Physics Undergraduate Research (PHYS497)

Alfaifi, Ammar – 201855360

Nov. 2023

1 Instroduction

This paper shows the idea of relizaion of braiding of Majorana fermions but as series of measurement istead. And then using Jordan-Wigner transformation to write Majorana operators in terms of spin (fermionic) system. Hence, we can use methods provided by `qiskit` package to simulate such an idea and system.

2 As Series of Measurement

As a demostration for the idea, we'll start with a system of 4 Majorana fermions, corresponding to two fermions. The configuration of Majorana fermions is shown in fig. The true braiding operator between γ_0 and γ_3 is given by

$$U = e^{\frac{\pi}{4}\gamma_0\gamma_3} \tag{1}$$

Then, to relize this braiding operator as just series of measurement we do this in four steps:

1. $(1 + i\gamma_1\gamma_2)$
2. $(1 + i\gamma_1\gamma_0)$
3. $(1 + i\gamma_3\gamma_1)$
4. $(1 + i\gamma_1\gamma_2)$

3 Jordan-Wigner Transformation

We shall redefine the our γ s in term of fermionic spin operators, giving us a way to model this system in much more fimiliar systems, such as qubits in quantum computing information. So we'll have:

- $\gamma_0 = Z_0$
- $\gamma_1 = X_0 Z_1$
- $\gamma_2 = X_0 Y_1$
- $\gamma_3 = Y_0$

Note: tensor product is understod, if there is one gate, tensor product with indentity of that subsystem is implicit. Then 4-step series of measurement on the system becomes

1. $(1 + iX_0Z_1X_0Y_1) = (1 + X_1)$
2. $(1 + iX_0Z_1Z_0) = (1 + Y_0Z_1)$
3. $(1 + iY_0X_0Z_1) = (1 + Z_0Z_1)$
4. $(1 + X_1)$

Also for true braiding operator we get

$$e^{i\frac{\pi}{4}X_0} = \frac{1}{\sqrt{2}}(1 + iX_0) \quad \text{or} \quad e^{-i\frac{\pi}{4}X_0} = \frac{1}{\sqrt{2}}(1 - iX_0) \tag{2}$$

4 Applying all Measurements

Let's understand the possible outcomes from the general case of the measurement operator, that is,

$$(1 + S_3X_1)(1 + S_2Z_0Z_1)(1 + S_1Y_0Z_1)(1 + S_0X_1) \tag{3}$$

Expanding the middle two factors as

$$(1 + S_3X_1)(1 + S_2Z_0Z_1 + S_1Y_0Z_1 + S_2S_1Z_0Z_1Y_0Z_1)(1 + S_0X_1) \tag{4}$$

Utilizing the Pauli gates anitcommutation relations, we move the LHS factor to RHS, as for first term we get

$$(1 + S_3X_1)(1 + S_0X_1) = \delta_{S_0,S_3}(1 + S_0X_1)$$

For second term,

$$(1 + S_3X_1)S_2Z_0Z_1(1 + S_0X_1) = \delta_{S_0,-S_3}S_2Z_0Z_1(1 + S_0X_1)$$

For the 3rd term,

$$(1 + S_3X_1)S_1Y_0Z_1(1 + S_0X_1) = \delta_{S_0,-S_3}S_1Y_0Z_1(1 + S_0X_1)$$

For the 3rd term,

$$(1 + S_3X_1)S_2S_1Z_0Z_1Y_0Z_1(1 + S_0X_1) = \delta_{S_0,S_3} - iX_0S_2S_1(1 + S_0X_1)$$

5 Constructing Protocol

Now, we'll investigate the protocol classic outcomes, then we shall decide based on it whether we did relize a braiding between γ_0 & γ_3, if not, what operators to apply to fix it. From Section 4, we simplify it to

$$[\delta_{S_0,S_3} + \delta_{S_0,-S_3}S_2Z_0Z_1 + \delta_{S_0,-S_3}S_1Y_0Z_1 + \delta_{S_0,S_3} - iX_0S_2S_1](1 + S_0X_1)$$

Let's study different cases:

Case 1: $S_0 = S_3$ We get

$$[1 - iX_0S_2S_1](1 + S_0X_1)$$

Note, the right factor just acts on subsystem 1 that we don't care about it outcomes.

Case 1.1: $S_1 = -S_2$

$$[1 + iX_0](1 + S_0X_1)$$

relizing counterclockwise braiding operator in Equation 2.

Case 1.2: $S_1 = S_2$

$$[1 - iX_0](1 + S_0X_1)$$

relizing clockwise braiding operator in Equation 2.

Case 2: $S_0 \neq S_3$ We get

$$[S_2Z_0Z_1 + S_1Y_0Z_1]\,(1 + S_0X_1)$$

let's factor out Z_0Z_1

$$Z_0Z_1[S_2 - iS_1X_0]\,(1 + S_0X_1)$$

In this case we always want to multiply by Z_0, then we'll have

Case 2.1: $S_1 = S_2$

$$S_1Z_0Z_1[1 - iX_0](1 + S_0X_1)$$

relizing the inverse braiding operator

Case 2.2: $S_1 = -S_2$

$$S_1Z_0Z_1[S_1S_2 - iX_0](1 + S_0X_1)$$

But $S_1S_2 = -1$, then

$$-S_1Z_0Z_1[1 + iX_0](1 + S_0X_1)$$

relizing the braiding operator

6 Statevector and Density Matrix

Assume you have a state vector $|\psi\rangle$, which is a column vector with complex numbers:

$$|\psi\rangle = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_n \end{bmatrix} \tag{5}$$

Here, n is the dimensionality of the quantum system. Take the conjugate transpose of the state vector, denoted by $\langle\psi|$. This is obtained by taking the complex conjugate of each element and then transposing the vector:

$$\langle\psi| = \begin{bmatrix} \psi_1^* & \psi_2^* & \dots & \psi_n^* \end{bmatrix}^T \tag{6}$$

Compute the outer product to obtain the density matrix ρ:

$$\rho = |\psi\rangle\langle\psi| = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_n \end{bmatrix} \begin{bmatrix} \psi_1^* & \psi_2^* & \dots & \psi_n^* \end{bmatrix} \tag{7}$$

The result is a square matrix of size $n \times n$, representing the density matrix. The density matrix ρ is Hermitian (equal to its conjugate transpose) and positive semi-definite. If the state vector $|\psi\rangle$ is normalized (has a magnitude of 1), then the trace of the density matrix is equal to 1, which is a requirement for a valid density matrix representing a physical state.

Pure State:

For a pure state density matrix ρ, which satisfies $\rho^2 = \rho$ (idempotent), you can find the state vector $|\psi\rangle$ by:

$$|\psi\rangle = \sqrt{\lambda} \cdot |\phi\rangle$$

Here, λ is the non-zero eigenvalue of ρ, and $|\phi\rangle$ is the corresponding eigenvector.

Mixed State: For a mixed state density matrix ρ with multiple non-zero eigenvalues, it is not possible to uniquely determine a single state vector. The system is in a statistical mixture of pure states. However, you can find a set of state vectors and their corresponding probabilities:

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|$$

Here, $|\psi_i\rangle$ are the eigenvectors of ρ, and p_i are the corresponding eigenvalues. The state vectors represent the pure states in the mixture, and the probabilities p_i give the weight of each pure state in the mixture.

Implementation: To implement this in practice, you can use a numerical linear algebra library (such as NumPy for Python) to compute the eigenvalues and eigenvectors of the density matrix. Depending on the specific form of the density matrix, you might need to use different methods. Keep in mind that this process may not be unique, especially for mixed states, where different sets of pure states and probabilities can lead to the same density matrix.

```
def to_statevector(self, atol: float | None = None, rtol: float | None = None) -> State
    """Return a statevector from a pure density matrix.

    Args:
        atol (float): Absolute tolerance for checking operation validity.
        rtol (float): Relative tolerance for checking operation validity.

    Returns:
        Statevector: The pure density matrix's corresponding statevector.
            Corresponds to the eigenvector of the only non-zero eigenvalue.

    Raises:
        QiskitError: if the state is not pure.
    """

    if atol is None:
        atol = self.atol
    if rtol is None:
        rtol = self.rtol

    if not is_hermitian_matrix(self._data, atol=atol, rtol=rtol):
        raise QiskitError("Not a valid density matrix (non-hermitian).")

    evals, evecs = np.linalg.eig(self._data)

    nonzero_evals = evals[abs(evals) > atol]
    if len(nonzero_evals) != 1 or not np.isclose(nonzero_evals[0], 1, atol=atol, rtol=r
        raise QiskitError("Density matrix is not a pure state")

    psi = evecs[:, np.argmax(evals)]  # eigenvectors returned in columns.
    return Statevector(psi)

@classmethod
def from_instruction(cls, instruction: Instruction | QuantumCircuit) -> DensityMatrix:
    """Return the output density matrix of an instruction.

    The statevector is initialized in the state :math:`|{0,\\ldots,0}\\rangle` of
    the same number of qubits as the input instruction or circuit, evolved
    by the input instruction, and the output statevector returned.

    Args:
        instruction (qiskit.circuit.Instruction or QuantumCircuit): instruction or circ

    Returns:
        DensityMatrix: the final density matrix.
```

```
    Raises:
        QiskitError: if the instruction contains invalid instructions for
                     density matrix simulation.
    """
    # Convert circuit to an instruction
    if isinstance(instruction, QuantumCircuit):
        instruction = instruction.to_instruction()
    # Initialize an the statevector in the all |0> state
    num_qubits = instruction.num_qubits
    init = np.zeros((2**num_qubits, 2**num_qubits), dtype=complex)
    init[0, 0] = 1
    vec = DensityMatrix(init, dims=num_qubits * (2,))
    vec._append_instruction(instruction)
    return vec
```