Task 1.1 - Getting started with dynamiqs

Overview

For this task, we simulate the time-evolution of the whole cat-system using qubits. We simulate the hamiltonian which corresponds to the resonator, and it's associated coupling to the buffer.

Math

Because we are working with operators acting on spaces with different degrees of freedom, we very quickly ran into a dimension error. The initial truncated Hilbert spaces were simply the wrong dimensions. The solution to that was simple enough, we used the tensor product. Any operator which only acted on the resonator in the space, was tensored with I_B so that it could operate in the composite Hilbert space.

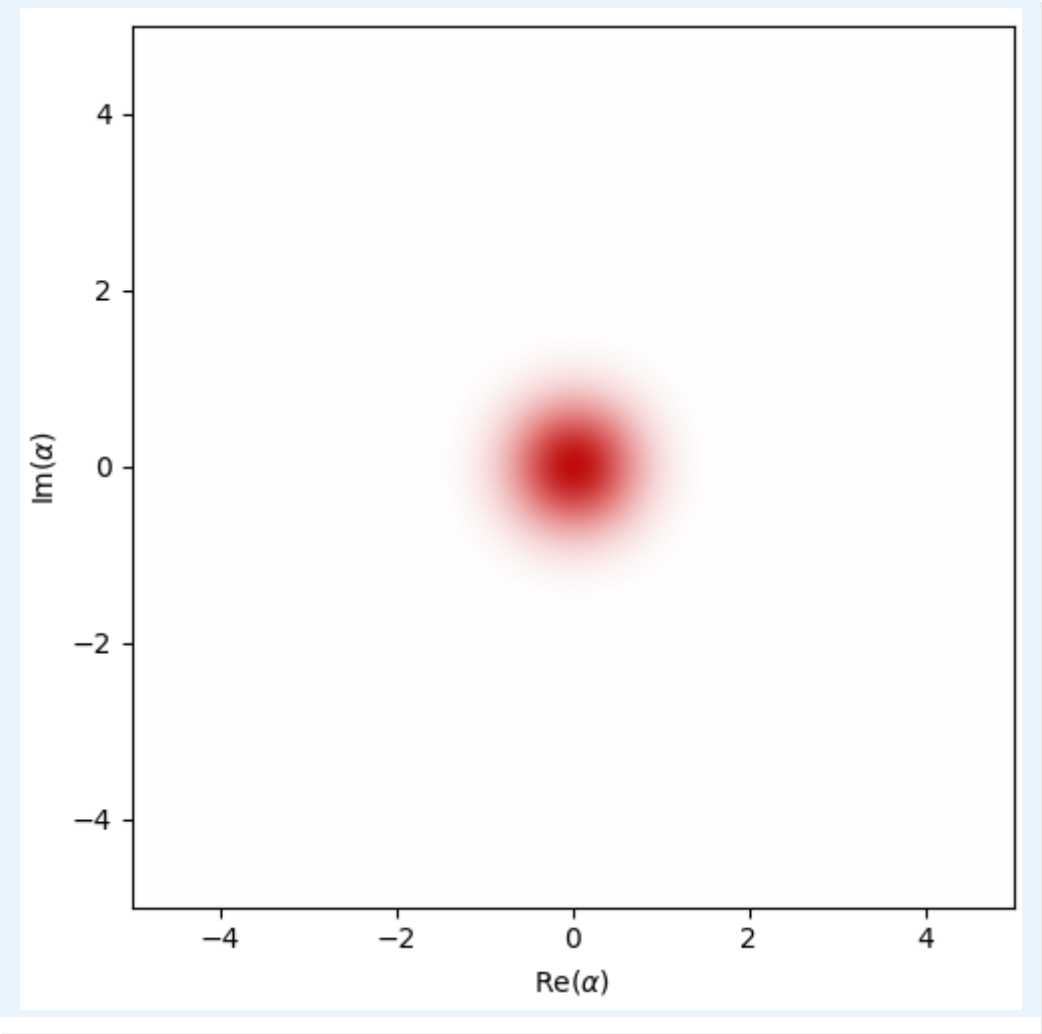
After finding our solution, we took the partial trace of the resonator to reconstruct the individual behavior.

Implementation

We set up the simulation in the following way:

```
Hbuff = g2 * dq.dag(a) @ dq.dag(a) @ b + jnp.conj(g2) * a @ a @ dq.dag(b) + jnp.conj(eps) * b + eps * dq.dag(b)

# Modelling our buffer interaction hamiltonian

jump_ops = [jnp.sqrt(Kb)*b] # jump operators

exp_ops = [dq.dag(a) @ a] # expectation operators

res = dq.mesolve(Hbuff,jump_ops, psi0, t_save, exp_ops=exp_ops) # solve the master equation



trace_a = dq.ptrace(res.states, 0,res.states.dims) # partial trace over b
```

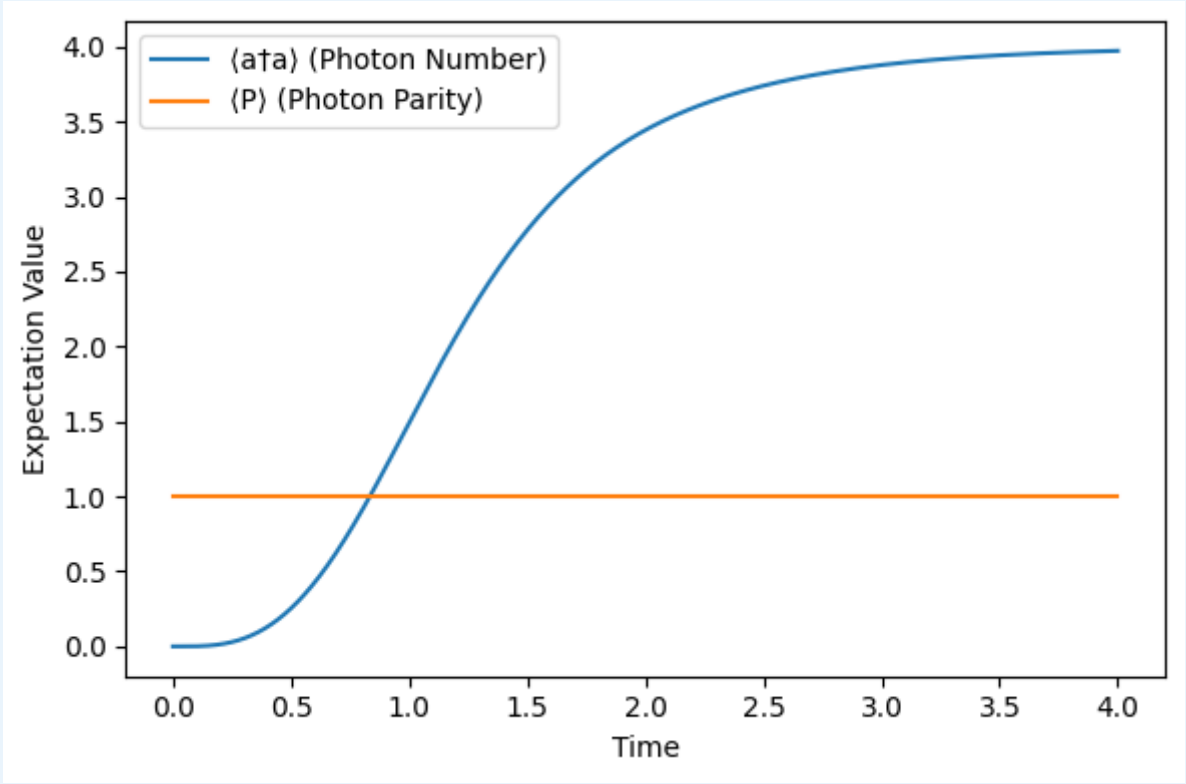
Below you can see the Wigner probability distribution



From here, we knew that to find the number of photons we needed the expectation value of the number operator. Similarly to get the flux we needed the expectation value of the parity operator.

```
numOp = dq.tensor(dq.number(na), dq.eye(nb))
parity_diag_matrix = parity(na)
```

Finally we looped through all the states in the evolution and calculated the parity and number. We expected the parity to be constant (as it implied the buffer was working), and the photon number to increase.



Above is the graph which demonstrates key characteristics of the system evolving over time.

Task 1.2- Comparison with eliminated buffer mode

Overview

We compared the dynamics between the full two-mode system (memory + buffer) to a simplified effective model where the buffer mode has been eliminated adiabatically. The key analysis tool was examining how the fidelity between these two approaches evolved over time.

Math

The full system evolves according to the master equation:

$$\frac{d\hat{\rho}}{dt} = -i\left[\hat{H}, \hat{\rho}\right] + \kappa_b \mathcal{D}(\hat{b})[\hat{\rho}]$$

With Hamiltonian:

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

After adiabatic elimination of the buffer mode, the system reduces to:

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

Where:

- $\kappa_2 = 4|g_2|^2/\kappa_b$ is the two-photon dissipation rate
- $\alpha^2 = -\epsilon_d/g_2^*$ defines the cat amplitude
- We can see that our jump operator is of the form

$$\hat{L} = \sqrt{\kappa_2}\left[\hat{a}^2 - \alpha^2\right]$$

  based on this link.

Implementation

1. Reduced system with eliminated buffer mode
2. Used DynamiQs fidelity function to compare states between 1.1 and 1.2
3. Created visualizations:
   - Generated animations to show temporal evolution
   - Plotted fidelity vs time for different κ_b values
4. Key findings:
   - Identified decay in fidelity between models over time
   - Demonstrated the lack of sensitivity to κ_b parameter

For details on parameter settings and specific visualization code, consult the associated Jupyter notebook.

Results

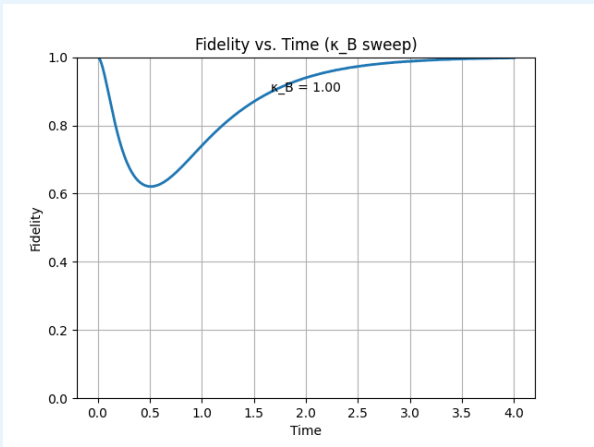


Figure of our fidelity over time, with changing k_B

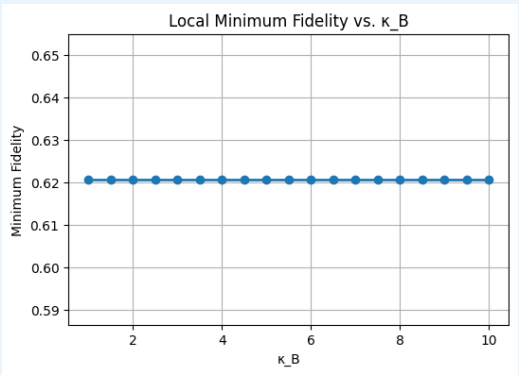


Figure of the change in the local minimum

Task 1.3 - Zeno Gate Implementation

Overview

For this challenge, we were tasked with finding the Z rotation gate for the cat state. In other words, we are to find the state that takes the all evens (Cat plus state) to the all odds (Cat minus state) and vice versa. Due to the form of the Hamiltonian as well as the system we are describing, we know that parity will not be preserved. Therefore, we conclude that the Hamiltonian describing the Z rotation is the sum of the creation and annihilation operators (weighted by constant ϵ_Z).

Math

We know that the unitary will take the form of the time evolution operator $U(t) = e^(- i\hat{H}_Z t)$ and that the Hamiltonian takes the form $\epsilon^* \hat{O} + \epsilon \hat{O}^\dagger$, which led us to our conclusion.

Implementation

Because a photon must be dissipated to change parity, we have the jump operation (argument of dissipation function) as:

```
jump_ops = [a_tensored- alpha*dq.tensor(dq.eye(na),
dq.eye(nb))]
```

And in latex as

$$\hat{L} = a - \alpha^* (I_A \otimes I_B)$$

We also have the given Hamiltonian:

```
Hbuff = g2 * dq.dag(a) @ dq.dag(a) @ b  + jnp.conj(g2) * a @ a
@ dq.dag(b) + jnp.conj(eps) * b + eps * dq.dag(b)
```

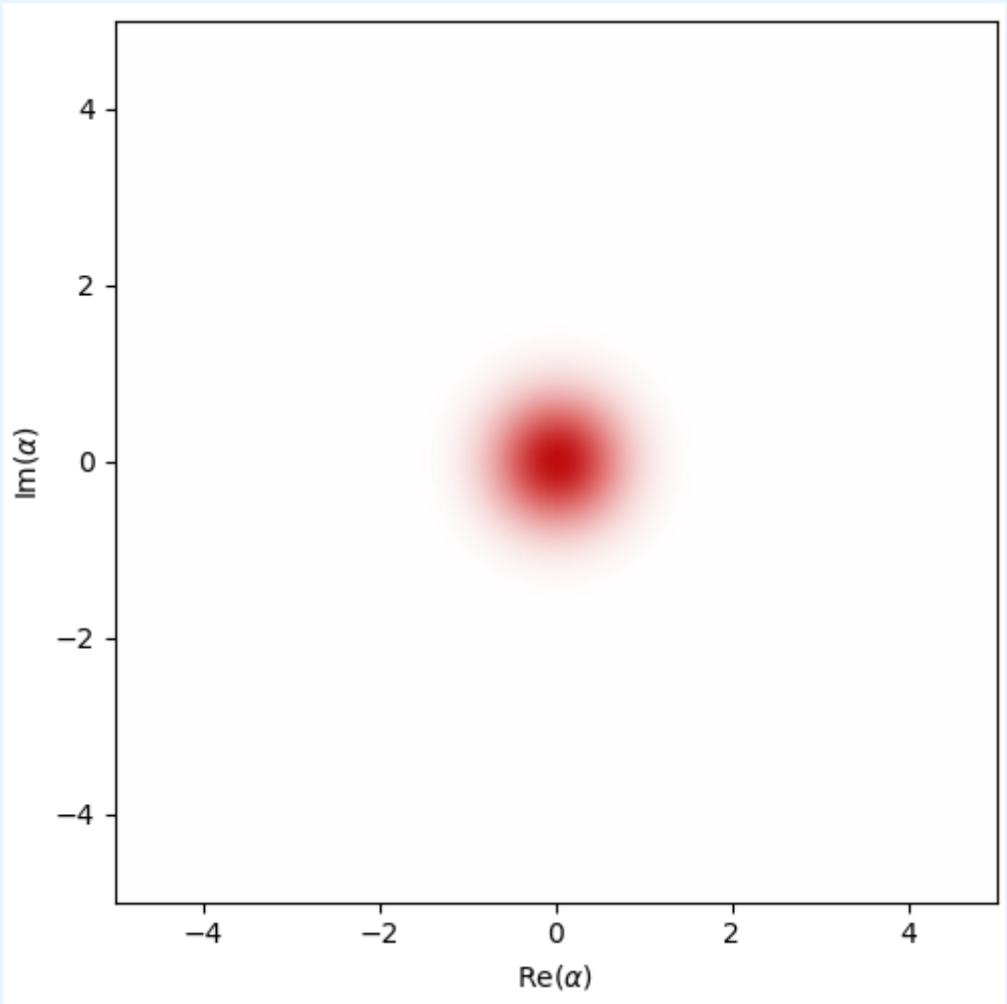
Written in latex as

$$H_{\text{buff}} = g_2 {a^\dagger}^2 b + g_2^* a^2 b^\dagger + \epsilon^* b + \epsilon b^\dagger$$

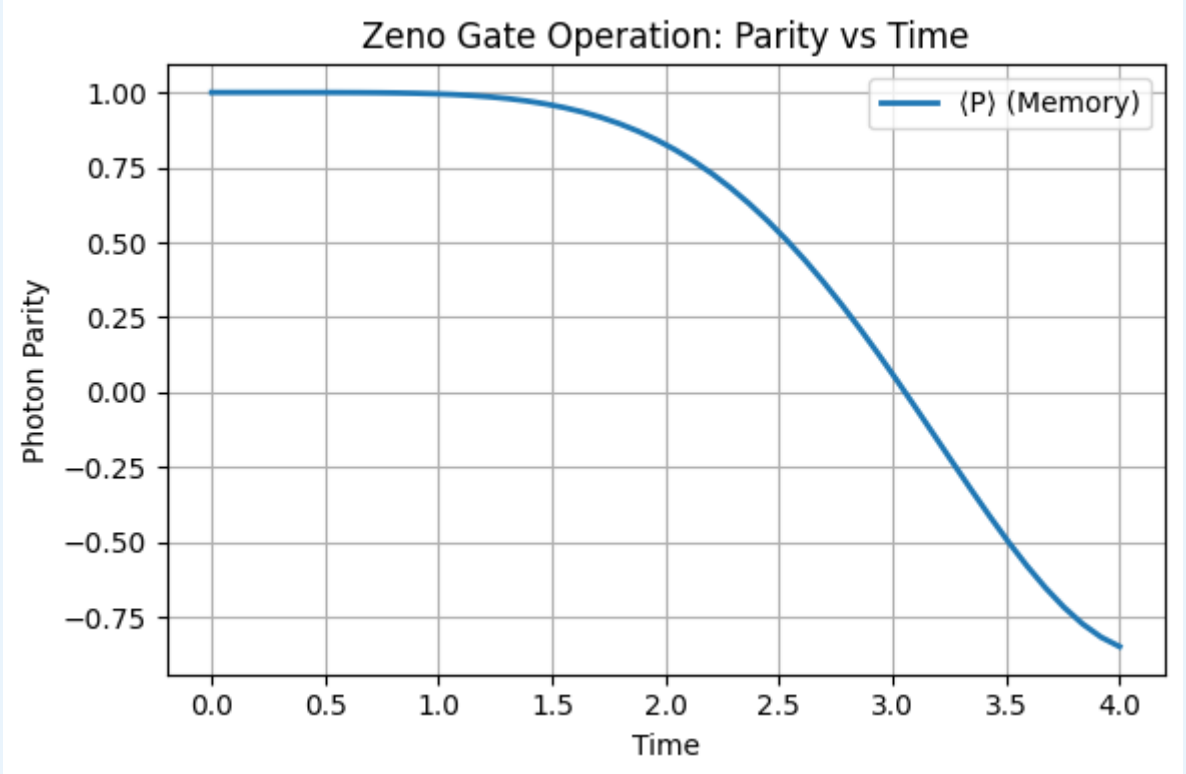
and the Zeno Hamiltonian:

```
H_zeno =
epsilon_z.conjugate()*a_tensored@a_tensored+epsilon_z*adag_tens
```

$$H_{\text{Zeno}} = \epsilon^* a^2 + \epsilon^* {a^\dagger}^2$$



The above figure showcases our simulated evolving wigner distribution



This above figure plots the Pairity of the system as a function of time while evolving in the hamiltonian

Task 1.4 - Optimal Control for State-Preparation

Overview

We created a gradient descent to model the optimization of the drive amplitude as a function of time in order to inflate the vacuum to the cat state as accurately as possible given the time frame and desired mean number of photons.

Math

The total Hamiltonian is described by the two-photon Hamiltonian and a driving Hamiltonian. The amplitude of the driving now varies as a function of time, $\epsilon(t)$, so the total Hamiltonian is now described by

$$H(t) = H^{2ph} + H^d(t)$$

, where

$$H^d(t) = \epsilon_d^*(t)a + \epsilon_d(t)a^\dagger$$

.

The target state is a cat state constructed from two coherent states,

$$|\psi_{cat}\rangle = \frac{|\alpha\rangle + |-\alpha\rangle}{\sqrt{2(1+\exp(-2|\alpha|^2))}}$$

, with the target amplitude chosen such that the average photon number is $\langle a^\dagger a\rangle = |\alpha|^2 = 4$.

The loss function is defined based on the seviatation of the expected photon number of the final state from the target value:

$$L(\epsilon_d(t)) = \left(\frac{\langle a^\dagger a\rangle_{t=T} - 4}{4}\right)^2$$

The drive amplitude $\epsilon_d(t) = \epsilon_{params}[k]$, for $t \in \left[\frac{kT}{N_{bins}}, \frac{(k+1)T}{N_{bins}}\right)$

Implementation

How the code works:

Initialization: The simulation sets up a Hilbert space with a cutoff $N_{cutoff} = 20$ and a total simulation time $T = 3$. The initial state is chosen as the vacuum state.

```
# Simulation parameters (use lower resolution for debugging)
N_cutoff = 20  # Hilbert space dimension for the mode
T = 3.0 # Total evolution time
numTimes = 200 # Reduced number of time steps
t_save = jnp.linspace(0, T, numTimes)

```

Defining the Fields: The drive amplitude is implemented as a piecewise constant function, where the vector eps_params holds the amplitude for each time bin.

```
# Define a piecewise constant drive function.
def eps_d(t, eps_params):
    t = jnp.asarray(t)
    N_bins = eps_params.shape[0]
    bin_index = jnp.minimum((t / T * N_bins).astype(jnp.int32),
N_bins - 1)
    return eps_params[bin_index]
```

Loss function: The loss function uses the expectation value of the number operator and compares it to the target value of 4. The loss function simulates the evolution (using dq.mesolve) and computes the loss as the squared difference between the expected photon number and 4, with normalization.

```
def loss(eps_params):
    H_time_current = make_H_time(eps_params)
    result = dq.mesolve(H_time_current,jumpOp, psi0, t_save)
    # result = dq.sesolve(H_time_current, psi0, t_save)

    final_state = result.states[-1]
    # fid = dq.fidelity(final_state, cat_target)
    a = dq.destroy(psi0.shape[0])
    adag = dq.dag(a)
    N = adag@a
    expecVal = dq.expect(N, final_state)
    return (jnp.real((expecVal-4)/(4**2))**2)

```

Optimization: The gradients of the loss with respect to the drive amplitudes are computed using JAX.

```
grad_loss = jax.grad(loss)
optimizer = optax.adam(learning_rate=0.09)
opt_state = optimizer.init(eps_params_init)
eps_params = eps_params_init

\# Iterative solver using gradient descent to find the driven amplitude as a function of time.
n_iterations = 10
for i in range(n_iterations):
    grads = grad_loss(eps_params)
    updates, opt_state = optimizer.update(grads, opt_state)
    eps_params = optax.apply_updates(eps_params, updates)
    print(f"Iteration {i}: Loss = {loss(eps_params):.6f}")


Plotting: The optimized drive amplitudes are plotted as a function of time.
\# Plot the driven amplitude as a function of time.
times = jnp.linspace(0, T, N_bins)
plt.plot(times, jnp.real(eps_params), label="Real part")
#plt.plot(times, jnp.imag(eps_params), label="Imaginary part")
plt.xlabel("Time")
plt.ylabel("Drive amplitude")
plt.legend()
plt.show()
```