5-qubit Quantum Error Correction Code in Q#

Leonard Woody

December 5, 2024

Author's Note: My original proposal was too big in scope, so I decided to go with a topic in the Research Paper assignment: #4 Quantum Error-Correcting Codes (QECC) - The 5-qubit code. I chose Q# since it is a Microsoft language and I am a Microsoft employee.

The 5-qubit quantum error correction code is sometimes referred to as the "perfect code" as it is the smallest amount of physical qubits that can detect and correct errors for one logical qubit. Before delving into the code, I think it would be helpful to look at error correction holistically, starting with classical error correction.

1 Classical Error Correction

Error Correction is a subject within Information Theory. Claude Shannon started Information Theory with the publication of his paper, "A Mathematical Theory of Communication" in 1948. Below is the first figure from that paper [Sha48]:

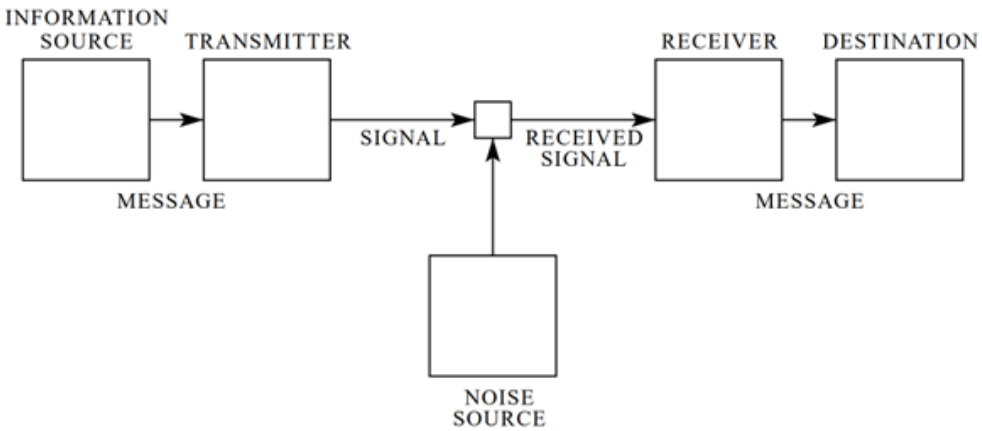

Figure 1: Original Figure from C. Shannon's Information Theory paper.

For us the transmitter will be an encoding scheme or encoder, the signal will be a codeword and the receiver will be a decoding scheme or decoder.

There are many types of error correction codes, but we will be looking at **linear codes**. Therefore, linear algebra will be used to quickly summarize the definitions we will need to make connections to quantum error correction. Please note that we are only concentrating on binary codes so our vector spaces will be $\mathbb{F}_2^n$,which are vectors of length n over the binary field $\mathbb{F}_2$. All arithmetic is modulo 2.

We start with a message, $\mathbf{m}$, represented as a vector of length k. To encode $\mathbf{m}$, we add redundancy through an **encoder**, resulting in a **codeword**, $\mathbf{c}$, of length n. An **error**, $\mathbf{e}$, from a noise source may alter $\mathbf{c}$, so the receiver gets a corrupted signal:

$$\mathbf{y} = \mathbf{c} + \mathbf{e}.$$

The signal $\mathbf{y}$ is **decoded**, using the redundancy in $\mathbf{c}$ to detect and/or correct errors. After decoding, the result is $\mathbf{m}'$, an estimate of the original message $\mathbf{m}$. Ideally, $\mathbf{m}' = \mathbf{m}$.

A code $\mathcal{C}$ is a subspace of the overall vector space $\mathbb{F}_2^n$, with its vectors being the **codewords**. If $\mathcal{C}$ is a k-dimensional subspace of $\mathbb{F}_2^n$, it is referred to as an $[n, k]$ **linear code**. Such a code contains 2^k codewords, since the dimension k defines the number of independent basis vectors in $\mathcal{C}$.

As an example, consider a $[3, 1]$ repetition code:

- Codewords $\mathbf{c}$ are vectors of length $n = 3$ and there are $2^k = 2$ of them [0,0,0] and [1,1,1].
- There are $2^n = 8$ possible vectors in $\mathbb{F}_2^3$, of which the code $\mathcal{C}$ forms a subspace.

You usually define $\mathcal{C}$ using a generator matrix G. If $\mathcal{C}$ is an $[n, k]$ code, then its generator matrix is a $k \times n$ matrix whose rows form a basis for $\mathcal{C}$. To give $\mathcal{C}$ a metric, we define the distance between two codewords to be:

$$d(x, y) = \text{Hamming weight of } (x - y),$$

where x and y are codewords in the code $\mathcal{C}$. The **Hamming weight** of a codeword is the number of non-zero elements in the codeword. Finally, the **minimum distance**, d, of a code $\mathcal{C}$ is the smallest distance between any two distinct codewords. This also happens to be the minimum weight of any non-zero codeword in $\mathcal{C}$.

All of this leads us to the traditional way of referencing a code as $[n, k, d]$, where n is the length of the codewords, k is the dimension of the code space, and d is the minimum Hamming distance of $\mathcal{C}$.[HP10]

2 Quantum Error Correction (QEC)

Many parts of classical error correction are carried over to QEC, most notably the $[n, k, d]$ notation, but double brackets are used to distinguish a QEC code from a classical error correcting code.

$$[[n, k, d]]$$

In QEC these parameters are defined as:

- n = number of physical qubits used in QECC
- k = number of logical qubits represented
- d = the code distance between quantum codewords

The code distance between quantum codewords is the minimum number of errors needed for one codeword to be turned into another. A QECC can detect d - 1 errors and correct $\lfloor \frac{d-1}{2} \rfloor$ errors.

The 5-qubit QECC is a [[5, 1, 3]] code meaning five physical qubits are used to encode one logical qubit and it can detect up to two errors. It can correct one error. I will refer to the 5-qubit QECC as the perfect code for the rest of this paper.

3 Q#

Q# is a domain specific language for quantum computing. It is not like Qiskit or Cirq that are based on python and import python libraries to do the quantum computation. It is a programming language in its own right. It has its pros and cons. It was developed by Microsoft. It comes along with libraries and developer's tools called the Quantum Development Kit (QDK).

A "Hello World" example is below.

```
operation Main() : Unit {
    Message($"Hello World");
}
```

This highlights that almost all containers are "operations". The "Unit" return value is essentially the "Null" type in other languages. It has built-in operations for common gates such as the X, Y, and Z gates. It also has a built-in Qubit data type as well.

This example from the documentation goes a little further. The "//" characters are used to denote comments. [Mic24]

```
    operation GenerateRandomBit() : Result {
    // Allocate a qubit.
    use q = Qubit();

    // Set the qubit into superposition of 0 and 1 using the Hadamard
    H(q);

    // At this point the qubit ‘q‘ has 50% chance of being measured in the
    // |0⟩ state and 50% chance of being measured in the |1⟩ state.
    // Measure the qubit value using the ‘M‘ operation, and store the
    // measurement value in the ‘result‘ variable.
```

```
    let result = M(q);

    // Reset qubit to the |0⟩ state.
    // Qubits must be in the |0⟩ state by the time they are released.
    Reset(q);

    // Return the result of the measurement.
    return result;
}
```

4 Perfect Code in Q#

I will now go into my implementation of the perfect code in Q#. The complete code listing is in the Appendix. I relied on [MP24] as a reference for coding this program.

The first thing the program does is import all the relevant libraries in the header section. Then everything is encapsulated in the Main operation.

4.1 Allocate Qubit Register

The first line allocates a register of five qubits to be used. It is named "logicalQubit".

```
use logicalQubit = Qubit[5];
```

4.2 Encode

Then the program calls an operation to Encode these five qubits into a logical qubit. The qubit in the zeroth index would be the original physical qubit.

```
operation Encode(qs : Qubit[]) : Unit is Adj {
    H(qs[0]); // Hadamard on qubit 0.
    S(qs[0]); // S gate on qubit 0.
    CY(qs[0], qs[4]); // Controlled-Y gate between qubits 0 and 4.
    H(qs[1]); // Hadamard on qubit 1.
    CNOT(qs[1], qs[4]); // Controlled-NOT between qubits 1 and 4.
    H(qs[2]);
    CZ(qs[2], qs[0]);
    CZ(qs[2], qs[1]);
    CNOT(qs[2], qs[4]);
    H(qs[3]);
    S(qs[3]);
```

```
    CZ(qs[3], qs[0]);
    CZ(qs[3], qs[2]);
    CY(qs[3], qs[4]);
}
```

4.3 Apply Error

A random error is then applied:

```
// Introduces a random error (bit flip, phase flip, or Y error) on a randomly selected qubit.
operation ApplyError(qs: Qubit[]) : Unit {
    let index = DrawRandomInt(0,4); // Randomly selects one of the 5 qubits.
    let error = DrawRandomInt(0,2); // Randomly selects the type of error: X (0), Z (1), or Y

    if (error == 0) {
        Message($"Applying Bit Flip Error on Qubit {index}");
        X(qs[index]); // Applies an X (bit-flip) error to the selected qubit.
    }

    elif (error == 1) {
        Message($"Applying Phase Flip Error on Qubit {index}");
        Z(qs[index]); // Applies a Z (phase-flip) error to the selected qubit.
    }

    else {
        Message($"Applying Y Error on Qubit {index}");
        Y(qs[index]); // Applies a Y error (bit + phase flip) to the selected qubit.
    }
}
```

4.4 Detect Error

The errors are then detected using the stabilizer generators. "Measure()" does a join measurement and "PauliX", "PauliY", and "PauliZ" say in what basis. A decimal value is returned to be used in a lookup table on how to make a correction.

```
operation Detect(qs: Qubit[]) : Int {
    let syndrome1 = Measure([PauliX,PauliZ, PauliZ, PauliX, PauliI], qs);
    let syndrome2 = Measure([PauliI,PauliX, PauliZ, PauliZ, PauliX], qs);
    let syndrome3 = Measure([PauliX,PauliI, PauliX, PauliZ, PauliZ], qs);
    let syndrome4 = Measure([PauliZ,PauliX, PauliI, PauliX, PauliZ], qs);

    Message($"Value of Syndromes in order {syndrome1}, {syndrome2}, {syndrome3}, {syndrome4}"
```

```
    let decimalValue = ResultArrayAsInt([syndrome4, syndrome3, syndrome2, syndrome1]);

    Message($"Decimal Value of Syndrome Measurement is {decimalValue}");

    return decimalValue;
}
```

4.5 Correct Error

Taking the decimal value from above, the "Correct" operation looks up the appropriate corrective action in a lookup table and applies it.

```
 operation Correct(i : Int, qs : Qubit[]) : Unit {
    if (i == 1){
        Message("Bit Flip Error on Qubit 0 detected and corrected.");
        X(qs[0]);
    }
    elif (i == 10){
        Message("Z Error on Qubit 0 detected and corrected.");
        Z(qs[0]);
    }
    elif (i == 11){
        Message("Y Error on Qubit 0 detected and corrected.");
        Y(qs[0]);
    }
    elif (i == 8){
        Message("Bit Flip Error on Qubit 1 detected and corrected.");
        X(qs[1]);
    }
    elif (i == 5){
        Message("Z Error on Qubit 1 detected and corrected.");
        Z(qs[1]);
    }
    elif (i == 13){
        Message("Y Error on Qubit 1 detected and corrected.");
        Y(qs[1]);
    }
    elif (i == 12){
        Message("Bit Flip Error on Qubit 2 detected and corrected.");
        X(qs[2]);
    }
    elif (i == 2){
        Message("Z Error on Qubit 2 detected and corrected.");
        Z(qs[2]);
```

```
    }
    elif (i == 14){
        Message("Y Error on Qubit 2 detected and corrected.");
        Y(qs[2]);
    }
    elif (i == 6){
        Message("Bit Flip Error on Qubit 3 detected and corrected.");
        X(qs[3]);
    }
    elif (i == 9){
        Message("Z Error on Qubit 3 detected and corrected.");
        Z(qs[3]);
    }
    elif (i == 15){
        Message("Y Error on Qubit 3 detected and corrected.");
        Y(qs[3]);
    }
    elif (i == 3){
        Message("Bit Flip Error on Qubit 4 detected and corrected.");
        X(qs[4]);
    }
    elif (i == 4){
        Message("Z Error on Qubit 4 detected and corrected.");
        Z(qs[4]);
    }
    elif (i == 7){
        Message("Y Error on Qubit 4 detected and corrected.");
        Y(qs[4]);
    }
    else{
        Message("Index out of range (1-15)");
    }
}
```

4.6 Logical Operations

Two logical operations are implemented, the X and Z operations.

```
// A logical X operation that applies an X gate to all qubits in the array.
operation LogicalX(qs: Qubit[]) : Unit {
    ApplyToEach(X, qs);
}

// A logical Z operation that applies a Z gate to all qubits in the array.
operation LogicalZ(qs: Qubit[]) : Unit {
```

```
    ApplyToEach(Z, qs);
}
```

5 Conclusion

This has been a great project. I learned a tremendous amount about QEC and specifically the perfect code. It really helped to internalize how QEC works.

6 Appendix - Complete Q# Program

```
// Q# program implementing a quantum error correction (QEC) code for 5 qubits.
// The program encodes a logical qubit, introduces random errors, detects errors using syndrc
// and corrects the detected errors before decoding the logical qubit back to its original st

@Config(Unrestricted)
import Std.Diagnostics.CheckAllZero; // Checks if all qubits are in the |0⟩ state.
import Std.Random.DrawRandomInt; // Generates a random integer within a specified range.
import Std.Convert.ResultArrayAsInt; // Converts a Result array (syndrome measurements) into
import Std.Diagnostics.DumpMachine; // Dumps the quantum state for debugging.

// Used "Quantum Circuits for Stablizer Error Correcting Codes: A Tutorial" by Mondal and Par
// as reference.
@Config(Unrestricted)
operation Main() : Unit {
    use logicalQubit = Qubit[5]; // Allocates 5 physical qubits for the logical qubit.

    Encode(logicalQubit); // Encodes the logical qubit using the 5-qubit code.

    ApplyError(logicalQubit); // Introduces a random error on one of the qubits.

    let index = Detect(logicalQubit); // Detects the error using syndrome measurements.
    if (index != 0) { // If an error is detected:
        Message("Error Detected");
        Correct(index, logicalQubit); // Corrects the detected error.
    } else {
        Message("No Error Detected");
    }

    Decode(logicalQubit); // Decodes the logical qubit back to its original state.

    ResetAll(logicalQubit); // Resets all qubits to the |0⟩ state.
}

// Implements the encoding circuit for the 5-qubit code, following the reference paper.
// From Fig. 10, Page 9
operation Encode(qs : Qubit[]) : Unit is Adj {
    H(qs[0]); // Hadamard on qubit 0.
    S(qs[0]); // S gate on qubit 0.
    CY(qs[0], qs[4]); // Controlled-Y gate between qubits 0 and 4.
    H(qs[1]); // Hadamard on qubit 1.
    CNOT(qs[1], qs[4]); // Controlled-NOT between qubits 1 and 4.
    H(qs[2]);
    CZ(qs[2], qs[0]);
    CZ(qs[2], qs[1]);
```

```
    CNOT(qs[2], qs[4]);
    H(qs[3]);
    S(qs[3]);
    CZ(qs[3], qs[0]);
    CZ(qs[3], qs[2]);
    CY(qs[3], qs[4]);
}

// Decodes the logical qubit by applying the adjoint of the encoding circuit.
operation Decode(qs: Qubit[]) : Unit {
    Adjoint Encode(qs);
}

// Measures the syndrome values for error detection using the stabilizer generators.
operation Detect(qs: Qubit[]) : Int {
    let syndrome1 = Measure([PauliX,PauliZ, PauliZ, PauliX, PauliI], qs);
    let syndrome2 = Measure([PauliI,PauliX, PauliZ, PauliZ, PauliX], qs);
    let syndrome3 = Measure([PauliX,PauliI, PauliX, PauliZ, PauliZ], qs);
    let syndrome4 = Measure([PauliZ,PauliX, PauliI, PauliX, PauliZ], qs);

    Message($"Value of Syndromes in order {syndrome1}, {syndrome2}, {syndrome3}, {syndrome4}"
    let decimalValue = ResultArrayAsInt([syndrome4, syndrome3, syndrome2, syndrome1]);
    Message($"Decimal Value of Syndrome Measurement is {decimalValue}");
    return decimalValue;
}

// Corrects the error on the affected qubit based on the detected syndrome value.
// Maps syndrome values (integer) to error type and affected qubit.
// This uses a lookup table in the paper.
operation Correct(i : Int, qs : Qubit[]) : Unit {
    if (i == 1){
        Message("Bit Flip Error on Qubit 0 detected and corrected.");
        X(qs[0]);
    }
    elif (i == 10){
        Message("Z Error on Qubit 0 detected and corrected.");
        Z(qs[0]);
    }
    elif (i == 11){
        Message("Y Error on Qubit 0 detected and corrected.");
        Y(qs[0]);
    }
    elif (i == 8){
        Message("Bit Flip Error on Qubit 1 detected and corrected.");
        X(qs[1]);
    }
    elif (i == 5){
```

```
        Message("Z Error on Qubit 1 detected and corrected.");
        Z(qs[1]);
    }
    elif (i == 13){
        Message("Y Error on Qubit 1 detected and corrected.");
        Y(qs[1]);
    }
    elif (i == 12){
        Message("Bit Flip Error on Qubit 2 detected and corrected.");
        X(qs[2]);
    }
    elif (i == 2){
        Message("Z Error on Qubit 2 detected and corrected.");
        Z(qs[2]);
    }
    elif (i == 14){
        Message("Y Error on Qubit 2 detected and corrected.");
        Y(qs[2]);
    }
    elif (i == 6){
        Message("Bit Flip Error on Qubit 3 detected and corrected.");
        X(qs[3]);
    }
    elif (i == 9){
        Message("Z Error on Qubit 3 detected and corrected.");
        Z(qs[3]);
    }
    elif (i == 15){
        Message("Y Error on Qubit 3 detected and corrected.");
        Y(qs[3]);
    }
    elif (i == 3){
        Message("Bit Flip Error on Qubit 4 detected and corrected.");
        X(qs[4]);
    }
    elif (i == 4){
        Message("Z Error on Qubit 4 detected and corrected.");
        Z(qs[4]);
    }
    elif (i == 7){
        Message("Y Error on Qubit 4 detected and corrected.");
        Y(qs[4]);
    }
    else{
        Message("Index out of range (1-15)");
    }
}
```

```
// Introduces a random error (bit flip, phase flip, or Y error) on a randomly selected qubit.
operation ApplyError(qs: Qubit[]) : Unit {
    let index = DrawRandomInt(0,4); // Randomly selects one of the 5 qubits.
    let error = DrawRandomInt(0,2); // Randomly selects the type of error: X (0), Z (1), or Y
    if (error == 0) {
        Message($"Applying Bit Flip Error on Qubit {index}");
        X(qs[index]); // Applies an X (bit-flip) error to the selected qubit.
    } elif (error == 1) {
        Message($"Applying Phase Flip Error on Qubit {index}");
        Z(qs[index]); // Applies a Z (phase-flip) error to the selected qubit.
    } else {
        Message($"Applying Y Error on Qubit {index}");
        Y(qs[index]); // Applies a Y error (bit + phase flip) to the selected qubit.
    }
}

// A logical X operation that applies an X gate to all qubits in the array.
operation LogicalX(qs: Qubit[]) : Unit {
    ApplyToEach(X, qs);
}

// A logical Z operation that applies a Z gate to all qubits in the array.
operation LogicalZ(qs: Qubit[]) : Unit {
    ApplyToEach(Z, qs);
}
```

References

[Sha48] Claude E. Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[HP10] W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2010.

[Mic24] Microsoft. *Quantum Random Number Generator Tutorial*. Accessed: 2024-12-02. 2024. URL: https://learn.microsoft.com/en-us/azure/quantum/tutorial-qdk-quantum-random-number-generator?tabs=tabid-copilot.

[MP24] Arijit Mondal and Keshab K. Parhi. "Quantum Circuits for Stabilizer Error Correcting Codes: A Tutorial". In: *IEEE Circuits and Systems Magazine* 24.1 (2024), pp. 33–51. DOI: 10.1109/MCAS.2024.3349668.