Using Web-based Multimedia Applications to Enhance Mathematics Instruction with Reed-Solomon Codes

Gregory A. Gibson
Email: gagibson@ncat.edu
Department of Mathematics
North Carolina A & T State University
Greensboro, NC 27411 USA

Neil P. Sigmon
Email: npsigmon@radford.edu
Department of Mathematics and Statistics
Box 6942
Radford University
Radford, VA 24142 USA

Abstract

In many introductory and upper level college mathematics courses, due to time constraints, students are often not exposed to many real-life applications of the material. As a result, students can fail to see the importance of the topics covered. However, the use of technological tools can quickly bring practical applications of mathematics to life. Using these tools, students can quickly see how the mathematical concepts they cover have great value without the cumbersome background that would normally be needed in a more traditional setting. This paper describes a web-based multimedia module designed for calculus involving Reed-Solomon codes. Reed-Solomon codes are currently being used to ensure reliable transformation of information for many applications, including satellite communications and compact discs. A discussion is given describing how these codes were instrumental in transmitting visual images of the outer planets during the Voyager satellite mission. All materials involving the module accompanying this paper can be reached here.

1 Introduction

A question that can often puzzle undergraduate students concerns the applicability of mathematics in standard introductory college mathematics courses. Many students regard these courses as a series of numbers and symbols that are never used in their major field and in so called "real-life" applications. As a result, students sometimes find these courses dull, uninteresting, and fail to see their importance. Hence, they lack the motivation necessary to obtain a thorough understanding of the material.

Technological innovation provides an excellent tool to present interesting applications of mathematics that would be too lengthy and cumbersome to present in a traditional manner. Through the use of web-based multimedia and mathematical software, students can quickly be exposed to the basics of the application and see how mathematics is used in a "real-life" setting.

This paper describes how multimedia modules can be designed to demonstrate practical applications of mathematics covered in typical engineering calculus. Specifically, a module created for calculus involving Reed-Solomon codes will be discussed. Normally, Reed-Solomon codes are introduced to students in abstract algebra or a first course in coding theory. However, the multimedia and mathematical software that will be described can allow students to quickly see how the concepts they study have value in performing some of the mathematics required with Reed-Solomon codes without the background that would be needed to understand the codes in a traditional setting.

The modules are web-based and interactive. In a typical module, students are introduced to an interesting application topic through the use of video and picture demonstrations. Through the use of two Maplet assignment labs per module, students explore and discover important mathematical aspects of the problem. These applications are designed to increase students' awareness of the importance of the mathematics in their lives and careers. Faculty benefit in having easy access to up-to-date applications that many times are not provided in current textbooks.

More details concerning this work can be found in the paper by [Tang et al., 1999]. A more detailed explanation of Reed-Solomon codes and the Voyager satellite mission can be found in [Klima et al., 2007], [Sigmon and Stitzinger, 1996], and [Wicker and Bhargava, 1994].

2 MODULE IMPLEMENTATION

Error correcting codes involve creating a scheme for correcting errors in the transmission of information to ensure that the transmission is done reliably. Reed-Solomon codes are a very common error correcting technique widely used today. These codes involve sending information in the form of polynomial coefficients. This code has been used in satellite communications and is currently being used to correct errors that occur in compact discs and cable television transmissions. The Reed-Solomon code module for this module emphasizes the use of these codes in ensuring reliable visual image transmissions of pictures from the Voyager II satellite of the planets Uranus and Neptune. The steps for the implementation of this module are now described.

2.1 Video Demonstration of the Application

The Voyager satellite mission and its purpose are introduced through pictures and the web-based QuickTime video software. Figure 1 and Figure 2 (next page) illustrate photographs given in the module taken by Voyager II while exploring the planets Uranus and Neptune.

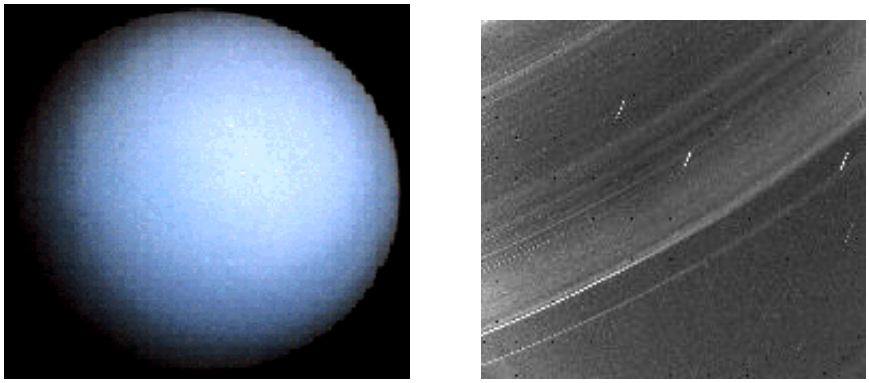

Figure 1: Pictures of Uranus and its rings (courtesy NASA/JPL).

To send pictures back to Earth from Uranus and Neptune, Voyager II had to transmit them in digitized form over a vast distance. Reed-Solomon codes were essential in ensuring these digitized pictures arrived on Earth in the correct form. The module carefully explains the history of the Voyager mission and the importance of Reed-Solomon codes for its success.

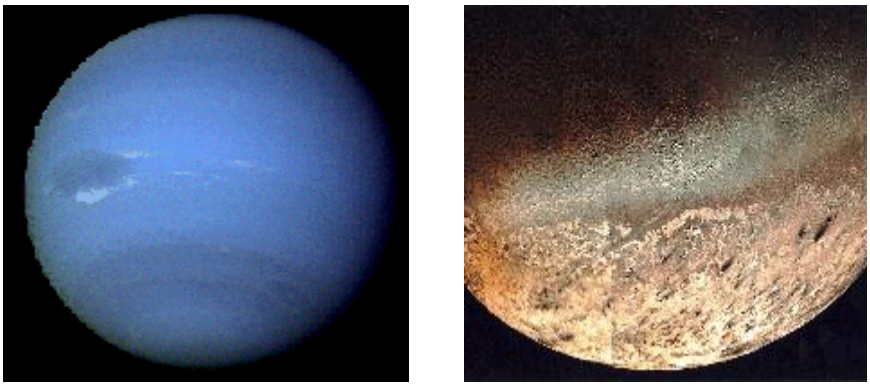

Figure 2: Picture of Neptune and one its moons, Triton (courtesy NASA/JPL).

2.2 Mathematical Computations

To send a picture from outer space to Earth, the image is commonly digitized into bits, which are strings of 0's and 1's, and sent over a space channel. Reed-Solomon codes encode these strings into polynomials where the transmission and error correction process can be done in a mathematical fashion. Many of the mathematical computations required to encode messages and correct errors in Reed-Solomon codes involve calculations such as polynomial multiplication, finding roots of polynomials, polynomial division, and differentiation that a typical calculus student can perform.

Due to the intensive computations required for Reed-Solomon codes, two Maplets are utilized to perform the mathematical calculations. Maplets allow one access to windows, dialogs, and other visual interfaces that are very simple to use and require no background programming knowledge. By simply typing information into text boxes and clicking buttons, students can easily execute the large amount of computations required almost instantaneously. The Maplets are produced using written code involving the symbolic manipulator Maple, which is a powerful programming language specially designed to perform symbolic, numeric, and graphic mathematical computations. However, to successfully use a Maplet, students do not have to have any working knowledge of Maple. They only need to have Maple installed on their machine and to run the end product of the Maple code used to construct the Maplet.

In this module, all numerical computations are done in modulo 2 (binary) arithmetic. Formally, this says that if a and b are integers, a is congruent to b modulo 2, denoted as $a \equiv b \pmod 2$, whenever 2 divides $a - b$. Hence, all numerical computations result in a 0 or 1. If 2 evenly divides a number, the resulting computation is 0. If not, the result is 1. For example, the number 6 in modulo 2 arithmetic reduces to 0, that is, $6 \equiv 0 \pmod 2$. However, the number 7 in modulo 2 arithmetic reduces to 1, that is, $7 \equiv 1 \pmod 2$. This arithmetic applies to all computations, including coefficients of polynomials.

For example, in modulo 2 arithmetic, the polynomial $7a^3 + 4a^2 - a + 2 = 1a^3 + 0a^2 + 1a + 0 = a^3 + a$.

We next demonstrate the basic mathematics required and how Maplets are used to perform the Reed-Solomon process. This discussion will present Reed-Solomon codes at the level that it is presented to a calculus student. A more advanced discussion of these codes can be found in [Klima et al., 2007].

2.2.1 Finite Field Generation

Reed-Solomon codes are constructed and decoded through the use of finite field arithmetic. Finite fields are a finite set of elements (in this paper polynomials) that exhibit special properties. Although calculus students are not introduced to all the properties that finite fields exhibit, some of the more basic and important ones are emphasized. We first state one of the most elementary facts:

Finite Fields contain p^m elements, where *p* is prime (an integer where the only divisors are 1 and *p*) and *m* is a positive integer. The positive integer *m* is the degree of a primitive polynomial (we discuss this below) used to generate the elements of the finite field.

In Reed-Solomon codes, the prime we use is $p = 2$. As an example, suppose we want to generate a finite field of $2^5 = 32$ elements. Note here that $p = 2$ and $m = 5$. Since $m = 5$, we need a primitive polynomial of degree 5. Only a selected number of polynomials are primitive. Tables for finding primitive polynomials can be found in [Lidl and Neiderreiter, 1986]. Fortunately, the Maplet we will soon display can quickly determine if a polynomial of a certain degree is primitive.

In the examples that follow, the primitive polynomial we will use is $p(x) = x^5 + x^3 + 1$. Note the degree of this polynomial is 5. Let $x = a$ be a root of $p(x)$. The root a is known as a *primitive element*. To generate the non-zero elements of a finite field, we perform repeated exponentiations on the primitive element a until all the non-zero elements are generated. For our example that contains $2^5 = 32$ elements, this requires generating the set $\{a, a^2, a^3, \ldots, a^{30}, a^{31} = 1\}$. Note the last non-zero element, in this case, a^{31}, will always be equal to 1 (a discussion of why this is true can be found in [Lidl and Niederreiter, 1986]. The finite field is completed by adding 0 to this set.

It turns out for reasons that will be seen later that a convenient way to represent the non-zero finite field elements is through a "polynomial representation". This representation is accomplished by using the fact the primitive element a is a root of the primitive polynomial $p(x)$. To see how this works, consider our finite field of $2^5 = 32$ elements generated by the primitive element a. We generate the first four non-zero elements of this field in the normal way, that is, we compute a, a^2, a^3, and a^4. However, once a^5 is reached, we note that a is a root of $p(x)$ and write

$$p(a) = a^5 + a^3 + 1 = 0.$$

Solving for a^5 and noting that all polynomial coefficients are reduced modulo 2 gives

$$a^5 = -a^3 - 1 = a^3 + 1.$$

Hence, $a^3 + 1$ is the polynomial we will use to represent a^5 in the finite field. Subsequent elements can be generated in the following manner:

$$a^6 = aa^5 = a(a^3 + 1) = a^4 + a,$$
$$a^7 = aa^6 = a(a^4 + a) = a^5 + a^2 = a^3 + 1 + a^2 = a^3 + a^2 + 1,$$
etc.

This process continues until the last non-zero element, a^{31}, is reached. Generating all the polynomial representations of non-zero elements by hand requires a tedious process. However, the Reed-Solomon codeword generator Maplet given in Figure 3 does this process quickly. We begin by entering the polynomial $p(x) = x^5 + x^3 + 1$ and verify that it is primitive by clicking the **Is p(x) primitive** button.

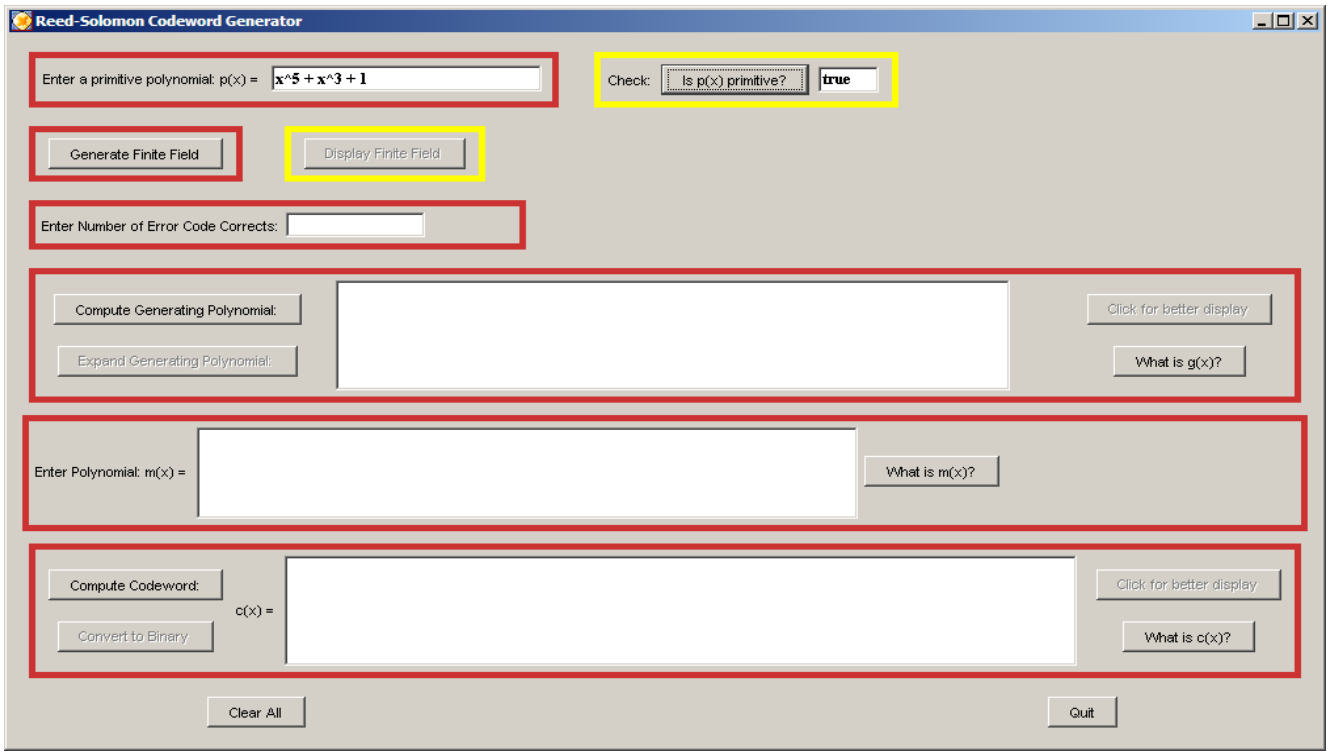

Figure 3: Reed-Solomon codeword generator Maplet verifying input polynomial is primitive

By clicking **Generate Finite Field** and the **Display Finite Field** buttons, the elements of the finite field are displayed in a separate window given by Figure 4.

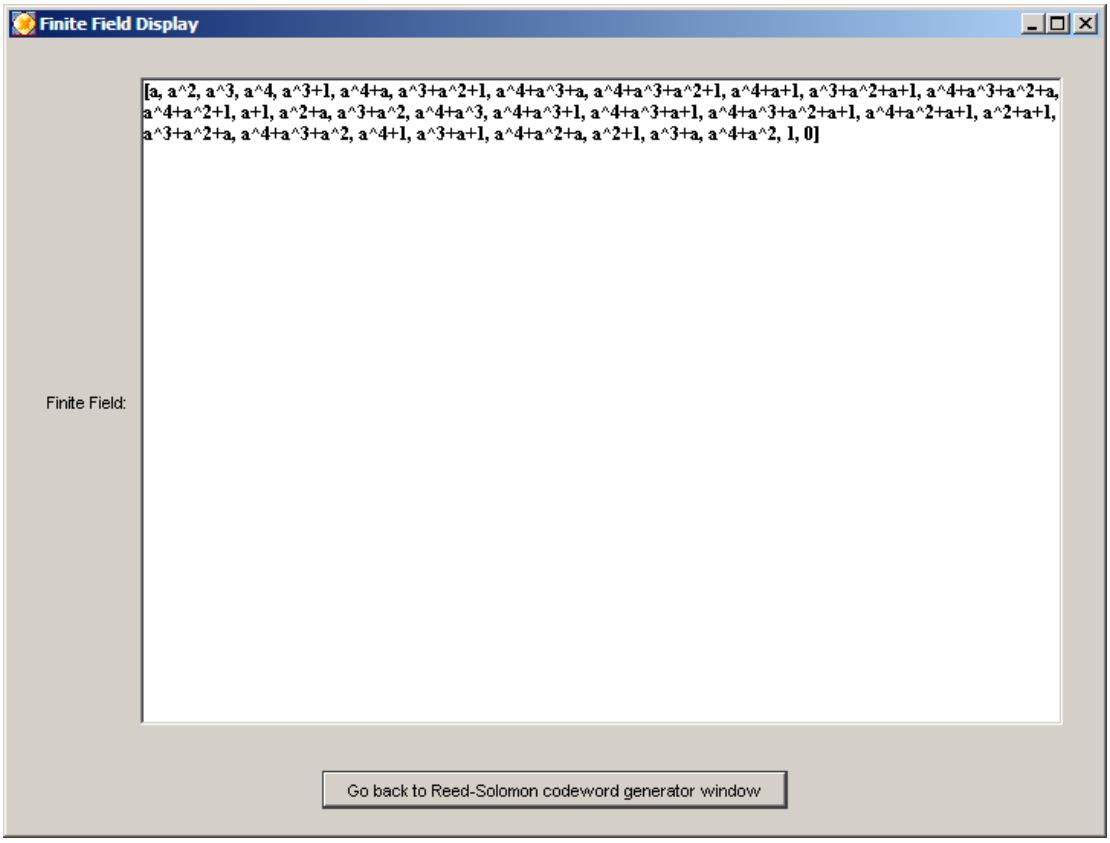

Figure 4: Finite field display

2.2.2 Encoding a Message to Be Sent

When setting up a scheme to transmit a message, we must specify the maximum number of errors that can be corrected in the transmission upon arrival to the message's destination. Reed-Solomon codes can correct up to a certain number of errors in a transmitted message. Let t be this specified number of errors. Then as long as $2t < 2^m - 1$, where 2^m is the number of finite field elements (recall for this example the number is $2^5 = 32$), then the scheme is guaranteed to correct t errors.

For the following example, we will use a $t = 4$ error correcting code. Note for this value of t that $2t = 8 < 2^5 - 1 = 31$. In error correcting codes, the transmitted source message is called a *codeword*. Codewords in Reed-Solomon codes are created by taking multiples of the polynomial

$$g(x) = (x-a)(x-a^2)(x-a^3)\cdots(x-a^{2t}).$$

Here, $g(x)$ is called the *generating polynomial*. That is, to create a codeword, we take a polynomial $m(x)$ of degree less than $2^m - 1 - 2t$ and multiply it by the generating polynomial $g(x)$. This gives the codeword $c(x) = m(x)g(x)$. Figure 5 illustrates the Reed-Solomon codeword generator Maplet where for a 4 error correcting code, a codeword is computed using the generating polynomial

$$m(x) = a^8 x^{11} + a^{27} x^{10} + a^{20} x^9 \ .$$

The generator polynomial is computed by clicking the **Compute Generating Polynomial Button** and the codeword is computed by clicking the **Compute Codeword** button.

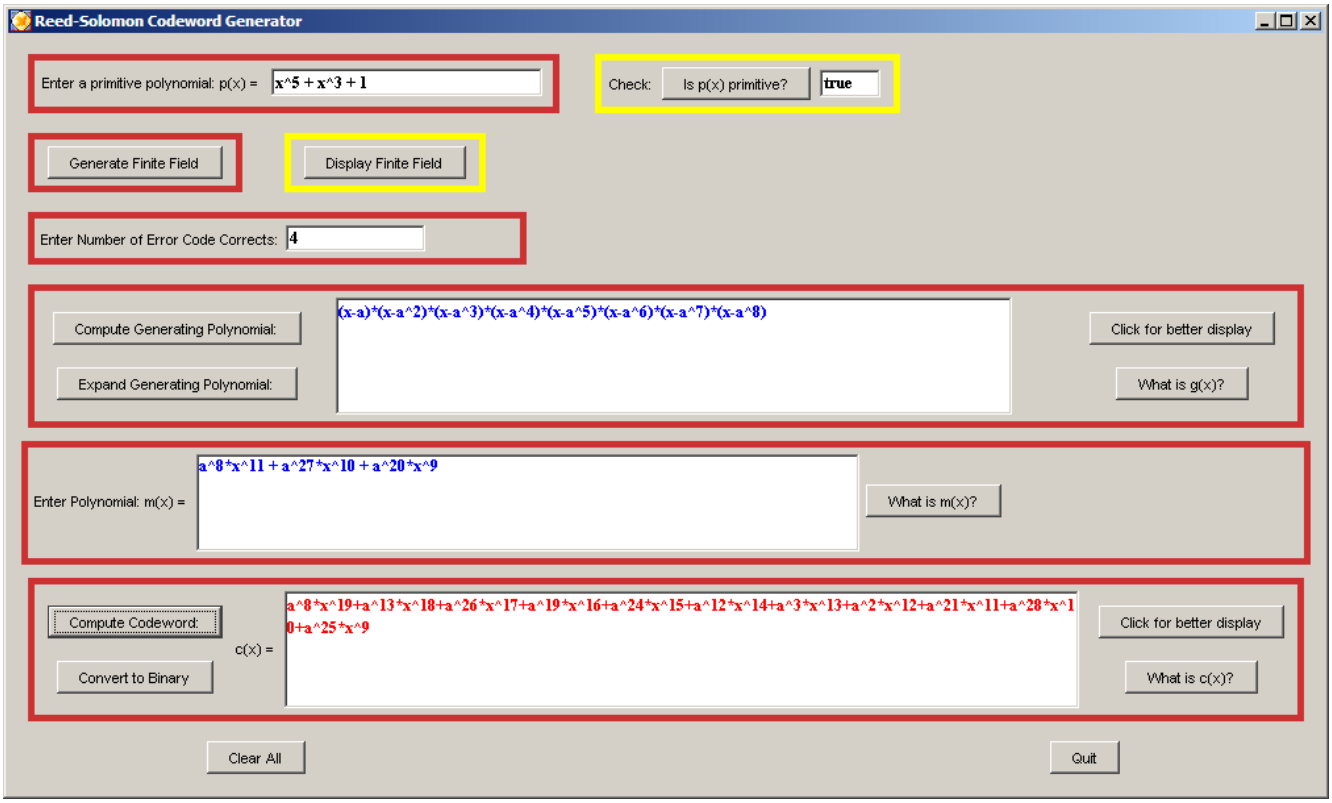

Figure 5: Reed-Solomon codeword generation example

Thus, the codeword is the polynomial

$$c(x) = a^8 x^{19} + a^{13} x^{18} + a^{26} x^{17} + a^{19} x^{16} + a^{24} x^{15} + a^{12} x^{14} + a^3 x^{13} + a^2 x^{12} + a^{21} x^{11} + a^{28} x^{10} + a^{25} x^9 \ .$$

The coefficients of the codeword contain the transmitted information. In practice, these coefficients are transmitted in binary, that is, as a string of 0's and 1's. To describe how the coefficients of $c(x)$ are translated to binary, consider the first coefficient of the codeword, a^8. If we use the process described

in Section 2.2.1 with the primitive polynomial $p(x) = x^5 + x^3 + 1$ for converting finite field elements to polynomials, this element has the polynomial representation

$$a^8 = a^4 + a^3 + a = (0)1 + (1)a + (0)a^2 + (1)a^3 + (1)a^4.$$

Reading off the coefficients gives the binary string 01011. The other coefficients are converted to binary in a similar manner. Note that the number of binary digits in this string, five, is the same as the degree of the primitive polynomial used to generate the finite field. In the Reed-Solomon codeword generator Maplet, clicking the **Convert to Binary** button converts the codeword to its equivalent binary representation. Figure 6 shows the result for this example.

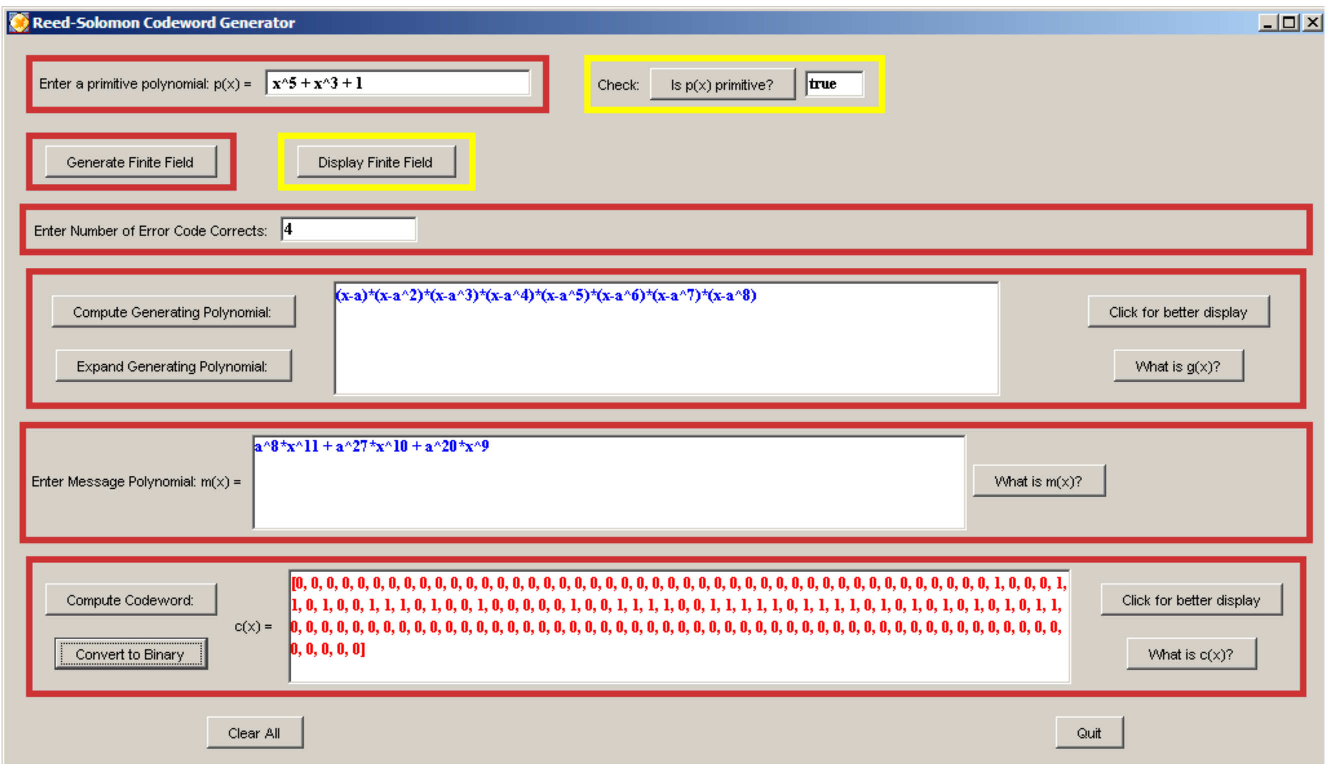

Figure 6: Binary Representation of the codeword

2.2.3 Checking a Received Message for Errors

Suppose the codeword is transmitted and a message received. Our goal is to check if there are errors in the transmission, and if so, correct them. It is in this process that calculus students see the importance of basic polynomial differentiation since computing the derivative of polynomials is essential in the error correction process. To demonstrate this process, suppose we receive the following message that has been converted from binary to polynomial form.

$$r(x) = a^{24}x^{15} + a^{13}x^{14} + ax^{13} + a^{10}x^{12} + a^{7}x^{11} + a^{10}x^{10} + a^{19}x^{9} + a^{22}x^{8} + a^{29}x^{7} + a^{20}x^{6} + a^{17}x^{5} + a^{21}x^{4} + a^{19}x^{3} + a^{3}x^{2} + a^{25}x + a^{12}$$

Recall that a codeword $c(x)$ is obtained by forming the product of the polynomial $m(x)$ and the generating polynomial $g(x)$. That is, $c(x) = m(x)g(x)$ where

$$g(x) = (x-a)(x-a^2)(x-a^3)\cdots(x-a^{2t}).$$

Because of how $c(x)$ is constructed with the generating polynomial $g(x)$, the terms $a,\ a^2,\ a^3,\ldots,\ a^{2t}$ are all roots of $c(x)$. This provides the basis for checking whether a received message is a codeword.

Suppose that we receive a message $r(x)$ which is not a codeword, that is, suppose $r(x) \neq c(x)$. Then $r(x)$ will not be a multiple of the generating polynomial $g(x)$. If so, then there will be at least one of the terms $a,\ a^2,\ a^3,\ldots,\ a^{2t}$ that is not a root of $r(x)$. In mathematical notation, this says

$$r(a^i) \neq 0 \text{ for at least one } i = 1\ldots 2t.$$

The values of $r(a^i)$ evaluated from 1..2t are known as *syndromes*. If we substitute $a, a^2, a^3, \ldots, a^{2t}$ into $r(x)$ and all evaluate to be 0, then $r(x)$ is a codeword and we can stop. If we substitute $a, a^2, a^3, \ldots, a^{2t}$ into $r(x)$ and any evaluate to not be 0, then $r(x)$ is not a codeword and we must proceed to the necessary steps to correct it.

Figure 7 (next page) illustrates the Reed-Solomon code error corrector Maplet designed to check a received message for errors and correct them if necessary. In this Maplet, the primitive polynomial $p(x) = x^5 + x^3 + 1$ and the maximum number of errors (t = 4) the code can correct are entered. The Maplet allows received messages to be entered in binary or polynomial form. Since polynomial form is used, the **Polynomial** button is clicked and a separate window is given for entering the received polynomial shown in Figure 8 (next page). The syndromes are found by clicking the **Compute syndromes** button. In Figure 7, the syndromes are $[a^4, a^{27}, a^{18}, a^5, a^{13}, a^3, a^{25}, a^{23}]$. Since the syndromes must all be zero for the message to be a codeword, we must proceed and correct the errors.

2.2.4 Error Correction

The initial step for correcting the received polynomial $r(x)$ involves using the Euclidean algorithm. A discussion of this algorithm can be found in [Klima et al., 2007]. Our goal for executing this algorithm is to determine a polynomial $R(z)$ known as the *error evaluator polynomial* and a polynomial $V(z)$ called the *error locator polynomial*.

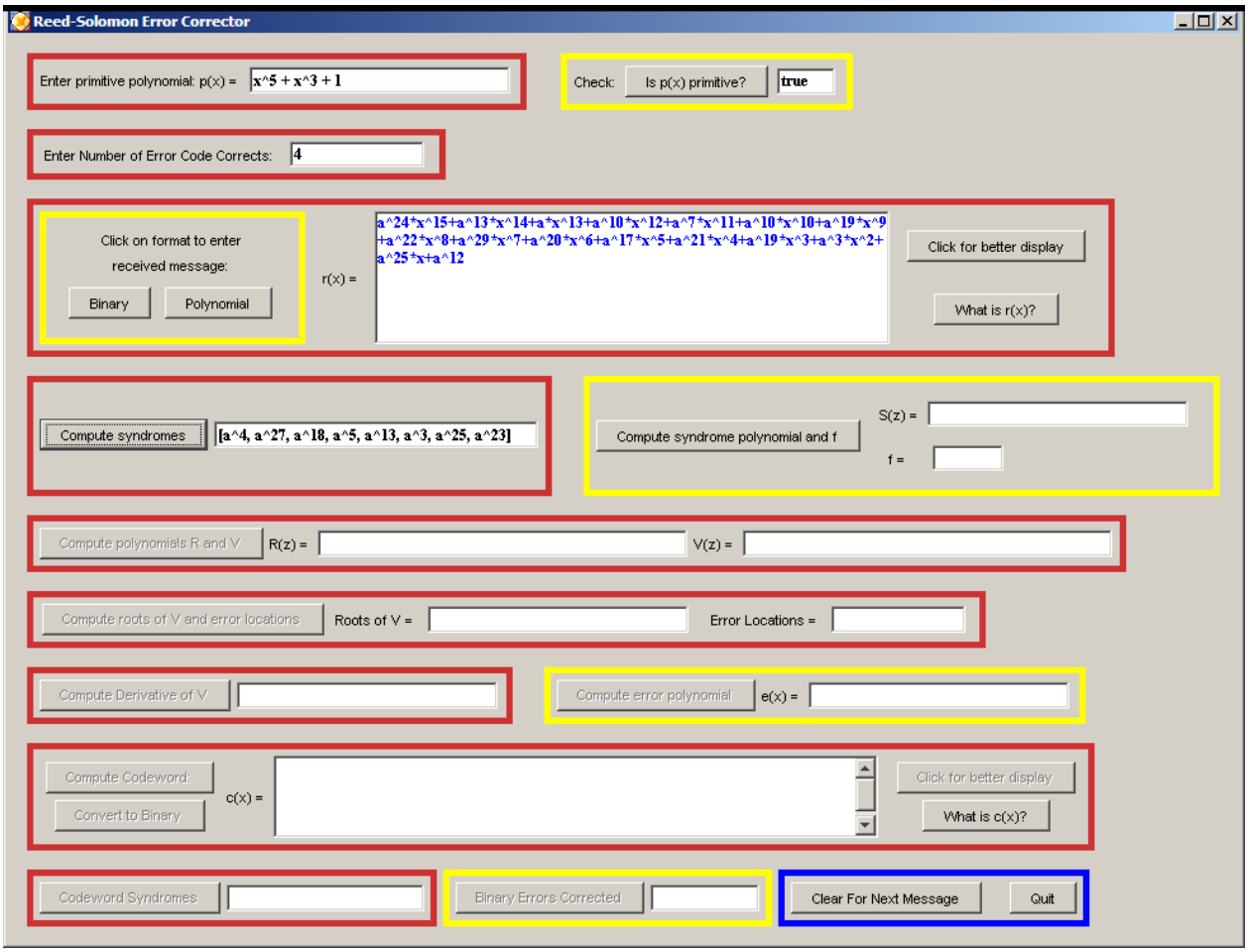

Figure 7: Reed-Solomon Error Corrector Maplet

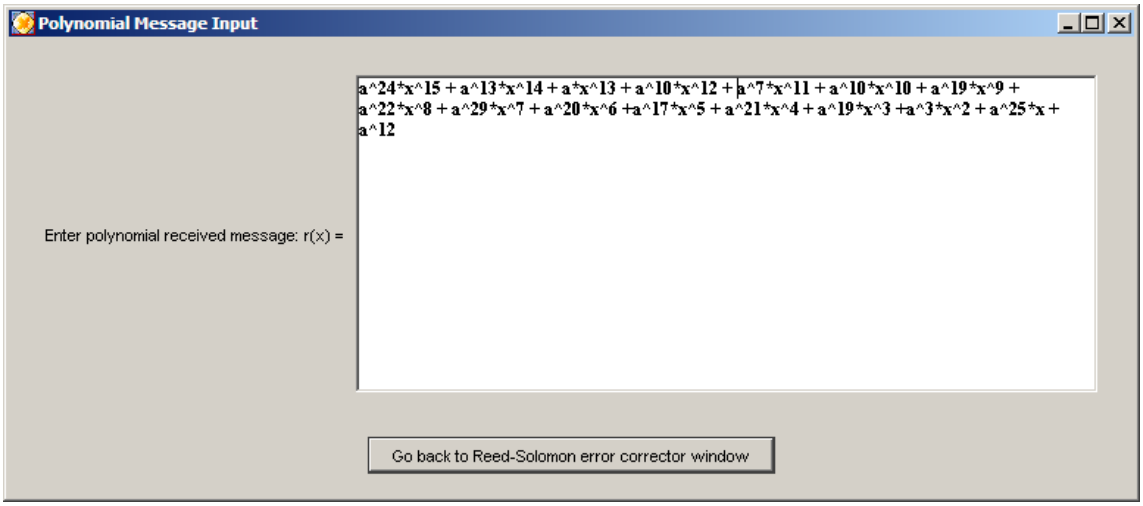

Figure 8: Maplet window for entering received polynomial

As a preliminary, using the syndromes found above as coefficients, we form a function of degree $2t - 1$ known as the *syndrome polynomial* $S(z)$. Using the syndromes $[a^4, a^{27}, a^{18}, a^5, a^{13}, a^3, a^{25}, a^{23}]$ for the $t = 4$ error correcting code, this polynomial would be

$$S(z) = a^4 + a^{27}z + a^{18}z^2 + a^5z^3 + a^{13}z^4 + a^3z^5 + a^{25}z^6 + a^{23}z^7.$$

The next step requires executing the Euclidean algorithm on the syndrome polynomial $S(z)$ and the polynomial $f(z) = z^{2t} = z^8$. This process, discussed in detail in [Klima et al., 2007], uses repeated polynomial division and is beyond the scope of what is normally discussed in calculus. However, the Reed-Solomon code error corrector Maplet quickly provides the necessary results. The output of this process gives the error evaluator polynomial $R(z) = a^{26}z^2 + a^{22}z + a^{14}$ and the error locator polynomial $V(z) = a^6z^3 + a^{18}z^2 + a^9z + a^{10}$. Figure 9 illustrates the Maplet output that is achieved by clicking the **Compute syndrome polynomial and f** and **Compute polynomials R and V** buttons.

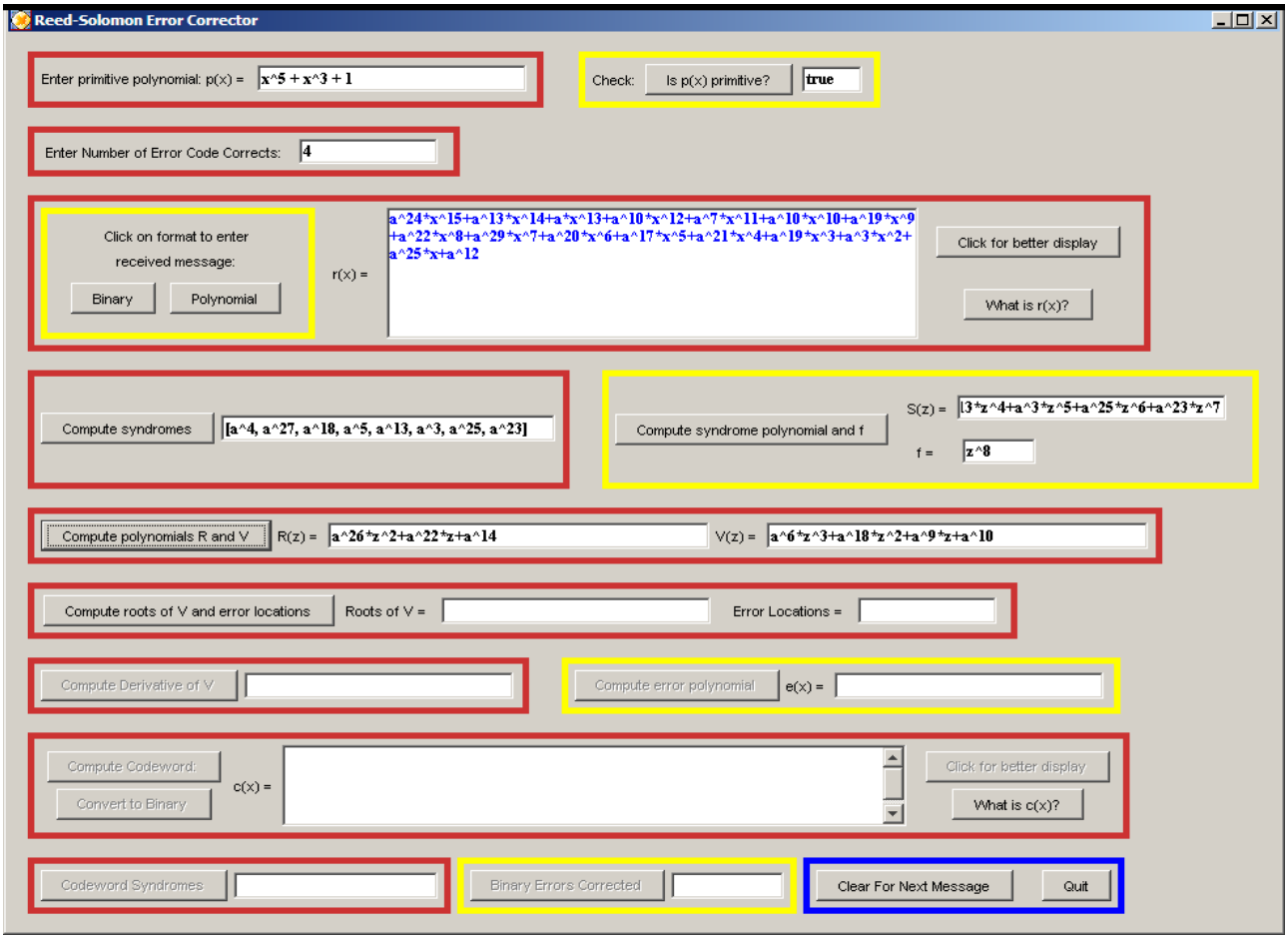

Figure 9: Maplet output for error evaluator polynomial *R*(*z*) and the error locator **polynomial *V*(*z*)**

The polynomials $R(z)$ and $V(z)$ are used to determine the polynomial coefficient location of the errors. Since $c(x)$, the codeword, and $r(x)$, the received message, are not equal, their polynomial coefficients must differ at certain powers of x**.** The difference in these coefficients makes up the coefficients of an error polynomial that we will denote as $e(x)$. Using $e(x)$, we can find the codeword $c(x)$ by computing $c(x) = r(x) + e(x)$. Our goal is to find $e(x)$, which is what $R(z)$ and $V(z)$ are used for.

The roots of $V(z)$ determine the polynomial location of the errors. In the Reed-Solomon process, each root of $V(z)$ will be one of the non-zero elements in the finite field. Recall in the above example that we generated a finite field of $2^5 = 32$ elements, which gave us 31 non-zero elements. By clicking the **Compute roots of V and error locations** button in the Reed-Solomon error corrector Maplet, the roots of V and corresponding error positions are computed. Figure 10 displays the result.

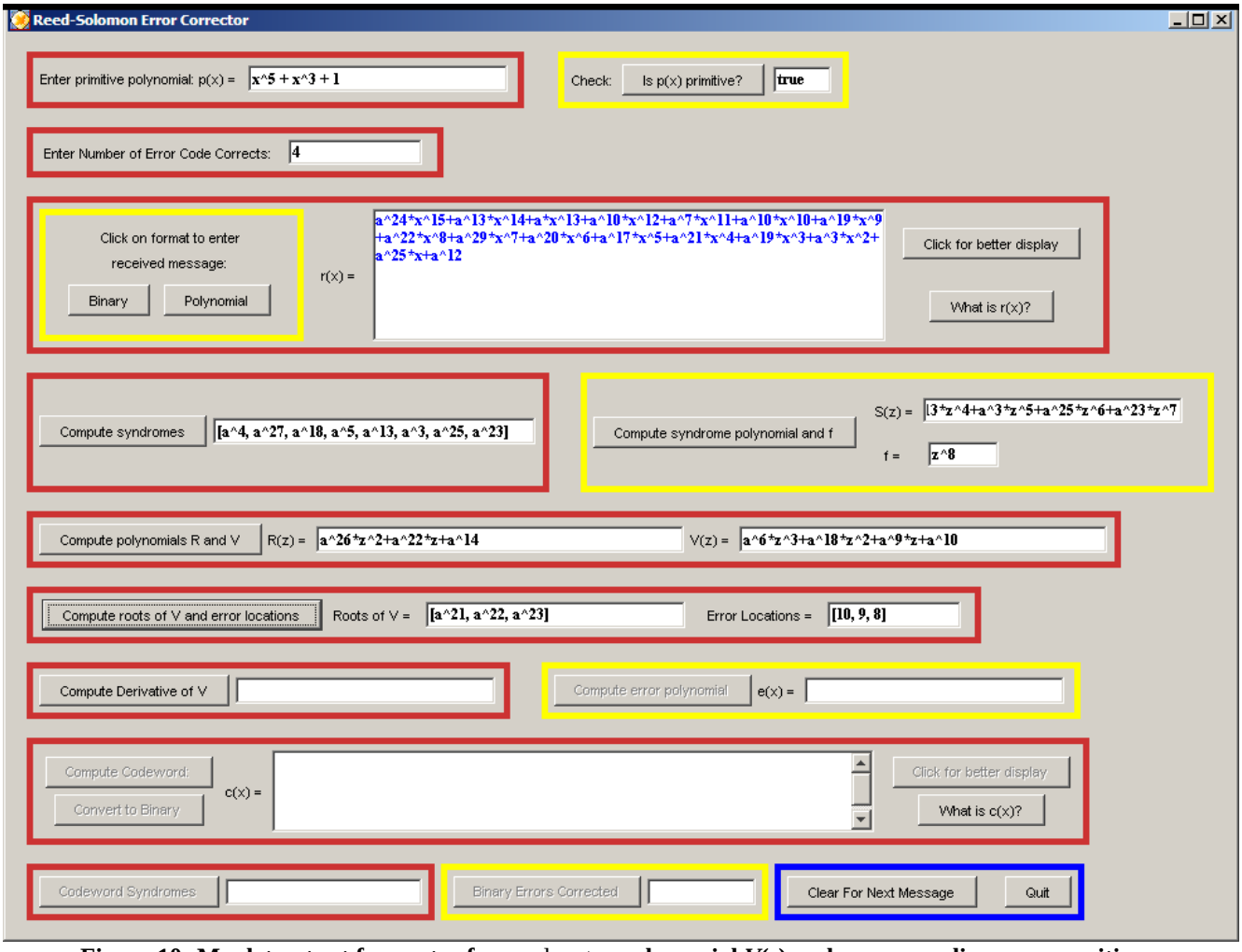

Figure 10: Maplet output for roots of error locator **polynomial *V*(*z*) and corresponding error positions.**

The output says, for example, that a^{21} is a root of $V(z)$, that is $V(a^{21}) = 0$, and that an error position occurs at the x^{10} location, which will be a term that makes up $e(x)$. There is a relationship between the exponent of the root a^{21} of $V(z)$, 21, and the error position of 10. Recall that there are 31 non-zero finite field elements. The exponent of the root of $V(z)$ and the error position will always add to give the number of non-zero finite field elements. Looking at the output from Figure 10, note that 21 + 10 = 31, 22 + 9 = 31, and 23 + 8 = 31. The output from Figure 10 also tells us that the difference between $r(x)$ and $c(x)$ occur in the coefficients of x^{10}, x^9, and x^8, which make up the terms of $e(x)$. Since we have three roots, this indicates that three errors have occurred.

Knowing the terms that make up $e(x)$, we now must determine what the coefficients of these terms are. That is, we must determine the coefficients of the terms x^{10}, x^9, and x^8. The error coefficient positions can be computed using the formula

$$e_j = \frac{R(a^i)}{V'(a^i)}, \tag{1}$$

where a^i is a root of the error locator polynomial $V(z)$, $i + j$ sums to the number of non-zero finite field elements, and e_j is the polynomial coefficient for the term x^j for the j^{th} error position in the error polynomial $e(x)$. Equation (1) is the error evaluator polynomial $R(z)$ divided by the derivative of the error locator polynomial $V(z)$. Both are evaluated at a root of the error locator polynomial $V(z)$ used to find the error position. For example, the term a^{21} was a root of the error locator polynomial $V(z)$ and the error position was 10. Thus, equation (1) says the coefficient of the error position x^{10} is

$$e_{10} = \frac{R(a^{21})}{V'(a^{21})}.$$

The coefficients e_9 and e_8 of $e(x)$ are determined similarly using (1). Once its coefficients are determined, the error polynomial $e(x)$ is computed and the codeword can be found. To get the codeword, we simply add the error polynomial $e(x)$ to the received polynomial $r(x)$, that is, we compute $c(x) = r(x) + e(x)$. Figure 11 (next page) illustrates the result of using the Reed-Solomon error Maplet to compute the derivative $V'(z) = a^6 z^2 + a^9$ of the error locator polynomial $V(z)$ and the error polynomial $e(x) = a^{14}x^{10} + ax^9 + a^{12}x^8$. The codeword is then computed to be

$$c(x) = a^{24}x^{15} + a^{13}x^{14} + ax^{13} + a^{10}x^{12} + a^7x^{11} + a^4x^{10} + a^{18}x^9 + a^{18}x^8 + a^{29}x^7 + a^{20}x^6 + a^{17}x^5 + a^{21}x^4 + a^{19}x^3 + a^3x^2 + a^{25}x + a^{12}.$$

These terms are computed in the Maplet by clicking the **Compute Derivative of V**, **Compute error polynomial**, and **Compute Codeword** buttons respectively. To verify that $c(x)$ is a codeword, click the **Codeword Syndromes** to verify that the syndromes are now all zero.

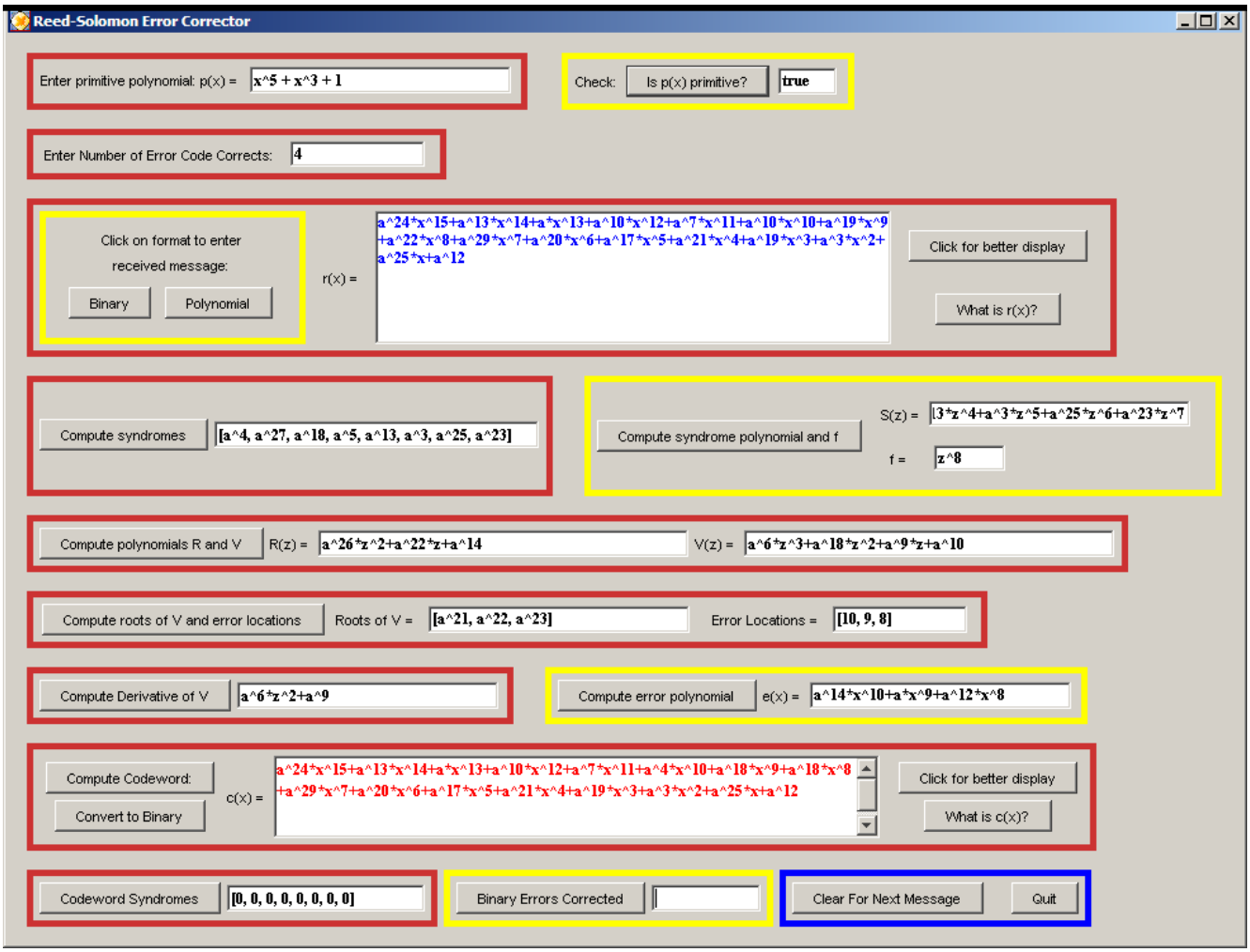

Figure 11: Maplet output error polynomial *e*(*x*) and codeword *c*(*x*).

By comparing the received polynomial

$$r(x) = a^{24}x^{15} + a^{13}x^{14} + ax^{13} + a^{10}x^{12} + a^{7}x^{11} + a^{10}x^{10} + a^{19}x^{9} + a^{22}x^{8} + a^{29}x^{7} + a^{20}x^{6} + a^{17}x^{5} + a^{21}x^{4} + a^{19}x^{3} + a^{3}x^{2} + a^{25}x + a^{12}$$

and codeword

$$c(x) = a^{24}x^{15} + a^{13}x^{14} + ax^{13} + a^{10}x^{12} + a^{7}x^{11} + a^{4}x^{10} + a^{18}x^{9} + a^{18}x^{8} + a^{29}x^{7} + a^{20}x^{6} + a^{17}x^{5} + a^{21}x^{4} + a^{19}x^{3} + a^{3}x^{2} + a^{25}x + a^{12},$$

we can see that 3 errors were corrected at the coefficients of x^8, x^9, and x^{10}. An important fact to note is that the errors occurred as consecutive coefficients. This constitutes what is called an *error burst*, an occurrence not uncommon in error correction.

Specifying the number of errors a code corrects refers to polynomial coefficient errors. In the example, the error polynomial $e(x)$ was made up of three terms, which constituted three polynomial errors. This will result in even more binary errors being corrected. By converting the codeword to binary by clicking the **Convert to Binary button** and clicking the **Binary Errors Corrected** button in the Reed-Solomon error corrector Maplet, Figure 12 illustrates that seven binary errors were corrected.

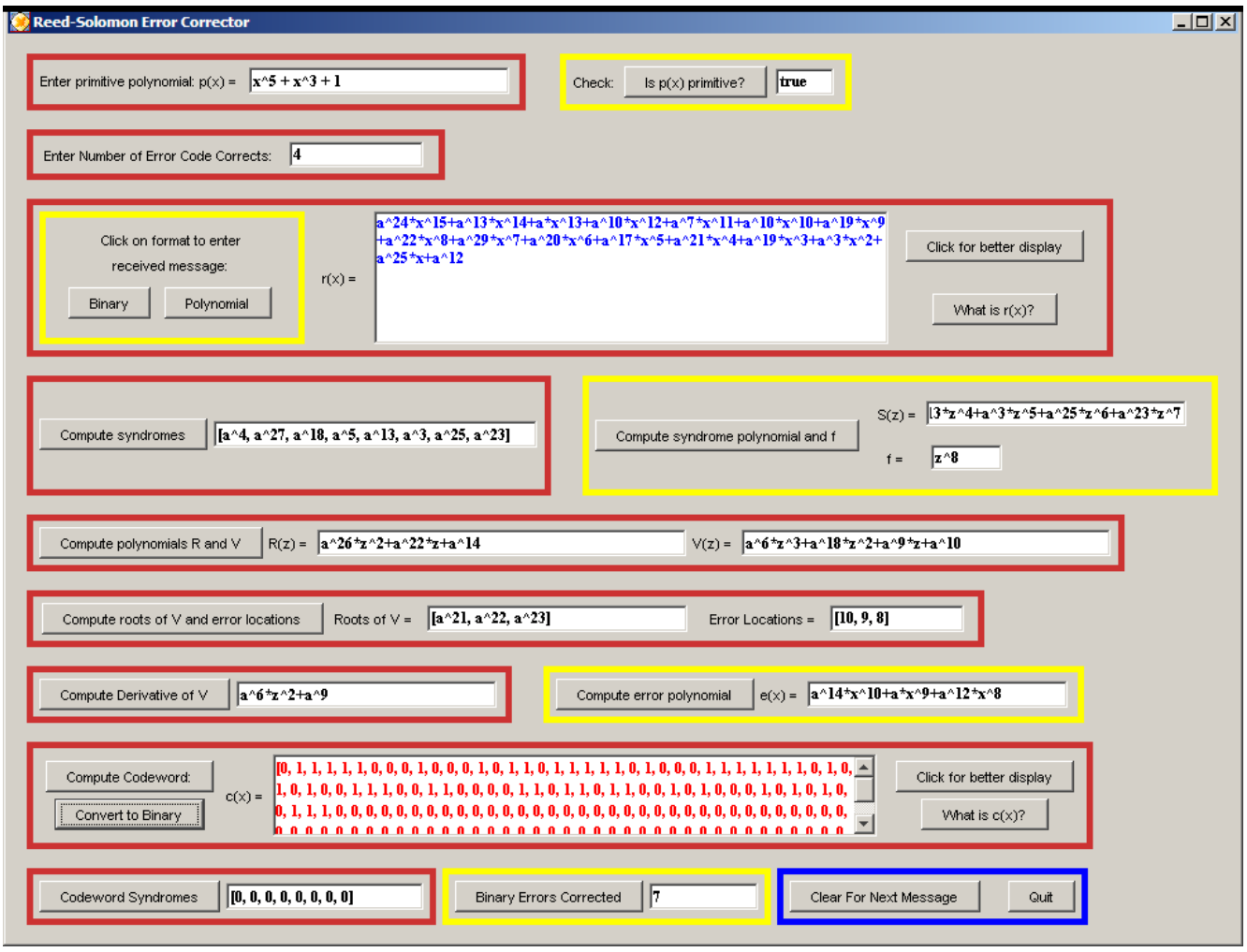

Figure 12: Maplet output for codeword $c(x)$ in binary and number of binary errors corrected.

2.3 Experimentation

After viewing the sample labs, the module provides several interactive questions that allow the student the opportunity to review basic concepts that are presented. Next, students are required as an assignment to work with the actual Reed-Solomon code used on Voyager II. This code consisted of a finite field made up of 256 elements generated using the primitive polynomial

$$p(x) = x^8 + x^4 + x^3 + x^2 + 1.$$

Codewords are constructed by taking multiples using the generator polynomial

$$g(x) = (x - a)(x - a^2)\cdots(x - a^{32}).$$

The assignment requires students to construct codewords and to correct messages using these parameters using the two Maplets.

2.4 Summary

To conclude, the module summarizes the Reed-Solomon process and emphasizes the basic mathematical computations that were utilized. The objective of this step is to again emphasize to the student that the mathematics involved with Reed-Solomon codes uses many of the mathematical techniques they have already been taught or are currently learning in calculus. By seeing these techniques used in a real-life setting, students will have a greater appreciation for their importance.

3 Teaching and Learning Methodology

The Reed-Solomon module is incorporated into calculus in a three-class sequence shortly after algebra review and basic differentiation concepts are discussed. In the first class, students are introduced to the multimedia software in an interactive computer lab. The Voyager application is introduced and the importance of Reed-Solomon codes in its success is emphasized in a video and picture format. Students are required as an assignment to view the entire multimedia presentation of the application and answer basic questions concerning its content. The second class introduces the mathematics involved with the application using Maplets. Students are required as an assignment to view the contents of the sample Maplet labs. The third class is designed to answer questions concerning the Maplet labs and to introduce the assignment where students are given the opportunity to implement the Reed-Solomon code used on Voyager. Once this final assignment is completed, students are asked give their perceptions and suggestions for improvement of the multimedia and Maplet presentations.

It should be pointed out that calculus students, after completing this module, are not expected to completely understand the theory behind why and how Reed-Solomon codes work. The major emphasis is to expose students to an interesting application of mathematics where they see how algebra concepts and polynomial differentiation are important in solving practical problems. In addition, this module can be extended to introduce Reed-Solomon codes to students in abstract algebra and introductory courses in coding. To do this successfully, more details describing how and why Reed-Solomon codes work can be added to the mathematical description (see [Klima et al., 2007] for details). The Maplets described in this paper can be used in their current form or students can be encouraged to write their own Maplets that perform the basic computations the code employs.

4 Current Status and Conclusion

The Reed-Solomon module has been tested in a section of Calculus I at Radford University and at North Carolina A & T State University. Overall, results were positive. Students have indicated that they enjoy learning about mathematics applications and would welcome the chance to learn more about other types. In addition, due to the fact that more advanced mathematics is needed to thoroughly understand Reed-Solomon codes, the module provides motivation for students to explore more advanced courses in mathematics.

Other modules involving circuits with applications to radio tuning and basic orbital mechanics have been developed. For more details on these applications, consult the paper [Sigmon, 2003] and the website [Sigmon, 2007].

Acknowledgement

This work was implemented with grant support from the National Science Foundation, DUE-9752266.

References

[1] Klima, Richard E., Sigmon, Neil P., and Stitzinger, Ernest L., 2007. *Applications of Abstract Algebra with Maple and MATLAB*, Taylor & Francis Group, LLC.

[2] Lidl, R. and Neiderreiter, H., 1986. *Introduction to Finite Fields and their Applications.* New York: Cambridge U. Press.

[3] Sigmon, Neil P., 2007. Website: http://www.radford.edu/~npsigmon/grant/nsfgrant.htm

[4] Sigmon, Neil P., 2003. "Determination of Satellite Orbits with Vector Calculus", *The UMAP Journal*, Volume 24, No. 1, 2003, 27-52.

[5] Sigmon, N. P., Stitzinger, E. L. , 1996. "Applications of Maple to Reed-Solomon Codes". *MapleTech*, Volume 3 (No. 3): 53-59.

[6] Tang, G., Ram, B., and Shah, M., 1999. Incorporating Engineering Applications into Calculus Instruction. *Proceedings of the 1999 American Society of Engineering Education Annual Conference*, Charlotte, NC.

[7] Wicker, S. B., and Bhargava, V. K., editors, (1994). *Reed-Solomon Codes and their Applications*. New York, NY: The Institute of Electrical and Electronics Engineers, Inc.