Braiding_project

June 20, 2025

1 FYS5419: Project 2 - Braiding of Poor Mans Majorana (PMM)

1.1 Authors: Hishem Kløvnes, Oskar Idland

Source: https://github.com/hishemok/FYS5419/tree/master/Project_2

1.2 a) Qubits vs Fermions

The usual model of quantum computation is expressed in terms of qubits and Pauli operators X Y and Z. A single fermionic mode is also a two-level system (either occupied or not), but is quite different from a qubit. Two key features are fermionic parity conservation and the anti-commutation relations.

Parity conservation means that we can't make superpositions between states of even and odd fermionic numbers. Therefore, a single fermionic mode can't be used as a qubit. It is a two-level system, but we are not allowed to make superpostions between the two states.

The anti-commutation relations imply that fermionic operators are non-local, which means they do not map cleanly to the local Pauli gates.

Let's take system with two fermionic, c_1 and c_2. The Hilbert space is 4-dimensional, but because of parity conservation, we can consider the two subspaces of even and odd total parity separately. Let's take the odd parity subspace to be concrete. This is a two level system, spanned by the states $+-$ (left mode is unoccupied (even parity) and the right mode is occupied (odd parity)) and $-+$. We can write down the standard Pauli operators in this space, i.e $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

- Express the Pauli operators in terms of the fermions c_1 and c_2.
- Define four hermitian Majorana operators $\gamma_0, \gamma_1, \gamma_2, \gamma_3$ with the mapping $c_k = \frac{\gamma_{2k-2} + i\gamma_{2k-1}}{2}$.
- What are the anti-commutation relations between the Majoranas?

- Express the Pauli operators in terms of Majorana operators.

1.2.1 Pauli gates in terms of fermions

c_i ammihilates a fermion in mode i and c_i creates a fermion in mode i.

We know that the Pauli X gate flips the bit. From this we can express it as:

$$X = c_1c_2 + c_2c_1$$

The Pauli Y gate does an imaginary bitflip

$$Y = -i(c_1c_2 - c_2c_1)$$

And Z changes the sign of the 1 and leaves 0 unchanged

$$Z = c_1c_1 - c_2c_2$$

1.2.2 Define four majorana operators

$$c_1 = \frac{\gamma_0 + i\gamma_1}{2} \quad c_1 = \frac{\gamma_0 - i\gamma_1}{2}$$
$$c_2 = \frac{\gamma_2 + i\gamma_3}{2} \quad c_2 = \frac{\gamma_2 - i\gamma_3}{2}$$

1.2.3 The anti-commutation relations between Majoranas

Hermitian: $\gamma_j = \gamma_j$

Canonical anti-commutation: $\{\gamma_j, \gamma_k\} = 2\delta_{jk}$

Meaning: $\gamma_j^2 = 1$ and $\gamma_j\gamma_k = -\gamma_k\gamma_j$ for jk

1.2.4 Express Pauli operators in terms of Majorana Operators

Firstly X

$$X = c_1c_2 + c_2c_1 = \frac{\gamma_0 - i\gamma_1}{2}\frac{\gamma_2 + i\gamma_3}{2} + \frac{\gamma_2 - i\gamma_3}{2}\frac{\gamma_0 + i\gamma_1}{2}$$
$$= \frac{1}{4}(\gamma_0\gamma_2 + i\gamma_0\gamma_3 - i\gamma_1\gamma_2 + \gamma_1\gamma_3 + \gamma_2\gamma_0 + i\gamma_2\gamma_1 - i\gamma_3\gamma_0 + \gamma_3\gamma_1)$$

Using the anti-commutation rules

$$= \frac{1}{4}(i\gamma_0\gamma_3 - i\gamma_1\gamma_2 + i\gamma_2\gamma_1 - i\gamma_3\gamma_0)$$

$$= \frac{i}{2}(\gamma_0\gamma_3 - \gamma_1\gamma_2)$$

From the odd parity subspace we have the total Parity operator $P_{tot} = i\gamma_0\gamma_1\gamma_2\gamma_3 = -I$. From this we can try to find a relation between $\gamma_0\gamma_3$ and $\gamma_1\gamma_2$.

$$\gamma_0\gamma_1\gamma_2\gamma_3 = I\gamma_0\gamma_1\gamma_2\gamma_3\gamma_3 = \gamma_3\gamma_0\gamma_1\gamma_2 = \gamma_3\gamma_1\gamma_2\gamma_0\gamma_0 = \gamma_3\gamma_0\gamma_1\gamma_2 = \gamma_3\gamma_0 = -\gamma_0\gamma_3$$

Thus X becomes

$$X = -i\gamma_1\gamma_2 \text{ or } X = i\gamma_0\gamma_3$$

Now for Y

$$Y = -i(c_1c_2 - c_2c_1) = -i(\frac{\gamma_0 - i\gamma_1}{2}\frac{\gamma_2 + i\gamma_3}{2} - \frac{\gamma_2 - i\gamma_3}{2}\frac{\gamma_0 + i\gamma_1}{2})$$

$$= \frac{-i}{4}(\gamma_0\gamma_2 + i\gamma_0\gamma_3 - i\gamma_1\gamma_2 + \gamma_1\gamma_3 - \gamma_2\gamma_0 - i\gamma_2\gamma_1 + i\gamma_3\gamma_0 - \gamma_3\gamma_1)$$

$$= \frac{-i}{2}(\gamma_0\gamma_2 + \gamma_1\gamma_3)$$

$$Y = -i\gamma_0\gamma_2 \text{ or } Y = -i\gamma_1\gamma_3$$

And lastly for Z

$$Z = c_1c_1 - c_2c_2 = \frac{\gamma_0 - i\gamma_1}{2}\frac{\gamma_0 + i\gamma_1}{2} - \frac{\gamma_2 - i\gamma_3}{2}\frac{\gamma_2 + i\gamma_3}{2}$$

$$= \frac{1}{4}(\gamma_0\gamma_0 + i\gamma_0\gamma_1 - i\gamma_1\gamma_0 + \gamma_1\gamma_1 - \gamma_2\gamma_2 - i\gamma_2\gamma_3 + i\gamma_3\gamma_2 - \gamma_3\gamma_3)$$

$$= \frac{i}{2}(\gamma_0\gamma_1 + \gamma_2\gamma_3)$$

$$Z = i\gamma_0\gamma_1 \text{ or } Z = i\gamma_2\gamma_3$$

1.2.5 Section a: Summary

We successfully expressed the Pauli matrices in terms of the creation and annhillation operators $\gamma_0, \gamma_1, \gamma_2$ and γ_3

1.3 b) Majorana Exchange gates

Let's take a step towards braiding and define Majorana exchange operators - Find unitaries B_{ij} that exchange the Majoranas γ_i and γ_j, meaning it should act as

$$B_{ij}\gamma_i B_{ij} = \gamma_j$$
$$B_{ij}\gamma_j B_{ij} = -\gamma_i$$

and trivially on other operators. Express the gates in terms of γ_i and γ_j as well as in the basis $+-$, $-+$

- Check the B_{ij}^2 of these operators, how are they related to the Pauli operators?
- Express the Hadamard gate in terms of exchange gates
- Bonus: Do the same relations hold in the even parity sector?

1.3.1 Find unitaries

If $B_{ij} = \sqrt{\frac{i}{2}}(1 - \gamma_i\gamma_j)$, then $B_{ij} = \frac{1-i}{2}(1 + \gamma_i\gamma_j)$ from $(AB) = BA$

In order to express the gates in terms of the basis \$+ - \$, $-+$, I will project the matrix B_{ij} into the 22 odd subspace. In the full 44 Hilbert space, $+- = [0, 1, 0, 0]^T$ and $+- = [0, 0, 1, 0]^T$.

$$B_{ij}^{(2\times 2)} = \begin{pmatrix} +-B_{ij}+- & +-B_{ij}-+ \\ -+B_{ij}+- & -+B_{ij}-+ \end{pmatrix}$$

By doing this, we obtain a few different braiding matrices. Comparing them to the pauli matrices we see that

$$B_{12} = X$$
$$B_{03} = -X$$
$$B_{02} = -Y$$
$$B_{13} = -Y$$
$$B_{01} = Z$$
$$B_{23} = -Z$$

1.3.2 Imports and configs

[6]:
```
import numpy as np
import matplotlib.pyplot as plt
from numba import njit, objmode
from scipy.linalg import expm
from tqdm import tqdm




np.set_printoptions(precision=3, suppress=True)

plt.rcParams.update({
    'font.size': 12,
    "figure.figsize":(8,6),
    "axes.grid": True,
    "grid.color": "#cccccc",
    "grid.linestyle": "--",
    "grid.linewidth": .5,
    "lines.linewidth": 2.5,
    "axes.linewidth": 1
})
```

1.3.3 Braiding operator in terms of Majoranas

[7]:
```
"""Check properties of the Braiding operators"""
def pauli_matrices():
    X = np.array([[0, 1],
                  [1, 0]])

    Y = np.array([[0, -1j],
                  [1j, 0]])
```

```
    Z = np.array([[1, 0],
                  [0, -1]])
    I = np.eye(2)
    return X, Y, Z, I

X, Y, Z, I = pauli_matrices()

Hadamard = np.array([[1, 1], [1, -1]]) / np.sqrt(2)

def majorana_operators():
    """
    Using the standard Majorana representation from JW transformation for simplicity
    """
    γ0 = np.array(np.kron(X,I), dtype=np.complex128)
    γ1 = np.array(np.kron(Y,I), dtype=np.complex128)
    γ2 = np.array(np.kron(Z,X), dtype=np.complex128)
    γ3 = np.array(np.kron(Z,Y), dtype=np.complex128)
    return γ0, γ1, γ2, γ3

def braiding_operators(i,j):
    """
    Returns braiding operators for adjacent Majorana operators
    """
    return np.sqrt(1j/2) * (np.kron(I,I) - i@j)


def test_braiding_ops():
    """
    Test the braiding operators
    """
    γ0, γ1, γ2, γ3 = majorana_operators()
    b01 = braiding_operators(γ0, γ1)
    b12 = braiding_operators(γ1, γ2)
    b23 = braiding_operators(γ2, γ3)
```

```
    # Check if they are unitary
    assert np.allclose(np.eye(4), b01 @ b01.conj().T), "b01 is not unitary"
    assert np.allclose(np.eye(4), b12 @ b12.conj().T), "b12 is not unitary"
    assert np.allclose(np.eye(4), b23 @ b23.conj().T), "b23 is not unitary"

    assert np.allclose(b01 @ γ0 @ b01.conj().T, γ1), "b01 does not braid γ0 to γ1"
    assert np.allclose(b01 @ γ1 @ b01.conj().T, -γ0), "b01 does not braid γ1 to γ0"
    assert np.allclose(b12 @ γ1 @ b12.conj().T, γ2), "b12 does not braid γ1 to γ2"
    assert np.allclose(b12 @ γ2 @ b12.conj().T, -γ1), "b12 does not braid γ2 to γ1"
    assert np.allclose(b23 @ γ2 @ b23.conj().T, γ3), "b23 does not braid γ2 to γ3"
    assert np.allclose(b23 @ γ3 @ b23.conj().T, -γ2), "b23 does not braid gamma3 to gamma2"

    print("All tests passed!")

test_braiding_ops()
```

```
All tests passed!
```

1.3.4 Braiding operators in $+-$, $-+$ basis

[8]:
```
# """ Express B in the \ket{+-}, \ket{+-} basis"""
ket_pm = np.array([0, 1, 0, 0])
ket_mp = np.array([0, 0, 1, 0])

def project_to_odd_subspace(B):
    # Basis vectors for |+-⟩ and |-+⟩
    return np.array([
        [ket_pm.conj() @ B @ ket_pm, ket_pm.conj() @ B @ ket_mp],
        [ket_mp.conj() @ B @ ket_pm, ket_mp.conj() @ B @ ket_mp]
    ])
```

```
def test_projected_braiding_ops():
    #Check if  X = B²_12 = B²_03 / Y = B²_02 = B²_13 / Z = B²_01 = B²_23
    γ0, γ1, γ2, γ3 = majorana_operators()
    # X
    b12 = braiding_operators(γ1, γ2)
    b03 = braiding_operators(γ0, γ3)
    # Y
    b02 = braiding_operators(γ0, γ2)
    b13 = braiding_operators(γ1, γ3)
    # Z
    b01 = braiding_operators(γ0, γ1)
    b23 = braiding_operators(γ2, γ3)




    b12_odd = project_to_odd_subspace(b12)
    b03_odd = project_to_odd_subspace(b03)

    b02_odd = project_to_odd_subspace(b02)
    b13_odd = project_to_odd_subspace(b13)

    b01_odd = project_to_odd_subspace(b01)
    b23_odd = project_to_odd_subspace(b23)

    #square
    b12_odd_sq = b12_odd @ b12_odd
    b03_odd_sq = b03_odd @ b03_odd
    b02_odd_sq = b02_odd @ b02_odd
    b13_odd_sq = b13_odd @ b13_odd
    b01_odd_sq = b01_odd @ b01_odd
    b23_odd_sq = b23_odd @ b23_odd

    assert np.allclose(b12_odd_sq, X), "B²_12 is not X"
```

```
    assert np.allclose(b03_odd_sq, -X), "B²_03 is not -X"
    assert np.allclose(b02_odd_sq, -Y), "B²_02 is not -Y"
    assert np.allclose(b13_odd_sq, -Y), "B²_13 is not -Y"
    assert np.allclose(b01_odd_sq, Z), "B²_01 is not Z"
    assert np.allclose(b23_odd_sq, -Z), "B²_23 is not -Z"
    assert np.allclose(b01_odd@b12_odd@b01_odd,np.exp(1j*np.pi/4)*Hadamard), "B_01 B_12 B_01 is not exp(iπ/4)␣
↪Hadamard"

    print("All tests passed!")
    print("Projected braiding operators:")
    print("B²_12  X")
    print("B²_03  -X")
    print("B²_02  -Y")
    print("B²_13  -Y")
    print("B²_01  Z")
    print("B²_23  -Z")
    print("B_01 B_12 B_01  Hadamard")
test_projected_braiding_ops()
```

```
All tests passed!
Projected braiding operators:
B²_12  X
B²_03  -X
B²_02  -Y
B²_13  -Y
B²_01  Z
B²_23  -Z
B_01 B_12 B_01  Hadamard
```

1.3.5 Section b: Summary

Trough unit testing, we confirmed our nummerical implemenation of the braiding operators to behave as expected. We verified with out analytical calculations from section a)

1.4 c) Jordan-Wigner transform

While we have a map between Majorana operators and Pauli operators acting on a subspace of a specific parity, we need to create yet another map in order to look at the braiding protocol. We saw how four Majoranas can make a qubit. In the braiding protocol, an additional pair of Majoranas will be used to make an exchange. In the braiding protocol, an additional pair of Majoranas will be used to make an exchange. The braiding protocol therefore needs six Majoranas to work. Four to make a qubit, and two extra auxiliary Majoranas to make the braid.

The braiding protocol in [C. W. J. Beenakker] is expressed in terms of four Majoranas. One pair is "half" the qubit and one qubit is the auxiliary ones. To interret the result in terms of a qubit, an additional pair must be added. See [A. Tsintzis, R. S. Souto, K. Flensberg, J. Danon, and M. Leijnse] to see the protocol with all six Majoranas.

To implement the Hamiltonian numerically, we should map the fermions to Pauli operators and tensor products of those. We already did so, but that was only mapping the operators in a specific parity sector. Now we want to be more general and express it in the full Hilbert space, in a version which is simple to implement numerically.

The general form that the Hamiltonian takes during the braiding protocol is

$$H(t) = \sum_{j=1}^{3} i\Delta_j(t)\gamma_0\gamma_j$$

Write down the representation of the Majorana operators in terms of tensor products of local Pauli or spin operators.

[9]:
```
def tensor_product(*args):
    """Compute the tensor product of multiple matrices."""
    result = args[0]
    for mat in args[1:]:
        result = np.kron(result, mat)
    return result

def sigma_site(j, n, operator):
    """Return the operator for site j in a chain of n sites."""
    ops = [I] * n
    ops[j] = operator
    return tensor_product(*ops)


def creation_annihilation(j,n):
```

```
    f_dag = (sigma_site(j, n, X) + 1j * sigma_site(j, n, Y)) * 0.5
    f = (sigma_site(j, n, X) - 1j * sigma_site(j, n, Y)) * 0.5

    Zops = np.eye(2**n)
    for i in range(0, j):
        Zops @= sigma_site(i, n, -Z)


    create = Zops @ f_dag
    annihilate = Zops @ f

    return create, annihilate


def majoranas(n):
    gammas = []
    for j in range(n):
        a_dag, a = creation_annihilation(j,n)

        gamma1 = a_dag + a
        gamma2 = 1j * (a_dag - a)

        gammas.append(gamma1)
        gammas.append(gamma2)
    gammas = np.array(gammas)
    return gammas

def delta(i,j):
    """Kronecker delta function."""
    return i == j

def braiding_operators(i,j):
```

```
    """
    Returns braiding operators for adjacent Majorana operators
    """
    return np.sqrt(1j/2) * (np.kron(I,I) - i@j)


def check_anti_commutation(gammas):
    """Check the anti-commutation relations of Majorana operators."""
    # n = len(gammas)'
    n = len(gammas)
    for i in range(n):
        for j in range(n):
            result = gammas[i] @ gammas[j] + gammas[j] @ gammas[i]
            expected = 2 * delta(i, j) * np.eye(2**(n//2))
            assert np.allclose(result, expected), f"Anti-commutation failed for {i}, {j}. Expected:
 ↪\n{expected},\ngot:\n{result}"

gammas = majoranas(3)
check_anti_commutation(gammas)
```

1.4.1 Section c: Summary

The Jordan-Wigner transformation expresses the creation and annihilation operators in terms of Pauli spin matrices.

$$c_j = \exp\left(+i\pi_{k=0}^{j-1} f_k f_k\right) f_j c_j = \exp\left(-i\pi_{k=0}^{j-1} f_k f_k\right) f_j$$

Where

$$f_j = \frac{(X_j - iY_j)}{2} \quad f_j = \frac{(X_j + iY_j)}{2}$$

and

$$\exp\left(\pm i\pi_{k=1}^{j-1} f_k f_k\right) = \Pi_{k=1}^{j-1} - Z_k$$

And the Majorana operators are expressed as

$$\gamma_{2j-1} = f_j + f_j \quad \gamma_{2j} = i(f_j - f_j)$$

These majorana operators must obey the following anticommutation relations

$$\{\gamma_i, \gamma_j\} = 2\delta_{ij}$$

and they did

1.5 d) Implement the Hamiltonian numerically

With the operators expressed as Pauli operators, they can be implemented numerically by a tensor product.

We also need to express the parameters $\Delta_j(t)$ as a function of time. One suggestion is to parameterize it using the following parameters. - $\Delta_{\max}$ is a large value that the coupling takes on in those steps of the protocol where it is turned on. - $\Delta_{\min}$ is the value it takes when it switches off (in an ideal experiment this is zero) - s some parameter that determines how fast one changes from $\Delta_{\min}$ to $\Delta_{\max}$. The particular shape of the step is not that important, but the steepness should be tunable. - T is the total time of the protocol

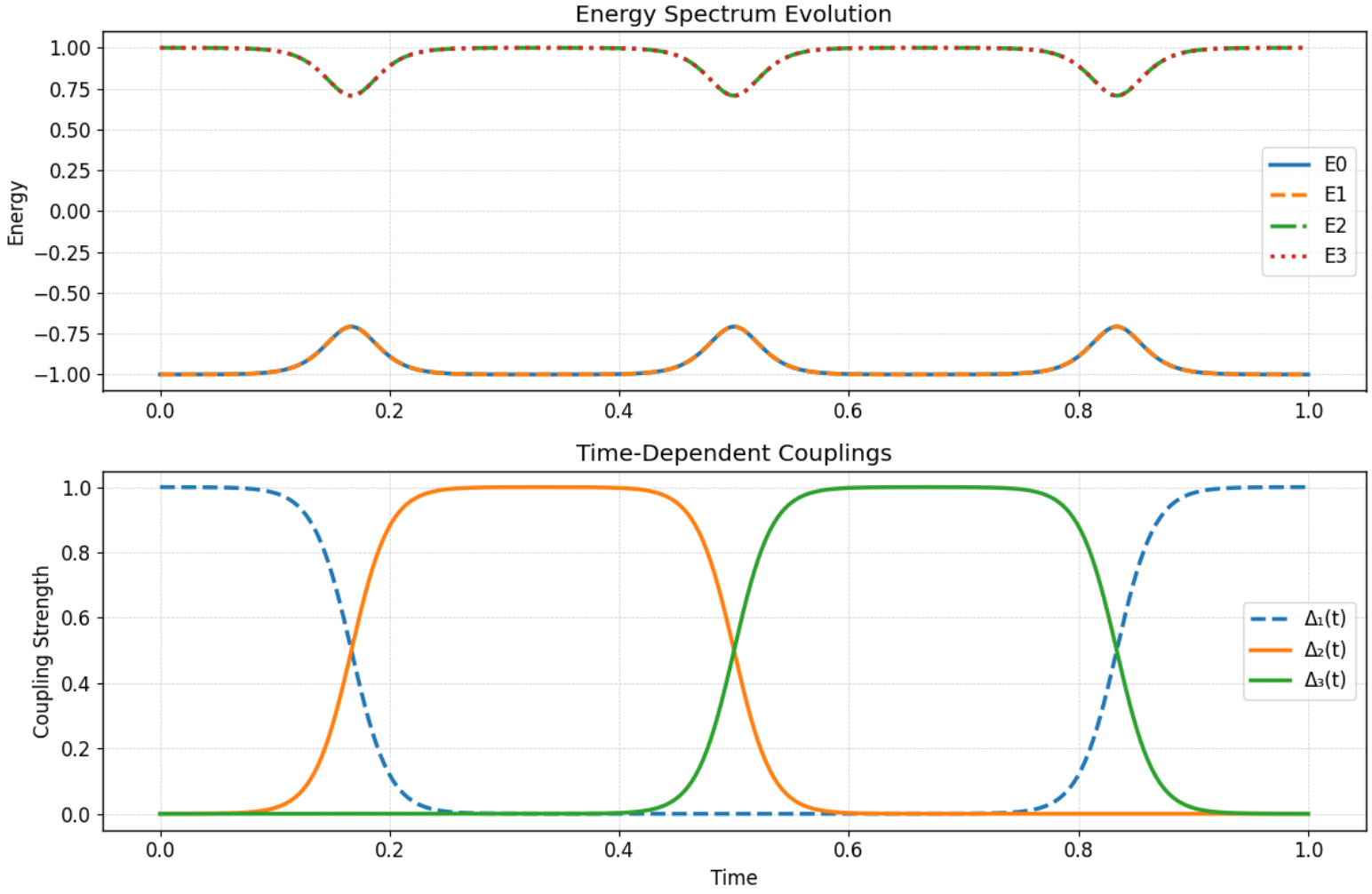
With $H(t)$ implemented, study how the energy spectrum evolves during the protocol. Does it match expectations?

[10]:
```
@njit
def delta_pulse(t, T_peak, width, s, Δ_max, Δ_min):
    """
    Improved smooth delta pulse with controllable width and steepness
    """
    # Calculate rise and fall times
    T_start = T_peak - width/2
    T_end = T_peak + width/2

    # Smooth step functions
    rise = 1/(1 + np.exp(-s*(t - T_start)))
    fall = 1/(1 + np.exp(s*(t - T_end)))

    return Δ_min + (Δ_max - Δ_min) * rise * fall

@njit
def build_hamiltonian(t, T_total, Δ_max, Δ_min, s, width, γ0, γ1, γ2, γ3):
    """
    Constructs the time-dependent Hamiltonian H(t) = Σ Δ_j(t) iγγ_j
```

```
    """

    # Time-dependent couplings
    Δ1 = delta_pulse(t, 0, width, s, Δ_max, Δ_min) + delta_pulse(t, T_total, width, s, Δ_max, Δ_min) - Δ_min
    Δ2 = delta_pulse(t, T_total/3, width, s, Δ_max, Δ_min)
    Δ3 =  delta_pulse(t, 2*T_total/3, width, s, Δ_max, Δ_min)


    # Construct Hamiltonian terms
    H = (Δ1 * 1j * γ0 @ γ1 +
         Δ2 * 1j * γ0 @ γ2 +
         Δ3 * 1j * γ0 @ γ3)


    return H, (Δ1, Δ2, Δ3)


@njit
def analyze_spectrum(T_total, Δ_max, Δ_min, s, width, γ0, γ1, γ2, γ3, n_points=1000):
    times = np.linspace(0, T_total, n_points)
    energies = np.zeros((n_points, 4))  # Store all 4 eigenvalues
    couplings = np.zeros((n_points, 3))  # Store Δ1, Δ2, Δ3

    for i, t in enumerate(times):
        H, couplings[i] = build_hamiltonian(t, T_total, Δ_max, Δ_min, s, width, γ0, γ1, γ2, γ3)

        e_vals = np.linalg.eigvalsh(H)
        energies[i] = e_vals

    return times, energies, couplings

def plot_results(times, energies, couplings):
    plt.figure(figsize=(12, 8))
```

```
    # Plot energy spectrum
    plt.subplot(2, 1, 1)
    linestyles = ['-', '--', '-.', ':']
    for i in range(4):
        plt.plot(times, energies[:, i], label=f'E{i}', linestyle=linestyles[i])
    plt.ylabel('Energy')
    plt.title('Energy Spectrum Evolution')
    plt.legend()

    # Plot couplings
    plt.subplot(2, 1, 2)
    labels = ['Δ(t)', 'Δ(t)', 'Δ(t)']
    linestyles = ['--', '-', '-']
    for i in range(3):
        plt.plot(times, couplings[:, i], label=labels[i], linestyle=linestyles[i])
    # for i in range(3):
    #     plt.plot(times, couplings[:, i], label=labels[i])
    plt.xlabel('Time')
    plt.ylabel('Coupling Strength')
    plt.title('Time-Dependent Couplings')
    plt.legend()

    plt.tight_layout()
    plt.show()


γ0, γ1, γ2, γ3 = majoranas(2)

T_total = 1.0
Δ_max = 1.0
Δ_min = 0
s = 60.0
width = T_total/3
# Parameters
```

```
params = {
    'T_total': T_total,
    'Δ_max': Δ_max,
    'Δ_min': Δ_min,
    's': s,
    'width': width,
    'γ0': γ0,
    'γ1': γ1,
    'γ2': γ2,
    'γ3': γ3,
    'n_points': 1000
}


times, energies, couplings = analyze_spectrum(**params)

# Visualize
plot_results(times, energies, couplings)
```


Energy Spectrum Evolution
Energy
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.2
0.4
0.6
0.8
1.0
E0
E1
E2
E3
Time-Dependent Couplings
Coupling Strength
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Time
Δ₁(t)
Δ₂(t)
Δ₃(t)

1.5.1 Section d: Summary

The Hamiltonian we created had the following form

$$H(t) = \sum_{j=1}^{3} i\Delta_j(t)\gamma_0\gamma_j$$

Where we defined the $\Delta_j(t)$ by using the Sigmoid function to controll the steepness of the change in coupling. We coupled it with a reversed Sigmoid function to make the $\Delta_j(t)$ fall just as smootly as it rose.

The $\Delta_j(t)$ function consists of the following parameters: - Δ_{max} is the (often large) maximum coupling strengh. - Δ_{min} is the minimum coupling strength. This is ideally zero. - s some parameter that determines how fast one changes from Δ_{min} to Δ_{max}. The particular shape of the step is not that important, but the steepness should be tunable. - T_{peak} is the time of the midpoint of the peak of the coupling. - t is the point in time we are evaluating the Hamiltonian at.

This allowed us to easily choose which Δ function we wanted to be active at a given time t.

The above figures shows us that the energy eigenvalues take the expected form. Here we have a twofold degeneracy with values at Δ_{max}, except for at the times where the couplings switches off, and the eigenvalues approach zero, but never reach it. This is expected, because if the eigenvalues are zero, there is a high chance that the ground states will mix with the excited states, which is not what we want.

1.6 e) Implement braiding protocol

Now we are readu to solve the Schrodinger equation

$$\frac{}{t}\psi_t = -iH_t\psi_t$$

The simplest way to do so is to approximate

$$dt\frac{}{t}\psi_t\psi_{t+dt} - \psi_t$$

and express the state at $t + dt$ in terms of the state at t. Then given an inital state, take short steps until reaching the final time T. The numerical error can be monitored by looking at the norm of the state, i.e. $\psi_t\psi_t$. How much this deviates from 1 gives a measure of how accurate the solution is. - Find parameters so that the protocol succeeds and implements the desired exchange gate - Measure parities during the protocol. Do they behave as expected?

[11]:
```
@njit
def adiabaticity_parameter(H_curr,H_prev, ground_vec, excited_vec, ground_energy, excited_energy, dt):
    """
    Calculate the adiabaticity parameter for a given Hamiltonian
    """
```

```
    # Calculate the derivative of the Hamiltonian
    ΔH = (H_curr - H_prev) / dt

    # Calculate the overlap with the eigenvector
    overlap = np.abs(np.vdot(excited_vec, ΔH @ ground_vec))
    ΔE = (excited_energy - ground_energy)**2 + 1e-10  # Avoid division by zero

    return overlap / ΔE



@njit
def evolve_wavefunction(ψ0 ,T_total, Δ_max, Δ_min, s, width,γ0,γ1,γ2,γ3, n_steps=1000):
    dt = T_total / n_steps
    times = np.linspace(0, T_total, n_steps)
    ψ = np.zeros((n_steps, 4), dtype=np.complex128)
    ψ[0] = ψ0

    couplings = np.zeros((n_steps, 3))  # Store Δ1, Δ2, Δ3

    norms = np.zeros(n_steps)
    norms[0] = np.linalg.norm(ψ0)


    evals = np.zeros((n_steps, 4))  # Store all 4 eigenvalues
    e_vecs = np.zeros((n_steps, 4, 4), dtype=np.complex128)  # Store eigenvectors

    ΔE = np.zeros((n_steps, 4), dtype=np.float64)  # Store derrivative of Hamiltonian
    ΔH = np.zeros((n_steps), dtype=np.float64)  # Store derrivative of Hamiltonian

    # Initial Hamiltonian
    H, couplings[0]  = build_hamiltonian(times[0], T_total, Δ_max, Δ_min, s, width, γ0, γ1, γ2, γ3)
    evals[0], e_vecs[0] = np.linalg.eigh(H)
    # ψ] = e_vecs[0,0]
```

```
    for i in range(1, n_steps):
        H_i, couplings[i] = build_hamiltonian(times[i], T_total, Δ_max, Δ_min, s, width, γ0, γ1, γ2, γ3)
        evals[i], e_vecs[i] = np.linalg.eigh(H_i)

        # ΔE[i] = (evals[i] - evals[i-1]) / dt  # Calculate the derivative of the Hamiltonian
        ΔE[i] = adiabaticity_parameter(H_i, H, e_vecs[i,1], e_vecs[i,2], evals[i,1], evals[i,2], dt)
        ΔH[i] = np.sum(np.abs(H_i - H)) / np.abs(dt*evals[i, 0])  # Calculate the derivative of the Hamiltonian

        with objmode(tmp ='complex128[:,:]'):
            tmp = expm(-1j * H_i * dt)

        ψ[i] = tmp @ ψ[i-1]  # Time evolution using matrix exponential

        norm = np.linalg.norm(ψ[i])
        norms[i] = norm

        H = H_i  # Update Hamiltonian for the next step


    return times, ψ, norms, couplings, evals, e_vecs, ΔE, ΔH

def plot_evolution(times, ψ, norms, couplings,evals):
    plt.figure(figsize=(12, 12))

    # Plot wavefunction probabilities
    plt.subplot(4, 1, 1)
    plt.plot(times, np.abs(ψ[:, 0])**2, label=r'$|++⟩$',linestyle='-')
    plt.plot(times, np.abs(ψ[:, 1])**2, label=r'$|+-⟩$',linestyle='--')
    plt.plot(times, np.abs(ψ[:, 2])**2, label=r'$|-+⟩$',linestyle=':')
    plt.plot(times, np.abs(ψ[:, 3])**2, label=r'$|--⟩$',linestyle='-.')
    plt.ylabel('Probability')
    plt.title('Wavefunction Probabilities')
    plt.legend()
```

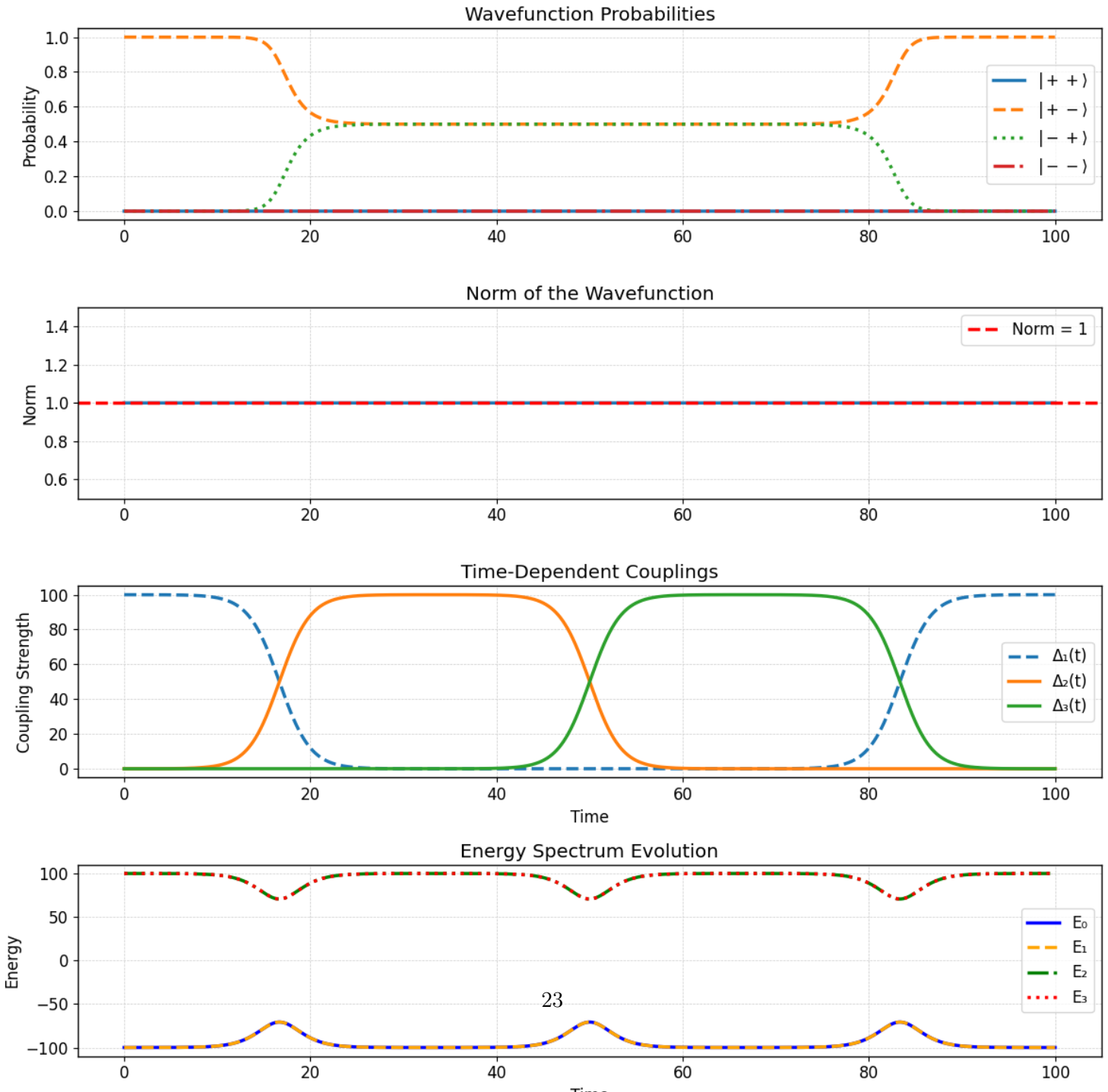
```
# Plot norm
plt.subplot(4, 1, 2)
plt.plot(times, norms)
plt.axhline(1, color='r', linestyle='--', label='Norm = 1')
plt.ylabel('Norm')
plt.ylim(0.5, 1.5)
plt.title('Norm of the Wavefunction')
plt.legend()

# Plot couplings
plt.subplot(4, 1, 3)
labels = ['Δ(t)', 'Δ(t)', 'Δ(t)']
linestyles = ['--', '-', '-']
for i in range(3):
    plt.plot(times, couplings[:, i], label=labels[i], linestyle=linestyles[i])
plt.xlabel('Time')
plt.ylabel('Coupling Strength')
plt.title('Time-Dependent Couplings')
plt.legend()

#plot evals
plt.subplot(4, 1, 4)
linestyles = ['-', '--', '-.', ':']
plt.plot(times, evals[:, 0], label='E', color='blue', linestyle=linestyles[0])
plt.plot(times, evals[:, 1], label='E', color='orange', linestyle=linestyles[1])
plt.plot(times, evals[:, 2], label='E', color='green', linestyle=linestyles[2])
plt.plot(times, evals[:, 3], label='E', color='red', linestyle=linestyles[3])
plt.xlabel('Time')
plt.ylabel('Energy')
plt.title('Energy Spectrum Evolution')
plt.legend()
plt.tight_layout()
plt.show()
```

```
γ0, γ1, γ2, γ3 = majoranas(2)

T_total = 100
Δ_max = 100
Δ_min = 0.0

width = T_total/3
s = 20/width  # steepness parameter, controls the width of the pulse

ψ0= np.array([0,1,0,0],dtype=np.complex128)
ψ0/= np.linalg.norm(ψ0)  # Normalize the initial state
```

1.6.1 Figure e.1

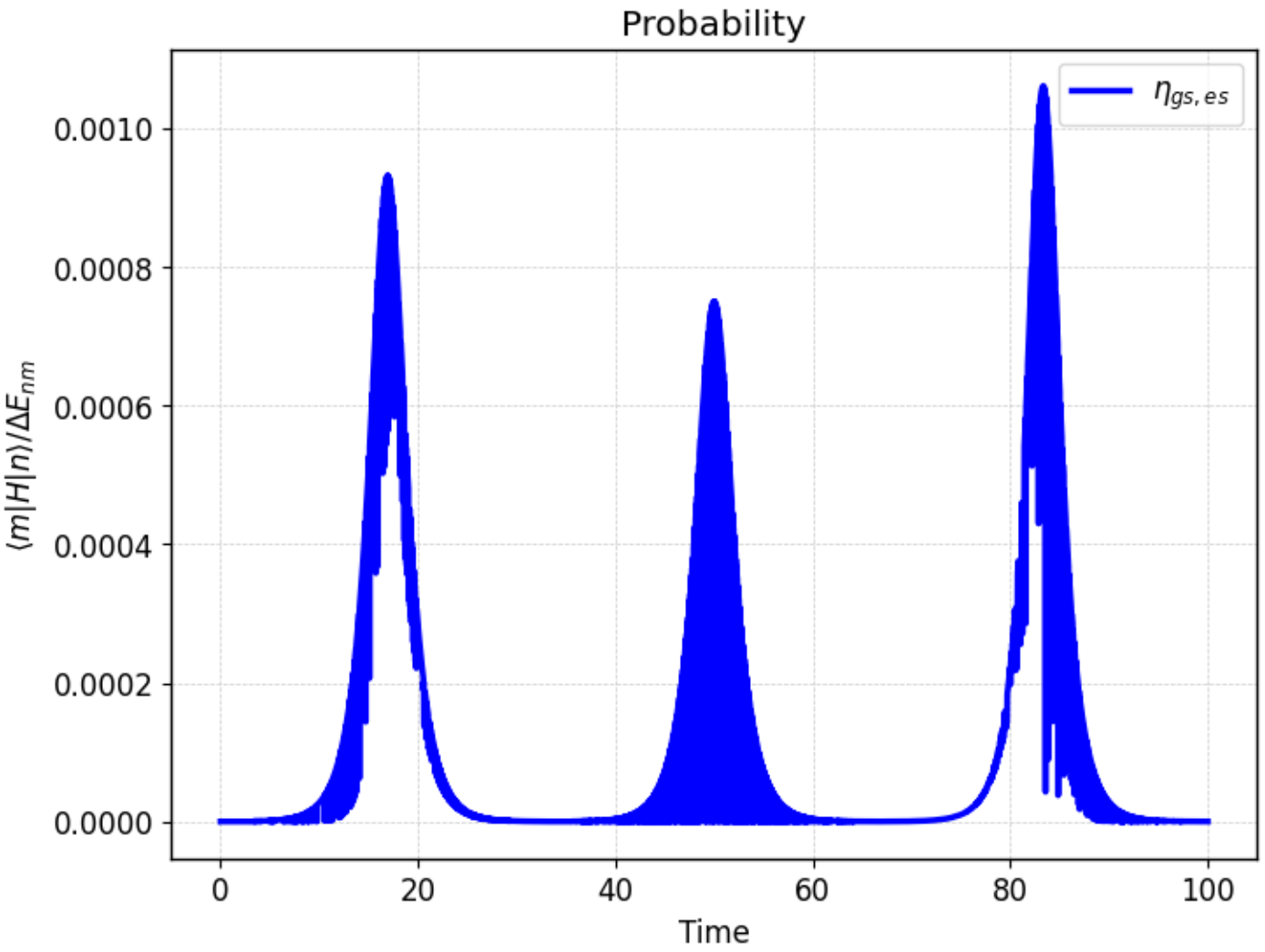
[12]:
```
times, ψ, norms, couplings, evals, evecs, ΔE, ΔH  = evolve_wavefunction(ψ0, T_total, Δ_max, Δ_min, s, width, γ0,␣
↪γ1, γ2, γ3, n_steps=4000)
plot_evolution(times, ψ, norms, couplings,evals)
```

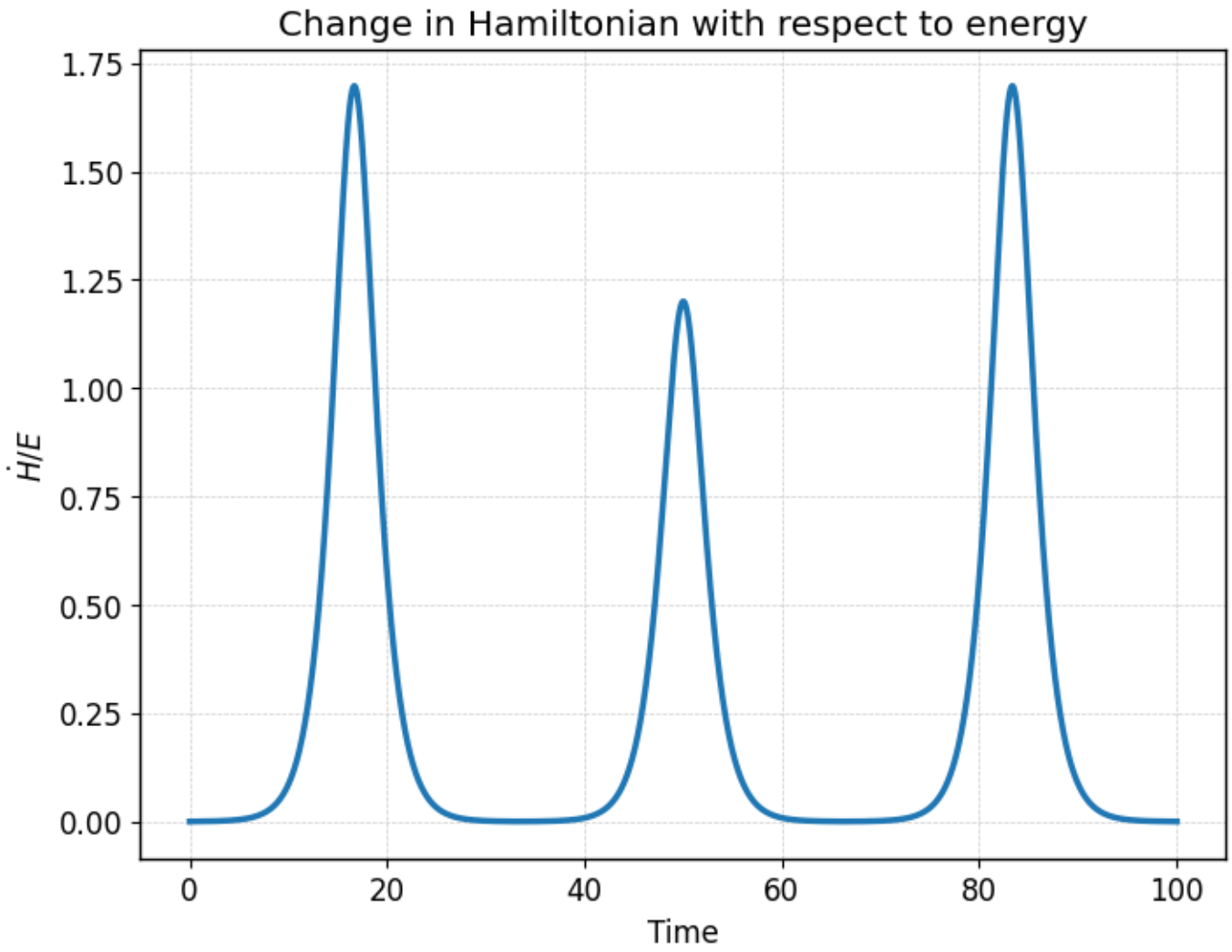
```
/tmp/ipykernel_94568/1270564510.py:52: NumbaPerformanceWarning: '@' is faster on
contiguous arrays, called on (Array(complex128, 2, 'A', False, aligned=True),
Array(complex128, 1, 'C', False, aligned=True))
  ψ[i] = tmp @ ψ[i-1]  # Time evolution using matrix exponential
```


Wavefunction Probabilities
Probability
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
|+ +⟩
|+ −⟩
|− +⟩
|− −⟩
Norm of the Wavefunction
Norm
1.4
1.2
1.0
0.8
0.6
Norm = 1
Time-Dependent Couplings
Coupling Strength
100
80
60
40
20
0
Δ₁(t)
Δ₂(t)
Δ₃(t)
Time
Energy Spectrum Evolution
Energy
50
−50
−100
E₀
E₁
E₂
E₃

1.6.2 Figure e.2

[13]:
```
plt.plot(times, ΔE[:,0], label=r'$η_{gs,es}$', color='blue')
plt.ylabel(r'$⟨m|\dot{H}|n⟩/ΔE_{nm}$')
plt.xlabel('Time')
plt.title('Probability')
plt.legend()
plt.show()



plt.plot(times, ΔH)
plt.ylabel(r'$\dot{H}/E$')
plt.xlabel('Time')
plt.title('Change in Hamiltonian with respect to energy')
plt.show()
```


Probability
$\eta_{gs,es}$
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
$\langle m|\dot{H}|n\rangle/\Delta E_{nm}$
0
20
40
60
80
100
Time

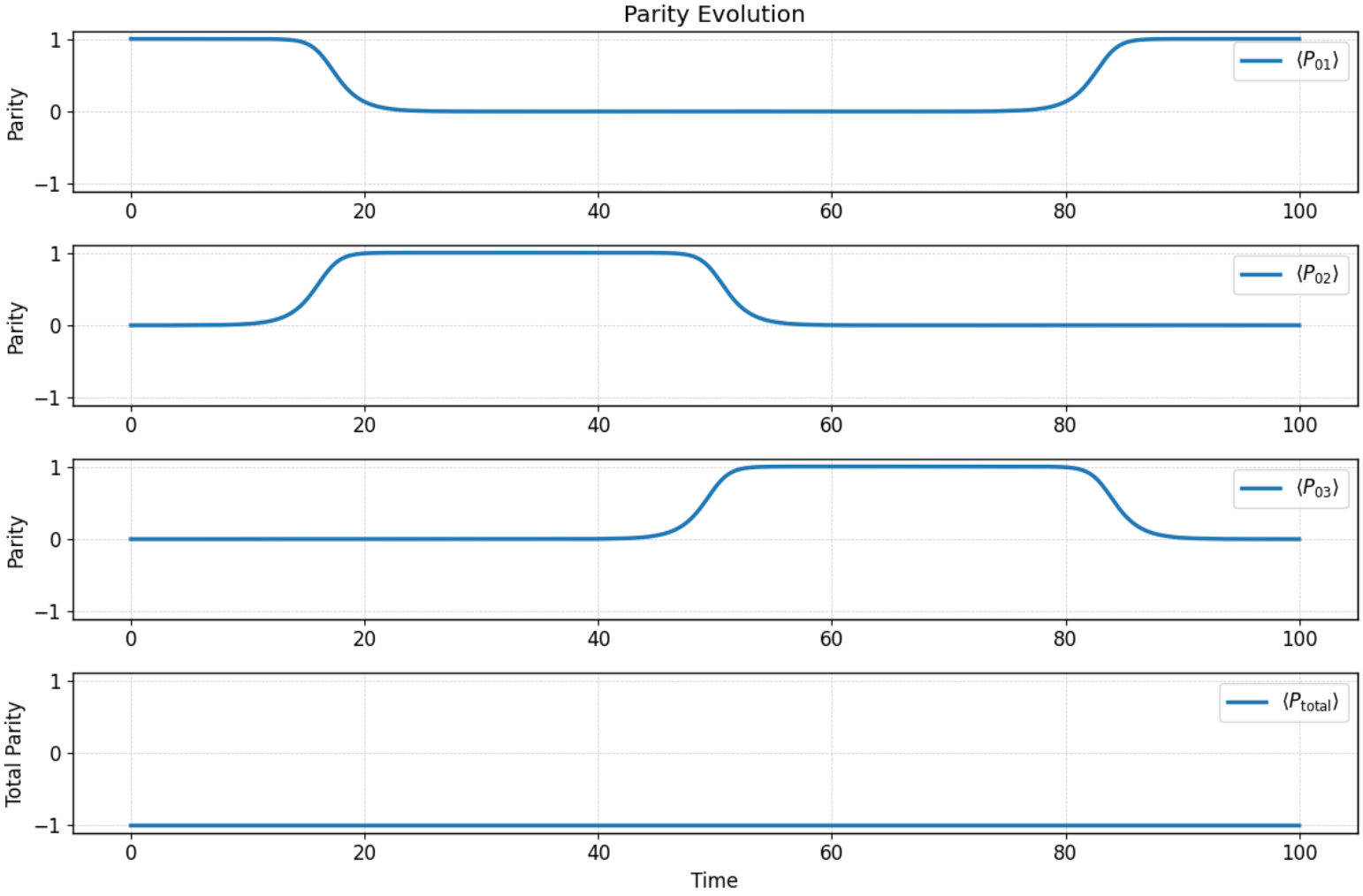

Change in Hamiltonian with respect to energy
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
$\dot{H}/E$
0
20
40
60
80
100
Time

1.6.3 figure e.3

[14]:
```
basis_parities = np.array([+1, -1, -1, +1])

Parities = np.zeros((ψ.shape[0], 4))
op1 = 1j * γ0 @ γ1
op2 = 1j * γ0 @ γ2
op3 = 1j * γ0 @ γ3

for i in range(ψ.shape[0]):
    Parities[i, 0] = (np.vdot(ψ[i], op1@ψ[i]))
    Parities[i, 1] = (np.vdot(ψ[i], op2@ψ[i]))
    Parities[i, 2] = (np.vdot(ψ[i], op3@ψ[i]))
    probs = np.abs(ψ[i])**2  # shape (4,)
    Parities[i, 3] = np.dot(probs, basis_parities)


plt.figure(figsize=(12, 8))

plt.subplot(4,1,1)
plt.ylabel('Parity')
plt.title('Parity Evolution')
plt.plot(times, Parities[:, 0], label=r'$⟨ P_{01} ⟩ $')
plt.ylim(-1.1, 1.1)
plt.legend()

plt.subplot(4,1,2)
plt.ylabel('Parity')
plt.plot(times, Parities[:, 1], label=r'$⟨ P_{02} ⟩ $')
plt.ylim(-1.1, 1.1)
plt.legend()

plt.subplot(4,1,3)
plt.plot(times, Parities[:, 2], label=r'$⟨ P_{03} ⟩ $')
```

```
plt.ylabel('Parity')
plt.ylim(-1.1, 1.1)
plt.legend()

plt.subplot(4,1,4)
plt.plot(times, Parities[:, 3], label=r'$\langle P_\mathrm{total} \rangle$')
plt.ylabel('Total Parity')
plt.xlabel('Time')
plt.ylim(-1.1, 1.1)
plt.legend()

plt.tight_layout()
plt.show()
```

```
/tmp/ipykernel_94568/1156144857.py:9: ComplexWarning: Casting complex values to
real discards the imaginary part
  Parities[i, 0] = (np.vdot(ψ[i], op1@ψ[i]))
/tmp/ipykernel_94568/1156144857.py:10: ComplexWarning: Casting complex values to
real discards the imaginary part
  Parities[i, 1] = (np.vdot(ψ[i], op2@ψ[i]))
/tmp/ipykernel_94568/1156144857.py:11: ComplexWarning: Casting complex values to
real discards the imaginary part
  Parities[i, 2] = (np.vdot(ψ[i], op3@ψ[i]))
```


Parity Evolution
Parity
Parity
Parity
Total Parity
Time
⟨P01⟩
⟨P02⟩
⟨P03⟩
⟨Ptotal⟩
1
0
−1
0
20
40
60
80
100

1.6.4 Section e: Summary

As seen in figure e.1, we keep track of the evolution of the wavefunction, the norm of the wavefunction, the time-dependent couplings and the energy eigenvalues. The norm of the wavefunction is always 1, which is great. The wavefunction ψ is initialized as \$ +-=[0,1,0,0]\$. During the protocol, ψ oscillates between the two states $+-$ and $-+$, and ends up in the state $+-$. The energy eigenvalues behaves as expected, with a twofold degeneracy at $\Delta_{\max}$, just as before. The time-dependent couplings $\Delta_j(t)$ are also as expected, as we have made sure that both the width and the steepness of the couplings are adjusted to fit the protocol, no matter how long it takes to run the protocol.

As seen in figure e.2, we have two measurements. The above plot measures

$$\eta_{gs,es} = \frac{ES\dot{H}GS}{(E_{ES} - E_{GS})}$$

which measures the probability of a ground state being excited. We want this probability to be as low as possible, which in this case is good. In the second plot, we measure $\dot{H}/E_0$, which in general should be as low as possible, for the protocol to be adiabatic.

In figure e.3, we have measured the parity during the protocol. We measure the parity of the state with the different parity operators $P_{ij} = i\gamma_i\gamma_j$. During the protocol we notice that as the coupling $\Delta_j(t)$ is turned on, the parity is 1, which is expected, and the total parity is -1, which is also expected.

1.7 f) Berry phase

If we follow a non-degenerate eigenstate of a Hamiltonian as we slowly (adiabatically) change a parameter in a loop in parameter space, we will end up with the same state up to a phase. The phase has two contributions, one is dynamical and depends on the actual time scale of the evolution and the energy of the state. But there is another contribution that is independent of the details of the evolution. It only depends on the path taken. It is a geometric quantity and comes from a sort of curvature in the space of eigenstates. It is called the Berry phase and can be calculated from the Berry connection $A = id\psi|\psi$ integrated during the evolution. If the state is degenerate, we can get more than a phase. Even for adiabatic evolution, the degenerate states will mix together. Thus, the Berry phase will be represented by a matrix that mixes the states together. The Berry connection is also a matrix with a dimension equal to the number of degenerate states: $A_{nm} = id\psi_n|\psi_m$.

A nice way to calculate non-abelian Berry phase is known as the Kato method. This method uses the projector P that projects on the two degenerate ground states. Then one solves the differential equation

$${}_tU_t = i\left[P,_t P\right] U_t U_0 = I$$

and the final unitary is the non-abelian Berry phase U_T. It shouldn't matter if the total time T is long or short. UT is the gate that a proper adiabatic evolution would implement. Note that the projector P is time-dependent. It can be calculated numerrically by diagonalizing the Hamiltonian, or analytically if the hamitonian is simple enough.

[15]:
```
# from scipy.linalg import expm
# from numba import njit, objmode
## Non-abelian berry phase kato method

def orthonormalize(vs):
    """Orthonormalize columns of vs using QR decomposition."""
    q, _ = np.linalg.qr(vs)
    return q

@njit
def berry_phase_kato(times, evecs):
    n = len(times)
    dt = times[1] - times[0]

    U = np.eye(4, dtype=np.complex128)

    for i in range(n - 1):
        # Pull out the two eigenvectors at step i and i+1
        v1 = evecs[i, :, 0]
        v2 = evecs[i, :, 1]
        w1 = evecs[i+1, :, 0]
        w2 = evecs[i+1, :, 1]

        # Build orthonormal subspace basis
        V = np.column_stack((v1, v2))
        W = np.column_stack((w1, w2))

        # [Optional] Align gauge by maximizing overlap
        # Replace with actual QR or SVD if needed
```

```
        P = V @ V.conj().T
        P_next = W @ W.conj().T

        dPdt = (P_next - P) / dt
        K = P @ dPdt - dPdt @ P

        with objmode(tmp='complex128[:,:]'):
            tmp = expm(-dt * K)

        U = tmp @ U

    return U
```

1.7.1 Section f: Summary

This was simply calculated by using the kato method. Here we created a projector P that projects on the two degenerate ground states. We calculated the time derivative of the projector, and then solved the differential equation using a matrix exponential. The final unitary is the non-abelian Berry phase U_T.

1.8 Part g) Parameter search

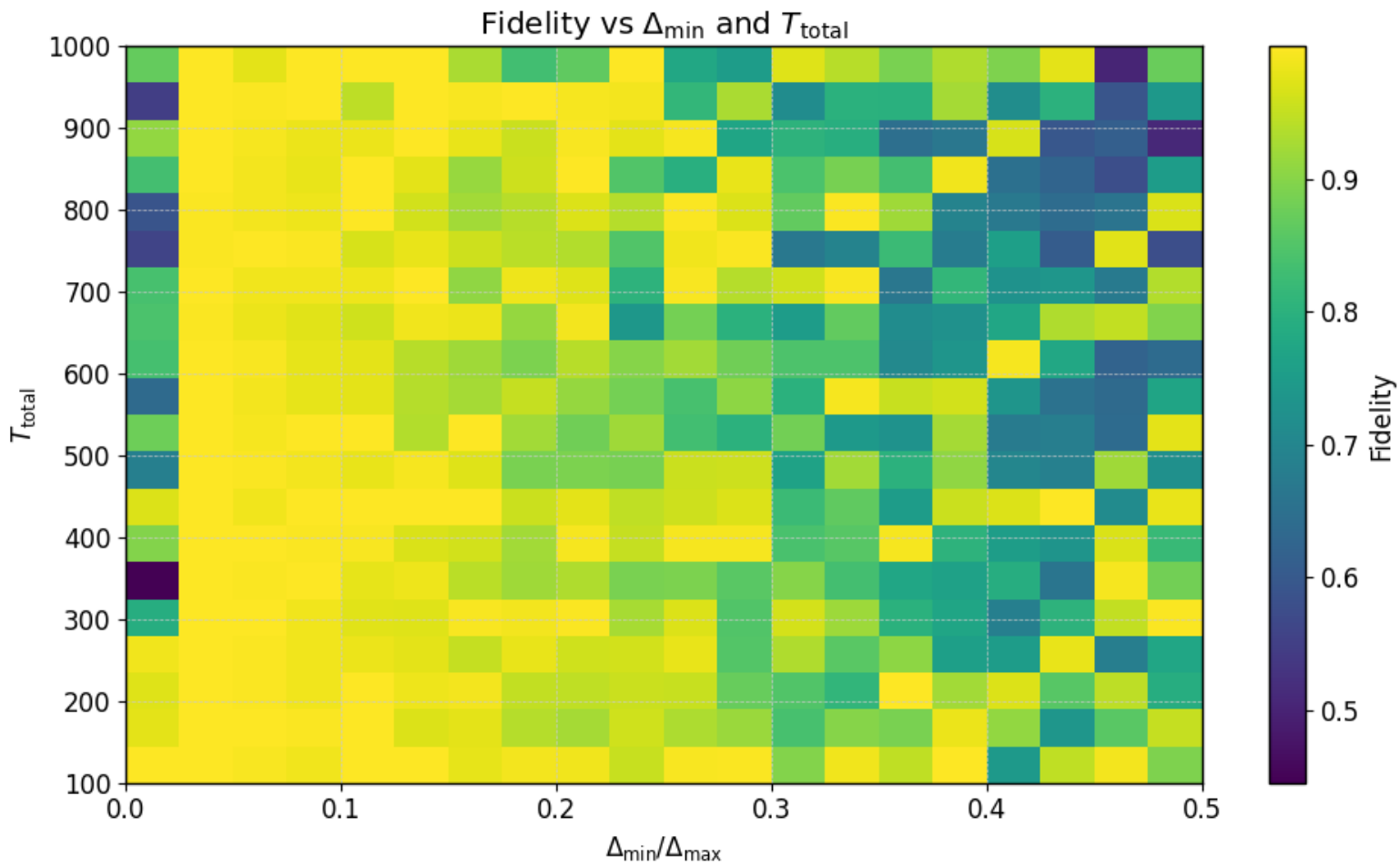
[ ]:
```
### Fidelity
@njit
def fidelity(ψ0, ψT):
    """
    Computes the fidelity between two states
    """
    return np.abs(np.vdot(ψ0, ψT))**2


@njit
def compute_fidelity(Total, s, Δ_max, Δ_min, width, γ0, γ1, γ2, γ3, ψ0, Target_state,n_steps=1000):
    """
    Computes the fidelity for given T and s parameters
```

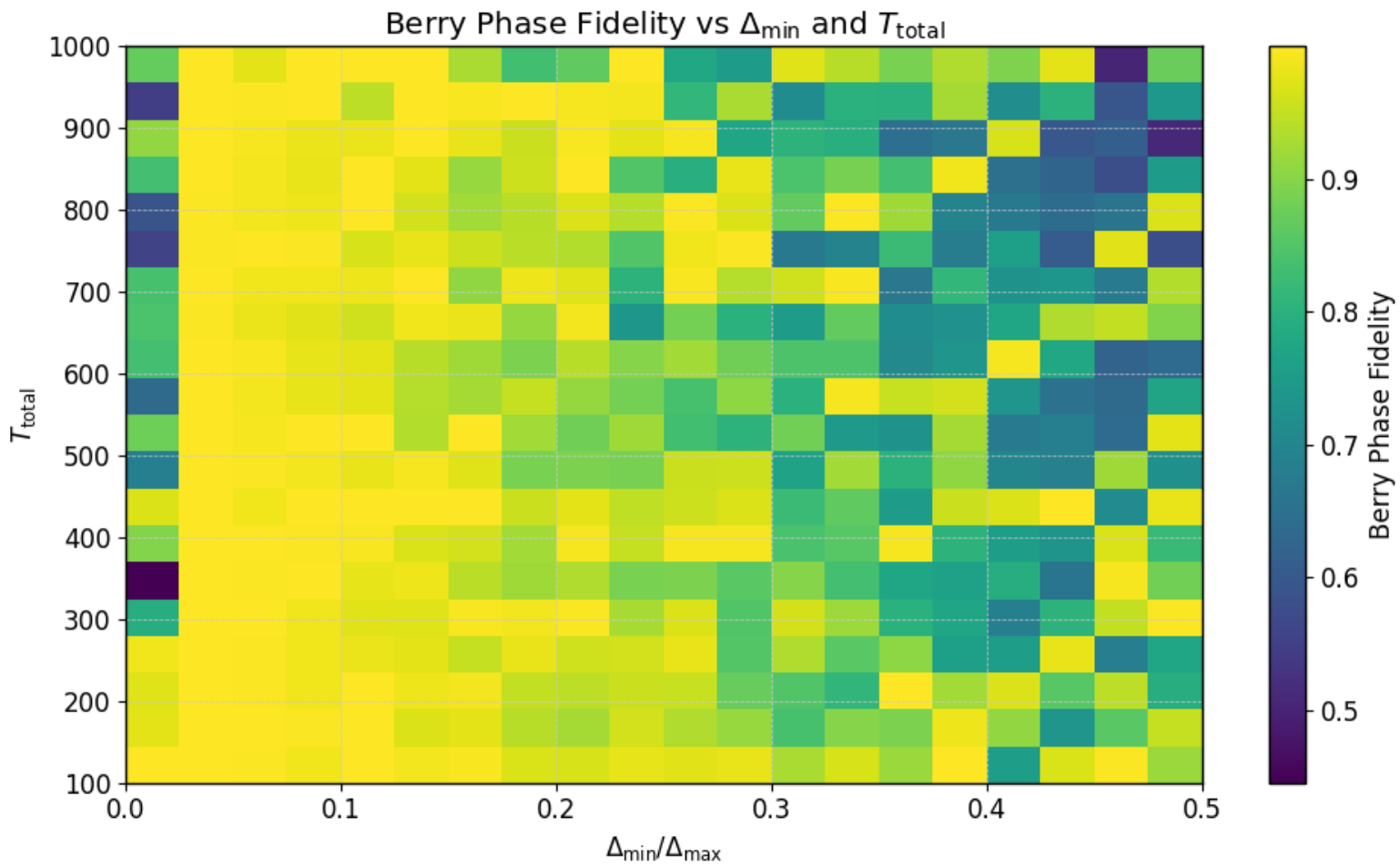
```
    """

    times, ψ, norms, couplings, evals, evecs, DE, DH = evolve_wavefunction(ψ0, T_total=Total, Δ_max=Δ_max,␣
↪Δ_min=Δ_min, s=s, width=width, γ0=γ0, γ1=γ1, γ2=γ2, γ3=γ3, n_steps=1000)

    Fidelity_target = fidelity(ψ[-1], Target_state)


    E1 = evals[:, 1]
    U_B = berry_phase_kato(times, evecs)
    U = np.exp(-1j * T_total * np.trapz(E1))*U_B

    Fidelity_Berry = fidelity(ψ[-1], U @ ψ0)


    return Fidelity_target, Fidelity_Berry, DE, DH

B23 = braiding_operators(γ2, γ3)
Target_state = B23 @ ψ0

Δ_max = 10
Width = T_total/3
T_vals = np.linspace(100, 1000, 20)
dmin_factor = np.linspace(0, 0.5, 20)


Fidelity_targets = []
Fidelity_Berry_targets = []

for T in tqdm(T_vals, desc="Calculating Fidelity and Berry Phase Error for different T_total"):
    for d in dmin_factor:
        Δ_min = d * Δ_max
        width = T/3
        s = 20/width
```

```
        Fidelity_target, Fidelity_Berry , ΔE, ΔH = compute_fidelity(T, s, Δ_max, Δ_min, Width, γ0, γ1, γ2, γ3,␣
↪ψ0, Target_state,n_steps=1000)
        Fidelity_targets.append(Fidelity_target)
        Fidelity_Berry_targets.append(Fidelity_Berry)
```

```
Calculating Fidelity and Berry Phase Error for different T_total:
100%|| 20/20 [01:30<00:00,  4.55s/it]
```

[21]:
```
Fidelity_targets = np.array(Fidelity_targets).reshape(len(T_vals), len(dmin_factor))
Fidelity_Berry_targets = np.array(Fidelity_Berry_targets).reshape(len(T_vals), len(dmin_factor))
plt.figure(figsize=(10, 6))
plt.imshow(Fidelity_targets, extent=(dmin_factor[0], dmin_factor[-1], T_vals[0], T_vals[-1]), aspect='auto',␣
↪origin='lower', cmap='viridis')
plt.colorbar(label='Fidelity')
plt.xlabel(r'$Δ_{\text{min}}/Δ_{\text{max}}$')
plt.ylabel(r'$T_{\text{total}}$')
plt.title(r'Fidelity vs $Δ_{\text{min}}$ and $T_{\text{total}}$')
plt.tight_layout()

plt.show()

plt.figure(figsize=(10, 6))
plt.imshow(Fidelity_Berry_targets, extent=(dmin_factor[0], dmin_factor[-1], T_vals[0], T_vals[-1]),␣
↪aspect='auto', origin='lower', cmap='viridis')
plt.colorbar(label='Berry Phase Fidelity')
plt.xlabel(r'$Δ_{\text{min}}/Δ_{\text{max}}$')
plt.ylabel(r'$T_{\text{total}}$')
plt.title(r'Berry Phase Fidelity vs $Δ_{\text{min}}$ and $T_{\text{total}}$')
plt.tight_layout()
plt.show()
plt.show()
```


Fidelity vs Δ_{min} and T_{total}
1000
900
800
700
600
500
400
300
200
100
T_{total}
0.0
0.1
0.2
0.3
0.4
0.5
$\Delta_{min}/\Delta_{max}$
0.9
0.8
0.7
0.6
0.5
Fidelity



[22]:
```
#print best fidelity and fidelity
min_fidelity_idx = np.unravel_index(np.argmax(Fidelity_targets), Fidelity_targets.shape)
optimal_T = T_vals[min_fidelity_idx[0]]
```

```
optimal_dmin = dmin_factor[min_fidelity_idx[1]]
print(f"Optimal T_total: {optimal_T:.3f}, Optimal Δ_min/Δ_max: {optimal_dmin:.3f}")
print(f"Maximum Fidelity: {Fidelity_targets[min_fidelity_idx]:.3f}")
max_fidelity_idx = np.unravel_index(np.argmax(Fidelity_Berry_targets), Fidelity_Berry_targets.shape)
optimal_T_fidelity = T_vals[max_fidelity_idx[0]]
optimal_dmax_fidelity = dmin_factor[max_fidelity_idx[1]]
print(f"Optimal T_total for Maximum fidelity: {optimal_T_fidelity:.3f}, Optimal Δ_min/Δ_max for Maximum fidelity:
 ↪ {optimal_dmax_fidelity:.3f}")
print(f"Maximum Berry Phase fidelity: {Fidelity_Berry_targets[max_fidelity_idx]:.3f}")
```

```
Optimal T_total: 100.000, Optimal Δ_min/Δ_max: 0.000
Maximum Fidelity: 1.000
Optimal T_total for Maximum fidelity: 100.000, Optimal Δ_min/Δ_max for Maximum
fidelity: 0.000
Maximum Berry Phase fidelity: 1.000
```

[24]:
```
#Find the parameters where both fidelities are above 0.99
above_99 = np.where((Fidelity_targets > 0.999) & (Fidelity_Berry_targets > 0.999))
optimal_Ts = T_vals[above_99[0]]
optimal_dmins = dmin_factor[above_99[1]]
print(f"Parameters where both fidelities are above 0.999:")

for i in range(len(optimal_Ts)):
    T = optimal_Ts[i]
    dmin = optimal_dmins[i]
    fid = Fidelity_targets[above_99][i]
    berry_fid = Fidelity_Berry_targets[above_99][i]
    print(f"T_total: {T:.3f}, Δ_min/Δ_max: {dmin:.3f}, Fidelity: {fid:.4f}, Berry Phase Fidelity: {berry_fid:.
 ↪4f}")
```

```
Parameters where both fidelities are above 0.999:
T_total: 100.000, Δ_min/Δ_max: 0.000, Fidelity: 1.0000, Berry Phase Fidelity:
1.0000
T_total: 147.368, Δ_min/Δ_max: 0.026, Fidelity: 0.9992, Berry Phase Fidelity:
0.9991
```

T_total: 147.368, Δ_min/Δ_max: 0.053, Fidelity: 1.0000, Berry Phase Fidelity:
0.9999
T_total: 194.737, Δ_min/Δ_max: 0.026, Fidelity: 0.9993, Berry Phase Fidelity:
0.9994
T_total: 194.737, Δ_min/Δ_max: 0.105, Fidelity: 0.9994, Berry Phase Fidelity:
0.9994
T_total: 194.737, Δ_min/Δ_max: 0.368, Fidelity: 0.9994, Berry Phase Fidelity:
0.9994
T_total: 336.842, Δ_min/Δ_max: 0.026, Fidelity: 1.0000, Berry Phase Fidelity:
1.0000
T_total: 384.211, Δ_min/Δ_max: 0.026, Fidelity: 0.9997, Berry Phase Fidelity:
0.9997
T_total: 384.211, Δ_min/Δ_max: 0.053, Fidelity: 0.9998, Berry Phase Fidelity:
0.9998
T_total: 431.579, Δ_min/Δ_max: 0.026, Fidelity: 0.9991, Berry Phase Fidelity:
0.9991
T_total: 431.579, Δ_min/Δ_max: 0.105, Fidelity: 1.0000, Berry Phase Fidelity:
1.0000
T_total: 431.579, Δ_min/Δ_max: 0.132, Fidelity: 0.9995, Berry Phase Fidelity:
0.9995
T_total: 431.579, Δ_min/Δ_max: 0.158, Fidelity: 0.9998, Berry Phase Fidelity:
0.9998
T_total: 431.579, Δ_min/Δ_max: 0.447, Fidelity: 0.9997, Berry Phase Fidelity:
0.9997
T_total: 478.947, Δ_min/Δ_max: 0.026, Fidelity: 0.9998, Berry Phase Fidelity:
0.9998
T_total: 526.316, Δ_min/Δ_max: 0.026, Fidelity: 0.9996, Berry Phase Fidelity:
0.9996
T_total: 526.316, Δ_min/Δ_max: 0.158, Fidelity: 0.9999, Berry Phase Fidelity:
0.9999
T_total: 573.684, Δ_min/Δ_max: 0.026, Fidelity: 0.9994, Berry Phase Fidelity:
0.9994
T_total: 621.053, Δ_min/Δ_max: 0.026, Fidelity: 0.9997, Berry Phase Fidelity:
0.9997

```
T_total: 715.789, Δ_min/Δ_max: 0.026, Fidelity: 0.9995, Berry Phase Fidelity:
0.9995
T_total: 715.789, Δ_min/Δ_max: 0.132, Fidelity: 1.0000, Berry Phase Fidelity:
1.0000
T_total: 763.158, Δ_min/Δ_max: 0.053, Fidelity: 0.9995, Berry Phase Fidelity:
0.9995
T_total: 810.526, Δ_min/Δ_max: 0.105, Fidelity: 0.9991, Berry Phase Fidelity:
0.9991
T_total: 857.895, Δ_min/Δ_max: 0.105, Fidelity: 1.0000, Berry Phase Fidelity:
1.0000
T_total: 857.895, Δ_min/Δ_max: 0.211, Fidelity: 0.9990, Berry Phase Fidelity:
0.9990
T_total: 905.263, Δ_min/Δ_max: 0.026, Fidelity: 0.9999, Berry Phase Fidelity:
0.9999
T_total: 905.263, Δ_min/Δ_max: 0.132, Fidelity: 0.9993, Berry Phase Fidelity:
0.9993
T_total: 952.632, Δ_min/Δ_max: 0.026, Fidelity: 0.9997, Berry Phase Fidelity:
0.9997
T_total: 952.632, Δ_min/Δ_max: 0.079, Fidelity: 0.9996, Berry Phase Fidelity:
0.9996
T_total: 1000.000, Δ_min/Δ_max: 0.079, Fidelity: 0.9999, Berry Phase Fidelity:
0.9999
T_total: 1000.000, Δ_min/Δ_max: 0.132, Fidelity: 0.9991, Berry Phase Fidelity:
0.9991
T_total: 1000.000, Δ_min/Δ_max: 0.237, Fidelity: 0.9995, Berry Phase Fidelity:
0.9995
```

1.8.1 Section g: Summary

In this section, we wanted to find the optimal parametsers for the braiding protocol. Earlier we made sure that the s parameter was set to a value that allowed the protocol to run smoothly, only dependent on T. During this grid search, we set $\Delta_{\max}$ to a value that was large and make $\Delta_{\min}$ be a small value compared to $\Delta_{\max}$, with a ratio between 0.1 and 0. This allowed us to measure the accuracy by only variying the T parameter and the the ratio between $\Delta_{\max}$ and $\Delta_{\min}$.

In order to find the accuracy of the braiding protocol, we measured the fidelity of the final state ψ_T . We did this in two ways. First, we

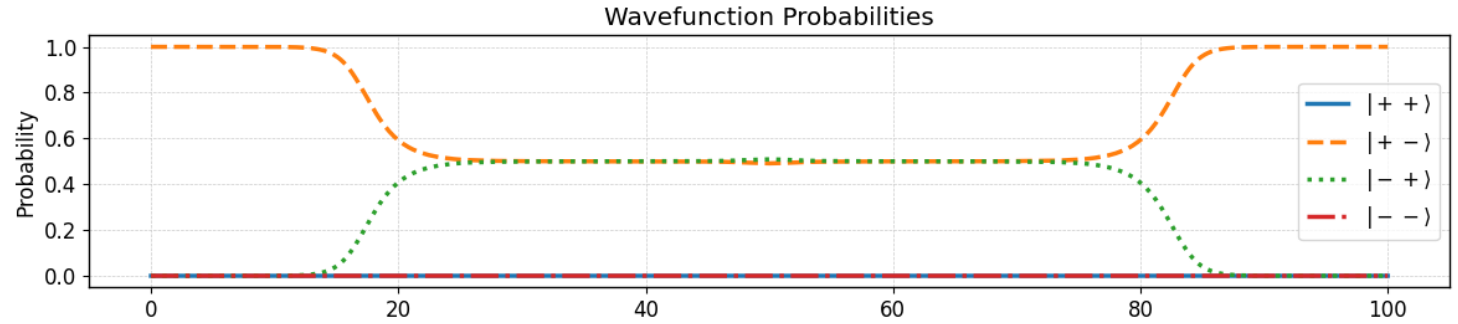
set the final state to be $\psi_T = B_{23}\psi_0$. This measurement tells us how well the final state matches the expected final state. The second way to measure the fidelity was to measure the overlap between the evolved state and the expected state calculated by the Kato method. This can be done by taking the berry state $U = \exp(-idtE_1)U_B$, where U_B is the Kato Berry phase, and applying it to the initial state ψ_0, obtaining the final state $\psi_T = U\psi_0$.

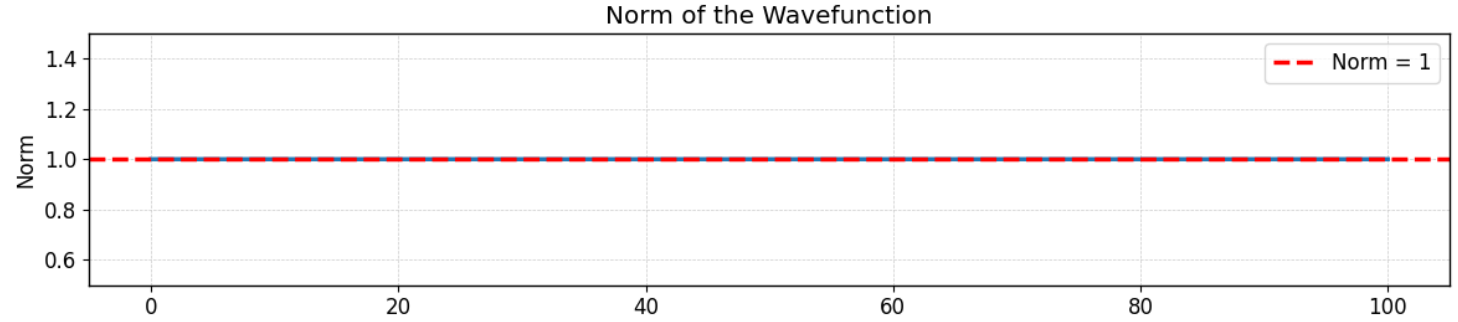
The goal of these two measurements was to find an overlapping region, because this would mean that the braiding protocol gave us the expected result, while at the same time being adiabatic.

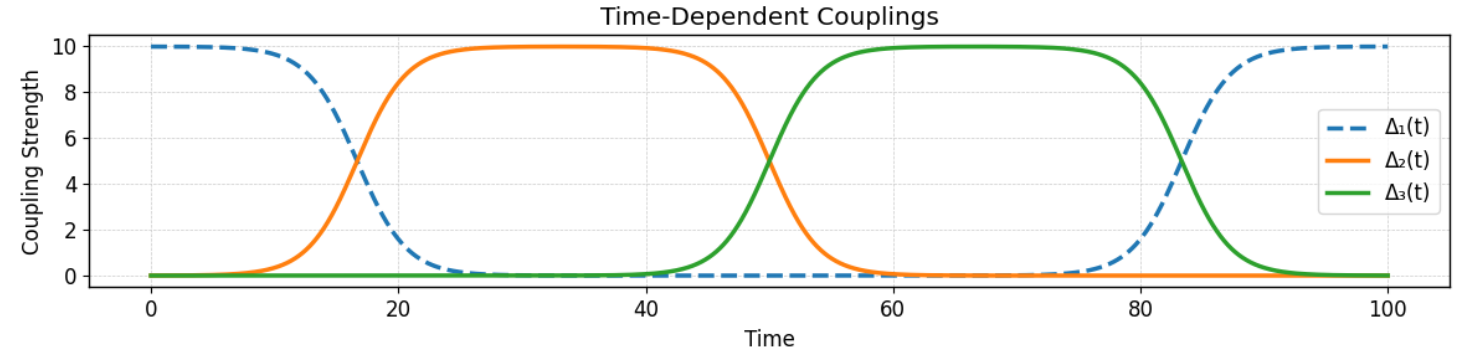
The plot of the target fidelity seems to show us that the braiding protocol prefers a larger total time in general, but for certain ratios between $\Delta_{\max}$ and $\Delta_{\min}$, the total time can be smaller. The plot of the berry fidelity seems chaotic, with no clear pattern or symmetry. This is not what we expected to see, and is most likely due to some fault in the implementation of the Berry phase.

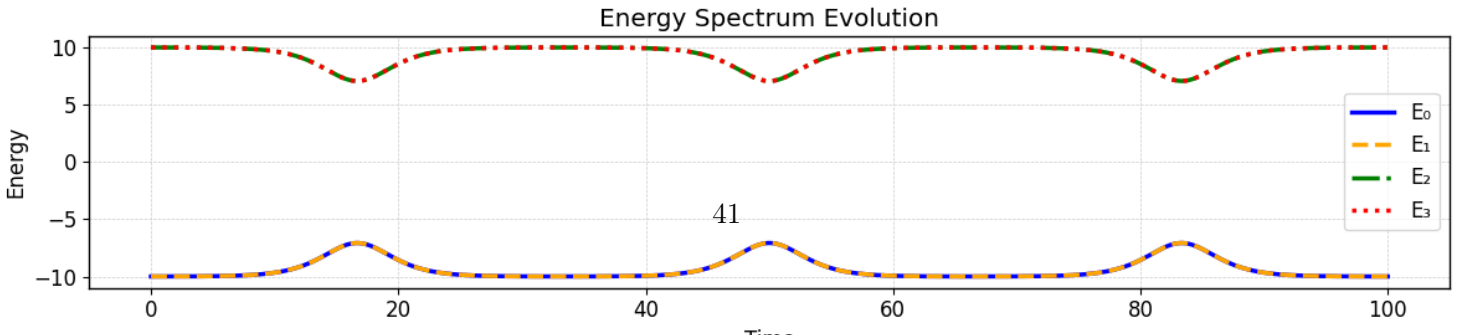
1.8.2 h) Evolve the wavefunction with optimal parameters

[25]:
```
## use the best parameters to replicate the results
T_total = optimal_T
Δ_max = 10
Δ_min = optimal_dmin* Δ_max
s = 50/T_total  # steepness parameter, controls the width of the pulse
width = T_total/3


ψ0 = np.array([0,1,0,0],dtype=np.complex128)
ψ0 /= np.linalg.norm(ψ0)  # Normalize the initial state


times, ψ, norms, couplings, evals, evecs, ΔE, ΔH  = evolve_wavefunction(ψ0, T_total, Δ_max, Δ_min, s, width, γ0,␣
↪γ1, γ2, γ3, n_steps=4000)
plot_evolution(times, ψ, norms, couplings,evals)
```


Wavefunction Probabilities
Probability
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
|+ +⟩
|+ −⟩
|− +⟩
|− −⟩


Norm of the Wavefunction
Norm
1.4
1.2
1.0
0.8
0.6
0
20
40
60
80
100
Norm = 1

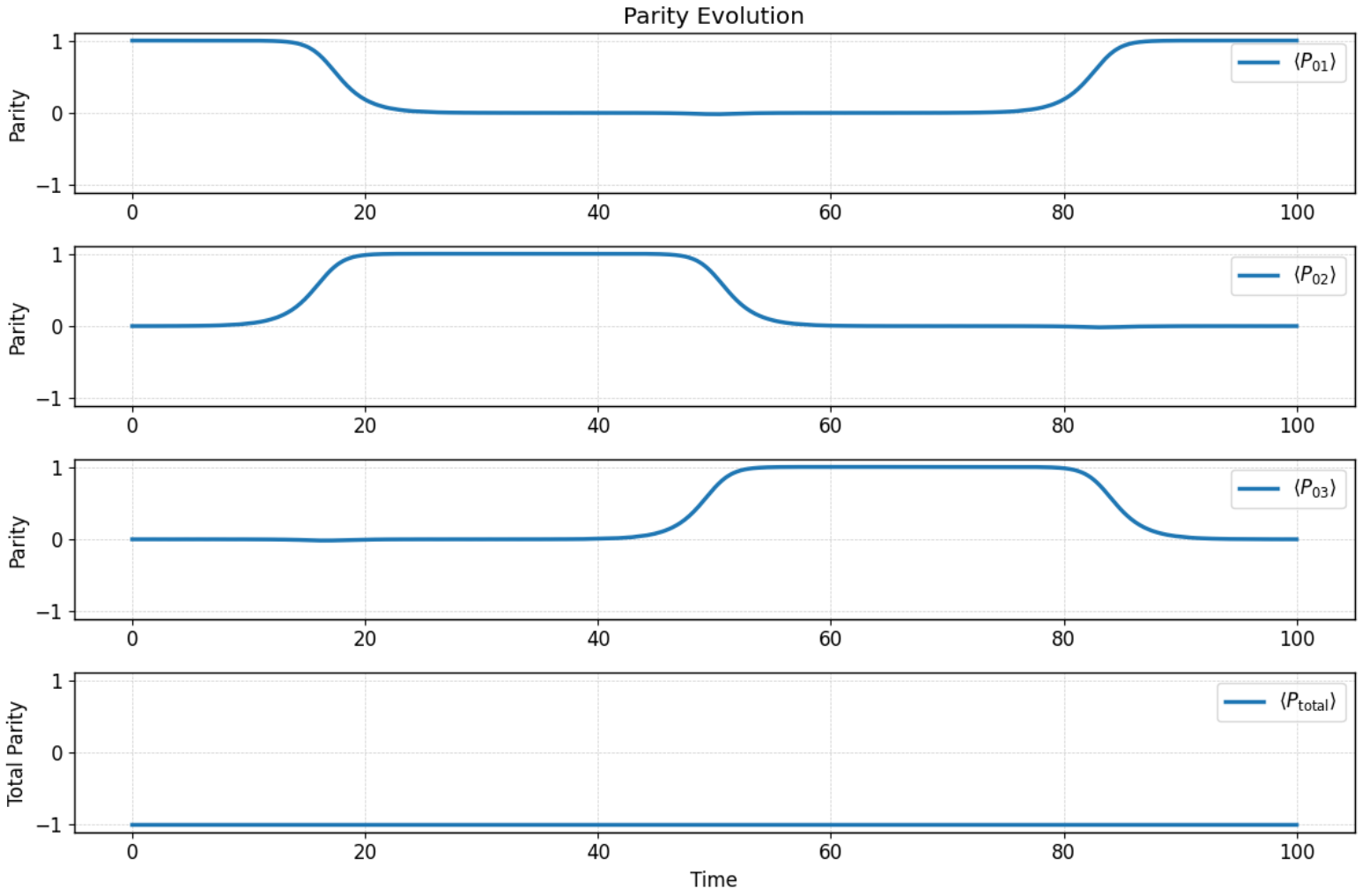

Time-Dependent Couplings
Coupling Strength
10
8
6
4
2
0
0
20
40
60
80
100
Time
Δ₁(t)
Δ₂(t)
Δ₃(t)


Energy Spectrum Evolution
Energy
10
5
0
−5
−10
0
20
40
60
80
100
E₀
E₁
E₂
E₃

[26]:
```
basis_parities = np.array([+1, -1, -1, +1])

Parities = np.zeros((ψ.shape[0], 4))
op1 = 1j * γ0 @ γ1
op2 = 1j * γ0 @ γ2
op3 = 1j * γ0 @ γ3

for i in range(ψ.shape[0]):
    Parities[i, 0] = (np.vdot(ψ[i], op1@ψ[i]))
    Parities[i, 1] = (np.vdot(ψ[i], op2@ψ[i]))
    Parities[i, 2] = (np.vdot(ψ[i], op3@ψ[i]))
    probs = np.abs(ψ[i])**2  # shape (4,)
    Parities[i, 3] = np.dot(probs, basis_parities)


plt.figure(figsize=(12, 8))

plt.subplot(4,1,1)
plt.ylabel('Parity')
plt.title('Parity Evolution')
plt.plot(times, Parities[:, 0], label=r'$⟨ P_{01} ⟩ $')
plt.ylim(-1.1, 1.1)
plt.legend()

plt.subplot(4,1,2)
plt.ylabel('Parity')
plt.plot(times, Parities[:, 1], label=r'$⟨ P_{02} ⟩ $')
plt.ylim(-1.1, 1.1)
plt.legend()

plt.subplot(4,1,3)
```

```
plt.plot(times, Parities[:, 2], label=r'$\langle P_{03} \rangle $')
plt.ylabel('Parity')
plt.ylim(-1.1, 1.1)
plt.legend()

plt.subplot(4,1,4)
plt.plot(times, Parities[:, 3], label=r'$\langle P_\mathrm{total} \rangle$')
plt.ylabel('Total Parity')
plt.xlabel('Time')
plt.ylim(-1.1, 1.1)
plt.legend()

plt.tight_layout()
plt.show()
```

```
/tmp/ipykernel_94568/1466126454.py:9: ComplexWarning: Casting complex values to
real discards the imaginary part
  Parities[i, 0] = (np.vdot(ψ[i], op1@ψ[i]))
/tmp/ipykernel_94568/1466126454.py:10: ComplexWarning: Casting complex values to
real discards the imaginary part
  Parities[i, 1] = (np.vdot(ψ[i], op2@ψ[i]))
/tmp/ipykernel_94568/1466126454.py:11: ComplexWarning: Casting complex values to
real discards the imaginary part
  Parities[i, 2] = (np.vdot(ψ[i], op3@ψ[i]))
```



[ ]:
```
print("Fidelity to target state:", fidelity(ψ[-1], Target_state))
```

```
Fidelity to target state: 0.999999994784438
```

1.9 Results

- We successfully analytically expressed the Pauli operators in terms of Majorana operators.
- We succssfully found the correct braiding operators B_{ij} and verified that they behave as expected.
- We were not able to implement the Jordan-Wigner transformation exactly as we wanted with, but purpously chose to redefine γ_2 as $i(f_1 - f_1)$ in order to get the correct anti-commutation relations.
- The Hamiltonian behaves as we expected with respect to the coupling strengths and the parameters we used.
- During the implementation of the braiding protocol, most of the values behaved as expected. There were some unexpected rapid oscillations in the wavefunction, but we assume it derives from some error in the implementation, but we did not manage to find the reason for it. We also noticed that it depends on the total time T of the protocol, which could make sense since the longer the protocol takes, the more time there is for the wavefunction to oscillate.
- We believe we managed to calculate the Berry phase using the Kato method, as the results matched the fidelity of the target state vs the obtained state.
- We did a grid search for the optimal parameters of the braiding protocol, and found a higher fidelity for larger total times T, while the ratio between $\Delta_{\max}$ and $\Delta_{\min}$ did not seem to matter as much. Except for some cases where in the ratio where we reached a higher fidelity for smaller total times T. The highest fidelity, when measuring the target state vs the obtained state, we reached was 1, with a total time of 952.632 timeunits and $\Delta_{\min}/\Delta_{\max} = 0.047$. While measuring the berry fidelity, we reached a maximum of 0.998, with a total time of 100 timeunits and $\Delta_{\min}/\Delta_{\max} = 0.042$. To ensure that we had a smooth coupling function, we set the width of the coupling to be T/3 and the steepness to be 20/width = 60/T.
- Lastly, we replicated the braiding protocol with the optimal parameters for the Berry phase fidelity, and got a fidelity of 1.000 when measuring the target state vs the obtained state. This means that the braiding protocol was adiabatic and the final state was very close to the expected state.

1.10 Discussion

We are very confident in the time-evolution of the system itself, as we tested both regular Euler method, and the exponential method. Both gave the same results.

The parity measurements behaves as expected when turning on and off the different couplings, indicating that the parity expectation value was implemented correctly.

Lasltly, using the optimal parameters we found, we were able to replicate the braiding protocol with a high fidelity, while also making sure that protocol was adiabatic, therefor indicating that we simulated a successful braiding protocol.

When comparing our results with the results in the paper by [C. W. J. Beenakker]. We did not manage to replicate their results, even though we tried to implement their Hamiltonian as closely as possible. Therefor it difficult to compare our results with theirs. However,

we took inspiration from their braiding protocol and used it as a basis for our own implementation, and some of our analytical calculations were based on their results.

1.11 Conclusion

We are very confident in our implementation of the time-evolution of the system and the braiding protocol. We successfully expressed the Pauli operators in terms of Majorana operators, and found the correct braiding operators B_{ij}. We succsesfully implement the Jordan-Wigner transformation. Overall, we believe that we managed to implement the braiding protocol successfully, and that we simulated a successful braiding protocol with a high fidelity. We also managed to find the optimal parameters for the braiding protocol, which allowed us to replicate the braiding protocol with a high fidelity and an adiabatic evolution.

1.12 Further work

We would then like to proceed with implementing a couple more braiding protocols, in order to replicate some other Pauli gates in terms of braiding. This would allow us to see how well the braiding protocol works in general. This seems like the natural next step, as we have already implemented the braiding protocol for half a Z gate, and we could try to implement the braiding protocol for a full Z gate, and other gates as well.

1.13 References

- [1] C. W. J. Beenakker, ‘‘Search for non-Abelian Majorana braiding statistics in superconductors,’’ SciPost Phys. Lect. Notes, p
- [2] S. B. Bravyi, ‘‘Majorana fermions and topological quantum computation,’’ Annals of Physics, vol. 326, no. 1, pp. 206-217, Ja
- [3] K. Vilkelis, A. L. R. Manesco, J. D. Torres Luna, S. Miles, M. Wimmer, and A. R. Akhmerov, ‘‘Fermionic quantum computation
- [4] A. Tsintzis, R. S. Souto, K. Flensberg, J. Danon, and M. Leijnse, ‘‘Majorana Qubits and Non-Abelian Physics in Quantum Dot-
- [5] K. Chhajed, ‘‘From Ising Model to Kitaev Chain: An Introduction to Topological Phase Transitions,’’ Resonance, vol. 26, no.
- [6] F. Hassler, ‘‘Topological quantum computing.’’ [Online]. Available: https://arxiv.org/abs/2410.13547
- [7] T. Kato, "On the adiabatic theorem of quantum mechanics," Department of Physics, University of Tokyo, 1950.
- [8] K. Snizhko, R. Egger, and Y. Gefen, ‘‘Non-Abelian Berry phase for open quantum systems: Topological protection vs geometric