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

02-06-2025

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

Table of contents

1. Home Page **3**

2. Introduction **4**
2.1. Why simulate open quantum systems? . . . 4
2.2. Why Python? . . . 4
2.3. How does Python differ from other mainstream languages? . . . 5
2.4. A glance at Julia and *QuantumToolbox.jl* . . . 6
2.5. Course scope . . . 6
2.6. First steps in Python: lists, loops, and functions . . . 7
2.6.1. Creating and using lists . . . 7
2.6.2. For loops . . . 7
2.6.3. Defining functions . . . 8
2.6.4. Lambda (anonymous) functions . . . 8
2.6.5. Complex numbers . . . 8
2.6.6. Why plain Python lists can be slow . . . 9
2.6.7. Enter `numpy` . . . 9

3. Linear Algebra with NumPy and SciPy **11**
3.1. NumPy: The Foundation of Dense Linear Algebra . . . 11
3.1.1. Summary of Core Functions . . . 12
3.1.2. Matrix–Matrix and Matrix–Vector Multiplication . . . 12
3.1.3. Diagonalization . . . 13
3.1.4. Kronecker Product . . . 14
3.2. SciPy: Advanced Algorithms and Sparse Data . . . 15
3.3. Some Useful Functions . . . 15
3.4. Solving Linear Systems . . . 16
3.5. Sparse Matrices . . . 17
3.5.1. Eigenvalues of Sparse Matrices . . . 18

4. Speeding up Python for Linear Algebra Tasks **20**
4.1. Numba: Just-In-Time Compilation . . . 20
4.2. JAX: XLA Compilation and Automatic Differentiation . . . 21
4.2.1. A Quick Overview of Automatic Differentiation . . . 22
4.3. Why Computing Gradients Is Important in Quantum Physics . . . 25
4.4. Summary . . . 26

5. Ordinary Differential Equations **27**
5.1. General Definition and Examples . . . 27
5.2. Solving Linear ODEs by Diagonalizing the System Matrix . . . 28
5.2.1. Eigenvalue Decomposition . . . 28
5.2.2. Relation to the Schrödinger Equation . . . 29
5.3. Numerical Solution via the Euler Method . . . 32
5.3.1. Forward Euler Method . . . 32
5.3.2. Stability Criterion for the Euler Method . . . 33
5.4. Applying These Methods to the Schrödinger Equation . . . 35
Time-Independent Hamiltonian . . . 35
5.4.1. Time-Dependent Hamiltonian . . . 35
5.4.2. Open Quantum Systems . . . 35
5.5. Conclusion . . . 36

6. From Hamilton's equations to the Liouville equation in phase space **37**
6.1. From Hamilton's equations to Liouville's continuity law . . . 37
6.2. Physical Interpretation . . . 38
6.3. Discretizing the Liouville Operator with Finite Differences . . . 39
6.3.1. Discretizing Phase Space . . . 39
6.3.2. Building the Liouville Matrix Operator . . . 40
6.4. Time Evolution . . . 40
6.5. Running example: a quartic non-linear oscillator . . . 41

7. Representing Quantum States and Operators with NumPy **45**
7.1. Pauli Operators . . . 46
7.2. Harmonic Oscillator . . . 46
7.2.1. Action of the Destroy Operator on a Fock State . . . 48
7.3. Partial Trace . . . 49
7.4. Why QuTiP? . . . 50

8. Introduction to QuTiP **51**
8.1. Quantum Operators . . . 52
8.1.1. Creating Operators . . . 52
8.1.2. Operator Functions and Operations . . . 55
8.2. Quantum States . . . 55
8.2.1. Fock States . . . 55
8.2.2. Superposition States . . . 56
8.2.3. Coherent States . . . 56
8.2.4. Spin States . . . 57
8.2.5. Density Matrices . . . 58
8.2.6. Partial Trace . . . 59
8.3. Eigenstates and Eigenvalues . . . 62
8.4. Computing Expectation Values . . . 63

Bibliography **65**

Appendices **66**

A. The quantum harmonic oscillator **66**

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

2.1. 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.

2.2. 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.

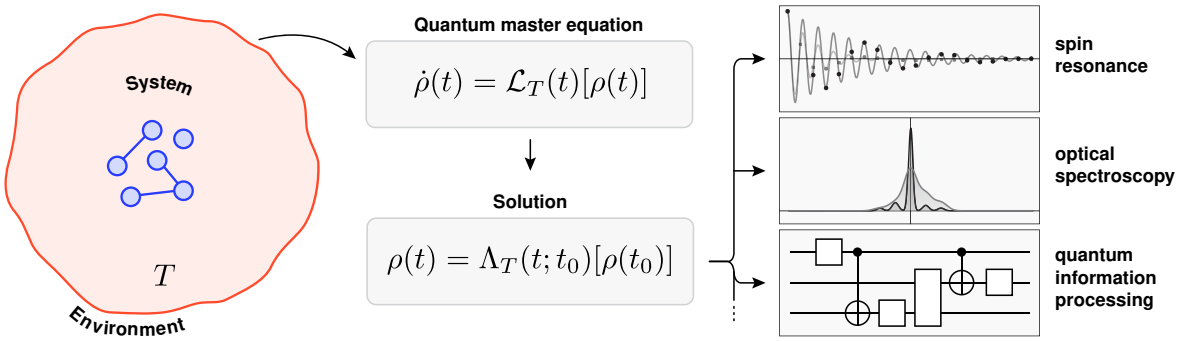


Figure 2.1.: Description of an open quantum system and its practical applications. A quantum system interacts with a macroscopic environment, leading to decoherence and dissipation. The evolution of the system is described the master equation $\dot{\hat{\rho}} = \mathcal{L}_T(t)[\hat{\rho}]$, where $\hat{\rho}$ 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.

2.3. 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.

2.4. 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.

2.5. 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.

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

2.6.1. 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:

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

```
First fruit: apple
```

2.6.2. 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.

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

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

2.6.3. 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.

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

print(square(5))
```

```
25
```

2.6.4. 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`.

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

```
25
```

2.6.5. 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.

```
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
```

2.6.6. 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.

2.6.7. 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.

```
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: 7.8745 milliseconds
Sum using numpy: 499999.5, Time taken: 0.4518 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.

```
# 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.

3. 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!

3.1. 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.

3.1.1. 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.

3.1.2. 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:

```
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 $A\mathbf{v}$.

3.1.3. 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:

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

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.

3.1.4. 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`:

```
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.

3.2. 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.

3.3. Some Useful Functions

Below are a few handy SciPy routines:

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

```python
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)
```

3.4. 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

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

We can reproduce this with NumPy:

```python
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.

3.5. 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:

```
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])
```

3.5.1. 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

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

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:

```
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])
```

4. 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.

4.1. 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):  0.34165382385253906
Numba time (ms):  0.2930164337158203
```

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.

4.2. 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.

4.2.1. 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.

4.2.1.1. 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.

4.2.1.2. 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.

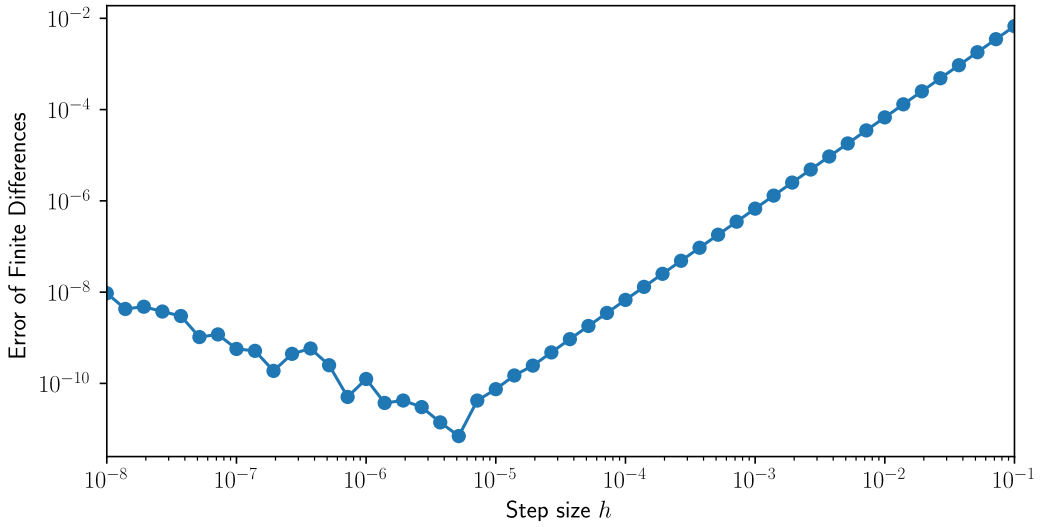
4.2.1.3. Comparing Accuracy: AD vs Finite Differences

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 $x = 1.0$.

```
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(filename='_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.

4.3. 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 $\theta = (\theta_1, \ldots, \theta_n)$. 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.

4.4. 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.

5. 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.

5.1. 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.

5.2. Solving Linear ODEs by Diagonalizing the System Matrix

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

$$\frac{d\mathbf{y}(t)}{dt} = A\,\mathbf{y}(t), \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{5.1}$$

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

5.2.1. 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 5.1, 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.

5.2.2. Relation to the Schrödinger Equation

When dealing with the time-dependent Schrödinger equation for a *time-independent* Hamiltonian $\hat{H}$, 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.

5.2.2.1. 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.

```
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:
```

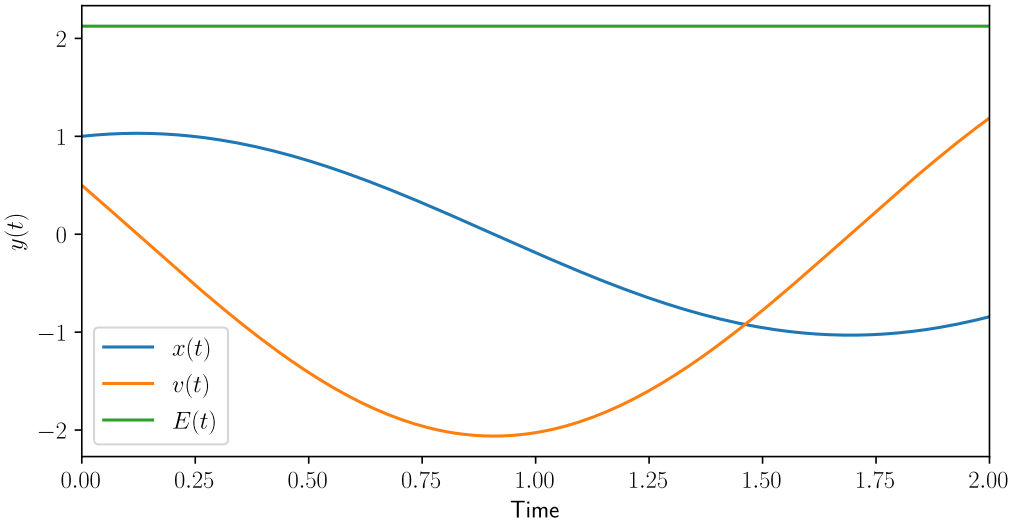
```
    # Compute the solution at time t
    z_t = np.diag(np.exp(eigs * t)) @ z0
    x_t = V @ z_t # Transform back to original coordinates
    X_t.append(x_t)

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(filename='_tmp_fig.svg')
```



5.3. 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.

5.3.1. 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 $n = 0, 1, 2, ...$, we obtain an approximate solution at discrete times t_n.

5.3.2. 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.

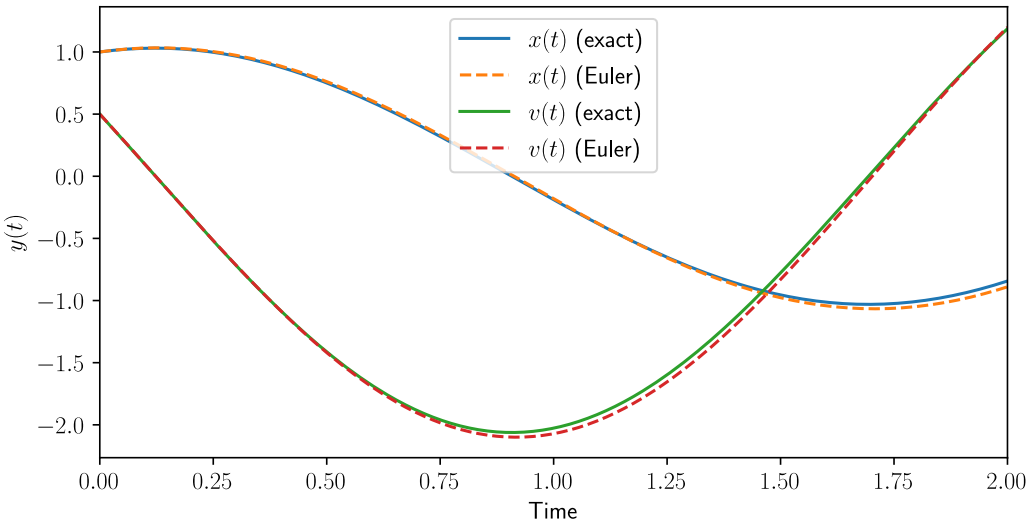
5.3.2.1. Example: Harmonic Oscillator with Euler Method

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(filename='_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 $|1 - \lambda h| \geq 1$).

5.4. 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:

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

5.4.1. Time-Dependent Hamiltonian

When $\hat{H}(t)$ 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 $\hat{H}(t)$ 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 $|\psi(t_n)\rangle$.

5.4.2. 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.

5.5. 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.

6. 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

$$\begin{cases} \dot{x} = \dfrac{p}{m}, \\ \dot{p} = -\dfrac{\partial V(x)}{\partial x} \end{cases} \tag{6.1}$$

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

6.1. 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 6.1. 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**.

6.2. 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**.

6.3. 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:

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

6.3.1. 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 $i = 0, \ldots, N_x - 1$
- $p_j = p_0 + j \cdot \Delta p$, for $j = 0, \ldots, N_p - 1$

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:

$$D_x = \frac{1}{2\Delta x}\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ -1 & 0 & 1 & \cdots & 0 \\ 0 & -1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & 1 \\ 0 & 0 & \cdots & -1 & 0 \end{pmatrix} \tag{6.2}$$

Analogously:

$$D_p = \frac{1}{2\Delta p}\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ -1 & 0 & 1 & \cdots & 0 \\ 0 & -1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & 1 \\ 0 & 0 & \cdots & -1 & 0 \end{pmatrix} \tag{6.3}$$

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

6.3.2. 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:

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

Then the full Liouville matrix L becomes:

$$L = (I_x \otimes D_p) \cdot (\partial_x V \otimes I_p) - (D_x \otimes I_p) \cdot (I_x \otimes P) \tag{6.4}$$

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.

6.4. Time Evolution

We can evolve the discretized density using an ODE solver:

$$\frac{d\vec{\rho}}{dt} = L\vec{\rho} \tag{6.5}$$

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 5.

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.

6.5. 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 6.2, Equation 6.3, and Equation 6.4. We can take advantage of the tools provided by `scipy.sparse` to create the sparse matrices efficiently. We start by importing the necessary libraries

```
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:

```
N_x = 101 # Number of grid points in position space
N_px = 101 # Number of grid points in momentum space
x_bound = 2 # Position space boundary
px_bound = 2 # 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, [x_op, Ipx]).todia()
px = ft.reduce(sparse.kron, [Ix, px_op]).todia()

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

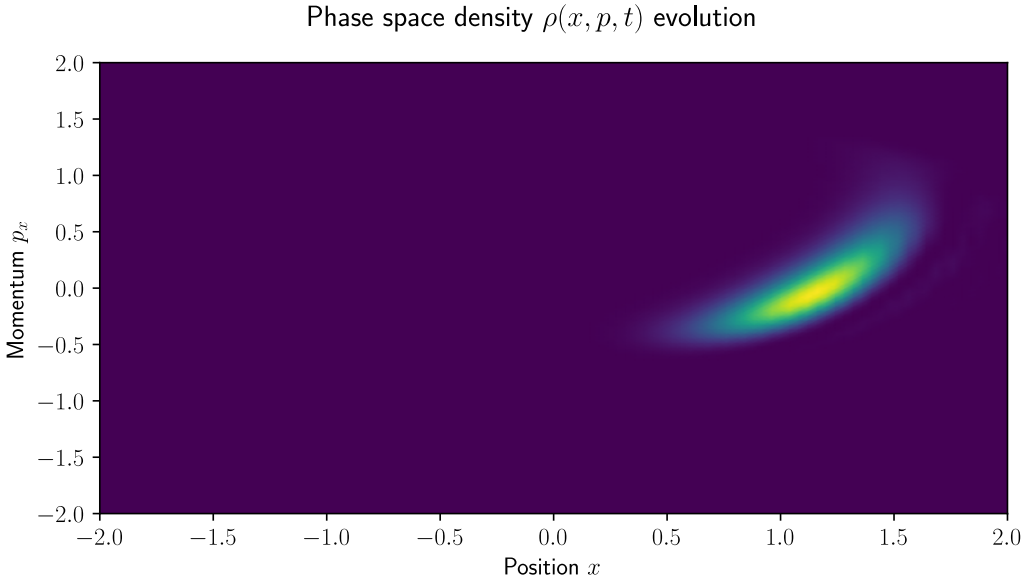
We can now compute the time evolution defined by Equation 6.5 by using the Euler method described in Section 5.3:

```
m = 0.5 # Mass of the particle
k = 2.0 # Spring constant
G = 0.3 # 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, 50000)

# Initial state: we will use a Gaussian wave packet to avoid singularities
x_0 = np.sqrt(gaussian(x_list, 1.0, 0.2)) * np.sqrt(dx)
```

```
p_0 = np.sqrt(gaussian(px_list, 0.1, 0.2)) * np.sqrt(dpx)
rho_0 = np.kron(x_0, p_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)**2,
                    shading='gouraud', rasterized=True)
ax.set_xlabel("Position $x$")
ax.set_ylabel("Momentum $p_x$")

# Show in Quarto
plt.savefig('_tmp_fig.svg')
plt.close(fig)
SVG(filename='_tmp_fig.svg')
```


Phase space density $\rho(x, p, t)$ evolution
Momentum p_x
Position x
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

7. 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 $\{|k\rangle : k = 1, ..., d\}$ 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

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

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.

7.1. 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

```
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]])
```

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

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

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 ($r = 0, \pm 1, \pm 2, ...$).

```
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 $M = \texttt{np.matmul(a, np.conj(a.T))}$ 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, $M_2 = \texttt{np.conj(a.T) * a + np.eye(d)}$, 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.

7.2.1. 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.

```
# 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.        ])
```

7.3. Partial Trace

In Section 3.1.4, 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:

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

```python
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]])
```

7.4. 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.

8. Introduction to QuTiP

The QuTiP package can be imported with

```
import qutip
```

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:     3.13.3
Number of CPUs:     4
BLAS Info:          Generic
INTEL MKL Ext:      None
Platform Info:      Linux (x86_64)
Installation path:  /opt/hostedtoolcache/Python/3.13.3/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()`
```

8.1. 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.

8.1.1. 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.

8.1.1.1. 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.

```python
# Define the annihilation operator for d-dimensional Hilbert space
d = 10

a = qutip.destroy(d)

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

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


Quantum object: dims=[[10], [10]], shape=(10, 10), type='oper', dtype=Dia, isherm=False
Qobj data =
[[0.         1.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         1.41421356 0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         1.73205081 0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         2.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         2.23606798
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  2.44948974 0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         2.64575131 0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         2.82842712 0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         3.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]]
```

8.1.1.2. 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.]]
```

8.1.2. 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.

8.1.2.1. 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
```

8.2. 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.

8.2.1. 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.

8.2.2. Superposition States

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

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

```
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=[[10], [1]], shape=(10, 1), type='ket', dtype=Dense
Qobj data =
[[0.70710678]
 [0.70710678]
 [0.        ]
 [0.        ]
 [0.        ]
 [0.        ]
 [0.        ]
 [0.        ]
 [0.        ]
 [0.        ]]
```

8.2.3. 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=[[10], [1]], shape=(10, 1), type='ket', dtype=Dense
Qobj data =
[[7.26149037e-01]
 [5.80919230e-01]
 [3.28617541e-01]
 [1.51781941e-01]
 [6.07127755e-02]
 [2.17212702e-02]
 [7.09408207e-03]
 [2.14540751e-03]
 [6.04780881e-04]
 [1.71316475e-04]]
```

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

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

```
np.float64(0.9999999837403134)
```

8.2.4. Spin States

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

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

8.2.5. Density Matrices

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

8.2.5.1. Creating a Density Matrix

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

```
# 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=[[10], [10]], shape=(10, 10), type='oper', dtype=Dense, isherm=True
Qobj data =
[[0.5 0.5 0.  0.  0.  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.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]]
```

8.2.6. 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 $m_2 = 1$.

```
d = 10
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=[[10, 10, 3], [10, 10, 3]], shape=(300, 300), type='oper', dtype=Dense,
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=[[10, 3], [10, 3]], shape=(30, 30), type='oper', dtype=Dense, isherm=Tr
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. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 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. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 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=[[10], [10]], shape=(10, 10), type='oper', dtype=Dense, isherm=True
Qobj data =
[[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.]]
```

8.3. 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)
```

8.4. 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>:
```

```
0.6399999989126254


The squared modulus of alpha is:


0.6400000000000001
```

Bibliography

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.

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.

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.

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.

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 5.1 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

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

```
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()
```

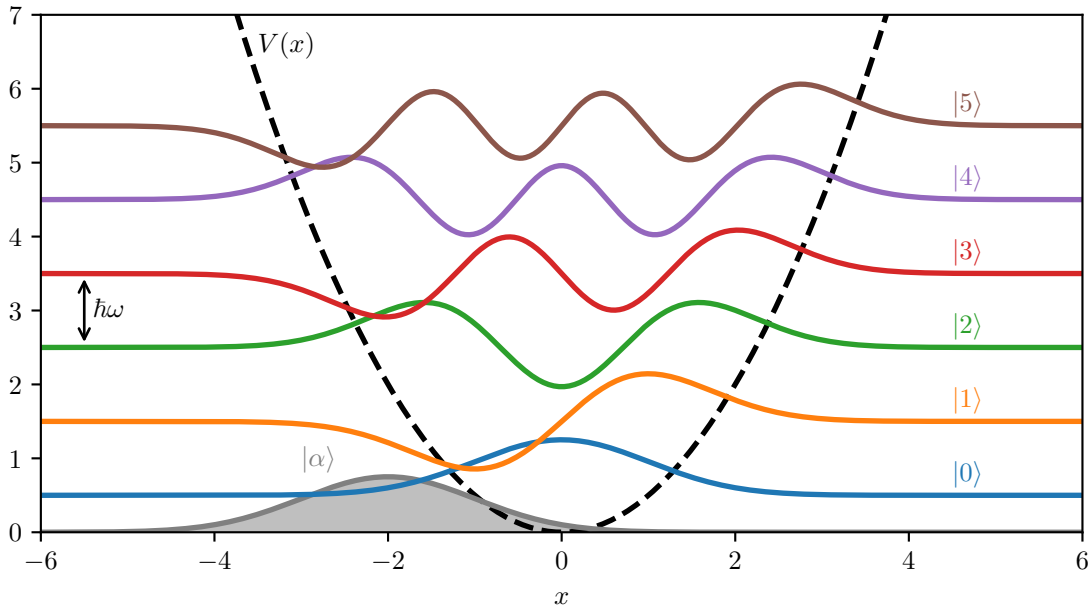


Figure A.1.: First eigenstates of one-dimensional the quantum harmonic oscillator, each of them vertically shifted by the corresponding eigenvalue. The grey-filled curve corresponds to a coherent state with $\alpha = -\sqrt{2}$. The used parameter are $m = 1$, $\omega = 1$, and $\hbar = 1$.

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 ($E_0 = \hbar\omega/2$).