Team Name: **WeAllNight**

```
%%capture
!pip install dynamiqs

import dynamiqs as dq
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
```

1. Simulate the dynamics of cat qubits at the effective Hamiltonian level.

In Tutorial 1, we have introduced that we can dissipatively stabilize a cat qubit by coupling a memory mode that will store our cat qubit to a lossy buffer mode with a specific interaction that exchanges two photons of the memory mode with one photon in the buffer mode.

For this, consider the Lindblad master equation:

$$\frac{d\hat{\rho}}{dt} = L[\hat{\rho}] = -i[\hat{H}, \hat{\rho}] + \kappa_b D[\hat{b}][\hat{\rho}]$$

The Hamiltonian of the system is given by

\begin{aligned} \hat{H} &= \hat{H}_{\mathrm{2ph}} + \hat{H}*d, \qquad \mathrm{with} \quad \hat{H}*{\mathrm{2ph}} &= g_2 {}\hat{a}^\dagger^2 \hat{b} + g_2^* \hat{a}^2 \hat{b}^\dagger, \ \hat{H}_d &= \epsilon_d^* \hat{b} + \epsilon_d \hat{b}^\dagger. \end{aligned}

Here, $\hat{H}_{2ph}$ is the two-photon exchange Hamiltonian and $\hat{H}_d$ is the buffer drive Hamiltonian.

Task 1.1: Getting started with `dynamiqs`

Using `dynamiqs`, simulate the time-evolution of this system with the following parameters:

\begin{aligned} g_2 = 1.0\ \epsilon_d = -4\ \kappa_b = 10 \end{aligned}

(For now, we pretend that the parameters are without dimensions)

Use an initial state $|\psi_0\rangle$ in which both the buffer and the memory are in the vacuum. Use a Hilbert-space truncation of $n_a = 20$ and $n_b = 5$ (number of Fock-states in mode a and mode b, respectively) to begin with. You can play with a different Hilbert-space truncation.

Simulate the dynamics for a time $T = 4$.

Plot the wigner function of mode a (as a GIF or as a mosaic plot).

Also plot the expectation value of the number of photons, as well as the photon number parity in the memory mode.

```
g2 = 1.0
ed = -3
kb = 10
na = 20
nb = 5
T = 4

a = dq.destroy(na)
b = dq.destroy(nb)

a_ext = dq.tensor(a, dq.eye(nb))
a_ext2 = dq.tensor(a.powm(2), dq.eye(nb))
b_ext = dq.tensor(dq.eye(na), b)

a_ext_dag = dq.tensor(a.dag(), dq.eye(nb))
a_ext_dag2 = dq.tensor(a.dag().powm(2), dq.eye(nb))
b_ext_dag = dq.tensor(dq.eye(na), b.dag())

H2p = g2 * (a_ext_dag2 @ b_ext) + jnp.conj(g2) * (a_ext2 @ b_ext_dag)
Hd = jnp.conj(ed) * b_ext + ed * b_ext_dag
H = H2p + Hd
```

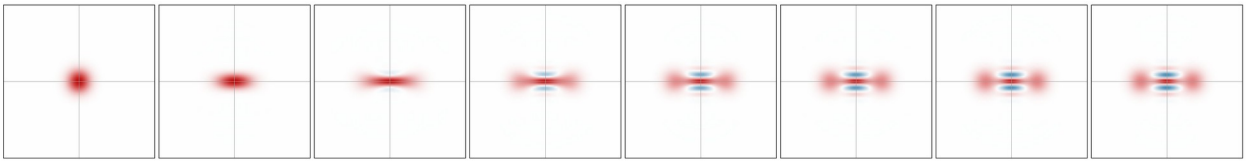
Here, we implemented the Hamiltonian as defined by the question. Here, we used `dq.tensor` with the identity matrix to make the dimensions of $\hat{a}$ and $\hat{b}$ match

```
tsave = jnp.linspace(0, T, 50)

rho = dq.tensor(dq.fock_dm(na, 0), dq.fock_dm(nb, 0))

tsave = jnp.linspace(0, T, 50)
loss_op = [jnp.sqrt(kb) * b_ext]

res1 = dq.mesolve(H, loss_op, rho, tsave)
```

```
 |          |   0.0% ◆ elapsed 0.00ms ◆ remaining ?
```

```
res_a = dq.ptrace(res1.states, 0, (na, nb))
dq.plot.wigner_mosaic(res_a, cross = True)
```

```
|██████████| 100.0% ◆ elapsed 913.08ms ◆ remaining 0.00ms
```

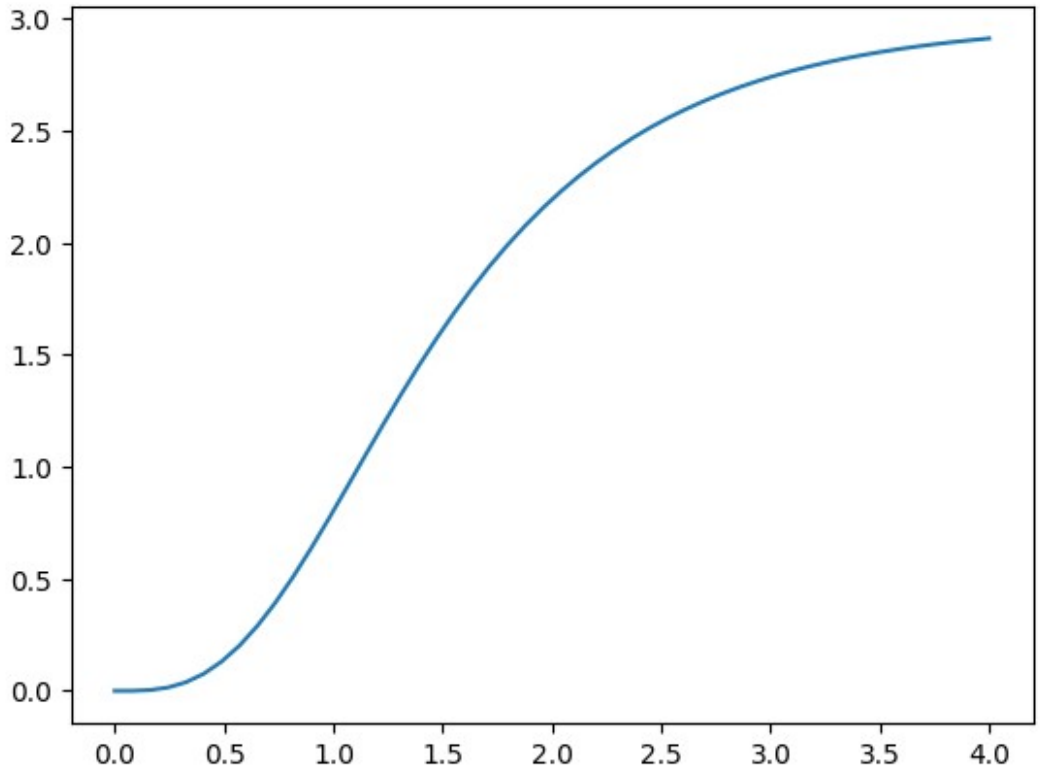


Here we did a partial trace to separate the activities of mode a from the combined activites of mode a and mode b.

```
n_exp = dq.expect(a.dag() @ a, res_a)
plt.plot(tsave, n_exp)
```

```
/usr/local/lib/python3.11/dist-packages/matplotlib/cbook.py:1709:
ComplexWarning: Casting complex values to real discards the imaginary
part
  return math.isfinite(val)
/usr/local/lib/python3.11/dist-packages/matplotlib/cbook.py:1345:
ComplexWarning: Casting complex values to real discards the imaginary
part
  return np.asarray(x, float)
```

```
[<matplotlib.lines.Line2D at 0x7f93112e96d0>]
```



In the above, we applied the photon number operator:

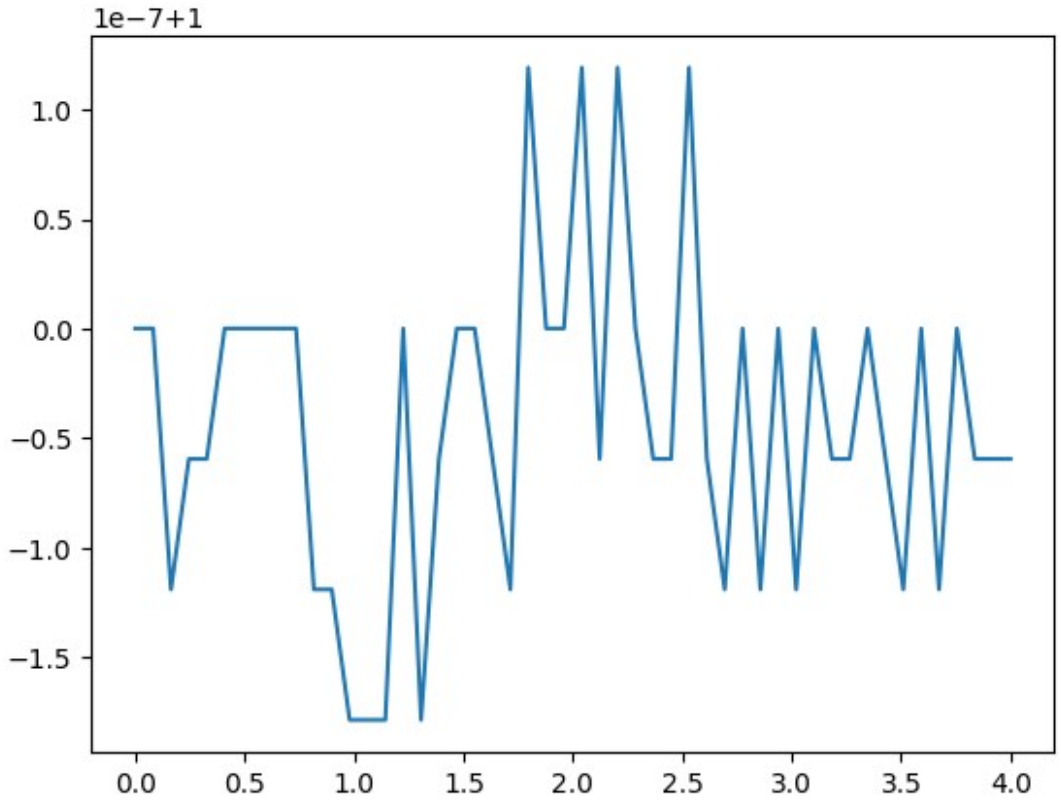
$$\hat{n} = \hat{a}^{\dagger}\hat{a}$$

on the resultant states, to obtain the expectation value of the number of photons throughout the process.

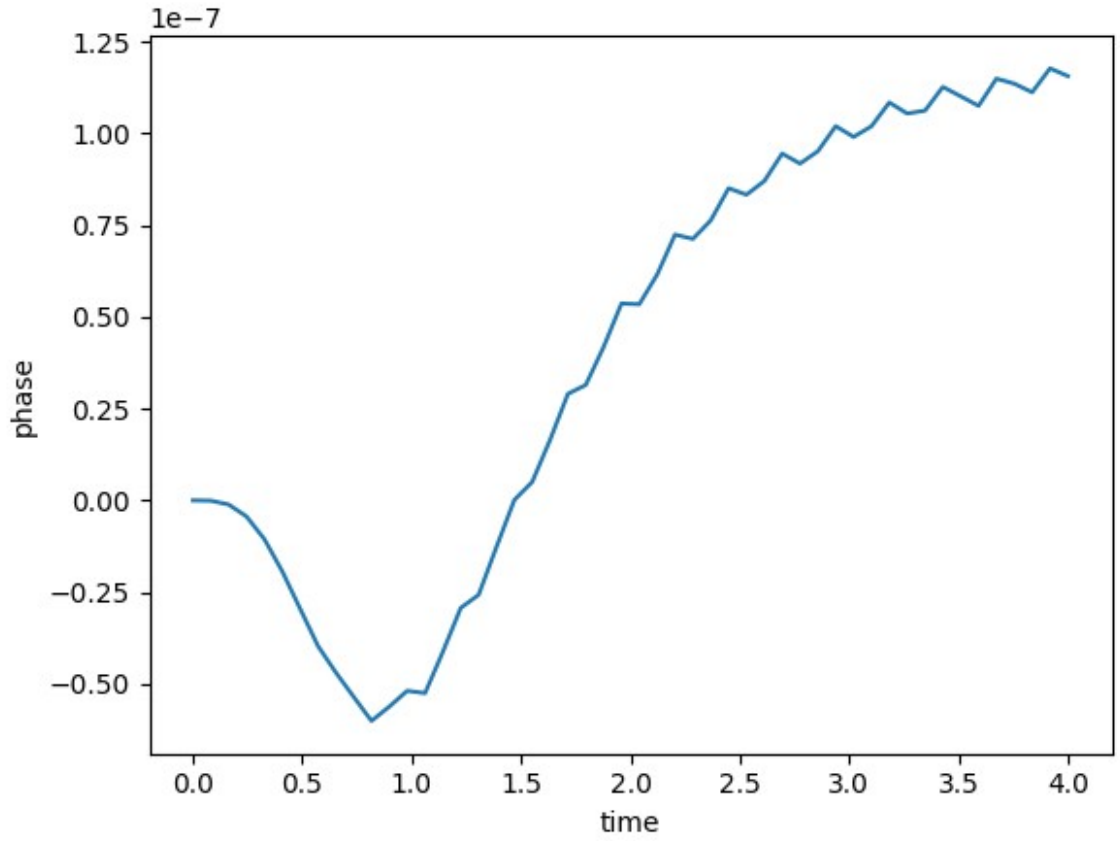
It makes sense that the number increases, as the systems reaches higher energy, it gains more photons.

```
parity_exp = dq.expect(dq.expm(1j * jnp.pi * (a.dag() @ a)), res_a)
plt.plot(tsave, jnp.abs(parity_exp))
```

```
/usr/local/lib/python3.11/dist-packages/equinox/_module.py:1096:
UserWarning: A `SparseDIAQArray` has been converted to a `DenseQArray`
while computing its matrix exponential.
  return self.__func__(self.__self__, *args, **kwargs)

[<matplotlib.lines.Line2D at 0x7f92fe56aad0>]
```




```
plt.plot(tsave, jnp.angle(parity_exp))
plt.xlabel("time")
plt.ylabel("phase")

Text(0, 0.5, 'phase')
```



Here we used the photon number parity operator:

$$\hat{P}=e^{i\pi\hat{n}}$$

where $\hat{n}$ is the photon number operator, as defined above.

Task 1.2: Comparison with eliminated buffer mode

Compare your result from Task 1.1 to the system where the buffer mode is adiabatically eliminated, in which the dynamics of the memory mode is given by:

$$\frac{d\hat{\rho}_a}{dt}=\kappa_2 D\left[\hat{a}^2-\alpha^2\right)\left(\hat{\rho}_a\right),$$

with two-photon dissipation rate $\kappa_2=4\left|g_2\right)^2/\kappa_b$ and cat amplitude $\alpha^2=-\epsilon_d/g_2^{\dot{c}}$.

Compute the time-evolution of the fidelity between the time-evolved states computed with the two-mode system from Task 1.1. What do you observe if you lower κ_b?

```
k2 = 4 * jnp.abs(g2)**2/kb
alpha2 = -ed/jnp.conj(g2)

rhoa = dq.fock(na, 0)
Ha = dq.zeros(na)
```

```
lop = jnp.sqrt(k2) * (a @ a - alpha2 * dq.eye(na))
res2 = dq.mesolve(Ha, [lop], rhoa, tsave)
```

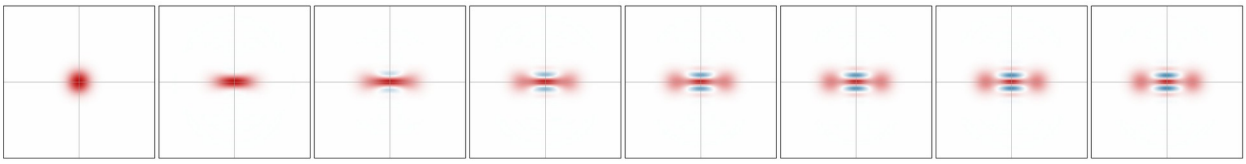
```
|          |   0.2% ◆ elapsed 9.18ms ◆ remaining 0.00ms
```

Here we applied the given Hamiltonian and obtained something that looked a lot like our results from 1.1.

This makes sense as 1.1 is the adiabatic limit of 1.2

```
dq.plot.wigner_mosaic(res2.states, cross = True)
```

```
|██████████| 100.0% ◆ elapsed 416.77ms ◆ remaining 0.00ms
```



Task 1.3: Performing a Zeno-gate

To fully control a cat qubit, we also need to be able to perform gates.

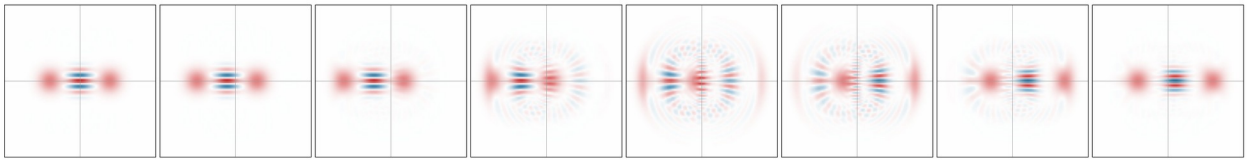
$Z(\theta)$**-rotation**: In addition to the dissipative stabilization mechanism simulated in Task 1.1, find a Hamiltonian that performs a contiuous rotation around the Z-axis of the qubit, also called Zeno gate, (as a reminder: the cat states $¿C_\alpha^\pm⟩$ define the logical X-eigenstates $¿\pm⟩$). This additional Hamiltonian has the form: $\hat{H}_Z = \epsilon_Z^{¿}\hat{O} + \epsilon_Z \hat{O}^\dagger$, where $\hat{O}$ is a bosonic operator.

a) Simulate the time-evolution that maps $¿+⟩$ to $¿-⟩$ in a time T_Z, where $2T_Z$ is the time it takes to make a full rotation. The speed of rotation will depend on the strenght of the parameter ϵ_Z in the Hamiltonian H_Z that generates the rotation.

```
tsave2 = jnp.linspace(0, T, 101)
ez = 1
Hz = jnp.conj(ez) * (jnp.exp(1j * 2 * jnp.pi/T) * a) + ez *
(jnp.exp(1j * 2 * jnp.pi/T) * a).dag()
# Hz = jnp.conj(ez) * a_ext + ez * a_ext_dag

alpha = 2

plus = dq.unit(dq.coherent(na, alpha) + dq.coherent(na, -alpha))
minus = dq.unit(dq.coherent(na, alpha) - dq.coherent(na, -alpha))

res3 = dq.mesolve(Hz, [], plus, tsave2)
```

```
/usr/local/lib/python3.11/dist-packages/dynamiqs/integrators/apis/
mesolve.py:138: UserWarning: Argument `jump_ops` is an empty list and
argument `rho0` is a ket, consider using `dq.sesolve()` to solve the
Schrödinger equation.
```

```
  _check_mesolve_args(H, Ls, rho0, exp_ops)
|████████  |  85.3% ◆ elapsed 10.26ms ◆ remaining 0.00ms

dq.plot.wigner_mosaic(res3.states, cross=True)

 |██████████| 100.0% ◆ elapsed 32.64ms ◆ remaining 0.00ms
```



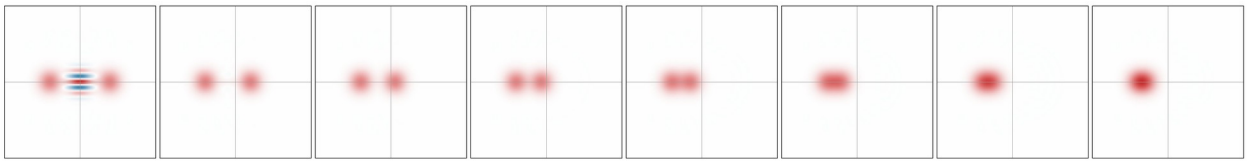
Here, we think that $\hat{O}=e^{i\omega}\hat{a}$ where $\hat{a}$ is the annihilation operator and $\omega=\frac{2\pi}{T}$

b) Optimize parameters: In a real-world scenario, also our memory mode is subject to losses of single photons. Let κ_a be the single-photon loss rate of mode a.

For various values of κ_a from the interval $\kappa_a \in [0.01, 2)$ and for various values of ϵ_Z, plot the parity as a function of time in the presence of $\hat{H}_Z$.

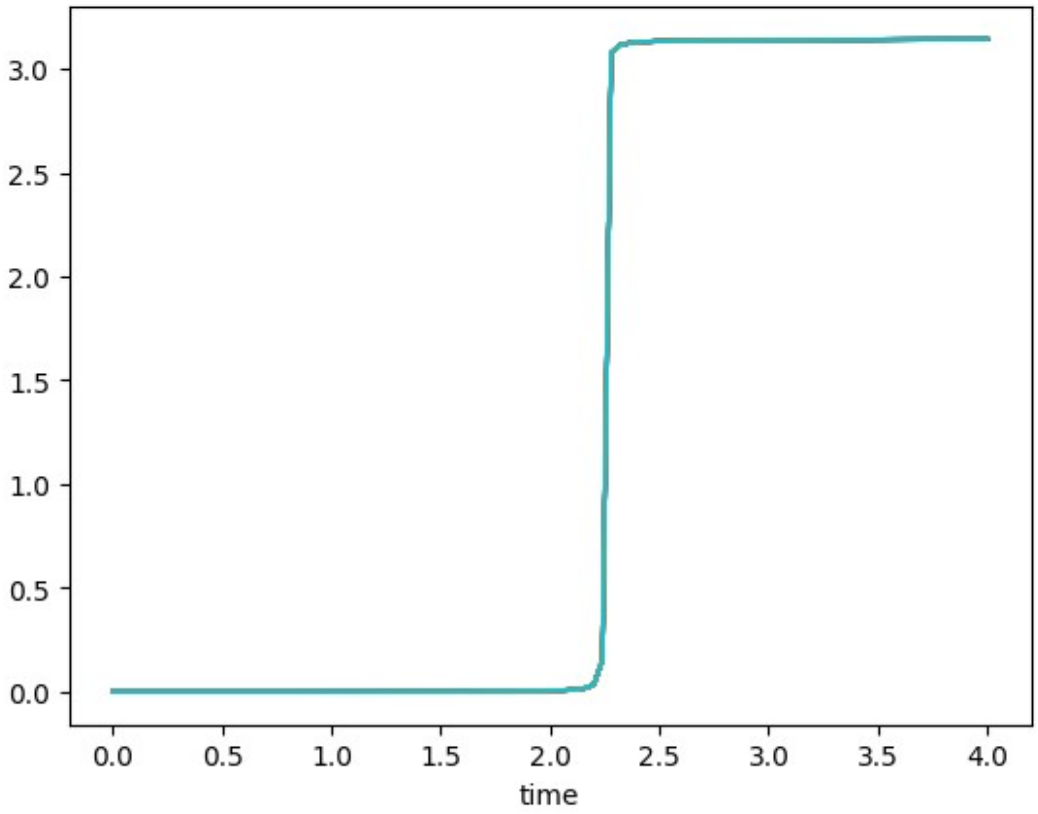
For the parameter range of κ_a above, find the optimal times T_Z for a rotation of $\theta=0 \to \pi$.

```
kappa_a = 1
lop = jnp.sqrt(kappa_a) * a

res4 = dq.mesolve(Hz, [lop], plus, tsave2)

|█████      |  50.2% ◆ elapsed 20.66ms ◆ remaining 0.00ms

dq.plot.wigner_mosaic(res4.states, cross=True)

|██████████| 100.0% ◆ elapsed 71.49ms ◆ remaining 0.00ms
```



```
for kappa_a in jnp.linspace(0.01, 2, 5):
  for ez in range(1,5):
      Hzz = jnp.conj(ez) * (jnp.exp(1j * 2 * jnp.pi/T) * a) + ez *
(jnp.exp(1j * 2 * jnp.pi/T) * a).dag()
      res3b = dq.mesolve(Hz, [], plus, tsave2)
      parity_exp_z = dq.expect(dq.expm(1j * jnp.pi * (a.dag() @ a)),
res3b.states)
      plt.plot(tsave2, jnp.angle(parity_exp_z))
plt.xlabel("time")
plt.ylabel("phase")
```

```
/usr/local/lib/python3.11/dist-packages/dynamiqs/integrators/apis/
mesolve.py:138: UserWarning: Argument `jump_ops` is an empty list and
argument `rho0` is a ket, consider using `dq.sesolve()` to solve the
Schrödinger equation.
  _check_mesolve_args(H, Ls, rho0, exp_ops)
|██████████| 100.0% ◆ elapsed 49.63ms ◆ remaining 0.00ms
/usr/local/lib/python3.11/dist-packages/equinox/_module.py:1096:
UserWarning: A `SparseDIAQArray` has been converted to a `DenseQArray`
while computing its matrix exponential.
  return self.__func__(self.__self__, *args, **kwargs)
|██████████| 100.0% ◆ elapsed 29.05ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 36.42ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 20.91ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 29.72ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 30.60ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 23.81ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 29.15ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 32.74ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 19.68ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 19.27ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 17.29ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 14.29ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 23.83ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 25.81ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 18.15ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 15.85ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 20.94ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 17.52ms ◆ remaining 0.00ms
|██████████| 100.0% ◆ elapsed 20.47ms ◆ remaining 0.00ms

Text(0, 0.5, '')
```



We believe that this spike at t = 2.25 approx indicates the optimal time.

We found the parity as a function of time in the presence of $\widehat{H}_z$ to be the above. It appears that they are all the same across the ranges.

$T_z = 2.2$ is the optimal time based on the graph above

Task 1.4: Optimal control for state-preparation

In Task 1.1, we assumed that the parameter ϵ_d in $\widehat{H}_d$ is constant throughout the time evolution. Now, you will simulate what happens if we let ϵ_d depend on time.

We would like to answer the question: What is the optimal function of time of $\epsilon_d(t)$ to inflate a cat from the vacuum to a target value of $\alpha^2 = 4$ in a given time $T = 3$?

For this, first find and define an adequate loss function. Then, optimize over $\epsilon_d(t)$. For this, you can assume that $\widehat{H}_d$ is piecewise constant. You can play with the number of bins in $\widehat{H}_d$. Plot the optimized value of $\epsilon_d(t)$ as a function of time. What happens if you increase or decrease T?

We expect to have the same approach as p1.1.

Investigation:

Received dimension error mismatch (1, 101, 100, 100) whearas states must have shape (N, n, 1) or (N, n, n); experimented with values of T and the third parameter of tsave and found that the

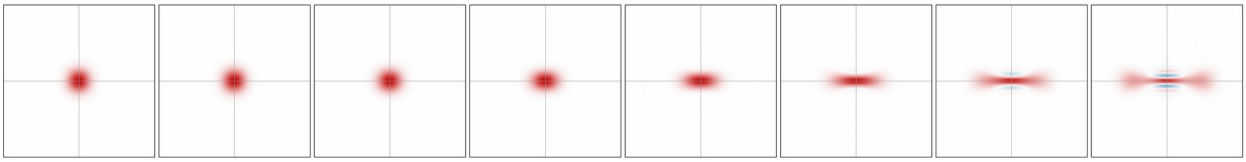
dq.expect value decreased for increasing values of T; this suggested an inverse proportional relationship between expect value and T, whearas we would expect a proportional relationship. We tried linear, linearithmic and exponential functions, with linearithmic having high run-time. Altering power function exponents resulted in finding acheiving an a^2 value near 4.

```
Tg = 3

tsave3 = jnp.linspace(0, Tg, 50)

Hdt = lambda t: H2p + dq.asqarray(jnp.conj(-(t ** 1.712)) * b_ext + (-
(t ** 1.712)) * b_ext_dag)

H4 = dq.timecallable(Hdt)

rho4 = dq.tensor(dq.fock_dm(na, 0), dq.fock_dm(nb, 0))

res4 = dq.mesolve(H4, [loss_op], rho4, tsave3)
```

```
 |          |   0.0% ◆ elapsed 0.00ms ◆ remaining ?
```

```
dq.plot.wigner_mosaic(dq.ptrace(res4.states.squeeze(), 0, (na, nb)),
cross = True)
```

```
|██████████| 100.0% ◆ elapsed 775.32ms ◆ remaining 0.00ms
```



```
jnp.abs(dq.expect(a.dag() @ a, dq.ptrace(res4.final_state.squeeze(),
0, (na, nb))))
```

```
Array(4.0108404, dtype=float32)
```