Math 25 and Maple

This is a brief document describing how Maple can help you avoid some of the more tedious tasks involved in your Math 25 homework. It is by no means a comprehensive introduction to using Maple.

1 Basic Commands

The command prompt in Maple is the greater than symbol: `>`. Everything entered by you will follow one of these prompts. You'll never have to actually type in the command prompt: Maple does that part for you. I just don't want anyone to be confused by it. After typing in a command, to end and execute it, type a semicolon (`;`) and hit "enter". For example

```
3 + 4;
```

followed by "enter" will ask Maple to add 3 to 4, producing `7` on the following line.

Addition, subtraction and division all use the usual symbols: `+`,`-`, `/`. Multiplication is done using the asterisk: `*`. Maple *does not* interpret concatenation as multiplication. So, to multiply 3 by 5 you need to enter `3 * 5` and *not*, say, `(3)(5)`. Exponentiation is achieved by using the caret: `^`. So, for example, to compute 18 multiplied by 2 to the seventh power, you type

```
18*2^7;
```

followed by "enter". Overuse of parentheses is never a bad idea when you aren't sure how Maple might choose to associate your operations. Among other things, this is especially important when using negative numbers. For example, the correct way to ask Maple to multiply 4 by negative 53 is by typing

```
4*(-53);
```

and hitting "enter". If you leave off the parentheses in this example, Maple will complain.

Okay, now for a few more interesting (and perhaps less obvious) commands. To assign values to variables you use the assignment operator `:=`. So, create a variable named x and set it equal to 8, you type

```
x := 8;
```

end hit "enter". To create a variable name *george* and give it the value of x, you'd type

```
george := x;
```

and hit "enter".

Fortunately for us, Maple understands modular arithmetic. In fact, if you ask Maple to perform a computation to a certain modulus it *always* returns the least nonnegative residue to that modulus. Modular arithmetic is input to Maple the same way it's written on paper, but with *no parentheses* around the word "mod". As an example

```
47*(-12) mod 8;
```

followed by "enter" will produce the result 4, which is the least nonnegative residue of the product indicated. One very nice thing is that Maple does modular exponentiation, even with negative exponents. What this means is that Maple knows how to compute modular inverses. For example

```
7^(-1) mod 9;
```

followed by "enter" will produce 4, the least nonnegative residue of any modular inverse of 7 mod 9.

You might also find it useful to know how to input matrices. Before you do anything with matrices, type in

```
with(LinearAlgebra):
```

and hit "enter". This may not be totally necessary, but it will guarantee that everything I have to say about matrices is correct. To input the matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

into Maple, you would type

```
Matrix([[a,b],[c,d]]);
```

and hit "enter". If you want to access this matrix frequently, you may want to assign it to a variable, as described above. For example, to call this matrix M you would type

```
M := Matrix([[a,b],[c,d]]);
```

and hit "enter". To compute determinants, you can use the function `Determinant`. This is rather long to type, so one thing you can do is assign this function to a variable with a shorter name. For example, to call it just *det*, type

```
det := Determinant;
```

and hit "enter". If you do this, then the command

```
det(M);
```

followed by ”enter” will produce the result $ad-bc$, as expected. If you've defined two matrices, say A and B, in Maple, and they have compatible dimensions, then you can multiply them using the command `Multiply` as follows

```
Multiply(A,B);
```

Sadly, Maple does not seem to understand how to do modular arithmetic with matrices, at least not with the `mod` command described earlier. What this means for you is that if you need to reduce a matrix to a certain modulus you'll have to do it entry by entry.

2 Cryptographic Tools

Now let me describe a few bits of code (Maple calls them *procedures*) I've written in Maple to implement some of the encryption methods we've learned about in class. Since they aren't for any sort of general use, they perform no sort of input checking. Therefore, if you accidentally give them invalid input, without warning you might get very strange output. For example, when inputting messages to be transformed, be sure you *always use only capital letters*. These procedures won't handle spaces or punctuation of any kind (at least, not in any sensible way).

In order to use these procedures, open the file `crypto.mw` in Maple. Insert the cursor in the very first line of code that you see, and keep hitting enter until you get past everything that I've written and see a blank command prompt. As long as you keep working in the same window, you will have access to all of the crypto tools.

The first procedure is called `AffineCipher`. This implements the affine shift transformation encryption method. It has three inputs: a string of only capital letters, an integer relatively prime to 26 and another integer. If these are, respectively, `S`, `a`, `b`, then a call to `AffineCipher` looks like

```
AffineCipher(S,a,b);
```

As a more concrete example, to encrypt the message MMMDONUTS with the key $a = 5$, $b = 24$, you would type

```
AffineCipher("MMMDONUTS",5,24);
```

and hit enter. This produces the encrypted message GGGNQLUPK. Notice the double quotes `"` around the message. Maple understands strings of text only when they are enclosed between pairs of double quotes, so be sure you don't forget these.

Actually, let me make a comment about strings here. These are just another type of data in Maple, and as a result they can be assigned to variables. This is convenient when you are trying to analyze a long message and you don't feel like typing it over and over again. In our above example, we could have typed

```
homer := "MMMDONUTS";
```

and hit "enter", followed by

```
AffineCipher(homer, 5, 24);
```

In this case, the *name* of the message is `homer`, but the message itself is MMMDONUTS.

The procedure `VigenereCipher` implements the Vigenère cipher. It has two inputs: the message to be encrypted and the key word used for encryption. For example, to encrypt the message BIGMOUTHSTRIKESAGAIN using the key MATH we would type

```
VigenereCipher("BIGMOUTHSTRIKESAGAIN", "MATH");
```

followed by "enter". This produces the output

```
"BIGMOUTHSTRIKESAGAIN"
"NIZTAUMOETKPWELHSABU"
```

the first line of which is our original message, printed again to indicate whether or not it was necessary to add additional X's before encryption. The second line is the encrypted message.

The procedures `HillCipher2` and `HillCipher3` implement the the Hill cipher with 2×2 and 3×3 matrices, respectively. They take as input the message to be encrypted as well as the encrypting matrix. So, to encrypt the message ITSWAYTOOLATE using the matrix

$$\begin{pmatrix} 1 & 4 \\ -1 & 3 \end{pmatrix}$$

we would proceed as follows. Just to make life easier, we attach this matrix to a variable by typing

```
A := Matrix([[1,4],[-1,3]]);
```

and hitting "enter". We then type

```
HillCipher2("ITSWAYTOOLATE", A);
```

end hit "enter". This produces the output

```
"ITSWAYTOOLATEX"
"GXCWSUXXGTYFSN"
```

The first line is just indicating that an extra X was added to the message to ensure that it could be broken into blocks of 2 before encryption. Even if no letters are added, both Hill cipher procedures reproduce the message to be encrypted. The second line of output, naturally, is the result of encryption.

`AutokeyCipher` implements the *generalized* autokey cipher discussed in class. Recall that this cipher has three parts to it's key: a letter k, and two integers

a and b in the range 0 to 25 (inclusive). For example, to encrypt the message "ICANTPUTMYFINGERONIT" with the genralized autokey using the choices $k =$ Z, $a = 5$ and $b = 7$, we would type

```
AutokeyCipher("ICANTPUTMYFINGERONIT", "Z", 5 , 7);
```

and press "enter". The result is the output HNXSCUBWBNILATLKNSRW. Keep in mind that the ordinary autokey cipher corresponds to the choice $a = 1$ and $b = 0$.

The procedure `ExpCipher` implements the exponentiation cipher. It needs to be used in conjunction with the procedures `TextToBlocks` and `BlocksToText`. These convert strings of text to blocks of integers and blocks of integers to strings of text, respectively, using the equivalence described in section 8.3 of the text. These procedures take as input the string/blocks to be converted as well as (half of) the block length. For example, to convert the message THEOUTERNATIONALIST into blocks of $2 \cdot 3 = 6$ integers you would type

```
TextToBlocks("THEOUTERNATIONALIST", 3);
```

and hit "enter", resulting in the output

```
"THEOUTERNATIONALISTXX"
[190704, 142019, 41713, 1908, 141300, 110818, 192323]
```

The first line indicates that in order to break the message into blocks of length 3, two X's had to be added. This line is printed whether or not any X's are added. The second line is a list whose entries are the corresponding blocks of integers (i.e. each block of integers corresponds to a block of 3 letters in the message).[1] To convert this list back into the original message we use the procedure `BlocksToText` as follows:

```
BlocksToText(Vector[row]([190704, 142019, 41713, 1908, 141300, 110818, 192323]), 3);
```

You can either type the `Vector[row]` part yourself, or you can copy and paste the output of the previous example: Maple inserts the additional commands itself.

[1]A word of caution: The list generated by `TextToBlocks` is an object of the type `Vector` in Maple. Consequently, if there are too many entries in this vector it *will not* be displayed correctly. For example, the command `TextToBlocks("WEDONTNEEDNOSTINKINGBADGERS",2);` yields the output

```
"WEDONTNEEDNOSTINKINGBADGERSX"
[ 1..14 Vector[row]       ]
[ Data Type: anything     ]
[ Storage: rectangular    ]
[ Order: Fortran_Order    ]
```

The `Vector` containing the blocks was indeed created, but is too long to be displayed. One way around this is to assign the output of `TextToBlocks` to a variable, such as `block := TextToBlocks("WEDONTNEEDNOSTINKINGBADGERS", 2);`. The blocks of numbers can be accessed by accessing the entries of this vector: `block[1]`, `block[2]`, `block[2]`, etc..

Finally, we are in a position to explain the use of `ExpCipher`. Suppose we want to encrypt the message ATMOSPHERE using an exponentiation cipher with the prime 2789 and the exponent 31. We first convert the message into blocks of integers using `TextToBlocks`:

```
TextToBlocks("ATMOSPHERE",2);
```

This yields

$$\text{"ATMOSPHERE"}$$
$$[19, 1214, 1815, 704, 1704]$$

Next we encrypt the resulting list of integers:

```
ExpCipher(Vector[row]([19, 1214, 1815, 704, 1704]), 2789, 31);
```

which gives the output

$$[590, 634, 633, 2313, 2356]$$

These are the encrypted blocks. Remember, with an exponentiation cipher we don't convert the output back into ordinary text because it may not be possible. To decrypt, we use `ExpCipher` the same way but with the decrypting exponent replacing the encrypting exponent.

Before I end, let me tell you about one more procedure: `FrequencyCount`. This doesn't do any encrypting, but rather makes frequency analysis problems easier by doing all of the counting for you. Specifically, it determines how often each letter occurs in a given message. To use it to count the frequency of letters appearing in MINTJULEP you would type

```
FrequencyCount("MINTJULEP");
```

which produces a list of all the letters in the alphabet and how often they occur in the message MINTJULEP. The particular output is too long to record here.

3 Further Information

Of course, Maple will do a lot more than just what I've described here, but you'll have to figure it out for yourself. I've given you enough information so that you should be able to do your Math 25 homework. For more general information the built in help file can extremely useful, but unfortunately it's missing in the student version that you can download! So if you'd like to know how to do anything else with Maple you'll have to find someone to ask.