Introduction to the Error Correction Coding Laboratories

Todd K. Moon

Introduction

This document is intended to provide some background for the rationale and technical aspects of the programming laboratories.

One of my cardinal goals as an educator is to ensure that my students get theoretical knowledge mixed with "practical" skills. An employment reality is that many students that want to work in areas impacted by error control coding (such as the wireless industry) after graduation will not be solving equations over finite fields, much as they might enjoy it. Many will be working on implementations in hardware of software. These labs will help bridge the gap between theory and practice, potentially leading toward realizable designs, and interesting employment.

Even for students who don't end up dealing with error control coding, the vast majority of practicing engineers will need programming skills, and C++ makes a valuable line item in a resume. The labs presented here make use of many of the concepts of C++, such as classes, templates and overloaded operators, and are thus provide a solid basis for C++ programming for engineering problems.

Note: The code presented as example code here represent C++ code that has been compiled and run using the `g++` compiler under a Linux system. **No guarantee is provided** that the code operates 100% correctly. It is, after all, the responsibility of the student to ensure that the program runs correctly. However, the commands provided do suffice to compile and link an operating program under the system described. **It is the student's individual responsibility to determine how to run appropriate editor, compiler, and linker facilities on the computer they choose to run.** The instructor (or author) cannot and will not be able to provide specific guidance on other systems.

Why C++?

It was while programming a BCH decoder in Fortran that all the pieces of my first coding course came together. In that programming effort, I had to build up the set of functions to do the Galois field operations, then build the decoder on top of it. The Berlekamp algorithm which was stated so concisely on paper ended up having a clumsy Fortran representation completely lacking in elegance (although it worked properly). Swallowing the entire decoder at once in a single programming assignment — including the Galois field arithmetic, the polynomial arithmetic, and the decoder — made for time-consuming debugging. Shortly after my first BCH decoder implementation, I become introduced to C++, and reworked the decoder in C++. The experience was more interesting and, in many ways, more natural, than in Fortran.

One of the particular features of C++ which I have used in these labs used is actually one of the more deprecated features of C++ — overloaded operators. C++ allows the standard operators of the language to be redefined so that they operate in a context-dependent manner. When used to excess or in poor designs, overloaded operators can lead to confusing code. For example, if you overload the $*$ operator to act like the $-$ operator, understanding the statement `c = a*b;` could be confusing. But when used in a natural way, overloaded operators provide a powerful representation in the code of the actual mathematical operations. By defining a Galois field type with its set of operators, mathematical statements are easily transformed into the programming language. For example, the statement

$$b = c + \alpha^n$$

could be represented in C++ as

```
b = c + (A^n);
```

The implementation nearly follows the mathematics. Another feature of C++ that fits these problems is templates, in which functions are defined to accept different kinds of operands. For example, in some problems you want to represent polynomials with real coefficients, in other problems you want to represent polynomials with coefficients from $GF(2)$, and in others you want polynomials with coefficients from $GF(2^8)$. Mathematically, that is, we want program representations for elements of $\mathbb{R}[x]$, $GF(2)[x]$, and $GF(2^8)[x]$. With the library functions developed here, the C++ syntax for constructing such polynomials could be written as

```
polynomialT<double> p1;
polynomialT<GF2> p2;
polynomialT<GFnum2m> p3;
```

Another benefit of C++ in real systems is the basic idea of objects which contain all the data associated with that object. Take, for example, a large system which contains two decoder objects. Each decoder needs its internal storage. Without the notion of objects, passing this data to functions which operate on that data would be the responsibility of the calling program. The object design idea encapsulates the data along with the functions that use it.

There are other environments that can provide many of the operations which are built up here, such as MATLAB or MATHEMATICA. Why, then, use C++? A primary motivation has been computation speed: Since the programs are compiled directly into machine code, the execution speed will be faster than for an interpreted environment, such as MATLAB. This speed becomes important when a very large number of computations are necessary (such as bit error rate curves). Furthermore, using another implementation (such as MATLAB) hides important details that I think students should be aware of.

How much C++ is prerequisite?

Familiarity with general principles of programming, and C language syntax, is assumed. Also, general familiarity with general concepts of C++ is assumed, including the idea of an object, member data, member functions, constructors, destructors, overloaded functions, and templates. Provided that there is general familiarity, the examples provided should suffice to clarify how the language is used.

Why these labs?

The labs selected are designed to build understanding by a step-by-step sequence of labs, to develop background in concepts to ECC (such as maximal-length sequences), and to build useful tools (such as CRC encoder/decoder). There are essentially three different tracks. The first track builds up tools and understanding for block codes, culminating in a general Reed-Solomon encoder/decoder. The second track deals with trellis codes, consisting of convolutional codes and trellis-coded modulation. The third track deals with two modern codes, LDPC codes and turbo codes. An attempt has been made to provide implementations of core topics, from which other coders can be built. Other aspects which contribute to the area of error control cording are also included, such as CRC codes for error detection, and the Euclidean algorithm.

The technical requirements start at a modest level, with extensive tutorials and background provided in the chapters associated with early labs, so that the labs can be started by students at the earliest point of the semester.

Some of the labs build up "infrastructure" for coding problems: linear feedback shift registers, Euclidean algorithm, or Galois field arithmetic. It is reasonable to ask: Why not provide the "infrastructure," so that students can better focus more on the core "coding" questions, and spend less time just writing programs. The answer is based on my pedagogical attitude: that students will understand better concepts which they have implemented themselves, and that the understanding should extend throughout the entire system.

In fact, a fair amount of code *is* provided. (For example, a templatized polynomial class is provided which supports the basic polynomial operations over a number of data types.) Given my pedagogical needs, why provide any infrastructure code at all? A partial justification is that this code provides potential examples, especially helpful for students who are not yet adept at C++. Also, the code provided is not at the "core" of ECC, and should be more or less background. (In fact, in previous years of teaching this material, my students *have* implemented the polynomial arithmetic entirely from scratch. But there are so many other ECC-related topics to cover that something had to give.)

On the polynomial class

At least technical aspect of the polynomial class deserves some explanation, that of the temporary structure. It was desired to be able to provide the ability to compute more-or-less arbitrarily complicated polynomial expressions, such as

$$p(x) = g(x)h(x) + a(x)b(x) - \lfloor g(x)/d(x) \rfloor$$

in a way that would be both fast and not unduly consumptive of memory. The approach taken here is based on an idea of my colleague Scott Budge which uses a linked list of temporary polynomials. A linked list of temporary polynomials is maintained. For each operator such as $+$ and $*$, the result is written into a temporary polynomial which is pulled from the linked list of temporaries. These temporaries contain a parameter describing the largest polynomial yet requested, and retain allocated space for this largest polynomial. When a temporary is requested, the next available temporary is examined to see if it is large enough. If it is, then the pointer to it is returned for use. If it it not large enough, then space is reallocated.

In many circumstances (particularly with code that is run multiple times, as in a probability of error computation), the pool of temporaries will quickly reach a steady state in which no more allocations are necessary. The temporaries run very quickly then.

Temporaries are requested until processing reaches an assignment operator, such as `=` or `+=`. Then the temporary values are copied into the left-hand side and the pointer to the list of temporaries is reset to the beginning of the list.

There are a few caveats and cautions to be aware of. First, the pool of temporaries does remain around. There are circumstances in which the pool might be quite large. Second, computing quotients and remainders makes use of their own pair of temporaries. One result of this is that multiple quotients in an expression could lead to incorrect result. But another, more positive, result is that if a quotient or remainder is computed, the other one is still available and can be obtained without recomputing.