QuTiP:

Applications from quantum technology and quantum biology

Neill Lambert

Senior Research Scientist

RIKEN

Resources: https://arxiv.org/abs/2412.04705, Lambert at al., v5 review.
www.qutip.org
https://qutip2024.wordpress.com/ v5 release developer's conference

QuTiP development recently supported by:

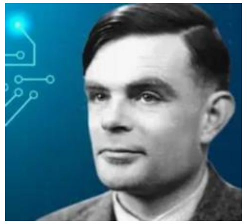
Here begins our new MIRAI
MOONSHOT
RESEARCH & DEVELOPMENT PROGRAM

RIKEN

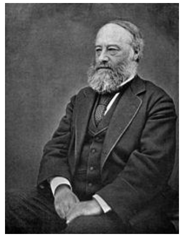
INSTITUT
QUANTIQUE
UNIVERSITÉ DE SHERBROOKE

Google Summer of Code

- Studied at the University of Manchester (Tobias Brandes)



Turing



Joule

- 2005: JSPS Fellow @ The University of Tokyo (Prof. Akira Shimizu)

- 2008-...: @ RIKEN

Part 1: History of QuTiP, v5 overview, and basic examples

Part 2: An example from quantum biology: photosynthesis, and a more nuanced explanation of noise with non-Markovian methods (HEOM)

Part 3: non-Markovian methods continued with input-output HEOM and pseudomodes + some additional QuTiP features with ENR states and more…

Part 1: History, v5 overview, basic examples

Overview:

- **History and background of QuTiP**

- QuTiP main functionality: noise simulation and open system dynamics

- QuTiP v5: what has changed?

- QuTiP-QIP: pulse-level simulator of quantum circuits

- Role of QuTiP and QuTiP-QIP in the future?
  - More developed circuit simulator?
  - Cloud-computer backends (IonQ, IBM, etc)?

Quantum software

Based on qosf.org/project_list/

2002	qoToolbox	Matlab	University of Auckland
2004	CHP	C	Berkeley, USA
2006	Qubiter	C++	Artiste-qb, Canada
2007	QCF	Matlab	Oxford University
2009	PyQu	**Python**	Google
2010	QuBit	C++	Steven Goodwin
2011	QuTiP	**Python**	Riken, Japan
2013	Q++	C++	Cybernet Systems Corp
2013	SQCT	C++	University of Waterloo
2014	QuanSuite	Java	Artiste-qb
2014	QC PG	qScript	Google
2014	Quipper	Haskell	Dalhousie University
2015	Quantum++	C++	University of Waterloo
2016	QETLAB	Matlab	University of Waterloo
2016	Liqui|>	F#	Microsoft
2016	Quant. Fog	**Python**	Artiste-qb
2016	Qubiter	**Python**	Artiste-qb
2017	ProjectQ	**Python**	ETH Zurich
2017	Forest (QUIL)	**Python**	Rigetti
2017	QISKit	**Python**	IBM
2017	Quantum Optics.jl	Julia	Universität Innsbruck
2017	PsiQuaSP	C++.	Gegg M, Richter M

2018	Strawberry Fields	**Python**	Xanadu, Canada
2018	PennyLane	**Python**	Xanadu, Canada
2018	Quantum Dev Kit	Q#.	Microsoft
2018	QCGPU	Rust, OpenCl	Adam Kelly
2018	NetKet	C++	The Simons Foundation
2018	OpenFermion	**Python**	Google, Harvard, ETH ..
2018	CirQ	**Python**	Google
2018	Qulacs	**Python**	QunaSys, Osaka, NTT, Fujitsu
2019	Yao.jl	Julia	Luo and Liu
2020	TensorFlow Q	**Python**	Google
2021	Pulser	**Python**	Pasqal
2021	MitiQ	**Python**	Unitary Fund

+............. Many more!

QuTiP

Original Developers

Paul Nation
IBM Q
Library designer and main contributor

Robert Johansson
Tokyo, Japan
Library designer and main contributor

QuTiP history

2012**: QuTiP v1** release: Functionality comparable to Matlab's quantum optics toolbox

2013: **QuTiP v2** release: Time-dependent Hamiltonian support, Bloch-Redfield solver, Floquet-Markov solver

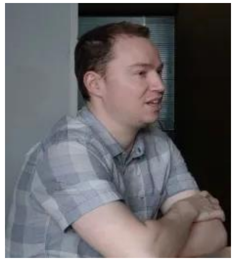
2014**: QuTiP v3** release: Stochastic master equation, steady-state solvers, first release of qutip.qip module (circuit simulator)

2016: **QuTiP v4** release: HEOM solver, quantum optimal control (Alex Pitchford and myself begin to contribute)

….. minor releases including PIQs solver (Nathan and Shahnawaz), updates to HEOM and optimal control solvers.
Robert Johansson and Paul Nation move on to new careers (Rakuten and IBM-Q, respectively), period of crisis!

2018/2019: New development team formed during RIKEN workshop, new guidelines for administration and developer responsibilities

Alex Pitchford
(Aberystwyth)



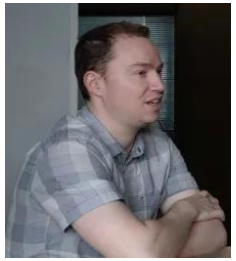
Eric Giguere
(Sherbrooke)

Shahnawaz Ahmed
(RIKEN, now Chalmers)

Nathan Shammah
(RIKEN, now Unitary
Fund)

2019: V4.4: QobjEvo introduced, and **first Google Summer of Code (GSOC) engagement and students.**
2020: v4.5: Major update to QuTiP-QIP (**result of GSOC student project of Boxi Li**)
2021: v4.6: OpenQASM support, QuTiP-QIP further development, better support for Windows (to solve Cython woes)
2022: v4.7: Major update to HEOM solver, Krylov subspace solver, etc..

Current administration and development team

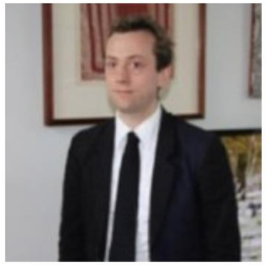


Eric Giguere
Sherbrooke

Shahnawaz Ahmed
Chalmers, former
RIKEN intern

Alex Pitchford
Aberystwyth

Nathan Shammah
Unitary Fund, former
RIKEN postdoc



Neill Lambert
RIKEN

Boxi Li, Julich,
Former GSOC

Jake Lishman
IBM, former GSOC
Note: quasi-retired

Simon Cross
Zurich Instr.,
former RIKEN
Tech. Staff.

Asier Galacia,
Julich, former GSOC

Paul Menczel,
RIKEN

Patrick Hopf
Munich, RIKEN intern

Additional contributors

- Denis Vasilyev (Leibniz)
- Kevin Fischer (Stanford)
- Anubhav Vardhan (New Dehli, India)
- Markus Baden (Zurich, Switzerland)
- Jonathan Zoller (Ulm University)
- Ben Criger (RWTH Aachen)
- Ben Bartlett (Stanford)
- Piotr Migdał (Warsaw, Poland)
- Arne Grismo (Amazon)
- Cassandra Granade (Microsoft)

+ 100s more

Advisory board:
Franco Nori, RIKEN
Anton Frisk Kockum, Chalmers
Robert Johansson, Rakuten
Daniel Burgarth, Erlangen
Will Zheng, Unitary Fund

QuTiP history

Developer’s workshop @ RIKEN, March 2024, v5 released!

Recent development team expanded through:
Google Summer of Code

Google
Summer of Code

GSOC has been hugely beneficial to QuTiP, and led to many new features, including QuTiP-QIP, and the data layer which forms the backbone of QuTiP v5.

2019-2024 we have supported **16** GSOC projects (2-3 per year).

Three of these students joined the admin team after continued contributions

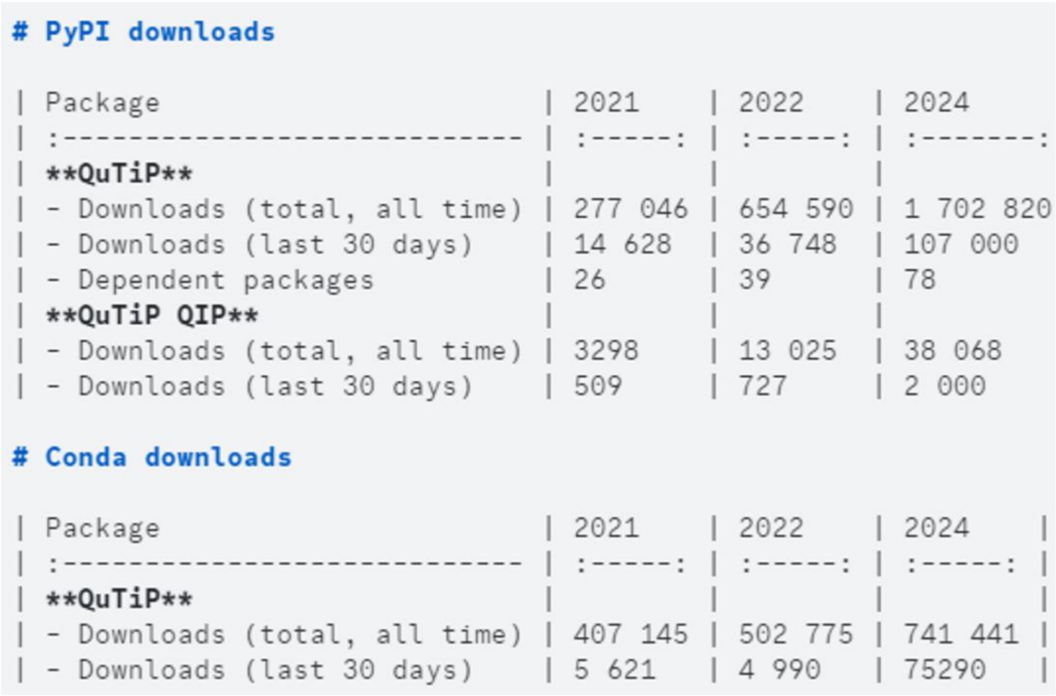
Boxi Li, Julich,
Project: QuTiP-QIP

Asier Galacia, Julich
Project: QuTiP-TensorFlow

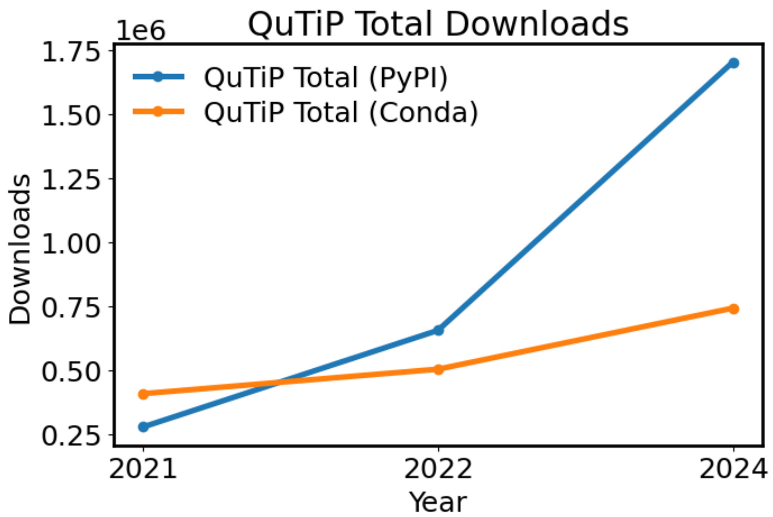
Jake Lishman, Imperial
Project: new data layer
Note: already quasi-retired (IBM-Q)

Statistics and growth:

https://www.wheelodex.org/projects/qutip/rdepends/



```
# PyPI downloads

| Package                       | 2021    | 2022    | 2024      
| :---------------------------- | :-----: | :-----: | :-------:
| **QuTiP**                     |         |         |
| - Downloads (total, all time) | 277 046 | 654 590 | 1 702 820
| - Downloads (last 30 days)    | 14 628  | 36 748  | 107 000
| - Dependent packages          | 26      | 39      | 78
| **QuTiP QIP**                 |         |         |
| - Downloads (total, all time) | 3298    | 13 025  | 38 068
| - Downloads (last 30 days)    | 509     | 727     | 2 000

# Conda downloads

| Package                       | 2021    | 2022    | 2024    |
| :---------------------------- | :-----: | :-----: | :-----: |
| **QuTiP**                     |         |         |         |
| - Downloads (total, all time) | 407 145 | 502 775 | 741 441 |
| - Downloads (last 30 days)    | 5 621   | 4 990   | 75290   |
```



Some interesting dependent package highlights:

- AtomCalc (simulates atomic energy level shifts in laser fields)
- **bosonic-qiskit (National Quantum Initiative Co-design Center for Quantum Advantage bosonic Qiskit simulator)**
- chalmers-qubit (A simulator of the Chalmers device to be used with qutip-qip)
- **dynamiqs (High-performance quantum systems simulation with JAX (GPU-accelerated & differentiable solvers)**
- filter_functions (A package for efficient numerical calculation of generalized filter functions to describe the effect of noise on quantum gate operations)
- kqcircuits (KQCircuits is a KLayout/Python-based superconducting quantum circuit library developed by IQM)
- netket (Machine Learning toolbox for many-body quantum systems)
- qiskit-metal (for quantum device design & analysis)
- **scqubits (superconducting qubits in Python)**
- **SQcircuit (superconducting quantum circuit analyzer)**

Installing QuTiP:

There are simple cloud options, but these are limited in compute power:

- Google collab: e.g., companion notebook for this talk

Local browser-based options:

- https://qutip.org/try-qutip/
  - Runs QuTiP in your browser

- QuTiP Virtual Lab (https://qutip.org/qutip-virtual-lab.html).

Local install with:

➢ pip install qutip

or for anaconda

➢ conda install qutip

Note: windows users, solve all your problems by using Windows Subsystems for Linux (WSL)

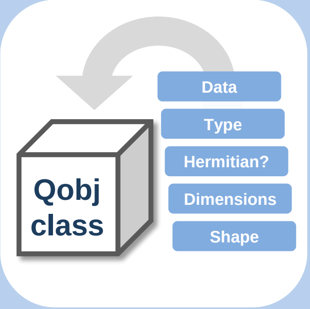
Overview:

- History and background of QuTiP
- **QuTiP main functionality: noise simulation and open system dynamics**
- QuTiP v5: what has changed?
- QuTiP-QIP: pulse-level simulator of quantum circuits
- Role of QuTiP and QuTiP-QIP in the future?
    - More developed circuit simulator?
    - Cloud-computer backends (IonQ, IBM, etc)?

Overview of QuTiP Functionality

1. QuTiP-Core

Qobj

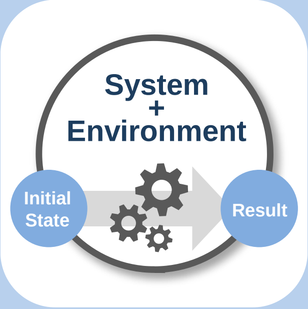


Data Layer
Dense, CSR and DIA

Functions
Eigenstates, Matrix elements, Norms, etc.

Utility Functions
Entanglement measures, Distances, Superoperator representations (Choi, Kraus, etc.), Channels, ENR states, Two-time correlation functions and spectra, MPI support, etc.

Solvers



mesolve
Lindblad master equation

mcsolve + nm_mcsolve
Monte-Carlo master equation

brmesolve
Bloch-Redfield master equation

fmmesolve
Floquet master equation

krylovsolve
Krylov subspace solver

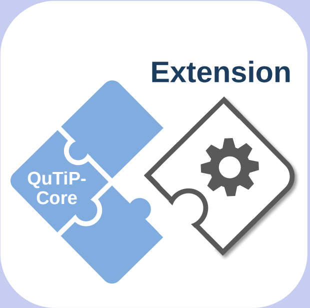
smesolve
Stochastic master equation

HEOMsolver
Hierarchical equations of motion

PIQS
Permutational invariant systems

2. QuTiP-Packages

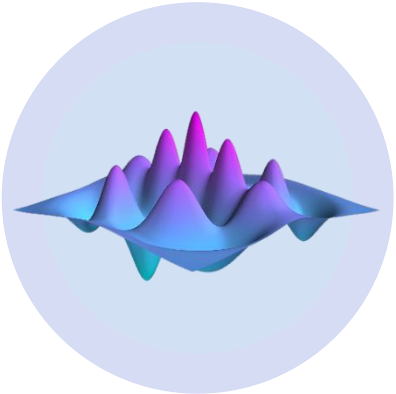
Packages



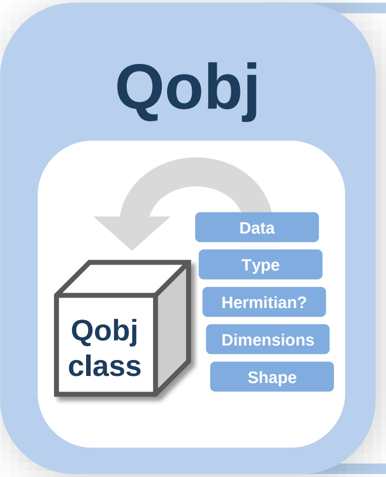
QIP: Pulse-based Quantum Circuit Simulator
allows circuits to be run on different hardware backend simulations at the level of time-dependent pulses and noise.

QOC: Quantum Optimal Control Package
supports for CRAB, GRAPE and GOAT algorithms.

JAX: JAX Data Layer
supports the popular JAX package, allowing for GPU and autograd.



1. QuTiP-Core

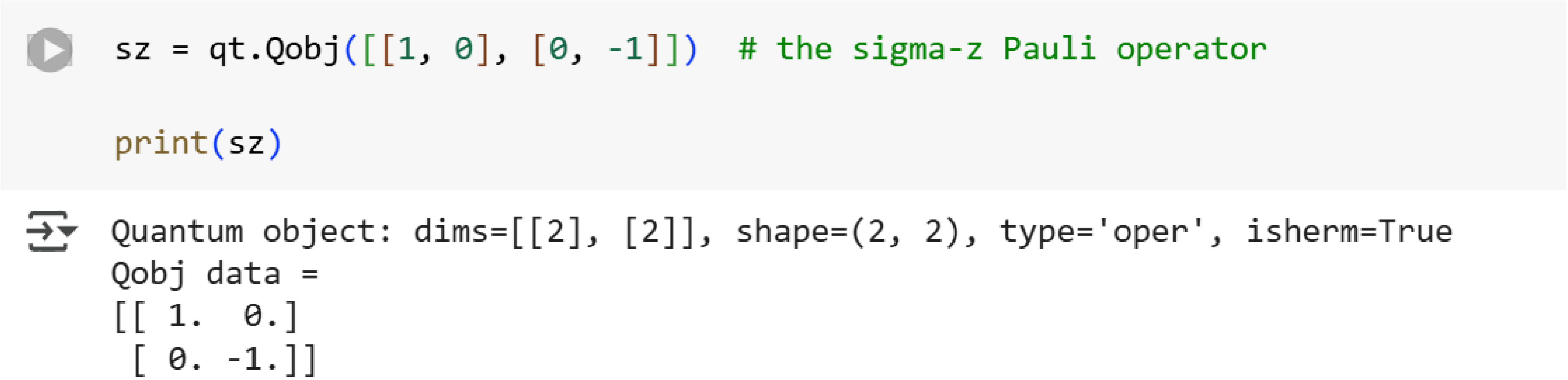


Data Layer
Dense, CSR and DIA

Functions
Eigenstates, Matrix elements, Norms, etc.

Utility Functions
Entanglement measures, Distances, Superoperator representations (Choi, Kraus, etc.), Channels, ENR states, Two-time correlation functions and spectra, MPI support, etc.

States and operators can be defined from arrays, or use predefined common structures.

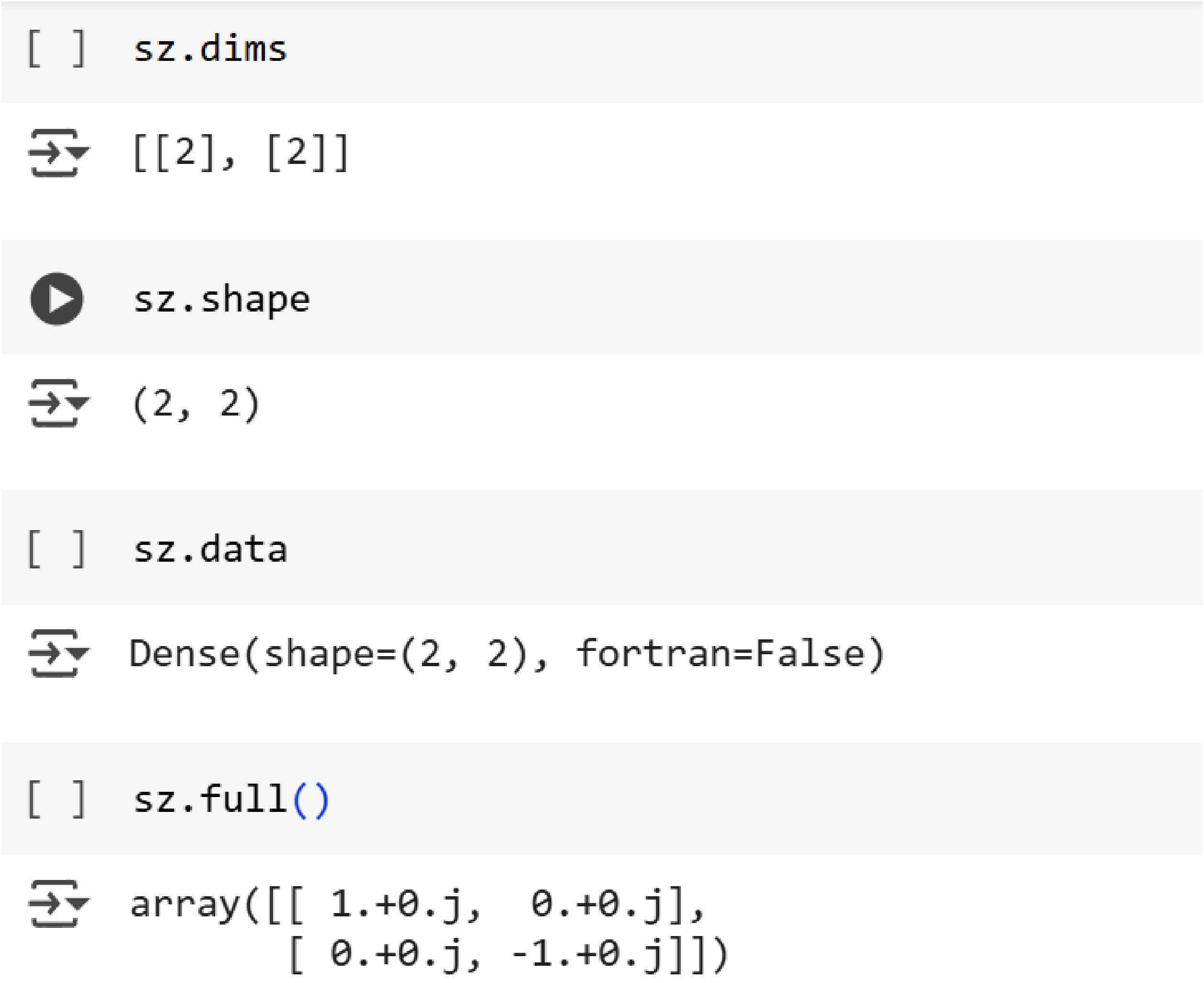


```
sz = qt.Qobj([[1, 0], [0, -1]])  # the sigma-z Pauli operator

print(sz)
```

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

1. QuTiP-Core, quantum object properties

All objects have a set of properties, most of which the user doesn't need to see



```
[ ] sz.dims
```

```
[[2], [2]]
```

```
sz.shape
```

```
(2, 2)
```

```
[ ] sz.data
```

```
Dense(shape=(2, 2), fortran=False)
```

```
[ ] sz.full()
```

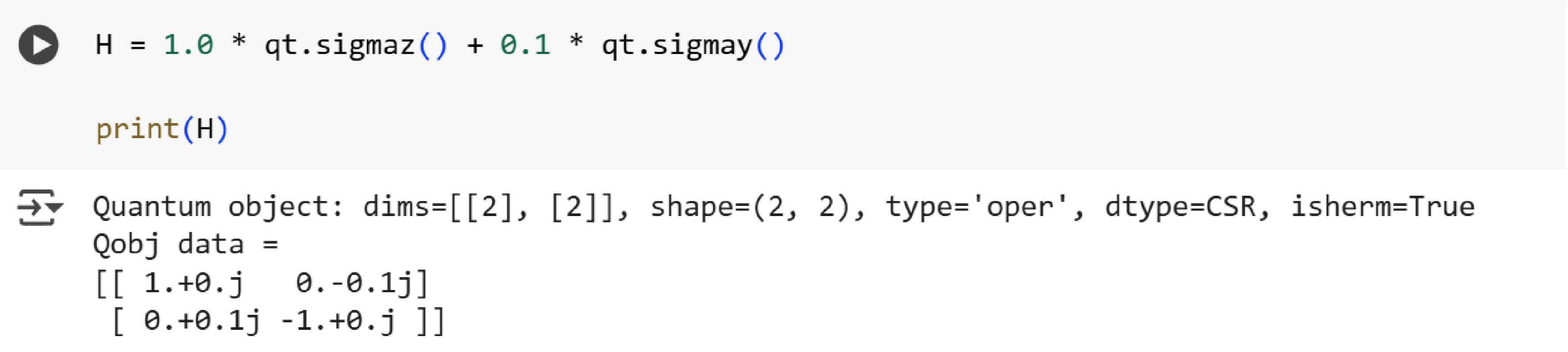
```
array([[ 1.+0.j,  0.+0.j],
       [ 0.+0.j, -1.+0.j]])
```

Dims, or dimensions, are mostly usefully thought of as maps:

- States (bra and kets), which are vectors, map vectors to scalars
- Operators, which are matrices, map vectors to vectors
- Super-operators, which are also matrices, map vectorized operators to vectorized operators (see later)
- Compound (tensor) objects operate in the same way, but dims properties store the compound structure

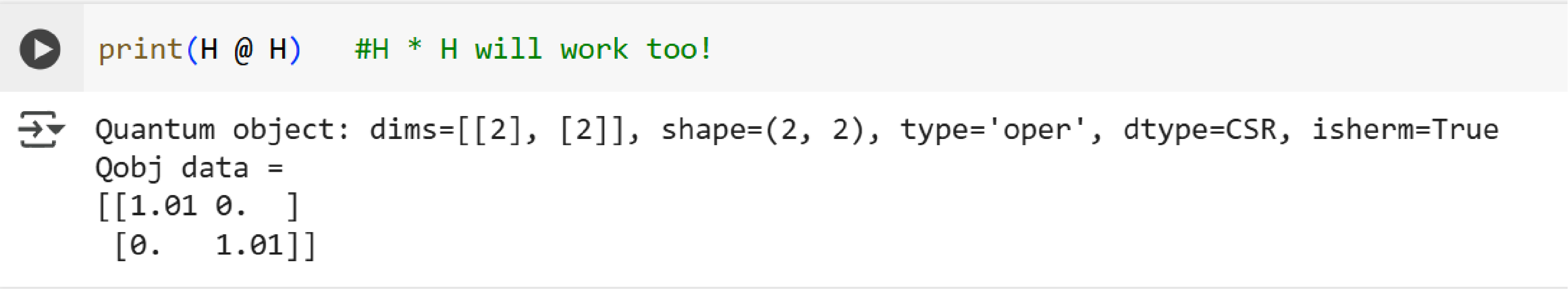
1. QuTiP-Core, object arithmetic

We can perform basic **arithmetic** with quantum objects; the rules are similar to standard matrix arithmetic



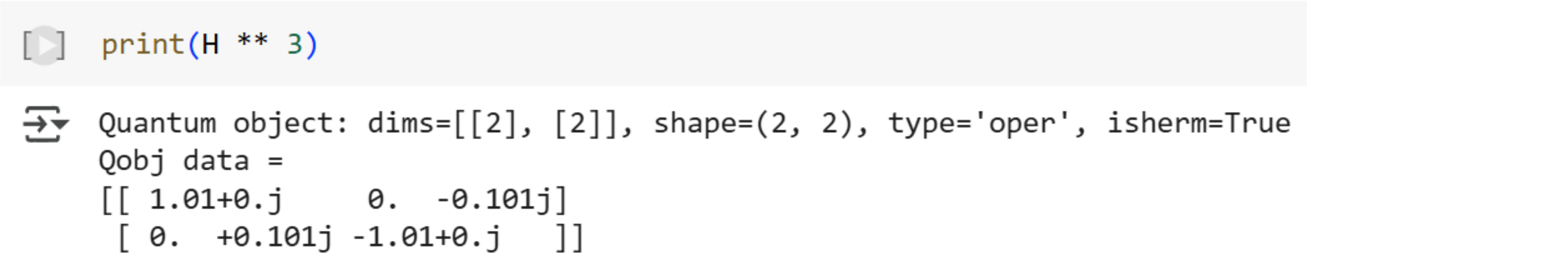
```
H = 1.0 * qt.sigmaz() + 0.1 * qt.sigmay()

print(H)
```

```
Quantum object: dims=[[2], [2]], shape=(2, 2), type='oper', dtype=CSR, isherm=True
Qobj data =
[[ 1.+0.j   0.-0.1j]
 [ 0.+0.1j -1.+0.j ]]
```



```
print(H @ H)   #H * H will work too!
```

```
Quantum object: dims=[[2], [2]], shape=(2, 2), type='oper', dtype=CSR, isherm=True
Qobj data =
[[1.01 0.  ]
 [0.   1.01]]
```

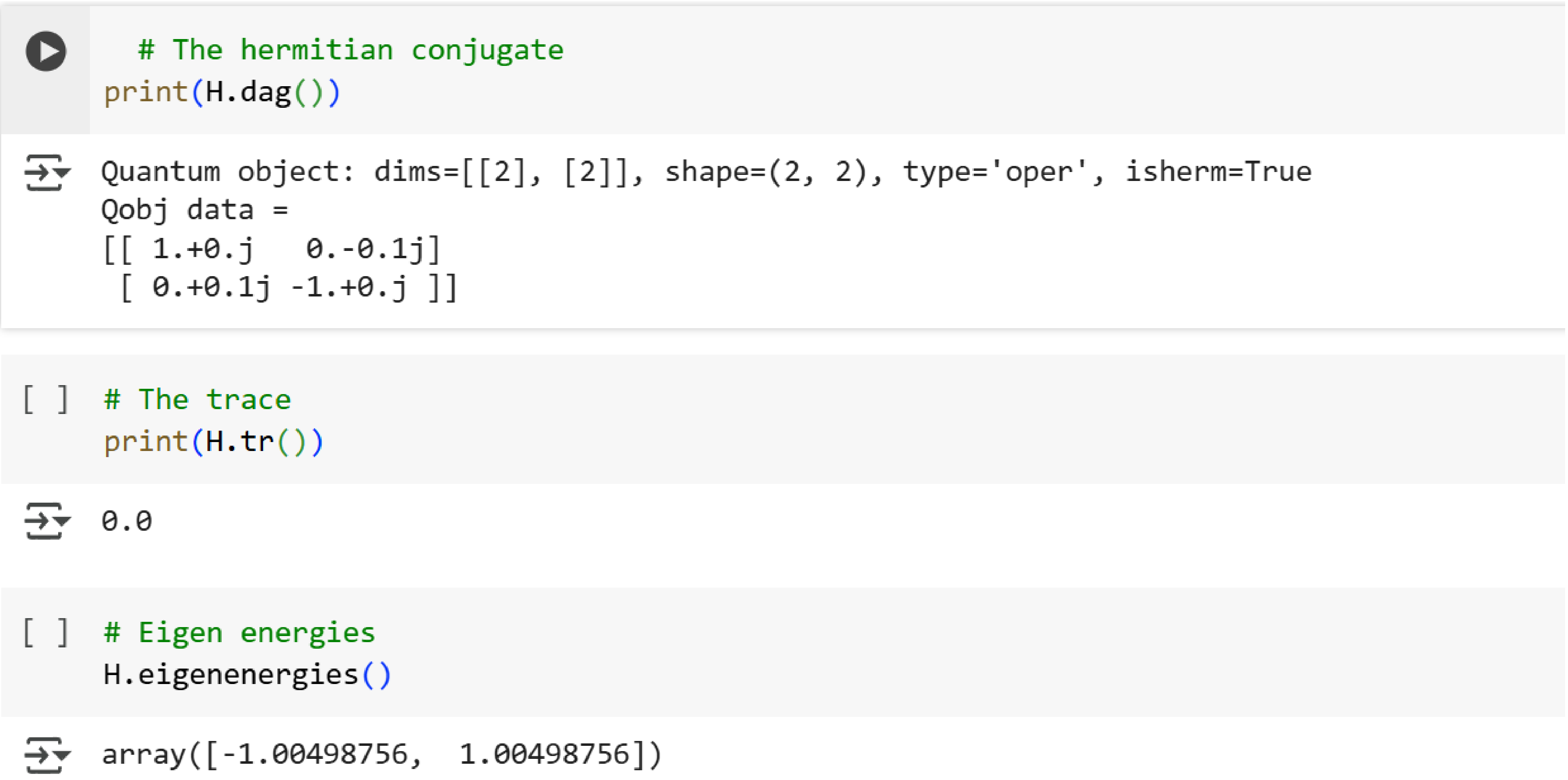


```
print(H ** 3)
```

```
Quantum object: dims=[[2], [2]], shape=(2, 2), type='oper', isherm=True
Qobj data =
[[ 1.01+0.j     0.  -0.101j]
 [ 0.  +0.101j -1.01+0.j   ]]
```

1. QuTiP-Core, object methods (functions)

There are many inbuilt functions, or methods, in quantum objects to perform common tasks:



```
  # The hermitian conjugate
print(H.dag())
```

```
Quantum object: dims=[[2], [2]], shape=(2, 2), type='oper', isherm=True
Qobj data =
[[ 1.+0.j   0.-0.1j]
 [ 0.+0.1j -1.+0.j ]]
```

```
[ ]  # The trace
     print(H.tr())
```

```
0.0
```

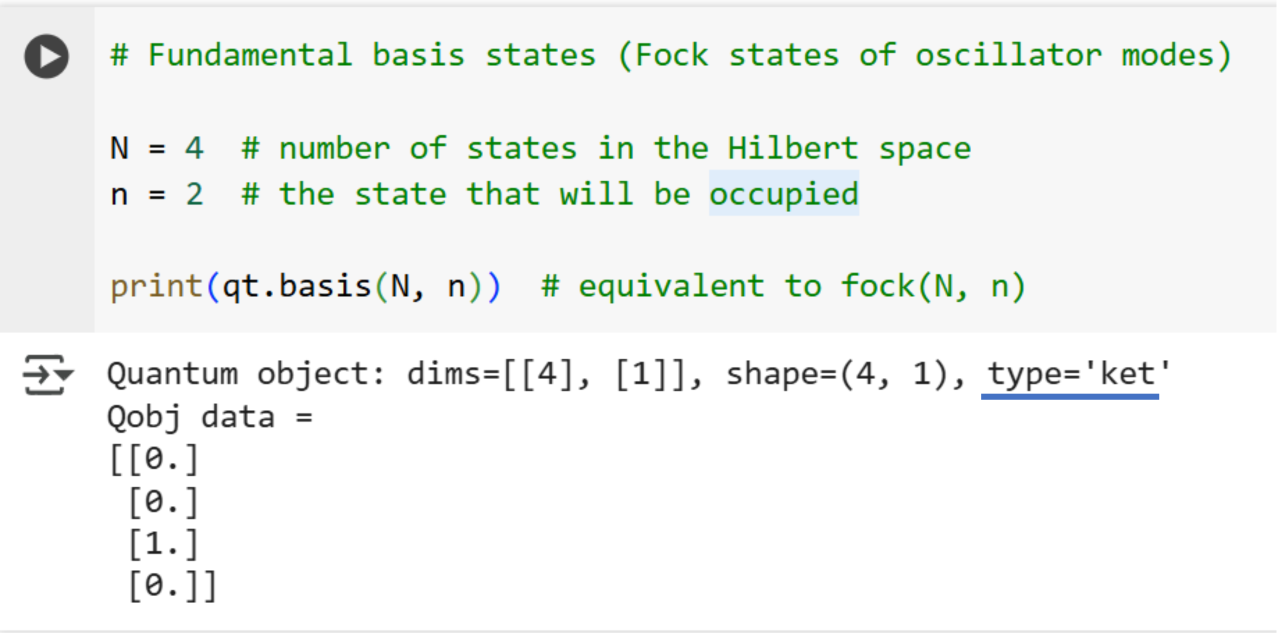
```
[ ]  # Eigen energies
     H.eigenenergies()
```

```
array([-1.00498756,  1.00498756])
```

1. QuTiP-Core, state vectors

When using QuTiP you normally need to make states and operators, and use them in some way. There are several convenient ways to build up the objects you might need:

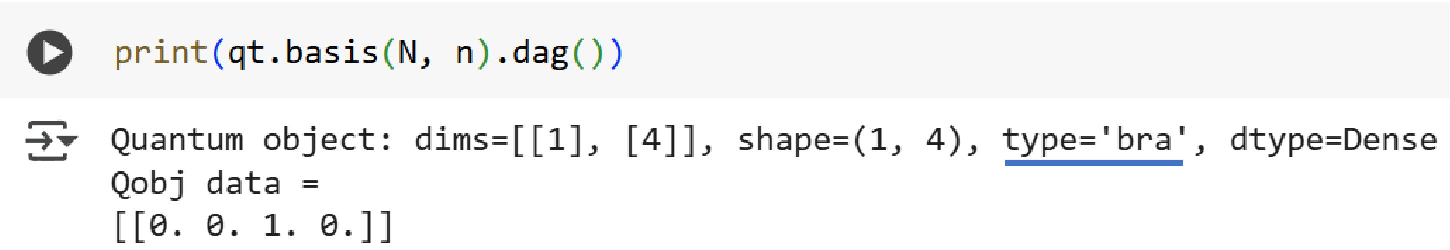
State vectors:



```
# Fundamental basis states (Fock states of oscillator modes)

N = 4  # number of states in the Hilbert space
n = 2  # the state that will be occupied

print(qt.basis(N, n))  # equivalent to fock(N, n)
```

```
Quantum object: dims=[[4], [1]], shape=(4, 1), type='ket'
Qobj data =
[[0.]
 [0.]
 [1.]
 [0.]]
```

It's common in discrete quantum systems to make a basis out of Fock states $|n\rangle$ which describes a excitation in the n-th level of the system

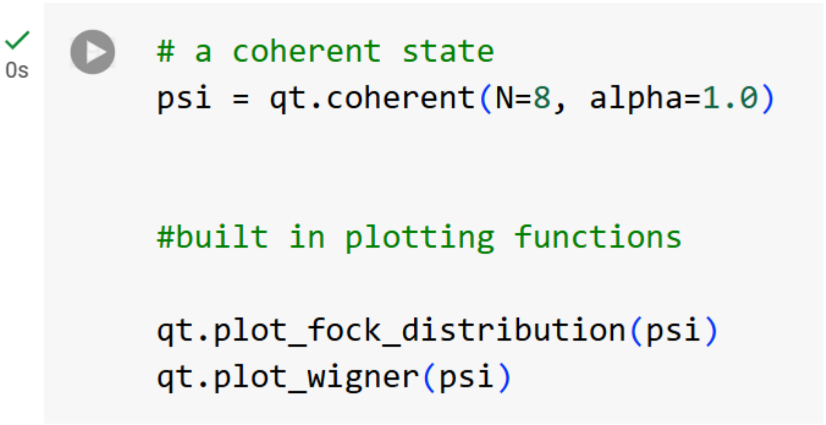


```
print(qt.basis(N, n).dag())
```

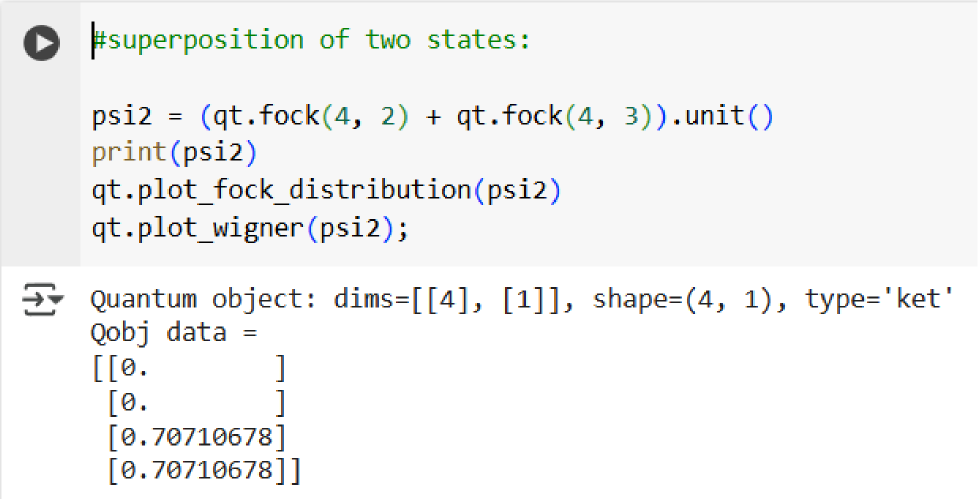
```
Quantum object: dims=[[1], [4]], shape=(1, 4), type='bra', dtype=Dense
Qobj data =
[[0. 0. 1. 0.]]
```

1. QuTiP-Core, Fock states and coherent states

When considering single-mode cavities Fock states are often thought of as labelling the number of photons in the mode, but coherent states and superpositions are also possible (among many more):

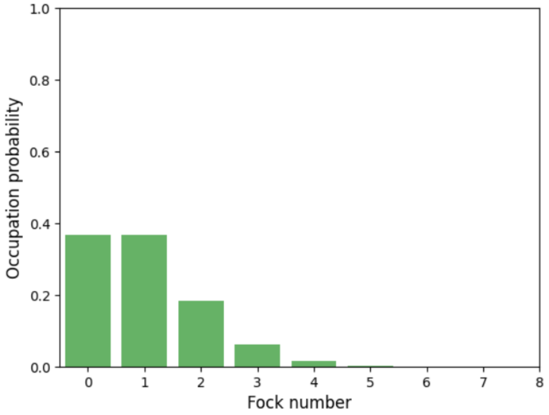


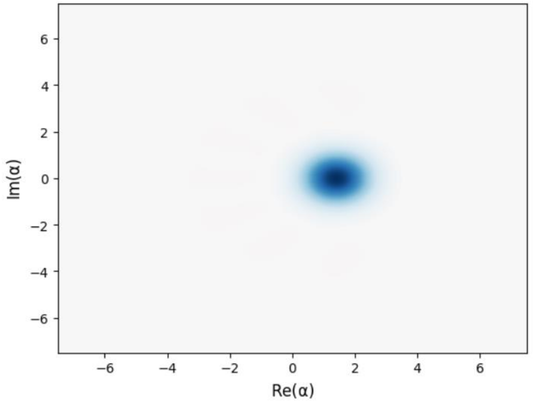
```
# a coherent state
psi = qt.coherent(N=8, alpha=1.0)


#built in plotting functions

qt.plot_fock_distribution(psi)
qt.plot_wigner(psi)
```

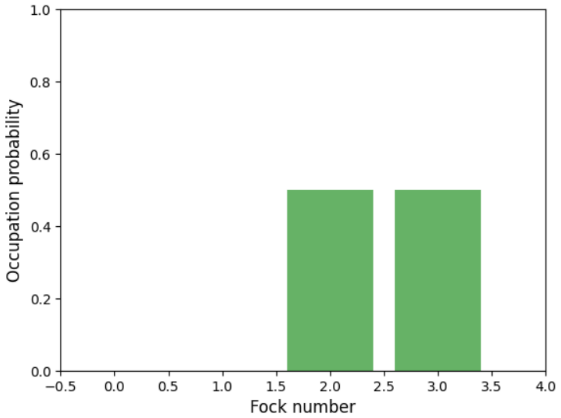


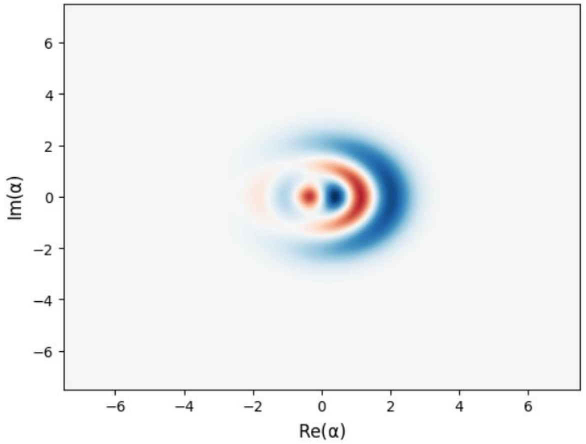
```
#superposition of two states:

psi2 = (qt.fock(4, 2) + qt.fock(4, 3)).unit()
print(psi2)
qt.plot_fock_distribution(psi2)
qt.plot_wigner(psi2);
```

```
Quantum object: dims=[[4], [1]], shape=(4, 1), type='ket'
Qobj data =
[[0.        ]
 [0.        ]
 [0.70710678]
 [0.70710678]]
```





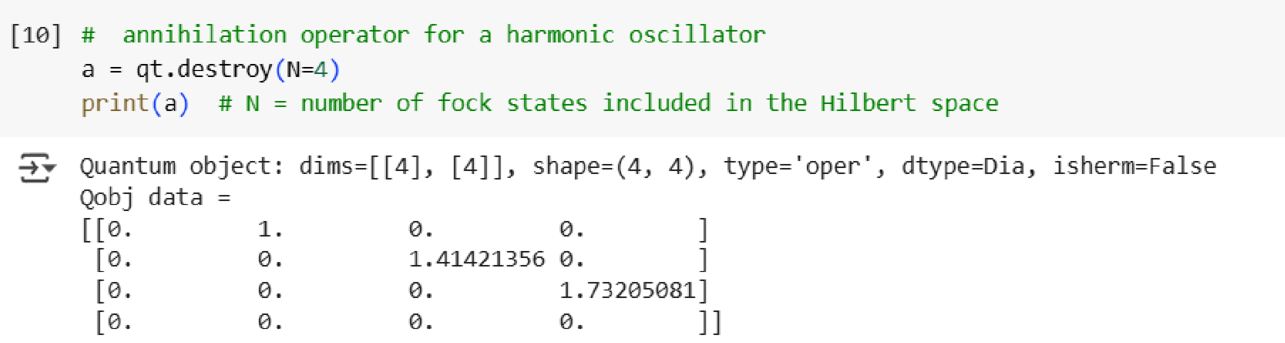




1. QuTiP-Core, operators

Annihilation and creation operators are also commonly used to create operators which connect Fock states:

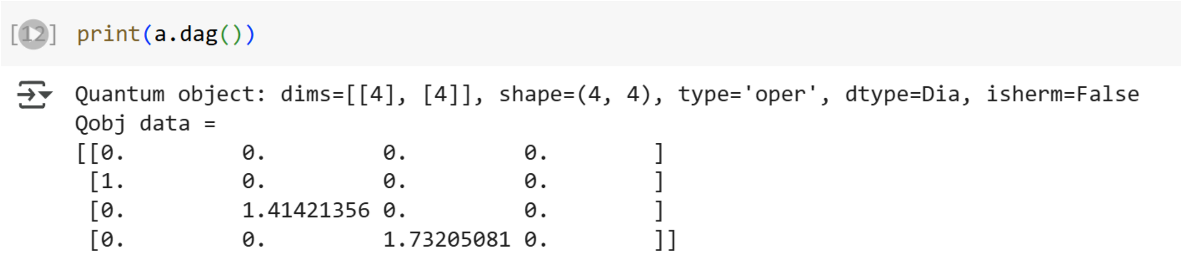
$$a|n\rangle = \sqrt{n}|n-1\rangle$$



```
[10] #  annihilation operator for a harmonic oscillator
     a = qt.destroy(N=4)
     print(a)  # N = number of fock states included in the Hilbert space
```

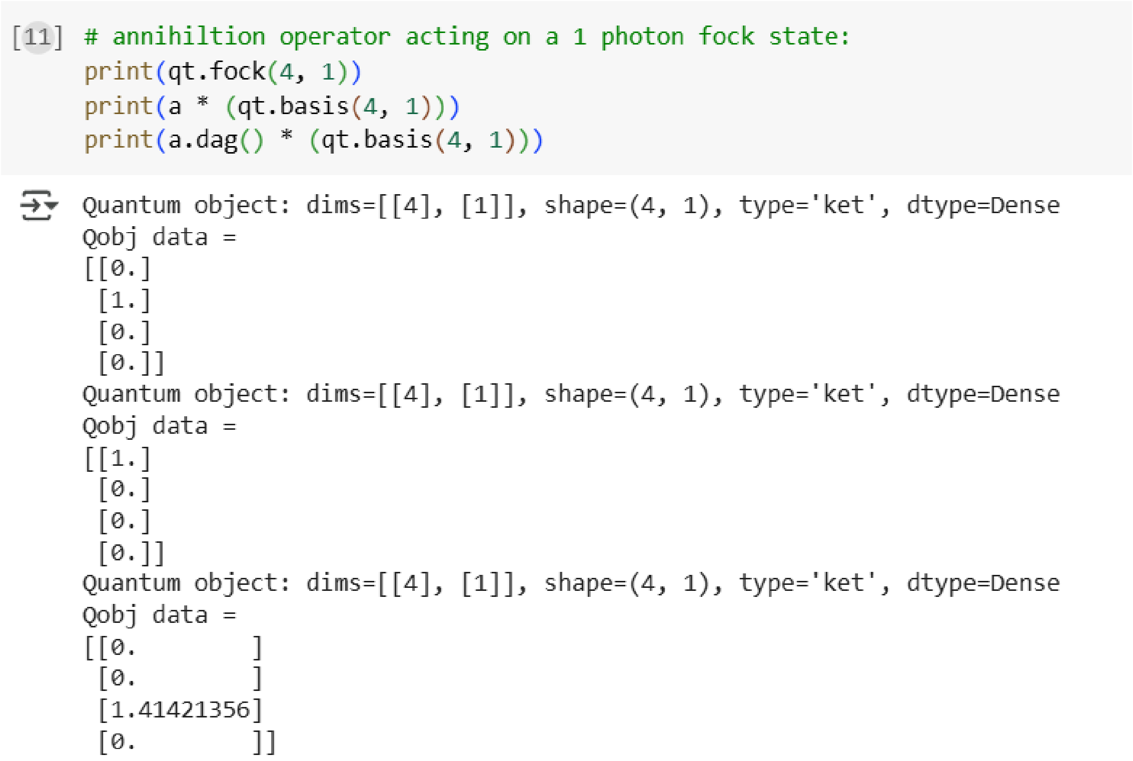
```
Quantum object: dims=[[4], [4]], shape=(4, 4), type='oper', dtype=Dia, isherm=False
Qobj data =
[[0.         1.         0.         0.        ]
 [0.         0.         1.41421356 0.        ]
 [0.         0.         0.         1.73205081]
 [0.         0.         0.         0.        ]]
```

$$a^{\dagger}|n\rangle = \sqrt{n+1}|n+1\rangle$$



```
[12] print(a.dag())
```

```
Quantum object: dims=[[4], [4]], shape=(4, 4), type='oper', dtype=Dia, isherm=False
Qobj data =
[[0.         0.         0.         0.        ]
 [1.         0.         0.         0.        ]
 [0.         1.41421356 0.         0.        ]
 [0.         0.         1.73205081 0.        ]]
```



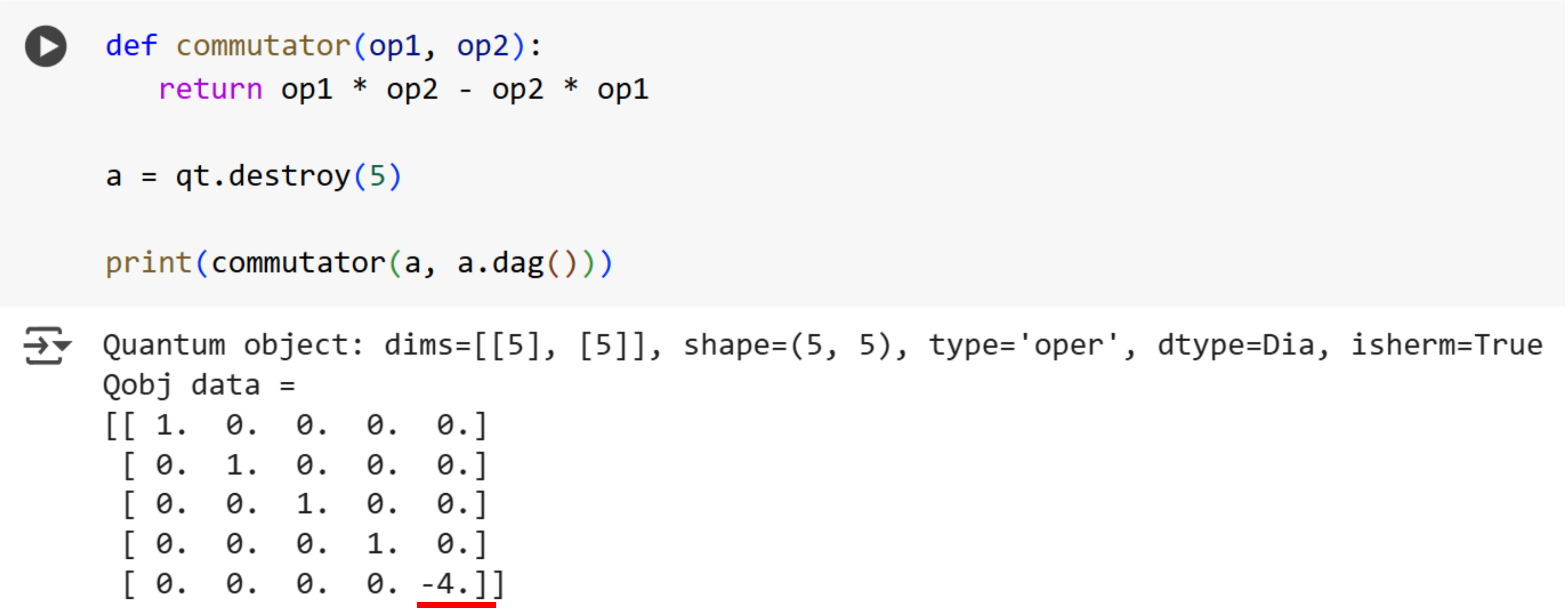
```
[11] # annihiltion operator acting on a 1 photon fock state:
     print(qt.fock(4, 1))
     print(a * (qt.basis(4, 1)))
     print(a.dag() * (qt.basis(4, 1)))
```

```
Quantum object: dims=[[4], [1]], shape=(4, 1), type='ket', dtype=Dense
Qobj data =
[[0.]
 [1.]
 [0.]
 [0.]]
Quantum object: dims=[[4], [1]], shape=(4, 1), type='ket', dtype=Dense
Qobj data =
[[1.]
 [0.]
 [0.]
 [0.]]
Quantum object: dims=[[4], [1]], shape=(4, 1), type='ket', dtype=Dense
Qobj data =
[[0.        ]
 [0.        ]
 [1.41421356]
 [0.        ]]
```

1. QuTiP-Core, Fock state truncation

Note: The creation and annihilation operators for Bosons should obey the commutation relation:

$$[a, a^\dagger] = 1$$



```python
def commutator(op1, op2):
    return op1 * op2 - op2 * op1

a = qt.destroy(5)

print(commutator(a, a.dag()))
```

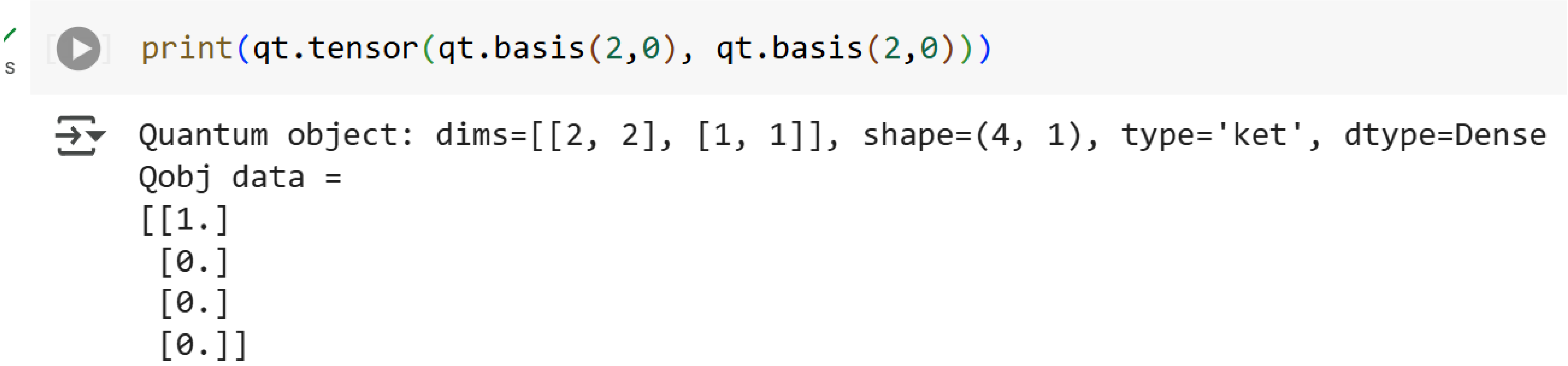
```
Quantum object: dims=[[5], [5]], shape=(5, 5), type='oper', dtype=Dia, isherm=True
Qobj data =
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0. -4.]]
```

Numerical **truncation of infinite** dimensional Hilbert space **will always include this error**; usually it is sufficient to choose a cut-off much larger than the states of interest, and things will be ok!

1. QuTiP-Core, tensor of composite systems

Composite systems (states): When we try to model two systems, like multiple spins, or a spin interacting with a cavity, we must take the tensor product of their spaces.

For example, the tensor product of two two-level systems gives us four possible states: 00, 01, 10, 11.



```
print(qt.tensor(qt.basis(2,0), qt.basis(2,0)))
```

```
Quantum object: dims=[[2, 2], [1, 1]], shape=(4, 1), type='ket', dtype=Dense
Qobj data =
[[1.]
 [0.]
 [0.]
 [0.]]
```

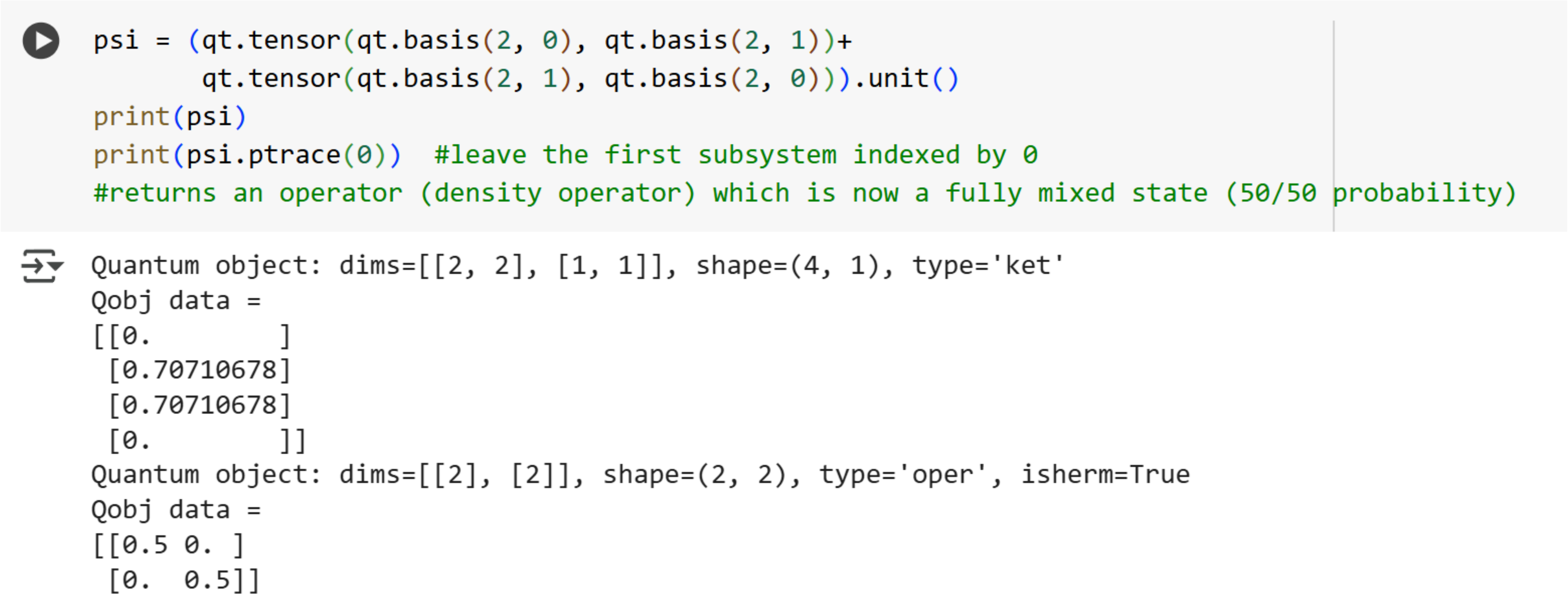
Note the dimensions are composite: the dims list [[2,2],[1,1]], tells us this is the tensor of two vectors.

A single 4-level system would have dims [[4], [1]].

1. QuTiP-Core, partial trace

Composite systems:
Sometimes we wish to remove information, or trace out, about one subsystem.
This is commonly used to make statistical mixtures (density operators) out of pure states on a larger space.

The act of tracing out introduces statistical uncertainty in the state of the subsystem.
QuTiP supports this with the 'ptrace' class method.
ptrace takes as an argument a list of integers determining **which subsystems to keep.**



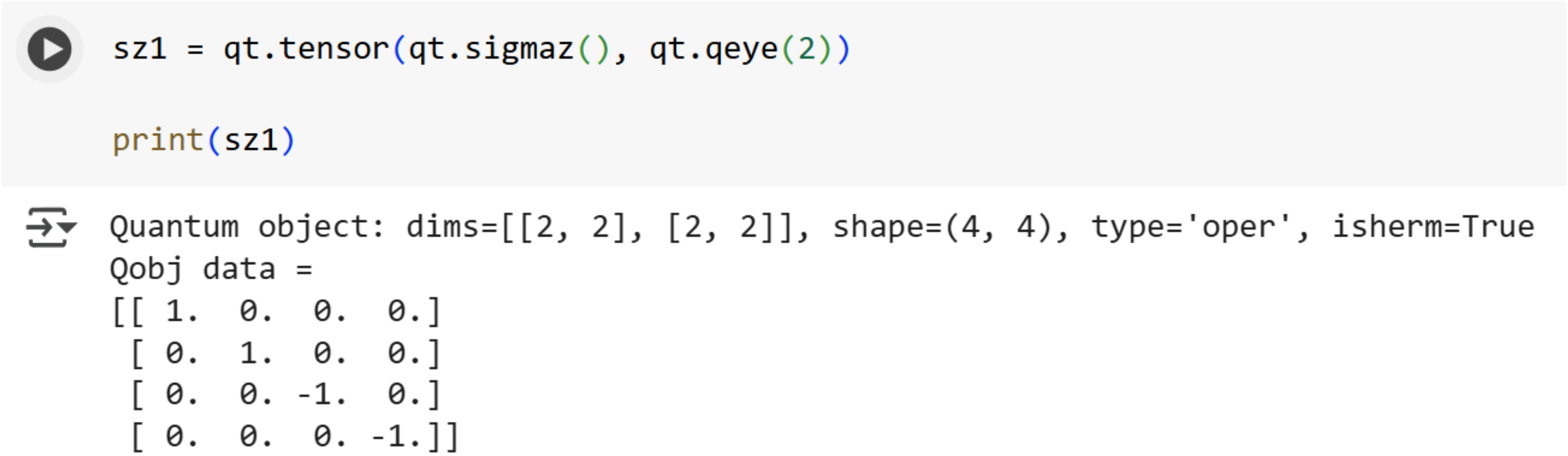
```
psi = (qt.tensor(qt.basis(2, 0), qt.basis(2, 1))+
       qt.tensor(qt.basis(2, 1), qt.basis(2, 0))).unit()
print(psi)
print(psi.ptrace(0))  #leave the first subsystem indexed by 0
#returns an operator (density operator) which is now a fully mixed state (50/50 probability)
```

```
Quantum object: dims=[[2, 2], [1, 1]], shape=(4, 1), type='ket'
Qobj data =
[[0.        ]
 [0.70710678]
 [0.70710678]
 [0.        ]]
Quantum object: dims=[[2], [2]], shape=(2, 2), type='oper', isherm=True
Qobj data =
[[0.5 0. ]
 [0.  0.5]]
```

1. QuTiP-Core, composite operators

Composite systems (operators):

If we want to create a Pauli operator that acts on the first qubit and leaves the second qubit unaffected, we would do:

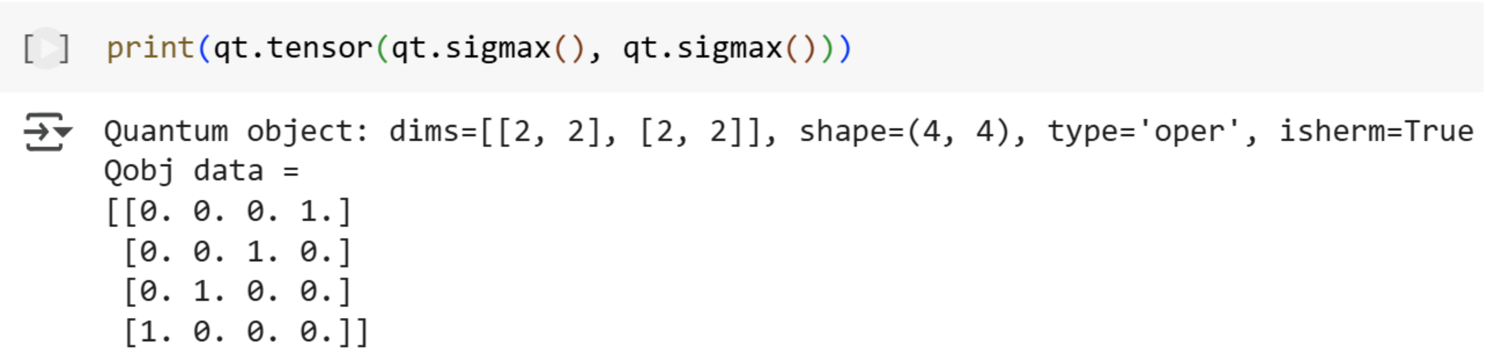


```
sz1 = qt.tensor(qt.sigmaz(), qt.qeye(2))

print(sz1)
```

```
Quantum object: dims=[[2, 2], [2, 2]], shape=(4, 4), type='oper', isherm=True
Qobj data =
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0. -1.  0.]
 [ 0.  0.  0. -1.]]
```

Above we used the qeye(N) function, which generates the identity operator with N quantum states

Using the same method we can create coupling terms like $\sigma_x \otimes \sigma_x$



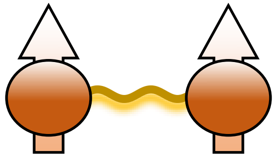
```
print(qt.tensor(qt.sigmax(), qt.sigmax()))
```

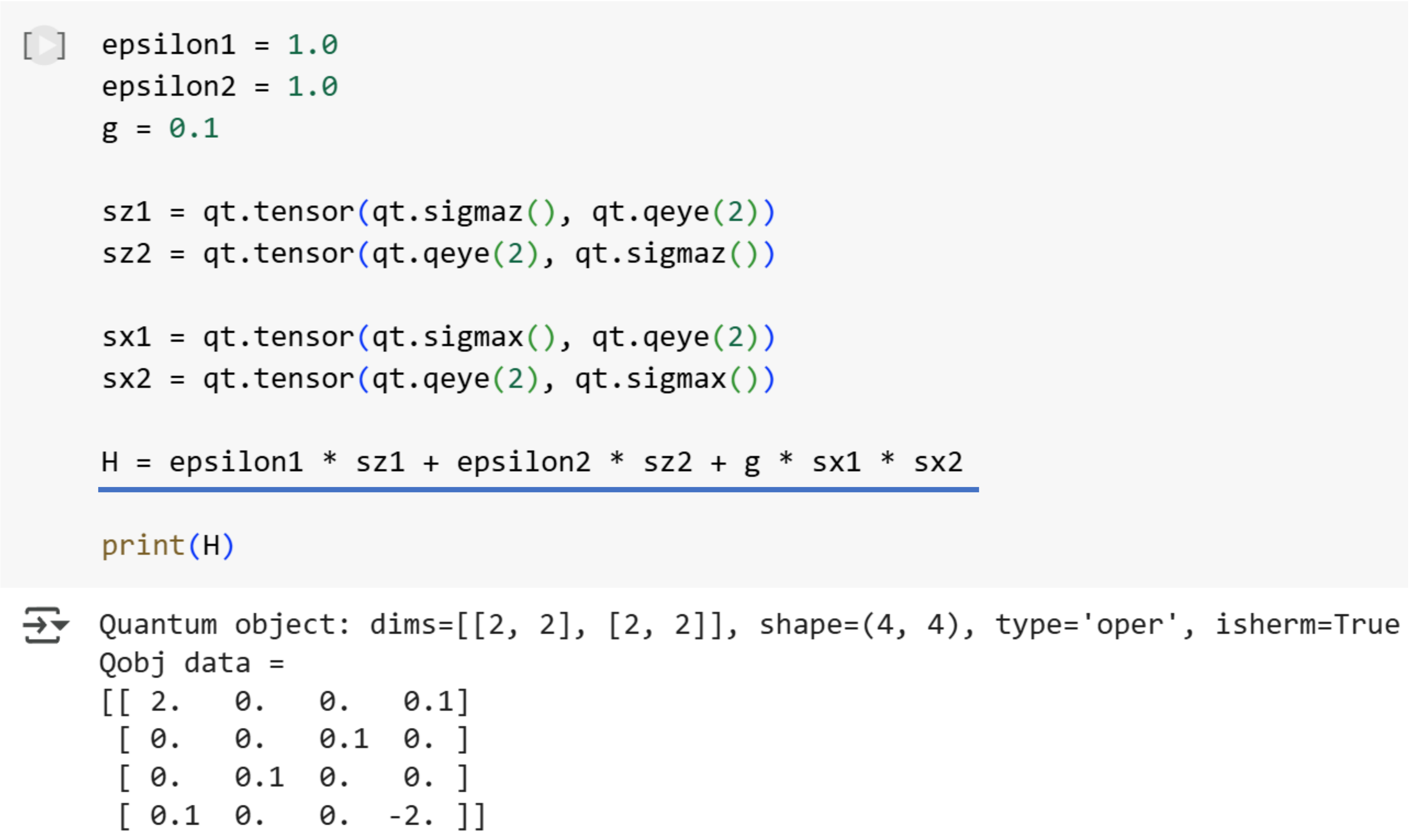
```
Quantum object: dims=[[2, 2], [2, 2]], shape=(4, 4), type='oper', isherm=True
Qobj data =
[[0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [1. 0. 0. 0.]]
```

1. QuTiP-Core, composite operators

Composite systems (Hamiltonians):

By combining all this functionality (operators, arithmetic, tensor structure), we can define useful things like Hamiltonians!





```
epsilon1 = 1.0
epsilon2 = 1.0
g = 0.1

sz1 = qt.tensor(qt.sigmaz(), qt.qeye(2))
sz2 = qt.tensor(qt.qeye(2), qt.sigmaz())

sx1 = qt.tensor(qt.sigmax(), qt.qeye(2))
sx2 = qt.tensor(qt.qeye(2), qt.sigmax())

H = epsilon1 * sz1 + epsilon2 * sz2 + g * sx1 * sx2

print(H)
```

```
Quantum object: dims=[[2, 2], [2, 2]], shape=(4, 4), type='oper', isherm=True
Qobj data =
[[ 2.   0.   0.   0.1]
 [ 0.   0.   0.1  0. ]
 [ 0.   0.1  0.   0. ]
 [ 0.1  0.   0.  -2. ]]
```

1. QuTiP-Core: Schrodinger equation

With states and operators (Hamiltonians) we are in a good position to solve something like the Schrodinger equation!

$$\frac{\mathrm{d}\,\psi}{\mathrm{d}\,t} = -\mathrm{i} H \psi$$

Numerically we have a few options (depending on if H itself is time-dependent or not!)

- Just exponentiate the Hamiltonian [easy with qutip, U=(-1.0j * t * H).expm()]:

$$\psi(t) = U(t)\psi_0 \qquad U(t) = \mathrm{e}^{-\mathrm{i}Ht}$$

- In practice, this is not ideal for large problems (U is a very dense object), but useful if you want to try many different initial conditions with the same propagator (also see propagator() function)

- **More commonly, we simply case the equation as a coupled initial-value problem ODE, and treat it with standard numerical ODE techniques (Runge-Kutte, Adams, Vern, are all provided as options).**

1. QuTiP-Core

Solvers

mesolve
Lindblad master equation

mcsolve + nm_mcsolve
Monte-Carlo master equation

brmesolve
Bloch-Redfield master equation

fmmesolve
Floquet master equation

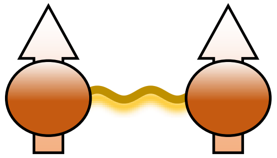
krylovsolve
Krylov subspace solver

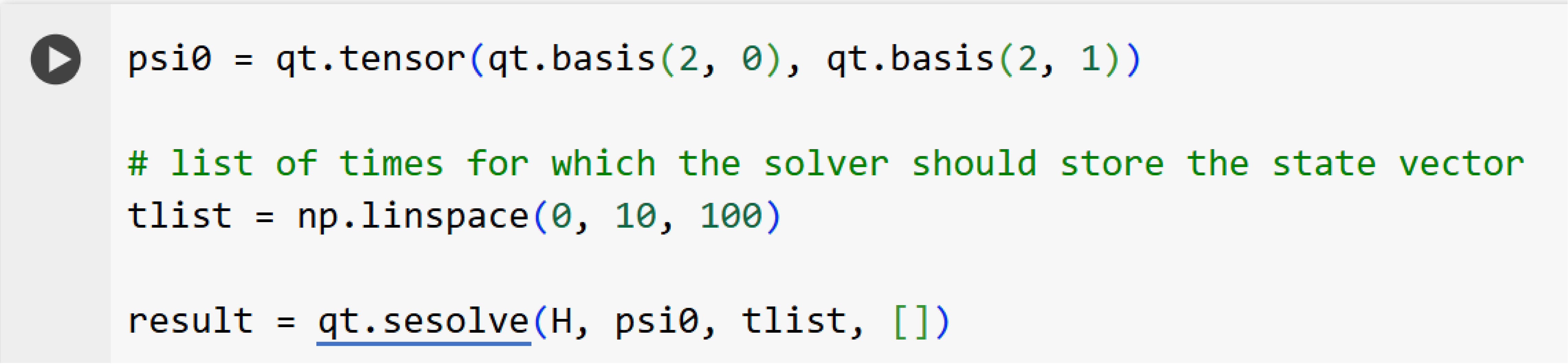
smesolve
Stochastic master equation

HEOMsolver
Hierarchical equations of motion

PIQS
Permutational invariant systems

```
H = epsilon1 * sz1 + epsilon2 * sz2 + g * sx1 * sx2
```

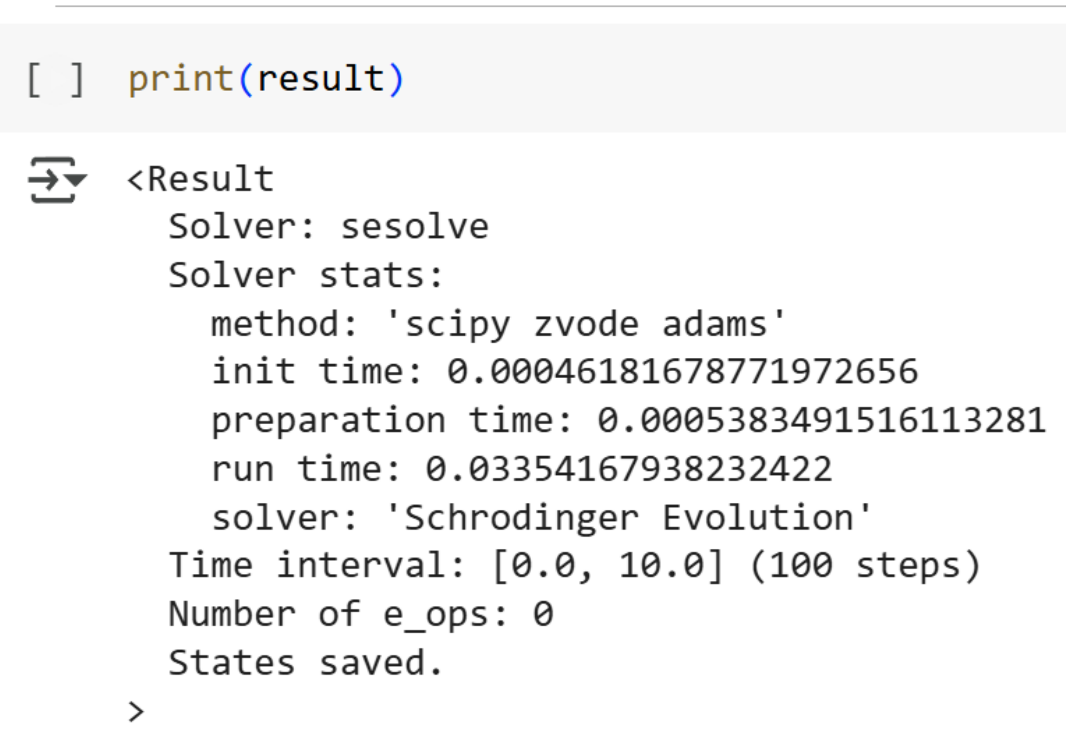




```
psi0 = qt.tensor(qt.basis(2, 0), qt.basis(2, 1))

# list of times for which the solver should store the state vector
tlist = np.linspace(0, 10, 100)

result = qt.sesolve(H, psi0, tlist, [])
```

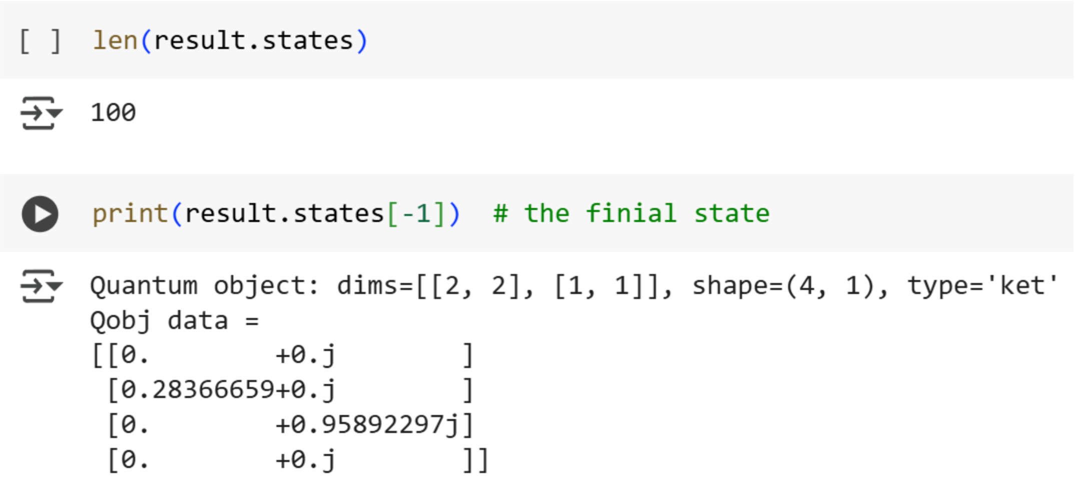
1. QuTiP-Core: Schrodinger equation

The result object contains information about how the solution went, and, importantly, a list of states for times provided in tlist. Note: tlist does not set the time-step of the solver! That is dynamic and done internally.



```
print(result)
```

```
<Result
  Solver: sesolve
  Solver stats:
    method: 'scipy zvode adams'
    init time: 0.00046181678771972656
    preparation time: 0.0005383491516113281
    run time: 0.03354167938232422
    solver: 'Schrodinger Evolution'
  Time interval: [0.0, 10.0] (100 steps)
  Number of e_ops: 0
  States saved.
>
```



```
len(result.states)
```

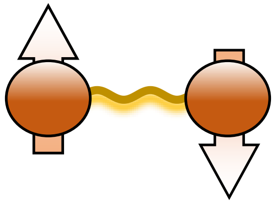
```
100
```

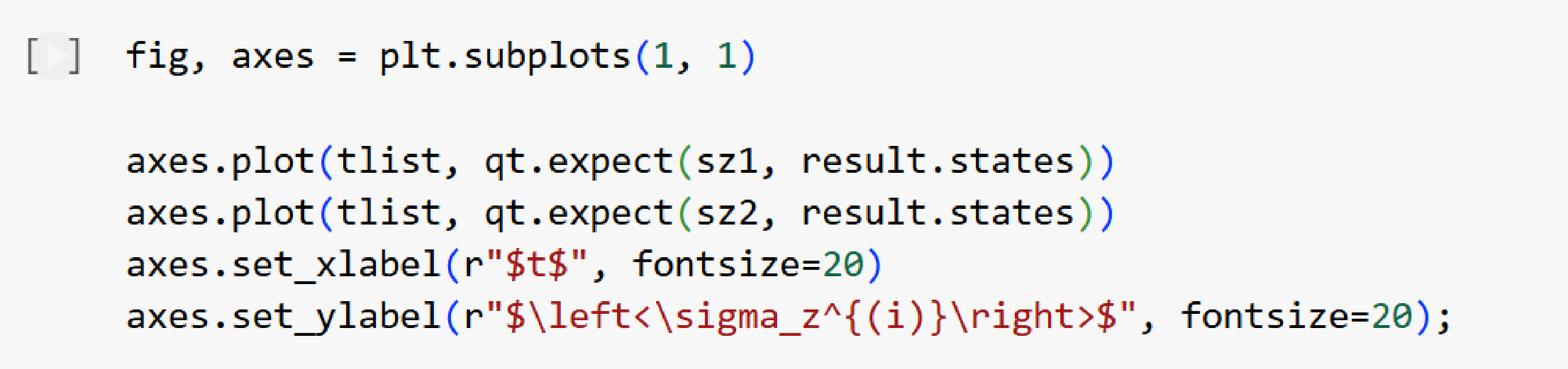
```
print(result.states[-1])  # the finial state
```

```
Quantum object: dims=[[2, 2], [1, 1]], shape=(4, 1), type='ket'
Qobj data =
[[0.        +0.j        ]
 [0.28366659+0.j        ]
 [0.        +0.95892297j]
 [0.        +0.j        ]]
```

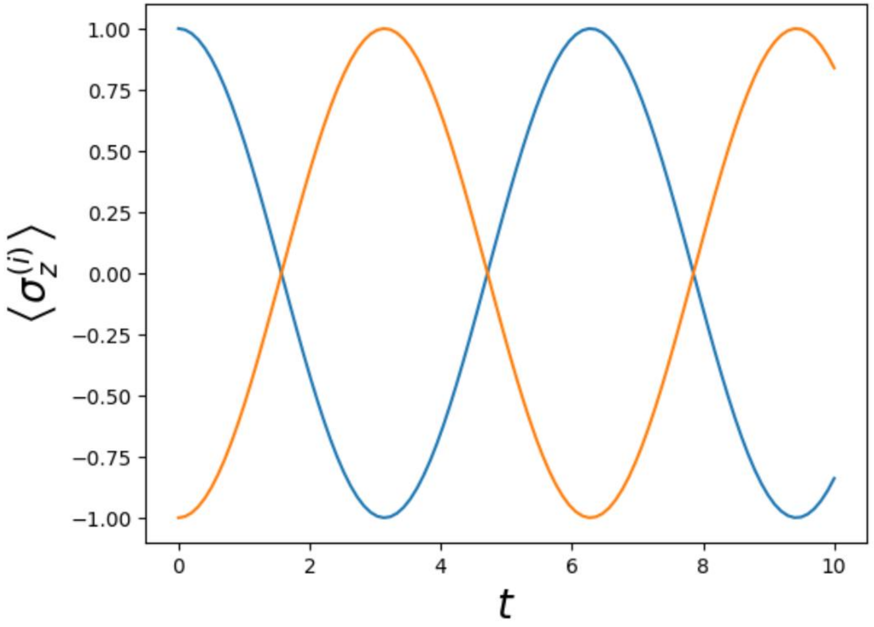
1. QuTiP-Core: Schrodinger equation

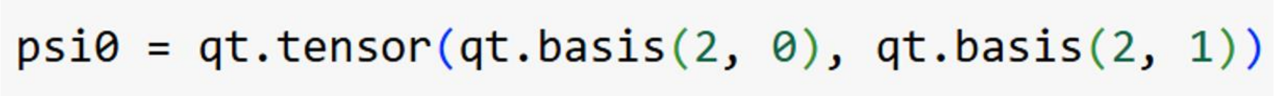
We the states we can calculate expectation values of observables, like $\langle \sigma_z^{(1)} \rangle$





```
fig, axes = plt.subplots(1, 1)

axes.plot(tlist, qt.expect(sz1, result.states))
axes.plot(tlist, qt.expect(sz2, result.states))
axes.set_xlabel(r"$t$", fontsize=20)
axes.set_ylabel(r"$\left<\sigma_z^{(i)}\right>$", fontsize=20);
```





```
psi0 = qt.tensor(qt.basis(2, 0), qt.basis(2, 1))
```

Note, with qubits, the state basis(2,0) is the excited state!

(this is a little counter-intuitive, and catches people out)

Here the Schrodinger equation tells us that the qubits, when on resonance just exchange energy with a frequency proportional to their coupling strength

1. QuTiP-Core: Dissipation and noise

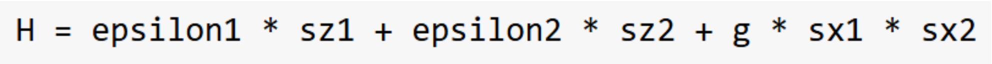
Dissipation: One of the primary features of QuTiP is in the easy ability to simulate open quantum systems.

In adding dissipation, we introduce classical uncertainty and move away from solving the Schrodinger equation with pure states (vectors). Instead, we consider mixtures of pure states, which most conveniently can be written as density operators (like mixtures of projection operators onto pure states) of the form:

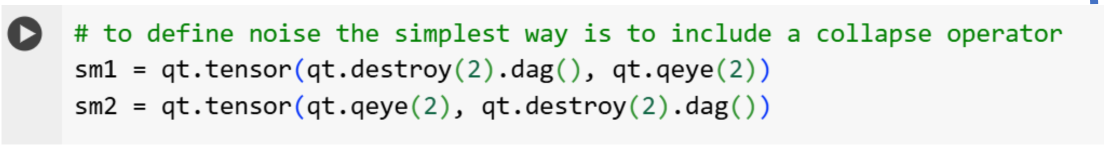
$$\rho = \sum_k p_k |\psi_k\rangle\langle\psi_k|$$

The most common equation of motion for open systems is a **Lindblad master equation:**

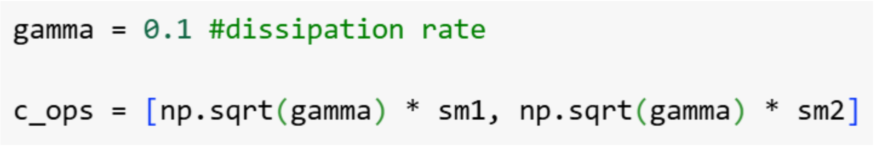
$$\dot{\rho}(t) = -\frac{i}{\hbar}[H(t), \rho(t)] + \sum_n \frac{1}{2}\left[2C_n\rho(t)C_n^\dagger - \rho(t)C_n^\dagger C_n - C_n^\dagger C_n\rho(t)\right]$$



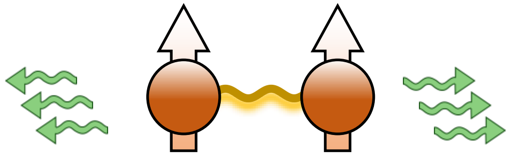
```
H = epsilon1 * sz1 + epsilon2 * sz2 + g * sx1 * sx2
```



```
# to define noise the simplest way is to include a collapse operator
sm1 = qt.tensor(qt.destroy(2).dag(), qt.qeye(2))
sm2 = qt.tensor(qt.qeye(2), qt.destroy(2).dag())
```



```
gamma = 0.1 #dissipation rate

c_ops = [np.sqrt(gamma) * sm1, np.sqrt(gamma) * sm2]
```



1. QuTiP-Core: Dissipation and noise

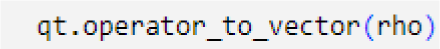
Dissipation: super-operators:

The master equation in this form is a matrix – matrix ODE, not a matrix-vector ODE. This can still be solved as is (Krauss representation), but for small system sizes it is convenient to instead Convert it to a matrix-vector equation using the concept of 'super-operators'.

Here we convert **N X N** density operators into 1 x N^2 vectors, and
NxN operators into N^2 x N^2 superoperators (sometimes called Liouville space)

$$|\psi_i\rangle\langle\psi_j| \longrightarrow |\psi_j\rangle\otimes|\psi_i\rangle$$

```
rho = qt.basis(2,0) * qt.basis(2,0).dag()
```



```
qt.operator_to_vector(rho)
```

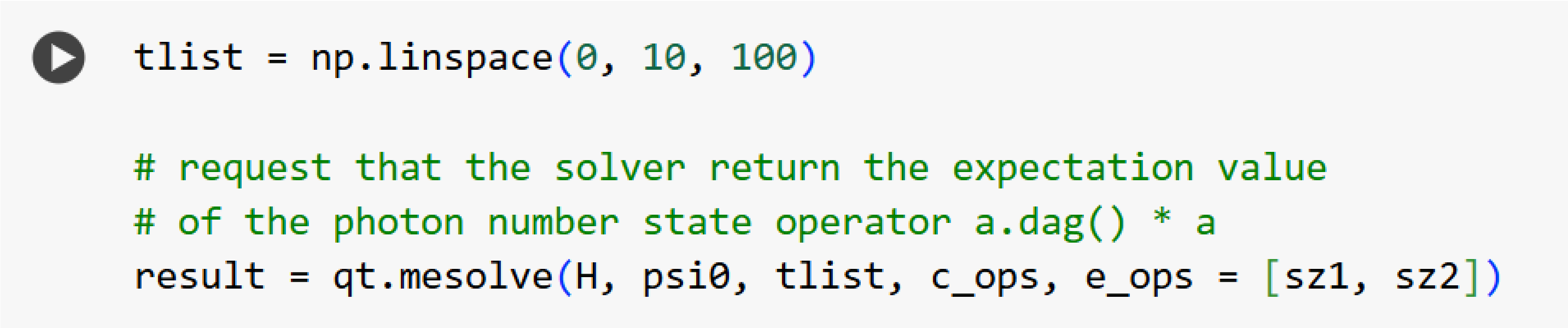
$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$A|\psi_i\rangle\langle\psi_j|B \longrightarrow (B^\dagger\otimes A)(|\psi_j\rangle\otimes|\psi_i\rangle)$$

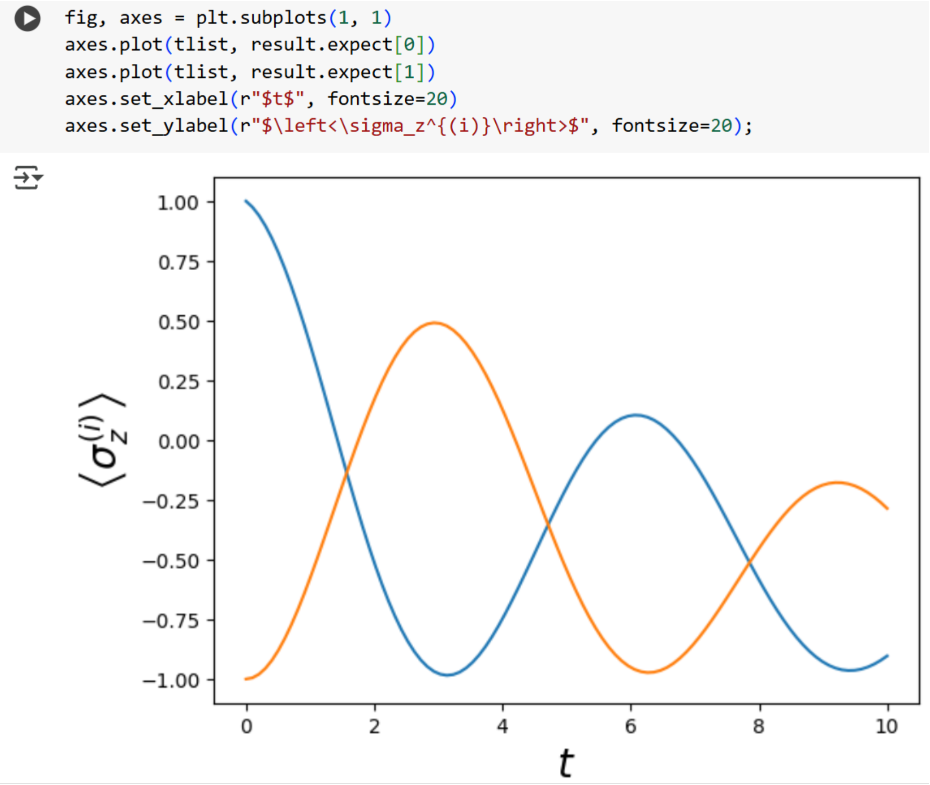
```
qt.sprepost(sz,sz)
```

$$\longrightarrow \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

1. QuTiP-Core: Dissipation and noise



```
tlist = np.linspace(0, 10, 100)

# request that the solver return the expectation value
# of the photon number state operator a.dag() * a
result = qt.mesolve(H, psi0, tlist, c_ops, e_ops = [sz1, sz2])
```



```
fig, axes = plt.subplots(1, 1)
axes.plot(tlist, result.expect[0])
axes.plot(tlist, result.expect[1])
axes.set_xlabel(r"$t$", fontsize=20)
axes.set_ylabel(r"$\left<\sigma_z^{(i)}\right>$", fontsize=20);
```

In this naïve example, the master equation just describes damping on each qubit as a rate ‘gamma’ in their local basis.

The validity of this kind of phenomenological noise depends on underlying physical parameter ranges.

1. QuTiP-Core

Solvers

mesolve
Lindblad master equation

mcsolve + nm_mcsolve
Monte-Carlo master equation

brmesolve
Bloch-Redfield master equation

fmmesolve
Floquet master equation

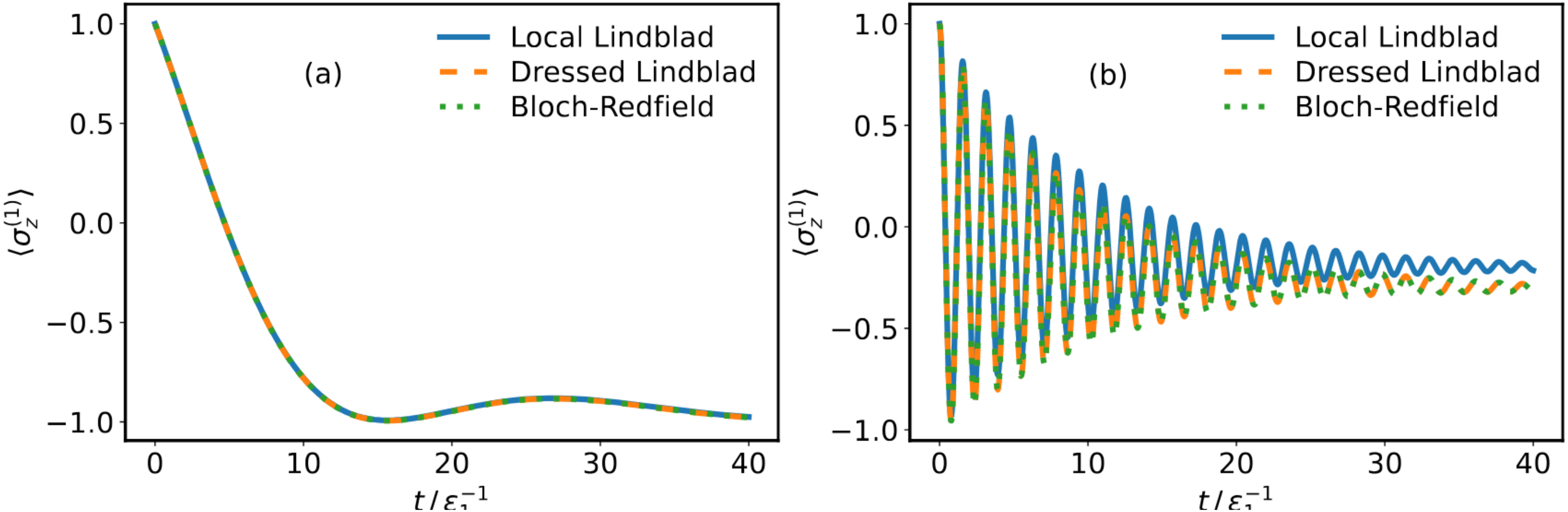
krylovsolve
Krylov subspace solver

smesolve
Stochastic master equation

HEOMsolver
Hierarchical equations of motion

PIQS
Permutational invariant systems

For this reason qutip has many more solvers! something we will revisit in part 2!



Spin-boson model: e.g., Bloch-Redfield

With a bit more work (!) one can arrive at something like

For eigenstates $|\psi_j\rangle$ of H_S, and $c_{j,l} = \langle\psi_j|\sigma_z|\psi_l\rangle$ $\Delta_{j,l} = E_j - E_l$ is the difference in eigenenergies.

$$\frac{\partial}{\partial t}\rho_S(t) = -i[H_S, \rho(t)]$$

$$+\sum_{j>l,l} J(\Delta_{j,l})|c_{j,l}|^2\,(n(\Delta_{j,l})+1)\,[2|\psi_l\rangle\langle\psi_j|\rho_S(t)|\psi_j\rangle\langle\psi_l| - \{|\psi_j\rangle\langle\psi_j|, \rho_S\}]$$

$$+\sum_{j>l,l} J(\Delta_{j,l})|c_{j,l}|^2 n(\Delta_{j,l})\,[2|\psi_j\rangle\langle\psi_l|\rho_S(t)|\psi_l\rangle\langle\psi_j| - \{|\psi_l\rangle\langle\psi_l|, \rho_S\}]$$

The first part is the coherent system evolution, the second part describes spontaneous and stimulated emission, and the third part describes absorption. $n(\omega) = (e^{\omega/T} - 1)^{-1}$

1. QuTiP-Core: adding time-dependence to the system

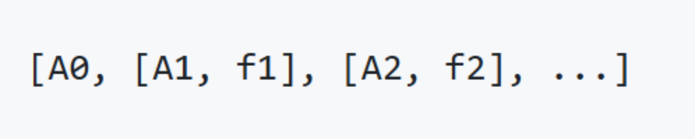
Time-dependent systems: We often need to capture coherent classical driving of a quantum system, which can require introducing time-dependent operators into a Hamiltonian.

In QuTiP we provide a generalization of the Qobj() for time-dependent problems: **QobjEvo**().

QobjEvo() can be instantiated with a list to represent objects like

$$A(t) = \sum_k f_k(t) A_k$$

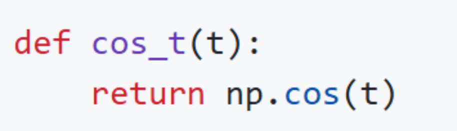
With



```
[A0, [A1, f1], [A2, f2], ...]
```

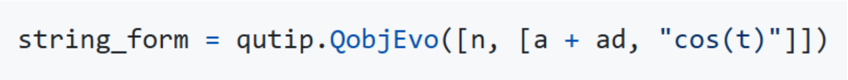
where the time-dependent functions are given by one of:

- A python function, which takes as an argument the time and returns a scalar:



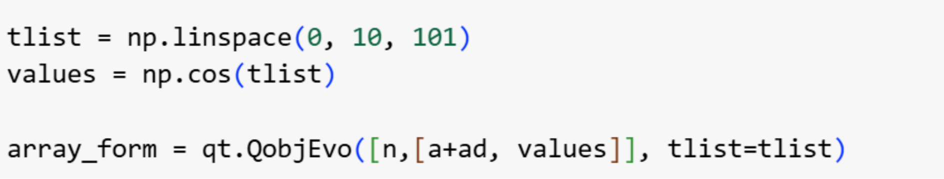
```
def cos_t(t):
    return np.cos(t)
```

- A string which corresponds to a valid combination of Python functions:



```
string_form = qutip.QobjEvo([n, [a + ad, "cos(t)"]])
```

- An array of values for different times (useful for costly complex functions or experimental data).
    - Intermediate undefined values are interpolated.



```
tlist = np.linspace(0, 10, 101)
values = np.cos(tlist)

array_form = qt.QobjEvo([n,[a+ad, values]], tlist=tlist)
```

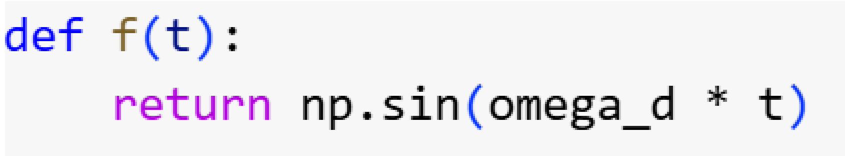
1. QuTiP-Core: adding time-dependence to the system

Time-dependent systems: Consider a standard example of a driven qubit with the time-dependent Hamiltonian

$$H = \frac{\Delta}{2}\sigma_z + \frac{A}{2}\sin(\omega_d t)\sigma_x.$$

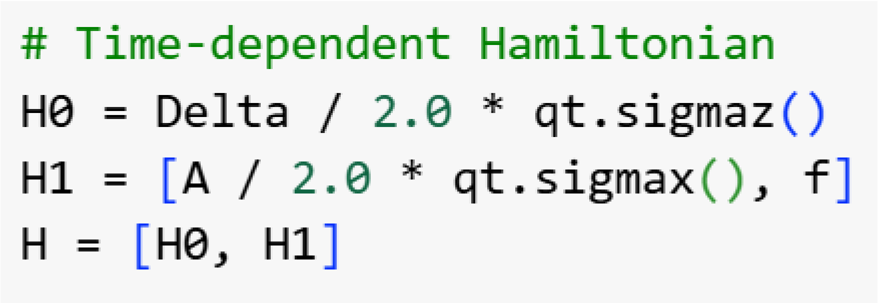
We also include dissipation with rate γ.

First define time-dependence (functional here):



```
def f(t):
    return np.sin(omega_d * t)
```

Then define time-dependent operators (actually calling QobjEvo() is optional):



```
# Time-dependent Hamiltonian
H0 = Delta / 2.0 * qt.sigmaz()
H1 = [A / 2.0 * qt.sigmax(), f]
H = [H0, H1]
```

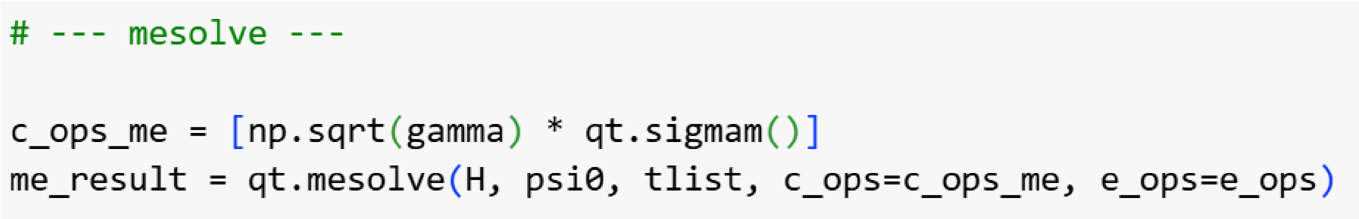
1. QuTiP-Core: adding time-dependence to the system

Time-dependent systems: Consider a standard example of a driven qubit with the time-dependent Hamiltonian

$$H = \frac{\Delta}{2}\sigma_z + \frac{A}{2}\sin(\omega_d t)\sigma_x \, ,$$

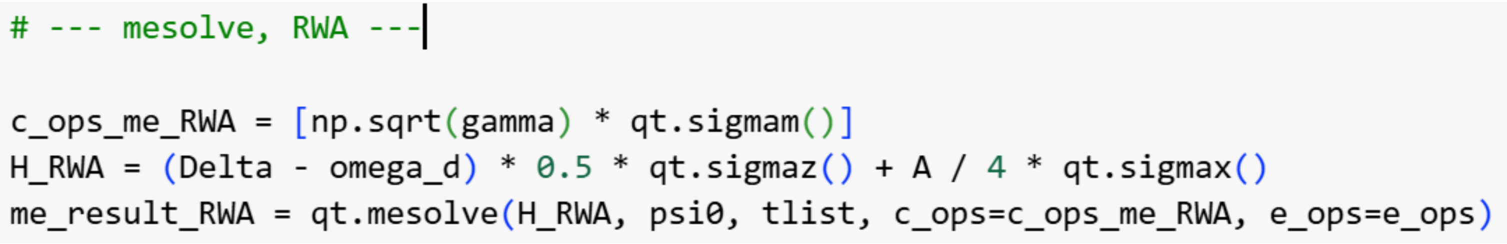
We also include dissipation with rate γ.

Call Lindblad master equation with either just lists or QobjEvo()



```
# --- mesolve ---

c_ops_me = [np.sqrt(gamma) * qt.sigmam()]
me_result = qt.mesolve(H, psi0, tlist, c_ops=c_ops_me, e_ops=e_ops)
```

Lets compare to Rotating wave approximation Hamiltonian: $H_{\mathrm{RWA}} = \frac{\Delta - \omega_d}{2}\sigma_z + \frac{A}{4}\sigma_x \, ,$



```
# --- mesolve, RWA ---

c_ops_me_RWA = [np.sqrt(gamma) * qt.sigmam()]
H_RWA = (Delta - omega_d) * 0.5 * qt.sigmaz() + A / 4 * qt.sigmax()
me_result_RWA = qt.mesolve(H_RWA, psi0, tlist, c_ops=c_ops_me_RWA, e_ops=e_ops)
```

1. QuTiP-Core: adding time-dependence to the system

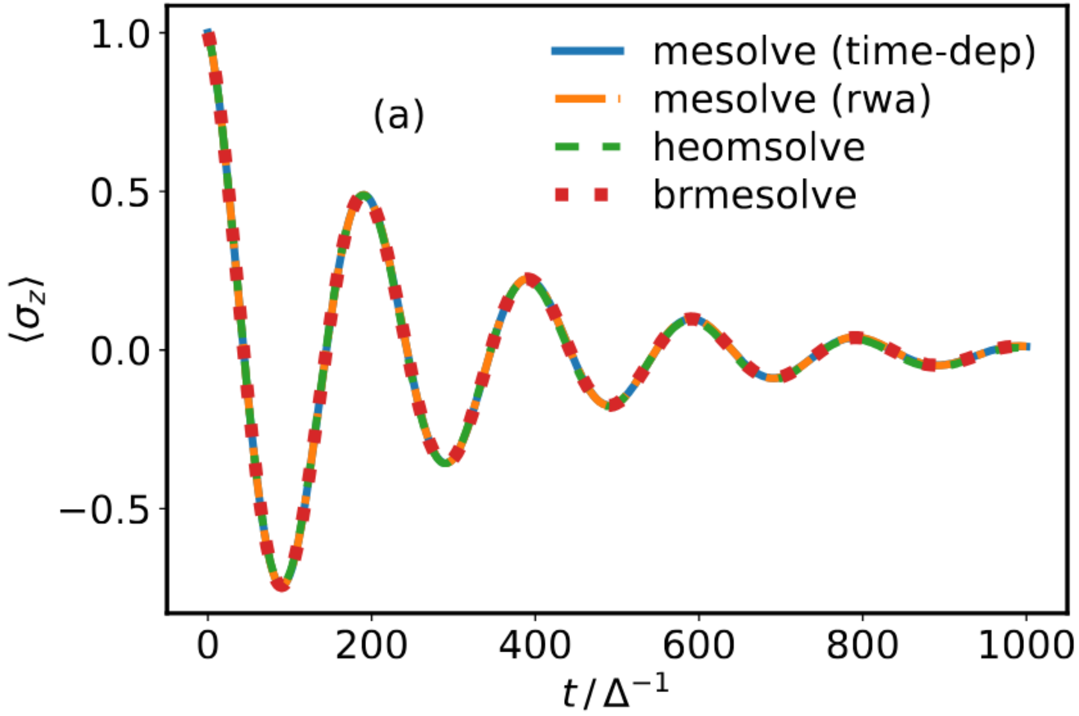
Time-dependent systems: Consider a standard example of a driven qubit with the time-dependent Hamiltonian

$$H = \frac{\Delta}{2}\sigma_z + \frac{A}{2}\sin(\omega_d t)\sigma_x$$

We also include dissipation with rate γ.

$$(A = 0.01\Delta)$$

$$\gamma = 0.005\Delta/(2\pi)$$



Is this always correct? No!

Come back tomorrow to see when we should do something a bit more complicated

Overview:

- History and background of QuTiP
- QuTiP main functionality: noise simulation and open system dynamics
- **QuTiP v5: what has changed?**
- QuTiP-QIP: pulse-level simulator of quantum circuits
- Role of QuTiP and QuTiP-QIP in the future?
  - More developed circuit simulator?
  - Cloud-computer backends (IonQ, IBM, etc)?

What is Qutip 5

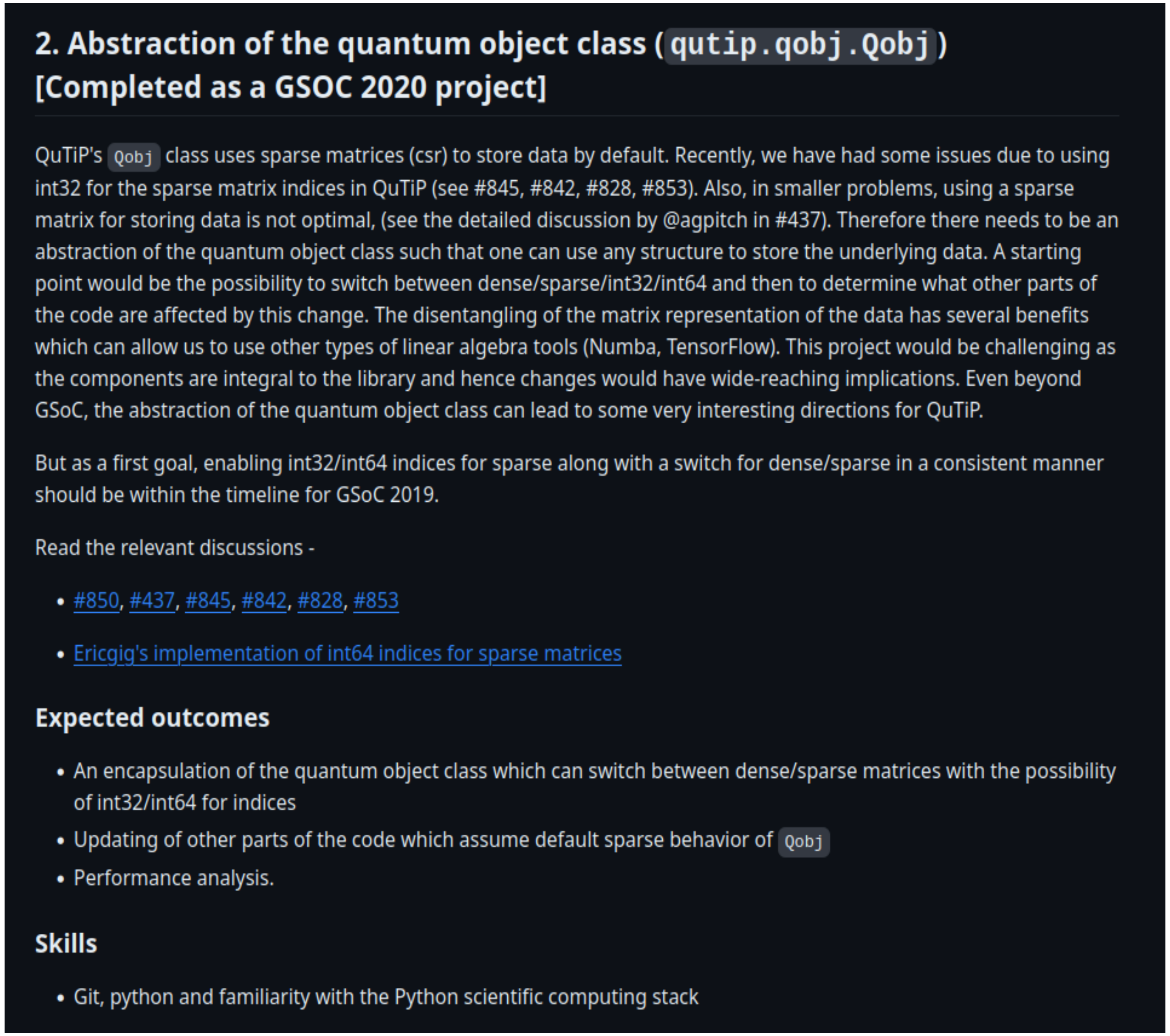
Major rework of the core of QuTiP:

- How Quantum object data is stored:
  - No longer limiter to scipy CSR
  - Dense, Sparse, GPU
- Quantum solver improvements:
  - Uniform class interface
  - Various choice of ODE solvers
  - Better HEOM solver
  - New non-markovian mcsolve method
- New features:
  - Animations
  - Better bloch sphere.
  - Dimension class
- More stability:
  - Support for cython 3
  - Better tests (70% to 85% coverage)

Some Statistics

- 33 Contributors
- >5 GSoC students
- ~4 years of development
- 286 Merged pull requests (out of 1245)
- 1984 Commits (out of 10560)
- Nearly half of github history
  (First v5 PR: #1282, now #2360)

Stage 1: Data Layer, Summer 2020



2. Abstraction of the quantum object class (`qutip.qobj.Qobj`) [Completed as a GSOC 2020 project]

QuTiP's `Qobj` class uses sparse matrices (csr) to store data by default. Recently, we have had some issues due to using int32 for the sparse matrix indices in QuTiP (see #845, #842, #828, #853). Also, in smaller problems, using a sparse matrix for storing data is not optimal, (see the detailed discussion by @agpitch in #437). Therefore there needs to be an abstraction of the quantum object class such that one can use any structure to store the underlying data. A starting point would be the possibility to switch between dense/sparse/int32/int64 and then to determine what other parts of the code are affected by this change. The disentangling of the matrix representation of the data has several benefits which can allow us to use other types of linear algebra tools (Numba, TensorFlow). This project would be challenging as the components are integral to the library and hence changes would have wide-reaching implications. Even beyond GSoC, the abstraction of the quantum object class can lead to some very interesting directions for QuTiP.

But as a first goal, enabling int32/int64 indices for sparse along with a switch for dense/sparse in a consistent manner should be within the timeline for GSoC 2019.

Read the relevant discussions -

- #850, #437, #845, #842, #828, #853
- Ericgig's implementation of int64 indices for sparse matrices

Expected outcomes

- An encapsulation of the quantum object class which can switch between dense/sparse matrices with the possibility of int32/int64 for indices
- Updating of other parts of the code which assume default sparse behavior of `Qobj`
- Performance analysis.

Skills

- Git, python and familiarity with the Python scientific computing stack

Started with feeling limited by Qobj's `fastsparse` CSR format.

Can we allow for arbitrary data formats, and make them interchangeable?

Jake Lishman who was actively contributing took it as GSoC 2020 project, supervised by Eric Giguere

`dev.major` branch created in June 2020.

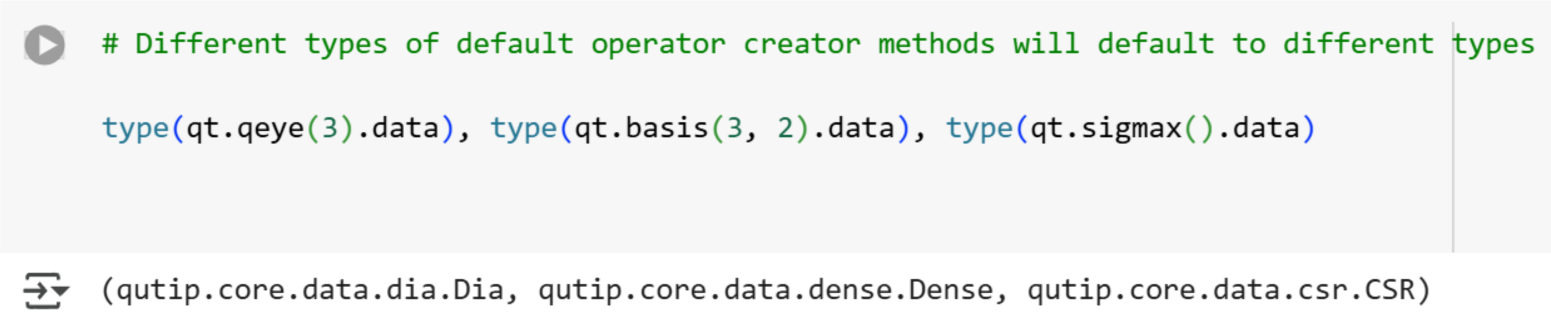
By August 2020:

- CSR, Dense created
- fastsparse removed
- dispatcher created
- Over 20000 lines of code changed.

Stage 1: Data Layer, Summer 2020

Most objects in QuTiP will come with a default data-type mostly suitable for it:

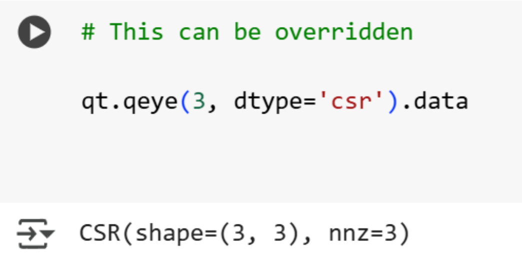
- Basis() will return a dense state
- Destroy() and identity qeye() will return a diagonal sparse format
- Pauli matrices (sigmaz() etc) will return sparse CSR:



```
# Different types of default operator creator methods will default to different types

type(qt.qeye(3).data), type(qt.basis(3, 2).data), type(qt.sigmax().data)
```

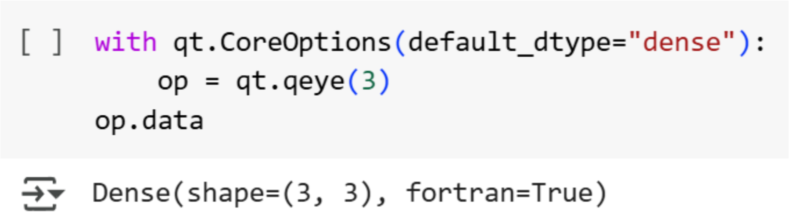
```
(qutip.core.data.dia.Dia, qutip.core.data.dense.Dense, qutip.core.data.csr.CSR)
```

We can override this default behavior by manually setting dtype, or using a default_dtype to override:



```
# This can be overridden

qt.qeye(3, dtype='csr').data
```

```
CSR(shape=(3, 3), nnz=3)
```

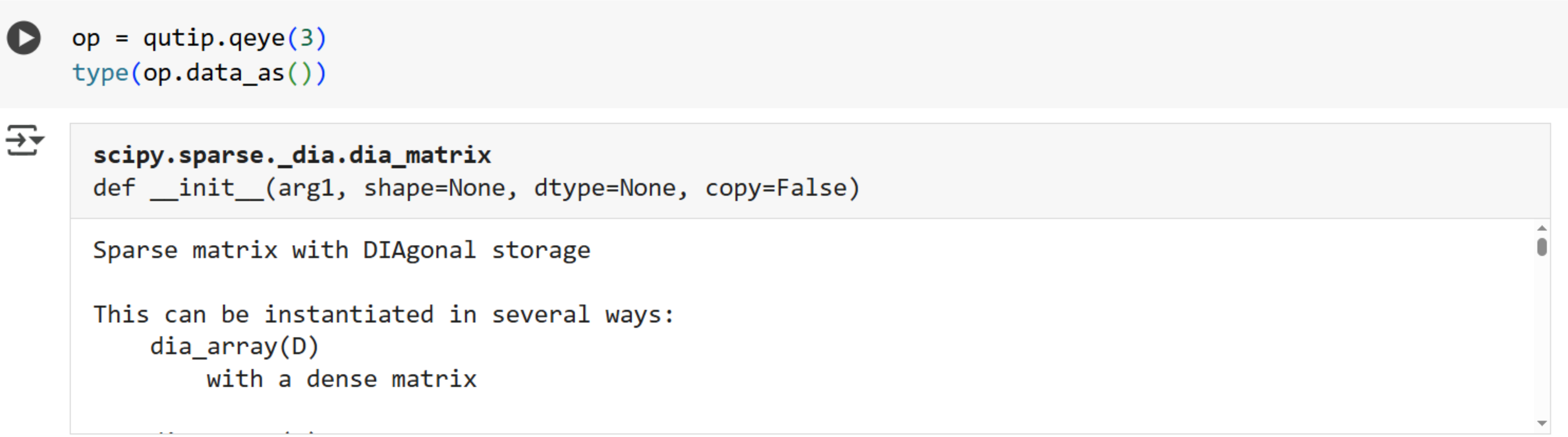


```
with qt.CoreOptions(default_dtype="dense"):
    op = qt.qeye(3)
op.data
```

```
Dense(shape=(3, 3), fortran=True)
```

Stage 1: Data Layer, Summer 2020

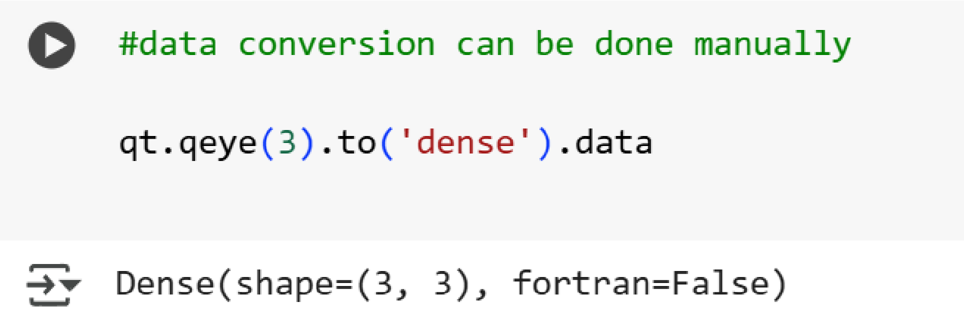
Important change: to obtain the underlying data structure, new call is:



```
op = qutip.qeye(3)
type(op.data_as())
```

```
scipy.sparse._dia.dia_matrix
def __init__(arg1, shape=None, dtype=None, copy=False)

Sparse matrix with DIAgonal storage

This can be instantiated in several ways:
    dia_array(D)
        with a dense matrix
```

To manually change an object from one data format to another we can do:



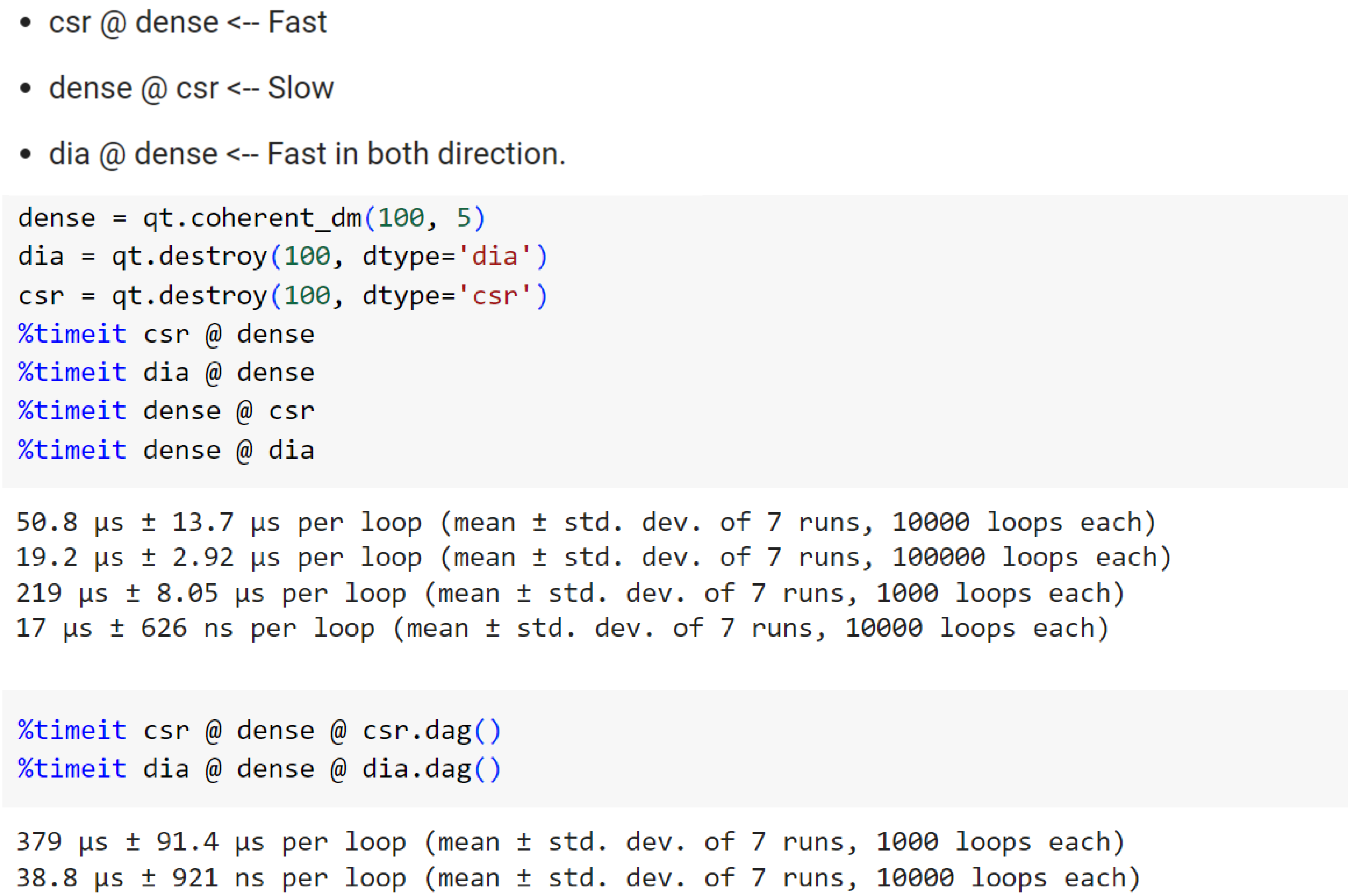
```
#data conversion can be done manually

qt.qeye(3).to('dense').data
```

```
Dense(shape=(3, 3), fortran=False)
```

`Qobj.full()` always returns a dense numpy array

Stage 1: Data Layer, Summer 2020

Arithmetic between different data types can be optimal (or not!):



- csr @ dense <-- Fast
- dense @ csr <-- Slow
- dia @ dense <-- Fast in both direction.

```
dense = qt.coherent_dm(100, 5)
dia = qt.destroy(100, dtype='dia')
csr = qt.destroy(100, dtype='csr')
%timeit csr @ dense
%timeit dia @ dense
%timeit dense @ csr
%timeit dense @ dia
```

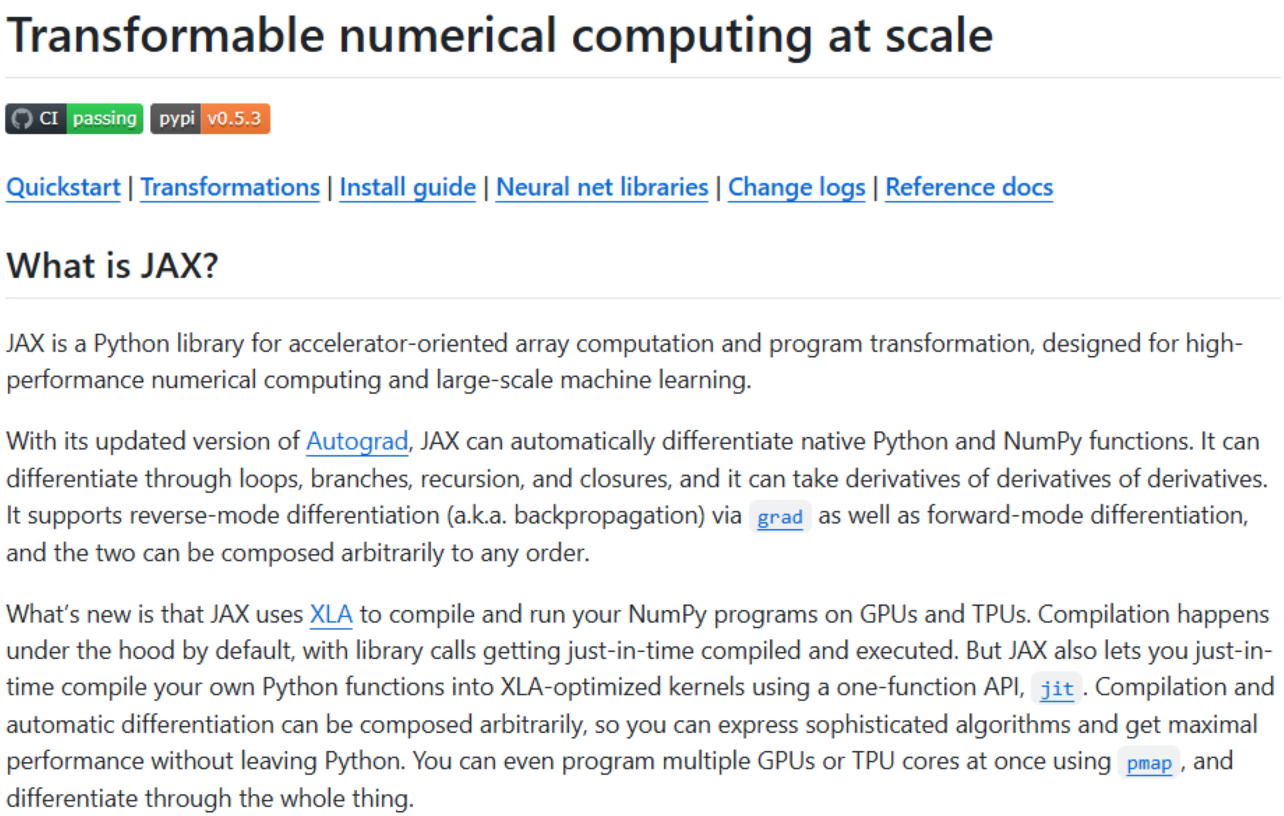
```
50.8 µs ± 13.7 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
19.2 µs ± 2.92 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
219 µs ± 8.05 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
17 µs ± 626 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%timeit csr @ dense @ csr.dag()
%timeit dia @ dense @ dia.dag()
```

```
379 µs ± 91.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
38.8 µs ± 921 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Stage 2: GPU, Fall 2022, QuTiP-JAX

The main purpose of this work was to enable more complex data formats to be added smoothly. First attempts with cuPY and tensorflow were abandoned in favour of the popular JAX library:



Transformable numerical computing at scale

CI passing | pypi v0.5.3

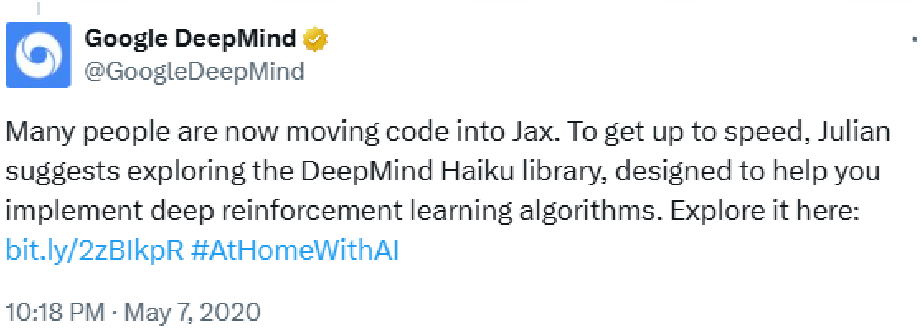
[Quickstart](#) | [Transformations](#) | [Install guide](#) | [Neural net libraries](#) | [Change logs](#) | [Reference docs](#)

What is JAX?

JAX is a Python library for accelerator-oriented array computation and program transformation, designed for high-performance numerical computing and large-scale machine learning.

With its updated version of Autograd, JAX can automatically differentiate native Python and NumPy functions. It can differentiate through loops, branches, recursion, and closures, and it can take derivatives of derivatives of derivatives. It supports reverse-mode differentiation (a.k.a. backpropagation) via `grad` as well as forward-mode differentiation, and the two can be composed arbitrarily to any order.

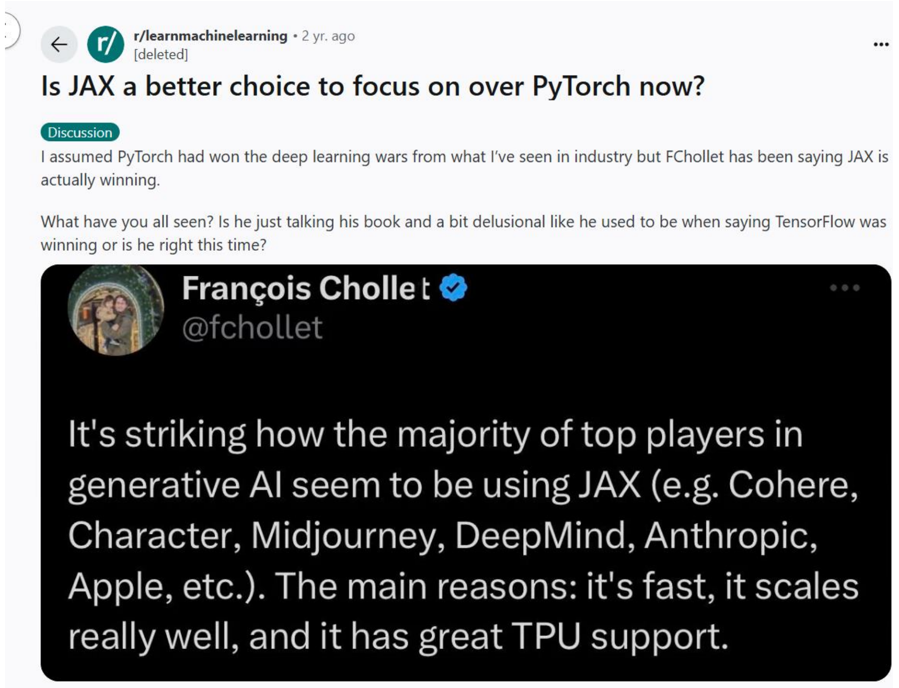
What's new is that JAX uses XLA to compile and run your NumPy programs on GPUs and TPUs. Compilation happens under the hood by default, with library calls getting just-in-time compiled and executed. But JAX also lets you just-in-time compile your own Python functions into XLA-optimized kernels using a one-function API, `jit`. Compilation and automatic differentiation can be composed arbitrarily, so you can express sophisticated algorithms and get maximal performance without leaving Python. You can even program multiple GPUs or TPU cores at once using `pmap`, and differentiate through the whole thing.



Google DeepMind
@GoogleDeepMind

Many people are now moving code into Jax. To get up to speed, Julian suggests exploring the DeepMind Haiku library, designed to help you implement deep reinforcement learning algorithms. Explore it here: bit.ly/2zBIkpR #AtHomeWithAI

10:18 PM · May 7, 2020



r/learnmachinelearning • 2 yr. ago
[deleted]

Is JAX a better choice to focus on over PyTorch now?

Discussion

I assumed PyTorch had won the deep learning wars from what I've seen in industry but FChollet has been saying JAX is actually winning.

What have you all seen? Is he just talking his book and a bit delusional like he used to be when saying TensorFlow was winning or is he right this time?

François Chollet
@fchollet

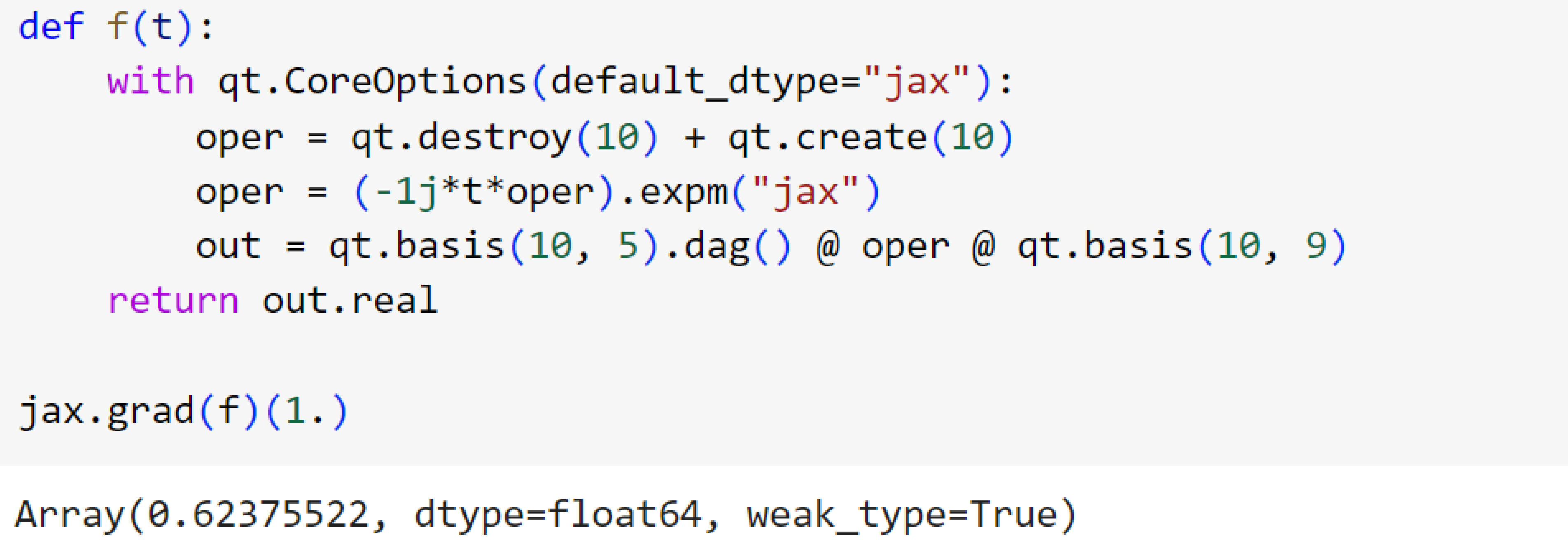
It's striking how the majority of top players in generative AI seem to be using JAX (e.g. Cohere, Character, Midjourney, DeepMind, Anthropic, Apple, etc.). The main reasons: it's fast, it scales really well, and it has great TPU support.

Stage 2: GPU, Fall 2022, QuTiP-JAX

New data-layer implemented by Eric as QuTiP-JAX plugin

Broader support across QuTiP enabled by GSOC project (2024, Rochisha Agarwal)

- **Custom DIA sparse** format
- **Diffrax ODE solver**
- **Autodifferentiation (take gradients of functions!)**

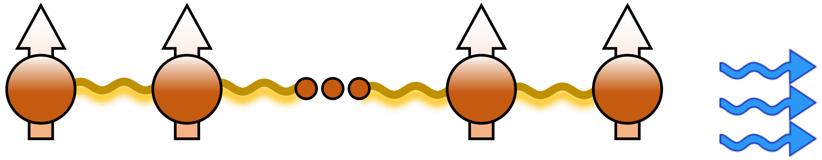


```
def f(t):
    with qt.CoreOptions(default_dtype="jax"):
        oper = qt.destroy(10) + qt.create(10)
        oper = (-1j*t*oper).expm("jax")
        out = qt.basis(10, 5).dag() @ oper @ qt.basis(10, 9)
    return out.real

jax.grad(f)(1.)
```

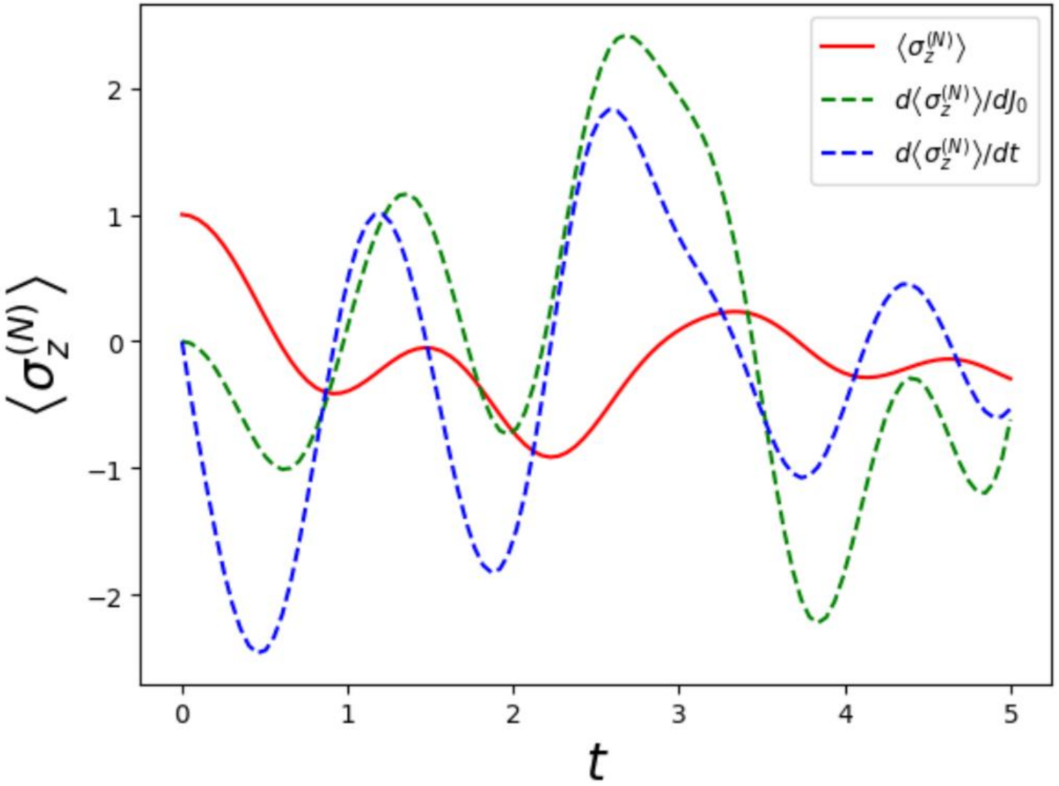
```
Array(0.62375522, dtype=float64, weak_type=True)
```

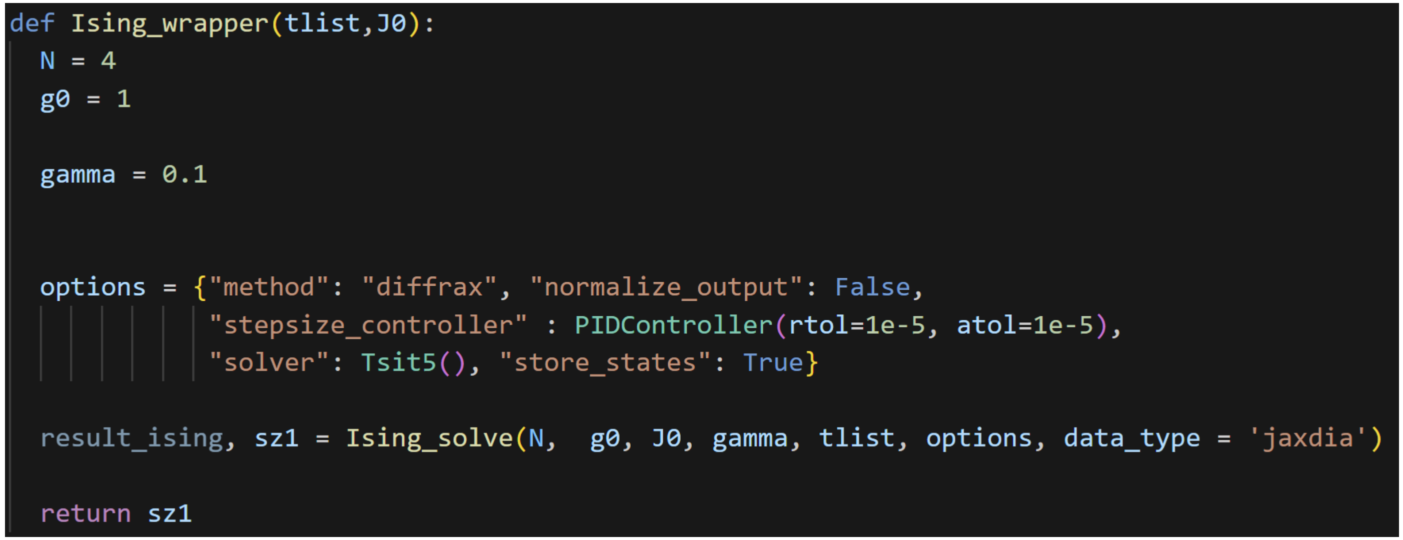
Stage 2: GPU, Fall 2022, QuTiP-JAX

GPU support gives some speedup on large problems (without any explicit effort to parallelize)

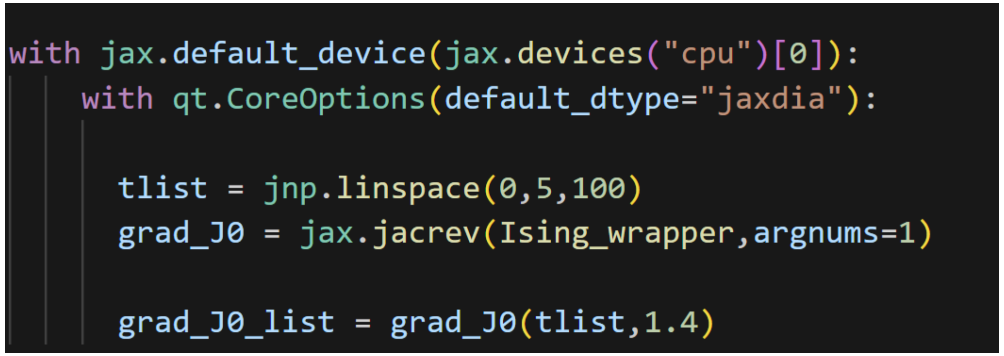


$$H_{\text{sys}} = g\sum_{i=1}^{N}\sigma_z^{(i)} - J\sum_{i=1}^{N-1}\sigma_x^{(i)}\otimes\sigma_x^{(i+1)}$$





```
def Ising_wrapper(tlist,J0):
    N = 4
    g0 = 1

    gamma = 0.1


    options = {"method": "diffrax", "normalize_output": False,
               "stepsize_controller" : PIDController(rtol=1e-5, atol=1e-5),
               "solver": Tsit5(), "store_states": True}

    result_ising, sz1 = Ising_solve(N,  g0, J0, gamma, tlist, options, data_type = 'jaxdia')

    return sz1
```

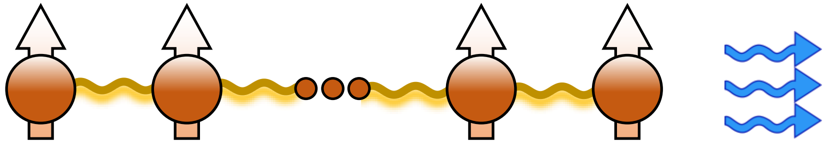


```
with jax.default_device(jax.devices("cpu")[0]):
    with qt.CoreOptions(default_dtype="jaxdia"):

        tlist = jnp.linspace(0,5,100)
        grad_J0 = jax.jacrev(Ising_wrapper,argnums=1)

        grad_J0_list = grad_J0(tlist,1.4)
```

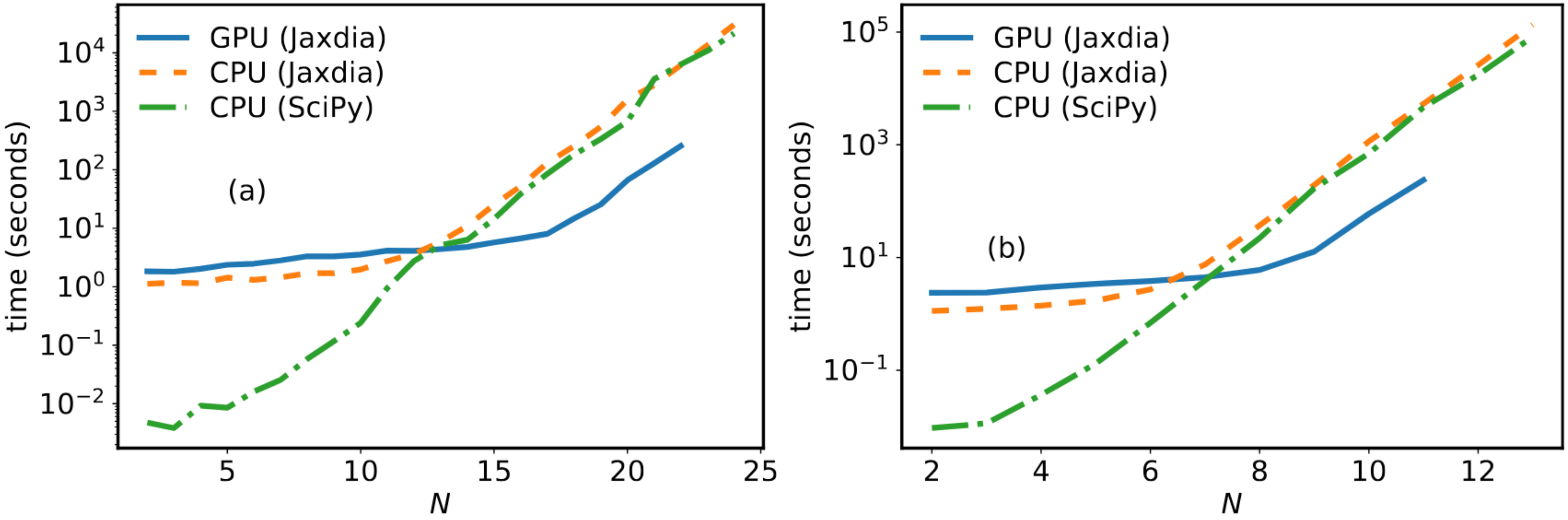
Stage 2: GPU, Fall 2022, QuTiP-JAX

GPU support gives some speedup on large problems (without any explicit effort to parallelize)

$$H_{\rm sys} = g\sum_{i=1}^{N}\sigma_z^{(i)} - J\sum_{i=1}^{N-1}\sigma_x^{(i)}\otimes\sigma_x^{(i+1)}$$



sesolve() mesolve()



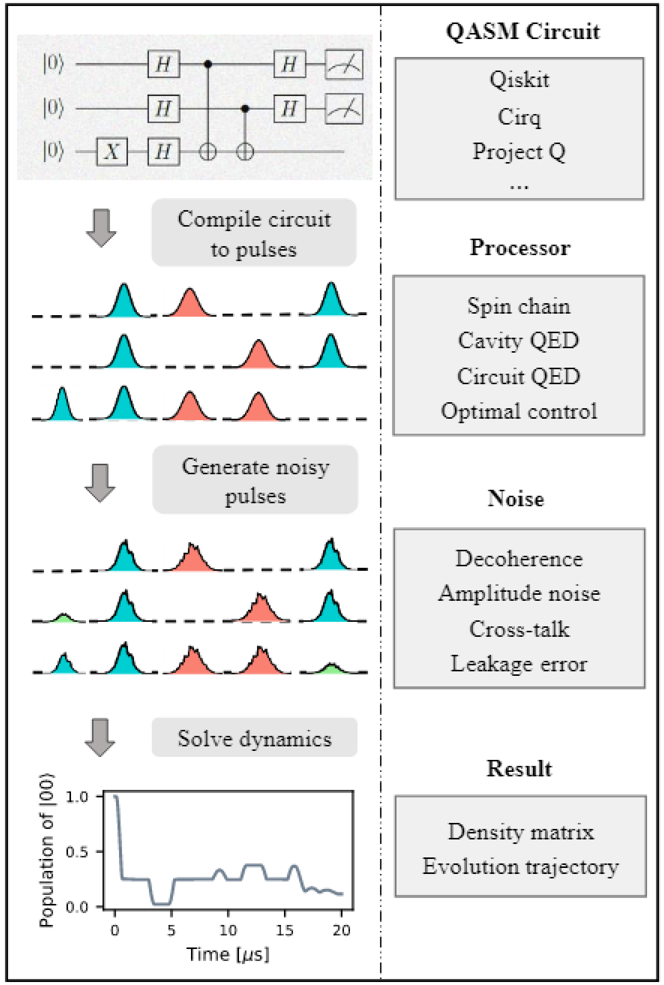
Overview:

- History and background of QuTiP
- QuTiP main functionality: noise simulation and open system dynamics
- QuTiP v5: what has changed?
- **QuTiP-QIP: pulse-level simulator of quantum circuits**
- Role of QuTiP and QuTiP-QIP in the future?
  - More developed circuit simulator?
  - Cloud-computer backends (IonQ, IBM, etc)?

Recent significant development for circuit simulations:

Google
Summer of Code

"Pulse-level noisy quantum circuits with QuTiP",
Boxi Li, Shahnawaz Ahmed, Sidhant Saraogi, Neill Lambert, Franco Nori, Alexander Pitchford, Nathan Shammah, Quantum (2021), arXiv:2105.09902



Compile circuit to pulses

Generate noisy pulses

Solve dynamics

QASM Circuit

Qiskit
Cirq
Project Q
...

Processor

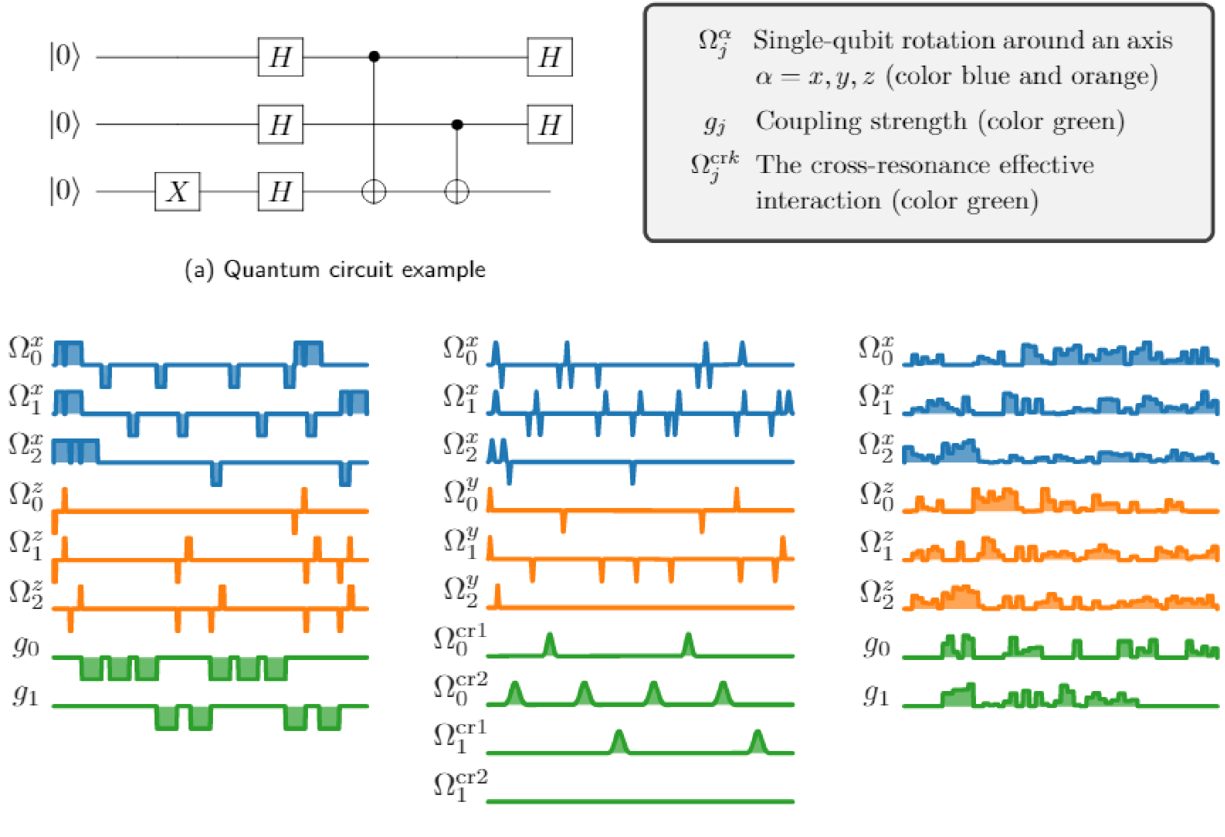
Spin chain
Cavity QED
Circuit QED
Optimal control

Noise

Decoherence
Amplitude noise
Cross-talk
Leakage error

Result

Density matrix
Evolution trajectory



Ω_j^α Single-qubit rotation around an axis $\alpha = x, y, z$ (color blue and orange)

g_j Coupling strength (color green)

Ω_j^{crk} The cross-resonance effective interaction (color green)

(a) Quantum circuit example

(b) Spin chain model

(c) Superconducting qubit model

(d) Optimal control model

A toy example, quantum simulation of quantum dynamics

Google
Summer of Code

$$\psi(t_f) = e^{-i(H_A+H_B)t_f}\psi(0) \approx \left[e^{-iH_A dt}e^{-iH_B dt}\right]^d \psi(0)\,,$$

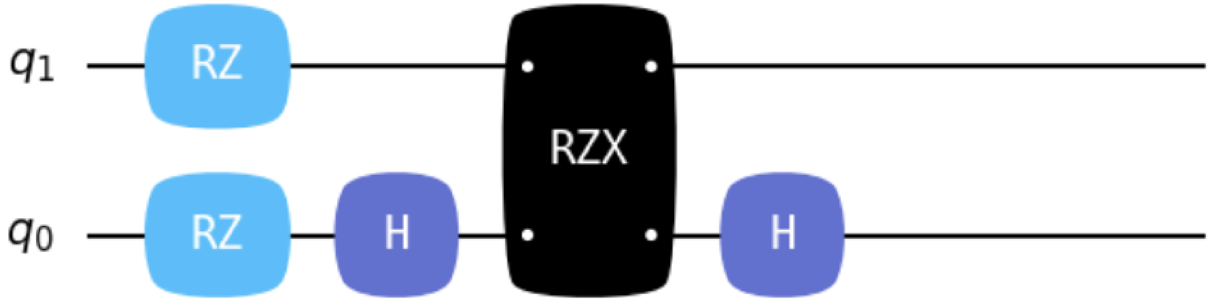
$$H_A = \frac{\epsilon_1}{2}\sigma_z^{(1)} + \frac{\epsilon_2}{2}\sigma_z^{(2)} \quad \text{and}$$

$$H_B = g\sigma_x^{(1)}\sigma_x^{(2)}\,.$$

```
trotter_simulation = QubitCircuit(2)

trotter_simulation.add_gate("RZ", targets=[0], arg_value=(epsilon1 * dt))
trotter_simulation.add_gate("RZ", targets=[1], arg_value=(epsilon2 * dt))

trotter_simulation.add_gate("H", targets=[0])
trotter_simulation.add_gate("RZX", targets=[0, 1], arg_value=g * dt * 2)
trotter_simulation.add_gate("H", targets=[0])
```

A toy example, quantum simulation of quantum dynamics

Google
Summer of Code

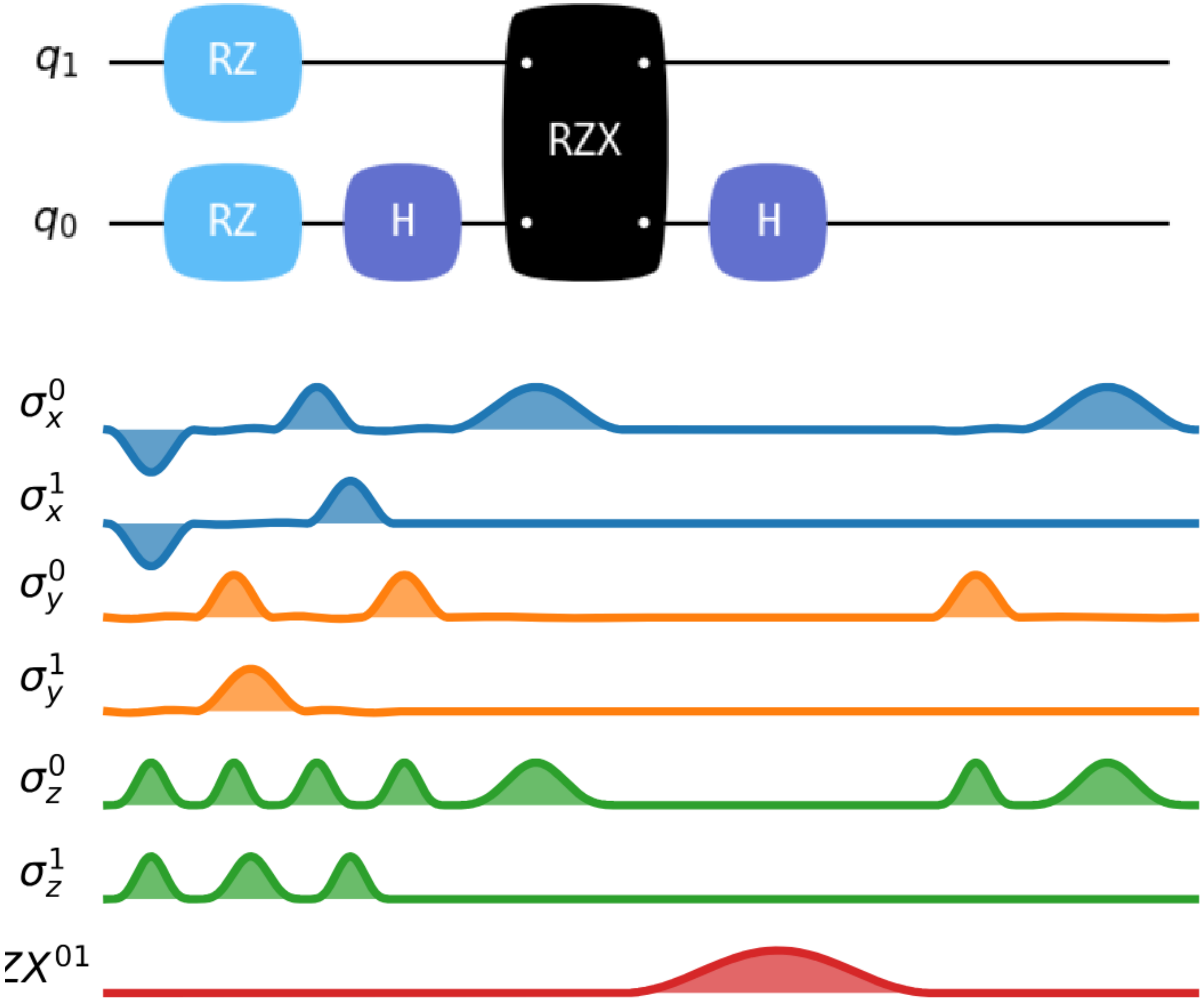


```
processor = SCQubits(num_qubits=2, t1=2e5, t2=2e5)
processor.load_circuit(trotter_simulation)
# Since SCQubit are modelled as qutrit, we need three-level systems here.
init_state = tensor(basis(3, 0), basis(3, 1))
```

```
processor.plot_pulses()
```

A toy example, quantum simulation of quantum dynamics

Google
Summer of Code



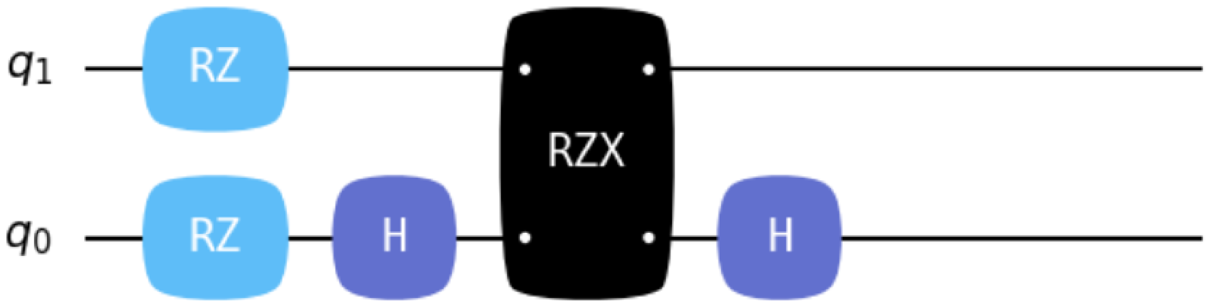
$$H = H_{\mathrm{d}} + \sum_{j=0}^{N-1} \Omega_j^x (a_j^\dagger + a_j) + \Omega_j^y i(a_j^\dagger - a_j)$$

$$+ \sum_{j=0}^{N-2} \Omega_j^{\mathrm{cr}1} \sigma_j^z \sigma_{j+1}^x + \Omega_j^{\mathrm{cr}2} \sigma_j^x \sigma_{j+1}^z,$$

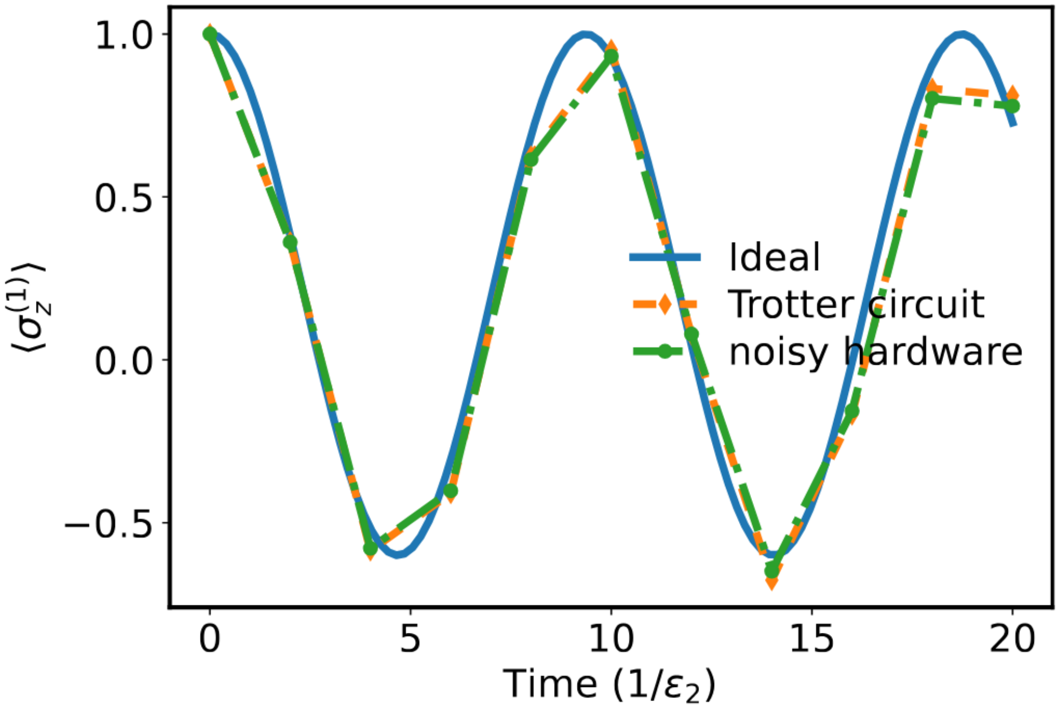
$$H_{\mathrm{d}} = \sum_{j=0}^{N-1} \frac{\alpha_j}{2} a_j^\dagger a_j^\dagger a_j a_j.$$

A toy example, quantum simulation of quantum dynamics

Google
Summer of Code

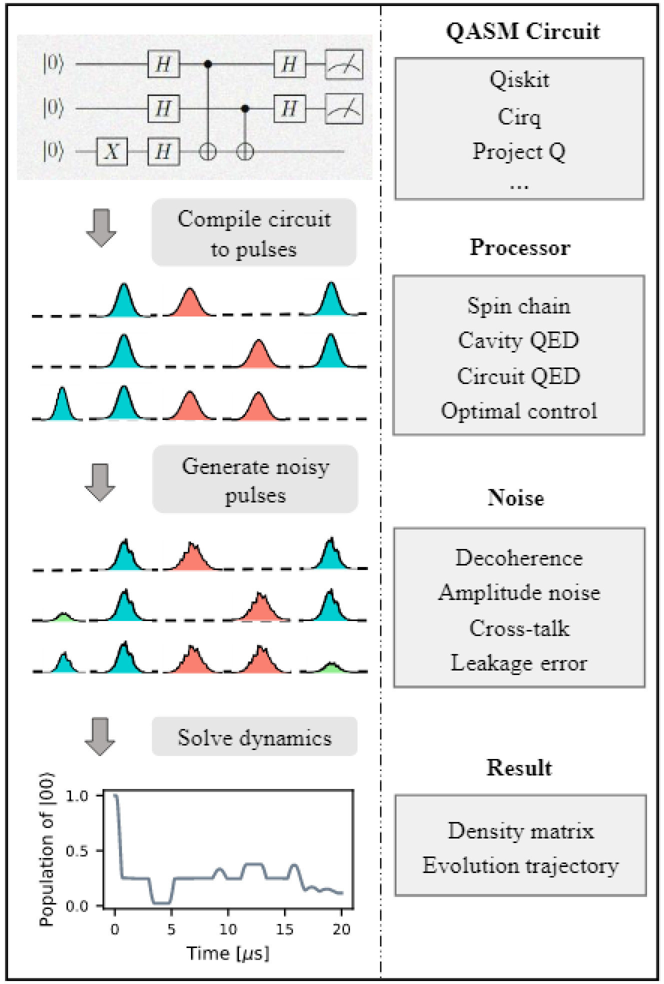


```
processor.run_state(init_state)
```



Recent significant development for circuit simulations:

"Pulse-level noisy quantum circuits with QuTiP",
Boxi Li, Shahnawaz Ahmed, Sidhant Saraogi, Neill Lambert, Franco Nori, Alexander Pitchford, Nathan Shammah, Quantum (2021), arXiv:2105.09902



Pros of QuTiP-QIP:

- Designed for pulse-level noisy simulations, can take advantage of QuTiP solvers
- Comparable in purpose to qiskit-pulse and pulser
- openQASM 2 support

Cons:

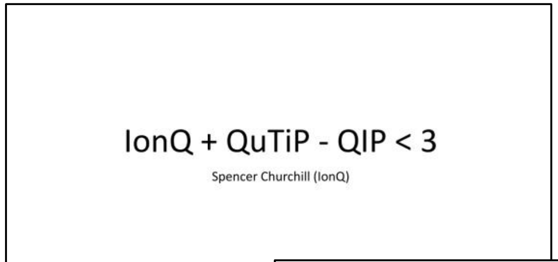
- No circuit optimization
- Limited circuit transpilation/compilation to hardware native gates
- Efficiency not prioritized

Improvement of compilation to native gates underway.

Recent significant development for circuit simulations:

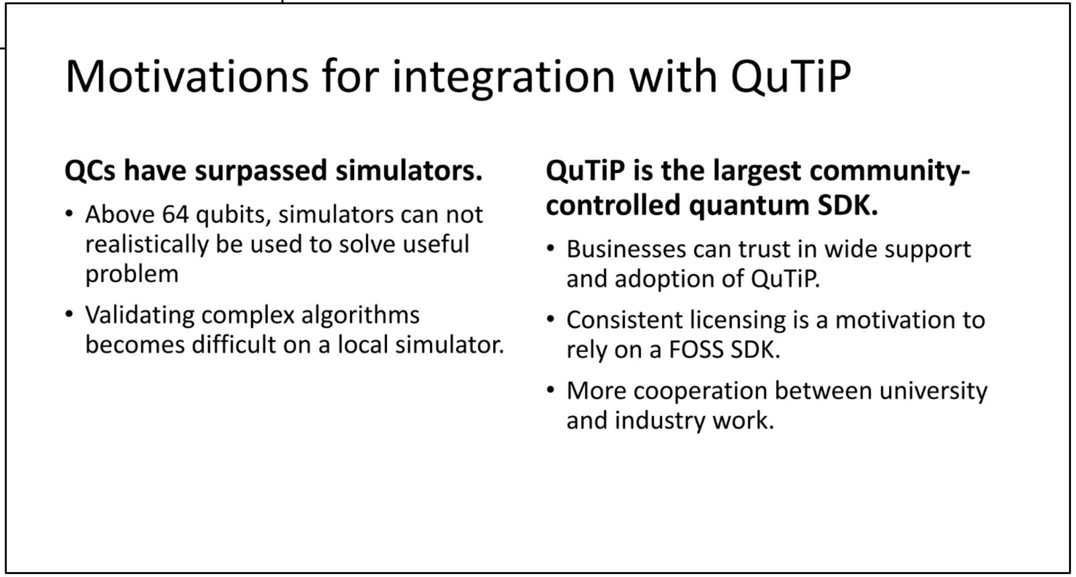
Future of QuTiP-QIP?

- “Potential to be the main academically independent QIP API” – Spencer Churchill
- Should we support as many hardware cloud platforms possible? This is man-power intensive (qiskit backend already fails with qiskit v1.0 release)
- Maybe we can just make it as easy as possible for hardware providers to add support to qutip



IonQ + QuTiP - QIP < 3

Spencer Churchill (IonQ)



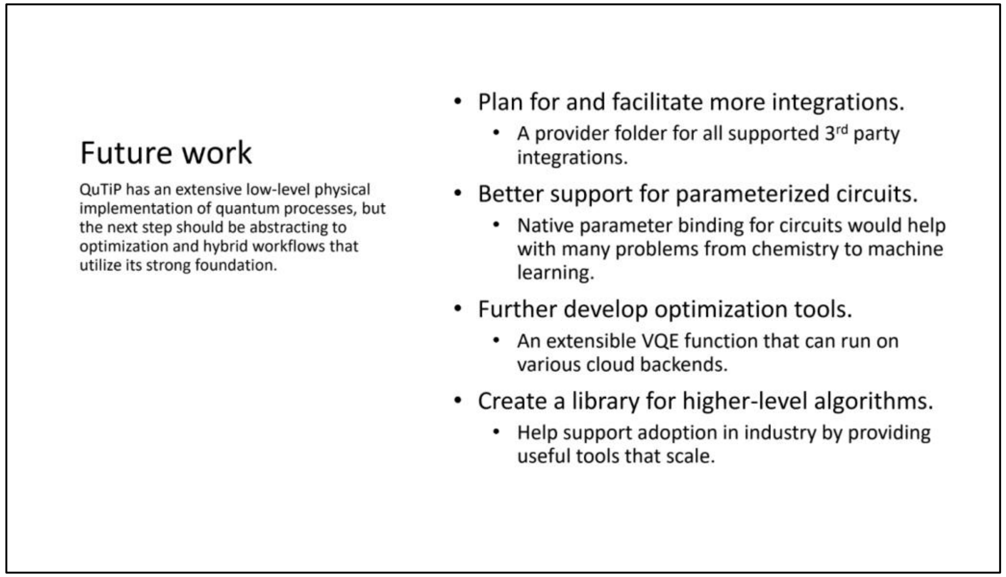
Motivations for integration with QuTiP

QCs have surpassed simulators.

- Above 64 qubits, simulators can not realistically be used to solve useful problem
- Validating complex algorithms becomes difficult on a local simulator.

QuTiP is the largest community-controlled quantum SDK.

- Businesses can trust in wide support and adoption of QuTiP.
- Consistent licensing is a motivation to rely on a FOSS SDK.
- More cooperation between university and industry work.



Future work

QuTiP has an extensive low-level physical implementation of quantum processes, but the next step should be abstracting to optimization and hybrid workflows that utilize its strong foundation.

- Plan for and facilitate more integrations.
  - A provider folder for all supported 3rd party integrations.
- Better support for parameterized circuits.
  - Native parameter binding for circuits would help with many problems from chemistry to machine learning.
- Further develop optimization tools.
  - An extensible VQE function that can run on various cloud backends.
- Create a library for higher-level algorithms.
  - Help support adoption in industry by providing useful tools that scale.

Conclusions and final thoughts

QuTiP is a constantly evolving package. Future goals include:

- More nuanced support for high performance computing
- Quasi-analytic representation of models
- Tensor network methods

What is the future for community driven software like QuTiP?

- **Maintaining** OSS is hard and time-intensive (academics need to write papers). People who do commit effort and time to OSS often move to industry.

- How can we guarantee it has a future?

Relies on users; please tell us what you want, and what doesn't work.