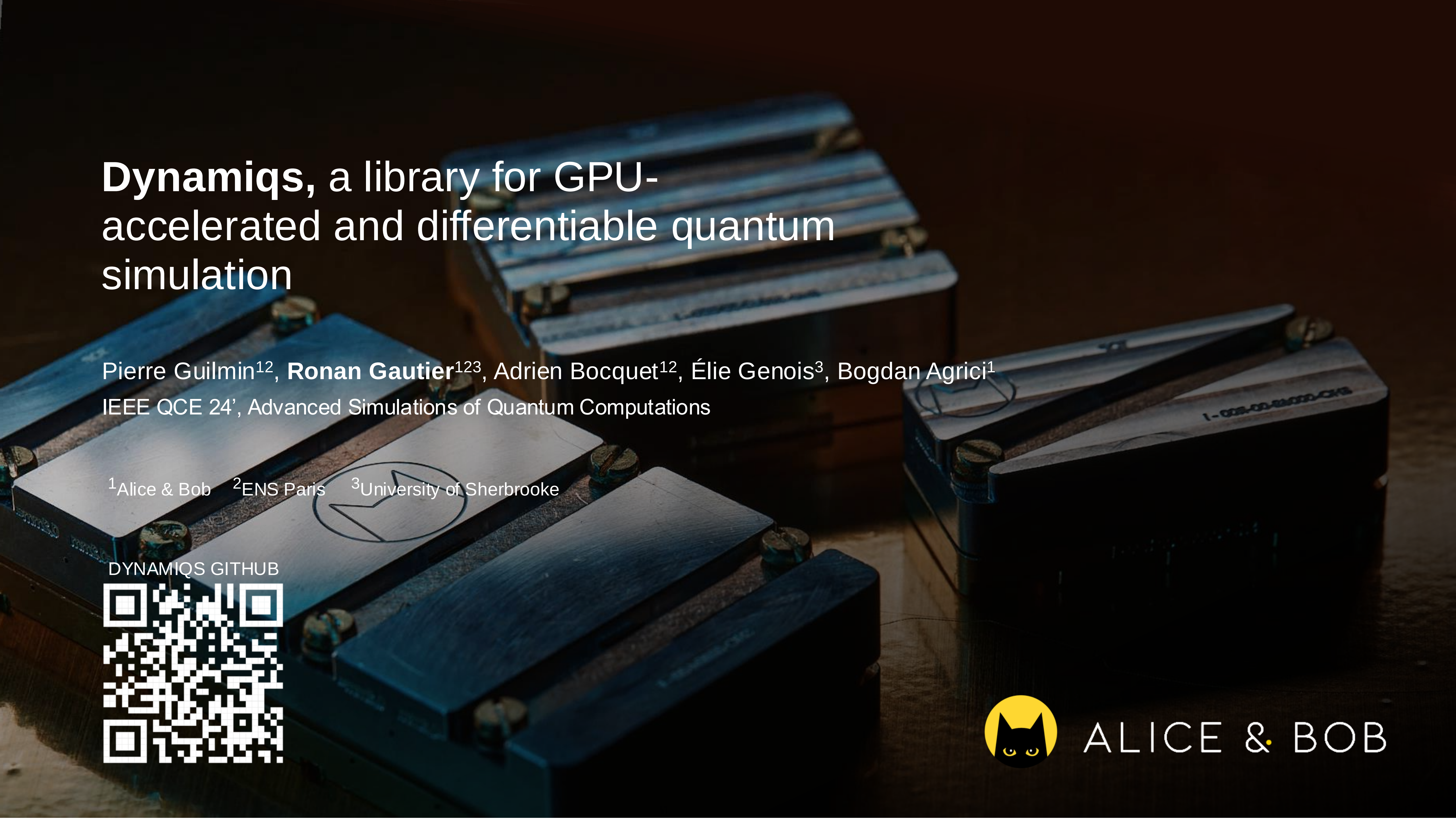


Dynamiqs, a library for GPU-accelerated and differentiable quantum simulation

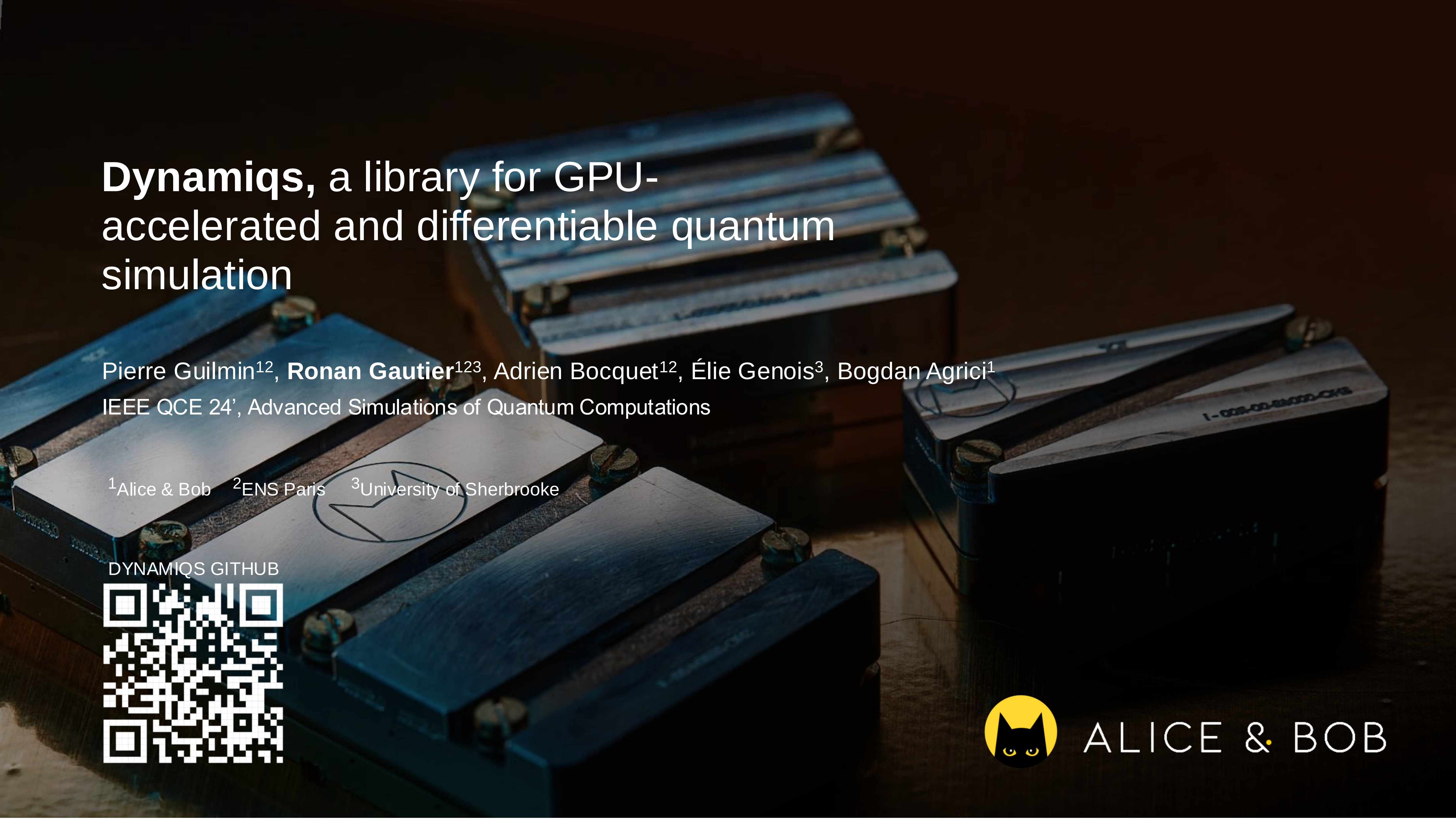
Pierre Guilmin[1,2], **Ronan Gautier**[1,2,3], Adrien Bocquet[1,2], Élie Genois[3], Bogdan Agrici[1]

IEEE QCE 24’, Advanced Simulations of Quantum Computations

[1]Alice & Bob [2]ENS Paris [3]University of Sherbrooke

DYNAMIQS GITHUB

ALICE & BOB



Dynamiqs, a library for GPU-accelerated and differentiable quantum simulation

Pierre Guilmin[1,2], **Ronan Gautier**[1,2,3], Adrien Bocquet[1,2], Élie Genois[3], Bogdan Agrici[1]

IEEE QCE 24', Advanced Simulations of Quantum Computations

[1]Alice & Bob [2]ENS Paris [3]University of Sherbrooke

DYNAMIQS GITHUB

ALICE & BOB

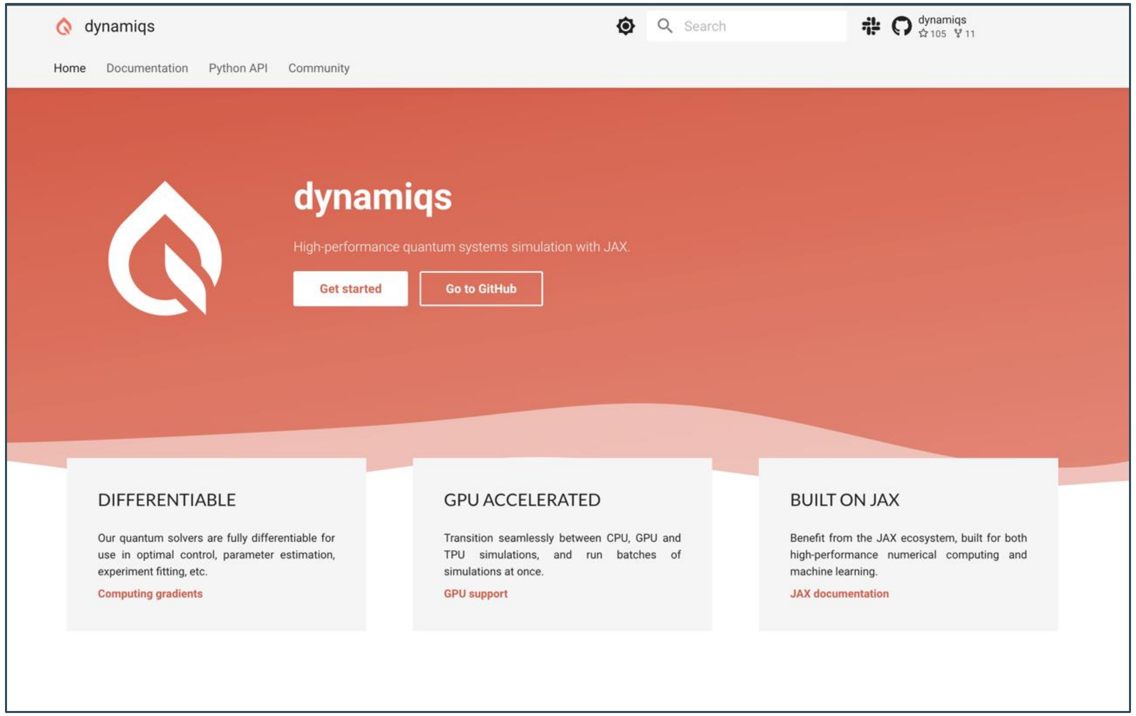
Dynamiqs in a nutshell

An open-source Python library based on JAX, for the simulation of

- the Schrödinger equation
- the Lindblad master equation
- stochastic master equations
- …

With

- CPU and **GPU** support
- **Batching**
- End-to-end **differentiability**
- Tailored **sparse** support
- QuTiP-like **API**



dynamiqs

High-performance quantum systems simulation with JAX.

DIFFERENTIABLE

Our quantum solvers are fully differentiable for use in optimal control, parameter estimation, experiment fitting, etc.

GPU ACCELERATED

Transition seamlessly between CPU, GPU and TPU simulations, and run batches of simulations at once.

BUILT ON JAX

Benefit from the JAX ecosystem, built for both high-performance numerical computing and machine learning.

www.dynamiqs.org

Why need for high-performance simulations?

Time dynamics of quantum systems is essential for:

- Characterization
- Design
- Control optimization
- Understanding of physical phenomena
- …

Numerical integration gets harder with:

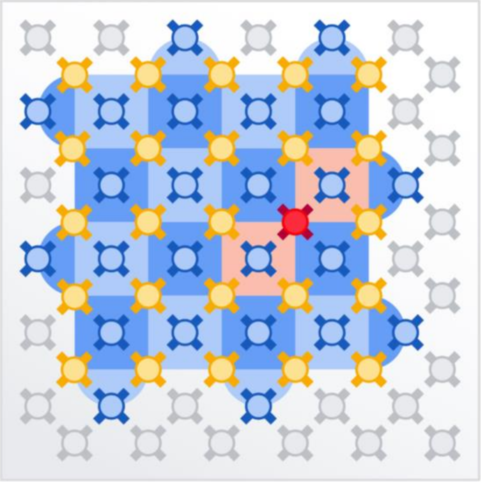
- **Large** Hilbert spaces (many ”small” or few “large” systems)
- **Open** systems (interacting with environment)
- **Fast** time-dependencies
- High-dimensional **parameter sweeps**

@ Alice & Bob: dissipative cat qubits

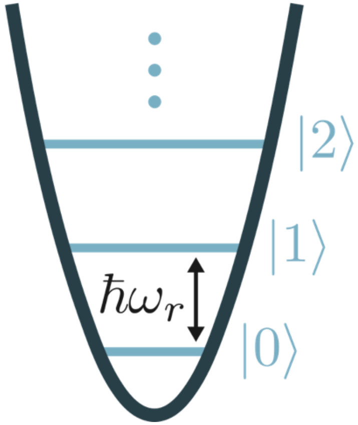
Simulating a CNOT = 2 coupled cavities (n=32) + 2 buffer modes (n=8)

➡ N = 32x32x8x8 = 65 536 ➡ 32GB / density matrix

Inherently dissipative + time-dependent



Surface code (Google Quantum AI)



Quantum harmonic oscillator

GPU-acceleration and performance

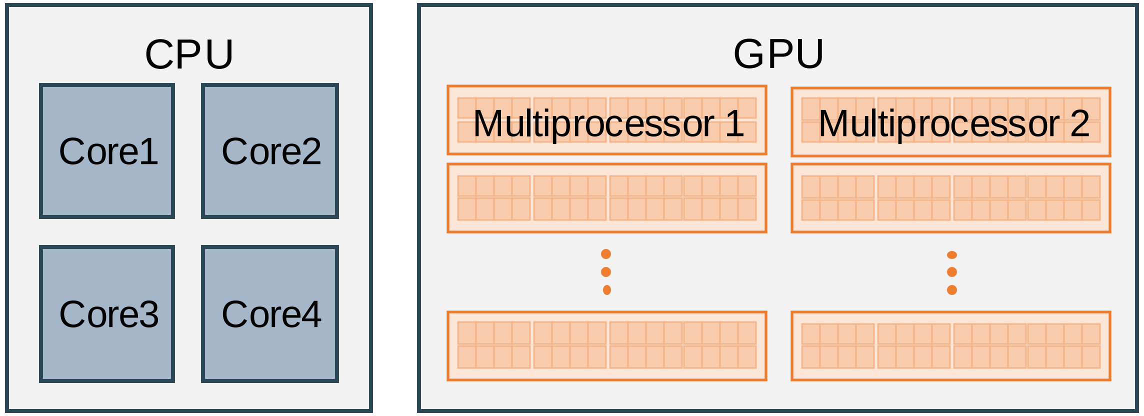
Why GPUs?

Bottleneck of solving a SE/ME/SME is **matrix products** (ODE solvers, propagator, Monte Carlo, …)

Example: Euler method for ME

$$\rho(t+dt) = \rho(t) - i[H, \rho(t)] + \sum \left(L\rho(t)L^\dagger - \frac{1}{2}\{L^\dagger L, \rho(t)\} \right)$$

- Leverage specialized hardware

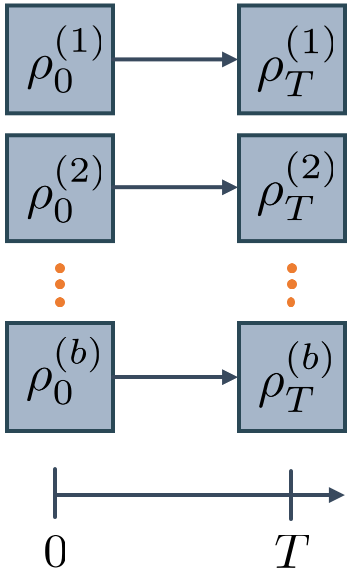


Batching simulations

Parameter sweeps are ubiquitous in characterization, design & control of quantum systems

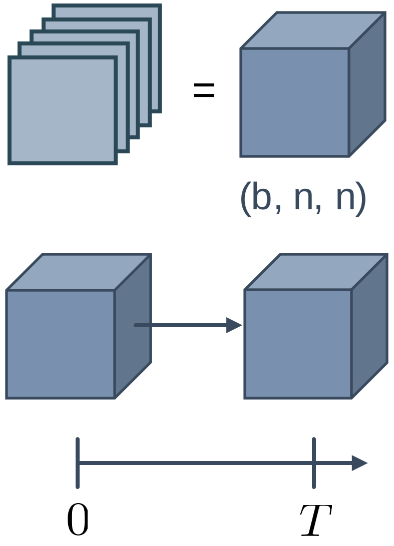
For loop

b simulations of size (n, n)



Batching

1 simulations of size (b, n, n)



Enabled by jax.vmap
\+ batched kernels (e.g. cuBLAS)

Example: Batched two-level system

$$H = \omega\sigma_z + \varepsilon\sigma_x$$

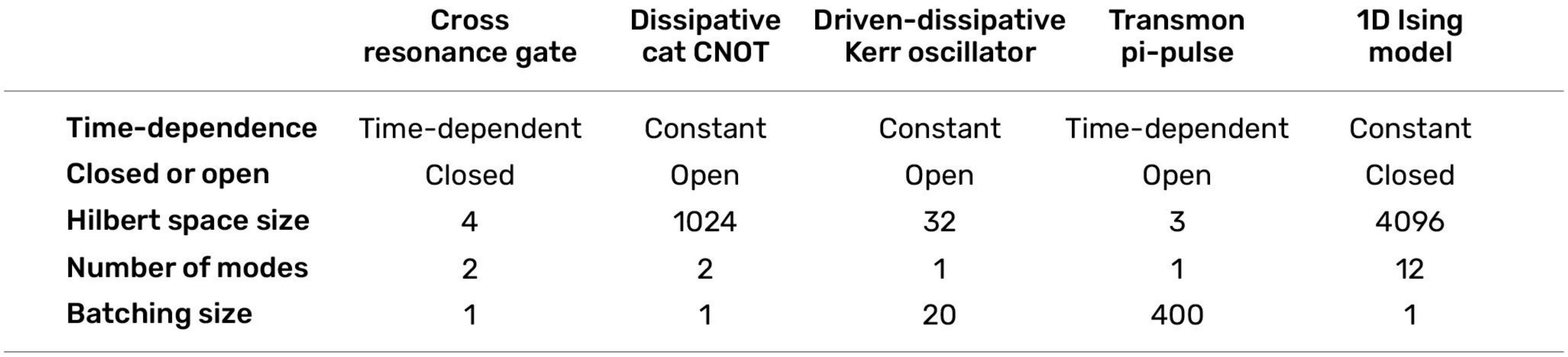
(sweep 3000 parameters)

- 2.59 s → 44.1 ms
- **x60 speedup** (only on CPU*)
- Concise code and practical

*CPU: Macbook Air M2

Benchmarking Dynamiqs

Set of representative benchmarks of time-dynamics simulations

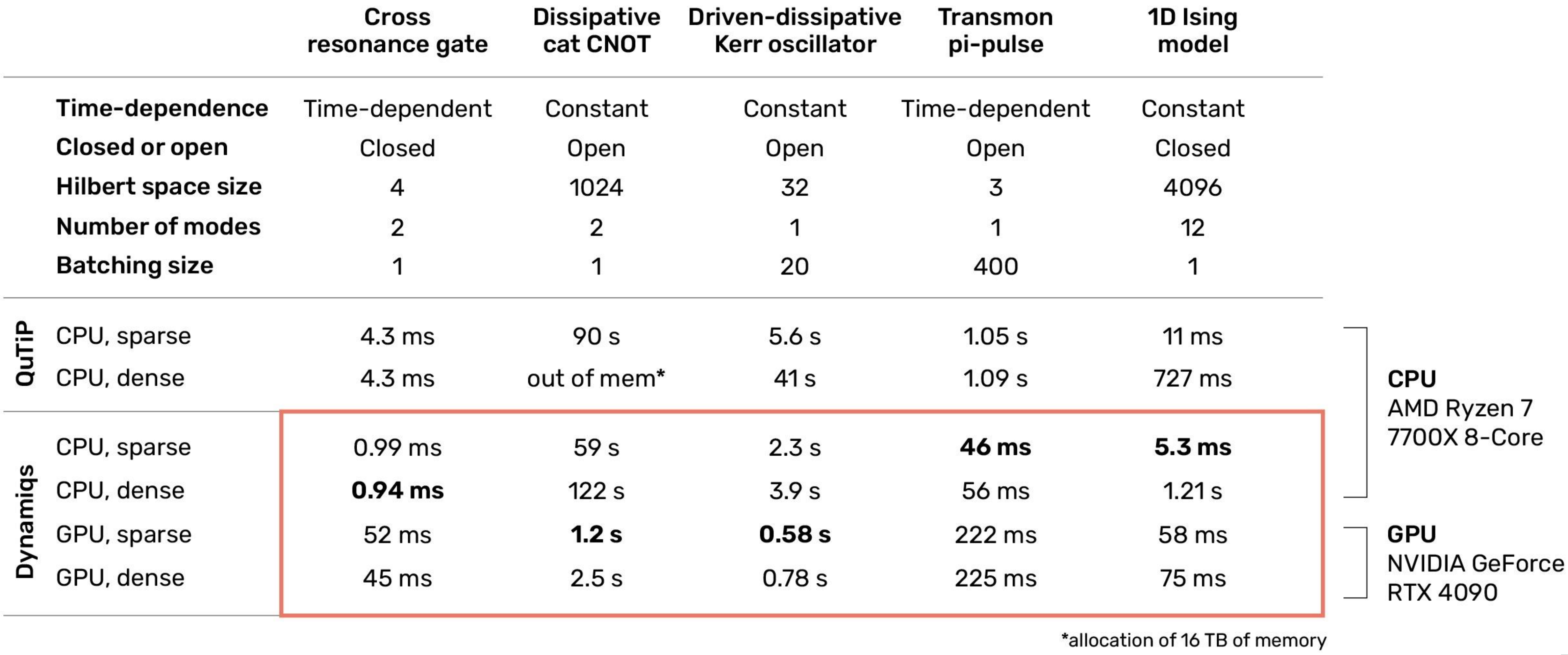


	Cross resonance gate	Dissipative cat CNOT	Driven-dissipative Kerr oscillator	Transmon pi-pulse	1D Ising model
Time-dependence	Time-dependent	Constant	Constant	Time-dependent	Constant
Closed or open	Closed	Open	Open	Open	Closed
Hilbert space size	4	1024	32	3	4096
Number of modes	2	2	1	1	12
Batching size	1	1	20	400	1

Full benchmarks: https://github.com/dynamiqs/dynamiqs/pull/704

Benchmarking Dynamiqs

Set of representative benchmarks of time-dynamics simulations



		Cross resonance gate	Dissipative cat CNOT	Driven-dissipative Kerr oscillator	Transmon pi-pulse	1D Ising model
	Time-dependence	Time-dependent	Constant	Constant	Time-dependent	Constant
	Closed or open	Closed	Open	Open	Open	Closed
	Hilbert space size	4	1024	32	3	4096
	Number of modes	2	2	1	1	12
	Batching size	1	1	20	400	1
QuTiP	CPU, sparse	4.3 ms	90 s	5.6 s	1.05 s	11 ms
	CPU, dense	4.3 ms	out of mem*	41 s	1.09 s	727 ms
Dynamiqs	CPU, sparse	0.99 ms	59 s	2.3 s	**46 ms**	**5.3 ms**
	CPU, dense	**0.94 ms**	122 s	3.9 s	56 ms	1.21 s
	GPU, sparse	52 ms	**1.2 s**	**0.58 s**	222 ms	58 ms
	GPU, dense	45 ms	2.5 s	0.78 s	225 ms	75 ms

CPU
AMD Ryzen 7
7700X 8-Core

GPU
NVIDIA GeForce
RTX 4090

*allocation of 16 TB of memory

Full benchmarks: https://github.com/dynamiqs/dynamiqs/pull/704

Why is Dynamiqs fast?

No Liouvillian vectorization

- Vectorized: $\mathcal{L}|\rho\rangle\rangle$ ➡ $\mathcal{O}(N^4)$
- Linear map: $\mathcal{L}(\rho) = -i[H,\rho] + \sum L\rho L^\dagger - \frac{1}{2}\{L^\dagger L, \rho\}$ ➡ $\mathcal{O}(N^3)$

(equivalent if sparse layout)

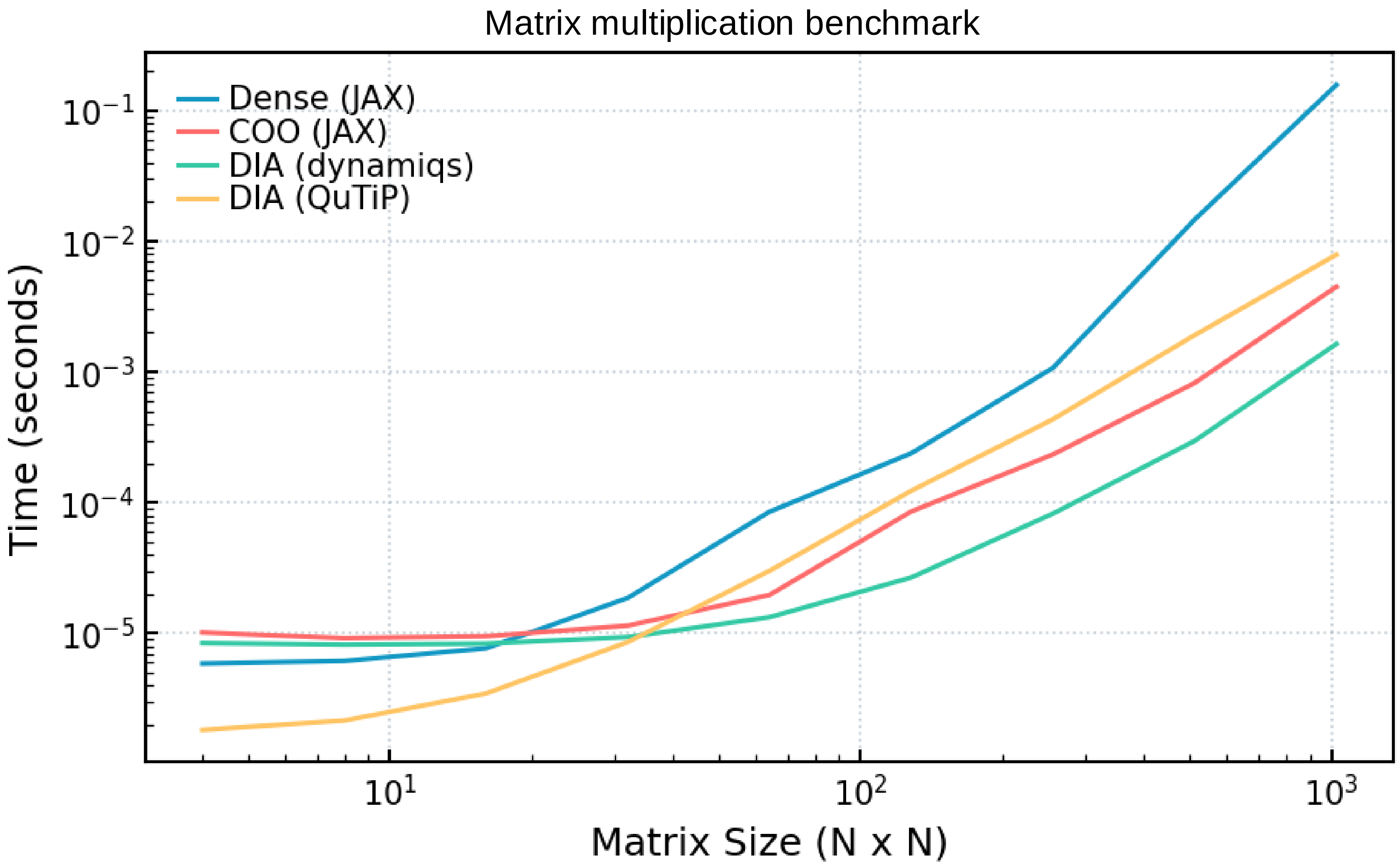
Sparse DIA format

Operators are often polynomial in $a, a^\dagger$

$$a = \begin{pmatrix} 0 & \sqrt{1} & & & & \\ & 0 & \sqrt{2} & & & \\ & & 0 & & & \\ & & & \ddots & & \\ & & & & 0 & \sqrt{n-1} \\ & & & & & 0 \end{pmatrix}$$

➡ Efficient sparse storage with (diagonals, offsets) = DIA format
+ coalescent memory accesses
- uncommon storage

Why is Dynamiqs fast?



Why is Dynamiqs fast?

No Liouvillian vectorization

- Vectorized: $\mathcal{L}|\rho\rangle\rangle$ ➡ $\mathcal{O}(N^4)$
- Linear map: $\mathcal{L}(\rho) = -i[H,\rho] + \sum L\rho L^\dagger - \frac{1}{2}\{L^\dagger L, \rho\}$ ➡ $\mathcal{O}(N^3)$

(equivalent if sparse layout)

Sparse DIA format

Operators are often polynomial in $a, a^\dagger$

$$a = \begin{pmatrix} 0 & \sqrt{1} & & & \\ & 0 & \sqrt{2} & & \\ & & 0 & & \\ & & & \ddots & \\ & & & 0 & \sqrt{n-1} \\ & & & & 0 \end{pmatrix}$$

➡ Efficient sparse storage with (diagonals, offsets) = DIA format
+ coalescent memory accesses
- uncommon storage

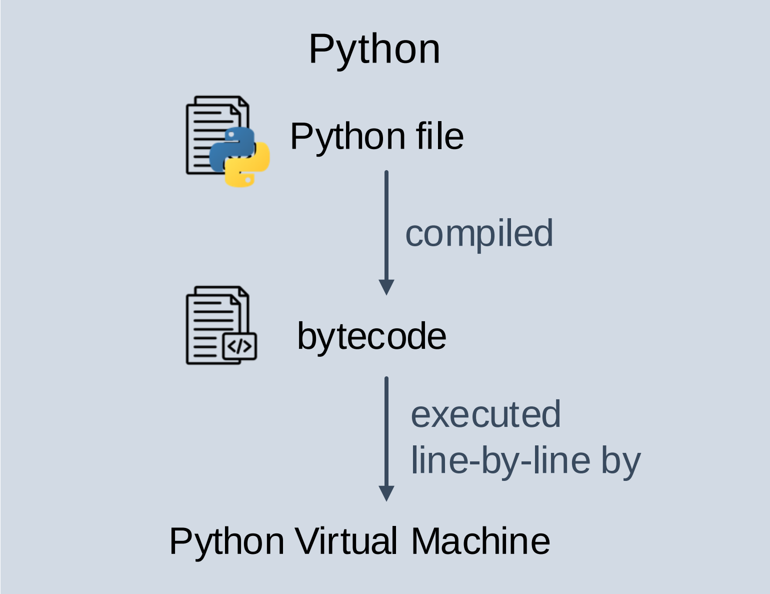
Leverage tensor product structure

Tensor product $\mathcal{H} = \mathcal{H}_1 \otimes \mathcal{H}_2 \otimes \cdots \mathcal{H}_K$ of size $N = n^K$

- Regular matmul: $H\rho$ ➡ $\mathcal{O}(N^3) = \mathcal{O}(n^{3K})$
- Tensprod matmul: $H = \sum_m^M H_m^{(1)} \otimes H_m^{(2)} \otimes \cdots H_m^{(m)}$ ➡ $\mathcal{O}(MKn^3n^{2K-2}) = \mathcal{O}(MKn^{2K+1})$

Under the hood: JAX and diffrax

Linear algebra on GPUs + automatic differentiation
→ same tools as machine learning
→ Dynamiqs built on JAX (Google) and Diffrax (Patrick Kidger)



Python

Python file

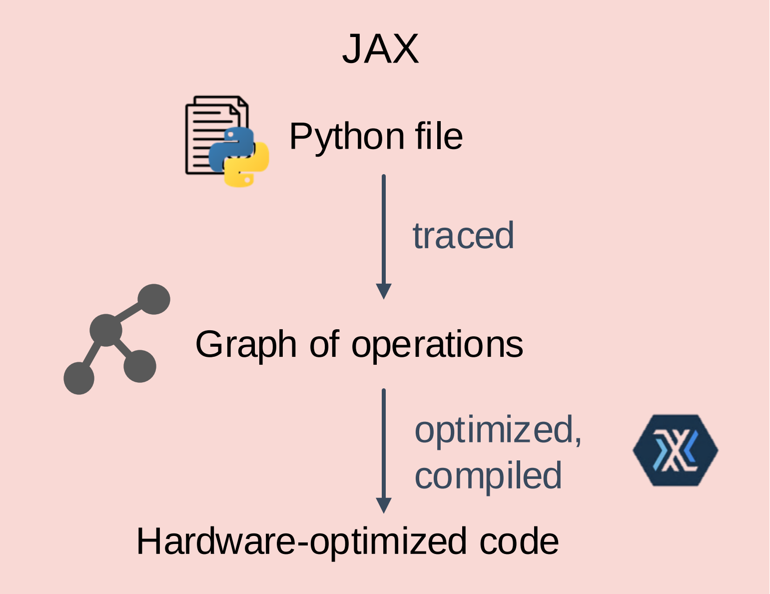
compiled

bytecode

executed line-by-line by

Python Virtual Machine

- Interpretation overhead
- Dynamic typing
- No low-level optim.



JAX

Python file

traced

Graph of operations

optimized, compiled

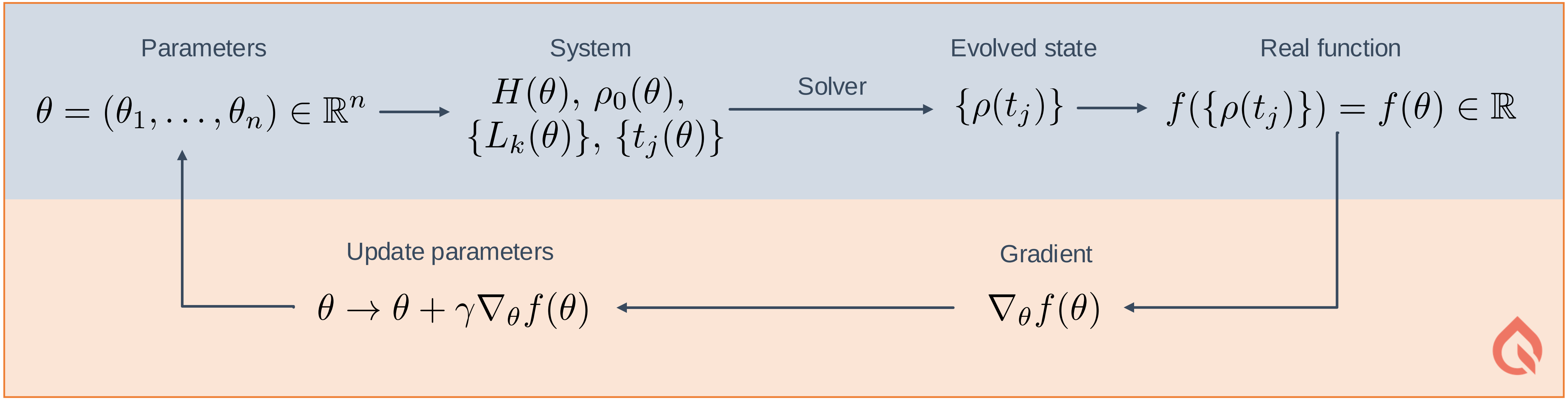
Hardware-optimized code

- Fused operations
- Memory access optimisations
- Loop unrolling

All ODE solving is handled by Diffrax, a specialized library built on JAX

Differentiability

Differentiable solvers



Parameters

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

System

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

Solver

Evolved state

$$\{\rho(t_j)\}$$

Real function

$$f(\{\rho(t_j)\}) = f(\theta) \in \mathbb{R}$$

Gradient

$$\nabla_\theta f(\theta)$$

Update parameters

$$\theta \to \theta + \gamma \nabla_\theta f(\theta)$$

<u>Project philosophy</u>: fast and reliable **building block**

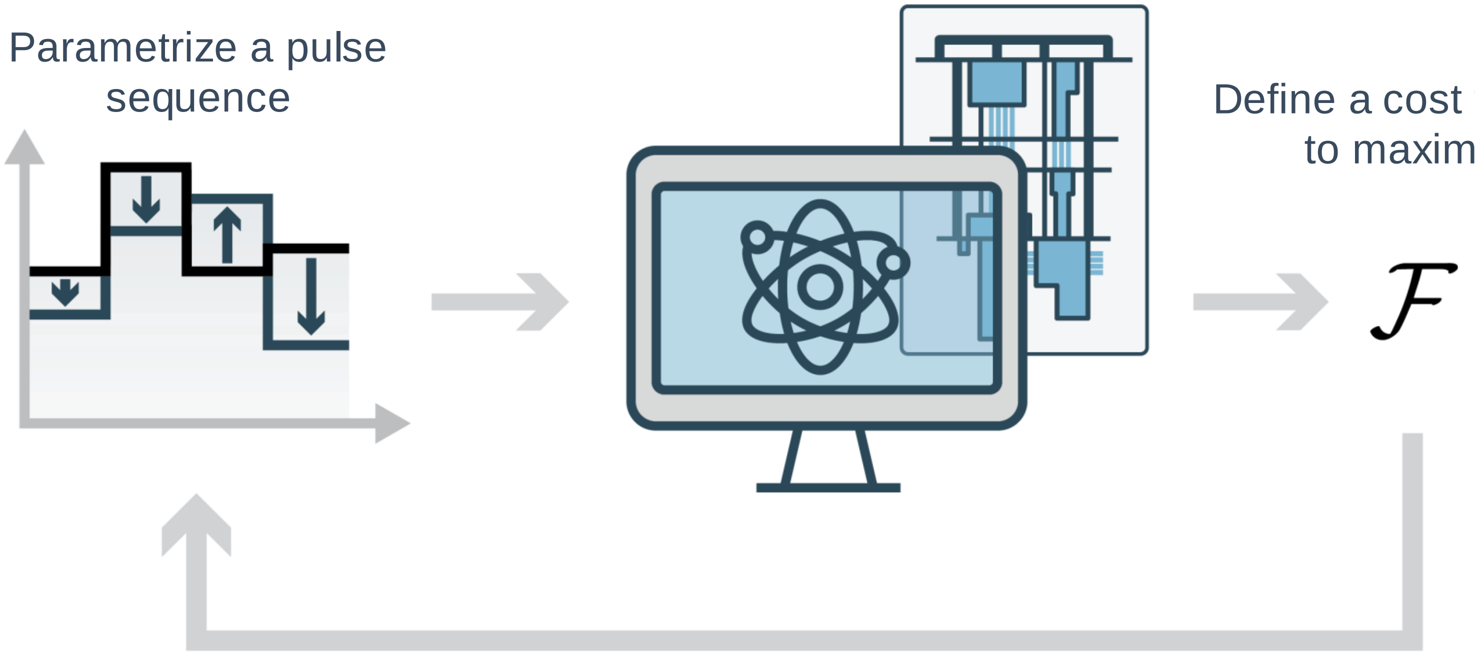
- Quantum optimal control
- Parameter estimation
- State tomography
- Sensitivity analysis
- …

<u>Computing gradients:</u>

- Automatic differentiation
  - → Fast and reliable, but large memory
- Adjoint state method*
  - → Low memory, but slower
- Recursive checkpointing
  - → Very strong tradeoff (recommended)

*see Gautier, Genois and Blais, arXiv:2403.14765

Quantum optimal control



Simulate with a model of
the experiment

Parametrize a pulse
sequence

Define a cost function
to maximize

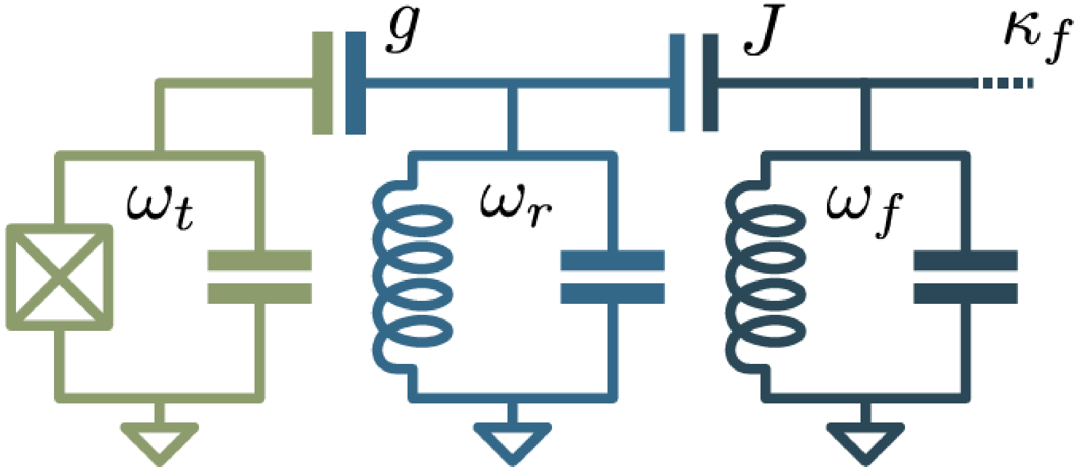
$\mathcal{F}$

Iterate with gradient
descent until
convergence

Example: transmon readout

Objective: optimize dispersive readout of a transmon
using minimal assumptions

$$H = 4E_C n_t^2 - E_J \cos(\phi_t) + \omega_r a^\dagger a + \omega_f f^\dagger f$$
$$+ ig n_t (a^\dagger - a) + J(f^\dagger a + a^\dagger f)$$



- Full cosine model, including Purcell filter
- MW drive on Purcell filter and/or transmon
- RWA on drives (simplifies numerical integration)

Difficult numerical problem

- ~400 parameters (1ns bins x 100ns x 2 drives)
- Hilbert space size ~ 2500 (5 x 5 x 100)
- GHz dynamics
- Open quantum system

- Full optimization in ~1 day
- Experimentally realistic pulses
- Interpretable results
- (Re-)discovered readout protocols, but optimized

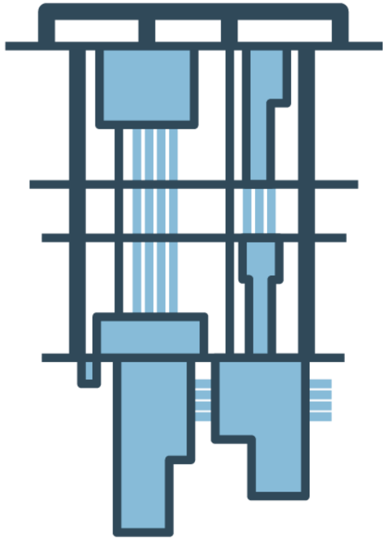
Presenting at WKS33 (Optimal control and calibration) – Friday 10am

Gautier, Genois and Blais, arXiv:2403.14765

Parameter estimation

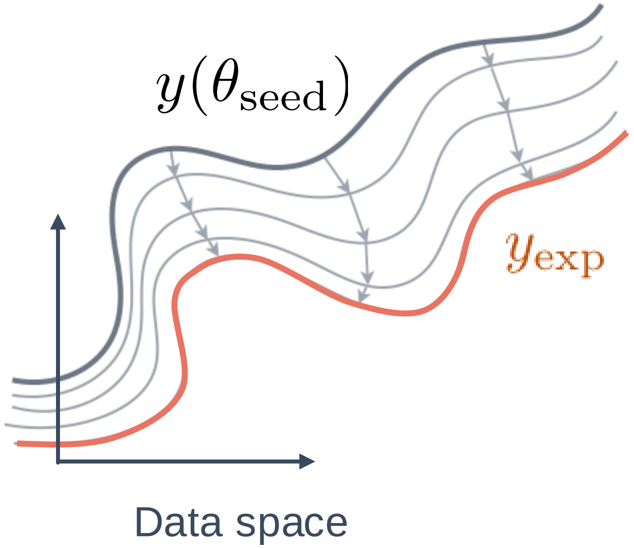
Quantum system

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



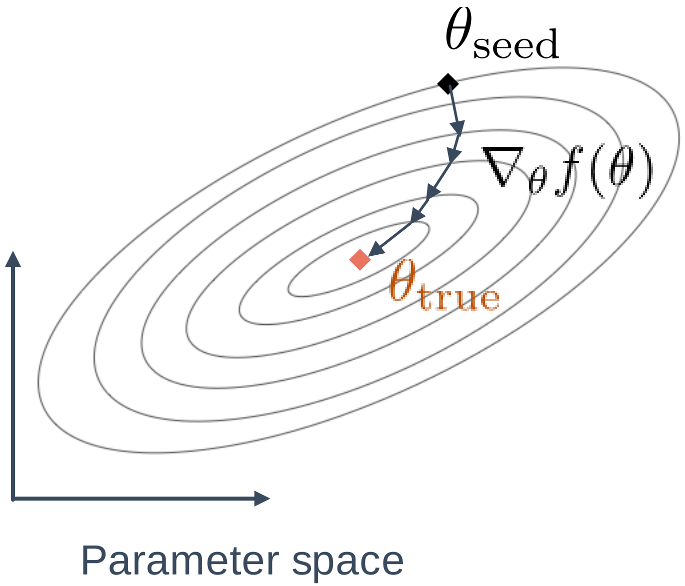
Measure data

Measure y_{exp}



Fit parameters

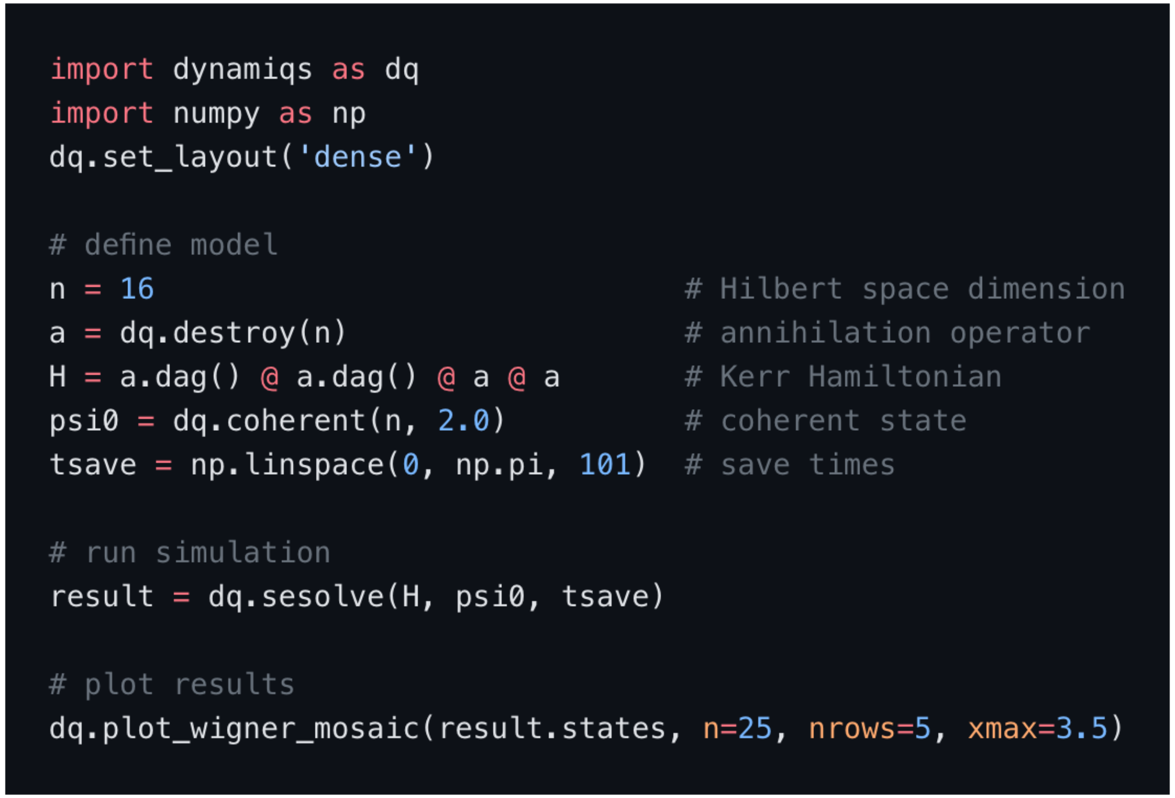
Minimise $f(\theta) = ||y(\theta) - y_{\mathrm{exp}}||^2$



Need gradient for efficient parameter search

Accessible API

A QuTiP-like API

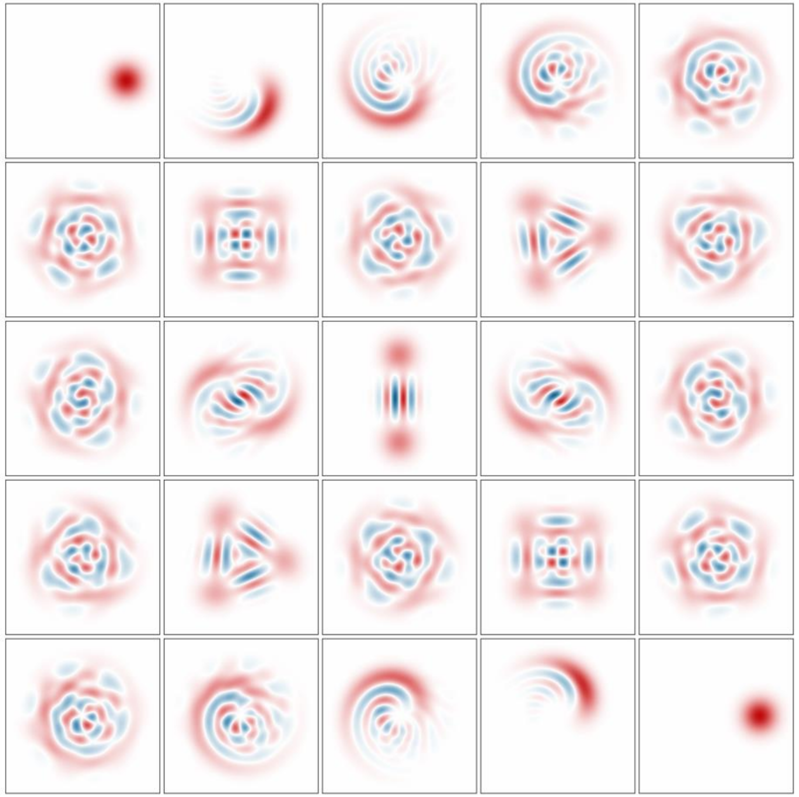


```
import dynamiqs as dq
import numpy as np
dq.set_layout('dense')

# define model
n = 16                                  # Hilbert space dimension
a = dq.destroy(n)                       # annihilation operator
H = a.dag() @ a.dag() @ a @ a           # Kerr Hamiltonian
psi0 = dq.coherent(n, 2.0)              # coherent 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, with **small** differences when appropriate (e.g. time-dependence)
- **Compatible with QuTiP** objects
- Smoothly runs on GPUs, computes gradients, or **set global settings** (matrix layout, precision)

With many more features…

Solvers

- ODE solvers from diffrax
  - Modern ODE solvers (Tsit5, Dopri8, Bosh3, …)
  - Implicit solvers
  - Adaptive-step SME solvers
  - Symplectic solvers
- Quantum-tailored solvers (Rouchon)
- Easily implement custom solvers
- Custom and optimized sparse format
  - JAX implementation
  - Low-level sparse CUDA kernels
- Krylov subspace methods for propagators

Gradients

- Compute gradients w.r.t. evolution time
- Compute high-order derivatives (Hessian)
- Freedom over gradient descent method (only examples provided)

Utilities

- Support for multiple Hamiltonian formats (constant, PWC, modulated, callable)
- Time-dependent jump operators
- User-defined save functions (partial trace, purity, …)
- Plotting functions (automated GIFs)
- All functions work on batched arrays
- Parralelisation across CPUs/GPUs
- Pulse composition API

Library

- Modern software development practices
  - Unit tests
  - Solver tests against analytical solutions
  - Accessible & complete documentation
  - Open-source
  - Continuous integration
  - Automatic benchmarking

No desire to **expand the scope** of the library…
Users can **build on top** of Dynamiqs depending on their specific needs!

01. Simulation of quantum systems with a focus on performance

02. End-to-end differentiable, using automatic differentiation

03. Focus on the essential: easy-to-use yet powerful and with just enough features

DYNAMIQS GITHUB