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

20-05-2025

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

Table of contents

1 Home Page

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

2 Introduction

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.

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
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.

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

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

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

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

```
Sum using list: 499999.50000000006, Time taken: 2.3680 milliseconds
Sum using numpy: 499999.5, Time taken: 0.2820 milliseconds
```

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

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

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

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

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`

```
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:

```
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):  0.049114227294921875
Numba time (ms):  0.1678466796875
```

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.

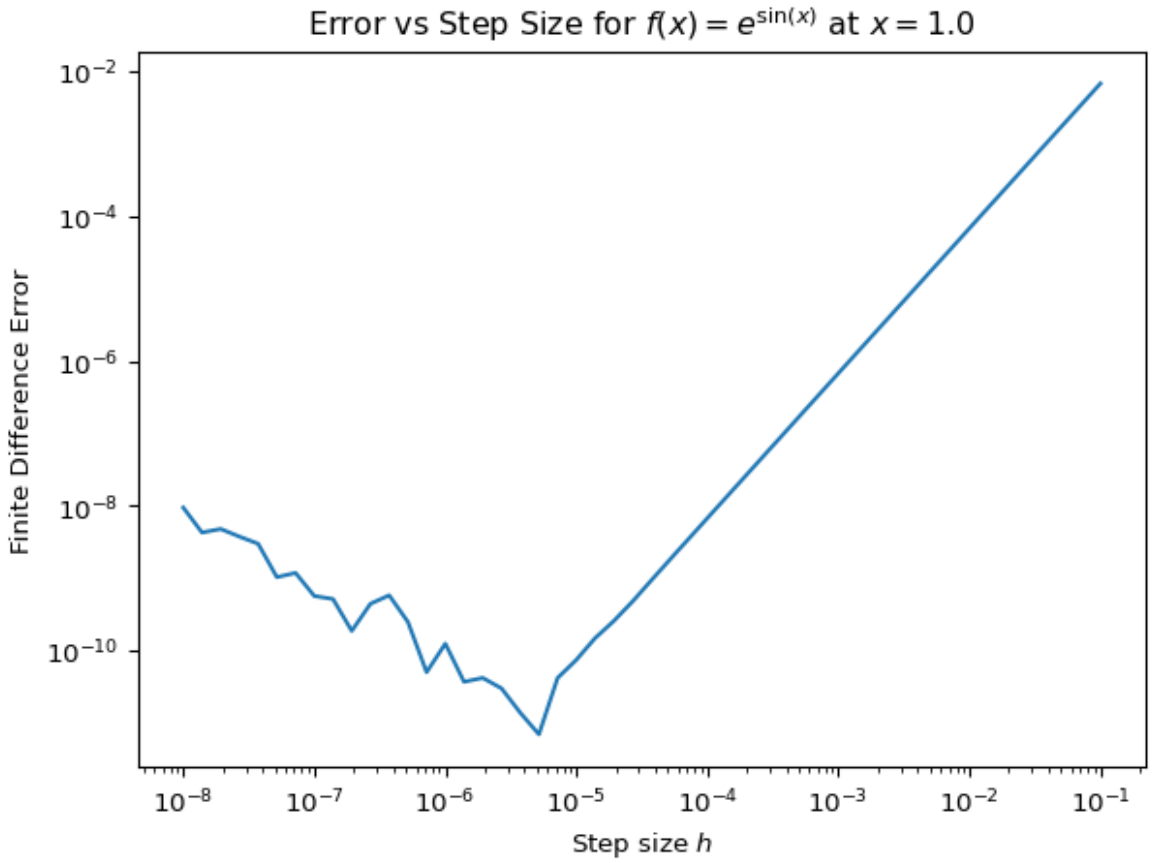
4.2.1.3 Comparing Accuracy: AD vs Finite Differences

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
plt.loglog(hs, errors_fd)
plt.xlabel("Step size $h$")
plt.ylabel("Finite Difference Error")
plt.title(r"Error vs Step Size for $f(x) = e^{\sin(x)}$ at $x=1.0$")
plt.show()
```

```
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 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.

Bibliography

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.

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.

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.