FYS5419 - Project 1

Insert Name

2025-06-01

Contents

1 Part a)

1.1 One-qubit Operations

A single qubit state $|\psi\rangle \in {}^{\P}\mathcal{H} \cong \mathbb{C}^2$ can be represented as a 2D complex vector

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle,$$

where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi)$ are the polar and azimuthal angle in the Bloch sphere, respectively. The computational basis states $|0\rangle$ and $|1\rangle$ are given by

$$|0\rangle = |\uparrow_{\hat{\mathbf{z}}}\rangle := \begin{bmatrix}1\\0\end{bmatrix}, \quad |1\rangle = |\downarrow_{\hat{\mathbf{z}}}\rangle := \begin{bmatrix}0\\1\end{bmatrix},$$

corresponding to spin-up and spin-down along the z-axis. Using the Numpy package in Python, a single qubit can be represented with the function:

```
import numpy as np
def create_qubit(theta: float, phi: float) -> np.ndarray:
```

```
return np.array(
  [np.cos(theta / 2), np.exp(1j * phi) * np.sin(theta / 2)],
  dtype=np.complex128
)
```

A more general Python implementation of n-ary qubit state is shown in Program 18

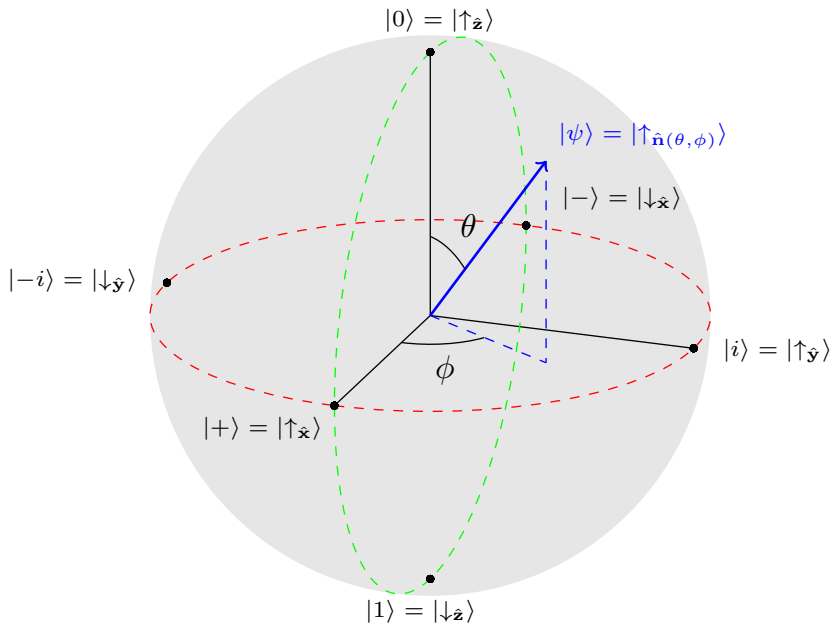

Figure 1: Bloch sphere representation of a qubit.

1.1.1 Pauli Gates

In the standard basis $\{\ |0\rangle,\ |1\rangle\ \}$, the Pauli gates are given by

$$\begin{aligned}
\hat{X} =& \hat{\sigma}_x = |0\rangle\langle 1| + \langle 1|\,|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\
\hat{Y} =& \hat{\sigma}_y = -i\,|0\rangle\langle 1| + i\,|1\rangle\,|0\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \\
\hat{Z} =& \hat{\sigma}_x = |0\rangle\langle 0| - |1\rangle\langle 1| = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.
\end{aligned}$$

Geometrically, the Pauli gates generate half-turns about their respective axes. This can be understood in the terms of the rotation gate $\hat{R}_{\hat{\mathbf{n}}}(\theta)$, which rotates a qubit by an angle θ around a unit axis $\hat{\mathbf{n}} \in \mathbb{S}^2$ in $\mathbb{R}^3$:

$$\hat{R}_{\hat{\mathbf{n}}}(\theta) = \exp\left(-i\frac{\theta}{2}\hat{\mathbf{n}}\cdot\boldsymbol{\sigma}\right) = \cos\left(\frac{\theta}{2}\right)\hat{I}_2 - i\sin\left(\frac{\theta}{2}\right)\hat{\mathbf{n}}\cdot\boldsymbol{\sigma},$$

where $\boldsymbol{\sigma} = (\hat{\sigma}_x, \hat{\sigma}_y, \hat{\sigma}_z)$ is the Pauli vector, and $\hat{I}_2$ is the identity operator. In particular for $\theta = \pi$, we obtain

$$\hat{R}_{\hat{\mathbf{n}}}(\pi) = \cos\left(\frac{\pi}{2}\right)\hat{I}_2 - i\sin\left(\frac{\pi}{2}\right)\hat{\mathbf{n}}\cdot\boldsymbol{\sigma} = -i\hat{\mathbf{n}}\cdot\boldsymbol{\sigma},$$

which shows that each Pauli gate (up to a global phase factor $-i$) is equivalent to a π-rotation about its respective axis, i.e.

$$\hat{R}_x(\pi) = -i\hat{X}, \quad \hat{R}_y(\pi) = -i\hat{Y}, \quad \hat{R}_z(\pi) = -i\hat{Z},$$

Pauli-X gate Applying the Pauli-X gate to a qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ with $\alpha, \beta \in \mathbb{C}$ with $|\alpha|^2 + |\beta|^2 = 1$ gives

$$\begin{aligned}\hat{X}(\alpha|0\rangle + \beta|1\rangle) =& (|0\rangle\langle 1| + |1\rangle\langle 0|)(\alpha|0\rangle + \beta|1\rangle) \\ =& \beta|1\rangle + \alpha|0\rangle,\end{aligned}$$

meaning that it flips the qubit. In Python, the Pauli-X gate can be implemented in the following way using Numpy:

```
import numpy as np
def apply_x_gate(qubit: np.ndarray) -> np.ndarray:
  x_gate = np.array([[0., 1.], [1., 0.]], dtype=np.complex128)
  return qubit @ x_gate

# Example
qubit = create_qubit(theta=0, phi=0) # |0> state: [1., 0.]
qubit_x_gate = apply_x_gate(qubit) # Returns |1> state: [0., 1.]
```

Pauli-Y gate Applying the Y gate to a qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ yields

$$\begin{aligned}\hat{Y}(\alpha|0\rangle + \beta|1\rangle) =& (-i|0\rangle\langle 1| + i|1\rangle\langle 0|)(\alpha|0\rangle + \beta|1\rangle) \\ =& i(\alpha|1\rangle - \beta|0\rangle),\end{aligned}$$

whichs shows that it flips the qubit and multiplies the phase by i. In Python, the Pauli Y-gate can be implemented in the following way using Numpy:

```
import numpy as np
def apply_y_gate(qubit: np.ndarray) -> np.ndarray:
  y_gate = np.array([[0., -1.j], [1.j, 0.]], dtype=np.complex128)
  return qubit @ y_gate

# Example
qubit = create_qubit(theta=0, phi=0) # |0> state: [1., 0.]
qubit_y_gate = apply_y_gate(qubit) # Returns i|1> state: [0.+0.j 0. -1.j]
```

Pauli-Z gate Applying the Pauli-Z gate to a qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ yields

$$\begin{aligned}\hat{Z}(\alpha|0\rangle + \beta|1\rangle) =& (|0\rangle\langle 0| - |1\rangle\langle 1|)(\alpha|0\rangle + \beta|1\rangle) \\ =& \alpha|0\rangle - \beta|1\rangle = \alpha|0\rangle + e^{i\pi}\beta|1\rangle,\end{aligned}$$

showing that it multiplies the phase of $|1\rangle$ by -1. In particular, the Z gate flips the $\hat{X}$ basis $\{|+\rangle, |-\rangle\}$, where

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

In Python, the Pauli Z-gate can be implemented in the following way using Numpy:

```
import numpy as np
def apply_z_gate(qubit: np.ndarray) -> np.ndarray:
  z_gate = np.array([[1., 0.], [0., -1.]], dtype=np.complex128)
  return qubit @ z_gate

# Example
qubit = create_qubit(theta=np.pi/2, phi=0) # |+> state: [0.70710678, 0.70710678]
qubit_z_gate = apply_z_gate(qubit) # Returns | -> state: [0.70710678, -0.70710678]
```

1.1.2 Hadamard gate

The Hadamard gate is given by

$$\hat{H} := \frac{1}{\sqrt{2}}(\hat{X} + \hat{Z}) = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$

Geometrically, it generates a half-turn about the diagonal $(x+z)$-axis on the Bloch sphere. This can be expressed in terms of the rotation operator $\hat{R}_{\hat{\mathbf{n}}}(\theta)$ about $\hat{\mathbf{n}} = \frac{1}{\sqrt{2}}(1,0,1)$, which corresponds to the diagonal $(x+z)$-axis:

$$\begin{aligned}\hat{R}_{\hat{\mathbf{n}}}(\theta) =& \exp\left(-i\frac{\theta}{2}\frac{1}{\sqrt{2}}(\hat{X} + \hat{Z})\right) \\ =& \cos\left(\frac{\theta}{2}\right)\hat{I}_2 - i\sin\left(\frac{\theta}{2}\right)\frac{1}{\sqrt{2}}(\hat{X} + \hat{Z})\end{aligned}$$

Particularly for $\theta = \pi$, we find

$$\hat{R}_{\hat{\mathbf{n}}}(\pi) = -\frac{i}{\sqrt{2}}(\hat{X} + \hat{Z}) = -i\hat{H}$$

The Hadamard gate transforms the $\hat{Z}$-basis $\{|0\rangle, |1\rangle\}$ to the $\hat{X}$-basis $\{|+\rangle, |-\rangle\}$:

$$\hat{H}\,|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$$
$$\hat{H}\,|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$$

This transformation creates quantum superposition from the standard computational basis.

In Python, the Hadamard gate can be implemented in the following way using Numpy:

```
import numpy as np
def apply_hadamard(qubit: np.ndarray) -> np.ndarray:
  h = (1/np.sqrt(2)) * np.array([[1., 1.], [1., -1.]], dtype=np.complex128)
  return qubit @ h

# Example
qubit = create_qubit(theta=0, phi=0) # |0> state: [0., 0.]
qubit_h = apply_hadamard(qubit) # Returns |+> state: [0.70710678, 0.70710678]
```

1.1.3 Phase shift-gate

The general phase shift gate is given by

$$\hat{P}(\nu) := |0\rangle\langle 0| + e^{i\nu}\,|1\rangle\langle 1| = \hat{Z}^{\nu/\pi} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\nu} \end{bmatrix},$$

where the Pauli-Z power gate is defined as

$$\hat{Z}^t := e^{-i\frac{\pi}{2}t(\hat{Z}-\hat{I}_2)} \sim \hat{R}_z(\pi t)$$

which is equivalent to $\hat{R}_z(\pi t)$ up to a global phase. The phase shift gate induces a relative phase to a qubit $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$:

$$\begin{aligned}\hat{P}(\nu)\,|\psi\rangle =& (\,|0\rangle\langle 0| + e^{i\nu}\,|1\rangle\langle 1|\,)(\alpha\,|0\rangle + \beta\,|1\rangle\,) \\ =& \alpha\,|0\rangle + e^{i\nu}\beta\,|1\rangle\end{aligned}$$

For $\nu = \pi/2$, we get the square root of $\hat{Z}$, which is also known as the phase gate, or S gate:

$$\hat{S} = \hat{Z}^{1/2} = \hat{P}(\pi/2) = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

The S gate transforms the $\hat{X}$ basis $\{\ |+\rangle,\ |-\rangle\ \}$ to the basis $\{\ |i\rangle,\ |-i\rangle\ \}$:

$$\hat{S}\,|+\rangle = (|0\rangle\langle 0| + i\,|1\rangle\langle 1|)\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle) = |i\rangle$$
$$\hat{S}\,|-\rangle = (|0\rangle\langle 0| + i\,|1\rangle\langle 1|)\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle) = |-i\rangle$$

In particular, the composition $\hat{S}\hat{H}$ transforms the $\hat{Z}$ basis $\{\ |0\rangle,\ |1\rangle\ \}$ to the $\{\ |i\rangle,\ |-i\rangle\ \}$:

$$\hat{S}\hat{H}\,|0\rangle = \hat{S}\,|+\rangle = |i\rangle$$
$$\hat{S}\hat{H}\,|1\rangle = \hat{S}\,|-\rangle = |-i\rangle$$

In Python, the S gate can be implemented in the following way using Numpy:

```
import numpy as np
def apply_s_gate(qubit: np.ndarray) -> np.ndarray:
  s_gate = np.array([[1., 0.], [0., 1.j]], dtype=np.complex128)
  return qubit @ s_gate

# Example
qubit = create_qubit(theta=np.pi/2, phi=0) # |+> state: [0.70710678, 0.70710678]
qubit_h = apply_s_gate(qubit) # Returns |i> state: [0.70710678, 0.70710678j]
```

1.2 Bell states

The Bell basis forms an orthonormal basis for the four-dimensional Hilbert space ¶$\mathcal{H}^{\otimes 2}$ and consists of the Bell states

$$|\Phi_\pm\rangle := \frac{1}{\sqrt{2}}(\,|00\rangle \pm |11\rangle\,)$$
$$|\Psi_\pm\rangle := \frac{1}{\sqrt{2}}(\,|01\rangle \pm |10\rangle\,).$$

The Bell states can be created from the computational basis by applying a Hadamard gate $\hat{H}_0$ to the first qubit, followed by a controlled NOT gate, $\mathrm{CNOT}_{0,1}$:

$$\mathrm{CNOT}_{0,1}\,\hat{H}_0\,|00\rangle = |\Phi_+\rangle$$
$$\mathrm{CNOT}_{0,1}\,\hat{H}_0\,|01\rangle = |\Psi_+\rangle$$
$$\mathrm{CNOT}_{0,1}\,\hat{H}_0\,|10\rangle = |\Psi_-\rangle$$
$$\mathrm{CNOT}_{0,1}\,\hat{H}_0\,|11\rangle = |\Phi_-\rangle\,,$$

where

- $\hat{H}_0 = \hat{H} \otimes \hat{I}$, and
- $\text{CNOT}_{0,1} = |0\rangle\langle 0| \otimes \hat{I} + |1\rangle\langle 1| \otimes \hat{X}$

A Python function for creating Bell states is shown in Program 2. This function uses methods of the NQubitState class to apply the Hadamard gate and controlled X gate (see Program 19 and Program 21 for their Python implementation).

```
def create_bell_state(state: int) -> NQubitState:
  """
  Create a qubit in a Bell state using Hadamard and CNOT gates.

  Args:
    state (int): Bell state to initialize the qubit in. Can be 0, 1, 2, or 3.
    - 0 initializes |00> resulting in | \Phi+> = (1/2)|00> + (1/2)|11>
    - 1 initializes |01> resulting in | \Psi+> = (1/2)|01> + (1/2)|10>
    - 2 initializes |10> resulting in | \Psi -> = (1/2)|00> - (1/2)|11>
    - 3 initializes |11> resulting in | \Phi -> = (1/2)|01> - (1/2)|10>
  """

  bell_state = NQubitState(2, basis_state=state)
  bell_state.hadamard_gate(0)  # Apply Hadamard gate to first qubit
  bell_state.cnot_gate(0, 1)  # Apply CNOT gate

  return bell_state
```

Figure 2: Python function for constructing Bell states using the NQubitState class.

1.2.1 Binary Qubit Measurement

Program 3 shows Python code for simulating sequential measurements on a binary qubit state using the NQubitState class. The Python implementation for simulating quantum measurements is detailed in Program 20. In this example, the qubit state is initialized to the Bell state $|\Psi_+\rangle$, followed by two operations:

1. A Hadamard gate $\hat{H}_1 = \hat{I} \otimes \hat{H}$ applied to the second qubit
2. A controlled-NOT gate $\text{CNOT}_{1,0}$ targeting the second qubit.

The corresponding quantum circuit for this operation is shown in Figure 4. The resulting binary state $|\Lambda\rangle$ can be expressed as follows:

$$
\begin{aligned}
|\Lambda\rangle &= \text{CNOT}_{1,0}\, \hat{H}_1\, |\Psi_+\rangle \\
&= \begin{bmatrix} 1&0&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \\ 0&1&0&0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1&1&0&0 \\ 1&-1&0&0 \\ 0&0&1&1 \\ 0&0&1&-1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \\
&= \frac{1}{2} \begin{bmatrix} 1&1&0&0 \\ 0&0&1&-1 \\ 0&0&1&1 \\ 1&-1&0&0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix} \\
&= \frac{1}{2} \big(|00\rangle + |01\rangle + |10\rangle - |11\rangle \big).
\end{aligned}
$$

This shows that $|\Lambda\rangle$ is a superposition of the binary computation basis states, with equal magnitude of the probability amplitudes. Specifically, the probability of measuring any of the four computation basis states is

$$\Pr(|00\rangle) = \Pr(|01\rangle) = \Pr(|10\rangle) = \Pr(|11\rangle) = \left|\frac{1}{2}\right|^2 = \frac{1}{4}.$$

Thus, we expect the measurement probabilities to be approximately equal for each basis state. Figure 5 shows the results for experimental measurements for $|\Lambda\rangle$ conducted in sequential order with 1000 shots. These results correspond to the following observed probabilities

$$\Pr(|00\rangle) = \frac{241}{1000} = 0.241, \quad \Pr(|01\rangle) = \frac{263}{1000} = 0.263,$$
$$\Pr(|10\rangle) = \frac{242}{1000} = 0.242, \quad \Pr(|11\rangle) = \frac{254}{1000} = 0.254$$

The probabilities are relatively close to the theoretical value of 0.25 for each state, as expected.

```
shots = 1000

# Initialize Bell state
selected_state = 1 # | \Psi+>
bell_state = create_bell_state(selected_state)

# Apply Hadamard and CNOT gates
bell_state.hadamard_gate(1)
bell_state.cnot_gate(1,0)

counts = bell_state.measure([0,1], shots)
```

Figure 3: Python code for simulating quantum measurements on a binary qubit state.

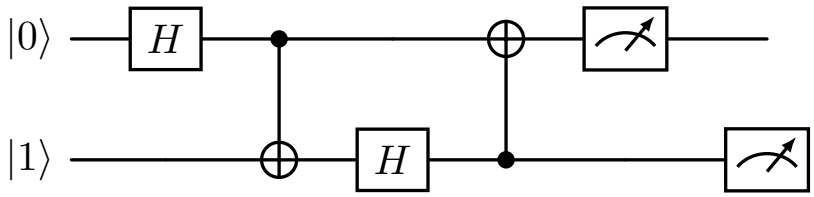

Figure 4: Corresponding quantum circuit for simulating quantum measurements on a binary qubit state.

2 Part b) One-Qubit Hamiltonian Eigenvalue Problem

In terms of the Pauli basis $\{\hat{I}, \hat{X}, \hat{Y}, \hat{Z}\}$, the total Hamiltonian $\hat{H}_1 = \hat{H}_0 + \hat{H}_{\mathrm{I}}$ can be written

$$\hat{H} = \underbrace{(\mathcal{E} + \lambda c)\hat{I}_2}_{=\hat{H}_0} + \underbrace{(\Omega + \lambda\omega_z)\hat{Z} + \lambda\omega_x\hat{X}}_{=\hat{H}_I} \tag{1}$$

Figure 5: Measurement results for the Bell state $|\Psi_+\rangle$ after applying a Hadamard gate to the first qubit, followed by a CNOT gate, measured over 1000 shots.

where the parameters are defined as

$$\mathcal{E} = \frac{E_1 + E_2}{2}, \quad \Omega = \frac{E_1 - E_2}{2}, \quad c = \frac{V_{11} + V_{22}}{2}, \quad \omega_z = \frac{V_{11} - V_{22}}{2}, \quad \omega_x = V_{12} = V_{21}.$$

To find the eigenvalues of $\hat{H}_1$, we must solve the characteristic equation $\det(\hat{H}_1 - E\hat{I}) = 0$. The identity matrix $\hat{I}$ contributes a constant energy shift $\mathcal{E} + \lambda c$, meaning the relevant part of the Hamiltonian for eigenvalue calculation is

$$\hat{H}' = (\Omega + \lambda\omega_z)\hat{Z} + \lambda\omega_x\hat{X}, \tag{2}$$

or in matrix form

$$\hat{H}' = \begin{bmatrix} \Omega + \lambda\omega_z & \lambda\omega_x \\ \lambda\omega_x & -(\Omega + \lambda\omega_z) \end{bmatrix} \tag{3}$$

To find the eigenvalues, we solve the characteristic equation $\det(\hat{H}' - E'\hat{I}) = 0$:

$$\begin{aligned} 0 =& \begin{vmatrix} \Omega + \lambda\omega_z - E' & \lambda\omega_x \\ \lambda\omega_x & -(\Omega + \lambda\omega_z) - E' \end{vmatrix} \\ =& (\Omega + \lambda\omega_z - E')(-(\Omega + \lambda\omega_z) - E') - (\lambda\omega_x)^2 \\ =& -(\Omega + \lambda\omega_z - E')(\Omega + \lambda\omega_z + E') - (\lambda\omega_x)^2 \\ =& -(\Omega + \lambda\omega_z)^2 + E'^2 - (\lambda\omega_x)^2 \end{aligned}$$

Rearranging yields

$$E'^2 = (\Omega + \lambda\omega_z)^2 + (\lambda\omega_x)^2. \tag{4}$$

Taking the square root, we find the reduced eigenvalues

$$E'_\pm = \pm\sqrt{(\Omega + \lambda\omega_z)^2 + (\lambda\omega_x)^2} \tag{5}$$

Adding the constant shift $\mathcal{E} + \lambda c$, we finally arrive at

$$E_\pm = \mathcal{E} + \lambda c \pm \sqrt{(\Omega + \lambda\omega_z)^2 + (\lambda\omega_x)^2} \tag{6}$$

The Python code for computing the eigenvalues of $\hat{H}_1$ is provided in Program 29. The resulting eigenvalues as a function of the interaction strength $\lambda \in [0, 1]$ are plotted in Figure 6, using the following parameters:

$$E_1 = 0, \quad E_2 = 4, \quad V_{11} = V_{22} = 3, \quad V_{12} = V_{21} = 0.2 \tag{7}$$

The eigenvalue structure of $\hat{H}_1$ exhibits an avoided crossing at around $\lambda = 2/3$, which is a common behaviour in two-level quantum systems. In the non-interacting limit $\lambda = 0$, the Hamiltonian reduces to the diagonal form $\hat{H}_1 = \hat{H}_0$, corresponding to two uncoupled states. In this regime, the system remains in the computational basis, with eigenstates $|0\rangle$ and $|1\rangle$ having eigenvalues $E_1 = 0$ and $E_2 = 4$, respectively.

As the interaction strength λ increases, the off-diagonal coupling terms $V_{12} = V_{21}$ induce mixing between the computational basis states. Around $\lambda = 2/3$, the eigenstates become maximally mixed and they swap their dominant character: the lower eigenstate transition from being predominantly $|0\rangle$ to primarily $|1\rangle$, while the higher eigenstate undergoes the reverse transition. Beyond this avoided crossing point, $\lambda > 2/3$, the eigenstates continue to separate in energy, with their dominant character exchanged.

3 Part c) VQE for One-Qubit Hamiltonian

In terms of a qubit state $|\psi\rangle$, the expectation value for the Hamiltonian (1) is given by

$$\langle H_1\rangle_\psi = \langle\psi|\hat{H}_1|\psi\rangle = (\mathcal{E} + \lambda c)\underbrace{\langle\psi|\hat{I}_2|\psi\rangle}_{=1} + (\Omega + \lambda\omega_z)\langle\psi|\hat{Z}|\psi\rangle + \lambda\omega_x\langle\hat{X}\rangle$$

To compute the expectation value $\langle H_1\rangle_\psi$, a quantum circuit must be configured with two measurement bases: one for the $\hat{Z}$-term and another for the $\hat{X}$-term. The expectation value of $\hat{Z}$ is measured directly in the computational basis, while the $\hat{X}$-term requires a basis transformation via a Hadamard gate before measurement. To implement the variational quantum eigensolver (VQE), each qubit is prepared in an ansatz state $|\psi(\theta, \phi)\rangle$, parametrized by θ and ϕ using rotations gates:

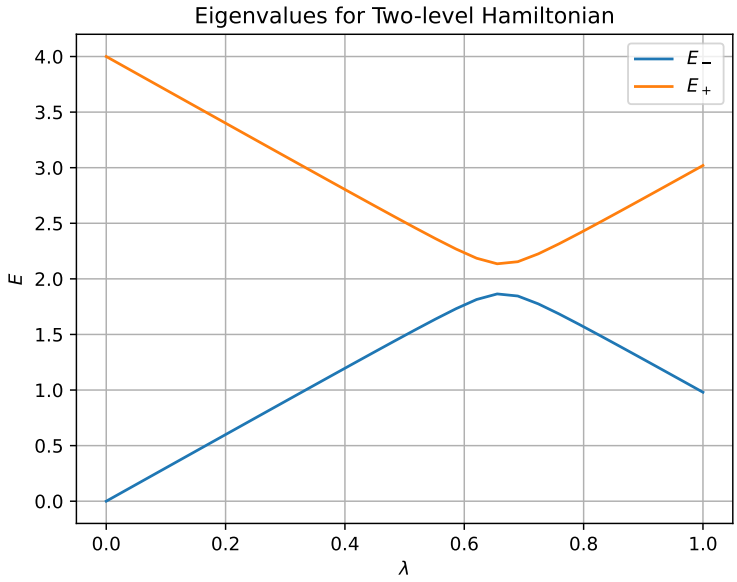

Figure 6: Exact eigenvalues of the one-qubit Hamiltonian as a function of the interaction strength λ.

$$|\psi(\theta,\phi)\rangle = \hat{R}_y(\phi)\hat{R}_x(\theta)\,|0\rangle \tag{8}$$

These variational parameters are iteratively optimized using classical algorithms to minimize $\langle H_1(\theta,\phi)\rangle$, approximating the ground-state energy of $\hat{H}_1$. Figure 7 illustrates the quantum circuit for this VQE setup. The Python function for computing the $\hat{Z}$ and $\hat{X}$ terms of $\langle H_1(\theta,\phi)\rangle$ using quantum circuit logic is given in Program 30. The gradient of $\langle H_1(\theta,\phi)\rangle$ can be computed using the parameter shift rule:

$$\begin{aligned}
\frac{\partial}{\partial\theta}\langle H(\theta,\phi)\rangle &= \frac{1}{2}\left[\hat{H}_1\left(\theta+\frac{\pi}{2},\phi\right) - \hat{H}_1\left(\theta-\frac{\pi}{2},\phi\right)\right] \\
\frac{\partial}{\partial\phi}\langle H(\theta,\phi)\rangle &= \frac{1}{2}\left[\hat{H}_1\left(\theta,\phi+\frac{\pi}{2}\right) - \hat{H}_1\left(\theta,\phi-\frac{\pi}{2}\right)\right]
\end{aligned}$$

With an analytic gradient expression, we can minimize $\langle H_1(\theta,\phi)\rangle$ using classical gradient descent optimization. A Python function for the parameter shift rule is given in Program 25, while a function for minimizing $\langle H_1(\theta,\phi)\rangle$ using gradient descent is provided in Program 26.

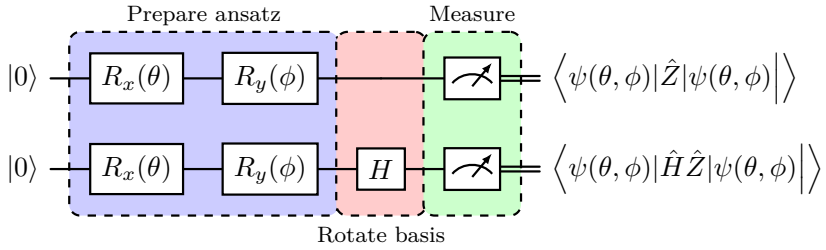

Figure 7: Quantum circuit for the variational quantum eigensolver (VQE) applied to $\langle\hat{H}_1(\theta,\phi)\rangle$.

Figure 8 shows the minimzed VQE energies $\langle H_1(\theta,\phi)\rangle$. The estimated energies closely align with the analytical values, demonstrating the ability of VQE to approximating ground-state

energies. However, repeated runs sometimes yield deviations, likely due to suboptimal random initialization of variational parameters. This can be solved by implementing a more informed method of initalizing the variational parameters.

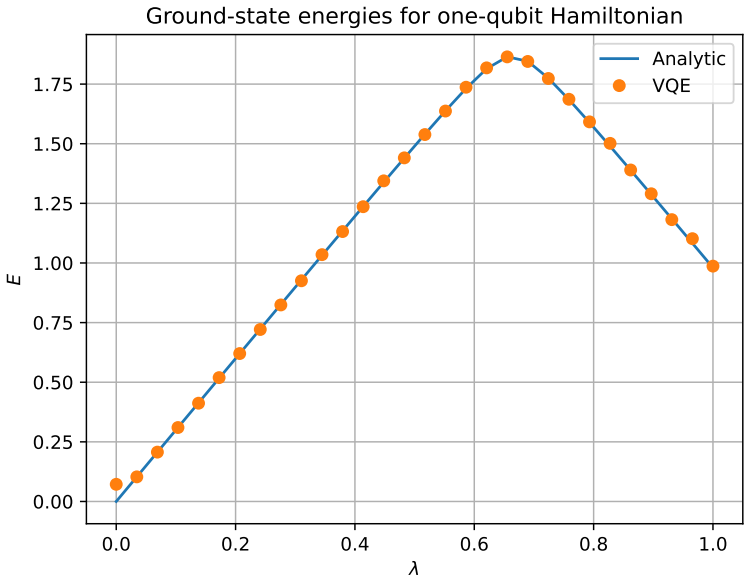

Figure 8: A plot of VQE energies of $\langle H_1(\theta, \phi)\rangle$, optimized using gradient descent with Adaptive Moment Estimation (ADAM) over a maximum of 500 epochs. The expectation values $\langle H_1\rangle$ were computed from quantum circuit measurements over 1000 shots.

4 Part d) Two-Qubit Hamiltonian Eigenvalue Problem

Program 31 shows the Python code for computing the eigenvalues of the two-qubit Hamiltonian $\hat{H}_2 = \hat{H}_0 + \lambda \hat{H}_\mathrm{I}$ with matrix representation

$$\hat{H}_2 = \begin{bmatrix} \epsilon_{00} + H_z & 0 & 0 & H_x \\ 0 & \epsilon_{01} - H_z & H_x & 0 \\ 0 & H_x & \epsilon_{10} - H_z & 0 \\ H_x & 0 & 0 & \epsilon_{11} + H_z \end{bmatrix}$$

The resulting plot of the eigenvalues as a function of the interaction strength $\lambda \in [0, 1]$ is shown in Figure 9. The following parameters were used in the calculation:

$$H_x = 2.0, \quad H_z = 3.0, \quad \epsilon_{00} = 0.0, \quad \epsilon_{01} = 2.5, \quad \epsilon_{10} = 6.5, \quad \epsilon_{11} = 7.0 \tag{9}$$

The two lowest energy states, corresponding to eigenvalues E_1 and E_2, display an avoided crossing around $\phi \approx 0.4$. This indicates strong mixing between the energy states, with a transition between dominant basis states occuring at around $\phi \approx 0.4$. This behaviour resembles the two-level system treated in Part b).

In contrast, the the two upper energy states, corresponding to eigenvalues E_3 and E_4, do not experience level repulsion. Instead, E_4 increases monotonically while E_3 decreases monotonically

as λ increases. This suggests that these higher-energy states are less affected by the interaction mechanism inducing mixing in the lower states.

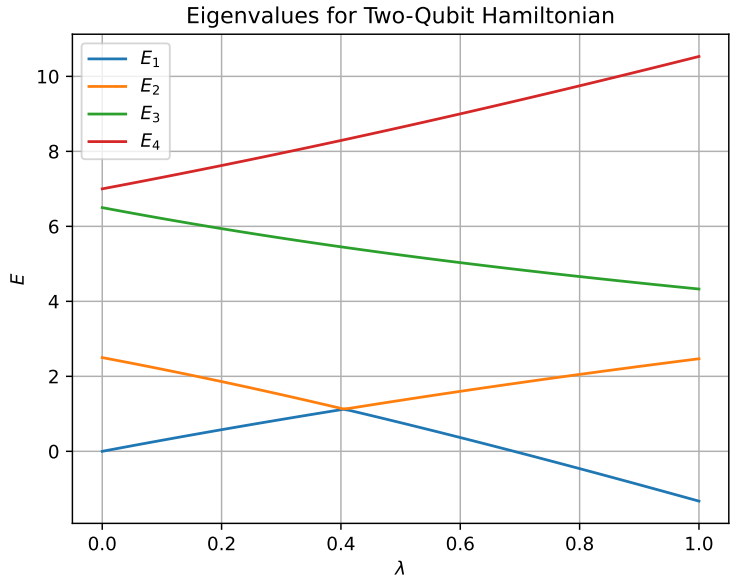

Figure 9: Exact eigenvalues of the two-qubit Hamiltonian as a function of the interaction strength λ.

Figure 10 shows the von Neumann entropy S of the ground state of $\hat{H}_2$, computed from its reduced density matrix, as a function of the interaction strength $\lambda \in [0, 1]$. In the non-interaction limit, $\lambda = 0$, the entropy is zero corresponding to a pure state in the computational basis. As λ increases, the entropy grows smoothly, indicating a gradual increase in entanglement. A sudden jump in entropy occurs around $\lambda \approx 0.4$, coinciding with the avoided crossing between the two lowest energy states. Beyond this region, the system transitions into a more entangled state, with entropy continuing to increase as interactions strengthen.

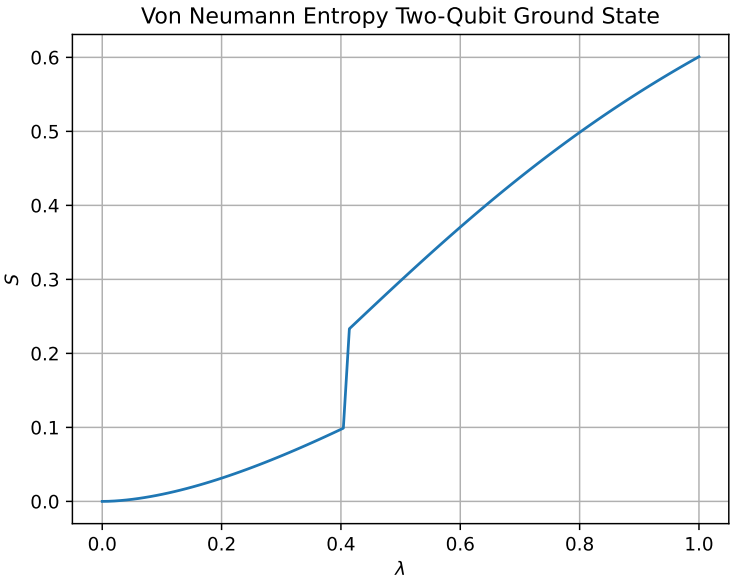

Figure 10: Von Neumann entropy for the ground state of the two-qubit Hamiltonian as a function of the interaction strength λ.

5 Part e) VQE for Two-Qubit Hamiltonian

In terms of Pauli strings, the parametrized expectation of the two-qubit Hamiltonian $\hat{H}_2$ is given by

$$\begin{aligned}\langle H_2(\boldsymbol{\theta})\rangle =& \epsilon_{II} \underbrace{\langle\psi(\boldsymbol{\theta})|\hat{I}\otimes\hat{I}|\psi(\boldsymbol{\theta})\rangle}_{=1} + \epsilon_{ZI}\langle\psi(\boldsymbol{\theta})|\hat{Z}\otimes\hat{I}|\psi(\boldsymbol{\theta})\rangle + \epsilon_{IZ}\langle\psi(\boldsymbol{\theta})|\hat{I}\otimes\hat{Z}|\psi(\boldsymbol{\theta})\rangle \\ &+ (\epsilon_{ZZ}+\lambda H_z)\langle\psi(\boldsymbol{\theta})|\hat{Z}\otimes\hat{Z}|\psi(\boldsymbol{\theta})\rangle + \lambda H_x\langle\psi(\boldsymbol{\theta})|\hat{X}\otimes\hat{X}|\psi(\boldsymbol{\theta})\rangle\end{aligned} \tag{10}$$

where $\boldsymbol{\theta} = (\theta_0, \phi_0, \theta_1, \phi_1)$ are the variational parameters. The basis transformations required to measure the Pauli strings in $\langle H_2(\boldsymbol{\theta})\rangle$ are summarized in Table 1. A Python function for calculating $\langle H_2(\boldsymbol{\theta})\rangle$ using quantum gate logic is given in Program 33, and the corresponding quantum circuit configuration is illustrated in Figure 11.

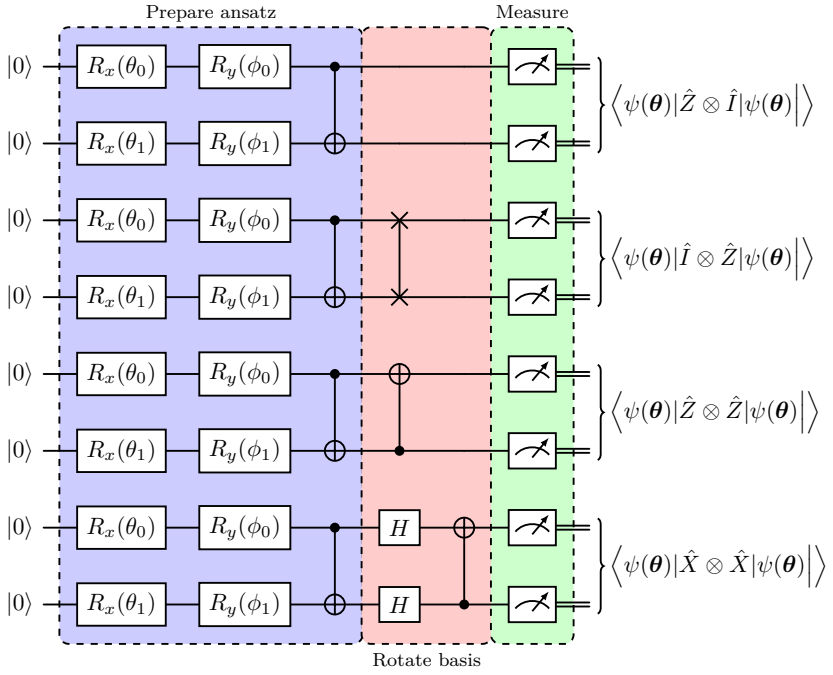

Figure 11: Quantum circuit for the variational quantum eigensolver (VQE) applied to $\langle\hat{H}_2(\boldsymbol{\theta})\rangle$.

Figure 12 shows the minimized VQE energies for $\langle H_2(\boldsymbol{\theta})\rangle$. The estimated energies closely align with the numerical eigenvalues, albeit with slightly greater discrepancies compared to the one-qubit Hamiltonian case. This can be attributed to the larger dimensionality of the parameter space in the two-qubit system, which introduces greater complexity to the optimization process. Additionally, the computational cost is significantly higher, highlighting the increased complexity of simulating larger quantum systems and underscoring the limitations of classical computing for scaling such simulations.

6 Part f) Lipkin Model Eigenvalue Problem

To find the matrix representation of Lipkin Hamiltonians, we use the quasispin operator equations:

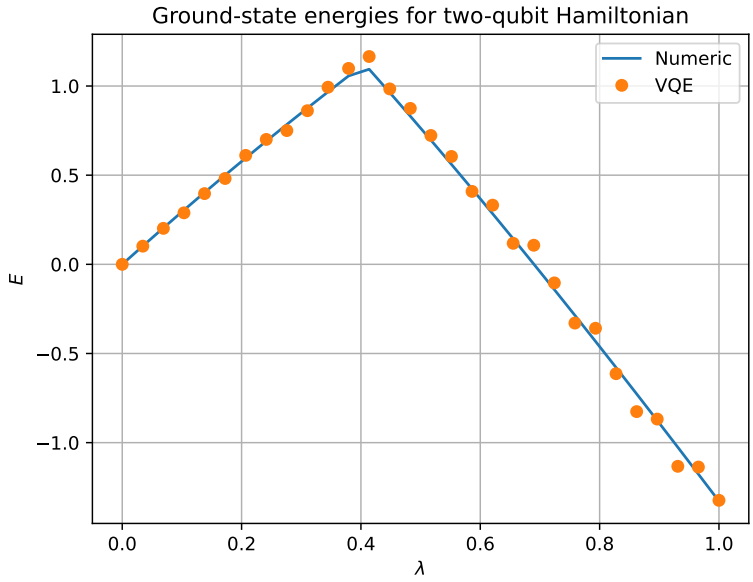

Figure 12: A plot of VQE energies of $\langle H_2(\boldsymbol{\theta})\rangle$, optimized using gradient descent with Adaptive Moment Estimation (ADAM) over a maximum of 500 epochs. The expectation values $\langle H_2\rangle$ were computed from quantum circuit measurements over 1000 shots.

$$
\begin{aligned}
\hat{J}^2 \left|j, j_z\right\rangle &= j(j+1) \left|j, j_z\right\rangle \\
\hat{J}_z \left|j, j_z\right\rangle &= j_z \left|j, j_z\right\rangle \\
\hat{J}_\pm \left|j, j_z\right\rangle &= \sqrt{j(j+1) - j_z(j_z \pm 1)} \left|j, j_z \pm 1\right\rangle \\
\hat{N} \left|j, j_z\right\rangle &= 2j \left|j, j_z\right\rangle
\end{aligned}
$$

where j denote the total spin and j_z the z-projection of the spin. For a system with N fermions, the total spin is $j = N/2$, while the eigenvalues of $\hat{J}_z$ form a discrete sequence $j_z \in \{-N/2, -N/2+1, \ldots, N/2-1, N/2\}$, which results in $N+1$ possible states for the system.

To calculate the matrix elements of the Hamiltonian, we solve the time-independent Schrödinger equation $\hat{H}_j \left|\psi_j\right\rangle = E_{j,j_z} \left|\psi_j\right\rangle$, where

$$
\left|\psi_j\right\rangle = \sum_{j_z=-N/2}^{N/2} c_{j_z} \left|j, j_z\right\rangle \tag{11}
$$

Applying $\langle j, j_z'|$ to both sides, we get

$$
\sum_{j_z', j_z=-N/2}^{N/2} c_{j_z} \left\langle j, j_z'|\hat{H}_j|j, j_z\right\rangle = \sum_{j_z', j_z=-N/2}^{N/2} c_{j_z} E_{j,j_z} \underbrace{\left\langle j, j_z'|j, j_z\right\rangle}_{=\delta_{j_z j_z'}} = E_{j,j_z} c_{j_z} \tag{12}
$$

In terms of the quantum numbers, the non-zero matrix elements are

$$
\begin{aligned}
\langle j, j_z|\hat{H}_j|j, j_z\rangle =& j_z\epsilon - W[j^2 - j_z^2] \\
\langle j, j_z|\hat{H}_j|j, j_z+2\rangle =& -\frac{V}{2}\sqrt{[j(j+1) - j_z(j_z-1)][j(j+1) - (j_z-1)(j_z-2)]} \\
\langle j, j_z+2|\hat{H}_j|j, j_z\rangle =& \langle j, j_z|\hat{H}_j|j, j_z+2\rangle
\end{aligned}
$$

Program 35 shows a Python function for constructing the Hamiltonian matrix for an N-fermion Lipkin model.

6.1 Two-Fermion Lipkin Model ($j = 1$)

Here we construct the Lipkin Hamiltonian matrix $\hat{H}_{j=1}$ for a two-fermion system with spin $j = 1$. The diagonal elements are given by

$$
\begin{aligned}
H_{00} &= \langle 1, -1|\hat{H}|1, -1\rangle = -1\varepsilon - W[1^2 - (-1)^2] = -\varepsilon \\
H_{11} &= \langle 1, 0|\hat{H}|1, 0\rangle = 0\varepsilon - W[1^2 - 0^2] = -W \\
H_{22} &= \langle 1, -1|\hat{H}|1, -1\rangle = 1\varepsilon - W[1^2 - 1^2] = \varepsilon
\end{aligned}
$$

The nonzero off-diagonal elements are

$$
\begin{aligned}
H_{02} &= \langle 1, 1|\hat{H}|1, 3\rangle = -\frac{V}{2}\underbrace{\sqrt{[1(1+1) - 1(1-1)][1(1+1) - (1-1)(1-2)]}}_{=2} = -V \\
H_{20} &= \langle 1, 3|\hat{H}|1, 1\rangle = \langle 1, 1|\hat{H}|1, 3\rangle = -V
\end{aligned}
$$

Thus, the Hamiltonian takes the form

$$
\mathbf{H} = \begin{bmatrix} -\varepsilon & 0 & -V \\ 0 & -W & 0 \\ -V & 0 & \varepsilon \end{bmatrix} \tag{13}
$$

To diagonalize $\mathbf{H}$, we solve the characteristic equation

$$
0 = \det(\mathbf{H} - \lambda\mathbf{I}) = \begin{vmatrix} -\varepsilon - \lambda & 0 & -V \\ 0 & -W - \lambda & 0 \\ -V & 0 & \varepsilon - \lambda \end{vmatrix} \tag{14}
$$

Using cofactor expansion along the second column, we can refactor the determinant as

$$
\det(\mathbf{H} - \lambda\mathbf{I}) = (-W - \lambda)\begin{vmatrix} -\varepsilon & -V \\ -V & \varepsilon - \lambda \end{vmatrix}. \tag{15}
$$

The determinant of the 2×2 submatrix is

$$\begin{aligned}(-\varepsilon - \lambda)(\varepsilon - \lambda) - (-V)(-V) =& (\lambda + \varepsilon)(\lambda - \varepsilon) - V^2 \\ =& \lambda^2 - \varepsilon^2 - V^2,\end{aligned}$$

leading to the characteristic eqution

$$(-W - \lambda)(\lambda^2 - \varepsilon^2 - V^2) = 0 \tag{16}$$

The first factor $(-W - \lambda) = 0$ gives the eigenvalue $\lambda_1 = -W$, while the quadratic equation $\lambda^2 - \varepsilon^2 - V^2 = 0$ has solutions

$$\lambda = \pm\sqrt{\varepsilon^2 + V^2} \tag{17}$$

Thus, the ordered eigenvalues of $\mathbf{H}_{j=1}$ are

$$\begin{aligned}E_1 =& -\sqrt{\varepsilon^2 + V^2} \\ E_2 =& -W \\ E_3 =& \sqrt{\varepsilon^2 + V^2}\end{aligned}$$

Figure 13 displays the eigenvalues of $\hat{H}_{j=1}$ as a function of the interaction strength V, with parameters $\epsilon = 1$ and $W = 0$. In this case the Hamiltonian simplifies to

$$\hat{H}_{j=1} = \epsilon \hat{J}_z^2 + \frac{V}{2}(\hat{J}_+^2 + \hat{J}_-^2) \tag{18}$$

The $\hat{J}_z$ represents the unperturbed energy levels, with eigenstates labeled by the spin projections $j_z \in \{-1, 0, 1\}$. The quadratic interaction term $\hat{J}_+^2 + \hat{J}_-^2$ introduces coupling of eigenstates that differ by two units of j_z, meaning that the states $|j = 1, j_z = -1\rangle$ and $|j = 1, j_z = 1\rangle$ are coupled, while the state $|j = 1, j_z = 0\rangle$ remains uncoupled.

As V increases, this interaction induces level repulsion, causing the energy gap between the coupled eigenstates to widen. This growing energy separation suppresses energy transitions, making excitations between states less likely as interactions strengthen. The middle eigenvalue $\hat{E}_2$ remains constant, indicating the presence of a conserved energy state that is unaffected by the interaction.

6.2 Four-Fermion Lipkin Model

Figure 14 plots the eigenvalues of $\hat{H}_{j=2}$ as a function of the interaction strength V, with parameters $\epsilon = 1$ and $W = 0$. In the four-fermion case, the interaction term $\hat{J}_+^2 + \hat{J}_-^2$ couples eigenstates that differ by two units of $j_z \in \{-2, -1, 0, 1, 2\}$, leading to the formation of two distinct parity groups:

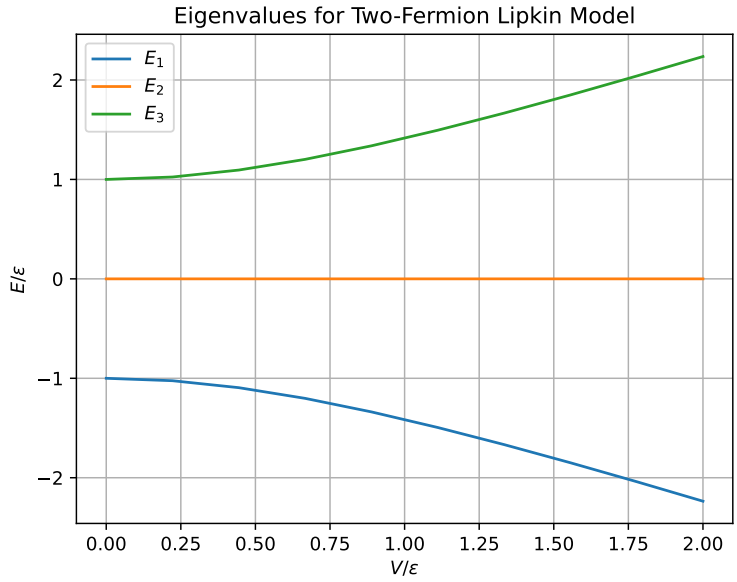

Figure 13: Energy levels for the the two-fermion Lipkin model as function of the interaction strength $V/\epsilon \in [0, 2]$.

- An even-parity group consisting of states with $j_z \in \{-2, 0, +2\}$
- An odd-parity group consisting of states with $j_z \in \{-1, 1\}$

Similar to the two-fermion case, level repulsion occurs between the coupled states, causing their energy separation as V increases. Additionally, the states within each uncoupled parity group remain closely spaced, forming nearly degenerate pairs with a constant energy gap.

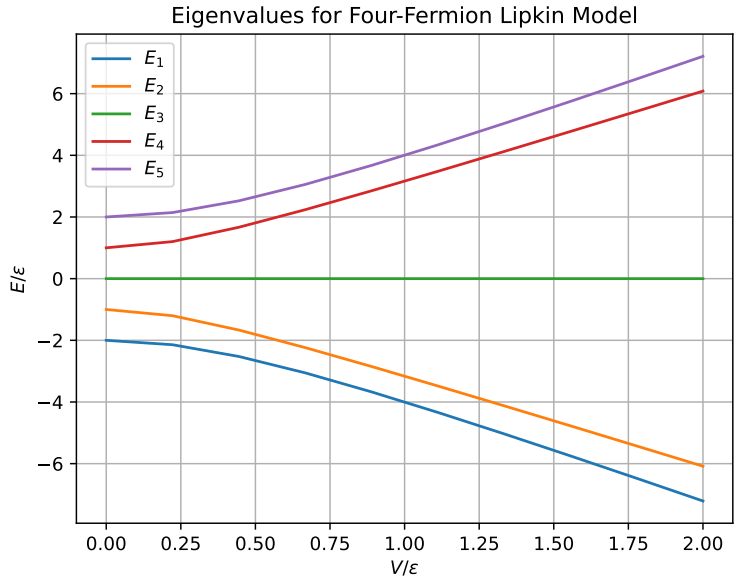

Figure 14: Energy levels for the the four-fermion Lipkin model as function of the interaction strength $V/\epsilon \in [0, 2]$.

6.3 Pauli String Hamiltonians

In terms of quasispin operators, the N-particle Lipkin Hamiltonian is composed of the terms

$$\begin{aligned}\hat{H}_0 =&\epsilon\hat{J}_z\\ \hat{H}_1 =&\frac{V}{2}(\hat{J}_+^2+\hat{J}_-^2)\\ \hat{H}_2 =&\frac{W}{2}(\hat{J}_+\hat{J}_-+\hat{J}_-\hat{J}_+-\hat{N})\end{aligned}$$

To express $\hat{H}=\hat{H}_0+\hat{H}_1+\hat{H}_2$ in terms of Pauli strings, we first represent it in terms of individual spin operators given by

$$\hat{J}_z=\sum_{p=1}^{N}\hat{j}_z^{(p)},\quad \hat{J}_\pm=\sum_{p=1}^{N}\hat{j}_\pm^{(p)}=\sum_{p=1}^{N}\left(\hat{j}_x^{(p)}\pm i\hat{j}_y^{(p)}\right), \tag{19}$$

For spin-1/2 particles, the spin operators are related to the Pauli matrices via:

$$\hat{j}_x^{(p)}=\frac{1}{2}\hat{X}_p,\quad \hat{j}_y^{(p)}=\frac{1}{2}\hat{Y}_p,\quad \hat{j}_z^{(p)}=\frac{1}{2}\hat{Z}_p. \tag{20}$$

Here, each Pauli operator $\sigma_p\in\{\hat{X}_p,\hat{Y}_p,\hat{Z}_p\}$ is embedded in the N-qubit Hilbert space as the tensor product:

$$\hat{\sigma}_p=\hat{I}_2^{\otimes(p-1)}\otimes\hat{\sigma}\otimes\hat{I}_2^{\otimes(N-p+1)}. \tag{21}$$

In terms of Pauli strings, the $\hat{H}_0$ term can be expressed as

$$\hat{H}_0=\epsilon\hat{J}_z=\epsilon\sum_{p=1}^{N}\hat{j}_z^{(p)}=\frac{\epsilon}{2}\sum_{p=1}^{N}\hat{Z}_p \tag{22}$$

For the $\hat{H}_1$ term, we need to expand the $\hat{J}_\pm^2$ operators:

$$\begin{aligned}\hat{J}_\pm^2&=\left(\sum_{p=1}^{N}\hat{j}_x^{(p)}\pm i\hat{j}_y^{(p)}\right)^2=\sum_{p,q=1}^{N}(\hat{j}_x^{(p)}\pm i\hat{j}_y^{(p)})(\hat{j}_x^{(q)}\pm i\hat{j}_y^{(q)})\\ &=\sum_{p,q=1}^{N}(\hat{j}_x^{(p)}\hat{j}_x^{(q)}-\hat{j}_y^{(p)}\hat{j}_y^{(q)}\pm i\hat{j}_y^{(p)}\hat{j}_x^{(q)}\pm i\hat{j}_x^{(p)}\hat{j}_y^{(q)})\end{aligned}$$

We can thus write $\hat{J}_+^2+\hat{J}_-^2$ as

$$\begin{aligned}
\hat{J}_+^2 + \hat{J}_-^2 =& 2\sum_{p,q=1}^{N}(\hat{j}_x^{(p)}\hat{j}_x^{(q)} - \hat{j}_y^{(p)}\hat{j}_y^{(q)}) \\
=& 2\sum_{p=1}^{N}[(\hat{j}_x^{(p)})^2 - (\hat{j}_y^{(p)})^2] + 4\sum_{p>q}^{N}(\hat{j}_x^{(p)}\hat{j}_x^{(q)} - \hat{j}_y^{(p)}\hat{j}_y^{(q)}) \\
=& \frac{1}{2}\sum_{p=1}^{N}(\underbrace{\hat{X}_p^2}_{=\hat{I}} - \underbrace{\hat{Y}_p^2}_{=\hat{I}}) + \sum_{p>q}^{N}(\hat{X}_p\hat{X}_q - \hat{Y}_p\hat{Y}_q) \\
=& \sum_{p>q}^{N}(\hat{X}_p\hat{X}_q - \hat{Y}_p\hat{Y}_q)
\end{aligned}$$

For the $\hat{H}_2$ term, we need to expand the $\hat{J}_\pm\hat{J}_\mp$ operators:

$$\begin{aligned}
\hat{J}_\pm\hat{J}_\mp =& \sum_{p,q=1}^{n}(\hat{j}_x^{(p)} \pm i\hat{j}_y^{(p)})(\hat{j}_x^{(q)} \mp i\hat{j}_y^{(q)}) \\
=& \sum_{pq}(\hat{j}_x^{(p)}\hat{j}_x^{(q)} + \hat{j}_y^{(p)}\hat{j}_y^{(q)} \mp i\hat{j}_x^{(p)}\hat{j}_y^{(q)} \pm i\hat{j}_y^{(p)}\hat{j}_x^{(q)})
\end{aligned}$$

We can thus express $\hat{J}_+\hat{J}_- + \hat{J}_-\hat{J}_+$ as

$$\begin{aligned}
\hat{J}_+\hat{J}_- + \hat{J}_-\hat{J}_+ =& \sum_{p,q=1}^{N}(\hat{j}_x^{(p)}\hat{j}_x^{(q)} + \hat{j}_y^{(p)}\hat{j}_y^{(q)}) \\
=& 2\sum_{p=1}^{N}[(\hat{j}_x^{(p)})^2 + (\hat{j}_y^{(p)})^2] + 4\sum_{p>q}^{N}(\hat{j}_x^{(p)}\hat{j}_x^{(q)} + \hat{j}_y^{(p)}\hat{j}_y^{(q)}) \\
=& \frac{1}{2}\sum_{p=1}^{N}(\underbrace{\hat{X}_p^2}_{=\hat{I}} + \underbrace{\hat{Y}_p^2}_{=\hat{I}}) + \sum_{p<q}^{N}(\hat{X}_p\hat{X}_q + \hat{Y}_p\hat{Y}_q) \\
=& \underbrace{N\hat{I}}_{=\hat{N}} + \sum_{p<q}^{N}(\hat{X}_p\hat{X}_q + \hat{Y}_p\hat{Y}_q)
\end{aligned}$$

This allows us to write the Hamiltonian components as

$$\begin{aligned}
\hat{H}_0 =& \frac{\epsilon}{2}\sum_{p=1}^{N}\hat{Z}_p \\
\hat{H}_1 =& \frac{V}{2}\sum_{p<q}^{N}(\hat{X}_p\hat{X}_q - \hat{Y}_p\hat{Y}_q) \\
\hat{H}_2 =& \frac{W}{2}\sum_{p<q}^{N}(\hat{X}_p\hat{X}_q + \hat{Y}_p\hat{Y}_q)
\end{aligned}$$

For a two-fermion system with $N = 2$ and $j = 1$, we get

$$\hat{H}_{j=1} = \frac{\epsilon}{2}(\hat{Z} \otimes \hat{I}_2 + \hat{I}_2 \otimes \hat{Z}) + \frac{W+V}{2}\hat{X} \otimes \hat{X} + \frac{W-V}{2}\hat{Y} \otimes \hat{Y} \tag{23}$$

For a four-fermion system with $N = 4$ and $j = 2$, we get

$$\begin{aligned}\hat{H}_{j=2} =& \frac{\epsilon}{2}(\hat{Z}_1 + \hat{Z}_2 + \hat{Z}_3 + \hat{Z}_4) \\ &+ \frac{W-V}{2}(\hat{X}_1\hat{X}_2 + \hat{X}_1\hat{X}_3 + \hat{X}_1\hat{X}_4 + \hat{X}_2\hat{X}_3 + \hat{X}_2\hat{X}_4 + \hat{X}_3\hat{X}_4) \\ &+ \frac{W+V}{2}(\hat{Y}_1\hat{Y}_2 + \hat{Y}_1\hat{Y}_3 + \hat{Y}_1\hat{Y}_4 + \hat{Y}_2\hat{Y}_3 + \hat{Y}_2\hat{Y}_4 + \hat{Y}_3\hat{Y}_4)\end{aligned} \tag{24}$$

The basis transformations required to measure the Pauli strings in $\hat{H}_{j=1}$ are given in Table 1.

7 Part g) VQE for Lipkin Model

7.1 Two-Fermion Model

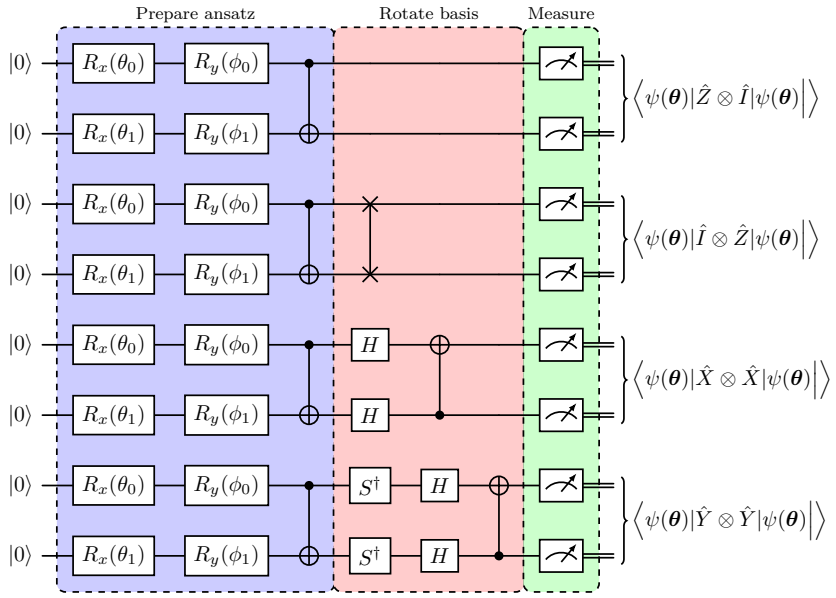

Figure 15: Quantum circuit for the variational quantum eigensolver (VQE) applied to $\langle\hat{H}_{j=1}(\boldsymbol{\theta})\rangle$.

Figure 15 illustrate the quantum circuit for computing the expectation value of $\hat{H}_{j=1}$ (23). A Python function simulating this circuit is given in Program 36.

Figure 16 shows the minimized VQE energies for $\langle H_{j=1}(\boldsymbol{\theta})\rangle$, with parameters $\epsilon = 1$ and $W = 0$. These estimated energies closely match the numerical eigenvalues. From a computational perspective, this optimization problem is similar to the two-qubit Hamiltonian previously discussed, both in terms of quantum circuit setup and optimization performance. The primary difference is that the Lipkin Hamiltonian includes a $\hat{Y} \otimes \hat{Y}$ terms, as opposed to the $\hat{Z} \otimes \hat{Z}$ term in the two-qubit Hamiltonian (10).

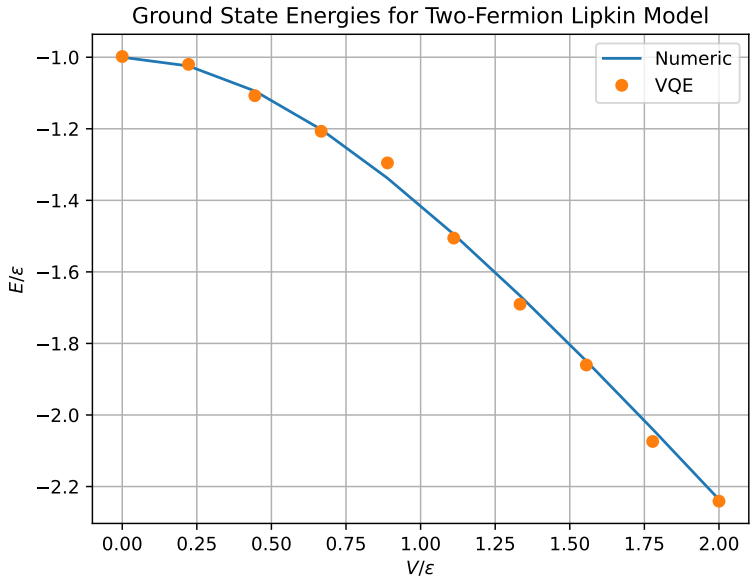

Figure 16: A plot of VQE energies of $\langle H_{j=1}(\boldsymbol{\theta})\rangle$, optimized using gradient descent with Adaptive Moment Estimation (ADAM) over a maximum of 500 epochs. The expectation values $\langle H_{j=1}\rangle$ were computed from quantum circuit measurements over 1000 shots.

7.2 Four-Fermion Model

A Python function using Qiskit for calculating the expectation value of $\hat{H}_{j=2}$ (24) is given in Program 37. Figure 16 shows the minimized VQE energies for $\langle H_{j=2}(\boldsymbol{\theta})\rangle$, with parameters $\epsilon = 1$ and $W = 0$. For small values of V, the VQE energies close the numerical ground state eigenvalue E_1. However, as V increases, the VQE energies tend to converge toward the first excited energy state, corresponding to the eigenvalue E_2. This behaviour is likely due to the near-degeneracy of the energy states, which introduces additional local minima. This behavior demonstrates the challenge of using VQE for Hamiltonian with near-degenerate energy levels, as the optimization algorithm can become trapped in local minima associated with excited states, rather than the global minimum corresponding to the true ground state.

8 Appendix

8.1 Code Repository

The Python source code used for this project is available at https://github.com/semapheur/fys5419.

8.2 Pauli String Basis Transformations

8.3 Derivation of Matrix Elements of Lipkin Hamiltonian

The matrix elements of $\hat{H}$ in the $|j, j_z\rangle$ basis are given by

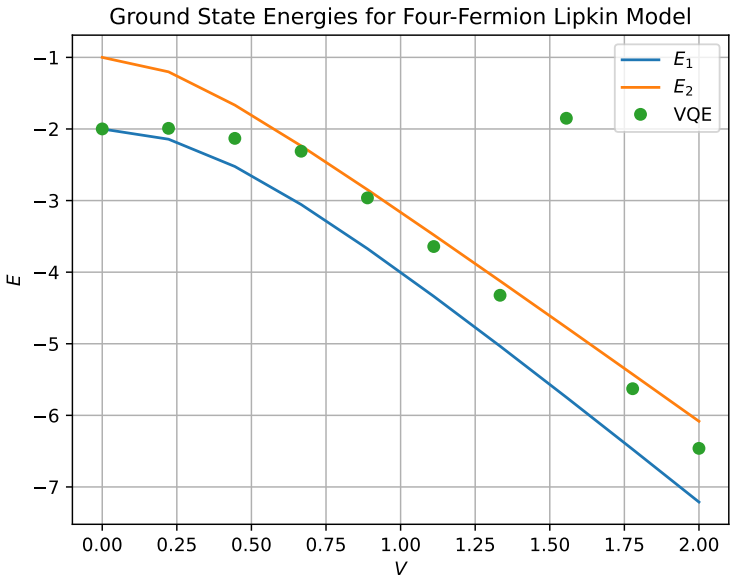

Figure 17: A plot of VQE energies of $\langle H_{j=2}(\boldsymbol{\theta})\rangle$, optimized using gradient descent with Adaptive Moment Estimation (ADAM) over a maximum of 500 epochs. The expectation values $\langle H_{j=1}\rangle$ were computed from quantum circuit measurements over 1000 shots.

Table 1: Transformations for measuring two-qubit Pauli strings in the $\hat{Z}\otimes\hat{I}_2$ and $\hat{Z}\otimes\hat{I}_2\otimes\hat{I}_2\otimes\hat{I}_2$ bases.

Pauli string $\hat{P}=\bigotimes_{i=1}^{N}\hat{\sigma}_i$	Quantum gate $\hat{U}_{\hat{P}}$
$\hat{Z}\otimes\hat{I}$	$\hat{I}_2\otimes\hat{I}_2$
$\hat{I}\otimes\hat{Z}$	$(\hat{I}_2\otimes\hat{I}_2)\,\mathrm{SWAP}$
$\hat{Z}\otimes\hat{Z}$	$\mathrm{CNOT}_{1,0}$
$\hat{X}\otimes\hat{X}$	$\mathrm{CNOT}_{1,0}(\hat{H}\otimes\hat{H})$
$\hat{Y}\otimes\hat{Y}$	$\mathrm{CNOT}_{1,0}(\hat{H}\hat{S}^\dagger\otimes\hat{H}\hat{S}^\dagger)$
$\hat{Z}\otimes\hat{I}\otimes\hat{I}\otimes\hat{I}$	$\hat{I}_2\otimes\hat{I}_2\otimes\hat{I}_2\otimes\hat{I}_2$
$\hat{I}_2\otimes\hat{Z}\otimes\hat{I}\otimes\hat{I}$	$\hat{U}_{IZ}\otimes\hat{I}_2\otimes\hat{I}_2$
$\hat{I}_2\otimes\hat{I}_2\otimes\hat{Z}\otimes\hat{I}$	$\hat{U}_{IZII}(\otimes\hat{I}_2\otimes\hat{U}_{IZ}\otimes\hat{I}_2)$
$\hat{I}_2\otimes\hat{I}_2\otimes\hat{I}_2\otimes\hat{Z}$	$\hat{U}_{IIZI}(\otimes\hat{I}_2\otimes\hat{I}_2\otimes\hat{U}_{IZ})$
$\hat{Z}\otimes\hat{I}_2\otimes\hat{Z}\otimes\hat{I}_2$	$(\mathrm{CNOT}_{1,0}\otimes\hat{I}_2\otimes\hat{I}_2)(\mathrm{SWAP}\otimes\hat{I}_2\otimes\hat{I}_2)$
$\hat{X}\otimes\hat{X}\otimes\hat{I}_2\otimes\hat{I}_2$	$\hat{U}_{XX}\otimes\hat{I}_2\otimes\hat{I}_2$
$\hat{X}\otimes\hat{I}_2\otimes\hat{X}\otimes\hat{I}_2$	$\hat{U}_{ZIZI}(\hat{U}_{XI}\otimes\hat{U}_{XI})$
$\hat{X}\otimes\hat{I}_2\otimes\hat{I}_2\otimes\hat{X}$	$\hat{U}_{ZIZI}(\hat{U}_{XI}\otimes\hat{U}_{IX})$
$\hat{I}_2\otimes\hat{X}\otimes\hat{X}\otimes\hat{I}_2$	$\hat{U}_{ZIZI}(\hat{U}_{IX}\otimes\hat{U}_{XI})$
$\hat{I}_2\otimes\hat{X}\otimes\hat{I}\otimes\hat{X}$	$\hat{U}_{ZIZI}(\hat{U}_{IX}\otimes\hat{U}_{IX})$
$\hat{I}_2\otimes\hat{I}_2\otimes\hat{X}\otimes\hat{X}$	$\hat{U}_{IIZI}(\hat{I}_2\otimes\hat{I}\otimes\hat{U}_{XX})$
$\hat{Y}\otimes\hat{Y}\otimes\hat{I}_2\otimes\hat{I}_2$	$\hat{U}_{YY}\otimes\hat{I}_2\otimes\hat{I}_2$
$\hat{Y}\otimes\hat{I}_2\otimes\hat{Y}\otimes\hat{I}_2$	$\hat{U}_{ZIZI}(\hat{U}_{YI}\otimes\hat{U}_{YI})$
$\hat{Y}\otimes\hat{I}_2\otimes\hat{I}_2\otimes\hat{Y}$	$\hat{U}_{ZIZI}(\hat{U}_{YI}\otimes\hat{U}_{IY})$
$\hat{I}_2\otimes\hat{Y}\otimes\hat{Y}\otimes\hat{I}_2$	$\hat{U}_{ZIZI}(\hat{U}_{IY}\otimes\hat{U}_{YI})$
$\hat{I}_2\otimes\hat{Y}\otimes\hat{I}\otimes\hat{Y}$	$\hat{U}_{ZIZI}(\hat{U}_{IY}\otimes\hat{U}_{IY})$
$\hat{I}_2\otimes\hat{I}_2\otimes\hat{Y}\otimes\hat{Y}$	$\hat{U}_{IIZI}(\hat{I}_2\otimes\hat{I}\otimes\hat{U}_{YY})$

$$\langle j, j_z'|\hat{H}_j|j, j_z\rangle = \langle j, j_z'|\hat{H}_{0,j}|j, j_z\rangle + \langle j, j_z'|\hat{H}_{1,j}|j, j_z\rangle + \langle j, j_z'|\hat{H}_{2,j}|j, j_z\rangle \tag{25}$$

The matrix elements of $\hat{H}_{0,j}$ are given by

$$\begin{aligned}\langle j, j_z'|\hat{H}_{0,j}|j, j_z\rangle =& \langle j, j_z'|\varepsilon\hat{J}_z|j, j_z\rangle \\ =& \varepsilon j_z \langle j, j_z'|j, j_z\rangle \\ =& \varepsilon j_z \delta_{j_z j_z'}\end{aligned}$$

To find the matrix elements of $\hat{H}_{j,1}$, we evaluate

$$\begin{aligned}\langle j, j_z'|\hat{H}_{1,j}|j, j_z\rangle =& \left\langle j, j_z' \middle| -\frac{1}{2}V(\hat{J}_+^2 + \hat{J}_-^2) \middle| j, j_z \right\rangle \\ =& -\frac{1}{2}V\left(\langle j, j_z'|\hat{J}_+^2|j, j_z\rangle + \langle j, j_z'|\hat{J}_-^2|j, j_z\rangle\right)\end{aligned}$$

Since

$$\begin{aligned}\hat{J}_\pm^2|j, j_z \pm 1\rangle =& \sqrt{j(j+1) - j_z(j_z \pm 1)}\hat{J}_\pm|j, j_z \pm 1\rangle \\ =& \sqrt{[j(j+1) - j_z(j_z \pm 1)][j(j+1) - (j_z \pm 1)(j_z \pm 2)]}\,|j, j_z \pm 2\rangle\end{aligned}$$

we get

$$\begin{aligned}\langle j, j_z'|\hat{H}_{1,j}|j, j_z\rangle =& -\frac{1}{2}V\left(\sqrt{[j(j+1) - j_z(j_z+1)][j(j+1) - (j_z+1)(j_z+2)]}\,\langle j, j_z'|j, j_z+2\rangle\right. \\ &\left. + \sqrt{[j(j+1) - j_z(j_z-1)][j(j+1) - (j_z-1)(j_z-2)]}\,\langle j, j_z'|j, j_z-2\rangle\right)\end{aligned}$$

Using the orthonormality condition $\langle j, j_z'|j, j_z+2\rangle = \delta_{j_z', j_z+2}$, the matrix elements are nonzero for $j_z' = j_z + 2$ and thus

$$\langle j, j_z+2|\hat{H}_{1,}|j, j_z\rangle = -\frac{1}{2}V\sqrt{[j(j+1) - j_z(j_z+1)][j(j+1) - (j_z+1)(j_z+2)]} \tag{26}$$

From the Hermiticity of $\hat{H}_{1,}$, we have

$$\langle j, j_z+2|\hat{H}_{1,}|j, j_z\rangle = \langle j, j_z|\hat{H}_{1,}|j, j_z+2\rangle^* = \langle j, j_z|\hat{H}_{1,j}|j, j_z+2\rangle \tag{27}$$

To find the matrix elements of $\hat{H}_{2,j}$, we evaluate

$$\begin{aligned}\langle j, j_z'|\hat{H}_{2,j}|j, j_z\rangle =& \left\langle j, j_z' \right| -\frac{1}{2}W\left(-\hat{N} + \hat{J}_+\hat{J}_- + \hat{J}_-\hat{J}_+\right)\left| j, j_z \right\rangle \\ =& -\frac{1}{2}W\left(-\langle j, j_z'|\hat{N}|j, j_z\rangle + \langle j, j_z'|\hat{J}_+\hat{J}_-|j, j_z\rangle + \langle j, j_z'|\hat{J}_-\hat{J}_+|j, j_z\rangle\right)\end{aligned}$$

Since $\hat{N}\,|j, j_z\rangle = 2j\,|j, j_z\rangle$, we get

$$\langle j, j_z'|\hat{N}|j, j_z\rangle = 2j\delta_{j_z,j_z'} \tag{28}$$

Furthermore, we find that

$$\begin{aligned}\hat{J}_+\hat{J}_-\,|j, j_z\rangle =& \sqrt{j(j+1) - j_z(j_z-1)}\hat{J}_+\,|j, j_z - 1\rangle \\ =& \sqrt{j(j+1) - j_z(j_z-1)}\sqrt{j(j+1) - (j_z-1)[(j_z-1)+1]}\,|j, j_z\rangle \\ =& j(j+1) - j_z(j_z-1)\,|j, j_z\rangle\end{aligned}$$

and similarly

$$\hat{J}_-\hat{J}_+\,|j, j_z\rangle = j(j+1) - j_z(j_z+1)\,|j, j_z\rangle \tag{29}$$

Thus,

$$\begin{aligned}\left\langle j, j_z' \right| \hat{J}_+\hat{J}_- + \hat{J}_-\hat{J}_+ \left| j, j_z \right\rangle =& j(j+1) - j_z(j_z-1)\,\langle j, j_z'|j, j_z\rangle \\ &+ j(j+1) - j_z(j_z+1)\,\langle j, j_z'|j, j_z\rangle \\ =& [j(j+1) - j_z(j_z-1) + j(j+1) - j_z(j_z+1)]\delta_{j_z,j_z'} \\ =& 2[j(j+1) - j_z^2]\delta_{j_z,j_z'}\end{aligned}$$

Combining the results, we obtain

$$\begin{aligned}\langle j, j_z'|\hat{H}_{2,j}|j, j_z\rangle =& -W\left[-j + j(j+1) - j_z^2\right]\delta_{j_z,j_z'} \\ =& -W(j^2 - j_z^2)\delta_{j_z,j_z'}\end{aligned}$$

8.4 Source Code

Some of the code snippets presented in this section have been shortened to better fit the PDF format. The complete source code can be found in the code respository for this project.

8.4.1 Qubit State Class

8.4.2 VQE Ansatz

8.4.3 VQE Measurement

8.4.4 Parameter Shift Gradient

8.4.5 VQE Gradient Descent (ADAM)

8.4.6 One-Qubit Hamiltonian Energies

8.4.7 One-Qubit Hamiltonian Quantum Circuit Computation

8.4.8 Two-Qubit Hamiltonian Energies and Entropies

8.4.9 Two-Qubit Hamiltonian Quantum Circuit Computation

8.4.10 Lipkin Hamiltonian

8.4.11 Two-Fermion Lipkin Quantum Circuit Computation

8.4.12 Four-Fermion Lipkin Quantum Circuit Computation

```
import numpy as np

class NQubitState:
  def __init__(
    self,
    num_qubits: int,
    state: NDArray[np.complex128] | None = None,
    basis_state: int | None = 0,
  ):
    self.num_qubits = num_qubits
    self.dim = 2**num_qubits

    if state is not None:
      if state.shape != (self.dim,):
        raise ValueError(f"State vector must be {self.dim} -dimensional")
      self._validate_and_set_state(state)
      return

    # Initialize to specified basis state (default |0...0)
    state = np.zeros(self.dim, dtype=np.complex128)
    state[basis_state] = 1.0
    self._state = state

  def _validate_and_set_state(self, state: NDArray[np.complex128]):
    # Ensure complex type
    state = state.astype(np.complex128)

    # Check normalization
    norm = np.linalg.norm(state)
    if not np.isclose(norm, 1.0, rtol=1e -6):
      raise ValueError(f"State vector not normalized. Norm: {norm}")

    self._state = state

  @property
  def state(self) -> NDArray[np.complex128]:
    return self._state

  @state.setter
  def state(self, state: NDArray[np.complex128]):
    self._validate_and_set_state(state)
```

Figure 18: Class constructer for an n-ary qubit state implemented in Python.

```
class NQubitState:
  ...

  def apply_unary_gate(self, gate: NDArray[np.complex128], target: int):
    """Apply a unary quantum gate to the specified qubit.

    Args:
      gate (NDArray[np.complex128]): 2x2 unitary matrix representing the gate
      target (int): Index of target qubit"""

    if gate.shape != (2, 2):
      raise ValueError("Gate must be a 2x2 matrix")

    # Construct full gate matrix using tensor products
    full_gate = np.array([[1]], dtype=np.complex128)
    for i in range(self.num_qubits):
      full_gate = np.kron(full_gate, gate if i == target else np.eye(2)).astype(
        np.complex128
      )

    self.state = full_gate @ self.state

  def apply_controlled_gate(
    self, gate: NDArray[np.complex128], control: int, target: int
  ):
    """Apply a controlled single -qubit gate.

    Args:
      gate (NDArray[np.complex128]): 2x2 unitary matrix representing the gate
      control (int): Index of control qubit
      target (int): Index of target qubit"""

    if gate.shape != (2, 2):
      raise ValueError("Gate must be a 2x2 matrix")

    # Initialize controlled gate
    controlled_gate = np.eye(self.dim, dtype=np.complex128)

    for i in range(self.dim):
      if (i >> target) & 1:  # Apply gate if target qubit is 1
        control_bit = (i >> control) & 1  # Get bit value of control qubit
        i_modified = i ^ (1 << control)  # Index of flipped control qubit

        controlled_gate[i, i] = gate[control_bit, control_bit]
        controlled_gate[i, i_modified] = gate[control_bit, 1 - control_bit]

    self.state = controlled_gate @ self.state
```

Figure 19: Methods for applying unary and controlled quantum gates in the the NQubitState class implemented in Python.

```
class NQubitState:
  ...

  def measure(
    self, targets: list[int], shots: int = 1, as_dict: bool = True
  ) -> dict[str, int] | list[tuple[int, ...]]:
    """Perform sequential measurements on specified qubits.

    Args:
      target (list[int]): List of qubit indices to measure in sequence
      shots (int): Number of measurements to perform

    Returns:
      dict[str, int] | list[tuple[int, ...]]: Measurement outcomes
    """

    # Generate measurement probabilities for each shot
    random_outcomes = np.random.random(size=(shots, len(targets)))

    outcomes = _measure_shots(self.state.copy(), targets, random_outcomes, self.dim)

    result = [tuple(outcome) for outcome in outcomes]
    if not as_dict:
      return result

    outcome_strings = ["".join(map(str, outcome)) for outcome in result]
    return dict(sorted(Counter(outcome_strings).items()))
```

Figure 20: A method for performing qubit measurements within the NQubitState class implemented in Python.

```
class NQubitClass:
  ...
  def hadamard_gate(self, target: int):
    """Apply a Hadamard gate to the specified qubit.

    Args:
      target (int): Index of target qubit"""

    H = np.array([[1, 1], [1, -1]], dtype=np.complex128) / np.sqrt(2)
    self.apply_unary_gate(H, target)

  def s_gate(self, target: int):
    """Apply the S gate to the specified qubit.

    Args:
      target (int): Index of target qubit"""

    s = np.array([[1, 0], [0, 1j]], dtype=np.complex128)
    self.apply_unary_gate(s, target)

  def s_gate_dagger(self, target: int):
    """Apply the S gate to the specified qubit.

    Args:
      target (int): Index of target qubit"""

    s_dagger = np.array([[1, 0], [0, -1j]], dtype=np.complex128)
    self.apply_unary_gate(s_dagger, target)

  def cnot_gate(self, control: int, target: int):
    """
    Apply the controlled -NOT (CNOT) gate with specified control qubit.

    Args:
      control (int): Index of control qubit
      target (int): Index of target qubit"""

    x = np.array([[0, 1], [1, 0]], dtype=np.complex128)
    self.apply_controlled_gate(x, control, target)
```

Figure 21: A selection quantum gate methods for the NQubitState class implemented in Python.

```
from numba import jit
import numpy as np

@jit(nopython=True)
def _measure_shots(
  state: NDArray[np.complex128],
  targets: list[int],
  random_outcomes: np.ndarray,
  dim: int,
) -> np.ndarray:

  shots = random_outcomes.shape[0]
  num_targets = len(targets)

  outcomes = np.zeros((shots, num_targets), dtype=np.int32)

  for shot in range(shots):
    current_state = state.copy()

    for i, target in enumerate(targets):
      # Bit mask for target qubit
      mask = 1 << target

      # Calculate probability for measuring |0 on target qubit
      prob_0 = 0.0
      for j in range(dim):
        if (j & mask) == 0:  # If target qubit is |0
          prob_0 += np.abs(current_state[j]) ** 2

      # Measure target qubit
      outcome = int(random_outcomes[shot, i] > prob_0)
      outcomes[shot, i] = outcome

      # Collapse state
      norm_squared = 0.0
      for j in range(dim):
        bit_value = (j & mask) >> target
        if bit_value != outcome:
          current_state[j] = 0.0
        else:
          norm_squared += np.abs(current_state[j]) ** 2

      # Normalize state
      if norm_squared > 1e -10:
        norm = np.sqrt(norm_squared)
        for j in range(dim):
          current_state[j] /= norm

  return outcomes
```

Figure 22: A Python function for simulating qubit state measurements, optimized using the JIT compiler of the Numba package for improved performance.

```
def prepare_ansatz(angles: np.ndarray) -> NQubitState:
  """
  Prepare a VQE ansatz state for an arbitrary number of qubits
  using rotation X and Y gates.

  Args:
    angles (np.ndarray): Parameter vector
      [theta_0, phi_0, theta_1, phi_1, ..., theta_n, phi_n]
    num_qubits (int): Number of qubits in the ansatz state

  Returns:
    NQubitState: VQE ansatz state
  """

  if len(angles) % 2 != 0:
    raise ValueError("Number of angles must be even")

  num_qubits = len(angles) // 2

  # Initialize the ansatz state
  qubit = NQubitState(num_qubits)

  # Apply rotation X and Y gates
  for i in range(num_qubits):
    theta_i = angles[2 * i]
    phi_i = angles[2 * i + 1]
    qubit.rotation_x_gate(theta_i, i)
    qubit.rotation_y_gate(phi_i, i)

  if num_qubits == 1:  # Special case for single qubit
    return qubit

  # Apply CNOT gates to entangle adjacent qubits
  for i in range(num_qubits - 1):
    qubit.cnot_gate(i, i + 1)

  return qubit
```

Figure 23: Python function for preparing a VQE ansatz state using $\hat{R}_x$ and $\hat{R}_y$ rotation gates.

```
def measurement_expectation(
  counts: dict[str, int], shots: int, qubit_index: int = -1
) -> float:
  """
  Calculate expectation value from measurement counts.

  Args:
    counts (dict[str, int]): Dictionary of measurement outcomes and their counts
    shots (int): Total number of shots
    qubit_index (int): Index of the qubit to compute expectation value for
      (default: last qubit).

  Returns:
    float: Expectation value in [ -1, 1]
  """
  expectation = sum(
    (1 if outcome[qubit_index] == "0" else -1) * count
    for outcome, count in counts.items()
  )

  return expectation / shots
```

Figure 24: Python function for calculating the expectation value from qubit measurements.

```
from concurrent.future import ThreadPoolExecutor
import numpy as np

def parameter_shift_gradient(
  angles: np.ndarray, interaction_strength: float, energy_fn: Callable
) -> np.ndarray:
  """
  Calculate gradient using parameter shift rule.

  Args:
    angles (np.ndarray): Parameter vector [theta, phi]
    interaction_strength (float): Interaction strength parameter
    energy_fn (Callable): Energy calculation function

  Returns:
    np.ndarray: Gradient vector
  """

  grad = np.zeros_like(angles)
  shift = np.pi / 2

  shifts: list[tuple[np.ndarray, float]] = []
  for i in range(len(angles)):
    angles_plus = angles.copy()
    angles_plus[i] += shift
    shifts.append((angles_plus, interaction_strength))

    angles_minus = angles.copy()
    angles_minus[i] -= shift
    shifts.append((angles_minus, interaction_strength))

  with ThreadPoolExecutor() as executor: # Parallelize
    energies = list(executor.map(lambda args: energy_fn(*args), shifts))

  for i, (energy_plus, energy_minus) in enumerate(zip(energies[::2], energies[1::2])):
    grad[i] = (energy_plus - energy_minus) / (2 * shift)

  return grad
```

Figure 25: Python function for calculating the gradient of energy expectation using parameter shift.

```
import numpy as np

def minimize_energy(
  angles_0: np.ndarray,
  energy_fn: Callable[[np.ndarray, float], float],
  lmb: float,
  learning_rate: float = 0.01,
  epochs: int = 1000,
  tolerance: float = 1e -6,
) -> tuple[np.ndarray, float, float, int]:

  angles = angles_0.copy()
  epoch = 0
  delta_energy = float("inf")
  min_energy = float("inf")
  no_improvement_epochs = 0

  energy = energy_fn(angles, lmb)
  energies = [energy]
  angle_store = [angles.copy()]

  dim = angles.shape[0]
  optimizer_state = {"momentums": np.zeros(dim), "velocities": np.zeros(dim)}

  while epoch < epochs and delta_energy > tolerance:
    # Calculate gradient
    grad = parameter_shift_gradient(angles, lmb, energy_fn)

    # Update learning rate and optimizer state
    update, optimizer_state = optimizer_update(
      method, grad, optimizer_state, learning_rate, epoch, **OPTIMIZER_PARAMS[method]
    )
    angles += update

    # Update energy and delta_energy
    energy_new = energy_fn(angles, lmb)
    delta_energy = np.abs(energy_new - energy)
    energy = energy_new

    epoch += 1

  final_energy = energy_fn(angles, lmb)
  return angles, energy, delta_energy, epoch
```

Figure 26: Python function for minimizing the energy expectation using gradient descent.

```
OPTIMIZER_PARAMS = {
  "adam": {"beta1": 0.9, "beta2": 0.999, "epsilon": 1e -8},
  "rmsprop": {"decay_rate": 0.9, "epsilon": 1e -8},
  "adagrad": {"epsilon": 1e -8},
  "sgd": {"momentum": 0.9},
}

class OptimizerState(TypedDict, total=False):
  squared_grad: np.ndarray | None
  grad_accumulator: np.ndarray | None
  momentum: float | None
  momentums: np.ndarray | None
  velocities: np.ndarray | None

def optimizer_update(
  method: str,
  gradient: np.ndarray,
  state: OptimizerState,
  learning_rate: float,
  epoch: int,
  **kwargs,
):
  if method == "adam":
    beta1 = cast(float, kwargs.get("beta1", 0.9))
    beta2 = cast(float, kwargs.get("beta2", 0.999))
    epsilon = cast(float, kwargs.get("epsilon", 1e -8))

    momentums = cast(np.ndarray, state["momentums"])
    velocities = cast(np.ndarray, state["velocities"])

    momentums = beta1 * momentums + (1 - beta1) * gradient
    velocities = beta2 * velocities + (1 - beta2) * np.square(gradient)

    momentums_hat = momentums / (1 - beta1 ** (epoch + 1))
    velocities_hat = velocities / (1 - beta2 ** (epoch + 1))

    update = -learning_rate * momentums_hat / (np.sqrt(velocities_hat) + epsilon)
    state["momentums"] = momentums
    state["velocities"] = velocities

  ...
  else:  # Default to gradient descent
    update = -learning_rate * gradient

  return update, state
```

Figure 27: Python function for updating the optimizer state in gradient descent. Only the implementation for the Adaptive Moment Estimation (ADAM) method is shown here. The full implementation is available in the code repository for this project.

```
from functool import partial

def vqe_energies(
  angle_parameters: int,
  energy_fn: Callable,
  lambdas: np.ndarray,
  shots: int,
  max_epochs: int,
  learning_rate: float,
  method: Optimizer,
  verbose: bool = False,
):
  partial_energy_fn = partial(energy_fn, shots=shots)
  vqe_energies = np.zeros(len(lambdas))
  epochs = np.zeros(len(lambdas))
  optimal_angles = []

  for i, lmb in enumerate(lambdas):
    if verbose:
      print(f"\nProcessing lambda = {lmb:.4f} ({i + 1}/{len(lambdas)})")

    best_energy = float("inf")
    best_angles = None
    best_epoch = 0

    angles_0 = np.random.uniform(0, np.pi, angle_parameters)
    angles, energy, _, epoch = minimize_energy(
      angles_0,
      partial_energy_fn,
      lmb,
      learning_rate=learning_rate,
      epochs=max_epochs,
      method=method,
      verbose=verbose,
    )

    if energy < best_energy:
      best_energy = energy
      best_angles = angles
      best_epoch = epoch

    vqe_energies[i] = best_energy
    epochs[i] = best_epoch
    optimal_angles.append(best_angles)

  return vqe_energies, epochs, optimal_angles
```

Figure 28: Python function for computing the VQE energies of a Hamiltonian as a function of interaction strength.

```
import numpy as np

def analytic_energies(lambdas: np.ndarray) -> np.ndarray:
  """
  Calculate the analytic energies of a single -qubit Hamiltonian
  for a set of interaction strengths.

  Args:
    lambdas (np.ndarray): Array of interaction strengths

  Returns:
    np.ndarray: 2 -column array of analytic energies,
    sorted in ascending order for each interaction strength
  """

  root = np.sqrt(np.square(omega + lambdas * omega_z) + np.square(lambdas * omega_x))

  E_minus = epsilon + lambdas * c - root
  E_plus = epsilon + lambdas * c + root

  return np.column_stack((E_minus, E_plus))

def numeric_energies(lambdas: np.ndarray) -> np.ndarray:
  """
  Calculate the exact energies of a single -qubit Hamiltonian
  for a set of interaction strengths.

  Args:
    lambdas (np.ndarray): Array of interaction strengths

  Returns:
    np.ndarray: 2 -column array of exact energies,
    sorted in ascending order for each interaction strength
  """

  return np.array([np.linalg.eigvalsh(hamiltonian(lmb)) for lmb in lambdas])
```

Figure 29: Python functions for computing the eigenvalues of the one-qubit Hamiltonian.

```
import numpy as np

def energy_expectation(
  angles: np.ndarray, interaction_strength: float, shots: int
) -> float:
  """
  Calculate energy expectation value for a single -qubit Hamiltonian of the form
  H = epsilon * I + omega * Z + c * I + omega_z * Z + omega_x * X

  Args:
    angles (np.ndarray): Parameter vector [theta, phi]
    interaction_strength (float): Interaction strength parameter
    shots (int): Number of measurement shots

  Returns:
    float: Energy expectation value
  """

  # Prepare ansatz
  ansatz = prepare_ansatz(angles)

  # Measure in Z -basis
  qubit = ansatz.copy()
  outcomes = cast(dict[str, int], qubit.measure([0], shots))
  measure_z = measurement_expectation(outcomes, shots)

  # Measure in X -basis
  qubit = ansatz.copy()
  qubit.hadamard_gate(0)  # Rotate to X -basis
  outcomes = cast(dict[str, int], qubit.measure([0], shots))
  measure_x = measurement_expectation(outcomes, shots)

  # Calculate expectation value
  exp_val_z = (omega + interaction_strength * omega_z) * measure_z
  exp_val_x = interaction_strength * omega_x * measure_x
  exp_val_i = epsilon + c * interaction_strength
  exp_val = exp_val_z + exp_val_x + exp_val_i
  return exp_val
```

Figure 30: Python function for computing the expectation of a one-qubit Hamiltonian using quantum circuit logic.

```
import numpy as np

sigma_x = np.array([[0.0, 1.0], [1.0, 0.0]])
sigma_z = np.array([[1.0, 0.0], [0.0, -1.0]])
I2 = np.eye(2)

H_x = 2.0
H_z = 3.0

eps_00 = 0.0
eps_01 = 2.5
eps_10 = 6.5
eps_11 = 7.0

def hamiltonian(interaction_strength: float):
  H_I = H_z * np.kron(sigma_z, sigma_z) + H_x * np.kron(sigma_x, sigma_x)
  H_0 = np.diag([eps_00, eps_01, eps_10, eps_11])
  H = H_0 + interaction_strength * H_I

  return H

def exact_energies_and_entropies(lambdas: np.ndarray):
  energies = np.zeros((len(lambdas), 4))
  entropies = np.zeros((len(lambdas), 4))

  for i, lmb in enumerate(lambdas):
    H = hamiltonian(lmb)
    eigenvalues, eigenvectors = np.linalg.eig(H)

    permute = eigenvalues.argsort()
    energies[i] = eigenvalues[permute]
    eigenvectors = eigenvectors[:, permute]

    # Calculate von Neumann entropy
    for j in range(4):
      sub_density = trace_out(eigenvectors[:, j], 0)
      eigenvals_density = np.linalg.eigvalsh(sub_density)
      eigenvals_density = np.ma.masked_equal(eigenvals_density, 0.0).compressed()
      entropies[i, j] = -np.sum(eigenvals_density * np.log2(eigenvals_density))

  return energies, entropies
```

Figure 31: Python functions for computing the eigenvalues of the two-qubit Hamiltonian, and the von Neumann entropies for the corresponding eigenstates.

```
import numpy as np

ket0 = np.array([1.0, 0.0])
ket1 = np.array([0.0, 1.0])

def trace_out(state: np.ndarray, index: int):
  """
  Calculate the reduced density matrix by tracing out a specified qubit.

  Args:
    state (np.ndarray): State vector of the quantum system.
    index (int): Index of the qubit to trace out.

  Returns:
    np.ndarray: Reduced density matrix after tracing out the specified qubit.
  """

  density = np.outer(state, np.conj(state))
  if index == 0:
    op0 = np.kron(ket0, I2)
    op1 = np.kron(ket1, I2)
  else:
    op0 = np.kron(I2, ket0)
    op1 = np.kron(I2, ket1)

  return op0.conj() @ density @ op0.T + op1.conj() @ density @ op1.T
```

Figure 32: Python functions for computing the reduced density matrix for a two-qubit state..

```
from concurrent.future import ThreadPoolExecutor
import numpy as np

def binary_energy_expectation(
  angles: np.ndarray, interaction_strength: float, shots: int
) -> float:
  """
  Calculate energy expectation value of a two -qubit Hamiltonian

  Args:
    angles (np.ndarray): Parameter vector [theta_0, phi_0, theta_1, phi_1]
    lmb (float): Interaction strength parameter
    shots (int): Number of measurement shots

  Returns:
    float: Energy expectation value
  """

  # Prepare ansatz
  ansatz = prepare_ansatz(angles)

  with ThreadPoolExecutor(max_workers=4) as executor:  # Parallelize
    future_zi = executor.submit(measure_z, ansatz.copy(), shots)  # Measure ZI
    future_iz = executor.submit(measure_nth_z, ansatz.copy(), 1, shots)  # Measure IZ
    future_zz = executor.submit(measure_zz, ansatz.copy(), shots)  # Measure ZZ
    future_xx = executor.submit(measure_xx, ansatz.copy(), shots)  # Measure XX

    try:
      expectation_zi = future_zi.result()
      expectation_iz = future_iz.result()
      expectation_zz = future_zz.result()
      expectation_xx = future_xx.result()
    except Exception as e:
      raise RuntimeError(f"Error in parallel measurement: {str(e)}")

  # Calculate expectation value
  exp_val = (
    eps_ii
    + eps_zi * expectation_zi
    + eps_iz * expectation_iz
    + (eps_zz + interaction_strength * H_z) * expectation_zz
    + (interaction_strength * H_x * expectation_xx)
  )

  return exp_val
```

Figure 33: Python function for computing the expectation of a two-qubit Hamiltonian using quantum circuit logic.

```
def measure_z(qubit: NQubitState, shots: int) -> float:
  # Measure a qubit state in the ZI...I basis
  outcome = cast(dict[str, int], qubit.measure(list(range(qubit.num_qubits)), shots))
  return measurement_expectation(outcome, shots)

def measure_nth_z(qubit: NQubitState, z_index: int, shots: int) -> float:
  """ Measure a qubit state in the I...IZI...I basis,
  where Z is in the z_index position """

  if z_index >= qubit.num_qubits:
    raise ValueError("Qubit index out of range")

  qubit.swap_gate(0, z_index)
  outcome = cast(dict[str, int], qubit.measure(list(range(qubit.num_qubits)), shots))
  return measurement_expectation(outcome, shots)

def measure_zz(qubit: NQubitState, shots: int) -> float:
  # Measure a two -qubit state in the ZZI...I basis

  qubit.cnot_gate(1, 0)
  outcome = cast(dict[str, int], qubit.measure(list(range(qubit.num_qubits)), shots))
  return measurement_expectation(outcome, shots)

def measure_xx(qubit: NQubitState, shots: int) -> float:
  # Measure a qubit state in the XXI...I basis

  qubit.hadamard_gate(0)
  qubit.hadamard_gate(1)
  qubit.cnot_gate(1, 0)
  outcome = cast(dict[str, int], qubit.measure(list(range(qubit.num_qubits)), shots))
  return measurement_expectation(outcome, shots)

def measure_yy(qubit: NQubitState, shots: int) -> float:
  # Measure a qubit state in the YYI...I basis

  qubit.s_gate_dagger(0)
  qubit.hadamard_gate(0)
  qubit.s_gate_dagger(1)
  qubit.hadamard_gate(1)
  qubit.cnot_gate(1, 0)
  outcome = cast(dict[str, int], qubit.measure(list(range(qubit.num_qubits)), shots))
  return measurement_expectation(outcome, shots)
```

Figure 34: Python helper functions for quantum measurements in different Pauli string bases. Additional functions for measuring in four-qubit bases are available in the code repository for this project.

```
def lipkin_hamiltonian(N: int, eps: float, V: float, W: float) -> np.ndarray:
  """
  Construct the Hamiltonian matrix for the Lipkin model.

  Args:
    N (int): Number of particles in the system.
    eps (float):Energy level spacing.
    V (float): Interaction strength of the Lipkin model.
    W (float): Interaction strength of the pairing force.

  Returns
    H (ndarray): The Hamiltonian matrix.
  """
  j = N / 2
  j_z = np.arange( -j, j + 1, 1)

  dim = len(j_z)
  H = np.zeros((dim, dim))

  # Diagonal elements
  np.fill_diagonal(H, j_z * eps + W * (j**2 - j_z**2))

  # Off -diagonal elements
  for i in range(dim - 2):
    jz = j_z[i]
    coeff = (
      0.5
      * V
      * np.sqrt((j * (j + 1) - jz * (jz + 1)) * (j * (j + 1) - (jz + 1) * (jz + 2)))
    )
    H[i, i + 2] = coeff
    H[i + 2, i] = coeff

  return H
```

Figure 35: Python function for constructing the Hamiltonian matrix for an N-fermion Lipkin model.

```
from concurrent.future import ThreadPoolExecutor
import numpy as np

def energy_expectation_two_fermions(
  angles: np.ndarray,
  interaction_strength: float,
  shots: int,
  eps: float,
  W: float,
) -> float:
  """
  Calculate energy expectation value of the Lipkin model for two fermions.

  Args:
    angles (np.ndarray): Parameter vector used to prepare the ansatz state.
    interaction_strength (float): Interaction strength of the Lipkin model.
    shots (int): Number of measurement shots.
    eps (float): Energy level spacing.
    W (float): Interaction strength of the pairing force.

  Returns:
    float: Energy expectation value.
  """

  # Prepare ansatz
  qubit = prepare_ansatz(angles)

  with ThreadPoolExecutor(max_workers=4) as executor:  # Parallelize
    future_zi = executor.submit(measure_z, qubit.copy(), shots)  # Measure ZI
    future_iz = executor.submit(measure_nth_z, qubit.copy(), 1, shots)  # Measure IZ
    future_xx = executor.submit(measure_xx, qubit.copy(), shots)  # Measure XX
    future_yy = executor.submit(measure_yy, qubit.copy(), shots)  # Measure YY

    try:
      expectation_iz = future_iz.result()
      expectation_zi = future_zi.result()
      expectation_xx = future_xx.result()
      expectation_yy = future_yy.result()
    except Exception as e:
      raise RuntimeError(f"Error in parallel measurement: {str(e)}")

  # Calculate expectation value
  exp_val = (
    0.5 * eps * (expectation_iz + expectation_zi)
    + 0.5 * (W + interaction_strength) * expectation_xx
    + 0.5 * (W - interaction_strength) * expectation_yy
  )

  return exp_val
```

Figure 36: Python function for computing the expectation of a two-fermion Lipkin Hamiltonian using quantum circuit logic.

```
import numpy as np
from qiskit.quantum_info import SparsePauliOp
from qiskit.primitives import Estimator

def qiskit_energy_expectation_four_fermions(
  angles: np.ndarray,
  interaction_strength: float,
  shots: int,
  eps: float,
  W: float,
) -> float:
  # Calculate energy expectation value of the Lipkin model for four fermions.

  # Prepare ansatz
  circuit = qiskit_prepare_ansatz(angles)

  # Construct Hamiltonian using Qiskit's SparsePauliOp
  z_coeff = 0.5 * eps
  xx_coeff = 0.5 * (W - interaction_strength)
  yy_coeff = 0.5 * (W + interaction_strength)

  hamiltionian_terms = [
    ("ZIII", z_coeff),
    ("IZII", z_coeff),
    ("IIZI", z_coeff),
    ("IIIZ", z_coeff),
    ("XXII", xx_coeff),
    ("XIXI", xx_coeff),
    ("XIIX", xx_coeff),
    ("IXXI", xx_coeff),
    ("IXIX", xx_coeff),
    ("IIXX", xx_coeff),
    ("YYII", yy_coeff),
    ("YIYI", yy_coeff),
    ("YIIY", yy_coeff),
    ("IYYI", yy_coeff),
    ("IYIY", yy_coeff),
    ("IIYY", yy_coeff),
  ]
  pauli_strings, coeffs = zip(*hamiltionian_terms)

  hamiltonian = SparsePauliOp(list(pauli_strings), list(coeffs))

  # Calculate expectation value
  estimator = Estimator()
  job = estimator.run(circuit, observables=hamiltonian, shots=shots)
  exp_val = job.result().values[0]

  return exp_val
```

Figure 37: Python function using Qiskit for computing the expectation of a four-fermion Lipkin Hamiltonian using quantum circuit logic. An alternative implementation using the NQubitState class is available in the code repository for this project.