A Comprehensive Overview of Convolutional Encoders and the Viterbi Algorithm

Malik Kurtz

December 16, 2024

Contents

1 Introduction

In modern digital communication systems, transmitting information reliably over noisy channels is a fundamental challenge. Noise, interference, and signal degradation can lead to errors that corrupt the transmitted data. To mitigate these issues, *error-correcting codes* are employed. These codes add structured redundancy to the data, enabling the receiver to detect and, in many cases, correct errors introduced during transmission.

Among a variety of coding techniques, *convolutional codes* are particularly noteworthy due to their continuous processing of data and their adaptability to various channel conditions. Instead of encoding data in fixed-length blocks, convolutional encoders treat the input stream as a continuous flow, using internal memory elements to produce a code sequence that depends not only on the current input bits but also on a portion of the past input.

To decode convolutionally encoded data, one of the most prominent algorithms is the *Viterbi algorithm*. This algorithm provides a maximum-likelihood decoding approach, meaning it selects the most likely transmitted sequence given the noisy received signal.

By effectively searching through a trellis of possible states and transitions, the Viterbi algorithm can reliably recover the original information bits with a high degree of accuracy.

This document will provide a deeply detailed exploration of convolutional encoders and the Viterbi algorithm, covering fundamental concepts, theoretical foundations, and practical insights, including:

- A thorough introduction to convolutional encoders, including their internal structure and key parameters.
- The concept of *free distance* and its importance in defining the error-correcting capability of a convolutional code.
- A step-by-step explanation of the Viterbi algorithm, including its trellis representation, path metrics, and traceback procedure.
- An illustrative C++ code example demonstrating how to implement a convolutional encoder, simulate a noisy channel, and decode using the Viterbi algorithm.

2 Convolutional Encoders

2.1 Definition and Intuition

A **convolutional encoder** is a type of forward error-correcting encoder that processes a continuous stream of input bits. Unlike block codes, where the input is partitioned into distinct blocks before encoding, convolutional codes incorporate memory. In other words, the output at any given time depends not only on the current input bit but also on some number of previous bits. This memory aspect creates a *convolution*—a sliding operation—of the input sequence with a set of generator polynomials.

Visually, a convolutional encoder can be represented by shift registers and XOR gates:

- The shift registers hold a certain number of past input bits, representing the encoder's memory.
- Each output bit is generated by XOR-ing certain taps (positions) in the register as specified by the generator polynomials.

Because the output at each time step depends on a combination of current and past inputs, small input changes can affect multiple subsequent output bits, distributing the information and redundancy throughout the transmitted sequence.

2.2 Key Parameters

Constraint Length (k): The *constraint length* determines the number of past input bits (plus the current bit) that the encoder memory considers. If the encoder's memory size is $k-1$ bits, then k is the total number of bits (current input plus memory) that determine the output at any instant.

For instance, if $k=3$, the encoder remembers two previous input bits and the current one, making the total span of influence three bits at a time.

Code Rate (R): The *rate* of a convolutional code is defined as:

$$R = \frac{k_{\text{in}}}{n_{\text{out}}},$$

where k_{in} is the number of input bits processed at each time step (often 1) and n_{out} is the number of output bits produced at each time step. For many practical convolutional encoders, $k_{\text{in}} = 1$, so the rate simplifies to $R = 1/n_{\text{out}}$. A lower rate (e.g., 1/3) means more redundancy is added, potentially improving error correction at the cost of higher bandwidth usage.

Generator Polynomials: The behavior of a convolutional encoder is fully determined by its **generator polynomials**. Each output bit is associated with a polynomial that dictates which taps from the memory (including the current input) are XOR-ed together. For example, if the constraint length is $k = 3$, and you have one output polynomial $G(D) = 1 + D + D^2$, it means the output bit is formed by XOR-ing the current input bit and the two previous bits in the register.

Multiple polynomials produce multiple outputs at each time step, creating a coded output sequence that combines information from the current and previous input bits in a structured manner.

2.3 Free Distance and Error-Correcting Capability

The traditional concept of *minimum distance* used in block codes is adapted in convolutional codes as the *free distance*, d_{free}. The free distance is the minimum Hamming distance between any two distinct infinite-length encoded sequences. Since convolutional codes operate on streams rather than blocks, this infinite-length consideration is essential.

The free distance sets a fundamental limit on the code's ability to distinguish transmitted sequences from one another after passing through a noisy channel. The larger the free distance, the more resilient the code is to errors. A high free distance implies that valid encoded sequences are more widely separated in Hamming space, making it easier for the decoder to identify the correct sequence despite noise-induced errors.

3 The Viterbi Algorithm

3.1 Overview and Motivation

The **Viterbi algorithm**, introduced by Andrew Viterbi in 1967, revolutionized the decoding of convolutional codes. It provides a maximum-likelihood decoding approach, ensuring that the decoded sequence is the most probable one given the received noisy sequence. The Viterbi algorithm is optimal, meaning it finds the path through the state trellis that yields the minimal cumulative distance to the received sequence.

3.2 The Trellis Diagram

To decode convolutional codes, we represent the code as a *trellis*, a time-indexed graph where:

- Each column of the trellis corresponds to a time step.

- Each node in the column represents a possible state of the encoder (the contents of its memory).
- Each arrow (branch) connecting states from one time step to the next represents a possible input bit that transitions the encoder from one state to another. Each branch also has an associated output (encoded bits).

Since the encoder has memory of length $k-1$, there are 2^{k-1} possible states at any given time. Each new input bit (0 or 1) typically splits from each current state into one or two possible next states.

3.3 How the Viterbi Algorithm Works in Detail

1. **Initialization:** At the start ($t = 0$), the encoder is assumed to be in the all-zero state. The Viterbi decoder initializes the cumulative path metric (often the sum of Hamming distances) for the zero state to 0 and sets the metrics for all other states to infinity. This enforces that initially, the only considered valid path is the one that starts from the all-zero state.

2. **Branch Metrics and Path Metrics:** For each received encoded symbol (which may be corrupted by noise), the decoder computes the *branch metric*. The branch metric is often the Hamming distance between the expected encoded output for a given state transition and the actual received bits at that time step.

After computing branch metrics, the algorithm updates the *path metrics* (cumulative metrics) for each state by adding the branch metric to the metric of the previous path that leads to this state. Since there might be multiple incoming paths to a state, the decoder selects the path with the smallest cumulative metric as the *survivor path*. This ensures that after each time step, only the most likely path leading into each state is retained.

3. **Survivor Paths and Traceback:** As the decoding progresses over many time steps, the Viterbi algorithm maintains a record (often stored in traceback memory) of which path survives at each state. After processing all received data, the decoder will pick the state with the lowest final cumulative metric. To recover the transmitted bits, it performs a traceback through the survivor paths, moving backward in time until it reaches the initial state. This traceback reveals the most likely sequence of input bits.

The Viterbi algorithm's computational complexity is linear in the length of the input sequence, but it grows exponentially with constraint length because of the number of states. In practice, k is chosen to be moderately small (often less than or equal to 10) to keep decoding manageable.

4 Example C++ Implementation

Below is an example C++ code outline that demonstrates a simplified end-to-end system:

1. **Input**: Reads a user-provided message as a string.
2. **Binary Conversion**: Converts this string into a binary vector of bits for processing.
3. **Encoding**: Applies a convolutional encoder (with a specified constraint length k and a set of generator polynomials) to produce a coded bitstream.

4. **Noise Simulation**: Introduces random bit errors into the encoded bitstream with probability p, simulating a noisy transmission channel.

5. **Viterbi Decoding**: Uses the Viterbi algorithm to decode the noisy received bitstream and attempts to recover the original input message.

6. **Performance Evaluation**: Computes performance metrics such as Bit Error Rate (BER) and success rate for different constraint lengths. These metrics help assess the effectiveness of the code under given channel conditions.

Note: The following code snippet is a shortened, illustrative example. In a real-world scenario, you should ensure:

- The C++ file (e.g., `ConvCode.cpp`) is located in the same directory as this LaTeX file or that you provide the correct path.

- Proper compiler installation (e.g., `g++`) and environment setup.

```
#include <iostream>
#include <stdio.h>
#include <vector>
#include <string>
#include <bitset>
#include <cstdlib>
#include <time.h>
#include <cmath>
#include <map>
#include <bitset>
#include <fstream>

// Alphabet: Binary i.e (A = {0,1}), length of alphabet q = 2

// Length: The length of the codeword will be equal to the number of
    generator polynomials corresponding to that particular k value (
    currently this is 5)

// Dimension: For convolutional encoders, the dimension generally
    refers to the number of input bits per unit time that the encoder
    processes
// rather than the number of independent vectors in a linear subspace (
    as in block codes). R = k / n, k = # input bits, n = # output bits

// Distance: In convolutional codes, a related metric called the free
    distance   free   is often used instead of the traditional minimum
    distance.
// it is the minimum Hamming weight of the difference between any two
    output sequences generated by distinct input sequences
// A convolutional code has the potential to correct floor(dfree - 1 /
    2) errors per code sequence


using namespace std;

map<int, vector<unsigned int>> generatorPolynomialsMap = {
    {4, {0x5, 0x5, 0x5, 0x5, 0x5}},
    {5, {0x9, 0x9, 0x9, 0x9, 0x9}},
    {6, {0x15, 0x15, 0x15, 0x15, 0x15}},
```

```
    {7, {0x23, 0x23, 0x23, 0x23, 0x23}},
    {8, {0x72, 0x72, 0x72, 0x72, 0x72}},
    {9, {0x9b, 0x9b, 0x9b, 0x9b, 0x9b}},
    {10, {0x13c, 0x13c, 0x13c, 0x13c, 0x13c}},
    {11, {0x29b, 0x29b, 0x29b, 0x29b, 0x29b}},
    {12, {0x4f5, 0x4f5, 0x4f5, 0x4f5, 0x4f5}},
    {13, {0xa4f, 0xa4f, 0xa4f, 0xa4f, 0xa4f}},
    {14, {0x10b7, 0x10b7, 0x10b7, 0x10b7, 0x10b7}},
    {15, {0x2371, 0x2371, 0x2371, 0x2371, 0x2371}},
    {16, {0x5a47, 0x5a47, 0x5a47, 0x5a47, 0x5a47}}
};

struct vNode {
    long long cumHammingDistance;
    bool inputArrivalBit;
    int state;

    bool operator<(const vNode& other) const {
        return state < other.state;
    }
};

int calculateHammingDistance(vector<bool>& code1, vector<bool>& code2); // DONE
vector<bool> generateOutput(vector<bool>& shiftregister, vector<unsigned int>& genPolynomials); // DONE
vector<bool> addNoise(vector<bool>& code, float prob_of_error); // DONE
vector<bool> getOriginalCode(const vector<vector<vNode>> &trellis); // DONE
vector<vector<bool>> generateStates(int k); // DONE
vector<bool> calculatePotentialInput(vector<bool>& curState, bool next_input); // DONE
vector<bool> stringToVecBool(string& message); // DONE
string vecBoolToString(vector<bool>& binary); // DONE
void exportData(map<int, vector<float>>& k_data_points, string& filename); // DONE
void printTrellisStates(const vector<vector<vNode>> &trellis); // DONE
string vecBoolToStringBinary(vector<bool>& binary);
int vecBoolToInt(vector<bool>& bits);


int timeSteps = 0;
vector<vector<vNode>> trellis;

// // Encode Method
vector<bool> encode(vector<bool> code, int k, vector<unsigned int> genPolynomials) { // k is the constraint length i.e length of the shift register we want to use
    vector<bool> encodedVector = {};
    vector<bool> sliding_window = {};
    vector<bool> outputVector = {};
    // Follow this same procedure till every bit is processed
    for (int i = 0; i < code.size(); i++) {
```

```
        sliding_window.clear();
        for (int m = 0; m < k-i-1; m++) {
            sliding_window.push_back(0);
        }
        for (int j = 0; j < k - (max((k-i-1), 0));j++) {
            if (i >= k) {
                sliding_window.push_back(code[j-(k-i-1)]);
            }
            else {
                sliding_window.push_back(code[j]);
            }

        }
        vector<bool> outputVector = generateOutput(sliding_window,
            genPolynomials);
        encodedVector.insert(encodedVector.end(), outputVector.begin(),
             outputVector.end());

        timeSteps += 1;
    }
    return encodedVector;
}


vector<bool> viterbiDecode(vector<bool> noisy_encoded_code, int k,
    vector<vector<bool>> states, vector<unsigned int> genPolynomials) {

    vector<bool> decoded = {};

    int outputBits = genPolynomials.size();
    trellis.resize(timeSteps + 1);

    //initialize the only node at t = 0 (0,0)

    for (int t = 0; t < timeSteps+1; t++) {
        for (int s = 0; s < states.size(); s++) {
            vNode defaultNode;

            defaultNode.state = s;
            defaultNode.inputArrivalBit = 0;
            defaultNode.cumHammingDistance = INT_MAX;
            trellis[t].push_back(defaultNode);

        }
    }

    trellis[0][0].cumHammingDistance = 0; // assuming out register will
         start at an intial state of State 0

    //printTrellisStates(trellis);



    for (int t = 1; t <= timeSteps; t++) {
        vector<bool> observedInput = vector<bool>(noisy_encoded_code.
            begin() + (outputBits*(t-1)), noisy_encoded_code.begin() + (
            outputBits*(t-1) + outputBits));
        for (int s = 0; s < states.size(); s++) {
```

```
            // cout << "---------------------------------------------------------------------------------" << endl;
            // cout << "State is: " << s << endl;
            bool transitionBit = s % 2;
            trellis[t][s].inputArrivalBit = transitionBit;

            int currentState = vecBoolToInt(states[s]);


            int firstPrevState = (currentState >> 1);
            if (trellis[t-1][firstPrevState].cumHammingDistance != INT_MAX) {
                // cout << "Previous state for first input is: " << to_string(firstPrevState) << endl;
                vector<bool> firstPotentialInput = calculatePotentialInput(states[firstPrevState], transitionBit);
                vector<bool> firstExpected = generateOutput(firstPotentialInput, genPolynomials);
                // cout << "Our expected is:    " << firstExpected << endl;
                // cout << "Our observation is: " << observedInput << endl;
                int firstHammingDistance = trellis[t-1][firstPrevState].cumHammingDistance + calculateHammingDistance(observedInput, firstExpected);
                // cout << "So the cumulative hamming distance would be: " << firstHammingDistance << endl;
                if (firstHammingDistance < trellis[t][s].cumHammingDistance) {
                    trellis[t][s].cumHammingDistance = firstHammingDistance;
                    trellis[t][s].inputArrivalBit = transitionBit;
                }
            }


            int secondPrevState = ((currentState) >> 1) | (1 << (k-2));
            if (trellis[t-1][secondPrevState].cumHammingDistance != INT_MAX) {
                // cout << "Previous state for second input is: " << to_string(secondPrevState) << endl;
                vector<bool> secondPotentialInput = calculatePotentialInput(states[secondPrevState], transitionBit);
                vector<bool> secondExpected = generateOutput(secondPotentialInput, genPolynomials);
                // cout << "Our expected is:    " << secondExpected << endl;
                // cout << "Our observation is: " << observedInput << endl;
                int secondHammingDistance = trellis[t-1][secondPrevState].cumHammingDistance + calculateHammingDistance(observedInput, secondExpected);
                // cout << "So the cumulative hamming distance would be: " << secondHammingDistance << endl;
```

```
                if (secondHammingDistance < trellis[t][s].cumHammingDistance) {
                    trellis[t][s].cumHammingDistance = secondHammingDistance;
                    trellis[t][s].inputArrivalBit = transitionBit;
                }
            }
        // cout << "------------------------------------------------------------------------" << endl;
        }
    }

    // printTrellisStates(trellis);

    return getOriginalCode(trellis);
}



int main() {
    float p;
    int lowerKlimit;
    int upperKlimit;
    string exportFile = "results.csv";

    // degree of any gen polynomial should always be less than or equal to k-1
    vector<vector<bool>> possibleStates;

    // used to get the random number between 0 and 1 when determining when to flip bits
    // unsigned int seed = 12345;  // Replace 12345 with any specific seed you want
    // srand(seed);
    srand( (unsigned)time( NULL ) );

    // the code that we want to encode
    string message;
    getline(cin, message);
    vector<bool> code = stringToVecBool(message);

    // string code = "1010";

    // string code = "1010";

    // the probability of ax single bit flipping after encoding the original code
    p = 0.1; // LOL
    p = 0.01; // Poor channel conditions, severe interference, or far-from-optimal signal quality.
    // p = 0.001; // Moderate noise, common in low-quality wireless connections or basic wired links with interference.
    p = 0.05;
    int numIterations = 1000;


    // if (code.length() < 50) {
```

```
    //      lowerKlimit = 4;
    //      upperKlimit = 5;
    // }
    // else if (code.length() < 100) {
    //      lowerKlimit = 5;
    //      upperKlimit = 7;
    // }
    // else {
    //      lowerKlimit = 7;
    //      upperKlimit = 8; // going above 8 will destroy your computer :)
    //}


    lowerKlimit = 4;
    upperKlimit = 10;

    map<int, vector<float>> k_averages;  // Adjusted to store averages for all possible_k values

    // Loop over possible values of k
    for (int possible_k = lowerKlimit; possible_k <= upperKlimit; possible_k++) {

        float average_ber = 0.0;
        float average_success_rate = 0.0;

        float ber;

        possibleStates = generateStates(possible_k);

        cout << "===============================================================" << endl;
        cout << "Processing for k = " << possible_k << " (" << numIterations << " iterations)" << endl;
        cout << "===============================================================" << endl;

        // Perform multiple iterations for each k value
        for (int i = 0; i < numIterations; i++) {
            // Reset the environment for each iteration
            timeSteps = 0;
            trellis.clear();

            // Encoding, adding noise, and decoding
            vector<bool> encoded = encode(code, possible_k, generatorPolynomialsMap[possible_k]);
            vector<bool> noisy_encoded = addNoise(encoded, p);
            vector<bool> originalCode = viterbiDecode(noisy_encoded, possible_k, possibleStates, generatorPolynomialsMap[possible_k]);
            string originalMessage = vecBoolToString(originalCode);
            // string originalMessage = originalCode;

            // Displaying results for each iteration
            cout << "
```

```cpp
            ----------------------------------------------------------------------" << endl;
            cout << "Iteration #" << i + 1 << ":" << endl;
            cout << "K = " << possible_k << endl;

            // print a newline after all polynomials have been printed
            cout << endl;
            cout << "Original Message : " << message << endl;
            cout << "Original Code   : " << vecBoolToStringBinary(code) << endl;
            cout << "Encoded Code    : " << vecBoolToStringBinary(encoded) << endl;
            cout << "Noisy Code      : " << vecBoolToStringBinary(noisy_encoded) << endl;
            cout << "Noise Added?    : " << (noisy_encoded == encoded ? "No" : "Yes") << endl;
            cout << "# bits flipped: " << calculateHammingDistance(encoded,noisy_encoded) << endl;
            ber = ((float) calculateHammingDistance(code, originalCode) / (float) code.size());
            cout << "Bit Error Rate : " << ber * 100 << "%" << endl;
            average_ber += ber;
            cout << "Decoded Code    : " << vecBoolToStringBinary(originalCode) << endl;
            cout << "Decoded Message: " << originalMessage << endl;

            bool success = (code == originalCode);
            if (success) {
                cout << "Result          : SUCCESS" << endl;
                average_success_rate += 1;
            } else {
                cout << "Result          : FAIL" << endl;
            }
            cout << "----------------------------------------------------------------------" << endl;
        }

        // Calculate and display the success rate for this k value
        average_ber /= numIterations;
        average_success_rate /= numIterations;
        k_averages[possible_k].push_back(average_ber);
        k_averages[possible_k].push_back(average_success_rate);

        cout << "======================================================================" << endl;
        cout << "Summary for k = " << possible_k << ":" << endl;
        cout << "Average Success Rate: " << average_success_rate * 100 << "% Success Rate" << endl;
        cout << "Average BER: " << average_ber * 100 << "% Bit Error Rate" << endl;
        cout << "======================================================================" << endl << endl;
    }

    // Final summary of all k values
```

```
    cout << "==================== Overall Results =====================" << endl;
    cout << "Noise was: " << p << endl;
    cout << "Original message was: " << message << endl;
    cout << "Message length was: " << code.size() << endl;
    // cout << "Output Bits per input: " << outputBits << endl;
    for (int i = lowerKlimit; i <= upperKlimit; i++) {
        cout << "For k = " << i <<
        " -> Average BER = " << std::fixed << std::setprecision(2) << k_averages[i][0] * 100 << "%" <<
        " -> Average Success Rate = " << k_averages[i][1] * 100 << "%" << endl;
    }
    cout << "============================================================" << endl;

    exportData(k_averages, exportFile);
    return 0;
}



int calculateHammingDistance(vector<bool>& code1, vector<bool>& code2) {
    if (code1.size() != code2.size()) {
        return -1;
    }

    int hammingDistance = 0;
    for (int i = 0; i < code1.size(); i++) {
        if (code1[i] != code2[i]) {
            hammingDistance++;
        }
    }

    return hammingDistance;
}

vector<bool> generateOutput(vector<bool>& shiftregister, vector<unsigned int>& genPolynomials) {
    vector<bool> toReturn = {};
    int k = shiftregister.size();

    bool registerParity=0;

    for (unsigned int genPoly : genPolynomials) {
        genPoly = genPoly << 1 | 1; // just need to add another 1 at the end due to implicit +1 notation
        registerParity = 0;
        // cout << "Current genPoly: " << bitset<8>(genPoly) << endl; // Print binary of genPoly for clarity
        // cout << "Shift Register: " << shiftregister << endl;
        // gen poly = 1011
        // k = 4
        //j = 0
        for (int j = 0; j < k; j++) {
            if (((genPoly >> j) & 1) == 1) {
```

```
                // cout << "  - XOR with shiftregister[" << k - 1 - j
                    << "] (" << shiftregister[k - 1 - j] << ")" << endl;
                registerParity ^= shiftregister[k - 1 - j];
            }
            else {}
        }
        // cout << "Intermediate registerParity: " << registerParity <<
            endl;
        toReturn.push_back(registerParity);
    }
// cout << "Final Output is: " << parityBits << endl;
    return toReturn;
}

vector<bool> addNoise(vector<bool>& code, float prob_of_error) {

    vector<bool> noisyEncoded = {};

    // For every  bit in the  code
    for (int i = 0; i < code.size(); i++) {
        // turn the char back to an int
        // calculate a random number between 0 and 1 and if its less
            than the p value passed in, flip the bit
        if ((float) rand()/RAND_MAX < prob_of_error) {
            switch (code[i]) {
                case 0:
                    noisyEncoded.push_back(1);
                    break;
                case 1:
                    //cout << "Flipping 1 to 0" << endl;
                    noisyEncoded.push_back(1);
                    //cout << std::bitset<64>(raw_code_as_vector[i]).
                        to_string() << endl;
                    break;
            }
        }
        // if the random number generated isn't less than p, dont flip
            the bit
        else {
            noisyEncoded.push_back(code[i]);
        }

    }
    return noisyEncoded;
}

void printTrellisStates(const vector<vector<vNode>> &trellis) {
    for (int i = 0; i < trellis.size(); i++) {
        cout << "For t = " << i << endl;
        for (int j = 0; j < trellis[i].size(); j++) {
            cout << "Node State: " << trellis[i][j].state
                << " Cumulative Hamming Distance: " << trellis[i][j].
                    cumHammingDistance << endl;
        }
    }
}
```

```
vector<bool> recursiveBackTrack(const vector<vector<vNode>>& trellis, int t, int state) {
    if (t == 0) {
        return {};
    }

    int bestPrevState = 0;
    int minDistance = INT_MAX;
    bool inputArrivalBit = 0;

    for (int prevState = 0; prevState < trellis[t-1].size(); prevState++) {
        if (trellis[t-1][prevState].cumHammingDistance < minDistance) {
            minDistance = trellis[t-1][prevState].cumHammingDistance;
            bestPrevState = prevState;
            inputArrivalBit = trellis[t][state].inputArrivalBit;
        }
    }

    vector<bool> result = recursiveBackTrack(trellis, t - 1, bestPrevState);

    result.push_back(inputArrivalBit);

    return result;
}


vector<bool> getOriginalCode(const vector<vector<vNode>> &trellis) {
    int finalState = 0;
    int minDistance = INT_MAX;


    for (int state = 0; state < trellis.back().size(); state++) {
        if (trellis.back()[state].cumHammingDistance < minDistance) {
            minDistance = trellis.back()[state].cumHammingDistance;
            finalState = state;
        }
    }

    return recursiveBackTrack(trellis, trellis.size()-1, finalState);
}



vector<vector<bool>> generateStates(int k) {
    vector<vector<bool>> statesVector = {};

    vector<bool> curState;
    //Ex. k = 4
    // this loop will run 8 times to generate all 8 possible states
    // i goes from 0-7
    // for i = 0
    for (int i = 0; i < pow(2, k-1); i++) {
        curState = {};
        // bit = 4 - 2 = 2
        // so bit goes from 2-0
        for (int bit = k - 2; bit >= 0; bit--) {
```

```
            // i = 0
            // bit = 2
            // 0 >> 2 = 0, & 1 = 0, 0 gets appended
            // bit = 1
            // 0 >> 1 = 0, & 1 = 0, 0 gets appended
            // bit = 0
            // 0 >> 0 = 0, & 1 = 0, 0 gets appended
            curState.push_back(((i >> bit) & 1));
        }
        statesVector.push_back(curState);
    }

    return statesVector;
}

vector<bool> calculatePotentialInput(vector<bool>& curState, bool
    next_input) {
    vector<bool> input = curState;
    input.push_back(next_input);
    return input;
}


vector<bool> stringToVecBool(string& message) {

    vector<bool> toReturn = {};

    bitset<8> curCharacter;

    // for every character in the string
    for (char c : message) {
        curCharacter = c;
        // for every bit in the character
        for (int i = 7; i >= 0; i--) {
            toReturn.push_back(curCharacter[i]);
        }
    }

    return toReturn;
}


string vecBoolToString(vector<bool>& binary) {
    string toReturn = "";

    char c = 0;

    for (int i = 0; i < binary.size() ; i+=8) {
        for (int j = 0; j < 8 ; j++) {
            c = c << 1 | binary[i+j];
        }
        toReturn += c;
    }

    return toReturn;
}

string vecBoolToStringBinary(vector<bool>& binary) {
```

```
    string stringBinary = "";

    for (int i = 0; i < binary.size(); i++) {
        stringBinary += to_string(binary[i]);
    }

    return stringBinary;
}

void exportData(map<int, vector<float>>& k_data_points, string&
    filename) {
    ofstream file(filename);
    if (file.is_open()) {
        file << "K, Success Rate, BER\n";
        for (auto& entry : k_data_points) {
            file << entry.first << "," << entry.second[1] * 100 << ","
                << entry.second[0] * 100 << "\n";
        }
        file.close();
    }

}

int vecBoolToInt(vector<bool>& bits) {
    int value = 0;
    for (bool bit : bits) {
        value = (value << 1) | bit;
    }
    return value;
}
```

Listing 1: C++ Code for Convolutional Encoding and Viterbi Decoding

5 How to Compile and Run the Code

Assuming a standard Unix-like environment with a C++ compiler:

```
g++ -o conv_decoder ConvCode.cpp -O2 -std=c++11
./conv_decoder
```

When running the executable:

1. You will be prompted to input a message (a string of characters).
2. The program encodes the message using the convolutional encoder.
3. It then simulates channel noise by flipping bits with probability p.
4. The Viterbi decoder attempts to recover the original message from the noisy encoded data.
5. Finally, the program prints out the Bit Error Rate (BER) and success rate for various constraint lengths k, showing how code complexity and memory depth affect error-correction performance.

6 Conclusion

Convolutional codes offer a powerful mechanism to combat channel noise by spreading input data across multiple output bits, thereby creating redundancy that can be exploited at the receiver. The Viterbi algorithm provides an optimal solution for decoding these codes, ensuring minimal decoding errors by selecting the most probable transmitted sequence based on a trellis representation of the encoding process.

In practical communication systems, convolutional codes and Viterbi decoding are widely used due to their balance of performance, complexity, and robustness. Applications range from deep-space communications (e.g., NASA missions) and satellite links to cellular networks and wireless LANs. Although newer code families like Turbo codes and LDPC codes have gained popularity, the foundational understanding of convolutional codes and the Viterbi algorithm remains an essential part of modern digital communications theory and practice.

References

1. J. G. Proakis, *Digital Communications*, 4th ed., McGraw-Hill, 2001.

2. S. Lin and D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, 2nd ed., Prentice Hall, 2004.

3. A. J. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

4. J. Hagenauer, “The Viterbi Algorithm,” *Proceedings of the IEEE*, vol. 69, no. 11, pp. 1397–1449, 1981.

5. G. David Forney Jr., “The Viterbi Algorithm,” *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.