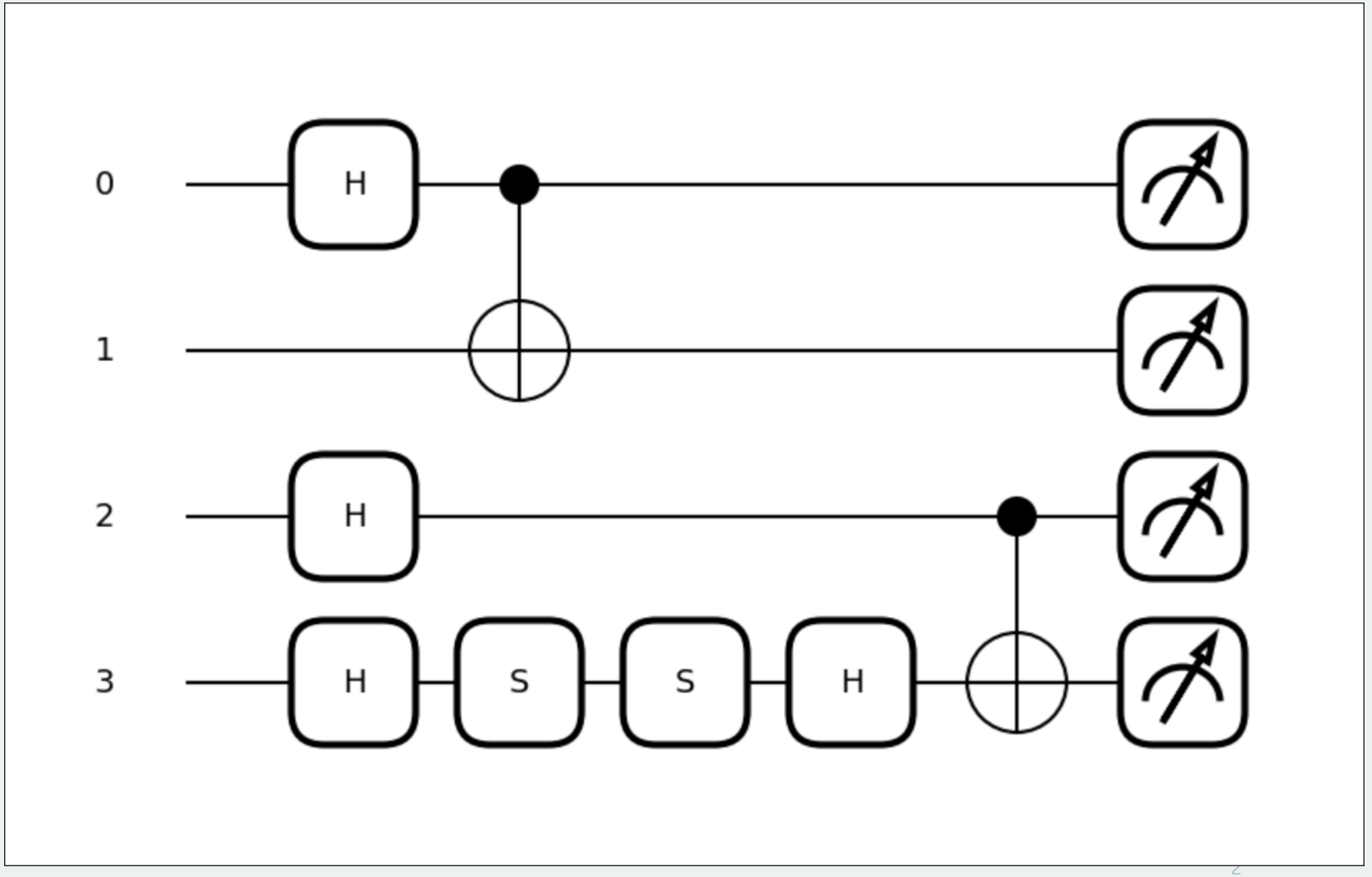
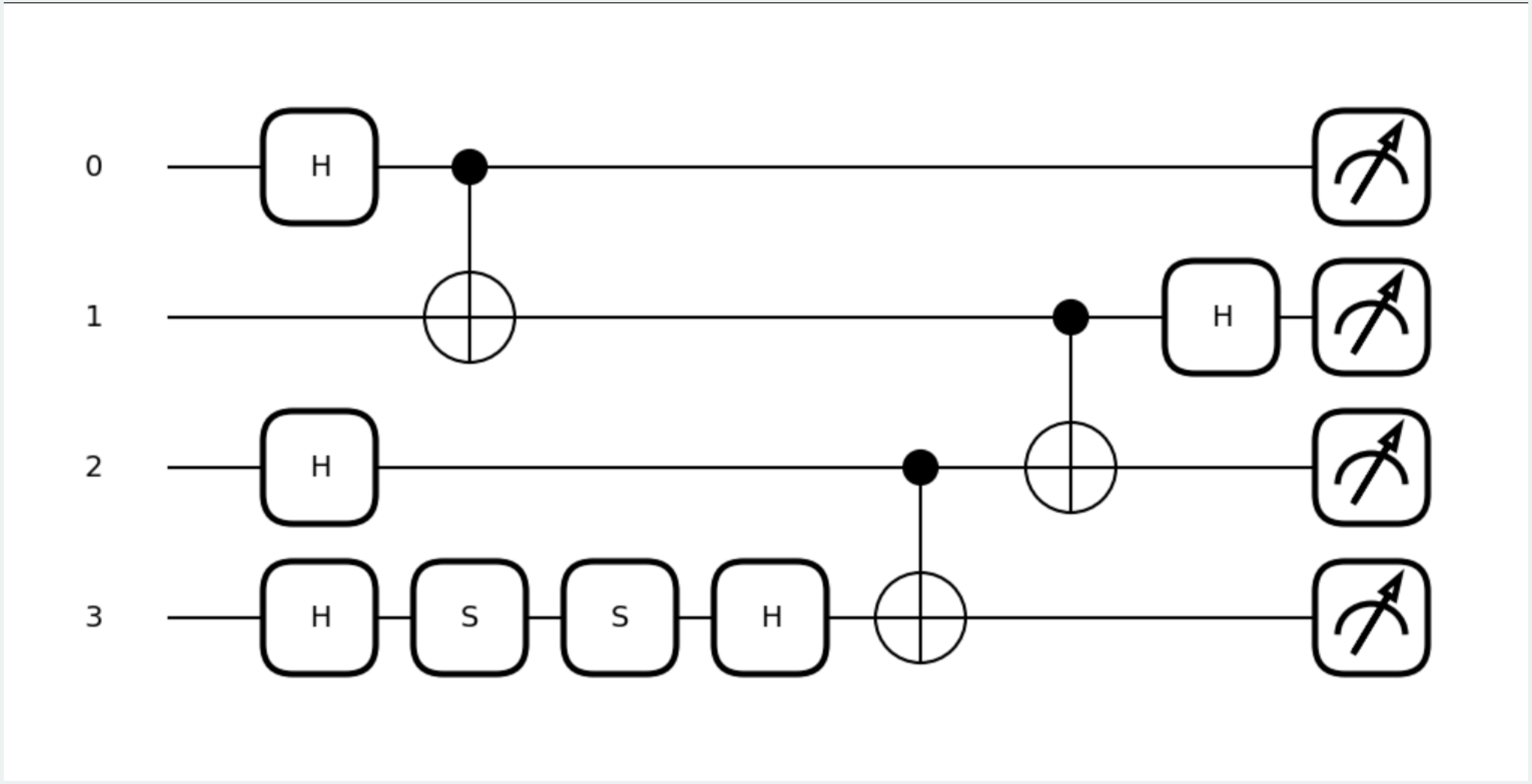
A+I

김현수 권기훈 손원희

PART 1 – a)



PART 1 – b)



PART 1 – c)

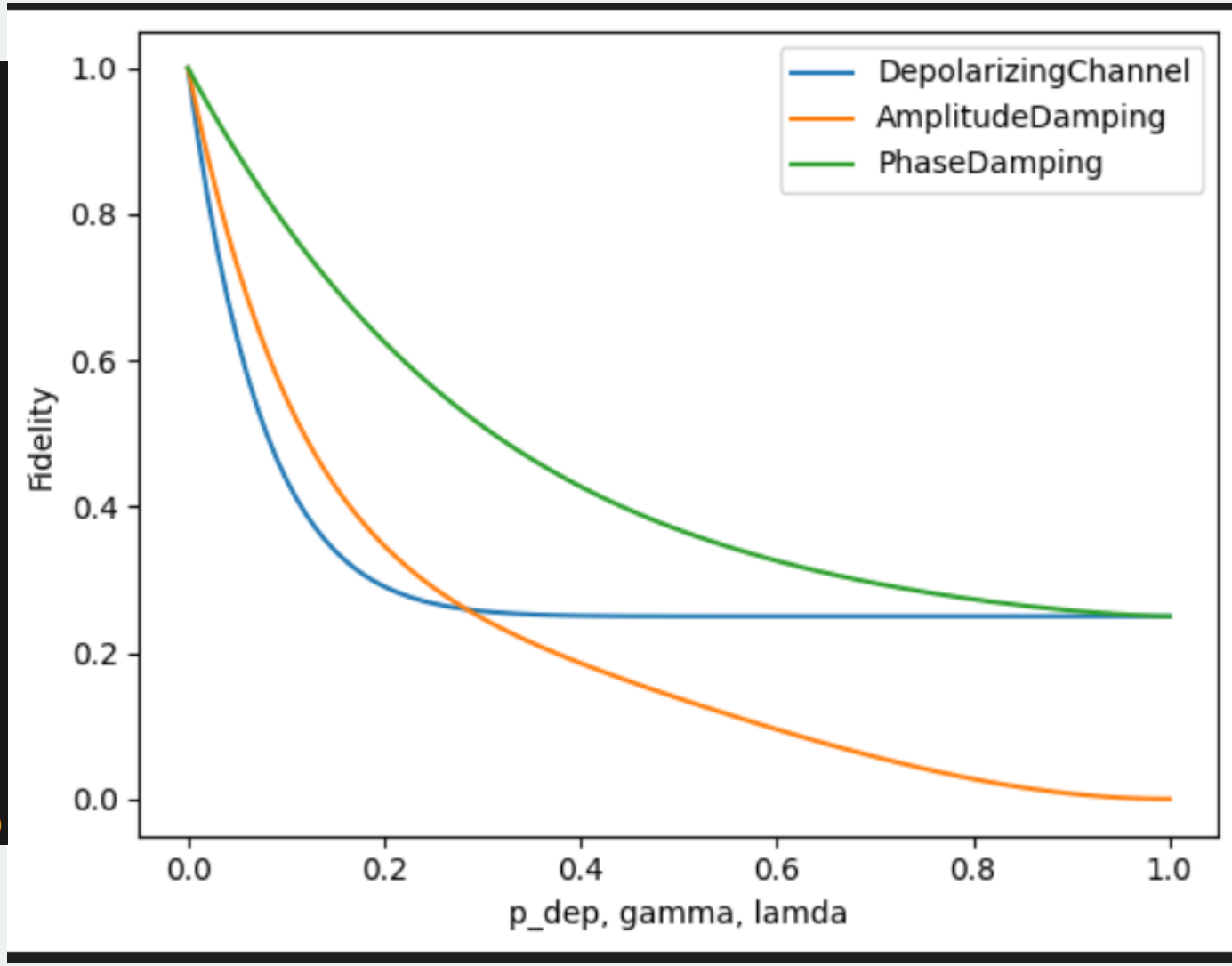
```
amp = noiseless_circuit().reshape(2,2,2,2)
AD_block = amp[:, 0, 0, :]
p00 = np.sum(np.abs(AD_block)**2)
phi_AD = (AD_block/np.sqrt(p00)).reshape(4)

print("p(BC=00) =", float(p00))
print("phi_AD = ", phi_AD)
```

✓ 0.0s

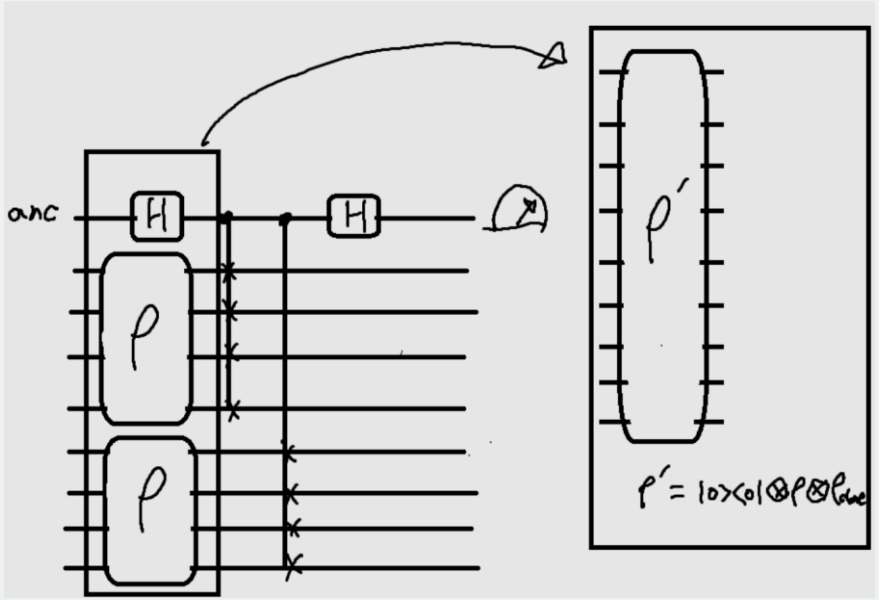
```
p(BC=00) = 0.2499999999999998
phi_AD =  [0.         +0.j 0.70710678+0.j 0.70710678+0.j 0.         +0.j]
```

PART 1 – d)

```
_idx = [i/100 for i in range(101)]
F_p = []
F_g = []
F_l = []
for k in _idx:
    rho4 = noisy_circuit(p_dep = k, gamma = 0, lam = 0)
    p00, rho_AD = ad_rho_given_BC00(rho4)
    F_p.append(fidelity(rho_AD, phi_AD))

    rho4 = noisy_circuit(p_dep = 0, gamma = k, lam = 0)
    p00, rho_AD = ad_rho_given_BC00(rho4)
    F_g.append(fidelity(rho_AD, phi_AD))

    rho4 = noisy_circuit(p_dep = 0, gamma = 0, lam = k)
    p00, rho_AD = ad_rho_given_BC00(rho4)
    F_l.append(fidelity(rho_AD, phi_AD))

plt.plot(_idx, F_p)
plt.plot(_idx, F_g)
plt.plot(_idx, F_l)
plt.xlabel("p_dep, gamma, lamda")
plt.ylabel("Fidelity")
plt.legend(["DepolarizingChannel", "AmplitudeDamping", "PhaseDamping"])
```



PART 2 – a)




```python
def embed_rho(rho, total_wires=9, target_wires=4):
    """rho(16x16)를 9qubit 공간(512x512)에 임베딩"""
    dim_total = 2**total_wires
    dim_rho   = 2**target_wires

    big_rho = np.zeros((dim_total, dim_total), dtype=complex)

    # ancilla=|0>, 나머지=|0>으로 두고 앞쪽 4 qubit에만 rho 배치
    for i in range(dim_rho):
        for j in range(dim_rho):
            big_rho[i, j] = rho[i, j]

    return big_rho
```

PART 2 – a)

```python
def rho_k(rho, k):
    n = int(np.log2(rho.shape[0]))

    rho = rho.astype(np.complex128)
    rho2 = rho.astype(np.complex128)

    for i in range(k-1):
        rho2 = np.kron(rho2, rho).astype(np.complex128)

    proj0 = np.array([[1,0],[0,0]], dtype=np.complex128)
    rho_big = np.kron(rho2, proj0).astype(np.complex128)


    @qml.qnode(qml.device("default.mixed", wires=n*k+1))
    def _2_a(rho_big):

        qml.QubitDensityMatrix(rho_big, wires=range(n*k+1))

        anc = n*k

        qml.H(anc)
        for j in range(k-1):
            for i in range(n):
                qml.CSWAP(wires=[anc, i+j*n, i+(j+1)*n])
        qml.H(anc)

        return qml.expval(qml.PauliZ(anc))
```

```python
ans = _2_a(rho_big)
print(f"Tr(rho^{k}) Quntum:{ans}")

rho_k = rho
for i in range(k-1):
    rho_k=rho_k @ rho
print(f"Tr(rho^{k}) Direct:{np.trace(rho_k).real}")

return ans
```

PART 2 – a)

```
n=3

A = np.random.randn(2**n,2**n) + 1j*np.random.randn(2**n,2**n)
rho = A @ A.conj().T
rho = rho / np.trace(rho)

rho_k(rho, 2)
```

✓ 0.0s

```
Tr(rho^2) Quntum:0.233176830986648
Tr(rho^2) Direct:0.23317683098664815

np.float64(0.233176830986648)
```

```
n=3

A = np.random.randn(2**n,2**n) + 1j*np.random.randn(2**n,2**n)
rho = A @ A.conj().T
rho = rho / np.trace(rho)

rho_k(rho, 3)
```

✓ 0.8s

```
Tr(rho^3) Quntum:0.06036943319538257
Tr(rho^3) Direct:0.06036943319538196

np.float64(0.06036943319538257)
```

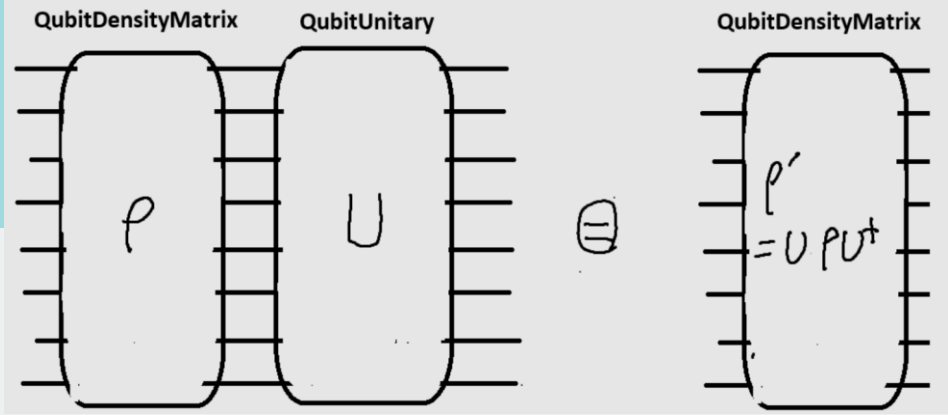
```
n=3

A = np.random.randn(2**n,2**n) + 1j*np.random.randn(2**n,2**n)
rho = A @ A.conj().T
rho = rho / np.trace(rho)

rho_k(rho, 4)
```

✓ 52.2s

```
Tr(rho^4) Quntum:0.018669749695259252
Tr(rho^4) Direct:0.018669749695257216

np.float64(0.018669749695259252)
```

PART 2 – b)

$$\hat{\rho} = (d+1)U^\dagger|b\rangle\langle b|U - I$$




```python
def haar_unitary(d, rng):
    """Haar 무작위 유니터리 생성 (QR 방식)."""
    Z = (rng.normal(size=(d, d)) + 1j * rng.normal(size=(d, d))) / np.sqrt(2.0)
    Q, R = np.linalg.qr(Z)
    # R의 대각 성분 위상의 보정으로 Haar 분포 확보
    D = np.diag(np.exp(1j * np.angle(np.diag(R))))
    return Q @ D

def classical_shadows_purity(rho, M=200, seed=0, max_pairs=None):

    rho = np.asarray(rho, dtype=np.complex128)
    d = rho.shape[0]
    assert rho.shape == (d, d)
    # d = 2^n 체크(옵션)
    n_log2 = np.log2(d)
    if abs(n_log2 - round(n_log2)) > 1e-9:
        raise ValueError("rho dimension must be a power of 2.")

    @qml.qnode(qml.device("default.mixed", wires=int(n_log2)))
    def _2_b(U, rho):
        qml.QubitDensityMatrix(rho, wires=range(int(n_log2)))
        qml.QubitUnitary(U, wires=range(int(n_log2)))

        return qml.density_matrix(wires=range(int(n_log2)))

    d_id = np.eye(d, dtype=np.complex128)

    rng = np.random.default_rng(seed)
```

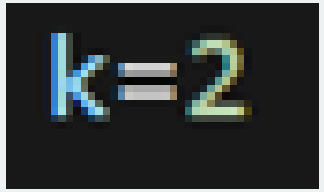
```python
# 1) M개의 shadow 샘플 생성
shadows = []
for _ in range(M):
    U = haar_unitary(d, rng)

    # sigma = U @ rho @ U.conj().T                # 회전된 상태
    temp = _2_b(U, rho)
    probs = np.real(np.diag(temp))              # 컴퓨테이셔널 기저 확률
    # 수치오차 보정
    probs = np.clip(probs, 0.0, 1.0)
    probs = probs / probs.sum()

    b = rng.choice(d, p=probs)                  # 측정 결과
    e_b = np.zeros((d, 1), dtype=np.complex128)
    e_b[b, 0] = 1.0
    proj_b = e_b @ e_b.conj().T                 # |b><b|

    # temp = U.conj().T @ proj_b @ U              # U^\dagger |b><b| U
    temp = _2_b(U.conj().T, proj_b)
    rho_hat = (d + 1) * temp - d_id             # shadow estimator (global 2-design)
    shadows.append(rho_hat)
```

PART 2 – b)

$$\mathrm{Tr}(\rho^k) = \mathbb{E}\left[\mathrm{Tr}(\hat{\rho}_1 \hat{\rho}_2 \cdots \hat{\rho}_k)\right]$$

```
all_tuples = list(itertools.combinations(range(M), k))
if (max_tuples is None) or (max_tuples >= len(all_tuples)):
    idx_tuples = all_tuples
else:
    chosen = rng.choice(len(all_tuples), size=max_tuples, replace=False)
    idx_tuples = [all_tuples[t] for t in chosen]

acc = 0.0
for tup in idx_tuples:
    mat = shadows[tup[0]]
    for idx in tup[1:]:
        mat = mat @ shadows[idx]
    acc += np.trace(mat).real

return acc / len(idx_tuples)
```

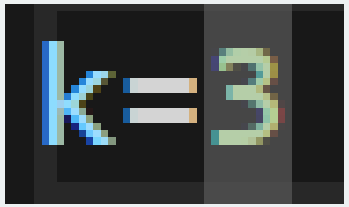
PART 2 – b)




```
d = 4
A = np.random.randn(d, d) + 1j*np.random.randn(d, d)
rho = A @ A.conj().T
rho = rho / np.trace(rho)
k=2
est = classical_shadows_purity(rho, k=k, M=500, seed=0)   # 샘플 수 M 늘릴수록 분산 ↓
exact = np.trace(rho @ rho).real

print(f"Classical-shadows estimate Tr(rho^{k}):{est}")
print(f"Exact Tr(rho^{k}):                    {exact}")
```

✓ 4.6s

```
Classical-shadows estimate Tr(rho^2):0.47812723490611
Exact Tr(rho^2):                     0.4907119181753472
```



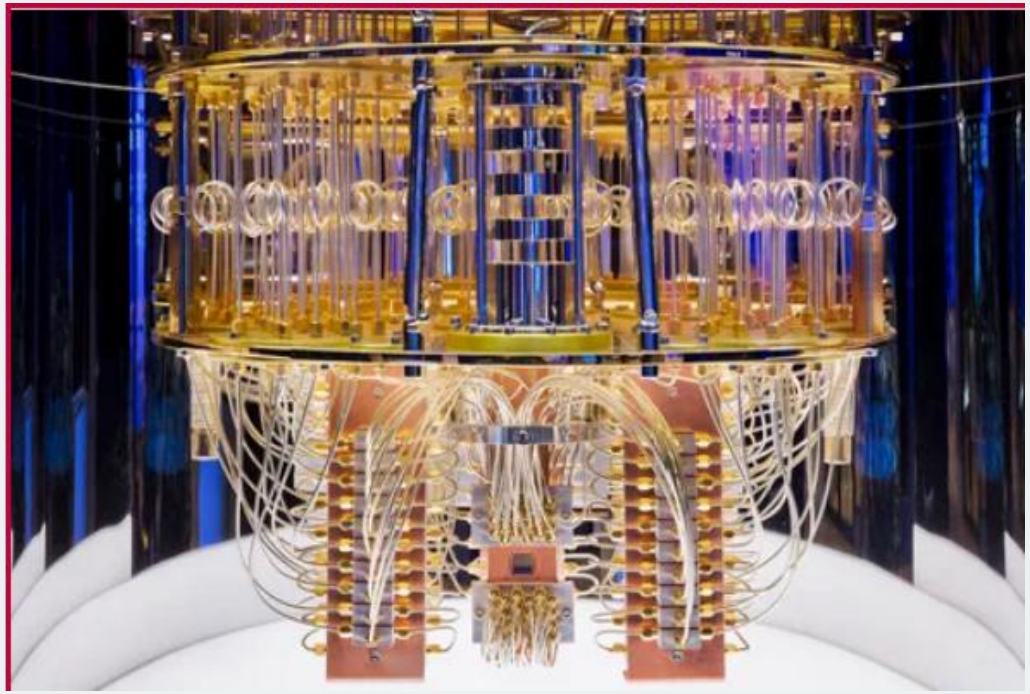

```
d = 2
A = np.random.randn(d, d) + 1j*np.random.randn(d, d)
rho = A @ A.conj().T
rho = rho / np.trace(rho)
k=3
est = classical_shadows_purity(rho, k=k, M=400, seed=0)   # 샘플 수 M 늘릴수록 분산 ↓
exact = np.trace(rho @ rho).real

print(f"Classical-shadows estimate Tr(rho^{k}):{est}")
print(f"Exact Tr(rho^{k}):                    {exact}")
```

✓ 3m 5.3s

```
Classical-shadows estimate Tr(rho^3):0.8479212081863114
Exact Tr(rho^3):                     0.8767157354350924
```

PART 2 – c)

$$\mathrm{Tr}(\rho^k)$$



Entropy Estimation

Purity Testing

Quantum Hypothesis Testing

Entanglement Detection

Quantum Simulation & Many-body physics

PART 2 – d)

$$S(\rho) = -\mathrm{Tr}(\rho \log \rho)$$

$$S(\rho) = \inf_{H=H^\dagger} \left\{ -\mathrm{Tr}(\rho H) + \log \mathrm{Tr}\left(e^H\right) \right\}$$

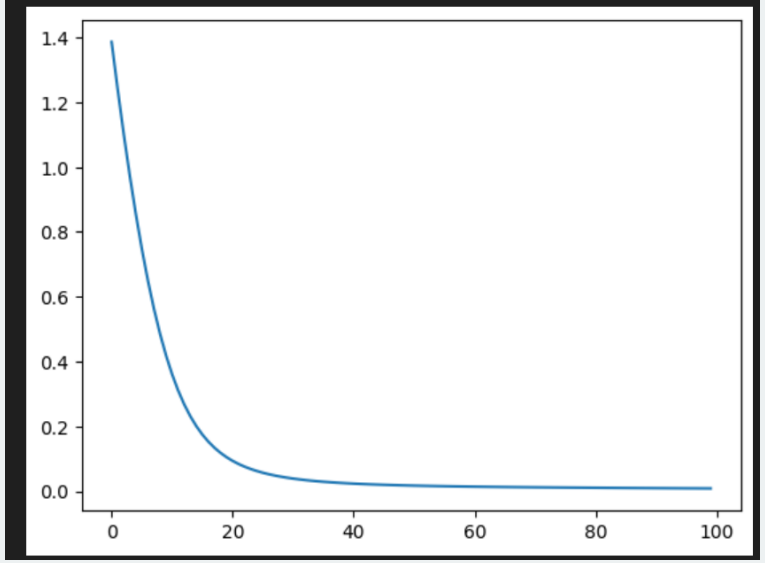
```python
@staticmethod
def logtr_expm_from_eig(T: torch.Tensor) -> torch.Tensor:
    evals = torch.linalg.eigvalsh(T)
    m = torch.max(evals)
    return m + torch.log(torch.sum(torch.exp(evals - m)))

def loss(self):
    """QMINE: L(θ) = -⟨T⟩_ρ + log Tr e^T"""
    first  = - torch.dot(self.theta, self.exps)     # -Σ θ_i <P_i>_ρ
    second = self.logtr_expm_from_eig(self.build_T())
    return first + second, first, second
```

PART 2 – d)

$$S(\rho) = -\mathrm{Tr}(\rho \log \rho)$$

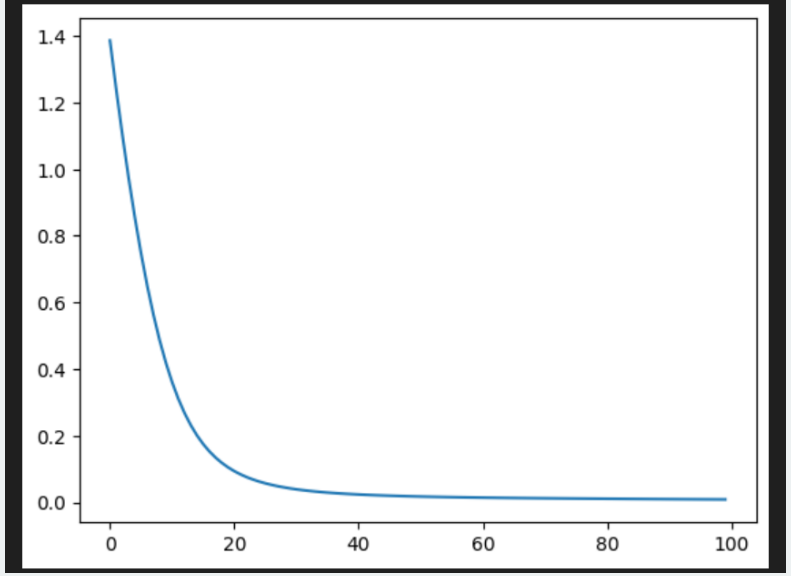
```
# ## pure state
rho1 = one_qubit_mixed_state(p=0)
# rho1 = one_qubit_mixed_state(p=1)
rho1 = torch.tensor(np.kron(rho1,rho1))
```

```
tensor([[0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j]], dtype=torch.complex128)
S_est ≈ 0.008484,  S_true = -0.000000
Train log (iter, loss, -<T>, logTr exp T):
(50, 0.017692653835951866, -3.8432690096693767, 3.8609616635053285)
(100, 0.008571077963321905, -4.390264089953039, 4.398835167916361)

[<matplotlib.lines.Line2D at 0x1cd46d65bb0>]
```




```
# ## pure state
# rho1 = one_qubit_mixed_state(p=0)
rho1 = one_qubit_mixed_state(p=1)
rho1 = torch.tensor(np.kron(rho1,rho1))
```

```
tensor([[1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j]], dtype=torch.complex128)
S_est ≈ 0.008484,  S_true = -0.000000
Train log (iter, loss, -<T>, logTr exp T):
(50, 0.017692653835951866, -3.8432690096693767, 3.8609616635053285)
(100, 0.008571077963321905, -4.390264089953039, 4.398835167916361)

[<matplotlib.lines.Line2D at 0x1cd4d716f90>]
```

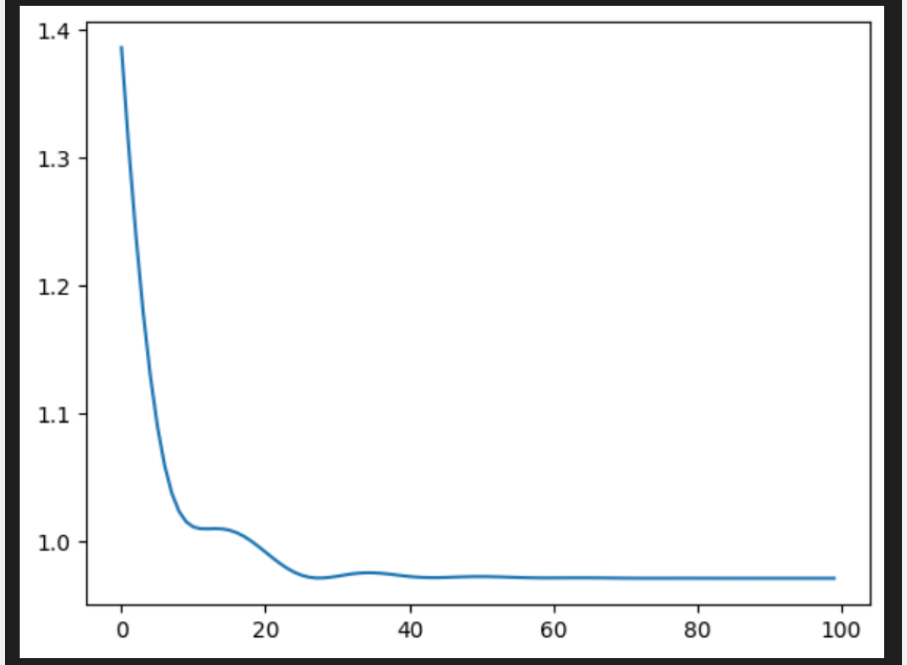


PART 2 – d)

$$S(\rho) = -\mathrm{Tr}(\rho \log \rho)$$

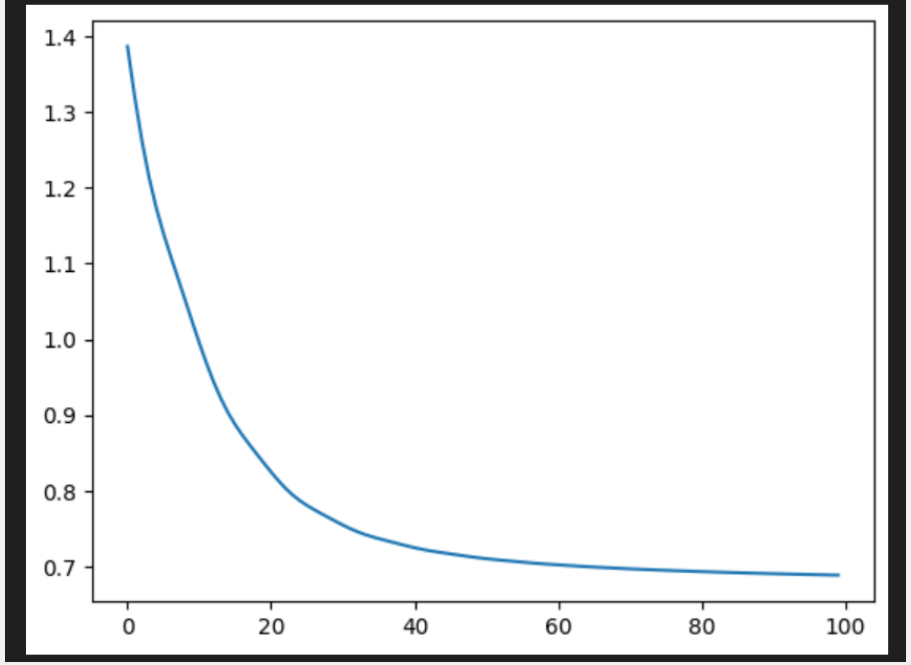
```python
# ## mixed state
rho1 = one_qubit_mixed_state(p=np.random.random())
rho1 = torch.tensor(np.kron(rho1,rho1))
```

```
S_est ≈ 0.970766,  S_true = 0.970762
Train log (iter, loss, -<T>, logTr exp T):
(50, 0.9720861476351244, -0.9663888443447723, 1.9384749919798967)
(100, 0.9707659781210882, -0.9000267023541229, 1.870792680475211)

[<matplotlib.lines.Line2D at 0x1cd4d753260>]
```




```python
## low rank state
ket0 = torch.tensor([[1.0],[0.0],[0.0],[0.0]], dtype=torch.complex128)  # |00>
ket1 = torch.tensor([[0.0],[1.0],[0.0],[0.0]], dtype=torch.complex128)  # |01>
rho01 = ket0 @ ket0.conj().T
rho02 = ket1 @ ket1.conj().T
rho1 = 0.6 * rho01 + 0.4 * rho02
```

```
S_est ≈ 0.688418,  S_true = 0.673012
Train log (iter, loss, -<T>, logTr exp T):
(50, 0.7113903929897789, -1.6537166426596308, 2.3651070356494097)
(100, 0.6886119123121093, -2.1069479771379465, 2.7955598894500557)

[<matplotlib.lines.Line2D at 0x1cd4d7f2b10>]
```



PART 2 – d)

$$S_\alpha(\rho) = \frac{1}{1-\alpha} \log \mathrm{Tr}(\rho^\alpha)$$

$$T(\theta) = \sum_i \theta_i P_i$$

$$\mathcal{L}_\alpha(\theta) = \frac{\alpha}{\alpha - 1} \log \mathrm{Tr}\left[e^{(\alpha-1)T}\rho\right] - \log\left(\tfrac{1}{d}\mathrm{Tr}[e^{\alpha T}]\right)$$

$$= \frac{\alpha}{\alpha - 1} \log \sum_j e^{(\alpha-1)\lambda_j} (\tilde{\rho})_{jj} - \left(\log \sum_j e^{\alpha\lambda_j} - \log d\right)$$

$$S_\alpha(\rho) \approx \log d - \max_\theta \mathcal{L}_\alpha(\theta)$$

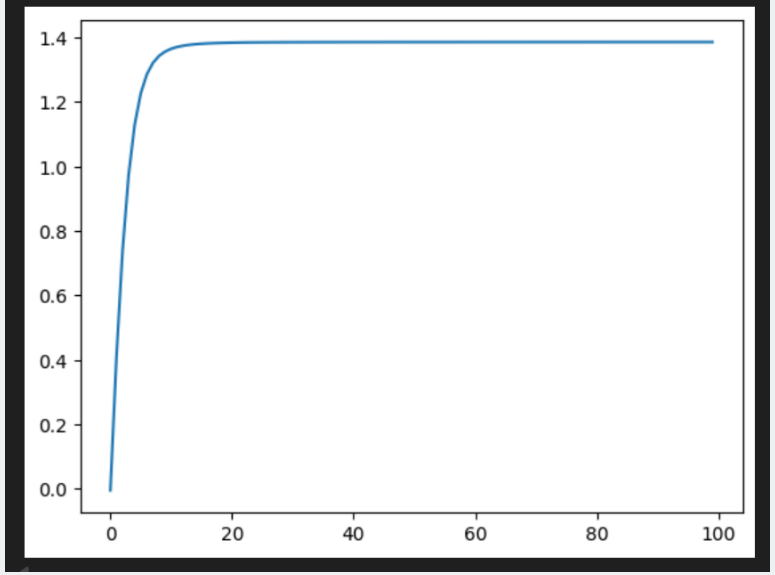
```python
@staticmethod
def _logsumexp(x: torch.Tensor) -> torch.Tensor:
    m = torch.max(x)
    return m + torch.log(torch.sum(torch.exp(x - m)))

def loss(self, alpha):
    """Renyi"""
    self.ensure_commuting_tables()
    T_diag = self.S @ self.theta                          # (d,)
    rho_diag = torch.real(torch.diagonal(self.rho)).clamp(min=0)  # (d,)
    eps = 1e-20
    term1 = (alpha/(alpha-1.0)) * torch.log(torch.sum(torch.exp((alpha-1.0)*T_diag) * (rho_diag + eps)))
    term2 = torch.log(torch.sum(torch.exp(alpha*T_diag))) - math.log(2**self.n)
    L = term1 - term2
    return -L, L.detach(), term1.detach(), term2.detach()
```

PART 2 – d)

$$S_\alpha(\rho) = \frac{1}{1-\alpha} \log \mathrm{Tr}(\rho^\alpha)$$

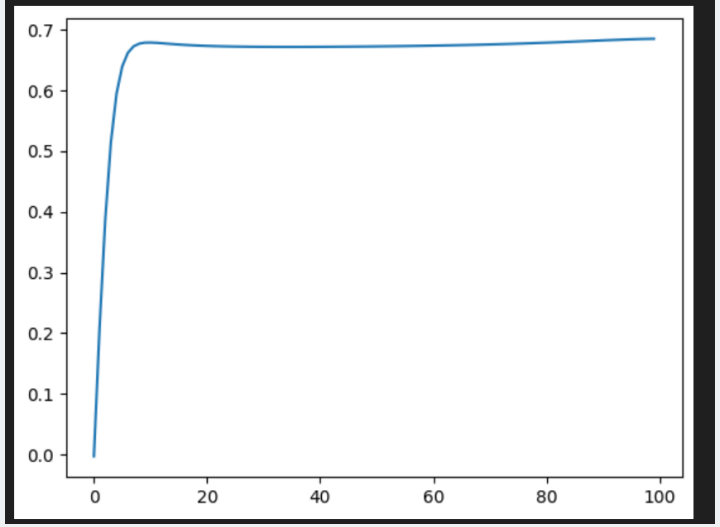
```
# ## pure state
rho1 = one_qubit_mixed_state(p=0)
# rho1 = one_qubit_mixed_state(p=1)
rho1 = torch.tensor(np.kron(rho1,rho1))
```

```
tensor([[0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
        [0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j]], dtype=torch.complex128)
Tr(rho^3) Quntum:0.9999999999999996
Tr(rho^3) Direct:1.0
S_est ≈ 0.000506,  S_true = -0.000000,  S_swap = 0.000000
Train log (step, L_alpha, term1, term2):
(50, 1.385663853680831, 6.347850398654578, 4.962186544973747)
(100, 1.3857883094608852, 6.512811729185858, 5.127023419724972)

[<matplotlib.lines.Line2D at 0x13d2de71cd0>]
```




```
## low rank state
ket0 = torch.tensor([[1.0],[0.0],[0.0],[0.0]], dtype=torch.complex128)  # |00>
ket1 = torch.tensor([[0.0],[1.0],[0.0],[0.0]], dtype=torch.complex128)  # |01>
rho01 = ket0 @ ket0.conj().T
rho02 = ket1 @ ket1.conj().T
rho1 = 0.6 * rho01 + 0.4 * rho02
```

```
tensor([[0.0457+0.j, 0.0000+0.j, 0.0000+0.j, 0.0000+0.j],
        [0.0000+0.j, 0.1680+0.j, 0.0000+0.j, 0.0000+0.j],
        [0.0000+0.j, 0.0000+0.j, 0.1680+0.j, 0.0000+0.j],
        [0.0000+0.j, 0.0000+0.j, 0.0000+0.j, 0.6183+0.j]],
       dtype=torch.complex128)
Tr(rho^3) Quntum:0.24592887274432712
Tr(rho^3) Direct:0.2459288727443272
S_est ≈ 0.701961,  S_true = 0.701356,  S_swap = 0.701356
Train log (step, L_alpha, term1, term2):
(50, 0.6716881003252038, 4.598849522862708, 3.9271614225375044)
(100, 0.6843336177552533, 3.027089461417242, 2.3427558436619886)

[<matplotlib.lines.Line2D at 0x13d2e059bb0>]
```



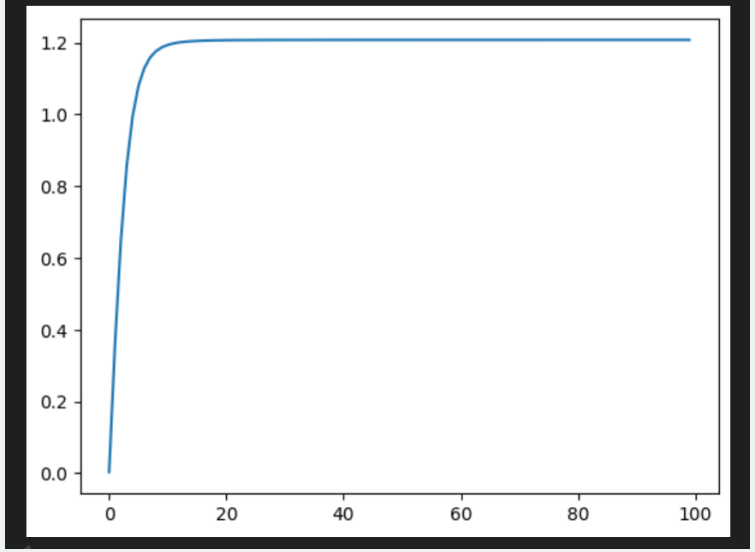
PART 2 – d)

$$S_\alpha(\rho) = \frac{1}{1-\alpha} \log \mathrm{Tr}(\rho^\alpha)$$

```
# ## mixed state
rho1 = one_qubit_mixed_state(p=np.random.random())
rho1 = torch.tensor(np.kron(rho1,rho1))
```

```
alpha=3
```

```
tensor([[0.8875+0.j, 0.0000+0.j, 0.0000+0.j, 0.0000+0.j],
        [0.0000+0.j, 0.0546+0.j, 0.0000+0.j, 0.0000+0.j],
        [0.0000+0.j, 0.0000+0.j, 0.0546+0.j, 0.0000+0.j],
        [0.0000+0.j, 0.0000+0.j, 0.0000+0.j, 0.0034+0.j]],
       dtype=torch.complex128)
Tr(rho^3) Quntum:0.6992624657302053
Tr(rho^3) Direct:0.6992624657302063
S_est ≈ 0.179035,  S_true = 0.178865,  S_swap = 0.178865
Train log (step, L_alpha, term1, term2):
(50, 1.2072302176329153, 6.119262875052509, 4.912032657419593)
(100, 1.2072590723720218, 6.21959562486548, 5.012336552493458)

[<matplotlib.lines.Line2D at 0x13d2dc553d0>]
```




```
alpha=2.5
```

```
tensor([[0.3412+0.j, 0.0000+0.j, 0.0000+0.j, 0.0000+0.j],
        [0.0000+0.j, 0.2429+0.j, 0.0000+0.j, 0.0000+0.j],
        [0.0000+0.j, 0.0000+0.j, 0.2429+0.j, 0.0000+0.j],
        [0.0000+0.j, 0.0000+0.j, 0.0000+0.j, 0.1729+0.j]],
       dtype=torch.complex128)
S_est ≈ 1.317359,  S_true = 1.317359
Train log (step, L_alpha, term1, term2):
(50, 0.06869098939994903, 0.2246267406577356, 0.15593575125778658)
(100, 0.06893421142359213, 0.24336049613646027, 0.17442628471286814)

[<matplotlib.lines.Line2D at 0x13d2df019a0>]
```