rangle. \tag{B.1}$$

A quick checklist:

- x and p are *simultaneous* variables (even though they do not commute quantum mechanically).
- W is **real**, but it can take **negative values** – an unmistakable quantum signature.
- The normalisation matches that of ρ: $\displaystyle\int dx\,dp\,W(x,p,t) = 1$.

Classical Quantum analogy

- $\rho(x,p,t)$ is always non-negative.
- $W(x,p,t)$ can be negative, revealing quantum interference.

B.2. Marginals and expectation values

Despite possible negativity, W returns the correct **probability densities** for position and momentum:

$$\int_{-\infty}^{+\infty} dp\, W(x,p,t) \;=\; \langle x|\hat{\rho}|x\rangle \;=\; |\psi(x,t)|^2,$$

$$\int_{-\infty}^{+\infty} dx\, W(x,p,t) \;=\; \langle p|\hat{\rho}|p\rangle.$$

Expectation values of symmetrised operators follow the **phase-space average** rule

$$\langle \hat{A}\rangle \;=\; \int dx\, dp\, A_{\mathrm{W}}(x,p)\, W(x,p,t),$$

where A_{W} is the **Weyl symbol** of $\hat{A}$ (the phase-space version of the operator).

B.3. Properties worth remembering

Property	Classical ρ	Quantum W
Real		
Non-negative		possible negativity
Normalised		
Obeys continuity / Liouville		, plus $\hbar$-corrections
Supports interference fringes		

B.4. Example : Gaussian wavepacket

For a *minimum-uncertainty* Gaussian wavepacket

$$\psi(x) \;=\; \frac{1}{(2\pi\sigma_x^2)^{1/4}}\exp\Big[-\frac{(x-x_0)^2}{4\sigma_x^2} + i\,p_0(x-x_0)/\hbar\Big],$$

the Wigner function is also Gaussian and **everywhere positive**:

$$W(x,p) \;=\; \frac{1}{\pi\hbar}\,\exp\Bigl[-\frac{(x-x_0)^2}{2\sigma_x^2}-\frac{2\sigma_x^2}{\hbar^2}(p-p_0)^2\Bigr].$$

Negativity appears only when the state contains quantum *interference*, for example in superpositions of spatially separated Gaussians.

B.5. Why the Wigner function matters

- It lets us **visualise quantum states** in the familiar (x,p) plane.
- Many semiclassical techniques expand around W and truncate the series.
- In quantum optics the Wigner function of an electromagnetic mode can be **measured** with homodyne tomography (see later lectures).

Using $W(x,p,t)$ we now have a complete trio:

1. **Classical ensemble** — $\rho(x,p,t)$ (Liouville).
2. **Quantum wavefunction/density operator** — $\psi(x,t)$ / $\hat{\rho}(t)$ (Schrödinger/von Neumann).
3. **Quantum phase-space picture** — $W(x,p,t)$ (Moyal evolution).

B.6. Example: the nonlinear oscillator

In Chapter 7 we solved the Liouville equation for a classical nonlinear oscillator with Hamiltonian

$$H = \frac{p^2}{2m} + \frac{1}{2}kx^2 + gx^4. \tag{B.2}$$

Moreover, in Chapter 10 we solved the Schrödinger equation using QuTiP for aq simple harmonic oscillator. We now combine these two approaches to study the **quantum nonlinear oscillator** with Hamiltonian Equation B.2, but with the **quantum operator** $\hat{x}$ and $\hat{p}$ instead of the classical variables x and p.

```python
import numpy as np
from qutip import *

N = 120
m = 0.5 # Mass of the particle
k = 2.0 # Spring constant
G = 0.15 # Nonlinear constant
w = np.sqrt(k/m) # Angular frequency

a = destroy(N)

x = np.sqrt(m * w / 2) * (a + a.dag())
p = 1j * np.sqrt(m * w / 2) * (a.dag() - a)

H = p**2 / (2 * m) + k * x**2 / 2 + G * x**4

# Initial state: coherent state
alpha = np.sqrt(1 / (2 * m * w)) * 1 + 1j * np.sqrt(m * w / 2) * 0.1
psi_0 = coherent(N, alpha)

tlist = np.linspace(0, 2, 500)
result = sesolve(H, psi_0, tlist)
result
```

```
<Result
  Solver: sesolve
  Solver stats:
    method: 'scipy zvode adams'
    init time: 0.00020766258239746094
    preparation time: 0.00016641616821289062
    run time: 0.9459991455078125
    solver: 'Schrodinger Evolution'
  Time interval: [0.0, 2.0] (500 steps)
  Number of e_ops: 0
  States saved.
>
```

The Wigner function can be computed from the resulting state using QuTiP's `wigner` function. Let's plot the Wigner function at the final time step:

```
fig, ax = plt.subplots()

img = ax.pcolormesh(x_list, px_list, wigner(result.states[-1], x_list, px_list),
                    shading="gouraud", rasterized=True,
                    vmin=-0.3, vmax=0.3, cmap="PuOr")
ax.set_xlabel("Position $x$")
ax.set_ylabel("Momentum $p_x$")

# Show in Quarto
plt.savefig("_tmp_fig.svg")
plt.close(fig)
SVG("_tmp_fig.svg")
```