DEPARTMENT OF PHYSICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

FOPRA 34: Simulating Quantum Many-Body Dynamics on a Current Digital Quantum Computer

Authors:	Franz von Silva Tarouca
	Christian Gnandt
	Cristian Emiliano Godinez
Supervisors:	Johannes Feldmeier
	Wilhelm Kadow
	Yujie Liu
Submission Date:	July 4, 2022

Contents

1. Introduction **1**

2. Day 1 – Quantum Computation with Cirq **2**

2.1. Readout Matrix Unfolding 2

2.2. Quantum Teleportation 3

2.3. Rabi Oscillation 4

2.4. Single-Qubit Tomography 9

2.5. Variational Quantum Eigensolver 11

3. Day 2 – Quantum Phase Transitions in an Ising Model **13**

3.1. Dynamical Quantum Phase Transitions 13

3.2. Transverse Field Ising Model 15

3.3. DQPTs in the TFI using Cirq 16

3.3.1. Implementing the time evolution 17

3.4. Tracking Entanglement Production 20

A. Appendix **23**

A.1. Day 1 24

A.2. Day 2 37

B. Bibliography **51**

1. Introduction

The field of quantum many-body physics covers a wide range of systems with increasing scale, with many relying on numerical simulations to solve their equations. Quantum computing devices are an ideal platform to explore these open questions, as they are able to naturally handle the exponential complexity of a quantum system. This report studies dynamical phase transitions, a quantum many-body phenomena, in a specific Ising model using simulations with the Google Cirq framwork for noisy intermediate scale quantum computing. The code uses common methods in quantum simulation, such as Trotterization, Generalized Amplitude Damping channel and Tomography, to run a local simulation that mirrors the protocol used to control and compute on a real quantum device. The theoretical groundwork and coding tasks are taken from a manual [1]. The report summarizes some of the core theoretical concepts needed for the exercises and discusses the results of the simulations.

2. Day 1 – Quantum Computation with Cirq

2.1. Readout Matrix Unfolding

Current and near-term quantum computing devices are subject to significant inaccuracies due to the multitudes of noise sources affecting the computation and the measurement process. With appropriate error mitigation, however, these noisy intermediate-scale quantum (NISQ) devices can still be utilized. Readout matrix unfolding is such an error mitigation protocol that combats readout errors. The protocol consists of a calibration and an unfolding sequence. Here we are considering an n qubit device. The calibration is measuring the readout matrix P that quantifies the measurement errors. The element P_{ab} is the probability that the bitstring b is measured after preparing the bitstring a. The matrix thus has $2^n \times 2^n$ entries. For small errors rates we approximate the relation between a measured arbitrary bitstring v' and the error-free bitstring v as

$$v'_a = \sum_b P_{ab} v_b. \tag{2.1}$$

The error mitigated result v is obtained by inverting the readout matrix P from the calibration and applying it to the measured bitstring v'. If the measurement is normalized before this procedure, then the mitigated result is still normalized and thus physical in that aspect. This is shown by looking at the absolute value of the sum over a in equation (2.1). The probabilistic definition of P demands that all possible outcomes, here the columns, sum to unity, $\sum_a P_{ab} = 1$.

Exercise 1

$$\left|\sum_a v'_a\right| = \left|\sum_{a,b} P_{ab} v_b\right| = \left|\sum_b v_b\right| = 1 \tag{2.2}$$

While normalization is therefore guaranteed for an appropriate input, the final result v might still be nonphysical, as negative entries are possible. For this report this method with an additional check against negatives values will be sufficient.

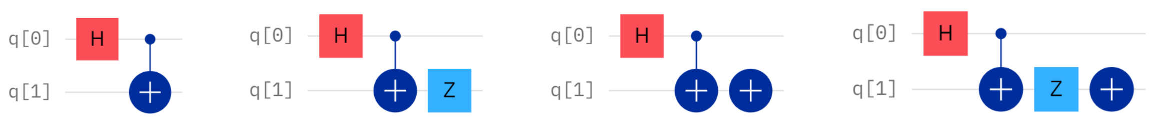
2.2. Quantum Teleportation

Quantum teleportation is a protocol to copy a quantum state $|\varphi\rangle$ over any distance between 2 locations. The sender and receiver need to share an entangled EPR qubit pair and the original state is destroyed in the process. The protocol involves preparing and measuring qubits in the EPR basis, seen in equation (2.3).

$$|\Phi^{\pm}\rangle = \frac{1}{\sqrt{2}}(|00\rangle \pm |11\rangle) \qquad |\Psi^{\pm}\rangle = \frac{1}{\sqrt{2}}(|01\rangle \pm |10\rangle) \tag{2.3}$$

Exercise 2

The preparation starting from the ground state $|00\rangle$ is achieved by creating a superposition with the Hadamard gate H and then entangling them with the CNOT gate. Figure 2.1 shows the circuits to create each EPR state. To measure in the EPR basis, the gate to create $|\Phi^{+}\rangle$ is applied



(a) $|\Phi^{+}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ **(b)** $|\Phi^{-}\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$ **(c)** $|\Psi^{+}\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$ **(d)** $|\Psi^{-}\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$

Figure 2.1.: Preparation circuit for all EPR states. The plus symbol denotes addition modulo 2 and therefore an X or NOT gate.

in reverse before the qubits are measured sequentially in the computational basis. The mapping between the computational basis and the EPR basis is determined by the circuit. With circuit 2.1a the mapping is $|\Phi^{+}\rangle \leftrightarrow |00\rangle$, $|\Phi^{-}\rangle \leftrightarrow |10\rangle$, $|\Psi^{+}\rangle \leftrightarrow |01\rangle$ and $|\Psi^{-}\rangle \leftrightarrow |11\rangle$.

Exercise 3

Depending on the outcome of the EPR measurement, a controlled unitary needs to be applied to obtain the exact original state. This is clear when the complete system state, including the original state $|\varphi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, is rewritten in the EPR basis

$$\begin{aligned} |\varphi\rangle_0 \frac{1}{\sqrt{2}}(|00\rangle_{12} + |11\rangle_{12}) = {} & |\Phi^{+}\rangle_{01}(\alpha\,|0\rangle_2 + \beta\,|1\rangle_2) + |\Phi^{-}\rangle_{01}(\alpha\,|0\rangle_2 - \beta\,|1\rangle_2) \\ & + |\Psi^{+}\rangle_{01}(\alpha\,|1\rangle_2 + \beta\,|0\rangle_2) + |\Psi^{-}\rangle_{01}(\alpha\,|1\rangle_2 - \beta\,|0\rangle_2)\,. \end{aligned} \tag{2.4}$$

The state of qubit q[2] after the EPR measurement will carry the α and β coefficients of the original q[0] state, however, sometimes swapped or with an additional sign. The sign is removed with a controlled Z gate (CZ), the switch reversed with a controlled X gate (CX).

```
0: ───Ry(0.381π)───────@───H───M───@───────
                       │       │   │
1: ───H────────────@───X───────M───┼───@───
                   │               │   │
2: ────────────────X───────────────@───X───

First qubit state after preparation     System state after teleportation
0.83|0⟩ + 0.56|1⟩                       0.83|000⟩ + 0.56|001⟩
First qubit state after preparation     System state after teleportation
0.83|0⟩ + 0.56|1⟩                       0.83|010⟩ + 0.56|011⟩
First qubit state after preparation     System state after teleportation
0.83|0⟩ + 0.56|1⟩                       0.83|000⟩ + 0.56|001⟩
First qubit state after preparation     System state after teleportation
0.83|0⟩ + 0.56|1⟩                       0.83|100⟩ + 0.56|101⟩
```

Figure 2.1.: Circuit diagram and output for quantum teleportation. The state $|\varphi\rangle$ is initially on qubit 0 and is then copied to qubit 2. The @ symbol denotes the control qubit in a controlled gate. M shows measurements on qubits. The log shows the original and teleported state for four trials.

Exercise 4

The complete quantum teleportation protocol is implemented and tested in the Google Cirq framework with the circuit seen in diagram 2.1. The corresponding code for all the exercises of Day 1 are found in the appendix chapter A.1, here we will focus on the discussion of the results. The original state $|\varphi\rangle$ is created by rotating around the y axis with a random angle, the example above uses 0.381π. The log included in 2.1 shows the α and β values of $|\varphi\rangle$ after the rotation and the complete system state after teleportation for 4 different trials. The circuit is working correctly as qubit 2 has matching α and β values in all trials, independent of the measurement results on qubit 0 and 1. In trial one and three the EPR measurement yielded $|\Phi^+\rangle$, in trial two the $|\Phi^-\rangle$ and in trial three the $|\Psi^+\rangle$ state.

2.3. Rabi Oscillation

Rabi osciallation is the primary phenomena to manipulate single qubits to any desired state. For a two-level system (TLS) quantified along the $\hat{\sigma}_z$ axis, an oscillating $\hat{\sigma}_x$ component in the Hamiltonian

$$H = -\frac{\omega_0}{2}\hat{\sigma}_z + \omega_1\cos(\omega t)\hat{\sigma}_x \tag{2.5}$$

will result in the system oscillating between the ground and excited state $|0\rangle$ and $|1\rangle$. With the initial state $|\Psi\rangle = \alpha(t)|0\rangle + \beta(t)|1\rangle = |0\rangle$ the dynamics in the rotating wave approximation

(RWA) are given by

$$\begin{aligned} \alpha(t) &= e^{-i\omega t/2}\left(\cos\left(\frac{\Omega t}{2}\right) - \frac{i\Delta}{\Omega}\sin\left(\frac{\Omega t}{2}\right)\right), \\ \beta(t) &= e^{i\omega t/2}\left(-\frac{i\omega_1}{\Omega}\sin\left(\frac{\Omega t}{2}\right)\right) \end{aligned} \tag{2.6}$$

where $\Delta = \omega - \omega_1$ is the detuning and $\Omega = \sqrt{\omega_1^2 + \Delta^2}$ the Rabi frequency. The condition for RWA to work well is a comparatively weak drive and a small detuning.

$$\omega_1 \ll \omega_0 \quad \text{and} \quad \Delta \ll 2\omega_0. \tag{2.7}$$

Exercise 5

The population of the groud and excited states are retrieved from equations (2.6) by taking the square of the absolute value.

$$p_0 = |\alpha(t)|^2 = \cos^2\frac{\Omega}{2}t + \left(\frac{\delta}{\Omega}\right)^2 \sin^2\frac{\Omega t}{2} \tag{2.8}$$

$$p_1 = |\beta(t)|^2 = \left(\frac{\omega_1}{\Omega}\right)^2 \sin^2\frac{\Omega t}{2} = \left(\frac{\omega_1}{\Omega}\right)^2 \frac{1}{2}(1 - \cos\Omega t) \tag{2.9}$$

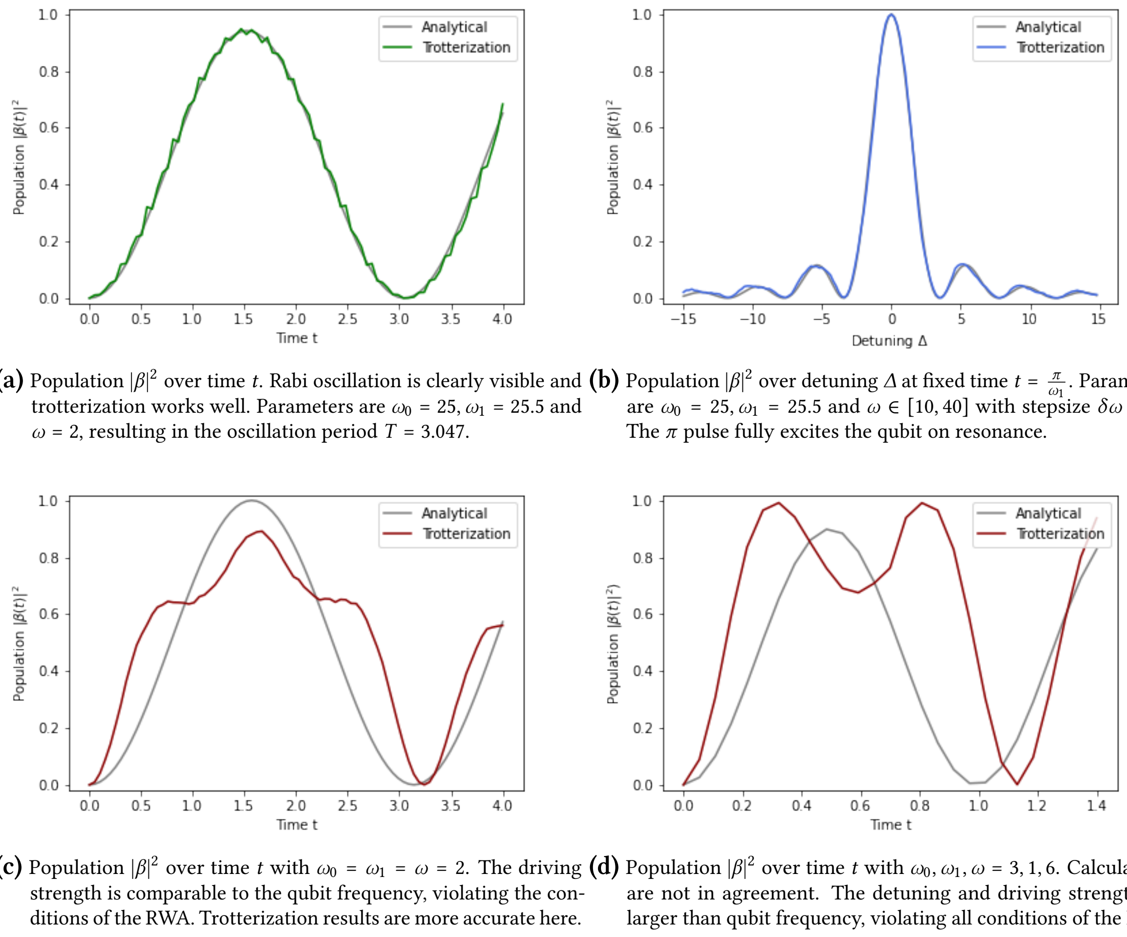
A common method to simulate the dynamics of quantum system with a known Hamiltonian is trotterization. The relevant time interval is discretized into sufficiently small steps and the precise time evolution $U(t)$ is approximated by repeatedly applying a gate that replaces the dynamics for one time step

$$U(t) = \exp\left(-i\int_0^t H(t)dt\right) \approx \prod_{n=0}^{N-1} \exp\left(-iH(n\delta t)\delta(t)\right). \tag{2.10}$$

Exercise 6

To test the performance of this approximation we simulate Rabi oscillations using trotterization and compare the final state with the analytical values from equation (2.9). For plot 2.2a and 2.2b the qubit frequency is set to $\omega_0 = 25$, the driving frequency to $\omega_1 = 25.5$ with a driving strength $\omega = 2$. The frequencies are given without units as this is how they are used in the code. What is relevant for their physical properties are their values relative to each other and the considered time range. Qubit and drive are almost in resonance here with only a slight detuning $\Delta = 0.5$, meaning the Rabi frequency $\Omega = 2.062$ is very close to the driving frequency ω_1 and the amplitude $\left(\frac{\omega_1}{\Omega}\right)^2$ is close to 1. This is validated by the plot 2.2a which shows the population of the excited state undergoing a full Rabi oscillation at close to maximum amplitude. The population over the detuning seen in 2.2b also confirms the correct timing for the simulation. At the fixed time $t = \frac{\pi}{\omega_1}$ the interaction equals a π-pulse that excites the TLS if the detuning Δ is close to zero, which is clearly visible in the plot.

Figure 2.2.: Comparison between analytical and trotterized calculations of population $|\beta|^2$ for a driven two-level system. The time step for trotterization is $\delta t = 0.05$.



(a) Population $|\beta|^2$ over time t. Rabi oscillation is clearly visible and trotterization works well. Parameters are $\omega_0 = 25, \omega_1 = 25.5$ and $\omega = 2$, resulting in the oscillation period $T = 3.047$.

(b) Population $|\beta|^2$ over detuning Δ at fixed time $t = \frac{\pi}{\omega_1}$. Parameters are $\omega_0 = 25, \omega_1 = 25.5$ and $\omega \in [10, 40]$ with stepsize $\delta\omega = 0.2$. The π pulse fully excites the qubit on resonance.

(c) Population $|\beta|^2$ over time t with $\omega_0 = \omega_1 = \omega = 2$. The driving strength is comparable to the qubit frequency, violating the conditions of the RWA. Trotterization results are more accurate here.

(d) Population $|\beta|^2$ over time t with $\omega_0, \omega_1, \omega = 3, 1, 6$. Calculations are not in agreement. The detuning and driving strength are larger than qubit frequency, violating all conditions of the RWA.

For the population plots seen in 2.2c and 2.2d the frequencies have been adjusted to $\omega_0 = \omega_1 = \omega$ and $\omega_0 = 3, \omega_1 = 1, \omega = 6$ respectively. In both cases the trotterization and analytical results yield signficantly different results. The discrepancy is explained by the applicability of the RWA used to derive the analytical equations. The driving strength $\omega_1 = 2$ in scenario 2.2c is comparable in size to the TLS frequency $\omega_0 \sim \omega_1$. The system is therefore not in a weak driving regime and RWA is not a good solution. In scenario 2.2d the detuning $\Delta = 3$ has been increased to be close to the TLS frequency $\Delta \sim \omega_0$. Here the parameters violate both conditions for RWA (2.7) and magnify the difference between the computations.

Exercise 7

To get closer to the behavior of physical quantum devices, we extend the simulator with a noise model. These models add the decoherence effects from a physical environment of a quantum computer without simulating the environment itself. We are implementing the generalized amplitude damping channel (GAD) that describes spontaneous emission and thermal excitations. The Kraus operators of the channel are

$$E_0 = \sqrt{p}\begin{pmatrix}1 & 0\\ 0 & \sqrt{1-\gamma}\end{pmatrix}, \quad E_1 = \sqrt{p}\begin{pmatrix}0 & \sqrt{\gamma}\\ 0 & 0\end{pmatrix}$$
$$E_2 = \sqrt{1-p}\begin{pmatrix}\sqrt{1-\gamma} & 0\\ 0 & 1\end{pmatrix}, \quad E_3 = \sqrt{1-p}\begin{pmatrix}0 & 0\\ \sqrt{\gamma} & 0\end{pmatrix}$$

with $1-p$ being the probability for thermal excitation and γ the probability for spontaneous emission. The channel Λ is then with standard definition

$$\Lambda(\rho) = \sum_{i=0}^{3} E_i \rho E_i^\dagger. \tag{2.11}$$

The diagram 2.3 shows a direct comparison between the simulation with and without the GAD.

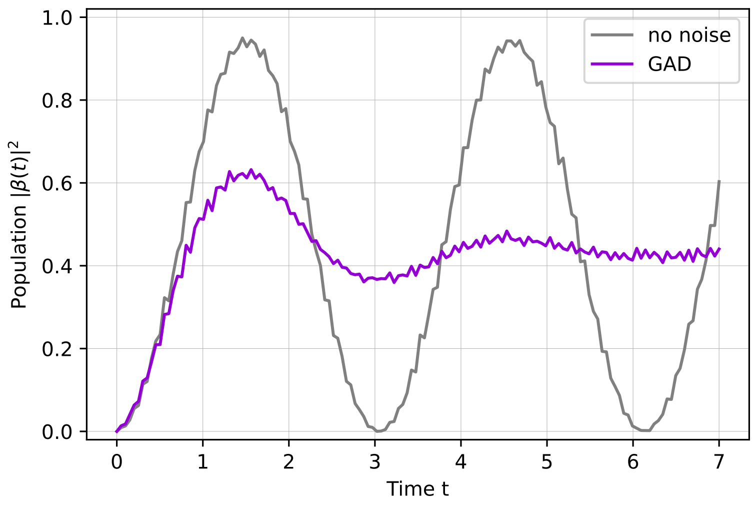
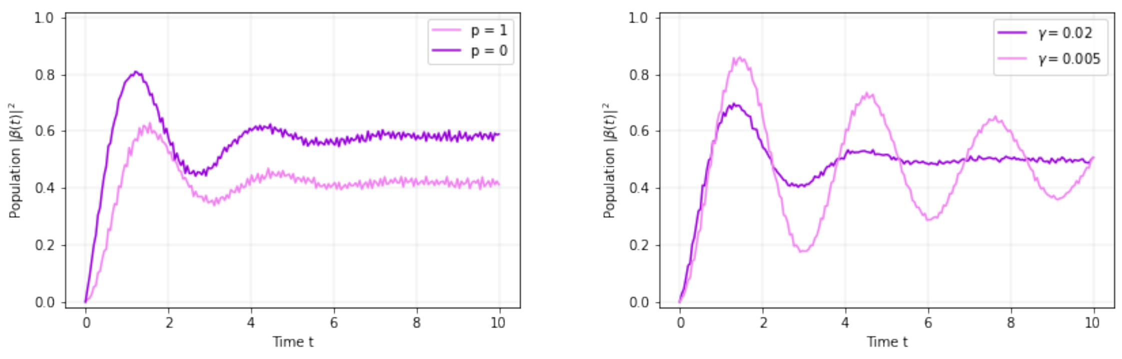


Figure 2.3.: Comparison between trotterization with and without noise. The addition of the dampening channel clearly reduces the amplitude of the Rabi oscillation. Parameters for the system, noise model and trotterization are $\omega_0 = 25.5$, $\omega_1 = 25$, $\omega = 2$, $p = 0.9$, $\gamma = 0.02$ and $\delta t = 0.05$.

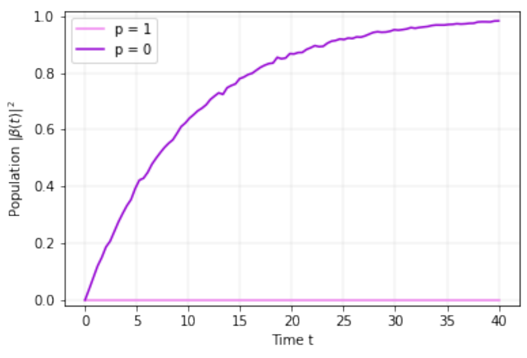
The introduction of decoherence results in a dampened oscillation that approaches a stable population. The strength of the dampening is determined by γ, which is clearly visible in 2.4b. The oscillation with $\gamma = 0.02$ fades out within the simulation time, while the population for $\gamma = 0.005$ oscillates even beyond the visible interval. The peaks of the Rabi oscillation fall exponentially, characterized by the T_1 decoherence time. The second dominating feature in the graphs is the asymptote, which is controlled by the thermal excitation probability p and the Rabi drive. This is demonstrated in 2.4a where the lower value for p, corresponding to higher thermal excitation rates, concludes in a higher limit after oscillation. Even between the edge cases $p = 0$ and 1 shown in 2.4a, the difference in the final limit is fairly small considering the available range is $[0, 1]$. This is due to the Rabi oscillations driving the TLS to a population around the median 0.5 and away from the extreme cases $\alpha = 1$ or $\beta = 1$. The progression without any Rabi drive is shown in 2.4c. For $p = 1$ the GAD reduces to the normal amplitude dampening channel, driving the system to the state $|0\rangle$, while for $p = 0$ the channel enhances the excited state amplitude, driving the system to the state $|1\rangle$ [2].

Figure 2.4.: Trotterized Rabi oscillation with GAD for varying p and γ. The time step is still $\delta t = 0.05$.



(a) Population $|\beta|^2$ over time t for $\gamma = 0.01$ and $p = 0.095$ and 0.05. A lower p corresponds to a higher thermal excitation rate and higher asymptote.

(b) Population $|\beta|^2$ over time t for $p = 0.5$ and $\gamma = 0.002$ and 0.005. A higher γ corresponds to a higher spontaneous emission rate and stronger dampening, which is clearly visible for $\gamma = 0.02$.



(c) Population $|\beta|^2$ over time t for $\gamma = 0.02$ and $p = 1$ and 0, without a driving term. At $p = 0$ GAD is an amplitude enhancing channel, that eventually turns all initial states to $|1\rangle$. With $p = 1$ GAD equals to the normal dampening channel, where all initial states end up in $|0\rangle$.

2.4. Single-Qubit Tomography

Quantum state tomography is the method for reconstructing a qubit state by measuring the expectation values of the Pauli operators $\hat{\sigma}_i$. One can reconstruct the qubit state from these expectation values because the Pauli operators form an orthonormal basis of the 2×2 Hilbert space. Thus every qubit density matrix ρ can be expressed as

$$\rho = \vec{c} \cdot \vec{\hat{\sigma}} \tag{2.12}$$

with $\vec{\hat{\sigma}} = \{\mathbb{1}, \hat{\sigma}_{\mathrm{x}}, \hat{\sigma}_{\mathrm{y}}, \hat{\sigma}_{\mathrm{z}}\}$ the pseudo-vector of Pauli matrices and the vector coefficients $\vec{c} = \frac{1}{2}\langle \vec{\hat{\sigma}} \rangle$. Measuring the identity matrix contribution is not necessary since it's defined by the normalization condition $\mathrm{Tr}(\rho) = 1 \longrightarrow c_0 = \frac{1}{2}$.
Usually the measurement basis for a quantum system is the z-basis. To measure $\langle \hat{\sigma}_{\mathrm{x}} \rangle$ and $\langle \hat{\sigma}_{\mathrm{y}} \rangle$ we need to perform a basis transformation from the measurement basis to the desired one using the cirq native gate set.

Exercise 8

The basis transformations for measurements in the x- and y-basis are

$$\mathrm{HZH} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \mathrm{X}, \tag{2.13}$$

$$\mathrm{SHZHS}^{\dagger} = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \mathrm{Y}. \tag{2.14}$$

We demonstrate the quantum state tomography using cirq by simulating the measurement of the coefficient vector $\vec{c}$ and reconstructing the density matrix for the state $|\Psi\rangle = TH|0\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + e^{i\frac{\pi}{4}}|1\rangle\right)$. The measurements yield

$$\vec{c} = \frac{1}{2} \begin{pmatrix} 1 \\ 0.000976 \\ 0.706706 \\ 0.706706 \end{pmatrix} \tag{2.15}$$

for the coefficients and thus for the reconstructed density matrix:

$$\rho = \begin{pmatrix} 0.500488 & 0.353353 - 0.353353i \\ 0.353353 + 0.353353i & 0.499512 \end{pmatrix} \tag{2.16}$$

To extract the qubit state from this density we must simply remind ourselves that an arbitrary desinty matrix for a pure qubit state is given by

$$\rho = \begin{pmatrix} |\alpha^2| & \alpha\beta^* \\ \alpha^*\beta & |\beta^2| \end{pmatrix}. \tag{2.17}$$

We can therefore simply read-off the qubit state to be: $|\Psi\rangle_{tomo} = 0.70745196\,|0\rangle + (0.4997558 + 0.4997558i)\,|1\rangle$. We can thus see that the single-qubit tomography method reconstructs $|\Psi\rangle$ very well.
Equipped with this knowledge, we can now confidently use this method to reconstruct the oscillating qubit state from the Rabi oscillation. Figure 2.5 shows the population of the $|1\rangle$ state calculated using the analytical solution from equation (2.9) and using single-qubit tomography. As expected, the tomography reconstruction agrees well with the analytical solution.

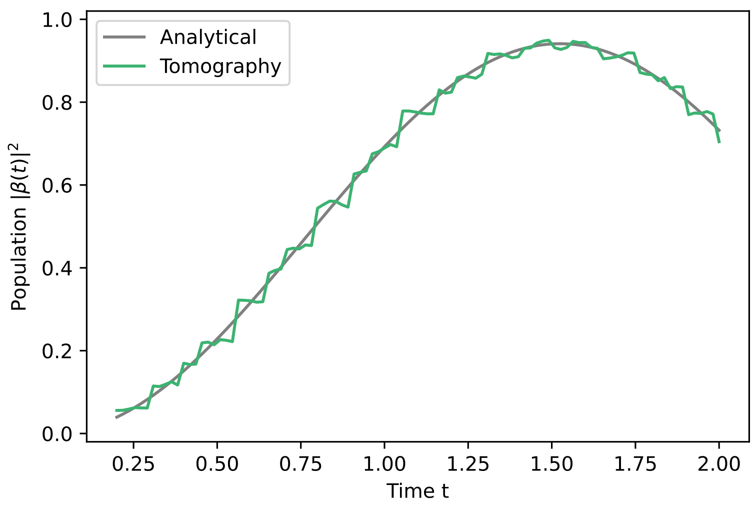


Figure 2.5.: Population $|\beta|^2$ over time t during a Rabi oscillation. State calculated using the analytical equation (2.9) and the state tomography sequence.

2.5. Variational Quantum Eigensolver

The variational quantum eigensolver (VQE) is a hybrid quantum-classical algorithm that makes use of the variational method to find eigenvalues of a Hamiltonian. Let us consider a two-qubit cluster state Hamiltonian:

$$\hat{H} = -\hat{\sigma}_x^1 \hat{\sigma}_z^2 - \hat{\sigma}_z^1 \hat{\sigma}_x^2 \tag{2.18}$$

There are two main steps to applying the VQE algorithm.
First, we construct a parameterized circuit ansatz, as seen in figure 2.6 which prepares the trial state:

$$|\psi(a,b)\rangle = \exp\left[ia\left(\hat{\sigma}_z^1 + \hat{\sigma}_z^2\right)\right] \exp\left[ib\left(\hat{\sigma}_x^1 \hat{\sigma}_z^2 + \hat{\sigma}_z^1 \hat{\sigma}_x^2\right)\right] |00\rangle \tag{2.19}$$

To do so we define the two two-qubit gates RZX $= \exp(ib(\hat{\sigma}_z \otimes \hat{\sigma}_x))$ and RXZ $= \exp(ib(\hat{\sigma}_x \otimes \hat{\sigma}_z))$.

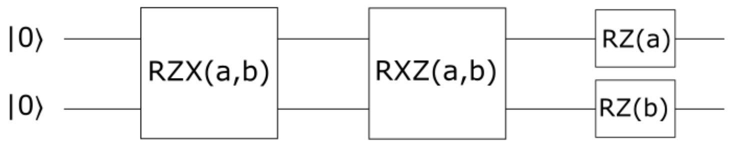


Figure 2.6.: Quantum circuit for creating the parameterized trial state from equation (2.19)

To find the minimum energy state, we make use of a simple grid search over the whole parameter space for a and b. For each choice a and b, we measure the energy expectation value and plot the results on a heat map to find the best trial state.

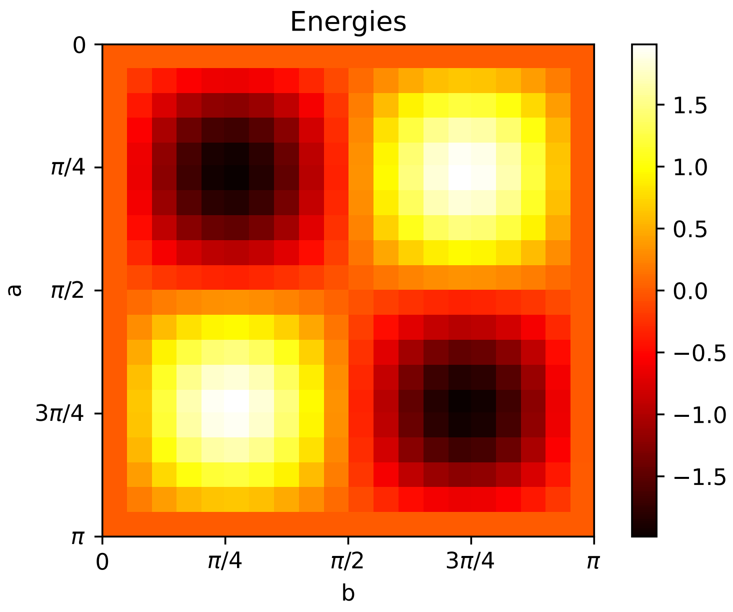


Figure 2.7.: Energy expectation value for the Hamiltonian (2.18) as a function of the trial state parameters a and b defined in (2.19).

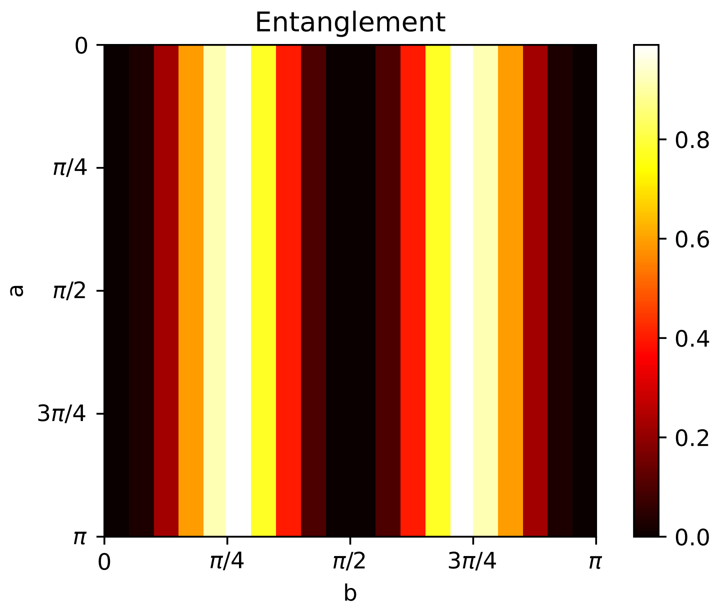


Figure 2.8.: Bipartition entanglement entropy of the trial state from equation (2.19) as a function of the parameters *a* and *b*.

Using the best trial state, we find an approximate ground state energy of $E_0 = -1.986$.
To benchmark the approximate solution, we compare it to the exact solution of the ground state:

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left(|1-\rangle + |0+\rangle\right) \tag{2.20}$$

$$\hat{H}\,|\psi\rangle = \left(-\hat{\sigma}_x^1\hat{\sigma}_z^2 - \hat{\sigma}_z^1\hat{\sigma}_x^2\right)\left(\frac{1}{\sqrt{2}}\left(|1-\rangle + |0+\rangle\right)\right) = -2\,|\psi\rangle \tag{2.21}$$

So we see that the approximation agrees well with the exact solution of $E_0 = -2$.
Lastly, we compute a landscape for the bipartition entanglement entropy over the parameter grid, see heat map 2.8, and compare it to the energy landscape. The first thing one notes is that, unlike the energy landscape, the entanglement entropy is independent of *a* (within numerical accuracy). This is to be expected, since the rotations that depend on *a* are local unitaries which cannot influence entanglement. The second thing one notices is that areas with high absolute energy value correspond to areas of high entanglement entropy except when $a \approx \frac{\pi}{2}$, in which case *a* introduces a relative phase between the terms such that the energy vanishes.

3. Day 2 – Quantum Phase Transitions in an Ising Model

3.1. Dynamical Quantum Phase Transitions

Dynamical quantum phase transitions (DQPT) are non-equilibrium phase transitions in quantum many-body systems that occour on transient time scales and are driven by progressing time as opposed to conventional phase transitions, which are driven by parameters such as temperature or pressure.
At the phase transition certain physical quantities become non-analytic as a function of time. We begin by taking a closer look at the relevant physical quantities.
For more conventional equilibrium phase transitions the central object is the partition function

$$Z = \mathrm{Tr}\left(\exp\left(-\beta\hat{H}(\alpha)\right)\right) = \exp\left(-\beta N f(\beta,\alpha)\right) \tag{3.1}$$

with the inverse of the temperature $\beta = \frac{1}{T}$, a Hamiltonian $\hat{H}$ that depends on a set of parameters α, the system size N and the free energy density $f(\alpha,\beta)$. If $f(\alpha,\beta)$ becomes non-analytic as a function of the parameters α and at zero-temperature, one speaks of a quantum phase transition.
For dynamical phase transitions we define quantities that are formally similiar to the partition function. Consider a many-body system with Hamiltonian H_0 in its ground state Ψ_0. In a process called quantum quench the Hamiltonian is changed nearly instantaneously to H at $t = 0$. Following the quench the state will evolve according to the Schroedinger equation with $|\Psi_0(t)\rangle = e^{-i\hat{H}t}|\Psi_0\rangle$. The amplitude and probability to return to the ground state at time t are then termed Loschmidt amplitude and Loschmidt echo

$$\mathcal{G}(t) = \langle \Psi_0| e^{-i\hat{H}t} |\Psi_0\rangle \,, \tag{3.2}$$

$$\mathcal{L}(t) = |\langle \Psi_0| |\Psi_0(t)\rangle|^2 \,. \tag{3.3}$$

The Loschmidt echo has an exponential dependence on system size N, revealing the formal similarity to the partition function (3.1) after defining the Loschmidt rate $\lambda(t)$ through

$$\mathcal{L} = e^{-N\lambda(t)}. \tag{3.4}$$

The DQPTs will show up as non-analytical points in the Loschmidt rate, in analogy with critical points in the equilibrium phase transitions.

Exercise 1

The rate function $g(t)$ in $\mathcal{G}(t) = e^{-Ng(t)}$ is connected to the Loschmidt rate with $\lambda(t) = 2\Re[g(t)]$, following directly from the definitions

$$\mathcal{L}(t) = |\mathcal{G}(t)|^2 = e^{-Ng(t)}e^{-Ng^*(t)} = e^{-2N\Re[g(t)]} = e^{-N\lambda(t)}. \tag{3.5}$$

For systems with N_{gs} degenerate ground states Ψ_i the Loschmidt echo is the probability to return to any of the ground states and therefore sums over all return probalities

$$\mathcal{L}(t) = \sum_{i=0}^{N_{\text{gs}}-1} |\langle \Psi_i| \, |\Psi_0(t)\rangle|^2 . \tag{3.6}$$

The total Loschmidt rate $\lambda(t)$ is still extracted by taking the logarithm $\lambda(t) = -\frac{1}{N}\ln \mathcal{L}(t)$, however, the additional ground states allow us to introduce individual rates $\lambda_i(t)$

$$e^{-N\lambda_i(t)} = |\langle \Psi_i| \, |\Psi_0(t)\rangle|^2 . \tag{3.7}$$

Exercise 2

In the thermodynamic limit $N \longrightarrow \infty$ the total Loschmidt rate $\lambda(t)$ becomes the smallest individual rate $\lambda(t) = \min[\lambda_i(t)]$. This equality is shown by factoring out the term $e^{-N\lambda_{\min}}$ in the Loschmidt definition, ensuring that all remaining exponential terms have a negative argument $-N(\lambda_i - \lambda_{\min}) < 0$. Then the product formula $\ln a \cdot b = \ln a + \ln b$ is applied.

$$\lambda(t) = -\lim_{N\longrightarrow\infty} \frac{1}{N} \ln \sum_{i=0}^{N_{\text{gs}}-1} e^{-N\lambda_i} = -\lim_{N\longrightarrow\infty} \frac{1}{N} \ln \left[e^{-N\lambda_{\min}} \sum_{i=0}^{N_{\text{gs}}-1} e^{-N(\lambda_i-\lambda_{\min})} \right]$$

All of the terms, except the minimum rate, vanish in the limit after using $\ln 1 + x \simeq x$ for $x \ll 1$.

$$\lambda(t) = \lambda_{\min} - \lim_{N\longrightarrow\infty} \frac{1}{N} \ln \left[1 + \sum_{i=\neq\min}^{N_{\text{gs}}-1} e^{-N(\lambda_i-\lambda_{\min})} \right] \simeq \lambda_{\min} + \lim_{N\longrightarrow\infty} \sum_{i=\neq\min}^{N_{\text{gs}}-1} e^{-N(\lambda_i-\lambda_{\min})} = \lambda_{\min}$$

3.2. Transverse Field Ising Model

The transverse field Ising model (TFI) describes a linear chain of N qubits with nearest neighbor interaction. Due to its known analytical solution it is a suitable system to test the performance of numerical simulations, including our state vector simulation with trotterization. The TFI Hamiltonian $\hat{H}$ reads

$$\hat{H} = -\frac{1}{2}\sum_{\langle i,j\rangle} Z_i Z_j - \frac{g}{2}\sum_i X_i \tag{3.8}$$

with g being the strength of the transverse field. The Z_iZ_j interaction strength is constant at unity. The system is symmetric under the parity operator

$$\hat{P} = \bigotimes_i X_i \tag{3.9}$$

with spectrum $\{-1, 1\}$, since the commutator $[\hat{H}, \hat{P}] = 0$ vanishes. This is derived using the identities $X_i^2 = \mathbb{1}$, $\hat{P} = \hat{P}^\dagger$ and $X_i Z_i X_i = -Z_i$. Each interaction term consists of two Z_i matrices. This cancels out the negative sign when using the latter identity in computing $\hat{P}\hat{H}$.

Exercise 3

$$\left[\hat{H}\hat{P} - \hat{P}\hat{H}\right]\hat{P}^\dagger = \hat{H} - \hat{P}\hat{H}\hat{P}^\dagger = \hat{H} - \hat{H} = 0 \Longrightarrow [\hat{H}, \hat{P}] = 0 \tag{3.10}$$

In the limit $g = 0$ the ground states of $\hat{H}$ are $|\Psi_0\rangle = |0\ldots 0\rangle$ and $|\Psi_1\rangle = |1\ldots 1\rangle$, while for $g \longrightarrow \infty$ only one ground state $|\Psi_+\rangle = |+\ldots+\rangle$ exists. The magnetization of the system is the average spin orientation

$$m_z = \frac{1}{N}\sum_i Z_i. \tag{3.11}$$

For the ground states above it takes the values $m_z^0 = \langle\Psi_0|\,\frac{1}{N}\sum_i Z_i\,|\Psi_0\rangle = -1$, $m_z^1 = +1$ and $m_z^+ = 0$. $|\Psi_+\rangle$ has parity symmetry but no magnetization, while the states Ψ_0 and Ψ_1 have no parity symmetry but finite magnetization. This indicates a symmetry-broken ferromagnetic phase with $m_z \neq 0$ for some critical field strength g_c. To find the critical point the Hamiltonian is reexpressed with a new set of Ising variables

$$\tilde{Z}_n = \bigotimes_{i\leq n} X_i \quad \tilde{X}_n = Z_n Z_{n+1} \tag{3.12}$$

to expose its self-dual structure. For identical behavior these variables need to share the same anticommutation relation $\{Z_i, X_i\} = 0$ as the original variables. Note that $\{Z_i, X_j\} = 0$ for any i and j, as the operators act on different qubits.

Exercise 4

The new anticommutator $\{\tilde{Z}_i, \tilde{X}_i\}$ is evaluated by factoring out the old anticommutator

$$\left\{\tilde{Z}_n, \tilde{X}_n\right\} = \bigotimes_{i\leq n} X_i Z_n Z_{n+1} + Z_n Z_{n+1} \bigotimes_{i\leq n} X_i = [X_n Z_n + Z_n X_n]\, Z_{n+1} \bigotimes_{i\leq n-1} X_i = 0. \tag{3.13}$$

The Hamiltonian H now has the two forms where the second form can be rescaled to $\hat{\tilde{H}}$ through division with with g.

$$\hat{H} = -\frac{1}{2}\sum_{i=1}^{N} Z_i Z_{i+1} - \frac{g}{2}\sum_{i=1}^{N} X_i \tag{3.14}$$

$$= -\frac{1}{2}\sum_{n=1}^{N} \tilde{X}_n - \frac{g}{2}\sum_{n=1}^{N} \tilde{Z}_n \tilde{Z}_{n+1} \tag{3.15}$$

$$\hat{\tilde{H}} = -\frac{1}{2}\sum_{n=1}^{N} \tilde{Z}_n \tilde{Z}_{n+1} - \frac{1}{2g}\sum_{n=1}^{N} \tilde{X}_n \tag{3.16}$$

Multiplying a Hamiltonian does not change the dynamics of the system. The absolute energy scale is an arbitrary choice, therefore only the relative magnitude between the Hamiltonian terms is physical. Specifically, at the critical field value g_c both Hamiltonians $\hat{H}$ and $\hat{\tilde{H}}$ predict the same phase transition. By comparison the critical field has to assume $g_c = \frac{1}{g_c} = 1$.

Exercise 5

The predictions for paramagnetic and ferromagnetic states in the TFI model have been made without considering any environment. For a canonical system at finite temperatures, the ferromagnetic order is not expected to survive over long ranges. In the 1D model the energy associated with a single domain wall is set by the $Z_i Z_{i+1}$ interaction term.

$$\Delta E = -\frac{1}{2}\langle Z_i Z_{i+1}\rangle = \frac{1}{2} \tag{3.17}$$

In addition to the energy cost, every domain wall will add to the entropy S of the chain. While the energy E does not increase with system size N, the number of domain walls and the entropy does. For a system in contact with a thermal bath, the free energy $F = E - TS$ is minimized. The presence of many domain walls is therefore prefered due the higher entropy, despite the associated energy cost. This prevents ferromagnetic phases to extend over long distances.

3.3. DQPTs in the TFI using Cirq

In this section we study the time evolution generated by the Hamiltonian from equation (3.8).

3.3.1. Implementing the time evolution

As in day 1 of the FOPRA, we will perform a Trotter decomposition of the time evolution operator $\hat{U}(t)$. We may write

$$U(t) = e^{-i\hat{H}t} = \left(e^{-i\hat{H}dt}\right)^{\frac{t}{dt}} = \left(e^{\hat{A}+\hat{B}}\right)^{\frac{t}{dt}} \tag{3.18}$$

where $\hat{A} = -\frac{g}{2}dt\sum_i X_i$ and $\hat{B} = -\frac{1}{2}dt\sum_{\langle i,j\rangle} Z_i Z_j$.

Exercises 6 and 7

To implement this unitary in Cirq we first create a circuit that represents the evolution of the system by a small time step dt up to second order Trotterization. To benchmark our results, we simulate the time evolution and extract the magnetization $m_z(t)$ as a function of time for a system of $L = 10$ at $g = 2$, starting from an initial state $|\psi_0\rangle = |0...0\rangle$ of all spins down. As seen in figure 3.1 the trotterization with timestep $dt = 0.25$ agrees very well with the exact solution obtained through diagonalizing the Hamiltonian.

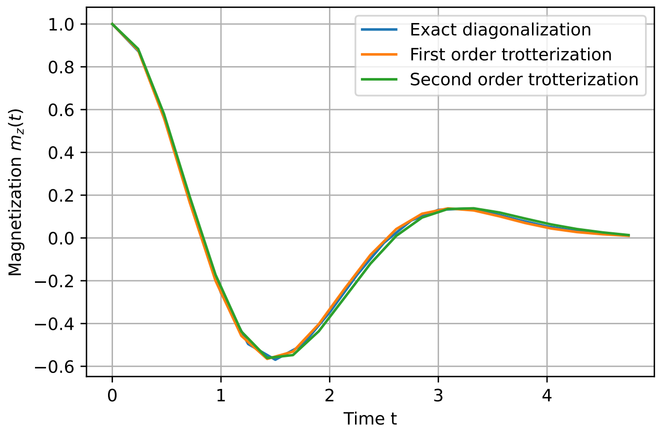
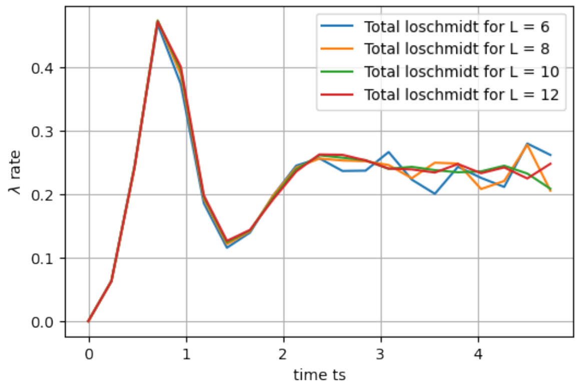


Figure 3.1.: Magnetization m_z over time t for a $L = 10$ qubit chain in the tranverse field Ising model. First and second order trotterization delivers very accurate results compared to the exact evolution computed with diagonalization.

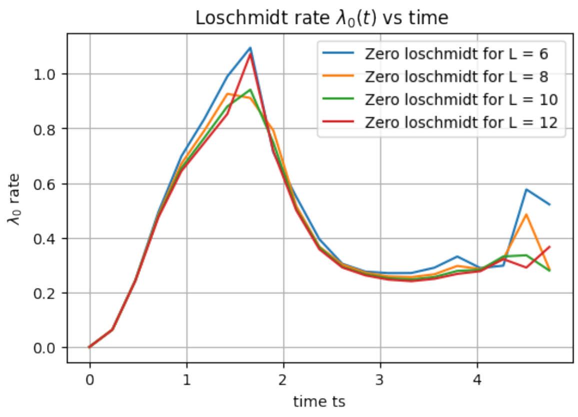
Exercise 8

We now want to evaluate the Loschmidt rate for an initial Hamiltonian $\hat{H}|_{g_0}$, where the two degenerate ferromagnetic ground states are $|\psi_0\rangle = |0...0\rangle$ and $|\psi_0\rangle = |1...1\rangle$. Using the same time evolution as above, we extract the simulated Loschmidt rate by projecting back onto the ground state manifold. To study the results, we first plot the simulated Loschmidt rates for different system sizes while keeping the transverse field strength constant. We plot the results in figure 3.2. We can clearly see DQPTs occuring at $ts \approx 1.8$ regardless of the system size.

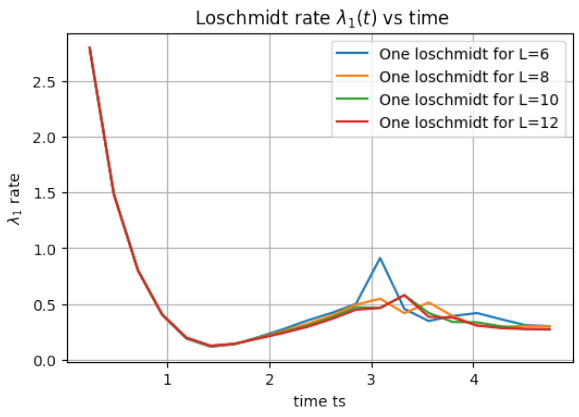
Figure 3.2.: Simulated Loschmidt rates for different system sizes L with a strength of the transverse field g=2.



(a) Loschmidt rate $\lambda(t)$



(b) Loschmidt rate $\lambda_0(t)$



(c) Loschmidt rate $\lambda_1(t)$

Next, in figure 3.3 we do the same but keep the system size fixed and varying the transverse field strength. Again we can clearly see DQPTs occuring for $g > 1$.

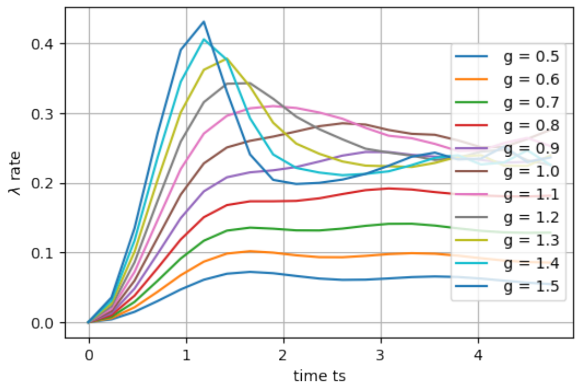
Exercise 9

Now we determine both the magnetization and Loschmidt echo as a function of time again, this time via repeated measurements. Figure 3.4 shows the results, which are essentially identical to the results from the exact solution.
To determine how much sampling is in principle necessary to obtain accurate results, we notice that the maximum value of the total Loschmidt rate is approximately 0.5. Therefore,

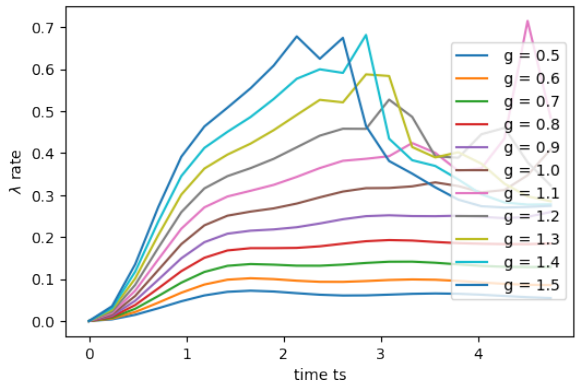
$$\frac{\min(\#\text{of } |0...0\rangle \text{ or } |1...1\rangle}{\#\text{of samplings}} \approx \exp(-0.5N) \tag{3.19}$$

In order to get accurate results, the expected frequency of getting the state $|0...0\rangle$ or $|1..1\rangle$must

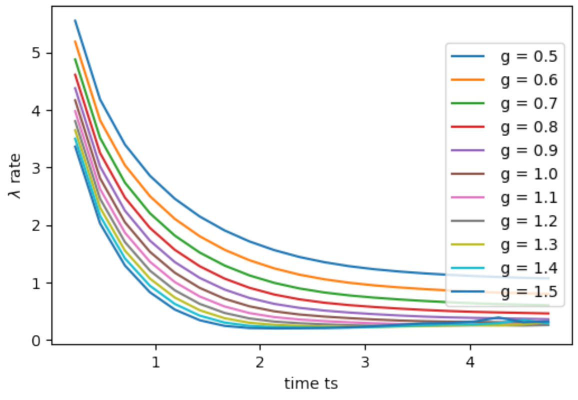
Figure 3.3.: Simulated Loschmidt rates for fixed sytsem size of L=12 and varying strength of the transverse field g=2.



(a) Loschmidt rate $\lambda(t)$

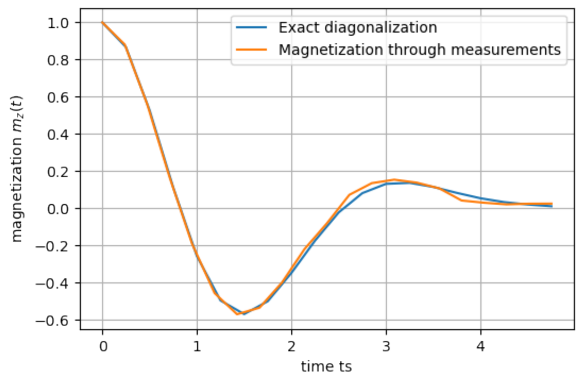


(b) Loschmidt rate $\lambda_0(t)$

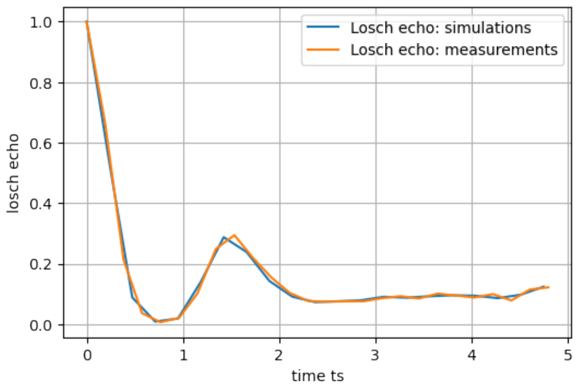


(c) Loschmidt rate $\lambda_1(t)$

Figure 3.4.: Loschmidt echo and magnetization as a function of time reconstructed using simulation of the exact solutions and using repeated measurements respectively.



(a) Magnetization as a function of time.



(b) Loschmidt echo as a function of time.

not be too small. In practice, we could set the expected frequency of ground state to be larger than 10, then the sampling times should fulfill.

$$\#\text{of samplings} \geq 10 \exp(-0.5N) \tag{3.20}$$

Exercise 10

By putting both the magnetization and Loschmidt echo in the same plot, as seen in figure 3.5 we find that the turning point of magnetization (at around t = 1.5) corresponds to the second turning point of $\lambda(t)$.

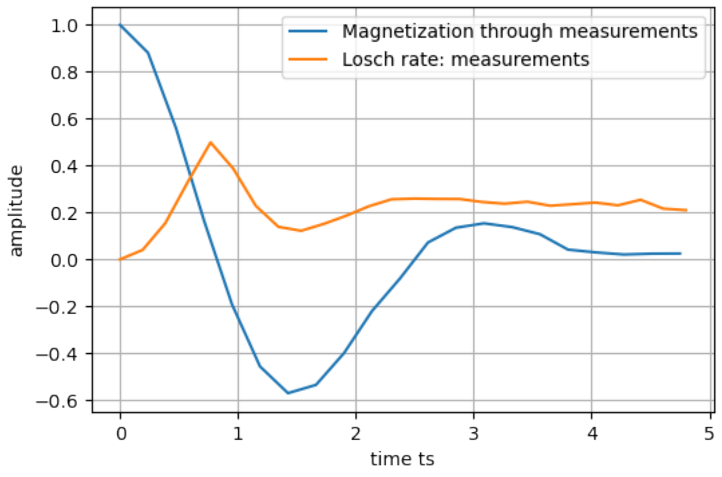


Figure 3.5.: Magnetization and Loschmidt echo as a function of time.

3.4. Tracking Entanglement Production

After the quantum quench takes place, our state evolves from the initial product state to a system with higher entanglement. Due to this effect, it becomes increasingly harder to express the system using classical methods such as matrix product states (MPS).
In this exercise we will quantify the half-chain entanglement using the second Renyi entropy. The reduced density matrix of the subsystem l in consideration is defined as:

$$\rho_l = \mathrm{Tr}_r[\rho] \tag{3.21}$$

Where the trace Tr_r is taken over the qubits that are not part of our subsystem, and ρ is the matrix of the full system. Then, the second Renyi entropy is defined as [1]:

$$S^{(2)} = -\log(\mathrm{Tr}[\rho_l^2]) \tag{3.22}$$

Exercise 17

From [3] we see there is an alternative way of obtaining the Renyi entropy through randomized measurements. First, we define the 1-qubit unitary gates from the circular unitary ensemble (CUE) acting on qubit i defined as:

$$\begin{aligned} U_i(a,b,c) &= \begin{pmatrix} e^{ia}\cos(b) & e^{ic}\sin(b) \\ -e^{-ic}\sin(b) & e^{-ia}\cos(b) \end{pmatrix}. \\ &= e^{i(a+c)Z/2} e^{ibY} e^{i(a-c)Z/2} \end{aligned} \tag{3.23}$$

Where a and c are uniformly distributed over $[0, 2\pi]$, and b is uniformly distributed over $[0, \pi/2]$

The second equality is useful for its code implementation. **Note**: This matrix differs by a minus sign from the one provided in the lab manual, since the gates proposed for cirq do not coincide.

The three parameters are sampled randomly generating a different U_i to be applied on each of the qubits of the system. Then the subsystem is measured in the computational basis. From this, the probabilities $P_U(s_A)$ of measuring the bitstrings s_A on the subsystem are obtained. With this information, an estimation of the purity ($\mathrm{Tr}\,[\rho_l^2]$) of the measured subsystem is obtained as:

$$X = 2^{L_a} \sum_{s_A, s_A'} (-2)^{-D(s_A, s_A')} P_U(s_A) P_U(s_A') \tag{3.24}$$

Where $D(s_A, s_A')$ represents the hamming distance between the bistrings s_A and s_A', i.e. the number of different bits between the two bitstrings. **Important**: the sum runs over all possible tuples containing the combinations of bitstrings, this includes the cases where $s_A = s_A'$ and symmetric cases such as (s_A, s_A') and (s_A', s_A).

This procedure is repeated for several independent realizations of the CUE unitaries, and the purity average $\overline{X}$ is used to obtain the final expression for the second Renyi entropy as:

$$S^{(2)} = -\log(\overline{X}) \tag{3.25}$$

For its implementation in cirq, we write a function `U2_CUE(qubit, symbs)` that applies the unitary U_i from CUE to the qubit in the parameter `qubit`. The unitary is generated from the list of three parameters provided through `symbs`. Then, the hamming function defined above is also implemented in the code as `hamming(s1,s2)` where `s1` and `s2` are the two bitstrings to compare.

The time evolution of our system is studied under the trotterization from previous exercises using the simple evolution method and the Renyi entropy is calculated after each time step. For this simulation, we choose a total of 2000 measurement repetitions, and 200 different instances of the CUE chosen using random generators over the correspondent ranges of the parameters.

The following graph shows the simulation for a hamiltonian using $g = 2$ applied on a two-qubit system that evolves with a time step of $\delta t = 0.25$ for $N = \text{int}(5/\delta t)$ time steps. For comparison, the exact value of the Renyi entropy was calculated through the density matrix of the total system after each time step provided by the function `cirq.Simulator().simulate`.

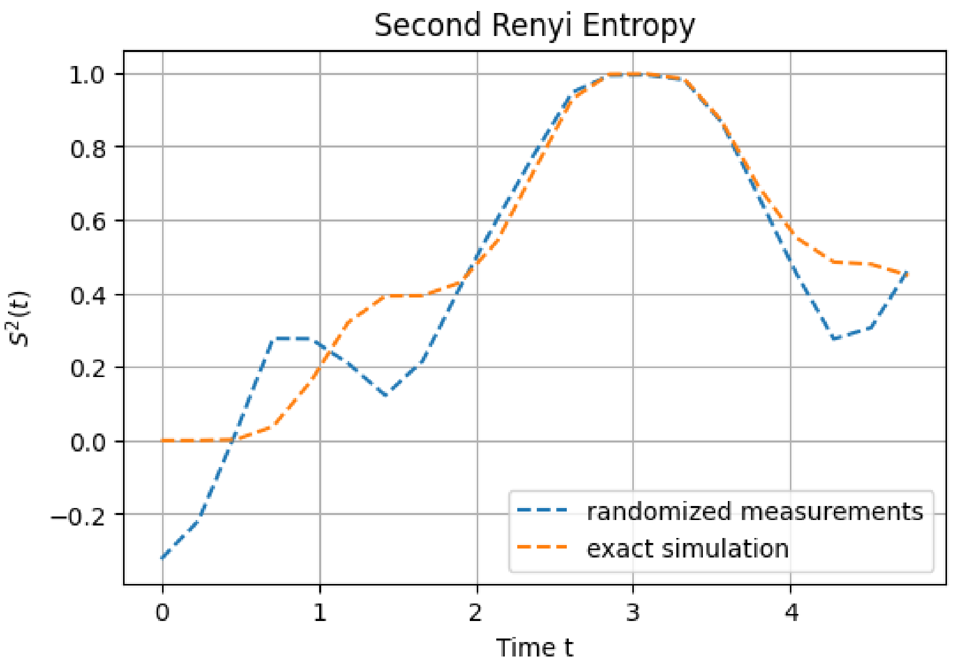


Figure 3.6.: Second Renyi entropy simulation for a two-qubit system evolved by a first degree troterization

A. Appendix

A.1. Day 1

In [ ]:
```
import cirq
import numpy as np
import matplotlib.pyplot as plt
# import cirq_google
from numpy import linalg as LA
from math import log, e

# print the supremacy chip
#print(cirq_google.Sycamore)
```

In [ ]:
```
# These are three qubits on a line
qubits = cirq.LineQubit.range(3)
a = qubits[0]
b = qubits[1]
c = qubits[2]

# This is a collection of operations
# Each operation is a gate
ops = [cirq.H(a), cirq.H(b), cirq.CNOT(b, c), cirq.H(b)]
circuit = cirq.Circuit(ops)

# print circuit diagram
print(circuit)
```

In [ ]:
```
def Bell(n):
    q_chain = cirq.LineQubit.range(2)
    bell = cirq.Circuit()

    bell.append(cirq.H(q_chain[0]))
    bell.append(cirq.CNOT(q_chain[0],q_chain[1]))

    if n == 1:
        return bell
    elif n==2:
        bell.append(cirq.Z(q_chain[1]))
        return bell
    elif n==3:
        bell.append(cirq.X(q_chain[1]))
        return bell
    elif n==4:
        bell.append(cirq.X(q_chain[1]))
        bell.append(cirq.Z(q_chain[0]))
        return bell

bell = Bell(4)
print(bell) #1:

simulator = cirq.Simulator()
result = simulator.simulate(bell)
print('Bra-ket notation for the wavefunction:')
print(result.dirac_notation())
```

Exercise 4

In [ ]:
```
import random
from math import pi

random.seed(41)

# qubit order:
# Message
# Alice
# Bob
# creating basics of circuit
qubits = 3
q_chain = cirq.LineQubit.range(qubits)
preparation = cirq.Circuit()
teleportation = cirq.Circuit()
message = q_chain[0]
alice = q_chain[1]
bob = q_chain[2]

# preparing the random state on qubit 0 and the Bell state on qubit 2 and 3

t = pi*random.uniform(0, 1)

gate = cirq.ry(t)

preparation.append(gate(message))
teleportation.append(gate(message))

teleportation.append(cirq.H(alice))
teleportation.append(cirq.CNOT(alice,bob))

# teleportation measurement and correction

teleportation.append(cirq.CNOT(message,alice))
teleportation.append(cirq.H(message))

teleportation.append(cirq.measure(message,alice))

teleportation.append(cirq.CZ(message,bob))
teleportation.append(cirq.CNOT(alice,bob))

print('\n', teleportation, '\n')

simulator = cirq.Simulator()

trials = 4

for i in range(trials):
        preparation_result = simulator.simulate(preparation)
        teleportation_result = simulator.simulate(teleportation)

        print('First qubit state after preparation \t System state after teleportation')
        print(preparation_result.dirac_notation(), '\t\t\t' ,teleportation_result.dirac_notat
```

In [ ]:

```
random.seed(42)

q_chain = cirq.LineQubit.range(1)
message_test = cirq.Circuit()

t = pi*random.uniform(0, 1)

gate = cirq.ry(t)

message_test.append(gate(message))

print(message_test)
simulator = cirq.Simulator()
result = simulator.simulate(message_test)
print('Bra-ket notation for the wavefunction:')
print(result.dirac_notation())
```

Exercise 6:

In [ ]:

```
from numpy import cos, sin
from math import sqrt
```

```
import matplotlib.pyplot as plt

def Population(w,wo,w1,t):
        Omega = sqrt(w1**2 + (w-wo)**2)
        population = (w1*sin(Omega*t/2)/Omega)**2
        return population

def U(w,wo,w1,t,dt):
        q_chain = cirq.LineQubit.range(1)
        troter = cirq.Circuit()

        N = int(t/dt)

        thetaz = -wo*dt
        thetax = 2*w1*dt

        for n in range(N):
                rx = cirq.rx(thetax*cos(w*n*dt))
                rz = cirq.rz(thetaz)
                troter.append(rz(q_chain[0]))
                troter.append(rx(q_chain[0]))
        return troter, q_chain


def Troterization(w,wo,w1,t,dt, mode='simulate', noise=''):

        reps = 10000
        troter, q_chain = U(w,wo,w1,t,dt)
        simulator = cirq.Simulator()

        if mode == 'tomography':
                sampler = cirq.DensityMatrixSimulator()

                result = cirq.experiments.state_tomography(sampler=sampler, qubits=q_chain, c

                rho =  result._density_matrix
                population = rho[1,1].real
                return population

        if mode == 'run':

                troter.append(cirq.measure(q_chain[0], key='0'))
                result = simulator.run(troter, repetitions=reps)
                counts = result.histogram(key = '0')
                # population for state |1> := prob(measurement=1)
                population = counts[1]/reps
                return population
        elif mode == 'simulate':
                result = simulator.simulate(troter)
                #print('Bra-ket notation for the wavefunction:')
                #print(result.dirac_notation())
                #print('Population of state |1>')
                population = abs(result.final_state_vector[-1])**2
                #print(np.around(population, 3))
                return population

        elif mode == 'noise':
                troter.append(cirq.measure(q_chain[0], key='0'))
                result = cirq.sample(program=troter, noise=noise, repetitions=reps) # type(re
                # type(histogram) = <class 'collections.Counter'>
                counts = result.histogram(key = '0')
                # population for state |1> := prob(measurement=1)
                population = counts[1]/reps
                return population

        else:
                raise Exception('Not a valid mode: {}'.format(mode))
```

In [ ]:
```
# ts = np.linspace(0.2,4,100)
t = 4
dt = 0.05
ts = np.linspace(0, t, int(t/dt))

pop_a = []
theo_a = []
for t in ts:
    w, wo, w1 = 25.5, 25, 2
    theo_a.append(Population(w,wo,w1,t))
    pop_a.append(Troterization(w,wo,w1,t,dt,'run'))

ws = np.arange(10,40,0.2)
w0, w1 = 25, 2
delta = np.arange(10-w0, 40-w0, 0.2)
pop_b = []
theo_b = []
for w in ws:
    t = pi/w1
    theo_b.append(Population(w,wo,w1,t))
    pop_b.append(Troterization(w,wo,w1,t,dt,'run'))

t = 4
dt = 0.05
timeIntervals2 = np.linspace(0, t, int(t/dt))

pop_c = []
theo_c = []
w, w0, w1 = 2, 2, 2
for t in timeIntervals2:
    theo_c.append(Population(w,w0,w1,t))
    pop_c.append(Troterization(w,w0,w1,t,dt,'run'))

t = 1.4
dt = 0.05
timeIntervals3 = np.linspace(0, t, int(t/dt))

pop_d = []
theo_d = []
for t in timeIntervals3:
    w, w0, w1 = 3, 1, 6
    theo_d.append(Population(w,w0,w1,t))
    pop_d.append(Troterization(w,w0,w1,t,dt,'run'))
```

In [ ]:
```
plt.figure()
plt.xlabel(r'Time t')
plt.ylabel(r'Population $|\beta(t)|^2$')
plt.plot(ts, theo_a, 'gray', label='Analytical')
plt.plot(ts, pop_a, 'green', label='Trotterization')
plt.ylim(-0.02, 1.02)
plt.legend()
plt.savefig("exercise06_01.png")
plt.show()

plt.figure()
plt.xlabel(r'Detuning $\Delta$')
plt.ylabel(r'Population $|\beta(t)|^2$')
plt.plot(delta, theo_b, 'gray', label='Analytical')
plt.plot(delta, pop_b, 'royalblue', label='Trotterization')
plt.ylim(-0.02, 1.02)
plt.legend()
plt.savefig("exercise06_02.png")
plt.show()

plt.figure()
plt.xlabel(r'Time t')
plt.ylabel(r'Population $|\beta(t)|^2$')
plt.plot(timeIntervals2, theo_c, 'gray', label='Analytical')
plt.plot(timeIntervals2, pop_c, 'darkred', label='Trotterization')
plt.ylim(-0.02, 1.02)
plt.legend()
plt.savefig("exercise06_03.png")
plt.show()

plt.figure()
plt.xlabel(r'Time t')
plt.ylabel(r'Population $|\beta(t)|^2$')
plt.plot(timeIntervals3, theo_d, 'gray', label='Analytical')
plt.plot(timeIntervals3, pop_d, 'darkred', label='Trotterization')
plt.ylim(-0.02, 1.02)
plt.legend()
plt.savefig("exercise06_04.png")
plt.show()
```

Exercise 7

In [ ]:
```
dt = 0.05
t = 7
w, wo, w1 = 25.5, 25, 2
p = 0.9
gamma = 0.02

pop_a = []
pop_noise = []

noise = cirq.ConstantQubitNoiseModel(cirq.GeneralizedAmplitudeDampingChannel(p=p,gamma=gamma)
ts_new = np.linspace(0, t, int(t/dt))

for t in ts_new:
    pop_noise.append(Troterization(w,wo,w1,t,dt,'noise',noise))

for t in ts_new:
    theo_a.append(Population(w,wo,w1,t))
    pop_a.append(Troterization(w,wo,w1,t,dt,'run'))
```

In [ ]:
```
plt.figure()
plt.xlabel('Time t')
plt.ylabel(r'Population $|\beta(t)|^2$')
plt.plot(ts_new, pop_a, 'gray', label='no noise')
plt.plot(ts_new, pop_noise, 'darkviolet', label='GAD')
plt.ylim(-0.02, 1.02)
plt.legend()
plt.grid(linestyle='-', linewidth=0.2)
plt.savefig("exercise07_01.pdf")
plt.show()
```

In [ ]:
```
w, wo, w1 = 25.5, 25, 2

ps = [1,0]
gammas = [0.02]
dt = 0.05
t = 10

ts_new = np.linspace(0, t, int(t/dt))
populations1 = np.zeros((len(ps),len(ts_new),len(gammas)))

for i, p in enumerate(ps):
    for j, gamma in enumerate(gammas):
        for k, t in enumerate(ts_new):
            noise = cirq.ConstantQubitNoiseModel(cirq.GeneralizedAmplitudeDampingChannel(p=p,
            populations1[i,k,j] = (Troterization(w,wo,w1,t,dt,'noise',noise))

ps = [0.5]
```

```
gammas = [0.02, 0.005]
dt = 0.05
t = 10

ts_new_2 = np.linspace(0, t, int(t/dt))
populations2 = np.zeros((len(ps),len(ts_new_2),len(gammas)))

for i, p in enumerate(ps):
    for j, gamma in enumerate(gammas):
        for k, t in enumerate(ts_new_2):
            noise = cirq.ConstantQubitNoiseModel(cirq.GeneralizedAmplitudeDampingChannel(p=p,
            populations2[i,k,j] = (Troterization(w,wo,w1,t,dt,'noise',noise))

ps = [1,0]
gammas = [0.02]
w, wo, w1 = 25.5, 25, 0
dt = 0.4
t = 40

ts_new_3 = np.linspace(0, t, int(t/dt))
populations3 = np.zeros((len(ps),len(ts_new_3),len(gammas)))

for i, p in enumerate(ps):
    for j, gamma in enumerate(gammas):
        for k, t in enumerate(ts_new_3):
            noise = cirq.ConstantQubitNoiseModel(cirq.GeneralizedAmplitudeDampingChannel(p=p,
            populations3[i,k,j] = (Troterization(w,wo,w1,t,dt,'noise',noise))
```

```
In [ ]: plt.figure()
plt.xlabel('Time t')
plt.ylabel(r'Population $|\beta(t)|^2$')
plt.plot(ts_new, populations1[0,:,0], 'violet', label='p = 1')
plt.plot(ts_new, populations1[1,:,0], 'darkviolet', label='p = 0')
plt.ylim(-0.02, 1.02)
plt.legend()
plt.grid(linestyle='-', linewidth=0.2)
plt.savefig("exercise07_02.png")
plt.show()

plt.figure()
plt.xlabel('Time t')
plt.ylabel(r'Population $|\beta(t)|^2$')
plt.plot(ts_new_2, populations2[0,:,0], 'darkviolet', label=r'$\gamma = 0.02$')
plt.plot(ts_new_2, populations2[0,:,1], 'violet', label=r'$\gamma = 0.005$')
plt.ylim(-0.02, 1.02)
plt.legend()
plt.grid(linestyle='-', linewidth=0.2)
plt.savefig("exercise07_03.png")
plt.show()

plt.figure()
plt.xlabel('Time t')
plt.ylabel(r'Population $|\beta(t)|^2$')
plt.plot(ts_new_3, populations3[0,:,0], 'violet', label='p = 1')
plt.plot(ts_new_3, populations3[1,:,0], 'darkviolet', label='p = 0')
plt.ylim(-0.02, 1.02)
plt.legend()
plt.grid(linestyle='-', linewidth=0.2)
plt.savefig("exercise07_04.png")
plt.show()
```

Notes:

For $p = 1$ the channel reduces to the normal amplitude damping channel \ For $p = 0$ the channel behaves like an amplitude enhancing channel, driving the system to the $|1 >< 1|$ state

Exercise 8:

In [ ]:
```
def Expectation(prepare='', ops='', reps=int(1e6), debug=False):

    """
    prepare: gates to prepare initial state
    ops: operators to change the Z measurement basis into another basis
    examples:
    - for measurement in X basis: H
    - for measurement in Y basis: Sdagger, H
    """

    qubits = cirq.LineQubit.range(1)
    q0 = qubits[0]

    simulator = cirq.Simulator(seed=np.random.seed(2))
    measurement_op = cirq.Circuit()

    if prepare:
        measurement_op.append(p(q0) for p in prepare)
    if ops:
        measurement_op.append(p(q0) for p in ops)

    measurement_op.append(cirq.measure(q0, key='0'))

    if debug:
        print(measurement_op)

    result = simulator.run(measurement_op, repetitions = reps)
    counts = result.histogram(key = '0')

    print(counts)

    N0, N1 = counts[0], counts[1]

    return (N0-N1)/reps


prepare = [cirq.H, cirq.T]

ops_X = [cirq.H]
ops_Y = [cirq.S**-1, cirq.H]

EZ = Expectation(prepare, debug=True)
EX = Expectation(prepare, ops_X, debug=True)
EY = Expectation(prepare, ops_Y, debug=True)

print('Expectation value of Z gate: ', EZ)
print('Expectation value of X gate: ', EX)
print('Expectation value of Y gate: ', EY)
```

Exercise 9:

Idea: to get the state we can diagonalize the matrix, and obtain the first (and only) eigen state, that would be our initial pure state

In [ ]:
```
# calculating the c_i for the density matrix

rho_tomo = 0.5*np.eye(2, dtype = np.complex128)

Es = [EZ, EX, EY] # list of expectation values

paulis = [cirq.Z, cirq.X, cirq.Y] # list of pauli gates
paulis_matrices = [cirq.unitary(P) for P in paulis] # list of pauli gates as numpy arrays
```

```
for i in range(3):
     c = 0.5*Es[i]
     rho_tomo += c*paulis_matrices[i]

print(rho_tomo)

w_tomo, v_tomo = LA.eig(rho_tomo)

print(abs(w_tomo[0])**2)
v_tomo[:,0]
```

```
In [ ]: #circuit for state tomography
sampler = cirq.DensityMatrixSimulator()

qchain = cirq.LineQubit.range(1)
q0 = qchain[0]

ops = [cirq.H(q0), cirq.T(q0)]

circuit9 = cirq.Circuit(ops)

# print circuit diagram
print(circuit9)

simulator = cirq.Simulator()
result = simulator.simulate(circuit9)
print('Bra-ket notation for the wavefunction:')
print(result.dirac_notation())

result = cirq.experiments.state_tomography(sampler=sampler, qubits=qchain, circuit= circuit9,

rho =  result._density_matrix

print(rho)
print(rho[0,0])


w_function, v_function = LA.eig(rho)

print(abs(w_function[0])**2)
v_function[:,0]
```

```
In [ ]: # comparison:
print(LA.norm(rho_tomo - rho))
```

Exercise 10

```
In [ ]: dt = 0.05
ts_10 = np.linspace(0.2,2,100)
pop_10 = []
theo_10 = []
for t in ts_10:
    w, wo, w1 = 25.5, 25, 2
    theo_10.append(Population(w,wo,w1,t))
    pop_10.append(Troterization(w,wo,w1,t,dt,'tomography'))
```

```
In [ ]: plt.figure()
plt.xlabel('Time t')
plt.ylabel(r'Population $|\beta(t)|^2$')
plt.plot(ts_10, np.array(theo_10), 'gray', label='Analytical')
plt.plot(ts_10, pop_10, 'mediumseagreen', label='Tomography')
plt.ylim(-0.02, 1.02)
plt.legend()
plt.savefig("exercise10.pdf")
plt.show()
```

Exercise 11:

In [ ]:
```
"""Define a custom gate with a parameter."""
class RXZ(cirq.Gate):
    def __init__(self, b):
        super(RXZ, self)
        self.b = b

    def _num_qubits_(self):
        return 2

    def _unitary_(self):
        return np.array([
            [np.cos(self.b), 0.0, 1j*np.sin(self.b),  0.0],
            [0.0,  np.cos(self.b), 0.0,  -1j*np.sin(self.b)],
            [1j*np.sin(self.b),  0.0, np.cos(self.b),  0.0],
            [0.0,  -1j*np.sin(self.b), 0.0, np.cos(self.b)]
        ])

    def _circuit_diagram_info_(self, args):
        return f"RXZ({self.b})",  f"RXZ({self.b})"

class RZX(cirq.Gate):
    def __init__(self, b):
        super(RZX, self)
        self.b = b

    def _num_qubits_(self):
        return 2

    def _unitary_(self):
        return np.array([
            [np.cos(self.b), 1j*np.sin(self.b), 0.0,  0.0],
            [1j*np.sin(self.b), np.cos(self.b), 0.0,  0.0],
            [0.0, 0.0, np.cos(self.b), -1j*np.sin(self.b)],
            [0.0, 0.0, -1j*np.sin(self.b), np.cos(self.b)]
        ])

    def _circuit_diagram_info_(self, args):
        return f"RZX({self.b})",  f"RZX({self.b})"

def expXZ(b):
    " exp(ib(XZ)) acting on two qubits"
    q_chain = cirq.LineQubit.range(2)
    rotation = cirq.Circuit()
    rotation.append(RXZ(b=b).on(*q_chain))
    return rotation

def expZX(b):
    " exp(ib(ZX)) acting on two qubits"
    q_chain = cirq.LineQubit.range(2)
    rotation = cirq.Circuit()
    rotation.append(RZX(b=b).on(*q_chain))
    return rotation


def ansatz(a,b):
    q_chain = cirq.LineQubit.range(2)
    circ = cirq.Circuit()

    rz = cirq.rz(-2*a)

    circ.append(expZX(b))
    circ.append(expXZ(b))

    circ.append(rz(q_chain[1]))
    circ.append(rz(q_chain[0]))

    return circ

# testing our implementation of the Ansatz circuit
ansatz_state = ansatz(np.pi/2,np.pi/2)
print(ansatz_state)

result = simulator.simulate(ansatz_state)
print('Bra-ket notation for the wavefunction:')
print(result.dirac_notation()) #this is correct
```

```
In [ ]: # Exercise 12

alist = np.linspace(0,np.pi,20)
blist = np.linspace(0,np.pi,20)

energies = np.empty(shape=(len(alist),len(blist)))

q_chain = cirq.LineQubit.range(2)
q0 = q_chain[0]
q1 = q_chain[1]


for i, a in enumerate(alist):
    for j, b in enumerate(blist):
        energies[i,j] = simulator.simulate_expectation_values(ansatz(a,b), observables=-cirq.
```

```
In [ ]: pos = plt.imshow(energies, cmap='hot', interpolation='nearest', extent=[0,alist[-1],blist[-1]

plt.title('Energies')
plt.ylabel('a')
plt.xlabel('b')
tick_pos = [0, np.pi/4, np.pi/2, 3*np.pi/4, np.pi]
labels = ['0', '$\pi/4$', '$\pi/2$', '$3\pi/4$', '$\pi$']
plt.xticks(tick_pos, labels)
plt.yticks(tick_pos, labels)
plt.colorbar(pos)
plt.savefig('exercise12.pdf')
plt.show()
```

```
In [ ]: print("The smallest energy is: ", np.amin(energies)) #should be -2 in theory
arg = np.argmin(energies)
print(arg)
bmin= arg%20
amin= (arg)//20
print("Indices for the arrays are: (i: {} and j: {}) ".format(amin,bmin) )
print("Values for the indices are: (a: {} and b: {}) ".format(alist[amin],blist[bmin]))
```

```
In [ ]: print(np.where(energies == np.amin(energies))) #printing the pair of indices for which the mi
# here this is (5,5) and (14,14)
```

```
In [ ]: test =simulator.simulate_expectation_values(ansatz(alist[14],alist[14]), observables=-cirq.Z(
print('Energy of ground state:', round(test.real,3))
```

```
In [ ]: def entanglement_entropy(circ, simulator=cirq.Simulator()):

    result = simulator.simulate(circ)
    state = result.final_state_vector

    C = np.reshape(state, newshape=(2,2))

    u, s, vh = LA.svd(C)
    ent = 0
    for sigma in s:
        if sigma != 0:
            ent -= sigma**2 * log(sigma**2,2)

    return ent
```

In [ ]:
```
#13
gs = ansatz(alist[amin],blist[bmin])
print(gs)

result = simulator.simulate(gs)
state = result.final_state_vector

print('=====================================')
print('Bra-ket notation for the wavefunction:')
print(result.dirac_notation())
print('=====================================')

#correct up to a global phase of e^(-ipi/2)
gs_state = np.array([1,1,1,-1])*0.5
print(gs_state)

print('Including the global phase and taking only the real part: ')
print((np.exp(-1j*np.pi/2)*state).real)
print(cirq.equal_up_to_global_phase(gs_state, state, atol=1))

entanglement_entropy(gs) #should be equal to 1 in theory
```

In [ ]:
```
# Exercise 15:

entanglement = np.empty(shape=(len(alist),len(blist)))

simulator = cirq.Simulator()

for i, a in enumerate(alist):
    for j, b in enumerate(blist):
        ansatz_state = ansatz(a,b)
        entanglement[i,j] = entanglement_entropy(ansatz_state, simulator)
```

In [ ]:
```
pos2 = plt.imshow(entanglement, cmap='hot', interpolation='nearest', extent=[0,alist[-1],blis

plt.title('Entanglement')
plt.ylabel('a')
plt.xlabel('b')
tick_pos = [0, np.pi/4, np.pi/2, 3*np.pi/4, np.pi]
labels = ['0', '$\pi/4$', '$\pi/2$', '$3\pi/4$', '$\pi$']
plt.xticks(tick_pos, labels)
plt.yticks(tick_pos, labels)
plt.colorbar(pos2)
plt.savefig('exercise15.pdf')
plt.show()
```

In [ ]:
```
print("The highest Entropy is: ", np.amax(entanglement))
arg = np.argmax(entanglement)
print('at',arg)
b= arg%20
a= (arg-b)//20
print("Indices in the arrays are: (i: {} and j: {}) ".format(a,b) )
print("Values for the indices are: (a: {} and b: {}) ".format(alist[a],blist[b]) )
```

In [ ]:
```
print(np.where(entanglement == np.amax(entanglement))) #printing the pair of indices for whic
```

```
In [ ]: test = ansatz(alist[14],blist[5])
        entanglement_entropy(test, simulator)
```

the two ground states encountered coincid with points where entanglement is highest

Not used:

```
In [ ]: def Troterization_noise(w,wo,w1,t):

            noise = cirq.ConstantQubitNoiseModel(cirq.GeneralizedAmplitudeDampingChannel(p=0.9,gamma=

            troter = U(w,wo,w1,t)

            reps = 10000

            troter.append(cirq.measure(q_chain[0]))
            result = cirq.sample(program=troter, noise=noise, repetitions=reps) # type(result) = <cla

            # other keys don't seem to work
            # type(histogram) = <class 'collections.Counter'>
            histogram = result.histogram(key = '0')

            # _ = cirq.plot_state_histogram(histogram, plt.subplot())
            # plt.show()

            # population for state |1> := prob(measurement=1)
            population = histogram[1]/reps
```

```
In [ ]: # Test for entanglement entropy in a bell state

        q0,q1 = cirq.LineQubit.range(2)

        circ = cirq.Circuit(cirq.H(q0), cirq.CNOT(q0,q1))

        print(circ)

        result = simulator.simulate(circ)
        state = result.final_state_vector

        C = np.reshape(state, newshape=(2,2))

        print(C)

        u, s, vh = LA.svd(C)

        ent = 0

        for sigma in s:
            if sigma != 0:
                ent -= sigma**2 * log(sigma**2,2)

        ent #should be equal to 1 by defintion
        #entanglement= cirq.von_neumann_entropy(density)
        #print(entanglement)
```

A.2. Day 2

In [ ]:
```
import cirq
import numpy as np
import scipy
import sympy
import matplotlib.pyplot as plt
import cirq.experiments.n_qubit_tomography as cen

import itertools
from functools import reduce
from tqdm import tqdm

from math import log

import random
```

In [ ]:
```
def expZZ(t):
    ''' return the gate exp( -it * ZZ ) '''

    return cirq.ZZPowGate(exponent=2*t/np.pi, global_shift=-0.5)

def expX(t):
    ''' return the gate exp( -it * X ) '''
    return cirq.XPowGate(exponent=2*t/np.pi, global_shift=-0.5)

def is_Hermitian(M, rtol = 1e-5, atol = 1e-9):
    return np.allclose(M, np.conjugate(M.T), rtol=rtol, atol=atol)

def is_positive(M, tol = 1e-7):
    s = np.linalg.eigvalsh(M)
    assert (s[0] > -tol)
    for i in range(len(s)):
      if s[i] <= 0:
         s[i] = 1e-12
    return s
```

In [ ]:
```
# define basic Pauli matrices
s_alpha = [np.array([[1,0],[0,1]],dtype=complex),np.array([[0,1],[1,0]],dtype=complex),np.arr


# define the many-body spin operators
def sp(alpha,n,N):
    Sa = s_alpha[alpha]
    for i in range(n):
        Sa = np.kron(s_alpha[0],Sa)
    for j in range(n+1,N):
        Sa = np.kron(Sa,s_alpha[0])
    return Sa



def magn_exact_diagonalization(L,g,t,Npoints):
  # array containing the magnetization of individual basis states
  magnetization_basis_states = -np.array( [np.sum(2*np.array(cirq.big_endian_int_to_bits(val

  # create the hamiltonian
  hamiltonian = np.zeros((2**L,2**L),dtype=complex)
  for i in range(L):
      hamiltonian += g/2*sp(1,i,L)
      if i != L-1:
          hamiltonian += -1/2*sp(3,i,L)@sp(3,i+1,L)

  # diagonalize
  E,V = np.linalg.eig(hamiltonian)

  # time evolve
  magnetization = np.zeros(Npoints)
  initial_state = np.array([int(n==0) for n in range(2**L)])
  overlap = V.transpose().conj() @ initial_state
  for ind,T in enumerate(np.linspace(0,t,Npoints)):
    state_evolved = V @ (np.exp(-1j*T*E) * overlap)
    magnetization[ind] = np.sum(magnetization_basis_states * np.abs(state_evolved)**2)

  return magnetization
```

In [ ]:
```
# System size
L = 10


# System initialization
chain = cirq.GridQubit.rect(1,L)


# Create a circuit
circuit_dummy = cirq.Circuit()
circuit_dummy.append(cirq.I(q) for q in chain)


# Simulate the wave function ...
result_exact = cirq.Simulator().simulate(circuit_dummy)

# ... and extract relevant objects
state = result_exact.state_vector()
state = state/np.linalg.norm(state) # in case not normalized for large system
print(state)
rho = result_exact.density_matrix_of(chain[ round(L/2):L ])

# compute an observable that consists of a sum of Pauli matrices
Paulix = cirq.PauliSum.from_pauli_strings([cirq.X(q) for q in chain])
q_map = result_exact.qubit_map
x_magntization = Paulix.expectation_from_state_vector(state, q_map).real/L

print(x_magntization)

# Perform repeated measurements ...
# repetition = 100
# circuit_measurement = cirq.Circuit()
# circuit_measurement.append(circuit_dummy)
# circuit_measurement.append( [cirq.measure(q) for q in chain], strategy = cirq.InsertStrateg
# result_measure = cirq.Simulator().run(circuit_measurement, repetitions = repetition)
# # ... and extract relevant observables
# keys = [f'(0, {i})' for i in range(L)]
# counts = result_measure.multi_measurement_histogram(keys = keys)
# key0 = tuple( [0] * L )
# probability_0 = counts[key0]/repetition # probability_0 = 1 for circuit_dummy



# Tomography experiments
tomo_qubits = chain[round(L/2):L]
tomo_repetition = 1000
exp = cen.StateTomographyExperiment(tomo_qubits)
sam = cirq.Simulator()
probs = cen.get_state_tomography_data(sam, tomo_qubits, circuit_dummy, exp.rot_circuit, exp.r
tomo_density_matrix = exp.fit_density_matrix(probs)._density_matrix # extract the density mat



# Also useful: convert numbers into bitstrings and vice versa
bit_string0 = [0] * L
number = cirq.big_endian_bits_to_int(bit_string0)
bit_string1 = cirq.big_endian_int_to_bits(val = number, bit_count = L)
print( bit_string0 == bit_string1 ) # True
```

Exercise 6

In [ ]:
```python
def evolve_basic(circ,qubits,g,dt):
    """ one step time evolution of qubits in circ by dt
    through first order approximation"""

    ta = -g*dt/2
    Ua = expX(ta)

    tb = -dt/2
    Ub = expZZ(tb)

    N = len(qubits)

    for i in range(0,N,2): # even sites
        circ.append(Ub(qubits[i],qubits[i+1]))

    for i in range(1,N-1,2): # odd sites
        circ.append(Ub(qubits[i],qubits[i+1]))

    for qubit in qubits:
        circ.append(Ua(qubit)) # e^A = mult_i(exp(-gdt/2*Xi))



def evolve_symmetric(circ,qubits,g,dt):

    """ one step time evolution of qubits in circ by dt
    through first order approximation"""

    ta = -g*dt/4
    Ua = expX(ta)

    tb = -dt/2
    Ub = expZZ(tb)

    N = len(qubits)

    for qubit in qubits:
        circ.append(Ua(qubit)) # e^A/2 = mult_i(exp(-gdt/4*Xi))

    for i in range(0,N,2): # even sites
        circ.append(Ub(qubits[i],qubits[i+1]))

    for i in range(1,N-1,2): # odd sites
        circ.append(Ub(qubits[i],qubits[i+1]))

    for qubit in qubits:
        circ.append(Ua(qubit)) # e^A/2 = mult_i(exp(-gdt/4*Xi))
```

Exercise 7

In [ ]:
```python
def compute_magnetization(L, g, dt, t, approx='one', method='simulate', reps=1000):

    N = int(t/dt)

    simulator = cirq.Simulator()
    qubits = cirq.LineQubit.range(L)
    ops = [cirq.I(q) for q in qubits]
    #mgntz = cirq.Circuit([ops, ops]) #initializing the circuit in all ups
    mgntz = cirq.Circuit(ops) #initializing the circuit in all ups

    mgntz_list = [] # magnetization list
    Z_all = cirq.PauliSum.from_pauli_strings([cirq.Z(q) for q in qubits]) # generating the su

    for i in range(N+1):
        if method == 'run':
            expectations = []
            mgntz_measure = mgntz.copy()
            mgntz_measure.append(cirq.measure(*qubits, key='measure all'), strategy=cirq.Inse
            result = simulator.run(mgntz_measure, repetitions = reps)
            counts = result.histogram(key='measure all')

            for i in range(L):
                N0 = 0
                N1 = 0
                for integer, times  in counts.items():
                    if cirq.big_endian_int_to_bits(val = integer, bit_count = L)[i] == 0:
                        N0 += times
                    else:
                        N1 +=times
                # print(N0)
                # print(N1)
                expect = (N0-N1)/reps
                expectations.append(expect)
                print('expectation value for qubit:', i,'is' ,expect)
            mgntz_list.append( sum(expectations)/L)



        elif method == 'simulate':
            result = simulator.simulate(mgntz)
            state = result.final_state_vector

            state = state/np.linalg.norm(state) # in case not normalized for large system

            q_map = result.qubit_map
            z_magntization = Z_all.expectation_from_state_vector(state, q_map).real/L

            mgntz_list.append(z_magntization)

        print('iteration: ',i)
        if approx == 'one':
            evolve_basic(mgntz,qubits,g,dt)
        elif approx == 'second':
            evolve_symmetric(mgntz,qubits,g,dt)
        else:
            raise 'not a valid mode'

        #expectation = simulator.simulate_expectation_values(mgntz, observables= Z_all)[0].re
        #print('expectation: ',expectation)



    return mgntz_list
```

In [ ]:
```
L = 10
g = 2

dt = 0.25
t = 5
N = int(t/dt)

mgntz_list_first = compute_magnetization(L, g, dt, t, approx='one')
mgntz_list_second = compute_magnetization(L, g, dt, t, approx='second')

ts_simulate = np.linspace(0, (len(mgntz_list_first)-2)*dt , len(mgntz_list_first))
```

In [ ]:
```
ts = np.arange(0,t,dt)
magnetization = magn_exact_diagonalization(L,g,t=5,Npoints=N)
```

In [ ]:
```
plt.figure(dpi=(100))
plt.plot(ts, magnetization, label='Exact diagonalization')
plt.plot(ts_simulate, mgntz_list_first, label='First order trotter')
plt.plot(ts_simulate, mgntz_list_second, label='Second order trotter')
plt.title(' Magnetization vs time')
plt.xlabel('time ts')
plt.ylabel('magnetization')
plt.legend()
plt.grid()
plt.show()
```

Comments:

for dt = 0.25 the results are already quite good.

for dt = 0.5 it looked still woozy

Exercise 8

In [ ]:
```
# exercise 8

simulator = cirq.Simulator()
qubits = cirq.LineQubit.range(L)
ops = [cirq.I(q) for q in qubits]
#mgntz = cirq.Circuit([ops, ops]) #initializing the circuit in all ups
mgntz = cirq.Circuit(ops)

result = simulator.simulate(mgntz)
state = result.final_state_vector

print(state.conj())

state = state/np.linalg.norm(state) # in case not normalized for large system

print(len(state))
```

In [ ]:
```
def compute_losch(L, g, dt, t, approx='one', method='simulate', reps=1000):

    N = int(t/dt)

    simulator = cirq.Simulator()
    qubits = cirq.LineQubit.range(L)
    ops = [cirq.I(q) for q in qubits]

    losch = cirq.Circuit(ops)

    gs_zero, gs_one = np.zeros(2**L), np.zeros(2**L)

    print(len(gs_zero))

    gs_zero[0] = 1
    gs_one[-1] = 1

    losch_zero = [] # projection onto the zero ground state
    losch_one = [] # projection onto the one ground state

    losch_total = [] #stores the values of lambda

    for i in range(N+1):

        if method == 'run':

            losch_measure = losch.copy()
            losch_measure.append(cirq.measure(*qubits, key='measure all'), strategy=cirq.Inse
            result = simulator.run(losch_measure, repetitions = reps)
            counts = result.histogram(key='measure all')

            losch_total.append((counts[0] + counts[2**L - 1])/reps) #this should already be t


        elif method == 'simulate':
            result = simulator.simulate(losch)
            state = result.final_state_vector

            state = state/np.linalg.norm(state) # in case not normalized for large system

            print(len(state))

            projection0 = abs(np.inner(gs_zero.conj(), state))**2
            projection1 = abs(np.inner(gs_one.conj(), state))**2

            losch_zero.append(projection0)
            losch_one.append(projection1)

        print('iteration: ',i)
        if approx == 'one':
            evolve_basic(losch,qubits,g,dt)
        elif approx == 'second':
            evolve_symmetric(losch,qubits,g,dt)
        else:
            raise 'not a valid mode'

    if method == 'run':
        return losch_total
    elif method == 'simulate':
        losch_total = np.array(losch_zero) + np.array(losch_one)
        return losch_zero, losch_one, losch_total
```

In [ ]:

```
Ls = [6,8,10,12]

lambda0 =[]
lambda1 = []
lambdat = []

for L in Ls:
    print(L)
    losch_zero, losch_one, losch_total = compute_losch(L=L, g=2.0, dt=0.25, t=5)
    lambda0.append(losch_zero)
    lambda1.append(losch_one)
    lambdat.append(losch_total)
```

In [ ]:

```
plt.figure(dpi=(100))
for i, L in enumerate(Ls):
    plt.plot(ts_simulate, -np.log(lambdat[i])/L, label='Total loschmidt for L = {}'.format(L)
plt.title(' Loschmidt rate $\lambda(t)$ vs time')
plt.xlabel('time ts')
plt.ylabel('$\lambda$ rate')
plt.legend()
plt.grid()
plt.show()
```

In [ ]:

```
plt.figure(dpi=(100))
for i, L in enumerate(Ls):
    plt.plot(ts_simulate, -np.log(lambda0[i])/L, label='Zero loschmidt for L = {}'.format(L))
plt.title(' Loschmidt rate $\lambda_0(t)$ vs time')
plt.xlabel('time ts')
plt.ylabel('$\lambda$ rate')
plt.legend()
plt.grid()
plt.show()
```

```
In [ ]: plt.figure(dpi=(100))
        for i, L in enumerate(Ls):
            plt.plot(ts_simulate, -np.log(lambda1[i])/L, label='One loschmidt for L={}'.format(L))
        plt.title(' Loschmidt rate $\lambda_1(t)$ vs time')
        plt.xlabel('time ts')
        plt.ylabel('$\lambda$ rate')
        plt.legend()
        plt.show()
```

```
In [ ]: gs = np.arange(0.5,1.6,0.1)


        lambda0_gs =[]
        lambda1_gs = []
        lambdat_gs = []

        for g in gs:
            print(g)
            losch_zero, losch_one, losch_total = compute_losch(L=12, g=g, dt=0.25, t=5)
            lambda0_gs.append(losch_zero)
            lambda1_gs.append(losch_one)
            lambdat_gs.append(losch_total)
```

```
In [ ]: L = 12

        plt.figure(dpi=(100))
        for i, g in enumerate(gs):
            plt.plot(ts_simulate, -np.log(lambdat_gs[i])/L, label=' g = {}'.format(round(g,3)))
        plt.title(' Loschmidt rate $\lambda(t)$ vs time')
        plt.xlabel('time ts')
        plt.ylabel('$\lambda$ rate')
        plt.legend(loc='right')
        plt.grid(True)
        plt.show()
```

```
In [ ]: L = 12

        plt.figure(dpi=(100))
        for i, g in enumerate(gs):
            plt.plot(ts_simulate, -np.log(lambda0_gs[i])/L, label=' g = {}'.format(round(g,3)))
        plt.title(' Loschmidt rate $\lambda(t)$ vs time')
        plt.xlabel('time ts')
        plt.ylabel('$\lambda$ rate')
        plt.legend(loc='right')
        plt.show()
```

```
In [ ]: L = 12

        plt.figure(dpi=(100))
        for i, g in enumerate(gs):
            plt.plot(ts_simulate, -np.log(lambda1_gs[i])/L, label=' g = {}'.format(round(g,3)))
        plt.title(' Loschmidt rate $\lambda(t)$ vs time')
        plt.xlabel('time ts')
        plt.ylabel('$\lambda$ rate')
        plt.legend(loc='right')
        plt.show()
```

Exercise 9

```
In [ ]: L = 4

        simulator = cirq.Simulator(seed=2)
        qubits = cirq.LineQubit.range(L)
        ops = [cirq.I(q) for q in qubits]
```

```
#mgntz = cirq.Circuit([ops, ops]) #initializing the circuit in all ups
test = cirq.Circuit(ops)

reps = 100

#est.append([cirq.X(q) for q in qubits])
test.append([cirq.H(q) for q in qubits])
test.append(cirq.measure(*qubits[:2], key='measure all'), strategy=cirq.InsertStrategy.NEW)
print(test)
result1 = simulator.run(test, repetitions = reps)
counts = result1.histogram(key='measure all')
print(dict(sorted(counts.items())))
```

```
In [ ]: srt = dict(sorted(counts.items()))
        print(srt)
        print(list(srt.keys()))
        print(list(srt.values()))

        bitstrings = [cirq.big_endian_int_to_bits(val = k, bit_count = L) for k in srt.keys()]
        bitstrings[0]
```

```
In [ ]: list(srt.items())[-1][1]
```

```
In [ ]: #cirq.plot_state_histogram(samples, plt.subplot())
        #plt.show()



        expectations = []

        for i in range(len(qubits)):
            N0 = 0
            N1 = 0
            for number, times  in counts.items():
                #print(cirq.big_endian_int_to_bits(val = number, bit_count = L), times)
                if cirq.big_endian_int_to_bits(val = number, bit_count = L)[i] == 0:
                    N0 += times
                else:
                    N1 +=times

            print(N0)
            print(N1)
            expect = (N0-N1)/reps
            expectations.append(expect)
            print('expectation value for qubit:', i,'is' ,expect)

        sum(expectations)/L
```

```
In [ ]: L = 10
        g = 2

        dt = 0.25
        t = 5

        N = int(t/dt)

        mgntz_list_9= compute_magnetization(L, g, dt, t, approx='one', method='run', reps=1000)

        ts_simulate_9 = np.linspace(0, (len(mgntz_list_first)-2)*dt , len(mgntz_list_first))
```

```
In [ ]: plt.figure(dpi=(100))
        plt.plot(ts, magnetization, label='Exact diagonalization')
        plt.plot(ts_simulate_9, mgntz_list_9, label='Magnetization through measurements')
        plt.title(' Magnetization vs time')
        plt.xlabel('time ts')
        plt.ylabel('magnetization')
        plt.legend()
        plt.grid()
        plt.show()
```

How many repetitions:

for 10 and 100 the results are not so accurate, the minimum should be on the order of 10^3

In [ ]:
```
L = 10
g = 2

dt = 0.20
t = 5

N = int(t/dt)

losch_echo9 = compute_losch(L, g, dt, t, approx='one', method='run', reps=10000)
ts_losch_9 = np.linspace(0, (len(losch_echo9)-2)*dt , len(losch_echo9))
```

In [ ]:
```
plt.figure(dpi=(100))
plt.plot(ts_simulate_9, lambdat[2], label='Losch echo: simulations',)
plt.plot(ts_losch_9, losch_echo9,label='Losch echo: measurements',)
plt.title(' losch echo vs time')
plt.xlabel('time ts')
plt.ylabel('losch echo')
plt.legend()
plt.grid()
plt.show()
```

In [ ]:
```
from math import log, e
```

In [ ]:
```
test = []

for x in losch_echo9:
    test.append(-log(x)/L)
```

In [ ]:
```
plt.figure(dpi=(100))
plt.plot(ts_simulate_9, mgntz_list_9, label='Magnetization through measurements')
plt.plot(ts_losch_9, -np.log(losch_echo9)/L,label='Losch rate: measurements')
plt.title('Losch echo & magnetization')
plt.xlabel('time ts')
plt.ylabel('amplitude')
plt.legend()
plt.grid()
plt.show()
```

In [ ]:
```
bit_string0 = [0] * L
number = cirq.big_endian_bits_to_int(bit_string0)
bit_string1 = cirq.big_endian_int_to_bits(val = number, bit_count = L)
print( bit_string0 == bit_string1 ) # True
```

Exercise 17

In [ ]:
```
def U2_CUE(qubit, symbs):

    """
    symbs: symbs[0] = a, symbs[1] = b, symbs[2] = c
    applie U = Rz(a+c)Ry(b)Rz(a-c) to qubit
    """
    a = symbs[0]
    b = symbs[1]
    c = symbs[2]
    Rz1 = cirq.unitary(cirq.rz(c-a))
    Ry = cirq.unitary(cirq.ry(-2*b))
    Rz2 = cirq.unitary(cirq.rz(-(a+c)))

    U = cirq.MatrixGate(Rz2@Ry@Rz1)

    return U(qubit)
```

In [ ]:
```
class U2_CUE_2(cirq.Gate):
    def __init__(self,symbs):
        super(U2_CUE_2, self)
        self.a = symbs[0]
        self.b = symbs[1]
        self.c = symbs[2]

    def _num_qubits_(self):
        return 1

    def _unitary_(self):
        return np.array([
            [np.exp(1j*self.a)*np.cos(self.b), np.exp(1j*self.c)*np.sin(self.b)],
            [np.exp(-1j*self.c)*np.sin(self.b), np.exp(-1j*self.a)*np.cos(self.b)],
        ])

    def _circuit_diagram_info_(self, args):
        return f"U2({[np.exp(1j*self.a)*np.cos(self.b), np.exp(1j*self.c)*np.sin(self.b)]} \n
```

In [ ]:
```
name = 5
print(f"{name} \n emiliano")
```

In [ ]:
```
q_chain = cirq.LineQubit.range(2)
test = cirq.Circuit()

X = np.array([[0,1],[1,0]])

#Egate = cirq.MatrixGate(Rz@Ry)

symbs = [0,np.pi/4,0]
a = symbs[0]
b = symbs[1]
c = symbs[2]

#test.append([U2_CUE(qubit=q, symbs=symbs) for q in q_chain])

test.append([U2_CUE_2(symbs).on(q) for q in q_chain])

print(test)

result = simulator.simulate(test)
print('Bra-ket notation for the wavefunction:')
print(result.dirac_notation()) #this is correct

test.append(cirq.measure(qubits[0], key='0'), strategy=cirq.InsertStrategy.NEW)
reps = 100
result = simulator.run(test, repetitions = reps)
counts = result.histogram(key='0')
X = computeX(La=1, counts=counts, reps=reps)
X
```

In [ ]:
```
def hamming(s1,s2):

    """ calculate the hamming distance between bitstring 1 and bitstring 2
    defined as the number of bits that these differed on.
    Assumption: s1 and s2 are datatype = list
    """
    s1 = np.array(s1)
    s2 = np.array(s2)

    return sum((s1 + s2)%2)
```

In [ ]:
```
s1 = [0,0,1,1]
s2 = [1,1,0,1]

hamming(s1,s2)
```

In [ ]:
```
# chose a and c uniformly from 2pi
samples = 10
a,c = np.random.uniform(0,2*np.pi, size=(2,1))[:,0]
b = np.random.uniform(0,np.pi/2)
```

In [ ]:
```
def computeX(La, counts, reps=2000):

    """
    input:

    - counts: object obtained from calling cirq.Simulator().run().histogram(). Contains both
    ->Note: dictionary will only contain the values of those bitstrings that have !=0 probabi
    - La = length of the qubit system to be measured

    output:

    - X: to be used in the -log(X) formula to calculate the second Renyi Entropy
    """

    counts = dict(sorted(counts.items()))

    list_counts = list(counts.items()) # list of tuples. Each tuple contains the pair of (key
    # bitstrings = [cirq.big_endian_int_to_bits(val = k, bit_count = La) for k in counts.keys
    # probabilities = list(counts.values()) #Pu(SA) in the formula
    #print(list_counts)
    X = 0

    for s in itertools.product(list_counts, list_counts):
        #print(s)
        # s is a tuple containing (list_counts[i],list_counts[j]) for all possible combinatio
        sa = cirq.big_endian_int_to_bits(val = s[0][0], bit_count = La)
        sa_prime = cirq.big_endian_int_to_bits(val = s[1][0], bit_count = La)
        D = hamming(sa,sa_prime)
        #print(D)
        pu_sa = s[0][1]/reps
        pu_sa_prime = s[1][1]/reps
        X += (-0.5)**D *pu_sa*pu_sa_prime

    return X*2**La
```

In [ ]:
```
def Bell(n):
    q_chain = cirq.LineQubit.range(2)
    bell = cirq.Circuit()

    bell.append(cirq.H(q_chain[0]))
    bell.append(cirq.CNOT(q_chain[0],q_chain[1]))

    if n == 1:
        return bell, q_chain
    elif n==2:
        bell.append(cirq.Z(q_chain[1]))
        return bell, q_chain
    elif n==3:
        bell.append(cirq.X(q_chain[1]))
        return bell, q_chain
    elif n==4:
        bell.append(cirq.X(q_chain[1]))
        bell.append(cirq.Z(q_chain[0]))
        return bell, q_chain

bell, q_chain = Bell(1)
print(bell)

simulator = cirq.Simulator()
result = simulator.simulate(bell)
print('Bra-ket notation for the wavefunction:')
print(type(result.density_matrix_of()))

# bell.append(cirq.measure(q_chain[0], key='m'), strategy=cirq.InsertStrategy.NEW)
# reps = 2000
# result = simulator.run(bell, repetitions = reps)
# counts = result.histogram(key='m')
# print(counts)

# -log(computeX(La=1, counts=counts, reps=2000),2)
```

In [ ]:
```
# functions for analytical calculations

def partial_trace(rho,dimA,dimB):

    rho_prime = rho.reshape(dimA,dimB,dimA,dimB)
    rhoA = np.trace(rho_prime, axis1=1, axis2=3)
    rhoB = np.trace(rho_prime, axis1=0, axis2=2)

    return (rhoA, rhoB)


def Simulate_RenyiEntropy(circ, La, L,simulator=cirq.Simulator()):

    result = simulator.simulate(circ)
    rho = result.density_matrix_of()
    Lb = L-La
    rhol,_ = partial_trace(rho,2**La,2**Lb)

    S = -log(np.trace(rhol@rhol),2)

    return S
```

In [ ]:
```
np.log
```

In [ ]:
```
def renyi_entropy(L, La, dt=0.1, t=5, g=2, instances=200, method='run', reps=2000):

    assert La <= L, 'Subsystem should be at most the size of the total system'

    seed = random.seed(2)

    N = int(t/dt)
    simulator = cirq.Simulator(seed=seed)
    qubits = cirq.LineQubit.range(L)
    ops = [cirq.I(q) for q in qubits] # all identities
    circuit = cirq.Circuit(ops) # initializing the circuit in all ups

    rentropy = [] # renyi entropy list depending on time

    for j in tqdm(range(N+1)): #this will simulate the time
        if method == 'run':
            X = 0
            for i in range(instances): # we will have 200 instances
                # simulating the a,b,c for this instance
                a,c = np.random.uniform(0,2*np.pi, size=(2,L))
                b = np.random.uniform(0,np.pi/2, L)
                symbs = [a,b,c]
                circuit_measure = circuit.copy()
                for i,q in enumerate(qubits[:La]):
                    circuit_measure.append(U2_CUE(qubit=q, symbs=list(list(zip(*symbs))[i])))
                    #circuit_measure.append(U2_CUE_2(symbs=list(list(zip(*symbs))[i])).on(q))
                circuit_measure.append(cirq.measure(*qubits[:La], key='measure all'), strateg
                result = simulator.run(circuit_measure, repetitions = reps)
                counts = result.histogram(key='measure all')

                X += computeX(La=La, counts=counts, reps=reps)
                    #print(X)

            S2 = -log(X/instances,2)
            rentropy.append(S2)

        elif method == 'simulate':
            rentropy.append(Simulate_RenyiEntropy(circuit, La, L))

        evolve_basic(circuit,qubits,g,dt)

    return rentropy
```

```
In [ ]: testing = [1,2,3,4,5,6]
        A =  3
        testing[:A]
```

```
In [ ]: L = 2

        a,c = np.random.uniform(0,2*np.pi, size=(2,L))
        b = np.random.uniform(0,np.pi/2, L)

        symbs = [a,b,c]

        print(symbs)

        for i in range(L):
            print(list(list(zip(*symbs))[i]))
```

```
In [ ]: dt = 0.25
        test2 = renyi_entropy(L=2, La=1, dt=0.25, t=5, instances=500, reps=2000)
        ts_17 = np.linspace(0, (len(test2)-2)*dt , len(test2))
```

```
In [ ]: import warnings
        warnings.filterwarnings('ignore')
        dt = 0.25
        test2_simulate = renyi_entropy(L=2, La=1, dt=0.25, t=5, method='simulate')
        ts_17_simulate = np.linspace(0, (len(test2_simulate)-2)*dt , len(test2_simulate))
```

```
In [ ]: plt.figure(dpi=(100))
        plt.plot(ts_17, test2, '--',label='randomized measurements')
        plt.plot(ts_17_simulate,test2_simulate,label='exact simulation')
        plt.title('Second Renyi Entropy')
        plt.xlabel('Time t')
        plt.ylabel('$S^2(t)$')
        plt.legend()
        plt.grid()
        plt.show()
```

```
In [ ]: renyi_entropy = _
```

```
In [ ]: renyi_entropy
```

```
In [ ]: np.savetxt('exercise17_test1.txt', renyi_entropy)
```

```
In [ ]:
```

B. Bibliography

[1] M. Knap, “Simulating Quantum Many-Body Dynamics on a Current Digital Quantum Computer”, (2022), https://www.ph.tum.de/academics/org/labs/fopra/?language=en (visited on 07/02/2022).

[2] S. Khatri, K. Sharma, and M. M. Wilde, “Information-theoretic aspects of the generalized amplitude-damping channel”, *Physical Review A* **102**, 10.1103/physreva.102.012401 (2020), https://doi.org/10.1103%2Fphysreva.102.012401.

[3] A. Elben, B. Vermersch, C. F. Roos, and P. Zoller, “Statistical correlations between locally randomized measurements: A toolbox for probing entanglement in many-body quantum states”, *Physical Review A* **99**, 10.1103/physreva.99.052323 (2019), https://doi.org/10.1103%2Fphysreva.99.052323.