SIMU 2001–Coding Theory Seminar
Computer Laboratory Exercises
Draft – May 24, 2001

Background and Goals

Part of our work this summer will be carrying out calculations with codes using the *Maple* computer algebra we will want to get familiar

General Information about Maple

Launching Maple

To get into Maple, you will need to:

1) Turn the computer and monitor on.
2) Wait a few moments while the computer "boots up" and launches Windows. *Add directions specifically for Humacao setup*
3) *Add directions specifically for Humacao setup*
4) *Add directions specifically for Humacao setup*

Maple Worksheets

Worksheets are integrated text/graphics/mathematics documents where any or all of the following can be done:

a) you can type in commands from the keyboard to ask Maple to perform many different kinds of calculations, read data from external files, store work in files, etc.
b) output generated by Maple from your input commands (numerical values, symbolic formulas, and graphics) will be displayed,
c) you can modify commands, generating new output, store your results for use later, etc.
d) you can enter text to annotate and explain the results of computations.

Much of your interactive "management" of a worksheet in progress will take place through the File, Edit, View, Insert, etc. pull-down menus across the top, the "tool bar" below them with the icons for various operations (can you guess what each of them does?) the "scroll bar" on the right that you can use to move around within the worksheet, to see previous input and output lines, etc. Most of the common operations appear both as items in one of the pull-down menus and as toolbar buttons. The toolbar buttons are faster to use, but somewhat limited in the options they offer, so you will want to have a general idea what is accessible from where in the pull-down menus.

Input and Text Regions in a Worksheet

A new worksheet window will be labeled something like Untitled (1) - [Server 1]. On the first line of the worksheet, there is a [> in red. This is the Maple *input prompt* – the signal that Maple is ready to accept a command from you and try to execute it. A new input prompt will be generated automatically in a worksheet each time you enter a command and execute it *at the end of a worksheet in progress*. You can also go back and insert new commands and output at any point in a worksheet as follows. You can *insert an input prompt* and an input region at any point in the worksheet by placing the cursor at the desired location and pressing the toolbar button marked with [>. (After executing an inserted command like this, Maple will drop to the next input line, so to insert several input lines in the middle of a worksheet, you will need repeat the above.)

As we mentioned before, Maple worksheets can contain text as well as commands and output. To create a text region you can press the [> button above, then the button marked by a capital T (this changes the region into a text region). Any text can now be entered, and this will be treated as an "inert" comment. That is, it will appear as you enter it when you print out the worksheet, and it will not be treated as Maple input. After you create a text region, if you press the capital sigma (Σ) toolbar button with the cursor placed on that region, you can type in a Maple expression in input format, and have it displayed (in "inert" form) in usual mathematical notation (with raised exponents, no asterisks for multiplication, etc. This is sometimes useful for creating "pretty" worksheets!

What is a Maple Command?

It may help to think of a command as a *program* or *function of one or more variables* that Maple knows how to compute, given "inputs" from you. You must then keep straight the distinction between these functions enter for Maple to work with!

Almost all of the built-in Maple commands have a syntax like the `plot` command discussed in Day 1 below. The format is either

`name(values);` or `name(values):`

In this general description, `name` is the name of the command or function (like "plot" yesterday), there is a matching set of open and close parentheses following the name, and they surround the `values` – the input from you the command needs to do its job (like the formula of the function to be plotted and the range of x-values you want). The values are listed separated by commas within the parentheses. The two forms of the command above tell Maple to execute the command when you press ENTER on that input line. In the first case – ending the command with a semicolon (;) – you are instructing Maple to display the output. With a colon (:), Maple will execute the command but not display the output (this is useful sometimes for intermediate steps in a big computation where you don't want or don't need to see the output).

Getting Help On-Line

The exact format Maple is expecting for each type of command is specified in the programming of the Maple system. Usually, there is *little or no freedom* in the ordering of what has to go where and in what format for all the built-in commands, and LOTS of instructive examples are contained in the Maple on-line HELP facility. For example try looking at the on-line help page for the `plot command`. (From the HELP pull-down menu, select Topic Search, type in the word "plot," and press OK.) Note the example plot commands at the bottom and the links to related help pages.

Saving and Reloading your Maple Worksheets

When you begin working on a worksheet, you will usually want to *save your work* every once in a while in case you need more than one lab session to complete the work you are doing. This can be done most directly by saving to your network drive. (You can also save to a 3 1/2 inch Follow these directions:

1) If you are using a diskette, insert it in the diskette drive (it's usually called the a: drive). Otherwise go directly to step 2.
2) In Maple, select the SAVE option from the FILE pull-down menu.
3) If you are saving your work for the first time, you will see a SAVE AS dialog box. In the Drives box, highlight the A: or network drive as appropriate, then go to the File Name box, and type in a name for the file where the worksheet will be stored (any string of letters and digits – no spaces – no more than 8 characters long is OK) type in a period after the file name you chose and then the "extension" *mws* for *Maple Work Sheet*. For instance, a good choice might be something like *lab1.mws*. Then click the OK button to write the worksheet to the network drive or to your diskette. (You will only need saves just update the file.)
4) When you have the worksheet saved as you want it, you can remove the diskette and exit Maple.
5) To update the worksheet further in a later session, you can read the worksheet back into Maple using the OPEN option from the FILE Menu, or the "open" toolbar button. Get into Maple as above, (if the worksheet is stored on diskette, insert it in the a: drive), and select OPEN from the FILE Menu. Maple will prompt you as above for the name of the worksheet with a dialog box very like the SAVE AS box described above. Highlight the A: or network drive as appropriate, then the worksheet you want, and click OK.

IMPORTANT NOTE: When new Maple session, you commands in it in the new session (even though output saved as part of the worksheet file will display). If you want to make use of any results in that execute the commands again. In the EDIT pull-down menu, you will see an EXECUTE option that lets you recompute an entire worksheet or a section of one.

Printing

With the worksheet you want to print open and highlighted, and press the toolbar icon that looks like a printer, and press OK on the PRINT dialog box (the settings should be set up correctly to print automatically). Your output should appear shortly on the printer.

Getting Out

When you leave, quit the Maple window (FILE/EXIT).

Week 1, Day 1: A First Sample Maple Session

Let's get right down to work and walk through a first sample session illustrating some of the different features of this program. First, you will need to get into Windows and Maple as described in the General Information section above.

2D Plotting

The basic Maple command for 2D plotting, used for graphs of the form $y = f(x)$ (and also for parametric curves $\alpha(t) = (x(t), y(t))$) is called `plot`. The basic format for graphs $y = f(x)$ is

```
plot(function,range,options);
```

where

1) `function` is the function to be plotted – the simplest way to specify one is via a formula for $f(x)$ (an *expression* in Maple)
2) `range` is the range of x-values you want to see plotted, entered in the form `x = low..high`, and
3) `options` can be used to control the form of the plot if desired. No options need be specified however if you don't want to.

Exercise 1

To plot $y = x^4 - 2x^3 + x - 5\sin(x^2)$ for $-2 \le x \le 1$, you could use the Maple `plot` command with no options:

```
plot(x^4-2*x^3+x-5*sin(x^2),x=-2..1);
```

Try this now. (Note: The close parenthesis comes directly after the range of x-values if there are no options.) Type in this command line (exactly as here) and press ENTER. (Until you enter either a semicolon or a colon and press ENTER, Maple will not do anything with your command. Also, if a command you want to enter doesn't all fit on one line, just keep typing, but don't press ENTER until you are finished. Maple automatically wraps around to a new line if you need it.)

If you make a typing mistake, Maple will let you know about it (!) One frequent source of errors in Maple sessions is unmatched parentheses, (or square or curly brackets, etc.) One "hacker's trick" is to count the open parens, count the close parens and make sure you get the same number (of course, if you don't, you also have to decide where to insert the missing one ...).

Fortunately, if this happens, the whole command does NOT need to be re-entered. Just move the cursor arrow to the place on the input line you want to change, press the left mouse button, and edit the input as needed. Typing from the keyboard will *insert* new stuff at the cursor location; the DELETE and BACKSPACE keys will remove stuff (DELETE removes the character in front of the "insert point"; BACKSPACE removes the character in back). You can also move around on the input line with the ARROW keys if more than one thing needs to be changed. When you think it's OK, press ENTER again to have Maple execute the command again. (Pressing ENTER with the cursor anywhere on an input line executes the whole input line.)

When all goes well you will see the graphics output displayed under the input command in the worksheet.

From the formula $y = x^4 - 2x^3 + x - 5\sin(x^2)$, you might guess that there is at least one other x-intercept for this graph for $x > 2$ (why?). To see that part of the graph as well, edit your previous command line to change the right hand endpoint of the interval of x values (do not retype the whole command). Press ENTER on that input line to have Maple execute the command again. Experiment until you are sure that your plot shows all the x-intercepts of this graph. (You can repeat this process of editing a command and re-running it as often as you want; the previous output is replaced by the new output each time.)

When you get a graphics window with *all* the x-intercepts shown, let's add one more thing. Sometimes, an informative title makes a graph much more understandable. To add a text title to a plot, you can insert a comma after the range of x-values, and include an option in the plot command of the form

```
title='whatever you want'
```

Add an appropriate title to your plot.

Maple Expressions

The formula for the function we plotted in the last example is a Maple *expression*. Expressions are entered in something like usual mathematical notation:

1) The symbols for addition, subtraction, multiplication, and division are $+, -, *, /$ respectively.
2) The caret (^) is the Maple symbol for raising to a power.
3) The asterisk symbol for multiplication MUST be included whenever you are performing a product in a formula. Moreover, everything must be entered in one linear string of characters, so you will need to use *parentheses* to group terms to get the expressions you want. The rule to keep in mind is: *Maple always evaluates expressions by doing powers first, then products and quotients, then sums and differences, all left to right, unless parentheses are used to override these built-in rules.* For example, the Maple expression `a + b*c^2/d + e` is the same as the mathematical formula:

$$a + \frac{bc^2}{d} + e.$$

If you really want

$$\frac{a + bc^2}{d + e}$$

you will need to enter the expression `(a + b*c^2)/(d + e)`.
4) Maple "knows" all the usual elementary functions from calculus. The names of the most common ones are `sin, cos, tan, arctan, arcsin, exp, ln, sqrt`. To use

one of these functions in a Maple formula, you put the name, followed by the "argument" (that is, the constant or expression you are applying the function to) *in parentheses*. exponentiation here – that is built into the definition of the `exp` function.

Exercise 2

a) What Maple expression corresponds to the mathematical formula

$$\frac{(a+bc)^2}{d+\frac{e}{f}}?$$

Enter your expression as input and check with Maple's output.

b) What Maple expression corresponds to the mathematical formula

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

(from the quadratic formula). Enter your expression as input and check with Maple's output.

c) What Maple expression corresponds to the mathematical formula

$$\frac{4t(\sin(5t^2)-e^{\frac{-3}{t^2}})}{1+t^2}$$

Symbolic Computation in Maple

Much of our work starting later this week in Maple will consist of *symbolic* and polynomials. To conclude today, we will look at some first examples of this sort of computation in Maple, with polynomials in one variable.

The assignment operator :=

Often we will want to save the result of one step in a computation for use in later steps. A Maple command of the form

```
name := expression;
```

assigns the right-hand side to the name on the left. (This is the same idea as the assignment statement in programming languages such as C++, or Pascal, Fortran, Basic, etc.) Usually, the right-hand side will be a known expression, or a command performing some operation on information we already know. The assignment can be understood as taking the result of the operation and storing it under the abbreviation `name` for later use. For instance,

```
cube := (x^2 - 16)^3;
```

Then later on, we could "reuse" the value of `cube` in other expressions, just by putting in the name `cube` at the appropriate place (e.g. `plot(cube, x=5..10);`). Note: Maple also gives you a second way to refer back to results of previous computations. At any point in a session, if you enter %, this stands for the result of the previously executed command, %% stands for the result of the command before that, and so on. (Caution: The sequence is the sequence of the commands you have executed in the current session, *not* the sequence of input lines in the worksheet. This can be confusing if you skip around, inserting new command lines between existing ones.)

Additional symbolic commands on polynomials

Maple has built-in commands:

- `expand` to multiply out (and also simplify) a factored polynomial. Format: `expand(poly);` This works on polynomials in any number of variables.
- `factor` to factor a polynomial. This gives the factorization with *rational number coefficients*. The factor command does not know about radicals, complex numbers, etc. Format: `factor(poly);` This also works on polynomials in any number of variables.
- `quo` to compute the quotient on division of one polynomial $f(x)$ by another $g(x)$. Format: `quo(f,g,x);` Note: the variable must be included. If f or g contain other variables besides x, they will be treated as symbolic constants in the coefficients.
- `rem` to compute the quotient on division of one polynomial $f(x)$ by another $g(x)$. Format: `rem(f,g,x);` Note: the variable must be included again.
- `diff` to compute derivative of a polynomial (or any other function) with respect to a given variable. Format `diff(f,x);`

Exercise 3

Here is a sample sequence of Maple input lines illustrating a possible symbolic computation using these commands. Enter and execute them in sequence. Look carefully at the output and make sure you understand what happened in each case:

```
cube := (x^2 - 16)^3;
dcube:=diff(cube,x);
expand(dcube);
dd:= x^2-x+1;
q:=quo(cubee,dd,x);
r:=rem(cubee,dd,x);
factor(expand(q*dd+r));
```

Assignment 1 – Due at end of Lab Period, Monday

Add text regions describing what you did in each of Exercises 1,2,3 above, then save and print out this worksheet and give to JC or Jesus.

Week 1, Day 2 – Matrices and Linear Algebra in Maple

In class today, we saw the definition of binary linear codes (vector subspaces of K^n). To work with these in Maple, we need to learn how to enter generator matrices and perform the basic matrix operations from linear algebra.

Maple has built-in data structures that can be used to represent vectors or matrices. The simplest of these is the *ordered list* structure, indicated by placing the components of the list, separated by commas, inside a pair of *square brackets* (`[, ]`). For example, the command

```
[1,0,1,1,0,1,1]
```

is an ordered list with 7 entries. To create a (row) vector with these entries, we use a command like this:

```
v:=vector([1,0,1,1,0,1,1]);
```

The result of executing this is a word of length 7 that could represent a word in a binary code of length 7. The individual entries of the vector can be accessed like this: `v[1]` is the first (left-most) entry, `v[2]` is the second (from the left), `v[3]` is the third, and so forth. (IMPORTANT NOTE: Maple has to be "told" that we want the zeroes and ones to represent binary digits or integers mod 2 when we work with this vector, though. We'll see how to do this in a moment! If you don't do that, the zeroes and ones are treated as ordinary integers.)

Matrices can be represented as *lists of lists*. Each component list gives one row of the matrix. For instance,

```
G:=matrix([[1,0,1,0,1,1,1],[0,1,0,1,1,0,1],[1,0,1,0,0,1,0],[1,0,0,1,1,1,1]]);
```

could be used to enter a 4×7 generator matrix for a linear code with $n = 7$. The entry in row i and column j of G is `G[i,j]`. The output from the assignment above shows the matrix in usual format, and without commas between the entries on each row. (The same IMPORTANT NOTE above applies here too.)

Because of the way Maple treats expressions involving matrices, in order to see the contents of a matrix or vector you have assigned to, you will need to use the `eval` function to "evaluate" the name. For instance after entering the assignment to the matrix G above, if you enter just `G;` as an input command, the output will be the letter G; to see the actual contents of the matrix, use `eval(G);`.

The Maple matrix and linear algebra commands are contained in a separate "package" called `linalg`. To use them, you will need to execute a command

```
with(linalg);
```

in the current session. The `linalg` package contains procedures for matrix and vector arithmetic, matrix operations such as determinants, row reduction, transposition, computing rank, etc., computation of eigenvalues and eigenvectors, solving systems of linear equations, finding the kernel of a matrix, etc. The Overview page on the `linalg` package in the on-line help contains a complete listing of the available commands with links to the help pages for each command.

Among the most important ones for us will be:

`submatrix` – select a submatrix from a given matrix
`concat,stackmatrix` – make a larger matrix by putting smaller matrices together
`matadd` – add two matrices of the same size
`multiply` – multiply two matrices of appropriate sizes
`Nullspace` – find the nullspace or kernel of a matrix
`Linsolve` – find the solutions of a general (i.e. possibly inhomogeneous) system of linear equations
`Gaussjord` – find the row-reduced echelon form of a matrix

Submatrices and combining matrices

To pick out a submatrix of a defined matrix, you just indicate the ranges of row and column subscripts for the submatrix. For example, with the 4×7 matrix G above, if you enter

```
submatrix(G,1..3,4..5);
```

the output will be the 3×2 submatrix of G formed from the entries in rows 1,2,3 and columns 4,5.

To create larger matrices out of smaller blocks, you can use either `concat` or `stackmatrix`. To see what each does, and the difference between them, enter the following sequence of commands:

```
A:=matrix([[1,1],[0,1]]);
B:=matrix([[1,0],[0,1]]);
concat(A,B);
stackmatrix(A,B);
```

Note that none of these operations depends in any way on what the entries of the matrix actually are.

Linear algebra operations mod p

For us, almost all our matrices will have entries in finite fields ($K = \{0, 1\}$ first, and other finite fields later). As mentioned above, Maple needs to be "told" to treat the entries of a matrix as elements of a finite field. This is done in two different ways, depending on the command.

- For the matrix arithmetic commands `matadd`, `multiply`, we simply perform the operation as though the matrices had ordinary integer entries, then take all entries in the matrix mod 2.

For example, with a vector and matrix defined as follows:

```
w:=[1,1,1,0];
G:=matrix([[1,0,1,0,1,1,1],[0,1,0,1,1,0,1],[1,0,1,0,0,1,0],[1,0,0,1,1,1,1]]);
```

The command

```
multiply(w,G);
```

computes the product wG taking the entries as *ordinary integers*. (By the way, what happens if you do `multiply(G,w);`?) There is a built-in Maple function `Normal` that reduces the coefficients in any expression modulo any prime. The format is `Normal(expression) mod p` where p is any prime number. For instance

```
modp(15,11);
```

returns 4. The command

```
Normal(34*x^2 - 2*x + 6) mod 3;
```

returns $x^2 + x$ (all coefficients reduced mod 3).

We want to apply this function to every entry in the product matrix. The appropriate command is:

```
map(x->Normal(x) mod 2,multiply(w,G));
```

This "maps" the function "reduce mod 2" to all entries of the product matrix and computes the product matrix mod 2.

- The linear algebra commands `Nullspace, Linsolve, Gaussjord` are simpler to use mod p (but also slightly subtle; see the discussion below and Exercise 3). These are set up so that if you enter, for instance,

  ```
  Gaussjord(G) mod 2;
  ```

  then all the computations in the row reduction to echelon form are done modulo 2.

(Technical note: The capital letters at the start of these commands *are significant*. In Maple terms, these are "inert" versions of the standard `nullspace, linsolve, gaussjord` commands that allow all computations to be done modulo a prime. The uncapitalized versions would compute the nullspace, solve the system, or row-reduce the matrix using ordinary *rational number arithmetic* on the matrix entries. The results produced by the "inert" versions can be different from the results of doing the standard command then reducing modulo 2. See Exercise 3 below.)

The `Nullspace` and `Linsolve` commands are similar. `Nullspace(A) mod 2` will compute a basis for the nullspace or kernel of the matrix A, doing all arithmetic modulo 2. For instance, if

```
C := matrix([[1, 1, 0, 0], [1, 0, 1, 1]]);
```

then `Nullspace(C) mod 2` produces output

$$\{[1,1,1,0],[1,1,0,1]\}.$$

These two vectors are found in the usual way by reducing C to row echelon form, identifying the "free variables" and setting each to 1 (others all 0) in turn.

The format of the `Linsolve` command is `Linsolve(A,B) mod 2`, where A is the coefficient matrix of the system, and B is a vector or matrix. For instance, with C as above, the command

```
Linsolve(C,vector([1,0])) mod 2;
```

produces:

$$[\_t_3 + \_t_4, 1 + \_t_3 + \_t_4, \_t_3, \_t_4]$$

The $\_t_3$ and $\_t_4$ are Maple's notation for arbitrary constants. Substituting any values (in $\{0,1\}$) for them produces some element of the solution space. Note that the vector above could also be written as the linear combination

$$[0,1,0,0] + \_t_3[1,1,1,0] + \_t_4[1,1,0,1]$$

using the vectors from the basis of the nullspace of C. This shows that *the set of solutions is a coset of the nullspace.*

Assignment 2, part a

1) Algorithm 2.5.1 in the text gives a method for computing a generator matrix for the binary linear code $\langle S \rangle$ spanned by a set of vectors S. Using the Maple commands discussed above, do parts e,f,h,i of Exercise 2.5.3 in the text.

2) Algorithm 2.5.7 gives a method for computing a generator matrix for the dual $C^{\perp}$ of the code $C = \langle S \rangle$ spanned by a set of vectors S. Using the Maple commands discussed above, do Exercise 2.5.12 parts b,d,f in the text.

3) In the discussion of the `Gaussjord` command above we saw how to perform the row-reduction modulo 2. A natural question is: If we start from a matrix of zeroes and ones, why couldn't we just do the *gaussjord* command, then reduce all the entries in the resulting matrix by 2 (as we did with sums and products)? In this question you will see that this is not possible in all cases (although it might work in some cases). Consider the matrix

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

What is the output from `gaussjord(A);`? What is the output from `Gaussjord(A) mod 2;`? Explain the difference, by performing the steps in the row reduction to echelon form one by one (by hand). What would happen if we did these same operations on a matrix with these first three columns, and some further columns consisting of zeroes and ones? What is the "bad" situation that can arise in doing a row reduction using rational arithmetic and reducing mod 2 at the end? (i.e. When will doing that lead to different results than doing all arithmetic mod 2?)

Week 1, Days 3 and 4: First Steps in Maple Programming For Coding Theory

Note: All of the Maple data structures and commands used here are also illustrated in the worksheet `Primer.mws` *in the class folder. You may also want to refer to that as you are working through the following examples and exercises. You can copy that worksheet and open it in Maple along with your lab assignment worksheet.*

Maple has a large collection of built-in commands for performing various mathematical computations, but unfortunately it does not contain any built-in functions for coding theory operations. However, it does contain a *programming language* that you can use to code and implement your own new commands or procedures extending the basic functionality of the program. (This is an imperative or procedural language similar in spirit and syntax to Pascal and, to a lesser degree, C. It does not incorporate objects, etc. as in C++.) In these lab meetings, we will work through the development of Maple procedures for several of the coding theory operations we have been discussing, leading up to a "Challenge problem" to develop a procedure for computing the SDA or coset leader table used in syndrome decoding (for small examples, of course!) As we go along, we will start to develop our own "library" or "package" of Maple procedures for coding theory and add to it as we go along over the next two weeks.

Open a text editor window, type in the following lines, and save to your diskette under the file name `CTL.map`:

```
with(linalg):

VMP := proc(v::vector,A::array,p::integer)
#
# computes the product of the vector v and the
# matrix A (in order v.A) modulo p
#
local P;
P:=map(x->Normal(x) mod p,multiply(v,A));
RETURN(eval(P))
end:
```

This is a first example of a Maple procedure. It takes a (row) vector v, a matrix A, and returns the product $P = vA$, modulo the prime integer p. In Maple, execute the command

```
read "a:/CTL.map";
```

to read in the code for this new procedure. Then to use the procedure to compute the product of w and B mod 2 for some vector w and matrix B, you would enter a command like this:

```
VMP(w,B,2);
```

The values w, B, and 2 are "filled in" for the parameters v, A, and p in the procedure heading, the product mod 2 is computed as we discussed earlier, and the result is returned and displayed as the output from executing the procedure call. The command

```
Prod:=VMP(w,B,2);
```

computes the product and assigns it to `Prod`. Test the procedure on several inputs and check that its output is correct.
The general format of a Maple procedure is as in this first example:

- The first line is the *procedure heading*, which always has the form

  ```
  procedurename:=proc(parameterlist)
  ```

  The parameter list is the list of "inputs" (information the procedure needs to do its job). The parameters are separated by commas in this list. Each parameter can be declared by type in the heading as we did in this example, but this is optional. (In general, it is a good idea to do this because it helps to catch errors.)
- The lines starting with # characters are *comment lines.* They are not processed or executed by Maple as part of the procedure. They are there to remind humans reading the procedure what it does and how.
- Then comes a list of local and global variable declarations. Local variables are known only within the procedure and their values are not saved once the procedure is finished, so they do not interfere with other variables in your Maple session. If a variable is not declared as local, it is assumed to be "global" (i.e. known in the current Maple session). This can cause problems if you have inadvertently used the same name more than once.
- Following the local and global variable declarations comes the procedure *body.* This can be any list of Maple commands. One or more of these commands can be RETURN statements which specify the output of the procedure.

Now let's build up other procedures for operations such as computing the minimum distance of a code. First add the following two procedures to your `CTL.map` file. The first of these computes the row-reduced echelon form for the linear code generated by a given set of vectors. Note that this determines the dimension, k of a code.

```
GenMat := proc(G::array,p::integer)
#
# computes the row-reduced echelon form generator
# matrix for the code spanned by a set of vectors modulo p
#
local RREF,k;
RREF:=Gaussjord(G) mod p;
k:=rank(RREF);
RETURN(eval(submatrix(RREF,1..k,1..coldim(RREF))))
end:
```

The following procedure computes the *Hamming weight* of a vector. It also illustrates a new operation on lists (`nops`), the basic Maple `for` loop, used for repeating an operation a specified number of times, and the `if` statement, used for selecting from two or more alternatives based on the values stored in variables.

```
HamWeight:=proc(w::vector)
#
# computes the Hamming weight of a vector
#
local d,i,wt;
d:=vectdim(w);
wt:=0;
for i to d do
  if evalb(w[i] <> 0) then wt:=wt+1; end if;
end do;
RETURN(wt)
end:
```

The line-by-line breakdown of what this is doing is as follows:

1) The line `d:=vectdim(w);` computes the number of entries in the vector w.
2) The line `wt:=0;` initializes the variable `wt` to zero. (What we will do next is go through the entries in the vector one by one and increase `wt` by 1 every time we find a nonzero entry.)
3) The next three lines can be thought of as a single Maple command, incorporating repetition and selection:

   ```
   for i to d do if evalb(w[i] <> 0) then wt:=wt+1; end if; end do;
   ```

   (the formatting used above is optional; I used it for readability).
4) The `for i to d do <stuff> end do;` is a basic loop structure. The variable `i` is called the *counter* for the loop and is used to control how many times the body of the loop (the <stuff>) is repeated. The `end do;` at the end is a marker or separator for the end of the loop body. With this form, `i` will start at 1, the body will be executed once, `i` will be increased by 1 to 2, the body will be executed again, etc. until `i` reaches `d`. If you wanted to start at some other value of the counter, the more general form

   ```
   for i from <start> to <finish> do <stuff> end do;
   ```

5) The body of a loop can be any sequence of Maple commands. Here the body of our loop consists of the command `if w[i] <> 0 then wt:=wt+1; fi;`. The general form of the `if` statement is this:

   ```
   if <condition> then <action1> else <action2> end if;
   ```

   If the condition is true (at the time that statement is reached in the execution of the procedure), then `action1` is performed once; if it is false then `action2` is performed, also once only. As in our example, the `else` block can be omitted. The effect is to *do nothing* if the condition is false. Here we add one to the weight if `w[i]` is not equal to zero, and do nothing if `w[i]` is equal to zero. The `evalb` means to evaluate the expression inside to a Boolean (true or false).
6) The final value of `wt` is returned as the output of the procedure.

Test your copy of the procedure on several inputs and check that its output is correct.

Now we have almost all of the ingredients to put together a procedure to compute the *minimum weight* of the code generated by a given set of vectors. For codes with no extra structure, we will need to use a direct, "brute force" approach. The idea will be to generate all the nonzero codewords one-by-one in a systematic way, compute the Hamming weight of each, and remember the smallest weight we have seen at all times. To generate all the codewords, we will take a row-reduced echelon form generator matrix G, and multiply on the left by all nonzero vectors in $K^k = \{0,1\}^k$, or $\mathbf{F}_p^k$ more generally (k = dimension).

To list all the nonzero vectors in $\mathbf{F}_p^k$, we will use a "trick." The vectors we want are in one-to-one correspondence with the integers $1, 2, 3, \ldots, p^k - 1$. One such correspondence is given by taking the integer and *computing the vector of digits in its base p representation.* (For instance, if $p = 2$, and $k = 4$, $n = 13$ corresponds to $[1, 1, 0, 1]$, since $13 = (1101)_2$). Maple has a built-in function that takes an integer and computes almost what we want: `convert(i,base,p)` gives the base p digits, but listed with the 1's digit first, then the p's digit, then the p^2's digit, etc. and only up to the last nonzero digit. We need to make a vector from these, "filled out" with zeroes at the start, and listed in reverse order(!) The two assignments to `pw` and `w` do this in the procedure below.

`ell` is the length of the list `pw`, computed using the `nops` function. `nops` of a list is the "number of operands" or number of entries in the list. The next new Maple command we are using there is the `seq` command, which builds up terms in a sequence. For example, `seq(0,j=1..k-ell)` makes a string of $k - ell$ zeroes, separated by commas. The `seq(pw[ell+1-j],j=1..ell)` lists the digits from the base p form of i in reverse order (when $j = 1$, we get `pw[ell]`, when $j = 2$, we get `pw[ell-1]`, and so forth up to `pw[1]` when $j = ell$).

The variable `mwt` will always have the smallest weight seen so far. It is initialized to the value of n (the block length) before we get into the loop. For each codeword we generate, we call the `HamWeight` procedure to find its Hamming weight. If the weight of that codeword is less than the smallest weight we have seen so far, then the smallest weight is updated accordingly.

Here is the finished procedure:

```
MinWeight:=proc(G::array,p::integer)
#
# computes the minimum weight for the code spanned by
# the rows of G modulo p (over F_p)
#
local GM,k,i,j,pw,w,cw,wt,mwt,ell;
GM:=GenMat(G,p);
k:=rank(GM);
mwt:=coldim(GM);
for i from 1 to p^k-1 do
  pw:=convert(i,base,p);
  ell:=nops(pw);
  w:=vector([seq(0,j=1..k-ell),seq(pw[ell+1-j],j=1..ell)]);
```

```
  cw:=VMP(w,GM,p);
  wt:=HamWeight(cw);
  if wt < mwt then mwt:= wt; end if;
end do;
RETURN(mwt);
end:
```

Note how the body of this procedure calls the other procedures `GenMat, VMP,HamWeight` we have worked on. That's OK as long as those procedures are "known" in the Maple session where you use this procedure. That will be true as long as you have read in your file `CTL.map` in the current session.

Assignment 2, part b

4) Following the examples above for the Maple syntax, develop your own Maple procedure to compute a parity check matrix for a code code C generated by the rows of an input matrix G mod p. (This one should not require any loops or selections.)

5) Following the examples above for the Maple syntax, develop your own Maple procedure to compute a row-reduced echelon form generator matrix for the *dual code* $C^{\perp}$ of a code C generated by the rows of an input matrix G mod p. (This one should not require any loops or selections either. Try to follow what you did "by hand" in part 2 of this assignment.)

6) Following the examples above, develop your own Maple procedure to compute a list of *all codewords* in the code generated by the rows of an input matrix G mod p. (Hint: Reuse as much of *MinWeight* as possible!) See the worksheet `Primer.mws` or the on-line help for info about building up lists one element at a time. Of course you won't want to use this on large codes (for instance binary codes with $k > 7, 8$ or so(!)).

7) (*Challenge*) Develop a Maple procedure that computes the SDA (standard decoding array, or list of coset leaders) used in syndrome decoding a linear code with given generator matrix. Your input should be a matrix G mod p whose rows generate the code. The output should be a list of lists, one for each coset, giving the the syndrome for that coset and minimum weight elements of that coset. (Hint: You can start out by taking G and row-reducing to echelon form as in `MinWeight` above. Then the set of vectors with a given syndrome is the set of solutions of a system of linear equations – a coset of the nullspace of the parity check matrix. You can list all the elements in the coset by the same process you used for part 6 above!)

Hand in the listing of your procedures in your `CTL.map` file, and a worksheet showing output from several tests of each.

Week 2, Day 1 – The Golay Code

In the discussions for this week, we saw an alternate construction of the binary Golay (24,12,8) code.

- We start from H, the $(7,4,3)$ Hamming code with generator matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

  and let K be the code obtained from H by *reversing the order of the bits* in each codeword. K is also a $(7,4,3)$ linear code, and $H \cap K = \{0000000, 1111111\}$.
- Let H' and K' be the $(8,4,4)$ extended codes obtained from H and K by adding a parity check bit to each codeword at the right end (making the number of 1's even in each case).
- Consider the code C obtained by forming all words (length 24) of the form $(a+x|b+x|a+b+x)$ where $a, b \in H'$ and $x \in K'$. (Note that this can be seen as a "souped up" version of the "$(u|u+v)$ construction" we saw last week.)

Assignment 3, part a

1) Using the built-in Maple linear algebra procedures, and the Maple coding theory procedures we worked out last week, construct a generator matrix for the code C described above. (Note: it is not enough just to use the rows of the generator matrices for H', K' in the $(a+x|b+x|a+b+x)$ vectors. Be sure you allow a, b to be the zero vector also.)
2) Verify the dimension and minimum distance of C using our procedures.
3) The *weight enumerator* of a code is the vector

$$(w_0, w_1, w_2, \ldots, w_n)$$

   in which w_i is *the number of words in the code with Hamming weight* i. The weight enumerator can also be "packaged" as a polynomial with the w_i as coefficients:

$$W(s) = w_0 + w_1 s + w_2 s^2 + \cdots + w_n s^n$$

   (same idea as in the *generating function* of a sequence as studied in combinatorics). Write a Maple procedure that computes the weight enumerator of a code generated by the rows of a given matrix, mod p and add it to your `CTL.map` file. Test your procedure on the codes H, K, H', K' above. Also test it on the code C described above. Your output should be:

$$[1, 0, 0, 0, 0, 0, 0, 0, 759, 0, 0, 0, 2576, 0, 0, 0, 759, 0, 0, 0, 0, 0, 0, 0, 1]$$

(that is, C has 1 word of weight zero, 759 words of weight 8, 2576 words of weight 12, 759 words of weight 16, and one word of weight 24).

As we know from Exercise 3.7.9 in the text, the code generated by the matrix given in our text has the same weight enumerator as C! A theorem of Pless shows that every binary code with parameters $(n, k, d) = (24, 12, 8)$ is *equivalent* to the Golay code C in the following sense: *Two binary block codes of length n are said to be equivalent if there is some fixed permutation of the entries of K^n that takes the first code to the second.* Finding the permutation that sets up the equivalence in this case is not a feasible problem if we use "brute force", though (24! is a large number!).

Week 2, Day 2,3 – Cyclic Codes

In class we have begun to study cyclic codes – codes that are taken to themselves by cyclic permutations of the coordinates. Polynomials over the binary field are an important tool in the study of cyclic codes, so we will need to begin by adding some basic utilities to our `CTL.map` files to convert back and forth between vector and polynomial representations of codewords, and to build a generator matrix for a cyclic code given its generator polynomial. We will use these building blocks to look at some examples of cyclic codes, and their duals.

Assignment 3, part b

4) Write a Maple procedure called `ToPoly` that takes as input a vector of length n representing a codeword, and the name of a variable (type `symbol`), and computes the corresponding polynomial of degree $\leq n-1$ mod p in the variable given. Test it on several examples.
5) Now, go the other way – Write a Maple procedure called ToVector that takes as input a polynomial of degree $\leq n-1$, the name of the variable, the integer n, and p, and computes the corresponding polynomial of degree $\leq n-1$. Technical note: There is a built-in Maple function called `coeff` that will be useful for picking off the coefficients. Check this out in the online help. Unfortunately, `coeff(p,1)` will produce an error. So you need to do something else to pick off the constant term. One way is to use the `subs` function to subsitute a value into a polynomial. After you pick off each coefficient, put it through `Normal( ) mod p` to put it in standard form before filling in the corresponding entry in the vector. Test your procedure on several examples, including cases where the constant term is zero, where the polynomial has degree less than $n-1$, and so forth.
6) The following procedure computes a generator matrix for the cyclic code with generator polyomial g, provided that g is a factor of $x^n - 1$ (the necessary condition we discussed in class).

```
CyclicGenMat:=proc(g::polynom,var::symbol,n::integer,p::integer,
                   reduce::boolean)
#
```

```
# finds generator matrix for the cyclic
# code of block length n with generator polynomial g, if
# one exists -- necessary condition is that g must divide x^n - 1
#
local k,i,mat;

if Rem(var^n-1,g,var) mod p = 0 then
  k:=degree(g);
  mat:=ToVector(g,var,n,p);
  for i to n-k-1 do
    mat:=stackmatrix(mat,ToVector(var^i*g,var,n,p));
  end do;
  if reduce then mat:=GenMat(mat,p); end if;
  RETURN(eval(mat))
else
  lprint("Error -- generator polynomial must divide", var^n - 1)
end if
end:
```

Enter this into your `CTL.map` file. Note: The procedure checks first to see whether $g(x)$ is a factor of $x^n - 1$, then builds the generator matrix up one row at a time by finding the coefficient vectors of the polynomials $x^i g(x)$ for $i = 0, \dots n - d - 1$, where d is the degree of g (the dimension of the cyclic code is $k = n - d$). These rows are shifts of the coefficient vector for $g(x)$. The parameter `reduce` is a Boolean (true or false) "switch" that determines whether the generator matrix will be row-reduced to echelon form or not.

7) Using your procedures, find generator and parity check matrices for the binary cyclic codes of given length n with the given generator polynomials:
   1) $n = 15$, $g(x) = 1 + x + x^4$ What is the minimum distance for this code?
   2) $n = 23$, $g(x) = 1 + x + x^5 + x^6 + x^7 + x^9 + x^1 1$ Also find the weight distribution for this code. What code is this?
   3) $n = 15$, $g(x) = 1 + x^4 + x^6 + x^7 + x^8$. What is the minimum distance of this code?
8) We saw in Discussion that the dual of a cyclic code is also cyclic. If $g(x)$ is the generator polynomial of C, then we have an equation $x^n - 1 = g(x)h(x)$ for some polynomial h. If $\dim(C) = k$, then $\deg(g(x)) = n - k$, and $\deg(h(x)) = k$. Hence, we have

$$x^n - 1 = g(x)h(x) \Rightarrow$$
$$(x^{-1})^n - 1 = g(x^{-1})h(x^{-1}) \Rightarrow$$
$$x^n - 1 = -x^n((x^{-1})^n - 1) = \left(x^{n-k}g(x^{-1})\right)\left(-x^k h(x^{-1})\right)$$

This argument shows that $-x^k h(x^{-1})$ gives a generator polynomial for the dual code. (Why?) Use this observation to write a Maple procedure that computes the generator polynomial of the dual $C^\perp$, given as input a generator polynomial for $g(x)$. The Maple `Quo` function will be useful here. Also note that the commands

```
d:=degree(p,x);
expand(x^d*subs(x=1/x,p));
```

can be used to compute the polynomial $x^d p(x^{-1})$ where d is the degree of p. Use your procedure to find the generator polynomials of the duals of the codes in question 7.

9) The built-in Maple function `factor` has an inert form that allows factoring mod p for a prime – format `Factor(f) mod p`. Use this command to factor $x^n + 1$ mod 2 for $n \leq 49$ odd. How many different cyclic codes of each of these block lengths are there?

Week 2, Day 4 – Finite Fields in Maple

We have now introduced the construction of the general finite (Galois) field $GF(2^r)$ starting from an irreducible polynomial $p(x)$ of degree r in $K[x]$. Maple allows us to compute with these fields by exactly the same method we have used for our hand calculations in the discussion for today. For example suppose we want to work with the field with 8 elements, $GF(2^3)$. There are two different irreducible polynomials of degree 3 in $K[x]$, $x^3 + x + 1$ and $x^3 + x^2 + 1$. Either one can be used and the results will be equivalent, so let's use the first. If a is a fixed root of this polynomial, then every element of the field $GF(2^3)$ will have the form $c_0 + c_1 a + c_2 a^2$ for some $c_0, c_1, c_2 \in K = \{0, 1\}$. We will call this the *standard representation* of the element.

To do all of this in Maple, we begin with the command

```
alias(aa=RootOf(z^3+z+1));
```

which specifies that from now on in the current session, `aa` will refer to a (fixed) root of our irreducible polynomial. (Note: to avoid potential conflicts, I used a different name for the variable in the polynomial in the `RootOf`.) Then we can do arithmetic in the field $GF(2^3)$ by using the ordinary $+, -, */$ operators, but placing the expression inside `Normal( ) mod 2` to reduce to a standard representation of the element. For example, try entering the following:

```
Normal((aa+1) + (aa^2+1)) mod 2;
Normal((aa+1) * (aa^2+1)) mod 2;
Normal(1/(aa^2+1)) mod 2;
```

and check the results by hand using remainders on division by the minimal polynomial of `aa`.

The elements of our finite fields $GF(2^r)$ can now appear in expressions, as entries in vectors or matrices, as coefficients in polynomials, etc. In fact, many of the procedures in your `CTL.map` library should work even on matrices and vectors with entries in $GF(2^r)$. The ones that *do not* are the ones that used our "trick" for enumerating the elements of $\mathbf{F}_p^k$ by using the base p expansions of the integers $0, \ldots, p^k - 1$.

Assignment 3, part c

10) Using the `Factor( ) mod p` command, determine all irreducible polynomials in $K[x]$ of degrees $r = 2, 3, 4, 5, 6, 7, 8$. (How?) (In question C of today's Discussion, you will see a way to prove that there are irreducible polynomials of every degree in $K[x]$.)

11)

a) Using Maple, compute 8×8 matrices giving the addition and multiplication tables in $GF(2^3)$, constructed using the irreducible polynomial $h_1(x) = x^3 + x + 1$. (You'll need to index the rows and columns of the matrices by the elements of the field in some specific way. Say how you are doing it.) Identify the primitive elements by the "brute force" method of determining the multiplicative order of each nonzero element. (You'll see a nice way to understand how this works in Exercise 5.1.18 from today's discussion.)

b) Next, construct 3×3 matrices giving the addition and multiplication tables in $GF(2^3)$ constructed using the different irreducible polynomial $h_2(x) = x^3+x^2+1$. Identify the primitive elements.

c) By examining the results of parts a and b, find a one-to-one correspondence between the elements in your two ways of writing $GF(2^3)$ that preserve the sum and product operations in the two fields (such a mapping is called an *isomorphism of fields*. It is a general fact that any choice of irreducible $h(x)$ of degree r leads to isomorphic fields $GF(2^r)$, so we do not indicate which polynomial was used in the notation.

12) Recall from class that if n is prime to p, the *p-cyclotomic coset* of $s \in \{0,\ldots,n-1\}$ is the set

$$C_s = \{p^j s \bmod n : j = 0, 1, \ldots\}$$

Since p is prime to n, some power of p will be congruent to 1 mod n, and the elements will start to repeat so we can stop. Write a Maple procedure called `CycCoset` that computes the p-cyclotomic coset of a given s mod n and add it to your `CTL.map` file. You will probably want to use the `while` loop structure for this, since the number of elements of the cyclotomic coset depends on s and will not be known when your procedure starts. The general format of the `while` loop is this: `while <condition> do <action> od;`. The condition is tested when the loop is reached. If it is true the action is done once, and the condition is tested again. If it is true, the action is performed again. The loop terminates the first time the action causes the condition to become false. The action is not performed at all if the condition is false the first time it is tested. See the online Help, and the `primer.mws` worksheet for more details. Test your procedure thoroughly.

13) Using your `CycCoset` procedure, write a procedure `MinPoly` that returns the `minimal polynomial` of the element α^s, where α is a primitive element of the field $GF(p^r)$ (a root of a particular irreducible polynomial of degree r defining the field).

Week 3, Day 1 – BCH Codes

The BCH (Bose-Chaudhuri-Hocquenghem) codes are cyclic codes constructed using the algebra of the fields $GF(p^r)$. Their main interesting feature is that we can *design* their minimum distance to be as large as we like by (picking n sufficiently big and) specifying the form of the generator polynomial. In the lab today, we will develop a Maple procedure to specify the general BCH codes from Question C in today's Discussion. Here's the idea once again:

1) Let β be a primitive element of the field $GF(p^r)$ (as always, constructed using some particular irreducible polynomial of degree r with β as a root). Usually we take $p = 2$, but the construction works in essentially the same way for any prime p.
2) We can define the general BCH code as follows. Given any $\delta \geq 1$, let $BCH(p^r, \delta)$ be the code with $n = p^r - 1$ consisting of polynomials $c(x)$ (degree $\leq p^r - 2$) such that

$$c(\beta) = c(\beta^2) = \cdots = c(\beta^{\delta-1}) = 0.$$

3) So we take the generator polynomial of $g(x)$ to be the lcm of the minimal polynomials of $\beta, \beta^2, \ldots, \beta^{\delta-1}$ in $\mathbf{F}_p[x]$. (Note: the coefficients of $g(x)$ are all in $\mathbf{F}_p$.)
4) With $p = 2$, we can say more. Since the 2-cyclotomic coset of an odd integer j contains $2j$, when we work over the binary field, it is enough to take the least common multiple of the minimal polynomials of $\beta, \beta^3, \ldots, \beta^o$, where o is the last odd number less than or equal to $\delta - 1$. This does not work if $p \neq 2$, however.
4) The minimum distance of $BCH(p^r, \delta)$ is *at least* δ. This is called the "designed distance" of the BCH code. For binary BCH codes, if δ is odd then $d \geq \delta$ and if δ is even $d \geq \delta + 1$, since the even number δ is also in the union of the cyclotomic cosets of the powers of β that we know are roots of $g(x)$.

Assignment 4, part a

1) Write a Maple procedure that implements the above method to compute the generator polynomial for $BCH(p^r, \delta)$ over F_p, using some particular irreducible polynomial h of degree r to construct the field $GF(p^r)$. Test your procedure by computing the generator polynomials of the binary codes $BCH(2^4, 3)$, $BCH(2^4, 5)$, and $BCH(2^5, 7)$. In each case, determine generator matrices using our general procedures for cyclic codes from last week. Compute the weight enumerators for your codes to check that the minimum distance $d \geq \delta$. The last computation will take a while by our "brute force" method for finding the weight enumerator. Do you see why?
2) When we form the lcm of the minimal polynomials of $\beta, \beta^2, \ldots, \beta^{\delta-1}$ in $K[x]$ the roots of the resulting polynomial are β^ℓ where ℓ is any element of $C_1 \cup C_2 \cup C_3 \cup \cdots \cup C_\delta$, where C_j is the p-cyclotomic coset of j mod $n = 2^r - 1$. This union of cyclotomic cosets can have strings of consecutive elements of $\{1, 2, \ldots, n-1\}$ longer than the string $\{1, 2, \ldots, \delta - 1\}$. Using Maple, find an explicit binary case where this happens. (Hint: there's an interesting one – one where the minimum distance is quite a bit larger than expected – with $n = 31$.)
3) In all of your examples in question 2, the weight enumerator is (i.e. should be!) symmetric about $(n-1)/2$ – that is: if there are A_ℓ words of weight ℓ, then $A_{n-\ell} = A_\ell$. Why is this true? Will it always happen for binary BCH codes? (Find a proof why this has to happen for binary BCH codes or a counterexample where it fails.)

Week 3, Day 2 – Reed Solomon Codes

We now come to the Reed-Solomon codes. While the BCH codes are codes over the prime field $\mathbf{F}_p$ constructed using the algebra of $GF(p^r)$, the Reed-Solomon codes are actually vector subspaces of a vector space over the larger field $GF(p^r)$. The entries of Reed-Solomon codewords can be any elements of the larger field. However, these codes are also cyclic, so they can be defined by generator polynomials. We will use the code $RS(p^r, \delta)$ defined as the cyclic code over the field $GF(p^r) = \mathbf{F}_{p^r}$ with generator polynomial

$$g(x) = (x - \beta^{m+1})(x - \beta^{m+2}) \cdots (x - \beta^{m+\delta-1}).$$

We have $n = p^r - 1$, $k = p^r - \delta$ (but careful, this is the dimension as a vector space over the field $\mathbf{F}_{p^r}$), $d = \delta$. One important property of the Reed-Solomon codes is that they all reach the Singleton bound from Week 1: $d = n - k + 1$. In other words, they have the maximum possible d for the given n and k(!).

Assignment 4, part b

4) Write a procedure that takes inputs p, r, δ, m, a particular irreducible polynomial of degree r over $\mathbf{F}_p$ to construct $GF(p^r)$, and the name of a variable, and computes the generator polynomial for $RS(p^r, \delta)$. This will be a polynomial with coefficients in $GF(p^r)$. Use your procedure to do Exercise 6.2.9 parts c,d,e from the text.
5) Reed-Solomon codes over $GF(2^r)$ are often used in the following way. Each component of the Reed-Solomon codewords is an element of $GF(2^r)$. We use the polynomial $h(\beta)$ of degree r to construct this field, so every element is uniquely represented by a remainder modulo h, i.e. by a linear combination of the form $c_0 + c_1\beta + \cdots + c_{r-1}\beta^{r-1}$, where $c_i \in K = \{0, 1\}$. Picking off the coefficients to make a vector $(c_0, \ldots, c_{r-1})$, we get from each codeword of the RS code a *binary word* of length $n' = r(2^r - 1)$. We call this the *binary image of the Reed-Solomon code*. It is the binary words from the binary image that are often transmitted or stored when Reed-Solomon codes are employed in practice. (One reason for this is that such codes have good *burst-error correction* capabilities. A long string of consecutive bit errors may affect only a relatively small number of entries considered as elements of $GF(2^r)$. Given a generator polynomial for a Reed-Solomon code over $GF(2^r)$, write a Maple procedure that produces a corresponding binary generator matrix for the binary image code. You may want to start first by constructing a conversion procedure that takes an element of $GF(2^r)$ and finds the corresponding binary vector of length r.
6) Given the Reed-Solomon code $C = RS(2^r, \delta)$, we can construct other codes that reach the Singleton bound by an operation called *shortening*. (Codes of this type are used, for instance, in the encoding of audio information on CD's(!)) Let s be an integer. The shortened Reed-Solomon code $C(s)$ is obtained by

   - Intersecting C with the vector subspace of $GF(2^r)^{2^r-1}$ consisting of vectors with zeros in the last s components, *then*
   - Deleting the last s components in the resulting codewords.

   There is a discussion starting at the bottom of page 133 in the text showing how to obtain a generator matrix for the shortened code. (Shortened codes are usually not cyclic so they will not have generator polynomials.) Implement this method in a Maple procedure. Use it to construct a generator matrix for the shortened code $C(4)$

for $C = RS(2^4, 5)$. Use the polynomial $h = x^4 + x + 1$ to construct $GF(2^4)$. What are the parameters n, k, d for the shortened code?

Week 3, Days 3,4 – Encoding and Decoding Reed-Solomon Codes

So far, we have concentrated on the *construction of codes with desirable parameters* n, k, d. In practice, in order for a code with good parameters to be attractive for most applications, it must also be true that there are efficient encoding and decoding algorithms available. Reed-Solomon codes meet both of these criteria.

Assignment 4, part c

7) (Encoding) In question A of the last section of lab problems, you wrote a Maple procedure to compute the generator polynomial for $RS(p^r, \delta)$ (with $m = 0$). Now we will use that generator polynomial to produce a very "compact" encoding function for $RS(p^r, \delta)$. The RS code has $n = p^r - 1, k = p^r - \delta, d = \delta$. We place the information symbols $c_1, \ldots, c_{p^r-\delta}$ in the coefficients of the *high-degree terms* in a polynomial:

$$f = c_1 x^{p^r-2} + c_2 x^{p^r-3} + \cdots + c_{p^r-\delta} x^{\delta-1}$$

The generator polynomial g has degree $\delta - 1$. Hence if we use polynomial division to write $f = qg + r$, the remainder will have degree $\delta - 2$ or less. Moreover, the difference $f - r$ is a multiple of g, so it is a codeword. Hence our encoding function is $E(f) = f - r$, where r is the remainder on division by g. Note that, as would be the case for a the vector-matrix product encoder with a generator matrix in the row-reduced echelon form $[I|B]$, $E(f)$ contains the information symbols in the high-degree terms. Encoders with this property are called *systematic encoders*. Write a Maple procedure that implements this process. Build up the polynomial f one term at a time from the information symbols c_j, $j = 1, \ldots, p^r - \delta$. The built-in function `Rem` can be used to compute the desired remainder mod p.

8) For the Berlekamp-Massey decoder for the $RS(p^r, \delta)$ code for a given m, for each received word w we need first to compute the *syndromes* $s_j = w(\beta^j)$, $j = m + 1, \ldots, m + \delta - 1$ (using the polynomial form of the codeword). Then these are used as coefficients in the "syndrome polynomial":

$$s(x) = 1 + s_{m+1}x + \cdots + s_{m+2t}x^{2t}$$

(where $2t = \delta - 1$). Write and test a Maple procedure that takes as parameters a received word (any word of length $n = p^r - 1$), the integers δ, m, p, and the irreducible polynomial used to construct $GF(p^r)$, and returns the corresponding syndrome polynomial.

9) In the class folder, you will find files containing Maple source code for two additional procedures implementing the subsequent steps of the Berlekamp-Massey algorithm. The first, called `RSBMErrorLoc`, takes a syndrome polynomial and applies the main loop in the Berlekamp-Massey algorithm to find the "reversed" *error locator polynomial*, $\sigma_R(x)$. The second, called `RSBMErrorVal` applies the "Forney formula" to

compute the error values, if the error pattern is correctable. *It returns the vector that should be added to the received word to obtain the nearest codeword*, or the value FAIL if the error has too large weight. The procedure headings for these two procedures look like this:

```
RSBMErrorLoc:=proc(syndromepoly::polynom,delta::integer,
                   irred::polynom, p::integer)
```

and

```
RSBMErrVal:=proc(errloc::polynom,synpoly::polynom,m::integer,
                 delta::integer,irred::polynom,p::integer)
```

Copy and paste these procedures into your `CTL.map` file. Write a Maple procedure `RSBMDecode` that takes a received word, the integers p, m, δ and the irreducible polynomial used to construct $GF(p^r)$, and decodes the received word. Test it by encoding several different information strings to obtain codewords for the $RS(2^4, 3)$, $RS(2^4, 5)$, and $RS(2^4, 9)$ codes. Add different error vectors to obtain received words, then apply your decoding procedure to see that the correct codeword is recovered. Also see what happens if you add on an error vector that has too large weight.