ALICE & BOB

dynamiqs: a library for GPU-accelerated and differentiable simulations of quantum systems

Ronan Gautier

QuantumÉTS, 15th May 2024

dynamiqs in a nutshell

dynamiqs

A **Python library** to simulate

- The Schrödinger equation
- The Lindblad master equation
- The stochastic master equation

> On CPU and **GPU** → speedup for large systems

> With **differentiable** solvers → to compute gradients

GPU-accelerated solvers

> To simulate **large quantum systems** → **30-60x** faster than QuTiP

e.g. simulate a CNOT between two cat qubits

$$H = g(a + a^\dagger)b^\dagger b$$
$$L = \sqrt{\kappa_2}(a^2 - \alpha^2)$$

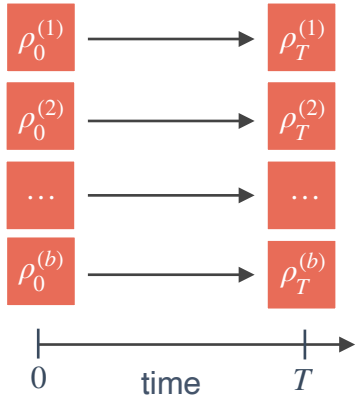
dimension	QuTiP on CPU	dynamiqs on GPU
$32 \times 32 = 1024$	1 minute 30 seconds	2.4 seconds
$64 \times 64 = 4096$	6 hours	6 minutes

CPU: AMD Ryzen 7 7700X
GPU: NVIDIA RTX 4090

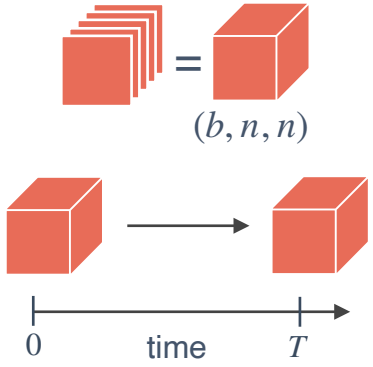
> To simulate the same system **with different parameters**

for loop
b simulations of size (n, n)

batching
1 simulation of size (b, n, n)



VS



e.g. 10,000 simulations of a 5-levels transmon (sweep δ and κ)

$$H = -\delta/2\ a^{\dagger 2}a^2 + \epsilon^* a + \epsilon a^\dagger$$
$$L = \sqrt{\kappa} a$$

for loop on CPU	dynamiqs on GPU
7 minutes	10 seconds

GPU-accelerated solvers

> Simulating a quantum systems = **matrix products**

↳ with ODE solvers, propagator, Monte-Carlo, etc…

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

e.g. 3 coupled oscillators truncated at 32
↳ **one billion elements**

Example: simulate Lindblad $\frac{\mathrm{d}\rho_t}{\mathrm{d}t} = \mathscr{L}_t(\rho_t)$

with an ODE solver

$$\rho_{t+\mathrm{d}t} = \rho_t + \mathscr{L}_t(\rho_t)\mathrm{d}t + \mathscr{O}(\mathrm{d}t^2) \quad \text{(Euler method)}$$

$$\approx \rho_t - i(H_t\rho_t - \rho_t H_t) + \sum_k \left(L_k \rho_t L_k^\dagger - \frac{1}{2} L_k^\dagger L_k \rho_t - \frac{1}{2} \rho_t L_k^\dagger L_k \right)$$

> Leverage **specialised hardware**

CPU

can perform many different computations

GPU

can only perform basic operations

GPU-accelerated solvers

> Simulating a quantum systems = **matrix products**

↳ with ODE solvers, propagator, Monte-Carlo, etc…

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

e.g. 3 coupled oscillators truncated at 32
↳ **one billion elements**

Example: simulate Lindblad $\frac{\mathrm{d}\rho_t}{\mathrm{d}t} = \mathscr{L}_t(\rho_t)$

with an ODE solver

$$\rho_{t+\mathrm{d}t} = \rho_t + \mathscr{L}_t(\rho_t)\mathrm{d}t + \mathscr{O}(\mathrm{d}t^2) \quad \text{(Euler method)}$$

$$\approx \rho_t - i(H_t\rho_t - \rho_t H_t) + \sum_k \left(L_k \rho_t L_k^\dagger - \frac{1}{2} L_k^\dagger L_k \rho_t - \frac{1}{2} \rho_t L_k^\dagger L_k \right)$$

> Leverage **specialised hardware**

CPU

can perform many different computations

can only perform basic operations

GPU

GPU-accelerated solvers

> Simulating a quantum systems = **matrix products**

↳ with ODE solvers, propagator, Monte-Carlo, etc…

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

e.g. 3 coupled oscillators truncated at 32
↳ **one billion elements**

Example: simulate Lindblad $\frac{\mathrm{d}\rho_t}{\mathrm{d}t} = \mathscr{L}_t(\rho_t)$

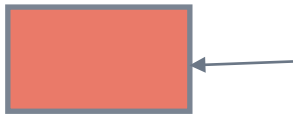
with an ODE solver

$$\rho_{t+\mathrm{d}t} = \rho_t + \mathscr{L}_t(\rho_t)\mathrm{d}t + \mathscr{O}(\mathrm{d}t^2) \quad \text{(Euler method)}$$

$$\approx \rho_t - i(H_t\rho_t - \rho_t H_t) + \sum_k \left(L_k \rho_t L_k^\dagger - \frac{1}{2} L_k^\dagger L_k \rho_t - \frac{1}{2} \rho_t L_k^\dagger L_k \right)$$

> Leverage **specialised hardware**

CPU



can perform many different computations

GPU

can only perform basic operations

GPU-accelerated solvers

> Simulating a quantum systems = **matrix products**

↳ with ODE solvers, propagator, Monte-Carlo, etc…

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

e.g. 3 coupled oscillators truncated at 32

↳ **one billion elements**

Example: simulate Lindblad $\frac{\mathrm{d}\rho_t}{\mathrm{d}t} = \mathscr{L}_t(\rho_t)$

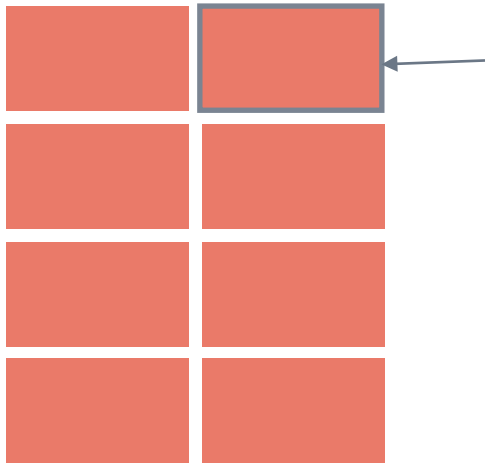
with an ODE solver

$$\rho_{t+\mathrm{d}t} = \rho_t + \mathscr{L}_t(\rho_t)\mathrm{d}t + \mathscr{O}(\mathrm{d}t^2) \quad \text{(Euler method)}$$

$$\approx \rho_t - i(H_t\rho_t - \rho_t H_t) + \sum_k \left(L_k \rho_t L_k^\dagger - \frac{1}{2} L_k^\dagger L_k \rho_t - \frac{1}{2} \rho_t L_k^\dagger L_k \right)$$

> Leverage **specialised hardware**

CPU

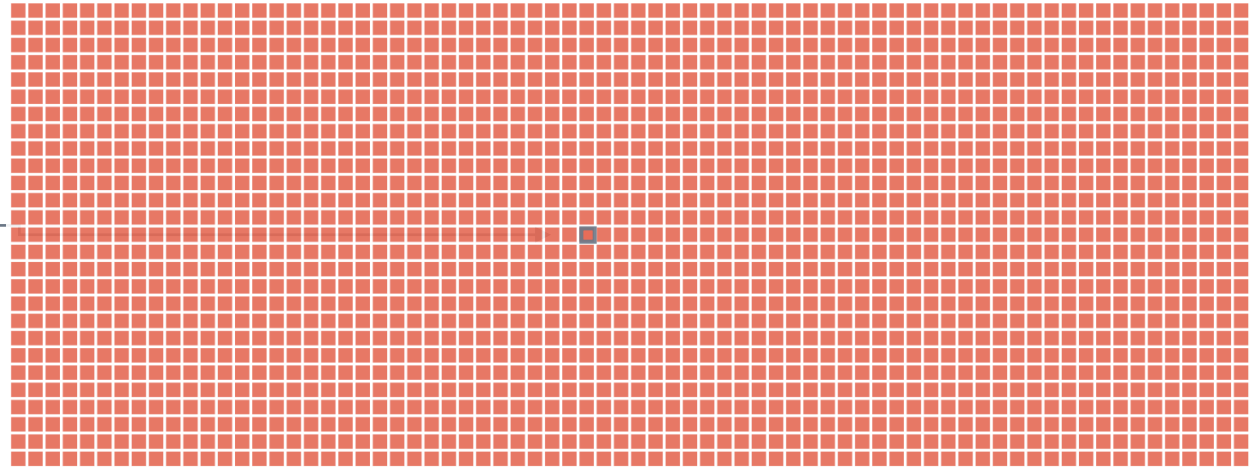


(8 - 32 cores)

can perform many different computations

can only perform basic operations

GPU



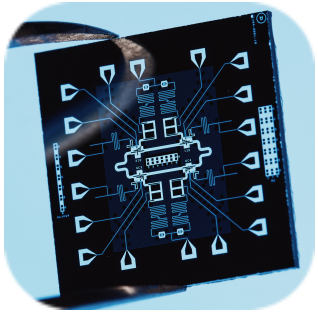
(thousands of cores)

Differentiable solvers

> **Calibration and control** → optimisation problems, e.g. **parameters estimation**

quantum system → **measure data** → **fit parameters**

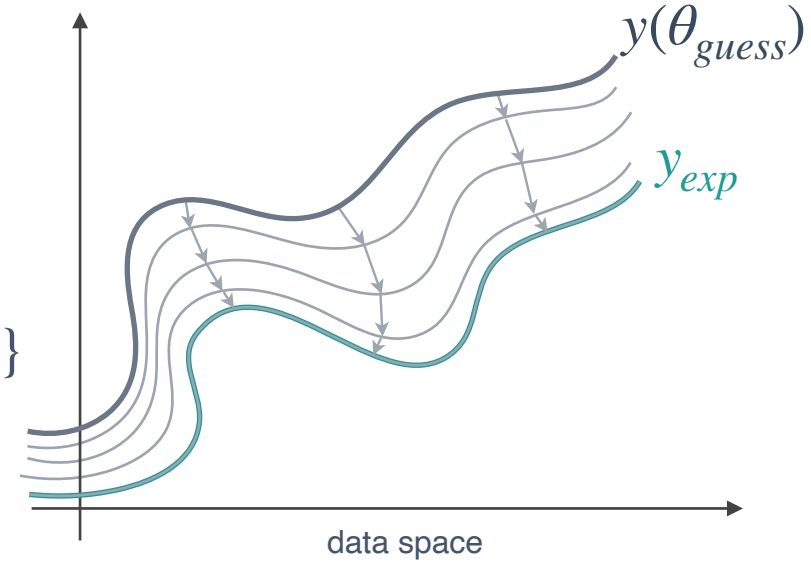
minimise $f(\theta) = \|y(\theta) - y_{exp}\|^2$

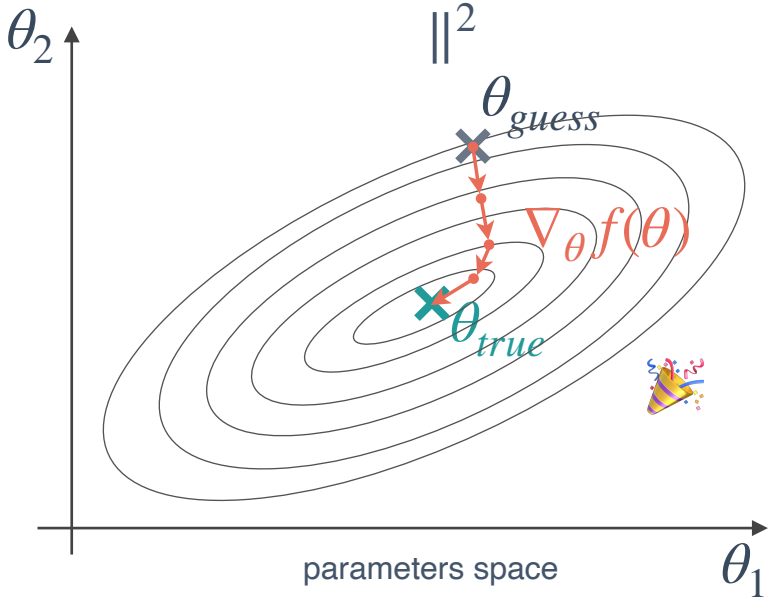


model $H(\theta)$, $\rho_0(\theta)$, $\{L_k(\theta)\}$

with parameters

$\theta = (\theta_1, \ldots, \theta_n) \in \mathbb{R}^n$



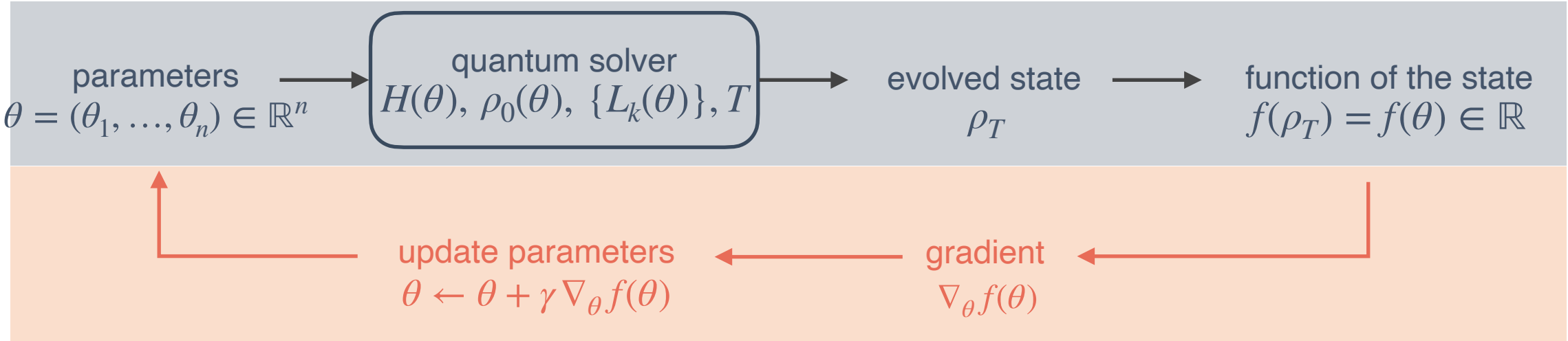


> Efficient parameter search: need **the gradient**

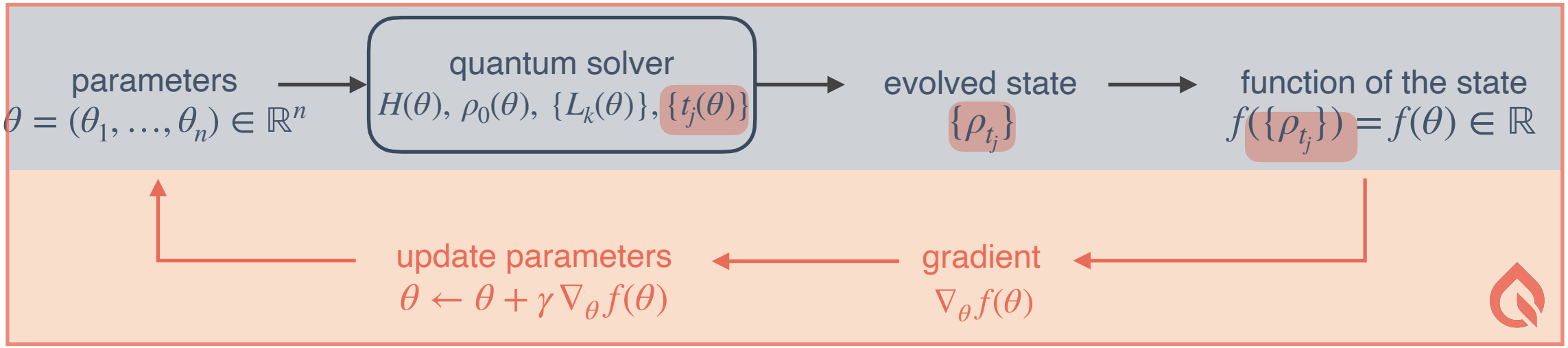
> Many applications: quantum optimal control, sensitivity analysis, state tomography, etc…

Leung et al. (2017)

Differentiable solvers



Differentiable solvers



> Project philosophy: fast and reliable **building block**

> Computing **the gradient**

- Automatic differentiation → fast but memory $\mathcal{O}(n_{steps})$
- Adjoint state method → slower but memory $\mathcal{O}(1)$
- Checkpointing → tradeoff between speed and memory

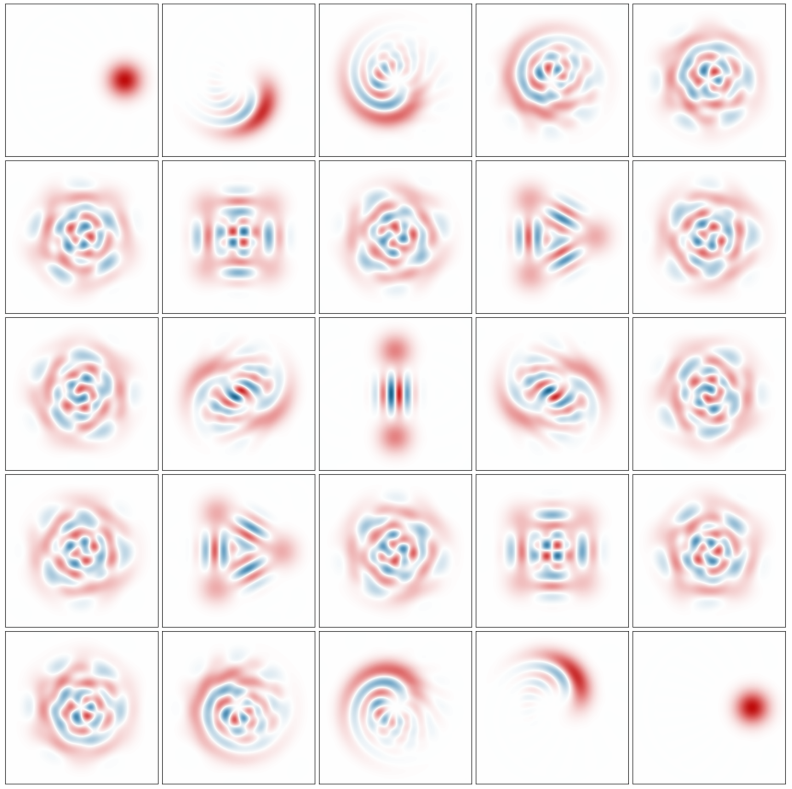
Example

```python
import dynamiqs as dq
import numpy as np

# define model
n = 16                                  # Hilbert space dimension
a = dq.destroy(n)                       # annihilation operator
H = dq.dag(a) @ dq.dag(a) @ a @ a       # Kerr Hamiltonian
psi0 = dq.coherent(n, 2.0)              # initial state
tsave = np.linspace(0, np.pi, 101)      # save times

# run simulation
result = dq.sesolve(H, psi0, tsave)

# plot results
dq.plot_wigner_mosaic(result.states, n=25, nrows=5, xmax=3.5)
```



> **QuTiP-like API**

> All functions work with QuTip objects

> Running on a GPU = one extra line

Under the hood

> Linear algebra on GPUs + automatic differentiation → **machine learning community**

> **JAX** by Google: « *NumPy on GPU with automatic differentiation* »

Python code

foo.py

Python file

↓

compile

↓

foo.pyc

bytecode

↓

Python Virtual Machine

(reads and executes the bytecode line-by-line)

- ✗ interpretation overhead
- ✗ dynamic type checking
- ✗ no low-level optimizations

PyThon code + JAX

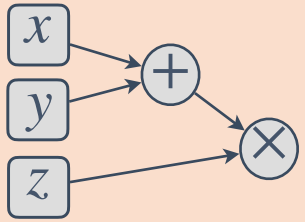
foo.py

Python file

↓

tracing

↓



computation graph

optimize

↓

compile

↓

CPU/GPU optimized machine-level code

- ✅ operation fusion
- ✅ memory access optimizations
- ✅ loop unrolling

XLA

And more!

> Solvers

- ODE solvers from the Diffrax library
  ↳ modern ODE solvers (Tsit5, PID controllers)
  ↳ optimal online checkpointing for gradient computation
  ↳ implicit ODE solvers
  ↳ adaptive step size SME solvers
- Quantum-tailored solvers
  ↳ preserve state trace and positivity
- Easily implement your own solvers
- Custom sparse format (coming soon)
  ↳ more than x10 speedup for large systems
- Krylov subspace methods for propagators (coming soon)

> Gradient

- Compute gradient w.r.t. evolution time
- Compute higher order derivatives
  ↳ e.g. the Hessian

> Utilities

- Support multiple Hamiltonian formats
  ↳ constant, piecewise-constant, callable, …
- Support for time-dependent jump operators
- User-defined save function during the evolution
  ↳ e.g. partial trace, purity, Fock population, etc…
- Beautiful plotting functions
- All functions work on batched arrays
- Parallelisation across multiple CPUs/GPUs

> Library

- Modern software development practices
  ↳ e.g. analytical tests in CI for states and gradients
- Carefully written documentation

And more!

> Solvers

- ODE solvers from the **Diffrax library**
  ↳ modern ODE solvers (Tsit5, PID controllers)
  ↳ optimal online checkpointing for gradient computation
  ↳ implicit ODE solvers
  ↳ adaptive step size SME solvers
- Quantum-tailored solvers
  ↳ preserve state trace and positivity
- Easily implement your own solvers
- **Custom sparse format** (coming soon)
  ↳ more than x10 speedup for large systems
- Krylov subspace methods for propagators (coming soon)

> Gradient

- Compute gradient w.r.t. evolution time
- Compute higher order derivatives
  ↳ e.g. the Hessian

> Utilities

- Support multiple Hamiltonian formats
  ↳ constant, piecewise-constant, callable, …
- Support for time-dependent jump operators
- User-defined save function during the evolution
  ↳ e.g. partial trace, purity, Fock population, etc…
- Beautiful plotting functions
- All functions work on batched arrays
- Parallelisation across multiple CPUs/GPUs

> Library

- Modern software development practices
  ↳ e.g. analytical tests in CI for states and gradients
- Carefully written documentation

dynamiqs

GPU-accelerated and differentiable quantum simulations

> An **open-source Python library**

> Developed by **physicists** and **developers**

> Available on **GitHub**

https://github.com/dynamiqs/dynamiqs