Technology Guides for Mathematica, Maxima, and Cryptography Explorer

With an Emphasis on Cryptographic Applications

Dr. Don Spickler

Department of Mathematics & Computer Science
Henson School of Science and Technology
Salisbury University
July 26, 2019

About this Document

This document is designed to be a technology supplement for a first course in cryptography aimed at the third or fourth year undergraduate mathematics major. The packages covered are Mathematica, Maxima, and Cryptography Explorer. It is not intended to be a general reference for Mathematica or Maxima, just a quick guide to calculations found in cryptographic applications.

When I first wrote the Cryptography Explorer program, I wanted to create a package that would take most of the tediousness out of classical cryptographic methods while still leaving the decisions up to the user. As the program has grown, I have tried to keep with that philosophy, although there are some additions that probably do too much for the user. I have also never felt that a single software package was good enough for all applications. This is why I have also incorporated both Mathematica and Maxima into these notes. These are computer algebra systems that have much more computational abilities and power than does the Cryptography Explorer program. For classical cryptographic methods, where manipulation is more prevalent than calculation, you will probably find the Cryptography Explorer program easier to use. When it comes to modern techniques, where there is more calculation than there is manipulation, a computer algebra system will probably be easier to use. Although you still may find the Cryptography Explorer program helpful in converting text to and from the numeric type input necessary for the modern techniques.

Most importantly, learn, experiment, and enjoy.

Don Spickler
2015

Mathematica

Mathematica is a commercial computer algebra system, the following description was taken from the Wolfram site (http://www.wolfram.com/).

> For more than 25 years, Mathematica has defined the state of the art in technical computingand provided the principal computation environment for millions of innovators, educators, students, and others around the world.
>
> Widely admired for both its technical prowess and elegant ease of use, Mathematica provides a single integrated, continually expanding system that covers the breadth and depth of technical computingand with Mathematica Online, it is now seamlessly available in the cloud through any web browser, as well as natively on all modern desktop systems.

You can purchase Mathematica from the Wolfram site,

http://www.wolfram.com/

Maxima

Maxima is an open-source computer algebra system, the following description was taken from the Maxima project site at sourceforge (http://maxima.sourceforge.net/).

> Maxima is a system for the manipulation of symbolic and numerical expressions, including differentiation, integration, Taylor series, Laplace transforms, ordinary differential equations, systems of linear equations, polynomials, sets, lists, vectors, matrices and tensors. Maxima yields high precision numerical results by using exact fractions, arbitrary-precision integers and variable-precision floating-point numbers. Maxima can plot functions and data in two and three dimensions.
>
> The Maxima source code can be compiled on many systems, including Windows, Linux, and MacOS X. The source code for all systems and precompiled binaries for Windows and Linux are available at the SourceForge file manager.
>
> Maxima is a descendant of Macsyma, the legendary computer algebra system developed in the late 1960s at the Massachusetts Institute of Technology. It is the only system based on that effort still publicly available and with an active user community, thanks to its open source nature. Macsyma was revolutionary in its day, and many later systems, such as Maple and Mathematica, were inspired by it.
>
> The Maxima branch of Macsyma was maintained by William Schelter from 1982 until he passed away in 2001. In 1998 he obtained permission to release the source code under the GNU General Public License (GPL). It was his efforts and skill which have made the survival of Maxima possible, and we are very grateful to him for volunteering his time and expert knowledge to keep the original DOE Macsyma code alive and well. Since his death, a group of users and developers has formed to bring Maxima to a wider audience.

> Maxima is updated very frequently, to fix bugs and improve the code and the documentation. We welcome suggestions and contributions from the community of Maxima users. Most discussion is conducted on the Maxima mailing list.

You can download Maxima from the Maxima project site at sourceforge,

`http://maxima.sourceforge.net/`

Cryptography Explorer

Cryptography Explorer is a tool that I developed for the investigation of cryptography and cryptanalysis. It was written mainly to ease the investigation of classical cryptography methods but it also contains features for modern ciphers as well as tools for investigating integer factorization and discrete logarithm calculations.

Cryptography Explorer can be downloaded from my website at,

`http://facultyfp.salisbury.edu/despickler/personal/CryptographyExplorer.html`

Edition
July 26, 2019

Publisher
Don Spickler
Department of Mathematics and Computer Science
Salisbury University
1101 Camden Ave.
Salisbury, Maryland 21801
USA

Contents

Chapter 1

Introduction to Mathematica

1.1 What is Mathematica?

You probably already know this by this time in your mathematical careers, but if you are not familiar with Mathematica. Mathematica is a commercial computer algebra system. Computer algebra systems are programs that are capable of doing exact mathematical computations in a wide range of mathematical subjects. That is, they can solve equations producing exact answers as opposed to giving decimal approximations. They can do symbolic algebra, trigonometry, calculus, differential equations, and so on. Some computer algebra systems have very specific uses, such as finite group theory, while others are built to be more comprehensive. Mathematica is one of the most comprehensive computer algebra systems on the market today.

The following description was taken from the Wolfram site (`http://www.wolfram.com/`)
.

> For more than 25 years, Mathematica has defined the state of the art in technical computingand provided the principal computation environment for millions of innovators, educators, students, and others around the world.
>
> Widely admired for both its technical prowess and elegant ease of use, Mathematica provides a single integrated, continually expanding system that covers the breadth and depth of technical computingand with Mathematica Online, it is now seamlessly available in the cloud through any web browser, as well as natively on all modern desktop systems.

You can purchase Mathematica from the Wolfram site,[4]

`http://www.wolfram.com/`

This introduction to Mathematica is not designed to be a general introduction to the software package. There are far better resources for that online than I could ever hope to write. Here we simply concentrate on what you need to do the cryptography exercises and examples in this set of notes.

As with all computer algebra systems, there are numerous ways to input your calculations to obtain the desired results, some methods are slicker than others. The downside of the slick methods is that they are usually hard to read and unless you are already familiar with the ins and outs of the system it is usually unclear what is happening. Since we are assuming that you have a limited exposure to Mathematica, we do not always take the slickest route to produce the needed calculation. In many cases we will break a calculation down into several steps, where is could be done in a single command. This is done for readability and clarity of the operation. As you become more acquainted with Mathematica you will see other equivalent methods to those in this set of notes.

If you are familiar with Matheiatica you probably already know everything in this introduction. In this case you may want to simply skim over these pages and read the unfamiliar sections.

1.2 The User Interface

Most computer algebra systems have very similar interfaces. There is usually a graphical interface for command input that is where the user enters their calculation commands and a calculation engine in the background, called the kernel in Mathematica, where the calculations are performed.

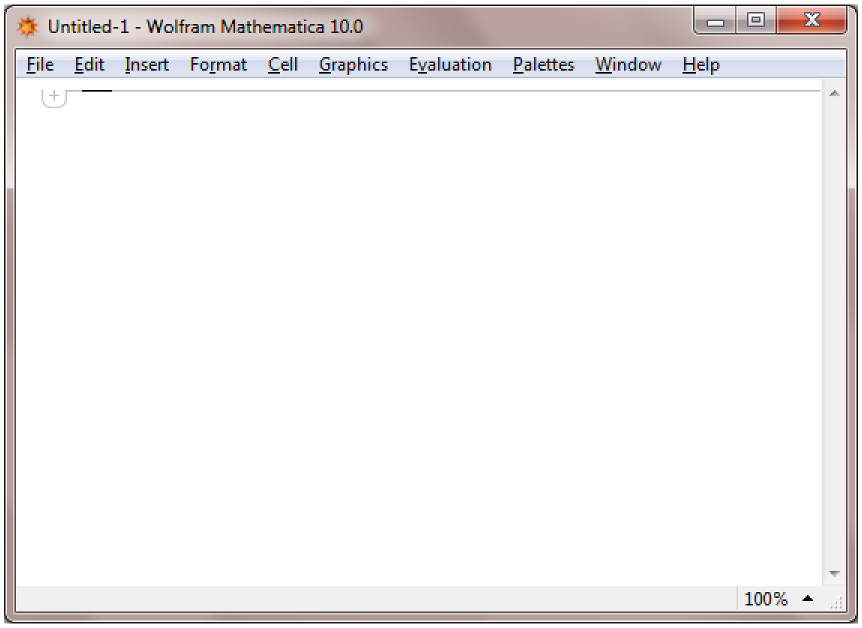
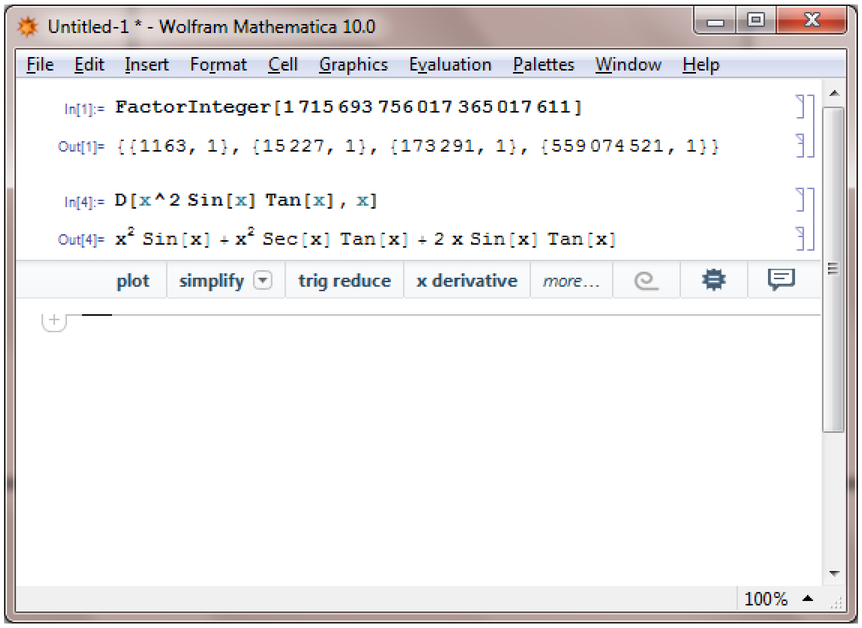


Figure 1.1: User Interface to Mathematica 10.0

When the user types in a command and sends it for calculation, the command is transferred to the kernel, calculated there, and the result is transferred back to the graphical interface. The kernel operations are hidden from the user and you will probably never need to deal with the Mathematica kernel but the reason we are going into this is that on occasions something, usually external to Mathematica, causes the interface to lose the communication link with the kernel. This happens rarely but if you notice that Mathematica is not doing the calculations you send it and you know you are using the correct syntax then you may have lost the kernel link. In these cases, there is a menu option, under the evaluation menu, to start the kernel. Selecting this should reestablish the link for you. Another option is to close Mathematica and restart it, the good old reboot solution.


Untitled-1 * - Wolfram Mathematica 10.0
File Edit Insert Format Cell Graphics Evaluation Palettes Window Help
plot simplify trig reduce x derivative more...
100%

```
In[1]:= FactorInteger[1 715 693 756 017 365 017 611]
Out[1]= {{1163, 1}, {15 227, 1}, {173 291, 1}, {559 074 521, 1}}

In[4]:= D[x^2 Sin[x] Tan[x], x]
Out[4]= x^2 Sin[x] + x^2 Sec[x] Tan[x] + 2 x Sin[x] Tan[x]
```

Figure 1.2: User Interface to Mathematica 10.0 with Commands

As you can see from the above image, the In lines are what the user has input into Mathematica and the Out lines are Mathematica's responses to the inputs. On the first line we simply asked Mathematica to factor a number for us. The Mathematica command for this is FactorInteger followed by square brackets containing the number to be factored.

```
In[1]:= FactorInteger [1 715 693 756 017 365 017 611 ]

Out[1]= {{1163, 1}, {15 227, 1}, {173 291, 1}, {559 074 521, 1}}
```

This in and out tracking comes in handy when you want to use a previous input or output. The % will automatically take the last output that was done. Be careful here, this is not always the output right above the new input. For example, if you go up several commands and reevaluate a command, that is the last output. You can also use the Out[1] notation for output number 1, Out[2] for output number 2, and so on. You will notice that if you redo a command, it will be renumbered with a different input and output number, the original input and output number still have the same values.

```
In[1]:= D[Tan[x^3], x]
```

Out[1]= $3\, x^2\, \text{Sec}\left[x^3\right]^2$

```
In[2]:= D[%, x]

Out[2]= 6 x Sec[x^3]^2 + 18 x^4 Sec[x^3]^2 Tan[x^3]

In[3]:= D[Out[1], x]

Out[3]= 6 x Sec[x^3]^2 + 18 x^4 Sec[x^3]^2 Tan[x^3]
```

Then if we reevaluate the first input it is labeled number 4 but then if we add a new entry the references number 1 (Out[1]+7) it uses the first output from the session.

```
In[4]:= D[Tan[x^3], x]

Out[4]= 3 x^2 Sec[x^3]^2

In[2]:= D[%, x]

Out[2]= 6 x Sec[x^3]^2 + 18 x^4 Sec[x^3]^2 Tan[x^3]

In[3]:= D[Out[1], x]

Out[3]= 6 x Sec[x^3]^2 + 18 x^4 Sec[x^3]^2 Tan[x^3]

In[5]:= Out[1] + 7

Out[5]= 7 + 3 x^2 Sec[x^3]^2
```

It is better practice to assign an output to a variable and use the variable name when needed, for example,

```
In[1]:= d = D[Tan[x^3], x]

Out[1]= 3 x^2 Sec[x^3]^2

In[2]:= D[d, x]

Out[2]= 6 x Sec[x^3]^2 + 18 x^4 Sec[x^3]^2 Tan[x^3]

In[3]:= d + 7

Out[3]= 7 + 3 x^2 Sec[x^3]^2
```

We will discuss assigning variables in more detail in the next section.

All Mathematica commands begin with a capital letter and multi-word commands usually capitalize each word. When applying a command to some input, the input is surrounded by square brackets, not parentheses, like we would write $f(x)$ to apply the function f to the input x. In Mathematica this would be **f[x]**. In Mathematica parentheses are used as grouping symbols for expressions, square brackets are for command or function input and curly brackets are to delimit lists, which includes matrices since these are stored as lists of lists.

Once you have input a command you send it to the kernel for evaluation by selecting Shift + Enter from the keyboard. Also, if your keyboard has a keypad, then simply selecting the keypad Enter (with no Shift) will work as well. When you do this, there may be a slight

Table 1.1: Brackets in Mathematica

Bracket	Usage
( )	Grouping
[ ]	Command and Function Input
{ }	Lists and Matrices

to a long pause while the calculation is being done and then the result will be displayed in the out line. If a calculation is taking too long to complete you can abort the calculation either from the Evaluation menu or by typing Alt+. from the keyboard.

As we pointed out above, computer algebra systems do exact arithmetic, unless otherwise told. So the user can easily input something into the computer algebra system that the computer will not be able to handle or not able to handle in a reasonable amount of time. For example, asking the computer to calculate 1000000! or asking it to factor the 600 digit semiprime that Amazon uses for customer purchases. So if Mathematica is taking a very long time to do a calculation, make sure you did not inadvertently ask it a bad question, and if you did, abort the calculation.

Mathematica also has a command assistant interface called Palettes . There are several different palettes the user can choose form and there is a way to customize your own palettes. If you find these palette systems to be useful then by all means use them. These notes will be concentrating on the commands you need to aide you in cryptography calculations, so we will not be using Mathematica's palette systems. Most of the palette operations are self-explanatory and there are numerous guides to using them on the Internet if you are interested. The palette system is a nice way to get started with Mathematica, to learn some of its functions and syntax. Once you are familiar with Mathematica you will probably find that typing in the commands is quicker.

If you opt to type in the command you will see Mathematica's command completion inter-

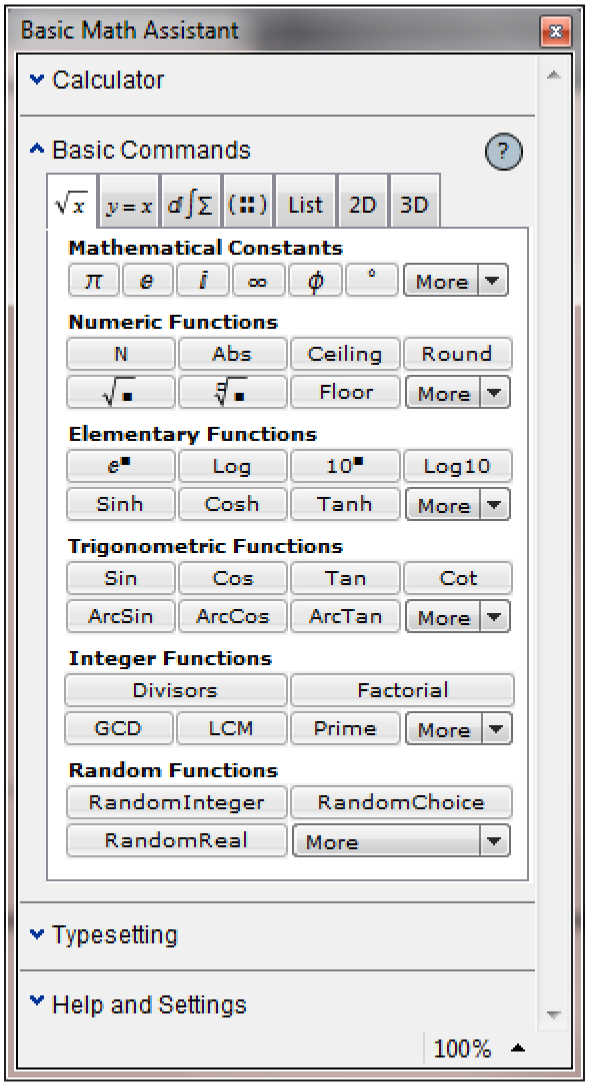
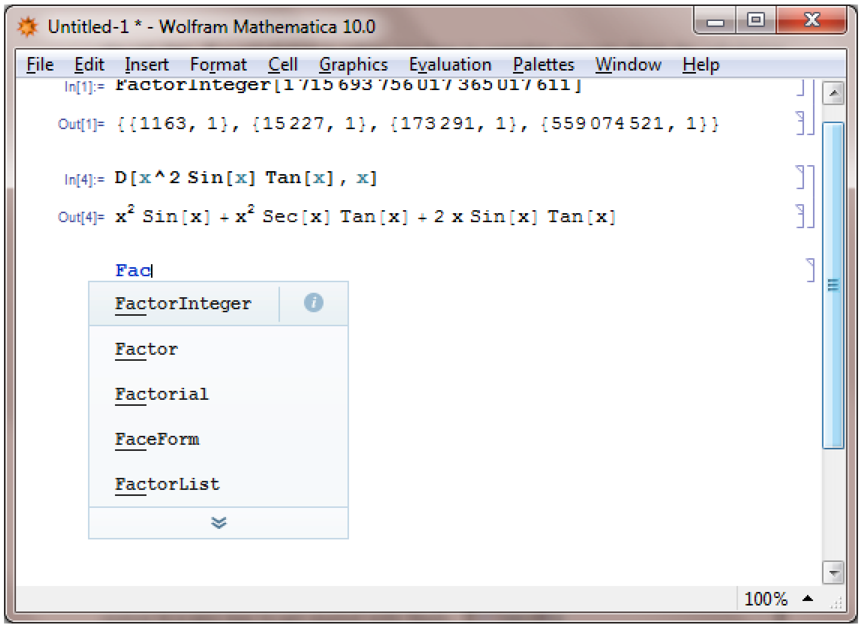


Figure 1.3: Mathematica Palette

face. While you are typing a command, a list will appear below what you are typing as suggestions of the command you want. You can select the command you want from the list by either using the mouse or by using the arrow keys to highlight the command and then use the Tab key to select the command.


Untitled-1 * - Wolfram Mathematica 10.0
File Edit Insert Format Cell Graphics Evaluation Palettes Window Help
100%

```
In[1]:= FactorInteger[1715693756017365017611]

Out[1]= {{1163, 1}, {15227, 1}, {173291, 1}, {559074521, 1}}

In[4]:= D[x^2 Sin[x] Tan[x], x]

Out[4]= x^2 Sin[x] + x^2 Sec[x] Tan[x] + 2 x Sin[x] Tan[x]

        Fac
          FactorInteger
          Factor
          Factorial
          FaceForm
          FactorList
```

Figure 1.4: User Interface to Mathematica 10.0 Command Completion Interface

There are a couple other very nice features to Mathematica we would like to mention before going into command specifics. One of these is Mathematica's user interface is the color coding it uses. Notice in the above screen shot The Fac that is a partial command is in blue. This means that Mathematica does not have a command Fac, but when we finish out the command FactorInteger the font turns to black, meaning that Matheamtica does have a FactorInteger command. So if you are not getting results from Mathematica check the color of your command. Remember that all Mathematica commands begin with a capital letter and Mathematica is case-sensitive. Along the lines of colors, notice that the x's in the derivative command in the screen shot are bluish green. This means that Mathematica is considering them to be variables. If you remove the last x the others will turn blue, meaning that Mathematica does not know what to do with them.

Another simple feature that I use all of the time is the zoom function in the lower right of the graphical window. It is a simple selection box that allows you to increase and decrease the font size of the window quickly.

The other very nice feature we would like to mention is Mathematica's help system. Admittedly most help systems for software packages are not that great, which is why most people will Google a question about software usage before checking out the software's help system. Mathematica is an exception to this rule, it has a very good help system. The built-in searching system is very efficient, there are examples for each command, nifty things you can do with the command, options that can be used with the command, and a section on possible issues that alert you to things you may need to be careful about when using the command. The neatest thing, in my opinion, is that the help system examples are dynamic and user changeable. The examples are written in a Mathematica notebook, so you can change and execute the examples inside the help system and do not need to copy and paste the example into another notebook.

1.3 Basic Calculations

1.3.1 Numeric Calculations

When starting out with any computer algebra system it is good to treat it simply as a fancy calculator, just to get the feel for how it works and basic expression format. Addition, subtraction, multiplication, division and powers are done with the standard mathematics symbols `+-*/^` as you would expect. There are several basic numerical types used in Mathematica but most of the time we will be working in either exact mode or approximate mode.

Computer algebra systems use exact mode whenever possible, this is how they are constructed and frankly what their main purpose is. When calculations are done in exact mode the outputs are integers, rational numbers or expressions involving them. Approximate mode is when we have decimal approximations as our output. In the example below, inputs 1–4 are all in exact mode, note the $\sqrt{2}$ and $\log 25$. Since these numbers are irrational Mathematica will not approximate them. Inputs 5 and 6 produce approximate outputs since we used a decimal in the input expression.

```
In[1]:= 29 147 + 789 273

Out[1]= 818 420

In[2]:= 2 ^ 92

Out[2]= 4 951 760 157 141 521 099 596 496 896

In[3]:= 2 ^ (1 / 2)

Out[3]= √2

In[4]:= Log[25]

Out[4]= Log[25]

In[5]:= 2.0 ^ (1 / 2)

Out[5]= 1.41421

In[6]:= Log[25.0]

Out[6]= 3.21888
```

So the easiest way to force Mathematica into approximation mode is to use decimal numbers in the expression. You can also use a couple commands to convert an exact expression into an approximate expression. The `N` command will convert an exact expression to decimal form and it has the option to change the number of displayed decimal places. In cryptography, we usually deal primarily with integers so there will be few times when we need to get approximations. Nonetheless, here are some examples,

```
In[1]:= 2^(1/2)

Out[1]= √2

In[2]:= N[2^(1/2)]

Out[2]= 1.41421

In[3]:= N[2^(1/2), 50]

Out[3]= 1.4142135623730950488016887242096980785696718753769

In[4]:= N[2^(1/2), 500]

Out[4]= 1.4142135623730950488016887242096980785696718753769480731766797\
         7379907324784621070388503875343276415727350138462309122970249\
         2483605585073721264412149709993583141322266592750559275579995\
         0501152782060571470109559971605970274534596862014728517418640\
         8891986095523292304843087143214508397626036279952514079896872\
         5339654633180882964062061525835239505474575028775996172983557\
         5220337531857011354374603408498847160386899970699004815030544\
         0277903164542478230684929369186215805784631115966687130130156\
         185689872372

In[5]:= 2^(1/2) // N

Out[5]= 1.41421
```

From the above examples you can see that using the `N` command alone gives the default number of decimal places in the approximation. If we add a number option then Mathematica displays that number of decimal places. The final command is an example of a Mathematica "pipe". That is, the result of what is before the `//` is piped into the command after the `//`. So in input number 5, we are taking the exact value of $\sqrt{2}$ and then asking for a decimal representation of it. These pipes come in handy when you want a quick way to change the format of the output.

1.3.2 Algebra

Computer algebra systems will also do algebra, imagine that. So they will do computations with variables just as we would. One thing to be careful with here is assigning values to variables. Once a variable is assigned a value it will replace the variable with that value in all expressions until the variable is reset. There are two basic ways to do assignments in Mathematica, the equal sign, =, for immediate assignments and the colon equal, `:=` for a delayed assignment. The difference between the two is as follows,

- `lhs=rhs` — This is an immediate assignment, the `rhs` is evaluated at the time of assignment.
- `lhs:=rhs` — This is a delayed assignment, the `rhs` is reevaluated every time it is used.

Once an assignment is made then any expression with the `lhs` in it is evaluated as if the `lhs` is the `rhs`. Care must be taken when variables are assigned values, since as long as the assignment is current, the substitution will be made. From time to time you will want to switch back to a variable from an assignment. There are a couple ways to do this. Say we defined `x` to be some numeric value, to reset it to `x`, we could either use the command `x=.` or `Clear[x]`. The following is a few short examples of immediate assignment.

```
In[1]:= x^2 + 3 x + 7

Out[1]= 7 + 3 x + x^2

In[2]:= x = 5

Out[2]= 5

In[3]:= x^2 + 3 x + 7

Out[3]= 47

In[4]:= x =.

In[5]:= x^2 + 3 x + 7

Out[5]= 7 + 3 x + x^2

In[6]:= x = 15

Out[6]= 15

In[7]:= x^2 + 3 x + 7

Out[7]= 277

In[8]:= Clear[x]
```

```
In[9]:= x^2 + 3 x + 7

Out[9]= 7 + 3 x + x^2
```

One thing that the above examples cannot show you is the color changes that happened when the variable x was set to a value. When x was set to a value all the x's in the notebook turned to black, signifying that it was assigned to a value.

Also note that for multiplication we can use the $\star$ symbol, but juxtaposition is also supported in Mathematica. This is both a good thing and a bad thing. While it makes typing a bit easier, it can lead to errors. Consider the following example,

```
In[1]:= x^2 + 3 x + y^3 - y + xy

Out[1]= 3 x + x^2 + xy - y + y^3

In[2]:= x = 5

Out[2]= 5

In[3]:= y = 3

Out[3]= 3

In[4]:= x^2 + 3 x + y^3 - y + xy

Out[4]= 64 + xy

In[5]:= x y

Out[5]= 15
```

Expressions numbers 1 and 4 were typed in without using any spaces. While the output of number 1 looks as we would expect but number 4 is a bit of a surprise. We got an xy and not an extra 15 added to our result. The reason for this is that with no space between the x and the y, Mathematica thought that this was a new variable, named `xy`, and not the product of x and y. For input number 5, we placed a space between the x and the y, and got the desired result.

Also note that expressions are automatically simplified, that is the easy simplifications are done automatically. More complex expressions will not be simplified until you give Mathematica a command to do so. In Mathematica there are two basic simplification commands, `Simplify` and `FullSimplify`. The `FullSimplify` command tends to be used with more difficult expressions, for the computations in this set of notes the `Simplify` command should be sufficient. There are many options that can be used with both but we should only need the basic command.

```
In[1]:= Sin[x]^2 + Cos[x]^2

Out[1]= Cos[x]^2 + Sin[x]^2
```

```
In[2]:= Simplify[%]

Out[2]= 1
```

1.3.3 Execution Timing

In cryptography, and other computationally intensive areas in mathematics and computing, one wants to know how different algorithms that accomplish the same task stack up against each other. Which algorithm factors integers the fastest or finds the discrete logarithm fastest? Or better questions are which algorithms are fastest in which situations? The way this is usually done, theoretically, is by counting the number of mathematical operations that need to be done for the algorithm to come up with a solution. We tend to look at best, average, and worst case scenarios and compare them.

Another method is to do empirical testing. Run several examples using each algorithm and compare the timings. With computer algebra systems, many complex tasks, such as factoring and finding discrete logarithms will implement several different algorithms that work together, and even in parallel. So separating them is sometimes difficult. Nonetheless, we would still like to know execution times for processes run on Mathematica.

In Mathematica, there are two basic timing functions, `Timing` and `AbsoluteTiming`. With both, you simply put the command around the function you wish to time and the output is a list where the first entry is the execution time and the second is the output of the command. The difference between the two commands is that the `Timing` command only tracks the CPU time used, whereas the `AbsoluteTiming` command tracks the total elapsed time. So outside operations, such a other programs running or data transfers form the internet could affect the absolute timing.

```
In[5]:= Timing[
         FactorInteger[
          66 473 167 017 650 137 560 371 563 761 037 563 451 364 913 758 731 751 ⋱
           334 111]]

Out[5]= {0.343202, {{3, 1}, {67, 1}, {1 400 964 127, 1},
          {236 060 486 736 254 900 318 008 190 491 187 456 774 869 150 393  , 1}}}
```

```
In[6]:= AbsoluteTiming[
         FactorInteger[
          66 473 167 017 650 137 560 371 563 761 037 563 451 364 913 758 731 751 ⋱
           334 111]]

Out[6]= {0.358801, {{3, 1}, {67, 1}, {1 400 964 127, 1},
          {236 060 486 736 254 900 318 008 190 491 187 456 774 869 150 393  , 1}}}
```

1.4 Defining Functions

Mathematica has hundreds of built-in functions, trigonometric, logarithmic, hyperbolic, complex valued, exponential, combinatorial, ... In cryptography, we do not tend to need transcendental functions too often and we will look at a few discrete mathematics and number theory functions in the following sections and throughout the body of these notes. There will be times when you will want to define your own functions, this tends to make typing and expression syntax easier when you are dealing with longer expressions. In Mathematica, to define a function, start with the function name, a list of variables (each followed by an underscore) in square brackets, := and then the expression. For example, to define the function $f(x) = x^2 - 3x + 5$,

```
In[1]:= f[x_] := x^2 - 3 x + 5

In[2]:= f[t]

Out[2]= 5 - 3 t + t^2

In[3]:= f[5]

Out[3]= 15

In[4]:= f[-x]

Out[4]= 5 + 3 x + x^2

In[5]:= f[x + h]

Out[5]= 5 - 3 (h + x) + (h + x)^2
```

After the function is defined, you can evaluate the function at values, or expressions, by placing the value or expression in the parentheses, just like we would do in mathematics. Functions can be defined on more than one variable, for example,

```
In[1]:= g[x_, y_] := x^2 - y^2

In[2]:= g[2, 3]

Out[2]= -5

In[3]:= g[t, 7]

Out[3]= -49 + t^2
```

We will discuss Mathematica lists later in these notes but will give a quick example here. Most computer algebra systems store and manipulate information in lists, this is the basis to what are called functional programming languages, like LISP. So computer algebra systems tend to work very efficiently on lists. In Mathematica, a list is a set of expressions separated by commas and delimited by curly brackets. The following is an example of how you can evaluate a function on a list.

```
In[1]:= g[x_, y_] := x^2 - y^2

In[2]:= g[{1, 2, 3, 4}, t]

Out[2]= {1 - t², 4 - t², 9 - t², 16 - t²}

In[3]:= g[{1, 2, 3, 4}, {5, 6, 7, 8}]

Out[3]= {-24, -32, -40, -48}

In[4]:= g[{1, 2, 3, 4}, {5, 6, 7}]

        Thread::tdlen : Objects of unequal length in {1, 4, 9, 16} + {-25, -36, -49} cannot be combined. >>

Out[4]= {-25, -36, -49} + {1, 4, 9, 16}
```

Functions can also be composed with each other and themselves. Furthermore, you can define a function using other function definitions.

```
In[1]:= f[x_] := Sqrt[x + 1]

In[2]:= f[x]
```

Out[2]= $\sqrt{1+x}$

```
In[3]:= f[f[x]]
```

Out[3]= $\sqrt{1+\sqrt{1+x}}$

```
In[4]:= f[f[f[f[x]]]]
```

Out[4]= $\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+x}}}}$

```
In[5]:= g[x_] := Sin[x]

In[6]:= f[g[x]]
```

Out[6]= $\sqrt{1+\mathrm{Sin}[x]}$

```
In[7]:= g[f[x]]
```

Out[7]= $\mathrm{Sin}\left[\sqrt{1+x}\right]$

```
In[8]:= h[x_] := f[f[f[x]]]
```

```
In[9]:= h[x]
```

Out[9]= $\sqrt{1+\sqrt{1+\sqrt{1+x}}}$

As with any computer program you need to be careful what you tell it to do. It will do exactly what you tell it. In the below string of examples we define a function $f(x)$ and then we define the value of x to be 5. Note that in line 3, $f(x)$ is now the expression defined at 5, since x is equal to 5.

```
In[1]:= f[x_] := Sqrt[x + 1]

In[2]:= x = 5

Out[2]= 5

In[3]:= f[x]
```

Out[3]= $\sqrt{6}$

1.5 Some Discrete Mathematics & Number Theory Commands

In this section we will look at a few commands that are related to the number theory and discrete mathematics that we tend to encounter most in the area of cryptography.

1.5.1 Modulus

To compute a simple modulus, $a \pmod{n}$ use the `Mod[a, n]` command.

```
In[1]:= Mod[35, 21]

Out[1]= 14

In[2]:= Mod[-123, 29]

Out[2]= 22
```

1.5.2 Power Calculations with a Modulus

Frequently we need to raise a number to a very large power modulo another number, that is, calculate $a^b \pmod{n}$, where b could be a very large number. The way not to do this is with the command `Mod[a^b, n]`. Although this will work fine for small values of a and b, when b gets large the calculation may become too large for your, or anyone's, computer to handle. The reason is that with this command, the program will first calculate a^b and then take the result modulo n. If b is sufficiently large, the calculation of a^b could produce a number that is too large to fit in your computer's memory. For this reason, a better computational method was devised. The command in Mathematica for this is, `PowerMod[a, b, n]`. The exponentiation algorithm used here is very fast, it raises a to the b power by successive squares and multiplications, each taken modulo n at each stage so that the intermediate calculations do not get too large.

```
In[1]:= PowerMod[5, 12 345, 98 765]

Out[1]= 82 160

In[2]:= PowerMod[12 345, 67 890, 1 000 000 000]

Out[2]= 931 640 625
```

The power modulus command will also find inverses of numbers modulo another, as long as it exists, that is, a^{-1} (mod n). The command `PowerMod[a, -1, n]` will find the inverse of a modulo n if it exists. If the inverse does not exist then this command will return an error. The algorithm used here is the extended euclidean algorithm, followed by a modulus if needed.

```
In[3]:= PowerMod[12 345, -1, 1 000 000 001]

Out[3]= 349 048 198
```

1.5.3 Greatest Common Divisor

To calculate the greatest common divisor of two, or more, numbers in Mathematica simply use the `GCD[a, b, c, ..., n]` command. The algorithm used here is the euclidean algorithm

```
In[1]:= GCD[23, 57]

Out[1]= 1

In[2]:= GCD[467 030, 31 817 075]

Out[2]= 5
```

1.5.4 Extended Greatest Common Divisor

We know that if $d = \gcd(a, b)$, then there exists numbers r and s such that $ar + bs = d$. To calculate the numbers r, s, and d we can use the `ExtendedGCD[a, b]` command. This will return the list $\{r, \{s, d\}\}$. The algorithm used here is the extended euclidean algorithm. This theorem can be extended to more than two numbers, as the last example illustrates.

```
In[1]:= ExtendedGCD[23, 57]

Out[1]= {1, {5, -2}}
```

```
In[2]:= 5 * 23 - 2 * 57

Out[2]= 1

In[3]:= ExtendedGCD[467 030, 31 817 075]

Out[3]= {5, {866 636, -12 721}}

In[4]:= 866 636 * 467 030 - 12 721 * 31 817 075

Out[4]= 5

In[5]:= ExtendedGCD[2364, 2748, 28 312]

Out[5]= {4, {-219 387, 188 720, 1}}

In[6]:= -219 387 * 2364 + 2748 * 188 720 + 28 312

Out[6]= 4
```

1.5.5 Least Common Multiple

To calculate the least common multiple of several numbers use the `LCM[a, b, c, ...]` command.

```
In[1]:= LCM[5, 15, 35]

Out[1]= 105
```

1.5.6 Chinese Remainder Theorem

The Chinese Remainder Theorem is really an algorithm for solving a system of congruences,

$$\begin{aligned} x &= r_1 \pmod{m_1} \\ x &= r_2 \pmod{m_2} \\ x &= r_3 \pmod{m_3} \\ &\vdots \\ x &= r_n \pmod{m_n} \end{aligned}$$

where the set $\{m_1, m_2, \ldots, m_n\}$ are positive and pairwise coprime integers. The Mathematica command to solve this system is `ChineseRemainder[{r_1, ..., r_n}, {m_1, ..., m_n}]`. Note that the residues and the moduli are in lists and the corresponding entries define each of the congruences. If the set of moduli are coprime the Chinese Remainder Theorem guarantees a solution. If, on the other hand, moduli are not coprime then there may or may not be a solution. If Mathematica cannot find a solution to the system it will return the command as output.

```
In[1]:= ChineseRemainder [{1, 2}, {5, 7}]

Out[1]= 16

In[2]:= ChineseRemainder [{1, 2, 3, 4}, {5, 7, 9, 11}]

Out[2]= 1731
```

1.5.7 Functions for Primes

There are numerous functions in Mathematica for working with prime numbers. The first we will look at is primality testing. The Mathematica command to test if a number is prime (or probably prime) is `PrimeQ[n]`. If `PrimeQ[n]` returns false, n is a composite number and if it returns true, n is a prime number with very high probability.

```
In[1]:= PrimeQ [17]

Out[1]= True

In[2]:= PrimeQ [620 743 261 954 923 659 141 ]

Out[2]= True

In[3]:= PrimeQ [4 294 967 297 ]

Out[3]= False
```

Mathematica also has a function for finding the next probable prime. This function comes in two forms, `NextPrime[n]` and `NextPrime[n, k]`. The `NextPrime[n]` function finds the next prime greater than n and the `NextPrime[n, k]` function finds the k^{th} prime above n. This second form has the added bonus that if $k = -1$ the function will return the next prime smaller than n.

```
In[1]:= NextPrime [19]

Out[1]= 23

In[2]:= NextPrime [39 196 736 173 617 367 361 073 651 769 157 617 369 164 ]

Out[2]= 39 196 736 173 617 367 361 073 651 769 157 617 369 187

In[3]:= NextPrime [19, 5]

Out[3]= 41

In[4]:= NextPrime [1 000 000 000 , -1]

Out[4]= 999 999 937
```

1.5.8 Jacobi and Legendre Symbols

Recall that the Legendre symbol is defined as follows, for an odd prime n,

$$\left(\frac{m}{n}\right) = \begin{cases} 0, & \text{if } m \equiv 0 \pmod{n} \\ 1, & \text{if } 0 \not\equiv m \equiv x^2 \pmod{n}, \text{ for some } x \\ -1, & \text{otherwise} \end{cases}$$

So for an odd prime n, the Legendre symbol will tell us if an integer m is a quadratic residue modulo n. The Jacobi symbol is a generalization of the Legendre symbol, it is defined for any odd number n as

$$\left(\frac{m}{n}\right) = \left(\frac{m}{p_1}\right)^{a_1}\left(\frac{m}{p_2}\right)^{a_2}\cdots\left(\frac{m}{p_r}\right)^{a_r}$$

where all of the p_i are distinct primes and $n = p_1^{a_1}p_2^{a_2}\cdots p_r^{a_r}$. Note that each of the terms in the above product are Legendre symbols, since all of the p_i are prime. One big difference between the Jacobi and Legendre symbols is that if n is not prime and $\left(\frac{m}{n}\right) = 1$ then we are not guaranteed that m is a quadratic residue modulo n. On the other hand, if $\left(\frac{m}{n}\right) = -1$ then we know that m is not a quadratic residue modulo n.

In Mathematica, the command to do both of these symbols is `JacobiSymbol[m, n]`. If n is prime, then this is the Legendre symbol and we can deduce if m is a quadratic residue modulo n. If n is not prime then we are working with the Jacobi symbol.

```
In[1]:= JacobiSymbol[5, 23]

Out[1]= -1

In[2]:= JacobiSymbol[3, 23]

Out[2]= 1

In[3]:= JacobiSymbol[19, 231]

Out[3]= 1

In[4]:= JacobiSymbol[17, 231]

Out[4]= -1

In[5]:= JacobiSymbol[3, 231]

Out[5]= 0
```

So in our above examples,

1. 5 is not a quadratic residue modulo 23.
2. 3 is a quadratic residue modulo 23. In fact, $3 \equiv 7^2 \pmod{23}$.
3. We do not know if 19 is a quadratic residue modulo 231, but it is possible.

4. 17 is not a quadratic residue modulo 231.
5. Since $\gcd(3, 231) \neq 1$, one of the Legendre symbols in the product definition of the Jacobi symbol is 0, making the product 0.

1.5.9 Continued Fractions

A continued fraction is when you take a number x and express it in the form,

$$x = a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cfrac{1}{\ddots}}}}$$

For some values of x their continued fraction representation will terminate, some will repeat and some will neither terminate nor repeat. For example,

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{\ddots}}}$$

$$\frac{1+\sqrt{5}}{2} = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}$$

$$\frac{5742}{2131} = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{11 + \cfrac{1}{5}}}}}}}}$$

There are several Mathematica commands that come in handy when working with continued fractions and we will create one that will make some of the computations in these notes a little easier. Mathematica's `ContinuedFraction[n]` and `ContinuedFraction[n, k]` commands will return a list representation of the continued fraction representation of n, the second command gives just the first k entries. Here n can be any real number, it does not have to be rational. So an output of $\{a_1, a_2, a_3, a_4, \ldots\}$ is a representation for,

$$a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cfrac{1}{\ddots}}}}$$

If the number has a terminating continued fraction representation, Mathematica will produce the entire representation, such as, in output number 4 below. In the case where the continued fraction representation is repeating, Mathematica will halt the representation once it notices that it has finished a period of the repetition. So in output number 1, Mathematica gives $\{1, \{2\}\}$ for the representation of $\sqrt{2}$. The curly brackets around the 2 represents the repeating part. So Mathematica is telling us that the representation is $\{1, 2, 2, 2, \ldots\}$.

```
In[1]:= ContinuedFraction [Sqrt[2]]

Out[1]= {1, {2}}

In[2]:= ContinuedFraction [Sqrt[2], 20]

Out[2]= {1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2}

In[3]:= ContinuedFraction [(1 + Sqrt[5]) / 2]

Out[3]= {1, {1}}

In[4]:= ContinuedFraction [632 816 312 / 5 321 548 121]

Out[4]= {0, 8, 2, 2, 3, 1, 8, 1, 2, 5, 3, 11, 6, 1, 5, 7, 3, 2, 2}
```

To convert a list to a continued fraction use the `FromContinuedFraction(L)` where L is a list of the form $\{a_1, a_2, \ldots, a_n\}$ or $\{a_1, a_2, \ldots, a_n, \{r_1, r_2, \ldots, r_m\}\}$.

```
In[1]:= lst = ContinuedFraction [632 816 312 / 5 321 548 121]

Out[1]= {0, 8, 2, 2, 3, 1, 8, 1, 2, 5, 3, 11, 6, 1, 5, 7, 3, 2, 2}

In[2]:= FromContinuedFraction [lst]

         632 816 312
Out[2]= -------------
        5 321 548 121

In[3]:= lst2 = ContinuedFraction [Sqrt[2]]

Out[3]= {1, {2}}

In[4]:= FromContinuedFraction [lst2]

Out[4]= √2
```

```
In[5]:= lst3 = ContinuedFraction [Sqrt[2], 20]

Out[5]= {1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2}

In[6]:= FromContinuedFraction [lst3]
```

Out[6]= $\frac{22\,619\,537}{15\,994\,428}$

```
In[7]:= N[%, 20]

Out[7]= 1.4142135623730964308

In[8]:= FromContinuedFraction [{5, 2, 4, {1, 2, 3}}]
```

Out[8]= $\frac{1}{71}\left(381+\sqrt{37}\right)$

There are times when we will want to find the continued fraction representation of a number and then look at successive approximations by taking more and more of the continued fraction. For example, with $\sqrt{2}$, we would look at

$$1+\frac{1}{2}=\frac{3}{2} \qquad 1+\frac{1}{2+\frac{1}{2}}=\frac{7}{5} \qquad 1+\frac{1}{2+\frac{1}{2+\frac{1}{2}}}=\frac{17}{12}$$

and so on. Mathematica has a built-in function, `Convergents[n,k]`, that returns a list of k continued fraction approximations of n. We can also create a simple function that can give these to us one at a time, which might be more convenient in some calculations. Define the new function `FromContinuedFractionN` as follows,

```
FromContinuedFractionN[L_, k_] := FromContinuedFraction[L[[1 ;; k]]]
```

This will take in a continued fraction list of the non-repeating type and an integer k and output the k^{th} approximation of the continued fraction. In Mathematica, the command `L[[1 ;; k]]` simply extracts the first k elements of the list L and returns the sublist.

```
In[9]:= FromContinuedFractionN [L_ , k_] :=
         FromContinuedFraction [L[[1 ;; k]]]

In[10]:= FromContinuedFractionN [lst3, 3]
```

Out[10]= $\frac{7}{5}$

```
In[11]:= FromContinuedFractionN [lst3, 10]
```

Out[11]= $\frac{3363}{2378}$

```
In[12]:= Convergents[Sqrt[2], 10]
```

$$\text{Out[12]=}\ \left\{1, \frac{3}{2}, \frac{7}{5}, \frac{17}{12}, \frac{41}{29}, \frac{99}{70}, \frac{239}{169}, \frac{577}{408}, \frac{1393}{985}, \frac{3363}{2378}\right\}$$

```
In[13]:= N[Convergents[Sqrt[2], 10], 10]

Out[13]= {1.000000000, 1.500000000, 1.400000000, 1.416666667, 1.413793103,
          1.414285714, 1.414201183, 1.414215686, 1.414213198, 1.414213625}
```

1.5.10 Solving Equations

Of course, we need to solve equations. Mathematica has a very powerful equation solver, `Solve`. We will only be using a small portion of what it is capable of doing. For example, it can find the, relatively ugly, exact solutions to $x^3 - 3x^2 + 2x + 5 = 0$.

```
In[1]:= Solve[x^3 - 3 x^2 + 2 x + 5 == 0, x]
```

$$\text{Out[1]=}\ \left\{\left\{x \to 1 - \left(\frac{2}{3\left(45-\sqrt{2013}\right)}\right)^{1/3} - \frac{\left(\frac{1}{2}\left(45-\sqrt{2013}\right)\right)^{1/3}}{3^{2/3}}\right\},\right.$$
$$\left\{x \to 1 + \frac{\left(1+i\sqrt{3}\right)\left(\frac{1}{2}\left(45-\sqrt{2013}\right)\right)^{1/3}}{2\times 3^{2/3}} + \frac{1-i\sqrt{3}}{2^{2/3}\left(3\left(45-\sqrt{2013}\right)\right)^{1/3}}\right\},$$
$$\left.\left\{x \to 1 + \frac{\left(1-i\sqrt{3}\right)\left(\frac{1}{2}\left(45-\sqrt{2013}\right)\right)^{1/3}}{2\times 3^{2/3}} + \frac{1+i\sqrt{3}}{2^{2/3}\left(3\left(45-\sqrt{2013}\right)\right)^{1/3}}\right\}\right\}$$

A little more down to earth, the solutions to $3x^2 - 2x - 5 = 0$ are $\frac{5}{3}$ and -1.

```
In[2]:= Solve[3 x^2 - 2 x - 5 == 0, x]
```

$$\text{Out[2]=}\ \left\{\{x \to -1\}, \left\{x \to \frac{5}{3}\right\}\right\}$$

Several things to notice about the syntax to the `Solve` command. When you are solving a single equation, the first argument is the equation to be solved and the second argument is the variable to solve the equation for. An equation in Mathematica is simply two Mathematica expressions with a double equal sign between them. If the equation does not have a double equal in it, Mathematica will complain. If there is only one variable in the equation, then the variable to be solved for can be omitted and Mathematica will take the one in the equation.

Mathematica can also solve modular equations. To tell Mathematica that we want to solve the equation over a modulus all we need to do is put in a `Modulus` option at the end of the command. The syntax for this is `Modulus -> m` where m is the desired modulus.

The arrow is a common Mathematica notation for setting options, it is created by a − and > characters next to each other. When you type this in, Mathematica will automatically shorten it to a single arrow character. It is possible that there are no solutions to an equation or modular equation, in that case Mathematica will return an empty set.

```
In[1]:= Solve[2 x - 5 == 0, x, Modulus → 7]

Out[1]= {{x → 6}}

In[2]:= Solve[3 x^2 - 2 x - 5 == 0, x, Modulus → 7]

Out[2]= {{x → 4}, {x → 6}}

In[3]:= Solve[3 x^2 - 2 x - 5 == 0, x, Modulus → 3]

Out[3]= {{x → 2}}

In[4]:= Solve[x^2 == 25, x, Modulus → 37]

Out[4]= {{x → 5}, {x → 32}}

In[5]:= Mod[32^2, 37]

Out[5]= 25

In[6]:= Solve[x^10 == 25, x, Modulus → 37]

Out[6]= {{x → 2}, {x → 35}}

In[7]:= Mod[35^10, 37]

Out[7]= 25

In[8]:= Mod[2^10, 37]

Out[8]= 25

In[9]:= Solve[x^2 == 2, x, Modulus → 37]

Out[9]= {}
```

1.5.11 Factoring

Factoring is essential for many cyptographic processes and cryptanalysis. In fact, finding faster factoring algorithms is one of the central goals in cryptography. To factor an integer in Mathematica use the `FactorInteger[n]` command, where n is the number to be factored.

```
In[1]:= FactorInteger [78 319 748 917 546 879 163 956 196 193 769 134 651  ]

Out[1]= {{3, 1}, {13, 1}, {37, 1},
          {19 151 901 878 983 , 1}, {2 833 955 636 283 909 138 479 , 1}}
```

As you can see from the output above, the `FactorInteger` command returns a list of factor lists, in each factor list the first entry is the factor and the second is the multiplicity of the factor.

1.5.12 Factoring Polynomials

In Mathematica, the command to factor a polynomial is `Factor`. In the first example below, the input and output is the factorization of $x^6 + x^5 + x^3 + 1$ using integer coefficients, that is, the coefficients are integers and the coefficients of the factorization are also integers.

To factor a polynomial modulo a prime in Mathematica, simply include the `Modulus` option at the end of the command, as we did with the second input. Now when the factor command is invoked the factorization will be modulo the prime.

```
In[1]:= Factor [x^6 + x^5 + x^3 + 1]

Out[1]= (1 + x) (1 + x^2) (1 - x + x^3)

In[2]:= Factor [x^6 + x^5 + x^3 + 1, Modulus → 2]

Out[2]= (1 + x)^3 (1 + x + x^3)
```

1.5.13 Euler Totient Function

The Euler totient function, also known as the Euler phi function, $\phi(n)$ is the number of integers less than or equal to n which are relatively prime to n. In Mathematica this command is simply, `EulerPhi[n]`.

```
In[1]:= EulerPhi [12]

Out[1]= 4

In[2]:= EulerPhi [58 741 398 061 036 107 365 103 746 301 374 560 173 465 071 346  ]

Out[2]= 18 873 378 929 238 574 252 365 598 155 418 446 287 441 510 400
```

Note that the calculation of the totient function requires the factorization of n, hence the calculation time of the totient of a large number could be lengthy.

1.5.14 Primitive Roots and Element Orders

A primitive root modulo n is a number whose powers modulo n generate all numbers less than n that are relatively prime to n. In more mathematical lingo, a primitive root modulo n is a number whose powers modulo n generate all numbers in $(\mathbb{Z}/n\mathbb{Z})^*$. If the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$ is cyclic, `PrimitiveRoot[n]` computes the smallest primitive root modulo n. $(\mathbb{Z}/n\mathbb{Z})^*$ is cyclic if n is equal to 2, 4, p^k or $2p^k$, where p is prime and greater than 2 and k is a natural number. Most of the time, for us, n will be a prime number.

```
In[1]:= PrimitiveRoot[13]

Out[1]= 2

In[2]:= PrimitiveRoot[48 130 750 178 370 514 570 138 771 ]

Out[2]= 12
```

Mathematica also has several other commands that are useful for working with primitive roots. The command `PrimitiveRootList[n]` will return a list of primitive roots modulo n and the `MultiplicativeOrder[k,n]` will return the multiplicative order of k modulo n.

```
In[3]:= PrimitiveRootList[373]

Out[3]= {2, 5, 6, 11, 14, 15, 24, 26, 32, 34, 35, 42, 43, 44, 47, 53, 54, 57,
         60, 61, 62, 65, 72, 76, 77, 78, 79, 80, 82, 85, 92, 98, 99, 102,
         105, 110, 118, 127, 128, 131, 132, 135, 141, 143, 149, 150, 151,
         155, 159, 162, 166, 168, 171, 172, 174, 178, 180, 182, 183, 186,
         187, 190, 191, 193, 195, 199, 201, 202, 205, 207, 211, 214,
         218, 222, 223, 224, 230, 232, 238, 241, 242, 245, 246, 255,
         263, 268, 271, 274, 275, 281, 288, 291, 293, 294, 295, 296,
         297, 301, 308, 311, 312, 313, 316, 319, 320, 326, 329, 330,
         331, 338, 339, 341, 347, 349, 358, 359, 362, 367, 368, 371}

In[4]:= MultiplicativeOrder[PrimitiveRoot[373], 373]

Out[4]= 372

In[5]:= MultiplicativeOrder[200, 373]

Out[5]= 12

In[6]:= MultiplicativeOrder[370, 373]

Out[6]= 93
```

Although Mathematica does not seem to have a command to test if an element is a primitive root, it is easy to construct from the `MultiplicativeOrder` command. Following the Mathematica naming conventions, a boolean valued function that asks if an input has a particular property is usually called something that describes the property and ends in a `Q`, so the name `PrimitiveRootQ` would be an obvious choice for this function. This function will return true if k is a primitive root modulo n, and false if it is not. If n is not of the form appropriate for the `PrimitiveRoot` or `MultiplicativeOrder` commands then the result will be either an error or a return of the calling command.

```
PrimitiveRootQ[k_, n_] :=
  MultiplicativeOrder[PrimitiveRoot[n], n] ==
   MultiplicativeOrder[k, n];
```

```
In[7]:= PrimitiveRootQ[k_, n_] :=
          MultiplicativeOrder[PrimitiveRoot[n], n] ==
           MultiplicativeOrder[k, n];

In[8]:= PrimitiveRootQ[7, 373]

Out[8]= False

In[9]:= PrimitiveRootQ[291, 373]

Out[9]= True
```

One should note that these commands rely on the factorization of the totient function of n, hence for large n the calculation could be lengthy.

1.5.15 Discrete Logarithms

Given three numbers, g, a, and n the solution x to the congruence $g^x \equiv a \pmod{n}$ is called the discrete logarithm of a, base g modulo n, if x exists.

If $(\mathbb{Z}/n\mathbb{Z})^*$ is a cyclic group (n is equal to 2, 4, p^k or $2p^k$, where p is prime and greater than 2 and k is a natural number), g a primitive root modulo n and let a be a member of this group. A more general form of the `MultiplicativeOrder` command will solve the discrete log problem. The command `MultiplicativeOrder[g, n, {a}]` will solve the congruence $g^x \equiv a \pmod{n}$.

```
In[1]:= MultiplicativeOrder[5, 7, {2}]

Out[1]= 4

In[2]:= Mod[5^4, 7]

Out[2]= 2

In[3]:= MultiplicativeOrder[3, 7, {6}]

Out[3]= 3
```

```
In[4]:= Mod[3^3, 7]

Out[4]= 6
```

1.6 Vectors and Matrices

We will start out with some basic operations on matrices and vectors in general and then we will discuss some ways of doing matrix operations over a modulus.

In Mathematica, vectors are simply lists and matrices are just lists of lists. One nifty thing with the way that Mathematica handles lists is that we do not need to distinguish between row vectors and column vectors, as we do in most other linear algebra software packages.

1.6.1 Defining a Matrix and a Vector

A vector is simply a list, recall that a list in Mathematica is a sequence of expressions separated by commas and enclosed in curly brackets. A matrix is a list of lists, each of the contained lists are the rows to the matrix. Since a matrix is a list of lists, Mathematica does not know if you, the user, wants to see a list of lists or a matrix. So there is a command `MatrixForm` that will display a matrix list as a matrix. You can also apply this command as a pipe at the end of a matrix expression.

```
In[1]:= v = {2, 5, 7}

Out[1]= {2, 5, 7}

    In[2]:= MatrixForm[v]

Out[2]//MatrixForm=
```

$$\begin{pmatrix} 2 \\ 5 \\ 7 \end{pmatrix}$$

```
In[3]:= m = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

Out[3]= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

    In[4]:= MatrixForm[m]

Out[4]//MatrixForm=
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
In[5]:= m // MatrixForm

Out[5]//MatrixForm=
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

We will discuss matrix operations a little later but the period is the multiplication operator for matrices. The two commands below show that Mathematica is treating the vector v as both a row vector and a column vector, without the need to explicitly convert it. In input 6, v is a column vector and in input 7, v is a row vector.

```
In[6]:= m.v

Out[6]= {33, 75, 117}

In[7]:= v.m

Out[7]= {71, 85, 99}
```

Another nifty thing you can do with matrices is to extract rows, columns and entries relatively easily. You can also join matrices together, add rows and columns to a matrix, and change entries

Once a matrix, say m is defined, you can extract the (i, j) entry using `m[[i,j]]` or `m[[i]][[j]]`. You can extract a row by `m[[i]]`, where i is the row to extract. Note that these operations do not alter the original matrix.

```
In[1]:= m = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}

Out[1]= {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}

In[2]:= m // MatrixForm

Out[2]//MatrixForm=
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

```
In[3]:= m[[1]]

Out[3]= {1, 2, 3, 4}

In[4]:= m[[2]][[3]]

Out[4]= 7

In[5]:= m[[2, 3]]

Out[5]= 7
```

You can extract a sequence of rows using the command `m[[i;;j]]`.

```
In[6]:= m[[2 ;; 3]]

Out[6]= {{5, 6, 7, 8}, {9, 10, 11, 12}}
```

Column extraction is similar, you simply need to put an `All` in for the row selection, so `m[[All, i]]` will extract the i^{th} column. Also, using the range operation can extract a sequence of columns, the command `m[[All, i;;j]]` will extract columns i to j.

```
In[7]:= m[[All, 2]]

Out[7]= {2, 6, 10}

In[8]:= m[[All, 2 ;; 3]] // MatrixForm

Out[8]//MatrixForm=
```

$$\begin{pmatrix} 2 & 3 \\ 6 & 7 \\ 10 & 11 \end{pmatrix}$$

You can assign one matrix to another, be careful which type of assignment you use, these is a difference between the immediate and the delayed assignment. For example, the immediate assignment of a to m, as in input 9, will copy the contents of m to a, but the delayed assignment of d to m will not. In input 13, we change the $(1,1)$ position of m to x, note that this alters m as expected and it does not alter a but it does alter d, since when d is used, it looks at the current state of m and not the state of m when the assignment was done, as it did with a.

```
In[9]:= a = m

Out[9]= {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}

In[10]:= a

Out[10]= {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}

In[11]:= d := m

In[12]:= d

Out[12]= {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}

In[13]:= m[[1, 1]] = x

Out[13]= x

In[14]:= m

Out[14]= {{x, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}
```

```
In[15]:= a

Out[15]= {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}

In[16]:= d

Out[16]= {{x, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}
```

You can also join matrices together, add rows and columns to matrices as well. As with extraction, these operations do not alter the original matrices, if you wish to do that you need to include an assignment of the result to a variable. The `Join` command will join matrices and/or vectors by rows, that is vertically. You can join matrices horizontally or add columns by adding a 2 at the end of the command, as in input 20.

```
In[17]:= b = {{-1, 4, 2, -1}, {3, 2, 1, 1}, {5, 2, 7, 1}, {2, 7, 5, 1}}

Out[17]= {{-1, 4, 2, -1}, {3, 2, 1, 1}, {5, 2, 7, 1}, {2, 7, 5, 1}}

    In[18]:= Join[m, b] // MatrixForm

 Out[18]//MatrixForm=
```

$$\begin{pmatrix} x & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ -1 & 4 & 2 & -1 \\ 3 & 2 & 1 & 1 \\ 5 & 2 & 7 & 1 \\ 2 & 7 & 5 & 1 \end{pmatrix}$$

```
    In[19]:= Join[m, {{3, 3, 3, 3}}] // MatrixForm

 Out[19]//MatrixForm=
```

$$\begin{pmatrix} x & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 3 & 3 & 3 & 3 \end{pmatrix}$$

```
    In[20]:= Join[m, Transpose[{{3, 3, 3}}], 2] // MatrixForm

 Out[20]//MatrixForm=
```

$$\begin{pmatrix} x & 2 & 3 & 4 & 3 \\ 5 & 6 & 7 & 8 & 3 \\ 9 & 10 & 11 & 12 & 3 \end{pmatrix}$$

Since matrices in Mathematica are simply lists of lists, the `Table` command gives a very versatile tool for the construction of a general matrix that has some pattern to the entries.

The following example constructs a Hilbert matrix, which is a square matrix whose (i, j) entry is $\frac{1}{i+j-1}$.

```
In[21]:= c = Table[1 / (i + j - 1), {i, 1, 5}, {j, 1, 5}] // MatrixForm
```

Out[21]//MatrixForm=

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{pmatrix}$$

Mathematica has many more commands for the creation of special matrices, extraction and joining, but this should be sufficient for our purposes. If you are interested in fancier manipulations, please see the Mathematica help system.

1.6.2 Matrix Arithmetic

Matrix addition and subtraction are done with the usual + and − operators. If the matrices are the same size then the operation returns the resulting matrix, if the matrices are not the same size then Mathematica displays an error. Matrix multiplication is not done with the $\star$ symbol, `M*A` will return an entry by entry product, which is not the standard matrix multiplication. The same is true for the / symbol, `M/A` will return an entry by entry quotient. There may be times you want to use these types of operations but we are more interested in the standard matrix multiplication. Matrix multiplication is done with the **.** symbol, `M.A` will return the matrix product as long as the matrices are of compatible size, if not, you will get an error.

```
In[1]:= m = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

Out[1]= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

In[2]:= a = {{1, 0, 1}, {2, 1, 5}, {3, 2, 0}}

Out[2]= {{1, 0, 1}, {2, 1, 5}, {3, 2, 0}}

In[3]:= b = {{-1, 3, 2}, {-2, 0, 4}}

Out[3]= {{-1, 3, 2}, {-2, 0, 4}}

In[4]:= c = {{-1, 3}, {-2, 0}, {1, 1}}

Out[4]= {{-1, 3}, {-2, 0}, {1, 1}}
```

```
In[5]:= a + m

Out[5]= {{2, 2, 4}, {6, 6, 11}, {10, 10, 9}}

In[6]:= a - m

Out[6]= {{0, -2, -2}, {-2, -4, -1}, {-4, -6, -9}}

In[7]:= a + b

        Thread::tdlen : Objects of unequal length in
              {{1, 0, 1}, {2, 1, 5}, {3, 2, 0}} + {{-1, 3, 2}, {-2, 0, 4}} cannot be combined. >>

Out[7]= {{-1, 3, 2}, {-2, 0, 4}} + {{1, 0, 1}, {2, 1, 5}, {3, 2, 0}}

In[8]:= b - Transpose[c]

Out[8]= {{0, 5, 1}, {-5, 0, 3}}

In[9]:= a.m

Out[9]= {{8, 10, 12}, {41, 49, 57}, {11, 16, 21}}

In[10]:= b.c

Out[10]= {{-3, -1}, {6, -2}}

In[11]:= c.b

Out[11]= {{-5, -3, 10}, {2, -6, -4}, {-3, 3, 6}}

In[12]:= m.b

         Dot::dotsh : Tensors {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}} and {{-1, 3, 2}, {-2, 0, 4}} have incompatible shapes. >>

Out[12]= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}.{{-1, 3, 2}, {-2, 0, 4}}
```

Matrix powers are not done by ^ but rather with the command `MatrixPower`. If A is a square matrix then `MatrixPower[A,3]` will return A^3. Using only the single power symbol will return a matrix with each entry raised to the power, again, this might be something you want to do but not for taking a matrix power. If A is not a square matrix, you will get an error when taking a power.

```
In[13]:= MatrixPower[a, 3]

Out[13]= {{11, 4, 14}, {62, 25, 74}, {50, 28, 17}}
```

Finding the inverse of a matrix can be done with `MatrixPower[A,-1]` or with the `Inverse(A)` command. If A is not square or if the matrix is not invertible you will get an error. You can find the determinant of a square matrix using the `Det[A]` command.

```
In[14]:= Det[a]

Out[14]= -9

In[15]:= Inverse[a]
```

Out[15]= $\left\{\left\{\frac{10}{9}, -\frac{2}{9}, \frac{1}{9}\right\}, \left\{-\frac{5}{3}, \frac{1}{3}, \frac{1}{3}\right\}, \left\{-\frac{1}{9}, \frac{2}{9}, -\frac{1}{9}\right\}\right\}$

The following is what happens if you use the standard multiplication, division and power operations on matrices, everything is carried out entry by entry.

```
In[16]:= a^3

Out[16]= {{1, 0, 1}, {8, 1, 125}, {27, 8, 0}}

In[17]:= a * m

Out[17]= {{1, 0, 3}, {8, 5, 30}, {21, 16, 0}}

In[18]:= a / m
```

Out[18]= $\left\{\left\{1, 0, \frac{1}{3}\right\}, \left\{\frac{1}{2}, \frac{1}{5}, \frac{5}{6}\right\}, \left\{\frac{3}{7}, \frac{1}{4}, 0\right\}\right\}$

1.6.3 Matrix Reduction

The Mathematica command for reducing a matrix to reduce row echelon form is `RowReduce[A]`, where A is the matrix to be reduced. The `RowReduce[A]` command returns the echelon form of the matrix A, as produced by Gaussian elimination. The reduced echelon form is computed from A by elementary row operations such that the first non-zero element in each row in the resulting matrix is one and the column elements over and under the first one in each row are all zero.

```
In[1]:= m = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

Out[1]= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

In[2]:= a = {{1, 0, 1}, {2, 1, 5}, {3, 2, 0}}

Out[2]= {{1, 0, 1}, {2, 1, 5}, {3, 2, 0}}

In[3]:= b = {{-1, 3, 2}, {-2, 0, 4}}

Out[3]= {{-1, 3, 2}, {-2, 0, 4}}

In[4]:= RowReduce[m]

Out[4]= {{1, 0, -1}, {0, 1, 2}, {0, 0, 0}}
```

```
In[5]:= RowReduce[a]

Out[5]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}

In[6]:= RowReduce[b]

Out[6]= {{1, 0, -2}, {0, 1, 0}}
```

1.6.4 Modular Matrix Operations

When doing modular arithmetic on matrices or matrix reduction in Mathematica, it takes a couple different techniques, most of which we have seen. You simply need to be careful which technique you use for which operation.

When doing modular matrix arithmetic, that is, addition, subtraction, multiplication, and positive powers, simply put the operation inside a `Mod` command. Inverse and negative powers require a different method which we will discuss below. One word of caution, the `PowerMod` function does not work on matrices, at least not the way we want to, so to take a modular matrix power, you first take the matrix power and then the modulus. So the matrix powers should not be too large.

```
In[1]:= m = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

Out[1]= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

In[2]:= a = {{1, 0, 1}, {2, 1, 5}, {3, 2, 0}}

Out[2]= {{1, 0, 1}, {2, 1, 5}, {3, 2, 0}}

In[3]:= b = {{-1, 3, 2}, {-2, 0, 4}}

Out[3]= {{-1, 3, 2}, {-2, 0, 4}}

In[4]:= Mod[m, 5]

Out[4]= {{1, 2, 3}, {4, 0, 1}, {2, 3, 4}}

In[5]:= Mod[a.m, 5]

Out[5]= {{3, 0, 2}, {1, 4, 2}, {1, 1, 1}}

In[6]:= Mod[b.a, 5]

Out[6]= {{1, 2, 4}, {0, 3, 3}}

In[7]:= Mod[MatrixPower[a, 10], 7]

Out[7]= {{2, 3, 2}, {5, 2, 6}, {5, 4, 1}}
```

Modular matrix inverses use a different technique. For the inverse we again use the `Inverse` command but we add the option of a modulus. To tell Mathematica that we want to invert the matrix over a modulus all we need to do is put in a `Modulus` option at the end of the command. The syntax for this is `Modulus -> m` where m is the desired modulus. The arrow is a common Mathematica notation for setting options, it is created by a − and > characters next to each other. When you type this in, Mathematica will automatically shorten it to a single arrow character. It is possible that the matrix is not invertible with the input modulus, in that case Mathematica will return an error. You can also use the modulus option in the determinant calculation but taking a mod of the determinant is just as easy.

```
In[8]:= Det[a]

Out[8]= -9

In[9]:= Mod[Det[a], 5]

Out[9]= 1

In[10]:= Det[a, Modulus → 5]

Out[10]= 1

In[11]:= Det[a, Modulus → 3]

Out[11]= 0

In[12]:= Inverse[a, Modulus → 5]

Out[12]= {{0, 2, 4}, {0, 2, 2}, {1, 3, 1}}

In[13]:= Inverse[a, Modulus → 3]

         Inverse::sing : Matrix {{1, 0, 1}, {2, 1, 2}, {0, 2, 0}} is singular. >>

Out[13]= Inverse[{{1, 0, 1}, {2, 1, 5}, {3, 2, 0}}, Modulus → 3]
```

Modular matrix reduction can be done the same way, simply include the modulus option inside the `RowReduce` command.

```
In[14]:= RowReduce[m, Modulus → 5] // MatrixForm

Out[14]//MatrixForm=
```

$$\begin{pmatrix} 1 & 0 & 4 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{pmatrix}$$

```
In[15]:= RowReduce[a, Modulus → 5] // MatrixForm

Out[15]//MatrixForm=
```

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

```
In[16]:= RowReduce[a, Modulus → 3] // MatrixForm

Out[16]//MatrixForm=
        1  0  1
        0  1  0
        0  0  0
```

1.7 Elliptic Curves

Many people have created functions for doing calculations on elliptic curves in Mathematica and what is below is certainly not new. The functions we have created below are to help with the experimentation of elliptic curves over a finite modulus. We have created enough functionality to work with Elliptic Curve Cryptography.

All of the Mathematica functions that are discussed in this section are in the `CryptDSEC.nb` Notebook file that can be found on my web site. To load all of the functions download the `CryptDSEC.nb` file then in Mathematica,

1. Open a new Notebook
2. Navigate to the CryptDSEC.nb file.
3. Select it and click Open.
4. From the Menu, evaluate all cells.

At this point all of the functions will be loaded into the Mathematica session. You can open another Notebook and use the new functions without working in the same notebook.

Although a general elliptic curve is represented by

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

we will make the assumption that we can reduce the equation to the form

$$y^2 = x^3 + bx + c$$

We will also be restricting ourselves to modular elliptic curves and hence our function will be working on curves of the form,

$$y^2 = x^3 + bx + c \pmod{n}$$

Most of our functions will take the parameters b, c, and n to define the elliptic curve we are working with.

1.7.1 Points on an Elliptic Curve

To do some basic point finding on an elliptic curve we can use the following functions. The first will find all of the point on the curve. The first function will find all the points except for the point at infinity. The second Function finds all the points including the point at infinity, which we denote as $\{\infty, \infty\}$. The third function finds the total number of points on the elliptic curve including the point at infinity.

```
ECPoints[b_,c_,n_]:=Module[{PtLst, i, j, lhs, rhs},
    PtLst={};
    For[i=0,i<n,i++,
        rhs=Mod[i^3+b*i+c,n];
        For[j=0,j<n,j++,lhs=Mod[j^2,n];
            If[rhs==lhs,PtLst=Append[PtLst,List[i,j]]];
        ]
    ];
    PtLst
]

ECAllPoints[b_,c_,n_]:=Module[{PtLst, i, j, lhs, rhs},
    PtLst={};
    For[i=0,i<n,i++,
        rhs=Mod[i^3+b*i+c,n];
        For[j=0,j<n,j++,
            lhs=Mod[j^2,n];
            If[rhs==lhs,PtLst=Append[PtLst,List[i,j]]];
        ]
    ];
    PtLst=Append[PtLst,List[Infinity,Infinity]];
    PtLst
]

ECOrder[b_,c_,n_]:=Module[{t, i, j, lhs, rhs},
    t=1;
    For[i=0,i<n,i++,
        rhs=Mod[i^3+b*i+c,n];
        For[j=0,j<n,j++,
            lhs=Mod[j^2,n];
            If[rhs==lhs,t++];
        ]
    ];
    t
]
```

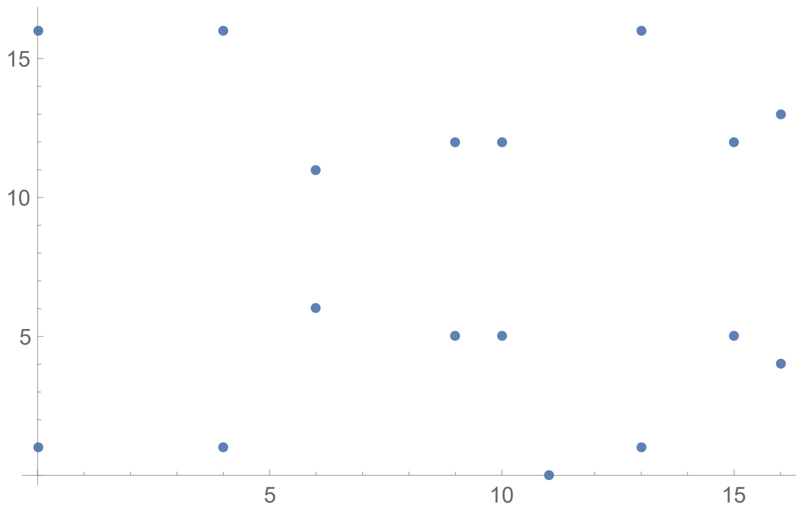
If one takes a quick look at the code it is clear, even if you never programmed in Mathematica, that these are brute force algorithms and hence not the most efficient. So one should be careful with using large moduli.

For example, if we wanted to find the points, all of the points including infinity, and the count of all the points (that is the group order), on the curve $y^2 = x^3 + x + 1 \pmod 5$ we could use the following commands.

```
In[•]:= ECPoints[1, 1, 5]

Out[•]= {{0, 1}, {0, 4}, {2, 1}, {2, 4}, {3, 1}, {3, 4}, {4, 2}, {4, 3}}

In[•]:= ECAllPoints[1, 1, 5]

Out[•]= {{0, 1}, {0, 4}, {2, 1}, {2, 4},
          {3, 1}, {3, 4}, {4, 2}, {4, 3}, {∞, ∞}}

In[•]:= ECOrder[1, 1, 5]

Out[•]= 9
```

Note that the output is a list of pair lists. Each pair is a single point on the curve and the single list notation makes it easy to load into other Mathematica functions. For example, to plot the points on the curve $y^2 = x^3 + x + 1 \pmod{17}$ we could use the following command.

```
In[•]:= ListPlot[ECPoints[1, 1, 17]]

Out[•]=
```

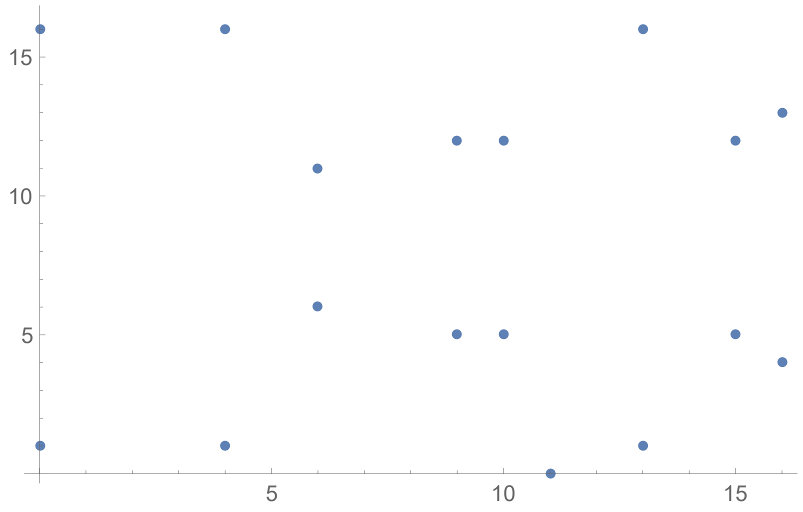


We have also included a function that will do the same thing in one step.

```
ECPlot[b_,c_,n_]:=Module[{},
    ListPlot[ECPoints[b,c,n]]
]
```

So the command `ECPlot[1,1,17]` will produce the same graph.

```
In[•]:= ECPlot[1, 1, 17]
```

Out[•]=


We can check if a point is on a given curve using,

```
ECPointOnCurve[b_,c_,n_,pt_]:=Module[{t, i, j, lhs, rhs,res},
    i=pt[[1]];
    j=pt[[2]];
    rhs=Mod[i^3+b*i+c,n];
    lhs=Mod[j^2,n];
    If[rhs==lhs,res=True,res=False];
    res
]
```

In this function the point we are checking needs to be an ordered pair in a list, just like the output of the point generators. For example,

```
In[•]:= ECPointOnCurve[1, 1, 5, {3, 1}]

Out[•]= True

In[•]:= ECPointOnCurve[1, 1, 5, {1, 1}]

Out[•]= False
```

We can also find points on a curve given either their x or y coordinate. The following functions will return a list of all points on the curve if any exists or an empty list if there is no point on the curve with the given x or y coordinate.

```
ECPointWithX[b_,c_,n_,x_]:=Module[{y,i,rhs, res,sols},
    res={};
    rhs=Mod[x^3+b*x+c,n];
    sols=Values[Solve[y^2==rhs,Modulus->n]];
    For[i=1,i<=Length[sols],i++,
        y=sols[[i,1]];
        res=Append[res,List[x,y]]
    ];
    res
]

ECPointWithY[b_,c_,n_,y_]:=Module[{x,i,lhs, res,sols},
    res={};
    lhs=Mod[y^2,n];
    sols=Values[Solve[x^3+b*x+c==lhs,Modulus->n]];
    For[i=1,i<=Length[sols],i++,
        x=sols[[i,1]];
        res=Append[res,List[x,y]]
    ];
    res
]
```

For example, if we wanted to find points on $y^2 = x^3 + x + 1 \pmod 5$ we could use the following commands.

```
In[•]:= ECPointWithX[1, 1, 5, 2]

Out[•]= {{2, 1}, {2, 4}}

In[•]:= ECPointWithX[1, 1, 5, 1]

Out[•]= {}

In[•]:= ECPointWithY[1, 1, 5, 1]

Out[•]= {{0, 1}, {2, 1}, {3, 1}}

In[•]:= ECPointWithY[1, 1, 5, 0]

Out[•]= {}
```

Note that we have also included brute force algorithms to do this. They have BF at the end of the function name and will return just the first point that is found. Also you will want to use a modulus of a moderate size. The above functions are far more efficient but in case you need these they have been included.

```
ECPointWithXBF[b_,c_,n_,x_]:=Module[{i, lhs, rhs, res},
    res={};
    rhs=Mod[x^3+b*x+c,n];
    i = 0;
    While[i<n,
        lhs=Mod[i^2,n];
        If[rhs==lhs,res={x,i};i = n];
        i++;
    ];
    res
]

ECPointWithYBF[b_,c_,n_,y_]:=Module[{i, j, lhs, rhs, res},
    res={};
    rhs=Mod[y^2,n];
    i = 0;
    While[i<n,
        lhs=Mod[i^3+b*i+c,n];
        If[rhs==lhs,res={i,y};i = n];
        i++;
    ];
    res
]
```

In Elliptic Curve Cryptography it is common to select the linear term and modulus of the curve and a particular point you want on the curve and then calculate the constant term from this information. While this is a simple calculation we created a function to do this.

```
ECGenerateCurveConstant[b_,n_,pt_]:=Module[{},
    Mod[pt[[2]]^2-(pt[[1]]^3+b*pt[[1]]),n]
]
```

For example, say we wanted the point $(7657, 74389)$ to be on the curve with linear term 3284 and modulus 3263561.

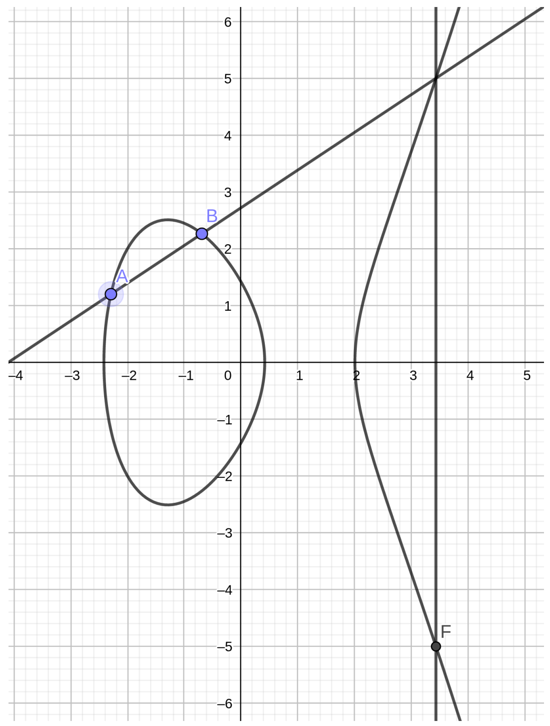
```
In[•]:= ECGenerateCurveConstant[3284, 3263561, {7657, 74389}]

Out[•]= 1388410
```

```
In[•]:= ECPointOnCurve[3284, 1388410, 3263561, {7657, 74389}]

Out[•]= True
```

We find that the curve $y^2 = x^3 + 3284x + 1388410 \pmod{3263561}$ does the trick.

1.7.2 Arithmetic on an Elliptic Curve

If you have studied elliptic curves you know that there is a method to add two points on an elliptic curve to obtain a third point on the curve. In fact, if you have studied group theory you know that this point addition defines an abelian group structure on the curve. In the case of finite groups this structure is sometimes cyclic. Although we will be dealing with curve modulo n we will briefly discuss the addition law geometrically for elliptic curves in R^2. The addition law in this case has a nice geometrical interpretation that also sheds some light on the formulas.

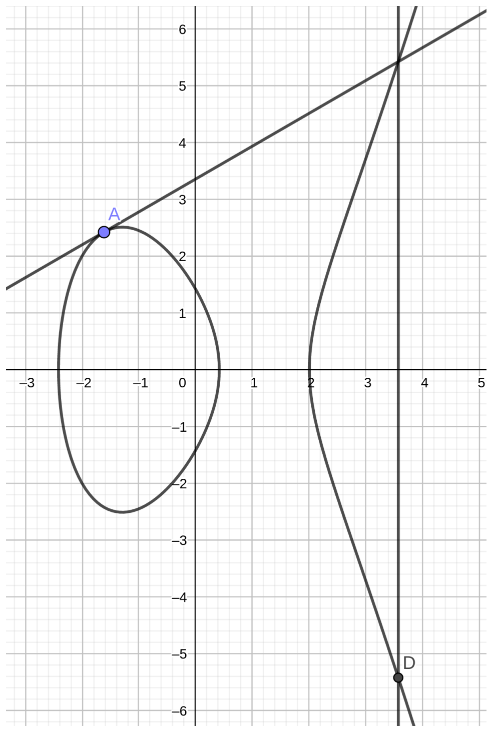


To add two points on an elliptic curve A and B where $A \neq B$ you first draw a straight line through the two points, this will intersect the curve in another point. Then reflect this point over the x axis to obtain the sum of A and B. So in the diagram on the right we have $A + B = F$.

In the case where $A = B$, in other words we want to calculate $2A$ we take the tangent line to the elliptic curve at A, this will intersect the curve in another point. Then reflect this point over the x axis to obtain $2A$. So in the diagram on the right we have $2A = D$.

In the cases where the line through A and B is vertical or if the tangent line in vertical when calculating $2A$ the sum is the point at infinity, ∞. If we translate this geometric description into algebraic formulas we have the following Addition Law on elliptic curves.

Let E be given by $y^2 = x^3 + bx + c$ and let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then $P_1 + P_2 = P_3 = (x_3, y_3)$ where



$$\begin{aligned} x_3 &= m^2 - x_1 - x_2 \\ y_3 &= m(x_1 - x_3) - y_1 \end{aligned}$$

and

$$m = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2 \\ \frac{3x_1^2 + b}{2y_1} & \text{if } P_1 = P_2 \end{cases}$$

If the slope m is infinite, then $P_3 = \infty$. There is one additional law: $\infty + P = P$ for all points P.

Although these equations were developed using continuous curves and derivatives the same formulas work for the discrete case of finite curves over a modulus. The tricky point here is that in the derivation of m we have either $x_2 - x_1$ or $2y_1$ in the denominator. So if we are working modulo n these values need to have multiplicative inverses modulo n. If they do not have a multiplicative inverse modulo n then the greatest common divisor between them and n is greater then 1 and in some cases this will lead to a factorization of n.

We have created four Mathematica functions to do some arithmetic operations. The first is a point addition function. This function will return the sum of the input points if it exists and if not it will return -1.

```
ECPointAdd[b_,c_,n_,pt1_,pt2_]:=Module[{x,y,invy,m},
    If[pt1==-1,Return[-1]];
    If[pt2==-1,Return[-1]];
    If[pt1[[1]]==Infinity,Return[pt2]];
    If[pt2[[1]]==Infinity,Return[pt1]];
    If[pt1==pt2,
        If[Mod[2*pt1[[2]],n]==0,Return[List[Infinity,Infinity]]];
        If[GCD[2*pt1[[2]],n]>1,Return[-1]];
        invy=ModularInverse[2*pt1[[2]],n];
        m=Mod[(3*pt1[[1]]^2+b)*invy,n];,
        If[Mod[pt1[[1]],n]==Mod[pt2[[1]],n],Return[List[Infinity,Infinity]]];
        If[GCD[pt1[[1]]-pt2[[1]],n]>1,Return[-1]];
        invy=ModularInverse[pt1[[1]]-pt2[[1]],n];
        m=Mod[(pt1[[2]]-pt2[[2]])*invy,n];
    ];
    x=Mod[m^2-pt1[[1]]-pt2[[1]],n];
    y=Mod[m*(pt1[[1]]-x)-pt1[[2]],n];
    List[x,y]
]
```

For example, say we wanted to add the two points $(2, 1)$ and $(4, 2)$ on the elliptic curve $y^2 = x^3 + x + 1 \pmod 5$. We see that the result is the point $(3, 1)$. Additionally, $(2, 1) +$

$(2,4) = \infty$ and $2 \cdot (3,1) = (0,1)$. Also, if we were to add the two points $(1,3)$ and $(1771,705)$ on the elliptic curve $y^2 = x^3 + 4x + 4 \pmod{2773}$. We see that the result is an error. This is because the GCD of $x_2 - x_1 - 1770$ and the modulus 2773 is 59 and hence 1770 is not invertible modulo 2773. The added information is that 59 is a nontrivial factor of 2773. This also shows that if the modulus is not prime (that is the base structure is not a field) then the resulting curve with the addition law does not form a group structure, addition is not closed. It is precisely this fact that is the driver of Lenstra's Elliptic Curve Factorization algorithm.

```
In[•]:= ECPointAdd[1, 1, 5, {2, 1}, {4, 2}]

Out[•]= {3, 1}

In[•]:= ECPointAdd[1, 1, 5, {2, 1}, {2, 4}]

Out[•]= {∞, ∞}

In[•]:= ECPointAdd[1, 1, 5, {3, 1}, {3, 1}]

Out[•]= {0, 1}

In[•]:= ECPointAdd[4, 4, 2773, {1, 3}, {1771, 705}]

Out[•]= -1
```

We have created two functions for doing scalar multiplication, the first calculates $t \cdot P$ and the second calculates $t! \cdot P$. The scalar multiple function uses a binary decomposition of the scalar and hence is very fast but the factorial scalar multiple needs to run through each scalar multiple and can be slow for large values of t.

```
ECPointScalarMult[b_,c_,n_,m_,pt1_]:=Module[{retpt,newy, t, pt},
    If[pt1==-1,Return[-1]];
    retpt=List[Infinity,Infinity];
    t = m;
    pt=pt1;
    If[t < 0,t=-t;newy=Mod[-pt[[2]],n];pt=List[pt[[1]],newy]];
    While[t > 0,
        If[Mod[t,2]==1,retpt=ECPointAdd[b,c,n,retpt,pt]];
        pt=ECPointAdd[b,c,n,pt,pt];
        t=Floor[t/2];
    ];
    retpt
]

ECPointFactorialScalarMult[b_,c_,n_,m_,pt1_]:=Module[{pt,i},
    pt=pt1;
    For[i=2,i<=m,i++,pt=ECPointScalarMult[b,c,n,i,pt]];
    pt
]
```

For example, say we wanted to calculate $5 \cdot (13,4)$, $738956431 \cdot (13,4)$, $5! \cdot (13,4)$, and $20! \cdot (13,4)$ on the curve $y^2 = x^3 + 2x + 3 \pmod{17}$. The following commands will do these calculations.

```
In[•]:= ECPointScalarMult[2, 3, 17, 5, {13, 4}]

Out[•]= {9, 6}
```

```
In[•]:= ECPointScalarMult[2, 3, 17, 738956431, {13, 4}]

Out[•]= {5, 11}

In[•]:= ECPointFactorialScalarMult[2, 3, 17, 5, {13, 4}]

Out[•]= {3, 6}

In[•]:= ECPointFactorialScalarMult[2, 3, 17, 20, {13, 4}]

Out[•]= {∞, ∞}
```

As we pointed out above, elliptic curves with prime modulus form a group structure. In group theory the order of an element is of some importance. We have included another brute force algorithm to calculate the order of a point on the elliptic curve.

```
ECPointOrder[b_,c_,n_,pt1_]:=Module[{t, pt,i,r},
    If[pt1==-1,Return[-1]];
    retpt=List[Infinity,Infinity];
    t = True;
    i=1;
    While[t,
        pt=ECPointScalarMult[b,c,n,i,pt1];
        If[pt==-1,Return[-1]];
        If[pt[[1]]==Infinity,r=i;t=False];
        i++;
    ];
    r
]
```

For example, calculate the order of $(13, 4)$ on the curve $y^2 = x^3 + 2x + 3 \pmod{17}$ we do the following.

```
In[•]:= ECPointOrder[2, 3, 17, {13, 4}]

Out[•]= 22
```

As another example, if we calculate the order of $(1, 3)$ on the curve $y^2 = x^3 + 4x + 4 \pmod{2773}$, which does not exist since this curve does not create a group structure and the point $(1, 3)$ never cycles back to the identity, we get the following.

```
In[•]:= ECPointOrder[4, 4, 2773, {1, 3}]

Out[•]= -1
```

1.8 CryptDSEC.nb

The Mathematica functions that were discussed in this section are in the `CryptDSEC.nb` file that can be found on my web site. To load all of the functions download the `CryptDSEC.nb` file then in Mathematica,

1. Open a new Notebook
2. Navigate to the CryptDSEC.nb file.

3. Select it and click Open.

4. From the Menu, evaluate all cells.

At this point all of the functions will be loaded into the Mathematica session. You can open another Notebook and use the new functions without working in the same notebook.

1.8.1 CryptDSEC.nb Code

```
ECPoints[b_,c_,n_]:=Module[{PtLst, i, j, lhs, rhs},
    PtLst={};
    For[i=0,i<n,i++,
        rhs=Mod[i^3+b*i+c,n];
        For[j=0,j<n,j++,
            lhs=Mod[j^2,n];
            If[rhs==lhs,PtLst=Append[PtLst,List[i,j]]];
        ]
    ];
    PtLst
]

ECAllPoints[b_,c_,n_]:=Module[{PtLst, i, j, lhs, rhs},
    PtLst={};
    For[i=0,i<n,i++,
        rhs=Mod[i^3+b*i+c,n];
        For[j=0,j<n,j++,
            lhs=Mod[j^2,n];
            If[rhs==lhs,PtLst=Append[PtLst,List[i,j]]];
        ]
    ];
    PtLst=Append[PtLst,List[Infinity,Infinity]];
    PtLst
]

ECOrder[b_,c_,n_]:=Module[{t, i, j, lhs, rhs},
    t=1;
    For[i=0,i<n,i++,
        rhs=Mod[i^3+b*i+c,n];
        For[j=0,j<n,j++,
            lhs=Mod[j^2,n];
            If[rhs==lhs,t++];
        ]
    ];
    t
]

ECPlot[b_,c_,n_]:=Module[{},
    ListPlot[ECPoints[b,c,n]]
]

ECPointOnCurve[b_,c_,n_,pt_]:=Module[{t, i, j, lhs, rhs,res},
    i=pt[[1]];
    j=pt[[2]];
    rhs=Mod[i^3+b*i+c,n];
    lhs=Mod[j^2,n];
    If[rhs==lhs,res=True,res=False];
    res
]

ECPointWithXBF[b_,c_,n_,x_]:=Module[{i, lhs, rhs, res},
    res={};
    rhs=Mod[x^3+b*x+c,n];
    i = 0;
    While[i<n,
        lhs=Mod[i^2,n];
        If[rhs==lhs,res={x,i};i = n];
        i++;
    ];
    res
]

ECPointWithYBF[b_,c_,n_,y_]:=Module[{i, j, lhs, rhs, res},
    res={};
    rhs=Mod[y^2,n];
    i = 0;
    While[i<n,
        lhs=Mod[i^3+b*i+c,n];
        If[rhs==lhs,res={i,y};i = n];
        i++;
    ];
    res
]

ECGenerateCurveConstant[b_,n_,pt_]:=Module[{},
    Mod[pt[[2]]^2-(pt[[1]]^3+b*pt[[1]]),n]
]

ECPointWithX[b_,c_,n_,x_]:=Module[{y,i,rhs, res,sols},
    res={};
    rhs=Mod[x^3+b*x+c,n];
    sols=Values[Solve[y^2==rhs,Modulus->n]];
    For[i=1,i<=Length[sols],i++,
        y=sols[[i,1]];
        res=Append[res,List[x,y]]
    ];
    res
]

ECPointWithY[b_,c_,n_,y_]:=Module[{x,i,lhs, res,sols},
    res={};
    lhs=Mod[y^2,n];
    sols=Values[Solve[x^3+b*x+c==lhs,Modulus->n]];
    For[i=1,i<=Length[sols],i++,
        x=sols[[i,1]];
        res=Append[res,List[x,y]]
    ];
    res
]

ECPointAdd[b_,c_,n_,pt1_,pt2_]:=Module[{x,y,invy,m},
    If[pt1==-1,Return[-1]];
    If[pt2==-1,Return[-1]];
    If[pt1[[1]]==Infinity,Return[pt2]];
    If[pt2[[1]]==Infinity,Return[pt1]];
    If[pt1==pt2,
        If[Mod[2*pt1[[2]],n]==0,Return[List[Infinity,Infinity]]];
        If[GCD[2*pt1[[2]],n]>1,Return[-1]];
        invy=ModularInverse[2*pt1[[2]],n];
        m=Mod[(3*pt1[[1]]^2+b)*invy,n];,
        If[Mod[pt1[[1]],n]==Mod[pt2[[1]],n],Return[List[Infinity,Infinity]]];
        If[GCD[pt1[[1]]-pt2[[1]],n]>1,Return[-1]];
        invy=ModularInverse[pt1[[1]]-pt2[[1]],n];
        m=Mod[(pt1[[2]]-pt2[[2]])*invy,n];
    ];
    x=Mod[m^2-pt1[[1]]-pt2[[1]],n];
    y=Mod[m*(pt1[[1]]-x)-pt1[[2]],n];
    List[x,y]
]
```

```
ECPointScalarMult[b_,c_,n_,m_,pt1_]:=Module[{retpt,newy, t, pt},
    If[pt1==-1,Return[-1]];
    retpt=List[Infinity,Infinity];
    t = m;
    pt=pt1;
    If[t < 0,t=-t;newy=Mod[-pt[[2]],n];pt=List[pt[[1]],newy]];
    While[t > 0,
        If[Mod[t,2]==1,retpt=ECPointAdd[b,c,n,retpt,pt]];
        pt=ECPointAdd[b,c,n,pt,pt];
        t=Floor[t/2];
    ];
    retpt
]

ECPointFactorialScalarMult[b_,c_,n_,m_,pt1_]:=Module[{pt,i},
    pt=pt1;
    For[i=2,i<=m,i++,pt=ECPointScalarMult[b,c,n,i,pt]];
    pt
]

ECPointOrder[b_,c_,n_,pt1_]:=Module[{t, pt,i,r},
    If[pt1==-1,Return[-1]];
    retpt=List[Infinity,Infinity];
    t = True;
    i=1;
    While[t,
        pt=ECPointScalarMult[b,c,n,i,pt1];
        If[pt==-1,Return[-1]];
        If[pt[[1]]==Infinity,r=i;t=False];
        i++;
    ];
    r
]
```

Chapter 2

Introduction to Maxima

2.1 What is Maxima?

Maxima is an open-source computer algebra system, the following description was taken from the Maxima project site at sourceforge (http://maxima.sourceforge.net/).[3] Computer algebra systems are programs that are capable of doing exact mathematical computations in a wide range of mathematical subjects. That is, they can solve equations producing exact answers as opposed to giving decimal approximations. They can do symbolic algebra, trigonometry, calculus, differential equations, and so on. Some computer algebra systems have very specific uses, such as finite group theory, while others are built to be more comprehensive. Maxima is one of the most comprehensive open-source computer algebra systems.

> Maxima is a system for the manipulation of symbolic and numerical expressions, including differentiation, integration, Taylor series, Laplace transforms, ordinary differential equations, systems of linear equations, polynomials, sets, lists, vectors, matrices and tensors. Maxima yields high precision numerical results by using exact fractions, arbitrary-precision integers and variable-precision floating-point numbers. Maxima can plot functions and data in two and three dimensions.
>
> The Maxima source code can be compiled on many systems, including Windows, Linux, and MacOS X. The source code for all systems and precompiled binaries for Windows and Linux are available at the SourceForge file manager.
>
> Maxima is a descendant of Macsyma, the legendary computer algebra system developed in the late 1960s at the Massachusetts Institute of Technology. It is the only system based on that effort still publicly available and with an active user community, thanks to its open source nature. Macsyma was revolutionary in its day, and many later systems, such as Maple and Mathematica, were inspired by it.
>
> The Maxima branch of Macsyma was maintained by William Schelter from 1982 until he passed away in 2001. In 1998 he obtained permission to release the source code under the GNU General Public License (GPL). It was his efforts and

> skill which have made the survival of Maxima possible, and we are very grateful to him for volunteering his time and expert knowledge to keep the original DOE Macsyma code alive and well. Since his death, a group of users and developers has formed to bring Maxima to a wider audience.
>
> Maxima is updated very frequently, to fix bugs and improve the code and the documentation. We welcome suggestions and contributions from the community of Maxima users. Most discussion is conducted on the Maxima mailing list.

You can download Maxima from the Maxima project site at sourceforge,

http://maxima.sourceforge.net/

This introduction to Maxima is not designed to be a general introduction to the software package. There are far better resources for that online than I could ever hope to write. Here we simply concentrate on what you need to do the cryptography exercises and examples in this set of notes.

As with all computer algebra systems, there are numerous ways to input your calculations to obtain the desired results, some methods are slicker than others. The downside of the slick methods is that they are usually hard to read and unless you are already familiar with the ins and outs of the system it is usually unclear what is happening. Since we are assuming that you have a limited exposure to Maxima, we do not always take the slickest route to produce the needed calculation. In many cases we will break a calculation down into several steps, where is could be done in a single command. This is done for readability and clarity of the operation. As you become more acquainted with Maxima you will see other equivalent methods to those in this set of notes.

If you are familiar with Maxima you probably already know everything in this introduction. In this case you may want to simply skim over these pages and read the unfamiliar sections.

There are several Maxima scripts that we discuss in this section that do specific operations on matrices and elliptic curves which are not included in the Maxima CAS. These can all be found in the `CryptDS.mac` file that is on my web site. To load all of the functions download the `CryptDS.mac` file then in Maxima,

1. Select File > Load Package from the main menu.
2. Navigate to the CryptDS.mac file.
3. Select it and click Open.

At this point all of the functions will be loaded into the Maxima session.

2.2 The User Interface

Most computer algebra systems have very similar interfaces. There is usually a graphical interface for command input that is where the user enters their calculation commands and a calculation engine the background where the calculations are performed. There are many different graphical interfaces that link up with Maxima. The one pictured below is wxMaxima , which is available on most major platforms. If you are using a different user interface then your screen will, of course, look different and the menu options we discuss here may or may not be available in your program.

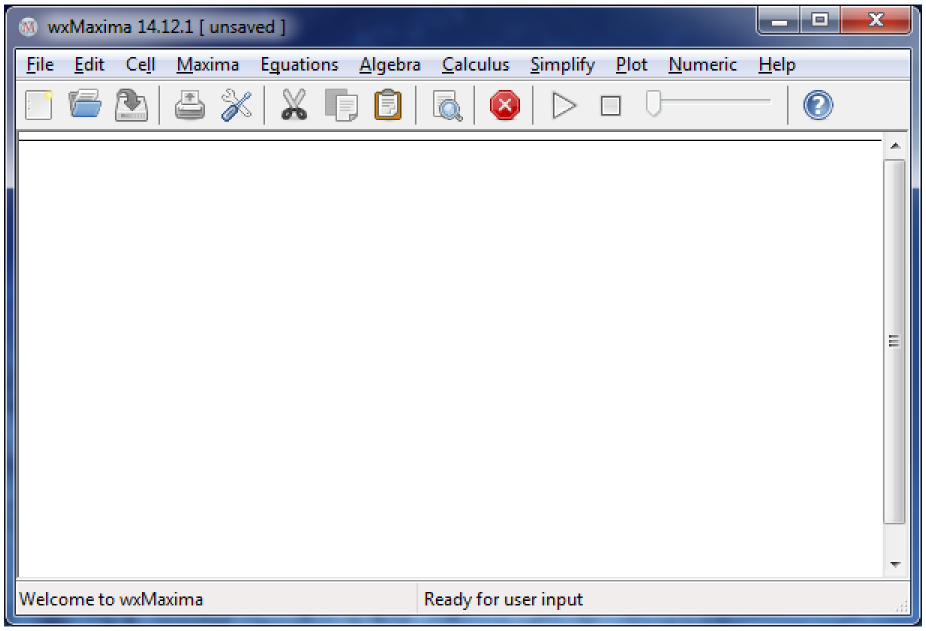
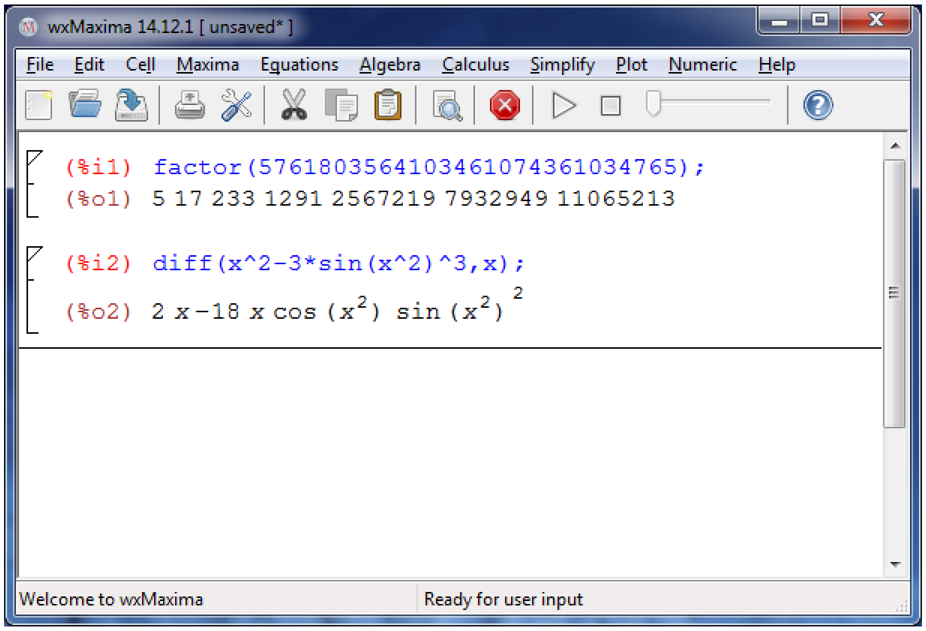

wxMaxima 14.12.1 [unsaved]
File Edit Cell Maxima Equations Algebra Calculus Simplify Plot Numeric Help
Welcome to wxMaxima
Ready for user input

Figure 2.1: User Interface to Maxima

When the user types in a command and sends it for calculation, the command is transferred to the engine, calculated there, and the result is transferred back to the graphical interface. The engine operations are hidden from the user and you will probably never need to deal with the engine but the reason we are going into this is that on occasions something, usually external to Maxima, causes the interface to lose the communication link with the engine. This happens rarely but if you notice that Maxima is not doing the calculations you send it and you know you are using the correct syntax then you may have lost the engine link. In these cases, there is a menu option, under the Maxima menu, to restart the Maxima engine. Selecting this should reestablish the link for you. Another option is to close Maxima and restart it, the good old reboot solution.

As you can see from the above image, the “In” lines (%i1) are what the user has input into Maxima and the “Out” lines (%o1) are Maxima’s responses to the inputs. On the first line we simply asked Maxima to factor a number for us. The one of the Maxima commands for this is factor followed by parentheses containing the number to be factored.

This in and out tracking comes in handy when you want to use a previous input or output.


wxMaxima 14.12.1 [unsaved*]
File Edit Cell Maxima Equations Algebra Calculus Simplify Plot Numeric Help
Welcome to wxMaxima
Ready for user input

```
(%i1) factor(5761803564103461074361034765);
(%o1) 5 17 233 1291 2567219 7932949 11065213

(%i2) diff(x^2-3*sin(x^2)^3,x);
(%o2) 2 x-18 x cos (x^2) sin (x^2)^2
```

Figure 2.2: User Interface to Maxima with Commands

The % will automatically take the last output that was done. Be careful here, this is not always the output right above the new input. For example, if you go up several commands and reevaluate a command, that is the last output. You can also use the %o1 notation for output number 1, %o2 for output number 2, and so on. You will notice that if you redo a command, it will be renumbered with a different input and output number, the original input and output number still have the same values.

(%i1) `diff(x^2,x);`

(%o1) $2 \cdot x$

(%i2) `%o1*5;`

(%o2) $10 \cdot x$

(%i3) `%^2;`

(%o3) $100 \cdot x^2$

(%i4) `%o2;`

(%o4) $10 \cdot x$

Then if we reevaluate the second input it is labeled number 5 but then if we add a new entry that references number 2 (%o2+7) it uses the second output from the session.

(%i1) `diff(x^2,x);`

(%o1) $2 \cdot x$

```
(%i5) %o1*5;
```

(%o5) $10 \cdot x$

```
(%i3) %^2;
```

(%o3) $100 \cdot x^2$

```
(%i4) %o2;
```

(%o4) $10 \cdot x$

```
(%i6) %o2+7;
```

(%o6) $10 \cdot x + 7$

It is better practice to assign an output to a variable and use the variable name when needed, for example,

```
(%i1) D:diff(x^2,x);
```

(%o1) $2 \cdot x$

```
(%i2) 3*D^2;
```

(%o2) $12 \cdot x^2$

We will discuss assigning variables in more detail in the next section.

All Maxima commands are lowercase and multi-word commands usually separate the words with an underscore. When applying a command to some input, the input is surrounded by parentheses, like we would write $f(x)$ to apply the function f to the input x. In Maxima parentheses are used as grouping symbols for expressions as well, square brackets are to delimit lists, and curly brackets are used to delimit sets. Matrices are treated a little differently in Maxima than they are in Mathematica. We will discuss matrix syntax in the section on matrices.

Table 2.1: Brackets in Maxima

Bracket	Usage
( )	Grouping and Function Input
[ ]	Lists
{ }	Sets

Once you have input a command you send it to the engine for evaluation by selecting Shift + Enter from the keyboard. Also, if your keyboard has a keypad, then simply selecting the keypad Enter (with no Shift) will work as well. When you do this, there may be a slight to a long pause while the calculation is being done and then the result will be displayed in the out line. If a calculation is taking too long to complete you can abort (Interrupt) the calculation either from the Maxima menu, click on the interrupt stop sign on the menu bar, or by typing Ctrl+G from the keyboard.

As we pointed out above, computer algebra systems do exact arithmetic, unless otherwise told. So the user can easily input something into the computer algebra system that the computer will not be able to handle or not able to handle in a reasonable amount of time. For example, asking the computer to calculate 1000000! or asking it to factor the 600 digit semiprime that Amazon uses for customer purchases. So if Maxima is taking a very long time to do a calculation, make sure you did not inadvertently ask it a bad question, and if you did, abort the calculation.

Maxima also has a couple command assistant interfaces, one is through the menu system and the other is through Panes . The Panes are a quick way to call some of the menu commands. With the Panes or the menu commands the command is applied to the last output if the command does not need any input and if it does, a dialog box will appear asking for user input on the command. If you find these panes or menu options to be useful then by all means use them. These notes will be concentrating on the commands you need to aide you in cryptography calculations, so we will not be using Maxima's pane system, and seldom using the menu options. Most of the pane operations are self-explanatory and there are numerous guides to using them on the Internet if you are interested. The pane and menu systems are a nice way to get started with Maxima, to learn some of its functions and syntax. Once you are familiar with Maxima you will probably find that typing in the commands is quicker.

Maxima does have a substantial help system, although it is not as nice as the one in Mathematica. There are examples for most commands, cross links for related topics, descriptions of the available options for a command and descriptions of what the command does and in some cases descriptions of the mathematical methods and algorithms used in the calculation.

2.3 Basic Calculations

2.3.1 Numeric Calculations

When starting out with any computer algebra system it is good to treat it simply as a fancy calculator, just to get the feel for how it works and basic expression format. Addition, subtraction, multiplication, division and powers are done with the standard mathematics symbols `+-*/^` as you would expect. There are several basic numerical types used in Maxima but most of the time we will be working in either exact mode or approximate mode. Maxima has a couple different options in approximate mode that we will look at a little later.

Computer algebra systems use exact mode whenever possible, this is how they are constructed and frankly what their main purpose is. When calculations are done in exact mode the outputs are integers, rational numbers or expressions involving them. Approximate mode is when we have decimal approximations as our output. In the example below, inputs 1–4 are all in exact mode, note the $\sqrt{2}$ and $\log 25$. Since these numbers are irrational Maxima will not approximate them. Inputs 5 and 6 produce approximate outputs since we used a decimal in the input expression.

```
(%i1) 123+456;
```

(%o1) 579

```
(%i2) 2^(1/2);
```

(%o2) $\sqrt{2}$

```
(%i3) 6463629734643562112/21854943257648216491454;
```

(%o3) $\frac{3231814867321781056}{10927471628824108245727}$

```
(%i4) log(25);
```

(%o4) $\log(25)$

```
(%i5) log(25.0);
```

(%o5) 3.218875824868201

```
(%i6) 2.0^(1/2);
```

(%o6) 1.414213562373095

So the easiest way to force Maxima into approximation mode is to use decimal numbers in the expression. You can also use a couple commands to convert an exact expression into an approximate expression. The `float` command and `numer` option will convert the expression to decimal and the `bfloat` command will convert the expression into a "Bigfloat", this is used if you want to see more decimal places to the approximation. In cryptography, we usually deal primarily with integers so there will be few times when we need to get approximations. Nonetheless, here are some examples,

```
(%i1) 2^(1/2);
```

(%o1) $\sqrt{2}$

```
(%i2) float(%);
```

(%o2) 1.414213562373095

```
(%i3) float(2^(1/2));
```

(%o3) 1.414213562373095

```
(%i4) 2^(1/2), numer;
```

(%o4) 1.414213562373095

```
(%i5) bfloat(2^(1/2));
(%o5) 1.414213562373095b0
(%i6) fpprec : 200;
(%o6) 200
(%i7) bfloat(2^(1/2));
(%o7) 1.4142135623730950488016887242[143digits]5927557999505011527820605715b0
(%i8) set_display('ascii)$
(%i9) bfloat(2^(1/2));
(%o9) 1.41421356237309504880168872420969807856967187537694807317667973799073247
846210703885038753432764157273501384623091229702492483605585073721264412149709
9935831413222665927505592755799950501152782060571 5b0
(%i10) set_display('xml)$
(%i11) bfloat(2^(1/2));
(%o11) 1.4142135623730950488016887242[143digits]5927557999505011527820605715b0
```

Notice that the big float (`bfloat` command) produces the same number of decimal places as the float command, unless you set `fpprec` to a larger number. The `fpprec` internal variable controls the floating point precision of the session. Also note with the big float there is a $b0$ at the end. This is read just like the $e0$ scientific notation you get in other programs and calculators.

Maxima has several ways to display lines that are too long or other multi-line outputs. In wxMaxima, the default is xml mode which in some cases produces a statement like, [143digits] in the middle of a number. This can be changed by either going into ascii or none mode . Modes can be changed with the `set_display` command. The wxMaxima interface has menu options for changing the precision, conversion to floats and bfloats, and changing the 2d display. The conversions to floats and bigfloats as well as precision changing are under the Numeric menu and changing the 2d display is under the Maxima menu. The above examples show the 2d display for xml and ascii, the none display option will put the entire number on a single line, with no elimination of the middle of the number.

2.3.2 Algebra

Computer algebra systems will also do algebra, imagine that. So they will do computations with variables just as we would. One thing to be careful with here is assigning values to variables. Once a variable is assigned a value it will replace the variable with that value in all expressions until the variable is reset. Assignment is done with the colon, so the command `x:3` assigns the value 3 to the variable x. Then any expression with an x in it is evaluated as if x is the value 3. The command, `x:'x` resets x back to x.

```
(%i1) x^2+3*x-2*x^2+x-5;
```

(%o1) $-x^2+4\cdot x-5$

```
(%i2) x:3;
```

(%o2) 3

```
(%i3) x^2+3*x-2*x^2+x-5;
```

(%o3) -2

```
(%i4) x:'x;
```

(%o4) x

```
(%i5) x^2+3*x-2*x^2+x-5;
```

(%o5) $-x^2+4\cdot x-5$

Also note that for multiplication we must use the $\star$ symbol, juxtaposition is not supported in Maxima . Also note that expressions are automatically simplified, that is the easy simplifications are done automatically. More complex expressions will not be simplified until you give Maxima a command to do so. Maxima has over 20 different simplification commands. At first this is a bit overwhelming and confusing on which one to use when, but once you see some of the differences in the outputs you will be glad to have this flexibility instead of a single simplify command that may or may not produce something you want. Also, some of the different simplification commands work on specific types of expressions, such as logarithms, trigonometric functions, or complex valued expressions. In wxMaxima, the simplification commands can be invoked from the Simplify menu.

```
(%i1) ((x+1)^2*(x-1)^2)/(x^2-1);
```

(%o1) $\dfrac{(x-1)^2\cdot(x+1)^2}{x^2-1}$

```
(%i2) ratsimp(%);
```

(%o2) x^2-1

The `ratsimp` command above is the one you will probably use the most, it simplifies expressions and subexpressions and puts the result into a rational type form. You can find a more specific description in the Maxima help system.

2.3.3 Execution Timing

In cryptography, and other computationally intensive areas in mathematics and computing, one wants to know how different algorithms that accomplish the same task stack up against each other. Which algorithm factors integers the fastest or finds the discrete logarithm fastest? Or better questions are which algorithms are fastest in which situations? The way this is usually done, theoretically, is by counting the number of mathematical operations that need to be done for the algorithm to come up with a solution. We tend to look at best, average, and worst case scenarios and compare them.

Another method is to do empirical testing. Run several examples using each algorithm and compare the timings. With computer algebra systems, many complex tasks, such as factoring and finding discrete logarithms will implement several different algorithms that work together, and even in parallel. So separating them is sometime difficult. Nonetheless, we would still like to know execution times for processes run on Maxima.

```
(%i1) factor(645317065016047164638196537311111);
```

(%o1) $17 \cdot 1376431130111329 \cdot 27578442919127277$

```
(%i2) if showtime#false then showtime:false else showtime:all$
```

Evaluation took 0.0000 seconds (0.0000 elapsed) using 184 bytes.

```
(%i3) factor(645317065016047164638196537311111);
```

Evaluation took 3.9610 seconds (3.9940 elapsed) using 142.667 MB.
(%o3) $17 \cdot 1376431130111329 \cdot 27578442919127277$

In wxMaxima, under the Maxima menu, there is an option to toggle the time display. When turned on, Maxima will display the amount of execution time and the amount of memory used to complete the process.

2.4 Defining Functions

Maxima has hundreds of built-in functions, trigonometric, logarithmic, hyperbolic, complex valued, exponential, combinatorial, ... In cryptography, we do not tend to need transcendental functions too often and we will look at a few discrete mathematics and number theory functions in the following sections and throughout the body of these notes. There will be times when you will want to define your own functions, this tends to make typing and expression syntax easier when you are dealing with longer expressions. In Maxima, to define a function, start with the function name, a list of variables in parentheses, `:=` and then the expression. For example, to define the function $f(x) = x^2 - 3x + 5$,

```
(%i1) f(x):=x^2-3*x+5;
```

(%o1) $\mathrm{f}(x) := x^2 - 3 \cdot x + 5$

```
(%i2) f(t);
```

(%o2) $t^2 - 3 \cdot t + 5$

```
(%i3) f(5);
```

(%o3) 15

```
(%i4) f(-x);
```

(%o4) $x^2 + 3 \cdot x + 5$

```
(%i5) f(x+h);
```

(%o5) $(x + h)^2 - 3 \cdot (x + h) + 5$

After the function is defined, you can evaluate the function at values, or expressions, by placing the value or expression in the parentheses, just like we would do in mathematics. Functions can be defined on more than one variable, for example,

```
(%i1) g(x, y):= x^2-y^2;
```

(%o1) $\mathrm{g}(x, y) := x^2 - y^2$

```
(%i2) g(2, 3);
```

(%o2) -5

```
(%i3) g(t, 5);
```

(%o3) $t^2 - 25$

```
(%i4) g(17, x+h);
```

(%o4) $289 - (x + h)^2$

We will discuss Maxima lists later in these notes but will give a quick example here. Most computer algebra systems store and manipulate information in lists, this is the basis to what are called functional programming languages, like LISP. So computer algebra systems tend to work very efficiently on lists. In Maxima, a list is a set of expressions separated by commas and delimited by square brackets. The following is an example of how you can evaluate a function on a list.

```
(%i1) g(x, y):= x^2-y^2;
```

(%o1) $\mathrm{g}(x, y) := x^2 - y^2$

```
(%i2) g([1,2,3], 7);
```

(%o2) $[-48, -45, -40]$

```
(%i3) g(1, 7);
```

(%o3) -48

```
(%i4) g(2, 7);
```

(%o4) -45

```
(%i5) g(3, 7);
```

(%o5) -40

```
(%i6) g([1,2,3], [7, 8, 9]);
```

(%o6) $[-48, -60, -72]$

```
(%i7) g(2, 8);
```

(%o7) -60

```
(%i8) g(3, 9);
```

(%o8) -72

Functions can also be composed with each other and themselves. Furthermore, you can define a function using other function definitions.

```
(%i1) f(x):=sqrt(x+1);
```

(%o1) $\mathrm{f}(x) := \sqrt{x+1}$

```
(%i2) f(f(x));
```

(%o2) $\sqrt{\sqrt{x+1}+1}$

```
(%i3) f(f(f(x)));
```

(%o3) $\sqrt{\sqrt{\sqrt{x+1}+1}+1}$

```
(%i4) f(f(f(f(x))));
```

(%o4) $\sqrt{\sqrt{\sqrt{\sqrt{x+1}+1}+1}+1}$

```
(%i5) f(f(f(f(2))));
```

(%o5) $\sqrt{\sqrt{\sqrt{\sqrt{3}+1}+1}+1}$

```
(%i6) h(x):=f(f(f(f(x))));
```

(%o6) $\mathrm{h}(x) := \mathrm{f}(\mathrm{f}(\mathrm{f}(\mathrm{f}(x))))$

```
(%i7) h(t);
```

(%o7) $\sqrt{\sqrt{\sqrt{\sqrt{t+1}+1}+1}+1}$

```
(%i8) h(2);
```

(%o8) $\sqrt{\sqrt{\sqrt{\sqrt{3}+1}+1}+1}$

Another composition example is below.

```
(%i1) f(x):=sqrt(x+1);
```

(%o1) $\mathrm{f}(x) := \sqrt{x+1}$

```
(%i2) g(x):=sin(x);
```

(%o2) $\mathrm{g}(x) := \sin(x)$

```
(%i3) f(g(x));
```

(%o3) $\sqrt{\sin(x)+1}$

```
(%i4) g(f(x));
```

(%o4) $\sin\left(\sqrt{x+1}\right)$

As with any computer program you need to be careful what you tell it to do. It will do exactly what you tell it. In the below string of examples we define a function $f(x)$ and then we define the value of x to be 5. Note that in line 3, $f(x)$ is now the expression defined at 5, since x is equal to 5. Similarly, input number 5 is asking for $f(6)$. Also note that $f(3)$, $f(t)$, and $f(t+1)$ act as we would expect. Also, when we redefine x as x, $f(x)$ returns the expression.

```
(%i1) f(x):=x^2+x-1;
```

(%o1) $\mathrm{f}(x) := x^2+x-1$

```
(%i2) x:5;
```

(%o2) 5

```
(%i3) f(x);
```

(%o3) 29

```
(%i4) f(3);
```

(%o4) 11

```
(%i5) f(x+1);
```

(%o5) 41

```
(%i6) f(t);
```

(%o6) t^2+t-1

```
(%i7) f(t+1);
```

(%o7) $(t+1)^2+t$

```
(%i8) x:'x;
```

(%o8) x

```
(%i9) f(x);
```

(%o9) x^2+x-1

2.5 Some Discrete Mathematics & Number Theory Commands

In this section we will look at a few commands that are related to the number theory and discrete mathematics that we tend to encounter most in the area of cryptography.

2.5.1 Modulus

To compute a simple modulus, $a \pmod{n}$ use the `mod(a, n)` command.

```
(%i1) mod(35, 21);
```

(%o1) 14

```
(%i2) mod(-123, 29);
```

(%o2) 22

2.5.2 Power Calculations with a Modulus

Frequently we need to raise a number to a very large power modulo another number, that is, calculate $a^b \pmod{n}$, where b could be a very large number. The way not to do this is with the command `mod(a^b, n)`. Although this will work fine for small values of a and b, when

b gets large the calculation may become too large for your, or anyone's, computer to handle. The reason is that with this command, the program will first calculate a^b and then take the result modulo n. If b is sufficiently large, the calculation of a^b could produce a number that is too large to fit in your computer's memory. For this reason, a better computational method was devised. The command in Maxima for this is, `power_mod(a, b, n)`. The exponentiation algorithm used here is very fast, it raises a to the b power by successive squares and multiplications, each taken modulo n at each stage so that the intermediate calculations do not get too large.

```
(%i1) power_mod(5, 12345, 98765);
(%o1) 82160
(%i2) power_mod(12345, 67890, 1000000000);
(%o2) 931640625
```

2.5.3 Inverse Calculations with a Modulus

The power modulus command will also find inverses of numbers modulo another, as long as it exists, that is, $a^{-1} \pmod{n}$. Maxima also has a special command for this as well. The `inv_mod(a, n)` command is equivalent to the `power_mod(a, -1, n)` command. If the inverse does not exist then both the `inv_mod(a, n)` and the `power_mod(a, -1, n)` commands will return false. The algorithm used here is the extended euclidean algorithm, followed by a modulus if needed.

```
(%i1) inv_mod(7, 23);
(%o1) 10
(%i2) inv_mod(7, 232);
(%o2) 199
(%i3) power_mod(7, -1, 23);
(%o3) 10
(%i4) power_mod(7, -1, 232);
(%o4) 199
```

2.5.4 Greatest Common Divisor

To calculate the greatest common divisor of two numbers simply use the `gcd(a, b)` command. The algorithm used here is the euclidean algorithm

```
(%i1) gcd(7, 23);
(%o1) 1
(%i2) gcd(82382464, 22689746432);
(%o2) 128
```

2.5.5 Extended Greatest Common Divisor

We know that if $d = \gcd(a, b)$, then there exists numbers r and s such that $ar + bs = d$. To calculate the numbers r, s, and d we can use the `gcdex(a, b)` command. This will return the list $[r, s, d]$. The algorithm used here is the extended euclidean algorithm. Many of the commands we are discussing in this section also work on polynomials with integer and in some cases rational and real coefficients. Maxima includes integer versions of some of these where the operands work only on integers. The integer version of this command is `igcdex(a, b)`, so here a and b must be integers.

```
(%i1) gcdex(23, 7);
(%o1) [−3, 10, 1]
(%i2) igcdex(23, 7);
(%o2) [−3, 10, 1]
(%i3) 7*10-3*23;
(%o3) 1
(%i4) gcdex(2346713056, 3671064);
(%o4) [1508, −963983, 536]
(%i5) igcdex(2346713056, 3671064);
(%o5) [1508, −963983, 536]
(%i6) 2346713056*1508-963983*3671064;
(%o6) 536
```

2.5.6 Greatest Common Divisor of Several Numbers

To calculate the greatest common divisor of several numbers use the `ezgcd(a, b, c, ...)` command, yes the command is `ezgcd`, believe it or not. What is returned is a list of numbers, the first is the GCD of the list of numbers and the rest are the inputs all divided by the GCD. The algorithm used here is simply multiple uses of the euclidean algorithm.

```
(%i1) ezgcd(7, 14, 21, 35);
```

(%o1) $[7, 1, 2, 3, 5]$

2.5.7 Least Common Multiple

To calculate the least common multiple of several numbers use the `lcm(a, b, c, ...)` command.

```
(%i1) lcm(5, 15, 35);
```

(%o1) 105

2.5.8 Chinese Remainder Theorem

The Chinese Remainder Theorem is really an algorithm for solving a system of congruences,

$$\begin{aligned} x &= r_1 \pmod{m_1} \\ x &= r_2 \pmod{m_2} \\ x &= r_3 \pmod{m_3} \\ &\vdots \\ x &= r_n \pmod{m_n} \end{aligned}$$

where the set $\{m_1, m_2, \ldots, m_n\}$ are positive and pairwise coprime integers. The Maxima command to solve this system is `chinese([r_1,..., r_n], [m_1,..., m_n])`. Note that the residues and the moduli are in lists and the corresponding entries define each of the congruences. If the set of moduli are coprime the Chinese Remainder Theorem guarantees a solution. If, on the other hand, moduli are not coprime then there may or may not be a solution. If Maxima cannot find a solution to the system it will return false.

```
(%i1) chinese([1, 2, 3, 4],[5, 7, 11, 24]);
```

(%o1) 8836

2.5.9 Functions for Primes

There are several functions in Maxima for working with prime numbers. The first we will look at is primality testing. The Maxima command to test if a number is prime (or probably prime) is `primep(n)`. If `primep(n)` returns false, n is a composite number and if it returns true, n is a prime number with very high probability. For n less than 341550071728321 a deterministic version of Miller-Rabin's test is used, so if `primep(n)` returns true in this case, then n is a prime number.

If n is bigger than 341550071728321, then `primep(n)` uses `primep_number_of_tests` Miller-Rabin's pseudo-primality tests and one Lucas pseudo-primality test. The probability that a non-prime n will pass one Miller-Rabin test is less than $\frac{1}{4}$. Using the default value 25 for `primep_number_of_tests`, the probability of n being composite when the command says that it is prime is less than 10^{-15}. If we increase the number of tests to 100 then the probability of n being composite when the command says that it is prime is less than 10^{-60}.

```
(%i1) primep(350193560150161);
(%o1) false
(%i2) primep(6731861687);
(%o2) true
(%i3) primep(854802562046504305734517048510457014356134456571826 7);
(%o3) true
(%i4) primep_number_of_tests;
(%o4) 25
(%i5) primep_number_of_tests:100;
(%o5) 100
(%i6) primep(8548025620465043057345170485104570143561344565718267);
(%o6) true
(%i7) 1/4.0^(100);
```

(%o7) $6.223015277861142 \cdot 10^{-61}$

So in our above examples, 350193560150161 is definitely a composite number, 6731861687 is definitely prime, and 8548025620465043057345170485104570143561344565718267 is probably prime with the probability of it actually being composite being less than 10^{-60}.

Maxima also has functions for finding the next probable prime and the previous probable prime. These functions use the `primep(n)` function for verification of the prime, so if the returned value is less than 341550071728321, the number is definitely prime and if the return value larger, then it is a probable prime, with the probably of being composite as above. To find the next prime number larger than n use the command `next_prime(n)` and to find the previous prime number smaller than n use the command `prev_prime(n)`.

```
(%i9) next_prime(350193560150161);
(%o9) 350193560150221
(%i10)primep(350193560150221);
(%o10) true
```

```
(%i11)prev_prime(6437825402430183470518750418750 15);
```

(%o11) 6437825402430183470518750418747 89

```
(%i12)primep(643782540243018347051875041874789);
```

(%o12) true

2.5.10 Jacobi and Legendre Symbols

Recall that the Legendre symbol is defined as follows, for an odd prime n,

$$\left(\frac{m}{n}\right) = \begin{cases} 0, & \text{if } m \equiv 0 \pmod{n} \\ 1, & \text{if } 0 \not\equiv m \equiv x^2 \pmod{n}, \text{ for some } x \\ -1, & \text{otherwise} \end{cases}$$

So for an odd prime n, the Legendre symbol will tell us if an integer m is a quadratic residue modulo n. The Jacobi symbol is a generalization of the Legendre symbol, it is defined for any odd number n as

$$\left(\frac{m}{n}\right) = \left(\frac{m}{p_1}\right)^{a_1}\left(\frac{m}{p_2}\right)^{a_2}\cdots\left(\frac{m}{p_r}\right)^{a_r}$$

where all of the p_i are distinct primes and $n = p_1^{a_1}p_2^{a_2}\cdots p_r^{a_r}$. Note that each of the terms in the above product are Legendre symbols, since all of the p_i are prime. One big difference between the Jacobi and Legendre symbols is that if n is not prime and $\left(\frac{m}{n}\right) = 1$ then we are not guaranteed that m is a quadratic residue modulo n. On the other hand, if $\left(\frac{m}{n}\right) = -1$ then we know that m is not a quadratic residue modulo n.

In Maxima, the command to do both of these symbols is `jacobi(m, n)`. If n is prime, then this is the Legendre symbol and we can deduce if m is a quadratic residue modulo n. If n is not prime then we are working with the Jacobi symbol.

```
(%i1) jacobi(5, 23);
```

(%o1) -1

```
(%i2) jacobi(3, 23);
```

(%o2) 1

```
(%i3) jacobi(19, 231);
```

(%o3) 1

```
(%i4) jacobi(17, 231);
```

(%o4) -1

```
(%i5) jacobi(3, 231);
```

(%o5) 0

So in our above examples,

1. 5 is not a quadratic residue modulo 23.
2. 3 is a quadratic residue modulo 23. In fact, $3 \equiv 7^2 \pmod{23}$.
3. We do not know if 19 is a quadratic residue modulo 231, but it is possible.
4. 17 is not a quadratic residue modulo 231.
5. Since $\gcd(3, 231) \neq 1$, one of the Legendre symbols in the product definition of the Jacobi symbol is 0, making the product 0.

2.5.11 Continued Fractions

A continued fraction is when you take a number x and express it in the form,

$$x = a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cfrac{1}{\ddots}}}}$$

For some values of x their continued fraction representation will terminate, some will repeat and some will neither terminate nor repeat. For example,

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{\ddots}}}$$

$$\frac{1+\sqrt{5}}{2} = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}$$

$$\frac{5742}{2131} = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{11 + \cfrac{1}{5}}}}}}}}$$

There are several Maxima commands that come in handy when working with continued fractions and we will create a couple that will make some of the computations in these notes a little easier. Maxima's `cf(n)` command will return a list representation of the continued fraction representation of n. Here n can be any real number, it does not have to be rational. So an output of $[a_1, a_2, a_3, a_4, \ldots]$ is a representation for,

$$a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cfrac{1}{\ddots}}}}$$

If the number has a terminating continued fraction representation, Maxima will produce the entire representation, such as, in output number 5 below. In the case where the continued fraction representation is repeating, Maxima will halt the representation once it notices that it has finished a period of the repetition. So in output number 1, Maxima gives $[1, 2]$ for the representation of $\sqrt{2}$. Since a terminating continued fraction representation would result in a rational number, it is clear that Maxima is telling us that the representation is $[1, 2, 2, 2, \ldots]$. If you would like to see more periods of repetition, as I usually do, all you need to do is set the `cflength` variable to the number of periods you want to see. Here we set it to 3 in input number 2 and then reran $\sqrt{2}$.

```
(%i1) cf(sqrt(2));
```

(%o1) $[1, 2]$

```
(%i2) cflength:3;
```

(%o2) 3

```
(%i3) cf(sqrt(2));
```

(%o3) $[1, 2, 2, 2]$

```
(%i4) cf((1 + sqrt(5))/2);
```

(%o4) $[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]$

```
(%i5) lst:cf(5742/2131);
```

(%o5) $[2, 1, 2, 3, 1, 1, 1, 11, 5]$

To view a list as a continued fraction use the `cfdisrep(L)` where L is a list. This will produce a nice continued fraction layout. To simplify the fraction, just apply the `ratsimp` command to the result. This can be done through the simplify menu as well.

```
(%i6) cfdisrep (lst);
```

(%o6) $$2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{11 + \cfrac{1}{5}}}}}}}}$$

```
(%i7) ratsimp(cfdisrep(lst));
```

(%o7) $\frac{5742}{2131}$

```
(%i8) ratsimp(cfdisrep([1,2,2,2,2,2,2,2,2,2,2]));
```

(%o8) $\frac{8119}{5741}$

```
(%i9) float(%), numer;
```

(%o9) 1.414213551646055

There are times when we will want to find the continued fraction representation of a number and then look at successive approximations by taking more and more of the continued fraction. For example, with $\sqrt{2}$, we would look at

$$1+\frac{1}{2}=\frac{3}{2} \qquad 1+\frac{1}{2+\frac{1}{2}}=\frac{7}{5} \qquad 1+\frac{1}{2+\frac{1}{2+\frac{1}{2}}}=\frac{17}{12}$$

and so on. Maxima does not have a built-in function to convert a list to a simplified continued fraction but it is easy enough to create. The function,

```
from_cf(L):=ratsimp(cfdisrep(L))
```

will take in a list, convert it to the displayed continued fraction and then simplify the result. We will go one step further, the following command will take the first n entries from input list, L, and convert that to a continued fraction and simplify the result.

```
from_cf_n(L, n):=ratsimp(cfdisrep(makelist(L[i], i, 1, n)))
```

Some examples of these functions are below.

```
(%i10) from_cf(L):=ratsimp(cfdisrep(L));
```

(%o10) $\text{from\_cf}(L) := \text{ratsimp}(\text{cfdisrep}(L))$

```
(%i11) from_cf(lst);
```

(%o11) $\frac{5742}{2131}$

```
(%i12) from_cf_n(L, n):=ratsimp(cfdisrep(makelist(L[i], i, 1, n)));
```

(%o12) $\text{from\_cf\_n}(L, n) := \text{ratsimp}(\text{cfdisrep}(\text{makelist}(L_i, i, 1, n)))$

```
(%i13) from_cf_n(lst, 4);
```

(%o13) $\frac{27}{10}$

```
(%i14) from_cf([2,1,2,3]);
```

(%o14) $\frac{27}{10}$

```
(%i15) from_cf_n(lst, 8);
```

(%o15) $\frac{1129}{419}$

```
(%i16) from_cf([2,1,2,3,1,1,1,11]);
```

(%o16) $\frac{1129}{419}$

2.5.12 Solving Equations

Of course, we need to solve equations. Maxima has a very versatile equation solver. We will only be using a small portion of what it is capable of doing. For example, it can find the, relatively ugly, exact solutions to $x^3 - 3x^2 + 2x + 5 = 0$.

```
(%i1) solve(x^3-3*x^2+2*x+5=0, x);
```

(%o1) $$[x = \frac{\frac{\sqrt{3}\cdot i}{2} - \frac{1}{2}}{3\cdot\left(\frac{\sqrt{671}}{2\cdot 3^{\frac{3}{2}}} - \frac{5}{2}\right)^{\frac{1}{3}}} + \left(\frac{\sqrt{671}}{2\cdot 3^{\frac{3}{2}}} - \frac{5}{2}\right)^{\frac{1}{3}} \cdot \left(-\frac{\sqrt{3}\cdot i}{2} - \frac{1}{2}\right) + 1, x = \left(\frac{\sqrt{671}}{2\cdot 3^{\frac{3}{2}}} - \frac{5}{2}\right)^{\frac{1}{3}} \cdot \left(\frac{\sqrt{3}\cdot i}{2} - \frac{1}{2}\right) + \frac{-\frac{\sqrt{3}\cdot i}{2} - \frac{1}{2}}{3\cdot\left(\frac{\sqrt{671}}{2\cdot 3^{\frac{3}{2}}} - \frac{5}{2}\right)^{\frac{1}{3}}} + 1, x = \left(\frac{\sqrt{671}}{2\cdot 3^{\frac{3}{2}}} - \frac{5}{2}\right)^{\frac{1}{3}} + \frac{1}{3\cdot\left(\frac{\sqrt{671}}{2\cdot 3^{\frac{3}{2}}} - \frac{5}{2}\right)^{\frac{1}{3}}} + 1]$$

A little more down to earth, the solutions to $3x^2 - 2x - 5 = 0$ are $\frac{5}{3}$ and -1.

```
(%i1) solve(3*x^2-2*x-5=0, x);
```

(%o1) $[x = \frac{5}{3}, x = -1]$

Several things to notice about the syntax to the solve function. When you are solving a single equation, the first argument is the equation to be solved and the second argument is the variable to solve the equation for. An equation in Maxima is simply two Maxima expressions with an equal sign between them. Although this is the preferred syntax, if the equation is simply an expression (no equal sign) Maxima will assume that the expression is set to equal 0. Also, if there is only one variable in the equation, then the variable to be solved for can be omitted and Maxima will take the one in the equation. So the following inputs also give the same solutions.

```
(%i2) solve(3*x^2-2*x-5, x);
```

(%o2) $[x = \frac{5}{3}, x = -1]$

```
(%i3) solve(3*x^2-2*x-5);
```

(%o3) $[x = \frac{5}{3}, x = -1]$

Maxima can also do some modular solving of equations, the solve command will solve linear and systems of linear equations over a modulus. For extracting square and cube roots there is another way to do this which we will talk about in the next section. To tell Maxima that we wish to work modulo a number n we reset the `modulus` variable. The default value of the `modulus` variable is false, which means that Maxima is not working over a modulus, it is working over the real and complex numbers systems. If we set this to a positive integer then Maxima shifts into modular calculation mode. So on input line number 2, we shift Maxima into working modulo 23. You can also invoke the modulus change with the solve command at the same time so that the value of `modulus` is not changed globally, to do this simply follow the solve command with the modulus command on the same line separated with a comma, as we did on input number 7.

```
(%i1) modulus;
```

(%o1) false

```
(%i2) modulus:23;
```

(%o2) 23

```
(%i3) solve(7*x=3);
```

(%o3) $[x = 7]$

```
(%i4) modulus;
```

(%o4) 23

```
(%i5) modulus:false;
```

(%o5) false

```
(%i6) solve(7*x=3);
```

(%o6) $[x = \frac{3}{7}]$

```
(%i7) solve(7*x=3),modulus:23;
```

(%o7) $[x = 7]$

```
(%i8) modulus;
```

(%o8) false

2.5.13 Modular Square Roots and Cube Roots

Unfortunately, the solver in modular arithmetic mode is not powerful enough to solve non-linear equations. Fortunately, in cryptography, there are not too many times that we want to solve a general non-linear modular equation, although there are times when we want to find a square root of a number modulo another number, if it exists. Maxima has several commands to do this, each use a slightly different algorithm. These are contained in the `gf` package that must be loaded before using them. The `gf` package is a special package of routines that allow the user to do finite field computations in Maxima. To load in the `gf` package, use the command `load(gf)$`, the $ simply suppresses output, which is not needed here. There are three square root functions, `msqrt(a, p)`, `ssqrt(a, p)`, and `gf_sqrt(a, p)`. For each, the value a is the one being rooted and p is the prime modulus. If a is not a quadratic residue modulo p you will get an error. The last command will probably not be used too much in these notes but it will find a modular cube root.

```
(%i1) load(gf)$

(%i2) msqrt(5, 29);

(%o2) [18, 11]

(%i3) mod(18^2,29);

(%o3) 5

(%i4) ssqrt(5, 29);

(%o4) [18, 11]

(%i5) gf_sqrt(5, 29);

(%o5) [11, 18]

(%i6) msqrt(8, 17);

(%o6) [5, 12]

(%i7) msqrt(7, 17);

ERROR: First argument must be a quadratic residue.
#0: msqrt(a=7,p=17)(gf.mac line 483)
– an error. To debug this try: debugmode(true);

(%i8) mcbrt(5, 29);

(%o8) 22

(%i9) mod(22^3,29);

(%o9) 5
```

2.5.14 Factoring

Factoring is essential for many cyptographic processes and cryptanalysis. In fact, finding faster factoring algorithms is one of the central goals in cryptography. Maxima has two factoring commands for integers, `factor` and `ifactors`. The `factor` command simply calls the `ifactors` command and displays the result in a slightly different form. So there is no difference in the runtime or algorithms used. To use these, simply input `factor(n)` or `ifactors(n)`, where n is the number to be factored. Factorization methods used are trial divisions by primes up to 9973, Pollard's rho and $p-1$ methods, and elliptic curves.

```
(%i1) factor(56584537362323645765968670670767558484);
```

(%o1) $2^2 \cdot 3^2 \cdot 197 \cdot 9011199181297 \cdot 885414139624867842841$

```
(%i2) ifactors(56584537362323645765968670670767558484);
```

(%o2) $[[2, 2], [3, 2], [197, 1], [9011199181297, 1], [885414139624867842841, 1]]$

As you can see from the output above, the `ifactors` command returns a list of factor lists, in each factor list the first entry is the factor and the second is the multiplicity of the factor. The `factor` command simply reorganizes the output into a more mathematical format.

2.5.15 Factoring Polynomials

In Maxima, the command to factor a polynomial is `factor`. In the first example below, the input and output is the factorization of $x^6 + x^5 + x^3 + 1$ using integer coefficients, that is, the coefficients are integers and the coefficients of the factorization are also integers.

To factor a polynomial modulo a prime in Maxima, simply set the `modulus` option to the desired prime, as we did with the second input. Now when the factor command is invoked the factorization will be modulo the prime. In the second and third inputs, we change the modulus to 2 and then factor $x^6 + x^5 + x^3 + 1$, the result is a factorization modulo 2.

```
(%i1) factor(x^6 + x^5 + x^3 + 1);
```

(%o1) $(x+1) \cdot \left(x^2+1\right) \cdot \left(x^3-x+1\right)$

```
(%i2) modulus:2;
```

(%o2) 2

```
(%i3) factor(x^6 + x^5 + x^3 + 1);
```

(%o3) $(x+1)^3 \cdot \left(x^3+x+1\right)$

To go back to non-modulus calculations simply set the `modulus` option to `false`.

2.5.16 Euler Totient Function

The Euler totient function, also known as the Euler phi function, $\phi(n)$ is the number of integers less than or equal to n which are relatively prime to n. In Maxima this command is simply, `totient(n)`.

```
(%i1) totient(24);
```

(%o1) 8

```
(%i2) totient(75);
```

(%o2) 40

```
(%i3) totient(56584537362323645765968670670767558484);
```

(%o3) 18765768736064642666886637160664545280

Note that the calculation of the totient function requires the factorization of n, hence the calculation time of the totient of a large number could be lengthy.

2.5.17 Primitive Roots

A primitive root modulo n is a number whose powers modulo n generate all numbers less than n that are relatively prime to n. In more mathematical lingo, a primitive root modulo n is a number whose powers modulo n generate all numbers in $(\mathbb{Z}/n\mathbb{Z})^*$. If the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$ is cyclic, `zn_primroot(n)` computes the smallest primitive root modulo n. $(\mathbb{Z}/n\mathbb{Z})^*$ is cyclic if n is equal to 2, 4, p^k or $2p^k$, where p is prime and greater than 2 and k is a natural number. Most of the time, for us, n will be a prime number.

```
(%i1) zn_primroot(139);
```

(%o1) 2

```
(%i2) zn_primroot(7);
```

(%o2) 3

```
(%i3) for a:1 thru 6 do display(mod(3^a, 7))$
```

$\text{mod}\,(3, 7) = 3$
$\text{mod}\,(9, 7) = 2$
$\text{mod}\,(27, 7) = 6$
$\text{mod}\,(81, 7) = 4$
$\text{mod}\,(243, 7) = 5$
$\text{mod}\,(729, 7) = 1$

```
(%i4) zn_primroot(9);
```

(%o4) 2

```
(%i5) for a:1 thru 8 do display(mod(2^a, 9))$
```

$\mathrm{mod}\,(2, 9) = 2$
$\mathrm{mod}\,(4, 9) = 4$
$\mathrm{mod}\,(8, 9) = 8$
$\mathrm{mod}\,(16, 9) = 7$
$\mathrm{mod}\,(32, 9) = 5$
$\mathrm{mod}\,(64, 9) = 1$
$\mathrm{mod}\,(128, 9) = 2$
$\mathrm{mod}\,(256, 9) = 4$

Maxima also has a function that will determine if a number is a primitive root modulo another number. The command `zn_primroot_p(a, n)` will return true if a is a primitive root modulo n and false otherwise. As with the `zn_primroot(n)` command, the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$ must be cyclic.

```
(%i1) zn_primroot_p(3, 7);
```

(%o1) true

```
(%i2) zn_primroot_p(2, 7);
```

(%o2) false

```
(%i3) zn_primroot_p(2, 139);
```

(%o3) true

```
(%i4) zn_primroot_p(13, 139);
```

(%o4) false

```
(%i5) zn_primroot_p(132, 139);
```

(%o5) true

These commands rely on the factorization of the totient function of n, hence for large n the calculation could be lengthy.

2.5.18 Discrete Logarithms

Given three numbers, g, a, and n the solution x to the congruence $g^x \equiv a \pmod{n}$ is called the discrete logarithm of a, base g modulo n, if x exists.

If $(\mathbb{Z}/n\mathbb{Z})^*$ is a cyclic group (n is equal to 2, 4, p^k or $2p^k$, where p is prime and greater than 2 and k is a natural number), g a primitive root modulo n and let a be a member of this group. Then `zn_log(a, g, n)` then solves the congruence $g^x \equiv a \pmod{n}$. The algorithm uses a Pohlig-Hellman-reduction and Pollard's Rho-method for discrete logarithms.

```
(%i1) zn_primroot(7);
(%o1) 3
(%i2) zn_log(6, 3, 7);
(%o2) 3
(%i3) mod(3^3, 7);
(%o3) 6
(%i4) zn_primroot_p(132, 139);
(%o4) true
(%i5) zn_log(23, 132, 139);
(%o5) 93
(%i6) power_mod(132, 93, 139);
(%o6) 23
```

2.5.19 Order of an Element

The order of an element a modulo n is the smallest positive power of a modulo n that results in 1. More specifically, a must be a unit in the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$. In Maxima, the command for this computation is `zn_order(a, n)`. The same restrictions on n hold here as with the primitive root calculations and the algorithm relies on the factorization of the totient of n, so the calculation could be lengthy if n is large.

```
(%i1) zn_order(4, 17);
(%o1) 4
(%i2) mod(4^1, 17);
(%o2) 4
(%i3) mod(4^2, 17);
(%o3) 16
(%i4) mod(4^3, 17);
(%o4) 13
(%i5) mod(4^4, 17);
```

(%o5) 1

2.6 Vectors and Matrices

We will start out with some basic operations on matrices and vectors in general and then we will discuss some ways of doing matrix operations over a modulus.

In Maxima, vectors are simply matrices with either a single row or a single column. Most of these functions will work if the vectors are represented as row vectors or column vectors, and some will work if the vectors are simply defined as a list.

Although this is not always necessary, it is a good idea to load the "eigen" package anytime you want to do work with matrices. The "eigen" package has many matrix manipulation functions built-in, more then just eigenvalues and eigenvectors as its name implies. Recall that to load a package we simply use the load command `load("eigen")`.

2.6.1 Defining a Matrix

To define a matrix or a vector we use a special matrix command,

```
matrix(row1, row2, ..., rown)
```

will define a matrix with n rows, each of the rows in the command must be lists. In the examples below, input 2 defines a 3×3 matrix, input 3 defines a 3-dimensional row vector and input 5 defines a 3-dimensional column vector. Note that input 4 defines a list with three entries, although this looks similar to the row vector A they are different. The wxMaxima interface has menu options for creating a matrix that allow the user to input entries into a dialog box in place of writing a command.

With some operations the list and row vector will work interchangeably and with other operations they will not. Since the syntax for creating a column vector is a bit cumbersome, there is another way to create one. The `covect(L)` or `columnvector(L)` commands will turn the list L into a column vector. One final way to create a column vector is to transpose a row vector or a list. The `transpose(M)` command will return the transpose of a matrix M, that is, change all of the rows of M into columns. So transposing a row vector will produce a column vector. The `transpose(M)` command will also work on a list, so in input number 8 we could still get a column vector with the command `transpose(B)`.

```
(%i1) load("eigen")$
(%i2) M:matrix([1,2,3],[4,5,6],[7,8,9]);
```

(%o2) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

```
(%i3) A:matrix([3,6,9]);
```

(%o3) $\begin{bmatrix} 3 & 6 & 9 \end{bmatrix}$

```
(%i4) B:[3,6,9];
```

(%o4) $[3,6,9]$

```
(%i5) C:matrix([1],[5],[7]);
```

(%o5) $\begin{bmatrix}1\\5\\7\end{bmatrix}$

```
(%i6) D:covect([1,5,7]);
```

(%o6) $\begin{bmatrix}1\\5\\7\end{bmatrix}$

```
(%i7) H:columnvector([4,5,10]);
```

(%o7) $\begin{bmatrix}4\\5\\10\end{bmatrix}$

```
(%i8) J:transpose(A);
```

(%o8) $\begin{bmatrix}3\\6\\9\end{bmatrix}$

Another nifty thing you can do with matrices is to extract rows, columns and entries relatively easily. You can also join matrices together, add rows and columns to a matrix, and change entries

Once a matrix, say M is defined, you can extract the (i, j) entry using either `M[i,j]` or `M[i][j]`. You can extract a row by either `M[i]` or `row(M,i)` where i is the row to extract. Note that these operations do not alter the original matrix. You can also extract the i^{th} column with `col(M,i)`. Adding rows and columns to a matrix can be done with the `addrow` and the `addcol` commands. They both have the form,

```
addrow(M1, M2, M3, ..., Mn)
```

where M_1, M_2, M_3, ..., M_n are either matrices or lists.

```
(%i1) M:matrix([1,2,3],[4,5,6],[7,8,9]);
```

(%o1) $\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}$

```
(%i2) M[2,3];
```

(%o2) 6

```
(%i3) M[1];
```

(%o3) $[1,2,3]$

```
(%i4) M[3];
```

(%o4) $[7,8,9]$

```
(%i5) row(M,2);
```

(%o5) $\begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$

```
(%i6) col(M,1);
```

(%o6) $\begin{bmatrix} 1 \\ 4 \\ 7 \end{bmatrix}$

```
(%i7) addcol(M,[10, 11, 12]);
```

(%o7) $\begin{bmatrix} 1 & 2 & 3 & 10 \\ 4 & 5 & 6 & 11 \\ 7 & 8 & 9 & 12 \end{bmatrix}$

```
(%i8) addcol(M,[10, 11, 12],[15, 16, 17]);
```

(%o8) $\begin{bmatrix} 1 & 2 & 3 & 10 & 15 \\ 4 & 5 & 6 & 11 & 16 \\ 7 & 8 & 9 & 12 & 17 \end{bmatrix}$

```
(%i9) addrow(M,[10, 11, 12]);
```

(%o9) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$

```
(%i10) addrow(M,[10, 11, 12],[15, 16, 17]);
```

(%o10) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 15 & 16 & 17 \end{bmatrix}$

```
(%i11) A:matrix([a,b,c],[d,e,f],[g,h,i]);
```

(%o11) $\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$

```
(%i12) addcol(M,A);
```

(%o12) $\begin{bmatrix}1 & 2 & 3 & a & b & c\\ 4 & 5 & 6 & d & e & f\\ 7 & 8 & 9 & g & h & i\end{bmatrix}$

```
(%i13) addrow(M,A);
```

(%o13) $\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\\ a & b & c\\ d & e & f\\ g & h & i\end{bmatrix}$

Submatrix construction along with replacing rows, columns and entries is quick as well. You can replace a row using the notation `M[i]:[a, b, ..., n]` where i is the row to change and the list of elements has the same number of columns as M. There does not seem to be a column replacement command but one can do that by transposing the matrix, replacing the desired row and then transposing again.

Maxima has a built-in function to construct the (i, j)-Minor of a matrix, `minor(M,i,j)`. Maxima also has an interesting function for the construction of a submatrix. The command

```
submatrix(r1, r2, ..., rm, M, c1, c2, ..., cn)
```

Will take the matrix M and remove rows $r_1, \ldots, r_m$ and columns $c_1, \ldots, c_n$.

```
(%i1) M:matrix([1,2,3],[4,5,6],[7,8,9]);
```

(%o1) $\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$

```
(%i2) M;
```

(%o2) $\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$

```
(%i3) M[2]:[7,7,7];
```

(%o3) $[7, 7, 7]$

```
(%i4) M;
```

(%o4) $\begin{bmatrix}1 & 2 & 3\\ 7 & 7 & 7\\ 7 & 8 & 9\end{bmatrix}$

```
(%i5) minor(M,1,2);
```

(%o5) $\begin{bmatrix}7 & 7\\ 7 & 9\end{bmatrix}$

```
(%i6) submatrix(1, M, 2);
```

(%o6) $\begin{bmatrix} 7 & 7 \\ 7 & 9 \end{bmatrix}$

```
(%i7) submatrix(1, 3, M, 2);
```

(%o7) $\begin{bmatrix} 7 & 7 \end{bmatrix}$

```
(%i8) M;
```

(%o8) $\begin{bmatrix} 1 & 2 & 3 \\ 7 & 7 & 7 \\ 7 & 8 & 9 \end{bmatrix}$

```
(%i9) col(M,3);
```

(%o9) $\begin{bmatrix} 3 \\ 7 \\ 9 \end{bmatrix}$

```
(%i10)M:transpose(M);
```

(%o10) $\begin{bmatrix} 1 & 7 & 7 \\ 2 & 7 & 8 \\ 3 & 7 & 9 \end{bmatrix}$

```
(%i11)M[1]:[5,4,3];
```

(%o11) $[5, 4, 3]$

```
(%i12)M:transpose(M);
```

(%o12) $\begin{bmatrix} 5 & 2 & 3 \\ 4 & 7 & 7 \\ 3 & 8 & 9 \end{bmatrix}$

```
(%i13)M;
```

(%o13) $\begin{bmatrix} 5 & 2 & 3 \\ 4 & 7 & 7 \\ 3 & 8 & 9 \end{bmatrix}$

One thing about matrices that is different from numeric values is the way that assignments work. If you are familiar with the way arrays are stored in a programming language line Java or C++ this will come as no surprise but if you are not familiar with this please look at the next examples carefully.

```
(%i1) M:matrix([1,2,3],[4,5,6],[7,8,9]);
```

(%o1) $$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$$

```
(%i2) M;
```

(%o2) $$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$$

```
(%i3) A:M;
```

(%o3) $$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$$

```
(%i4) A;
```

(%o4) $$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$$

```
(%i5) A[2,2]:x;
```

(%o5) x

```
(%i6) A;
```

(%o6) $$\begin{bmatrix}1 & 2 & 3\\4 & x & 6\\7 & 8 & 9\end{bmatrix}$$

```
(%i7) M;
```

(%o7) $$\begin{bmatrix}1 & 2 & 3\\4 & x & 6\\7 & 8 & 9\end{bmatrix}$$

```
(%i8) M[2,2]:5;
```

(%o8) 5

```
(%i9) A;
```

(%o9) $$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$$

```
(%i10)M:matrix([1,2,3],[4,5,6],[7,8,9]);
```

(%o10) $$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$$

```
(%i11)M;
```

(%o11) $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$

```
(%i12) A;
```

(%o12) $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$

```
(%i13) B:copymatrix(M);
```

(%o13) $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$

```
(%i14) B;
```

(%o14) $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$

```
(%i15) M;
```

(%o15) $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$

```
(%i16) B[2,2]:t;
```

(%o16) t

```
(%i17) B;
```

(%o17) $\begin{bmatrix}1 & 2 & 3\\4 & t & 6\\7 & 8 & 9\end{bmatrix}$

```
(%i18) M;
```

(%o18) $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$

To summarize what happened above, we assigned A the matrix M using `A:M`. What happened is that the variables A and M both referenced the same matrix, in other words, A was not a new matrix with the same entries as M, as we might have expected. So when we changed an entry in A it was also changed in M, since there is really only one matrix in memory. To make Maxima create a new matrix we use the `copymatrix` command. So `B:copymatrix(M)` creates a new matrix with the same entries as M. So when we change B, M is not altered.

2.6.2 Matrix Arithmetic

Matrix addition and subtraction are done with the usual + and − operators. If the matrices are the same size then the operation returns the resulting matrix, if the matrices are not the same size then Maxima displays an error. Matrix multiplication is not done with the $\star$ symbol, `M*A` will return an entry by entry product, which is not the standard matrix multiplication. The same is true for the / symbol, `M/A` will return an entry by entry quotient. There may be times you want to use these types of operations but we are more interested in the standard matrix multiplication. Matrix multiplication is done with the `.` symbol, `M.A` will return the matrix product as long as the matrices are of compatible size, if not, you will get an error.

Matrix powers are not done by ˆ but rather ˆˆ. If A is a square matrix then `A^^3` will return A^3. Using only the single power symbol will return a matrix with each entry raised to the power, again, this might be something you want to do but not for taking a matrix power. If A is not a square matrix, you will get an error when taking a power.

Finding the inverse of a matrix can be done with `A^^-1` or with the `invert(A)` command. If A is not square or if the matrix is not invertible you will get an error. You can find the determinant of a square matrix using the `determinant(A)` command.

```
(%i1) M:matrix([1,2,3],[4,5,6],[7,8,9]);
```

$$(\%o1)\quad \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$$

```
(%i2) A:matrix([1,0,1],[2,1,5],[3,2,0]);
```

$$(\%o2)\quad \begin{bmatrix}1 & 0 & 1\\ 2 & 1 & 5\\ 3 & 2 & 0\end{bmatrix}$$

```
(%i3) B:matrix([-1,3,2],[-2,0,4]);
```

$$(\%o3)\quad \begin{bmatrix}-1 & 3 & 2\\ -2 & 0 & 4\end{bmatrix}$$

```
(%i4) C:matrix([-1,3],[-2,0],[1,1]);
```

$$(\%o4)\quad \begin{bmatrix}-1 & 3\\ -2 & 0\\ 1 & 1\end{bmatrix}$$

```
(%i5) A+M;
```

$$(\%o5)\quad \begin{bmatrix}2 & 2 & 4\\ 6 & 6 & 11\\ 10 & 10 & 9\end{bmatrix}$$

```
(%i6) A-M;
```

(%o6) $\begin{bmatrix} 0 & -2 & -2 \\ -2 & -4 & -1 \\ -4 & -6 & -9 \end{bmatrix}$

```
(%i7) A+B;
```

fullmap: arguments must have same formal structure.
– an error. To debug this try: debugmode(true);

```
(%i8) B-transpose(C);
```

(%o8) $\begin{bmatrix} 0 & 5 & 1 \\ -5 & 0 & 3 \end{bmatrix}$

```
(%i9) A.M;
```

(%o9) $\begin{bmatrix} 8 & 10 & 12 \\ 41 & 49 & 57 \\ 11 & 16 & 21 \end{bmatrix}$

```
(%i10) B.C;
```

(%o10) $\begin{bmatrix} -3 & -1 \\ 6 & -2 \end{bmatrix}$

```
(%i11) C.B;
```

(%o11) $\begin{bmatrix} -5 & -3 & 10 \\ 2 & -6 & -4 \\ -3 & 3 & 6 \end{bmatrix}$

```
(%i12) M.B;
```

MULTIPLYMATRICES: attempt to multiply nonconformable matrices.
– an error. To debug this try: debugmode(true);

```
(%i13) A^^3;
```

(%o13) $\begin{bmatrix} 11 & 4 & 14 \\ 62 & 25 & 74 \\ 50 & 28 & 17 \end{bmatrix}$

```
(%i14) invert(M);
```

expt: undefined: 0 to a negative exponent.
– an error. To debug this try: debugmode(true);

```
(%i15) invert(A);
```

(%o15) $\begin{bmatrix} \frac{10}{9} & -\frac{2}{9} & \frac{1}{9} \\ -\frac{5}{3} & \frac{1}{3} & \frac{1}{3} \\ -\frac{1}{9} & \frac{2}{9} & -\frac{1}{9} \end{bmatrix}$

```
(%i16) determinant(A);
```

(%o16) -9

```
(%i17) determinant(M);
```

(%o17) 0

```
(%i18) A^3;
```

(%o18) $\begin{bmatrix} 1 & 0 & 1 \\ 8 & 1 & 125 \\ 27 & 8 & 0 \end{bmatrix}$

```
(%i19) A*M;
```

(%o19) $\begin{bmatrix} 1 & 0 & 3 \\ 8 & 5 & 30 \\ 21 & 16 & 0 \end{bmatrix}$

```
(%i20) A/M;
```

(%o20) $\begin{bmatrix} 1 & 0 & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{5} & \frac{5}{6} \\ \frac{3}{7} & \frac{1}{4} & 0 \end{bmatrix}$

2.6.3 Matrix Reduction

There are two commands for reducing matrices in Maxima, they are the `echelon(M)` and `triangularize(M)` commands. The echelon command returns the echelon form of the matrix M, as produced by Gaussian elimination. The echelon form is computed from M by elementary row operations such that the first non-zero element in each row in the resulting matrix is one and the column elements under the first one in each row are all zero. The triangularize command also carries out Gaussian elimination, but it does not normalize the leading non-zero element in each row.

```
(%i1) M:matrix([1,2,3],[4,5,6],[7,8,9]);
```

(%o1) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

```
(%i2) A:matrix([1,0,1],[2,1,5],[3,2,0]);
```

(%o2) $\begin{bmatrix} 1 & 0 & 1 \\ 2 & 1 & 5 \\ 3 & 2 & 0 \end{bmatrix}$

```
(%i3) B:matrix([-1,3,2],[-2,0,4]);
```

(%o3) $\begin{bmatrix} -1 & 3 & 2 \\ -2 & 0 & 4 \end{bmatrix}$

```
(%i4) C:matrix([-1,3],[-2,0],[1,1]);
```

(%o4) $\begin{bmatrix} -1 & 3 \\ -2 & 0 \\ 1 & 1 \end{bmatrix}$

```
(%i5) echelon(M);
```

(%o5) $\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix}$

```
(%i6) echelon(A);
```

(%o6) $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$

```
(%i7) echelon(B);
```

(%o7) $\begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & 0 \end{bmatrix}$

```
(%i8) echelon(C);
```

(%o8) $\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$

```
(%i9) triangularize(M);
```

(%o9) $\begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 0 \end{bmatrix}$

```
(%i10)triangularize(A);
```

(%o10) $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & -9 \end{bmatrix}$

```
(%i11)triangularize(B);
```

(%o11) $\begin{bmatrix} -2 & 0 & 4 \\ 0 & -6 & 0 \end{bmatrix}$

```
(%i12)triangularize(C);
```

(%o12) $\begin{bmatrix} -2 & 0 \\ 0 & -6 \\ 0 & 0 \end{bmatrix}$

Unfortunately, Maxima does not have a built-in function for finding the reduced row echelon form of a matrix, but it is easy to create one. The reduced row echelon form of a matrix has the same properties as the echelon form except that the leading one in each row is the only nonzero element in its column. The following script for the `rref` command will find the reduced row echelon form of the input matrix.

```
rref(a):=block([r,c,pc,pcf],[r,c]:matrix_size(a),a:echelon(a),
for i:r thru 2 step -1 do (
pc:0,pcf:false,
for j:1 thru c do (
if (a[i,j]=1 and pcf=false) then (pc:j,pcf:true)),
if pcf then (for j:1 thru i-1 do (a:rowop(a,j,i,a[j,pc])))),
a)$
```

```
(%i1) rref(a):=block([r,c,pc,pcf],[r,c]:matrix_size(a),a:echelon(a),
      for i:r thru 2 step -1 do (
      pc:0,pcf:false,
      for j:1 thru c do (
      if (a[i,j]=1 and pcf=false) then (pc:j,pcf:true)),
      if pcf then (for j:1 thru i-1 do (a:rowop(a,j,i,a[j,pc])))),
      a)$

(%i2) A:matrix([1,2,3,4,5],[4,5,6,7,8],[7,8,9,11,12]);
```

(%o2) $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 6 & 7 & 8 \\ 7 & 8 & 9 & 11 & 12 \end{bmatrix}$

```
(%i3) rref(A);
```

(%o3) $\begin{bmatrix} 1 & 0 & -1 & 0 & -1 \\ 0 & 1 & 2 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$

2.6.4 Modular Matrix Operations

When doing modular arithmetic on matrices or matrix reduction in Maxima, it takes a couple different techniques, most of which we have seen. You simply need to be careful which technique you use for which operation.

When doing modular matrix arithmetic, that is, addition, subtraction, multiplication, and positive powers, simply put the operation inside a `mod` command. Inverse and negative powers require a different method which we will discuss below. One word of caution, the `power_mod` function does not work on matrices, so to take a modular matrix power, you first take the matrix power and then the modulus. So the matrix powers should not be too large.

```
(%i1) M:matrix([1,2,3],[4,5,6],[7,8,9]);
```

(%o1) $$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$$

```
(%i2) A:matrix([1,0,1],[2,1,5],[3,2,0]);
```

(%o2) $$\begin{bmatrix}1 & 0 & 1\\2 & 1 & 5\\3 & 2 & 0\end{bmatrix}$$

```
(%i3) mod(A+M, 7);
```

(%o3) $$\begin{bmatrix}2 & 2 & 4\\6 & 6 & 4\\3 & 3 & 2\end{bmatrix}$$

```
(%i4) mod(A.M, 7);
```

(%o4) $$\begin{bmatrix}1 & 3 & 5\\6 & 0 & 1\\4 & 2 & 0\end{bmatrix}$$

```
(%i5) mod(A^^15, 7);
```

(%o5) $$\begin{bmatrix}2 & 4 & 3\\5 & 2 & 5\\3 & 6 & 5\end{bmatrix}$$

Modular matrix inverses are a little more tricky, we will discuss the technique and then give you a function definition that will do all the steps in a single function.

When studying the determinant in your linear algebra class you may have come across the formula,

$$A^{-1} = \frac{1}{\det(A)}\text{adj}(A)$$

where adj(A) is the adjugate (or classical adjoint) of the matrix. The adjugate of A is the transpose of the cofactor matrix. This can be done modulo n as well. Cofactors are just determinants and determinants are just multiplications and additions, hence we simply do all of our operations modulo n. Taking all of these determinants is very computationally expensive but for moderate sized matrices this is a viable solution.

So if we want to invert the matrix M modulo n, we do the following,

1. Mod the matrix M by the modulus n.
2. Find the determinant of M, and mod it by n.
3. Take the GCD of the determinant and n. If the GCD is not 1 then we know that the determinant is not invertible modulo n and the process stops, since the matrix M will not be invertible modulo n. On the other hand, if the GCD is 1 we continue.
4. Find the inverse of the determinant modulo n.

5. Find the adjugate (or classical adjoint) of the matrix.
6. Find the product of the determinant inverse and the adjugate.
7. Finally, take the matrix from the last step modulo n.

For example,

```
(%i1) M:matrix([1,2,3],[4,5,6],[1,5,1]);
```

(%o1) $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\1 & 5 & 1\end{bmatrix}$

```
(%i2) M:mod(M, 7);
```

(%o2) $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\1 & 5 & 1\end{bmatrix}$

```
(%i3) determinant(M);
```

(%o3) 24

```
(%i4) gcd(24, 7);
```

(%o4) 1

```
(%i5) inv_mod(24, 7);
```

(%o5) 5

```
(%i6) IM:5*adjoint(M);
```

(%o6) $\begin{bmatrix}-125 & 65 & -15\\10 & -10 & 30\\75 & -15 & -15\end{bmatrix}$

```
(%i7) InvM:mod(IM, 7);
```

(%o7) $\begin{bmatrix}1 & 2 & 6\\3 & 4 & 2\\5 & 6 & 6\end{bmatrix}$

```
(%i8) mod(M.InvM, 7);
```

(%o8) $\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$

The above technique is not difficult but it could be lengthy if you had several matrices to invert. The following is a function definition for a function that will do all of these steps.

```
mat_mod_inverse(M, n):=block(
     [TEMPMAT, DET, GCD, INVDET, MADJ, MADJINVDET],
     TEMPMAT:mod(M, n),
     DET:mod(determinant(TEMPMAT), n),
     GCD:gcd(DET, n),
     if GCD # 1 then return (false),
     INVDET:inv_mod(DET, n),
     MADJ:adjoint(TEMPMAT),
     MADJINVDET:INVDET*MADJ,
     mod(MADJINVDET, n)
)$
```

We will not discuss creating function blocks in Maxima, the interested reader can find many references online for programming in Maxima. The function itself is not hard to read, and it is easy to see the steps being done. The syntax for the function is simple, `mat_mod_inverse(M,n)` will invert M modulo n, if the inverse exists. If the inverse does not exist then the function will return false.

```
(%i1) M:matrix([1,2,3],[4,5,6],[1,5,1]);
```

(%o1) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 5 & 1 \end{bmatrix}$

```
(%i2) mat_mod_inverse(M, n):=block(
      [TEMPMAT, DET, GCD, INVDET, MADJ, MADJINVDET],
      TEMPMAT:mod(M, n),
      DET:mod(determinant(TEMPMAT), n),
      GCD:gcd(DET, n),
      if GCD # 1 then return (false),
      INVDET:inv_mod(DET, n),
      MADJ:adjoint(TEMPMAT),
      MADJINVDET:INVDET*MADJ,
      mod(MADJINVDET, n)
      )$

(%i3) mat_mod_inverse(M,7);
```

(%o3) $\begin{bmatrix} 1 & 2 & 6 \\ 3 & 4 & 2 \\ 5 & 6 & 6 \end{bmatrix}$

Modular matrix reduction can be done by using the modulus variable, for prime moduli. Simply set the modulus variable to the desired modulus before invoking the echelon or triangularize commands. Note that when the modulus is not false, the matrix entries are in "balanced" modular format, that is between $-\frac{n}{2}$ and $\frac{n}{2}$. If you want the values to be between 0 and $n-1$, simply apply the `mod` command to the result.

```
(%i1) A:matrix([1,2,3],[4,5,6],[1,0,1]);
```

(%o1) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 0 & 1 \end{bmatrix}$

```
(%i2) modulus:false;
```

(%o2) false

```
(%i3) triangularize(A);
```

(%o3) $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 5 & 2 \\ 0 & 0 & 6 \end{bmatrix}$

```
(%i4) echelon(A);
```

(%o4) $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & \frac{2}{5} \\ 0 & 0 & 1 \end{bmatrix}$

```
(%i5) modulus:5;
```

(%o5) 5

```
(%i6) triangularize(A);
```

(%o6) $\begin{bmatrix} -1 & 0 & 1 \\ 0 & -2 & 1 \\ 0 & 0 & 1 \end{bmatrix}$

```
(%i7) echelon(A);
```

(%o7) $\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$

```
(%i8) determinant(A);
```

(%o8) -6

```
(%i9) modulus:2;
```

(%o9) 2

```
(%i10)triangularize(A);
```

(%o10) $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

```
(%i11)echelon(A);
```

(%o11) $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

```
(%i12)modulus:3;
```

(%o12) 3

```
(%i13) triangularize(A);
```

(%o13) $\begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$

```
(%i14) echelon(A);
```

(%o14) $\begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$

```
(%i15) mod(echelon(A),modulus);
```

(%o15) $\begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$

If your modulus is not prime then you could have a problem. The echelon command may try to invert an element modulo a non-prime number that does not have an inverse, in which case you will get an error. For example,

```
(%i1) A:matrix([1,2,3],[4,5,6],[7,8,9]);
```

(%o1) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

```
(%i2) modulus:6;
```

warning: assigning 6, a non-prime, to 'modulus'
(%o2) 6

```
(%i3) echelon(A);
```

CRECIP: attempted inverse of zero (mod 2)
– an error. To debug this try: debugmode(true);

To take care of the case where we have a composite modulus we can simply write a script that does Gaussian elimination and checks for invertibility modulo n in the reduction process. This script will also work for prime moduli and we will not need to change the modulus variable in Maxima. One note, when using a composite modulus, a matrix may not have an echelon or reduced row echelon form. These scripts will attempt to put a matrix in echelon and reduced echelon form but in the case where the forms are not possible the script will reduce the matrix to be close to echelon or reduced echelon form. We produce two scripts here, the first `mod_echelon(a,n)` takes a matrix a and a modulus n and reduces the matrix to echelon form modulo n, if it can. The second, `mod_rref(a,n)` takes a matrix a and a modulus n and reduces the matrix to reduced row echelon form modulo n, if it can.

```
mod_echelon(a,n):=block([r,c,k,pc,rn,cn,m,pcf,zpos1,zpos2,cm],
[r,c]:matrix_size(a),a:mod(a,n),
pc:1,
for i:1 thru r do (
if (pc > c) then return(),
pcf:false,
for j:i thru r do (
k:a[j,pc],
if (k#0 and gcd(k,n)=1) then (a:rowswap(a,i,j),pcf:true,return())),
if pcf then (
ik:inv_mod(k,n),
for cn:1 thru c do (a[i,cn]:mod(ik*a[i,cn],n)),
for rn:i+1 thru r do (
m:a[rn,pc],
for cn:1 thru c do (a[rn,cn]:mod(-a[i,cn]*m+a[rn,cn],n)))),
pc:pc+1),
for j:1 thru r-1 do (
cm:false,
for i:1 thru r-1 do (
zpos1:c+1,zpos2:c+1,
for cn:1 thru c do (if (a[i,cn]#0 and zpos1=c+1) then (zpos1:cn)),
for cn:1 thru c do (if (a[i+1,cn]#0 and zpos2=c+1) then (zpos2:cn)),
if zpos1 > zpos2 then (a:rowswap(a,i,i+1),cm:true)),
if not cm then return()),
a)$

mod_rref(a,n):=block([r,c,k,pc,rn,cn,m,pcf,npc,zpos1,zpos2,cm],
[r,c]:matrix_size(a),a:mod(a,n),
pc:1,
for i:1 thru r do (
if (pc > c) then return(),
pcf:false,
for j:i thru r do (
k:a[j,pc],
if (k#0 and gcd(k,n)=1) then (a:rowswap(a,i,j),pcf:true,return())),
if pcf then (
ik:inv_mod(k,n),
for cn:1 thru c do (a[i,cn]:mod(ik*a[i,cn],n)),
for rn:i+1 thru r do (
m:a[rn,pc],
for cn:1 thru c do (a[rn,cn]:mod(-a[i,cn]*m+a[rn,cn],n)))),
pc:pc+1),
for i:r thru 2 step -1 do (
pcf:false,npc:false,
for j:1 thru c do (
k:a[i,j],
if (k#0) then (if (k=1) then (pc:j,pcf:true) else npc:true),
if (pcf or npc) then return()),
if pcf then (
for rn:i-1 thru 1 step -1 do (
m:a[rn,pc],
for cn:1 thru c do (a[rn,cn]:mod(-a[i,cn]*m+a[rn,cn],n))))),
for j:1 thru r-1 do (
cm:false,
for i:1 thru r-1 do (
zpos1:c+1,zpos2:c+1,
for cn:1 thru c do (if (a[i,cn]#0 and zpos1=c+1) then (zpos1:cn)),
for cn:1 thru c do (if (a[i+1,cn]#0 and zpos2=c+1) then (zpos2:cn)),
if zpos1 > zpos2 then (a:rowswap(a,i,i+1),cm:true)),
if not cm then return()),
a)$
```

For example,

```
(%i1) mod_echelon(a,n):=block([r,c,k,pc,rn,cn,m,pcf,zpos1,zpos2,cm],
      [r,c]:matrix_size(a),a:mod(a,n),
      pc:1,
      for i:1 thru r do (
      if (pc > c) then return(),
      pcf:false,
      for j:i thru r do (
      k:a[j,pc],
      if (k#0 and gcd(k,n)=1) then (a:rowswap(a,i,j),pcf:true,return())),
      if pcf then (
      ik:inv_mod(k,n),
      for cn:1 thru c do (a[i,cn]:mod(ik*a[i,cn],n)),
      for rn:i+1 thru r do (
      m:a[rn,pc],
      for cn:1 thru c do (a[rn,cn]:mod(-a[i,cn]*m+a[rn,cn],n)))),
      pc:pc+1),
      for j:1 thru r-1 do (
      cm:false,
      for i:1 thru r-1 do (
      zpos1:c+1,zpos2:c+1,
      for cn:1 thru c do (if (a[i,cn]#0 and zpos1=c+1) then (zpos1:cn)),
      for cn:1 thru c do (if (a[i+1,cn]#0 and zpos2=c+1) then (zpos2:cn)),
      if zpos1 > zpos2 then (a:rowswap(a,i,i+1),cm:true)),
      if not cm then return()),
      a)$
```

```
(%i2) mod_rref(a,n):=block([r,c,k,pc,rn,cn,m,pcf,npc,zpos1,zpos2,cm],
      [r,c]:matrix_size(a),a:mod(a,n),
      pc:1,
      for i:1 thru r do (
      if (pc > c) then return(),
      pcf:false,
      for j:i thru r do (
      k:a[j,pc],
      if (k#0 and gcd(k,n)=1) then (a:rowswap(a,i,j),pcf:true,return())),
      if pcf then (
      ik:inv_mod(k,n),
      for cn:1 thru c do (a[i,cn]:mod(ik*a[i,cn],n)),
      for rn:i+1 thru r do (
      m:a[rn,pc],
      for cn:1 thru c do (a[rn,cn]:mod(-a[i,cn]*m+a[rn,cn],n)))),
      pc:pc+1),
      for i:r thru 2 step -1 do (
      pcf:false,npc:false,
      for j:1 thru c do (
      k:a[i,j],
      if (k#0) then (if (k=1) then (pc:j,pcf:true) else npc:true),
      if (pcf or npc) then return()),
      if pcf then (
      for rn:i-1 thru 1 step -1 do (
      m:a[rn,pc],
      for cn:1 thru c do (a[rn,cn]:mod(-a[i,cn]*m+a[rn,cn],n))))),
      for j:1 thru r-1 do (
      cm:false,
      for i:1 thru r-1 do (
      zpos1:c+1,zpos2:c+1,
      for cn:1 thru c do (if (a[i,cn]#0 and zpos1=c+1) then (zpos1:cn)),
      for cn:1 thru c do (if (a[i+1,cn]#0 and zpos2=c+1) then (zpos2:cn)),
      if zpos1 > zpos2 then (a:rowswap(a,i,i+1),cm:true)),
      if not cm then return()),
      a)$

(%i3) A:matrix([1,2,3,4,5],[4,5,6,7,8],[7,8,9,11,12]);
```

(%o3) $\begin{bmatrix}1 & 2 & 3 & 4 & 5\\ 4 & 5 & 6 & 7 & 8\\ 7 & 8 & 9 & 11 & 12\end{bmatrix}$

```
(%i4) mod_echelon(A,26);
```

(%o4) $$\begin{bmatrix}1 & 2 & 3 & 4 & 5\\ 0 & 1 & 2 & 3 & 4\\ 0 & 0 & 0 & 1 & 1\end{bmatrix}$$

```
(%i5) mod_rref(A,26);
```

(%o5) $$\begin{bmatrix}1 & 0 & 25 & 0 & 25\\ 0 & 1 & 2 & 0 & 1\\ 0 & 0 & 0 & 1 & 1\end{bmatrix}$$

```
(%i6) B:matrix([12,25,10,18,7],[0,15,24,24,23],[7,18,7,15,10],
               [17,16,3,0,17],[9,20,19,10,11]);
```

(%o6) $$\begin{bmatrix}12 & 25 & 10 & 18 & 7\\ 0 & 15 & 24 & 24 & 23\\ 7 & 18 & 7 & 15 & 10\\ 17 & 16 & 3 & 0 & 17\\ 9 & 20 & 19 & 10 & 11\end{bmatrix}$$

```
(%i7) mod_echelon(B,26);
```

(%o7) $$\begin{bmatrix}1 & 10 & 1 & 17 & 20\\ 0 & 1 & 12 & 12 & 5\\ 0 & 0 & 20 & 18 & 8\\ 0 & 0 & 12 & 1 & 21\\ 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

```
(%i8) mod_rref(B,26);
```

(%o8) $$\begin{bmatrix}1 & 0 & 11 & 1 & 22\\ 0 & 1 & 12 & 12 & 5\\ 0 & 0 & 20 & 18 & 8\\ 0 & 0 & 12 & 1 & 21\\ 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

```
(%i9) mod_rref(B,2);
```

(%o9) $$\begin{bmatrix}1 & 0 & 1 & 0 & 1\\ 0 & 1 & 0 & 0 & 1\\ 0 & 0 & 0 & 1 & 1\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

```
(%i10)mod_rref(B,13);
```

(%o10) $$\begin{bmatrix}1 & 0 & 0 & 0 & 7\\ 0 & 1 & 0 & 0 & 12\\ 0 & 0 & 1 & 0 & 6\\ 0 & 0 & 0 & 1 & 1\\ 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

2.7 Elliptic Curves

Maxima does not have elliptic curve functions built in but with a little programming we can create enough functionality to do some experimentation with Elliptic Curve Cryptography.

Although a general elliptic curve is represented by

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

we will make the assumption that we can reduce the equation to the form

$$y^2 = x^3 + bx + c$$

We will also be restricting ourselves to modular elliptic curves and hence our function will be working on curves of the form,

$$y^2 = x^3 + bx + c \pmod{n}$$

Most of our functions will take the parameters b, c, and n to define the elliptic curve we are working with.

2.7.1 Points on an Elliptic Curve

To do some basic point finding on an elliptic curve we can use the following functions. The first will find all of the point on the curve except for the point at infinity.

```
ec_points(b,c,n):=block([a,lhsv,rhsv],a:[],
for x:0 thru n-1 do (
for y:0 thru n-1 do (
lhsv:mod(y^2, n),rhsv:mod(x^3+b*x+c, n),
if (lhsv=rhsv) then (a:append(a,[[x,y]])))),
a)$
```

The next function simply adds in the point at infinity. We chose to use Maxima's `inf` infinity for this.

```
ec_allpoints(b,c,n):=block([a,t],a:ec_points(b,c,n),t:inf,
a:append(a,[t]),
a)$
```

To simply return the number of points on the curve we could use the following. Note that this function included the point at infinity as part of the count.

```
ec_order(b,c,n):=block([a,lhsv,rhsv,x,y],a:1,
for x:0 thru n-1 do (
for y:0 thru n-1 do (
lhsv:mod(y^2, n),rhsv:mod(x^3+b*x+c, n),
if (lhsv=rhsv) then (a:a+1))),
a)$
```

If one takes a quick look at the code it is clear, even if you never programmed in Maxima, that these are brute force algorithms and hence not the most efficient. So one should be careful with using large moduli.

For example, if we wanted to find the points on the curve $y^2 = x^3 + x + 1 \pmod{5}$ we could use the following commands.

(% i2) ec_points(1,1,5);

$$[[0,1],[0,4],[2,1],[2,4],[3,1],[3,4],[4,2],[4,3]] \quad (\% \text{ o2})$$

(% i3) ec_allpoints(1,1,5);

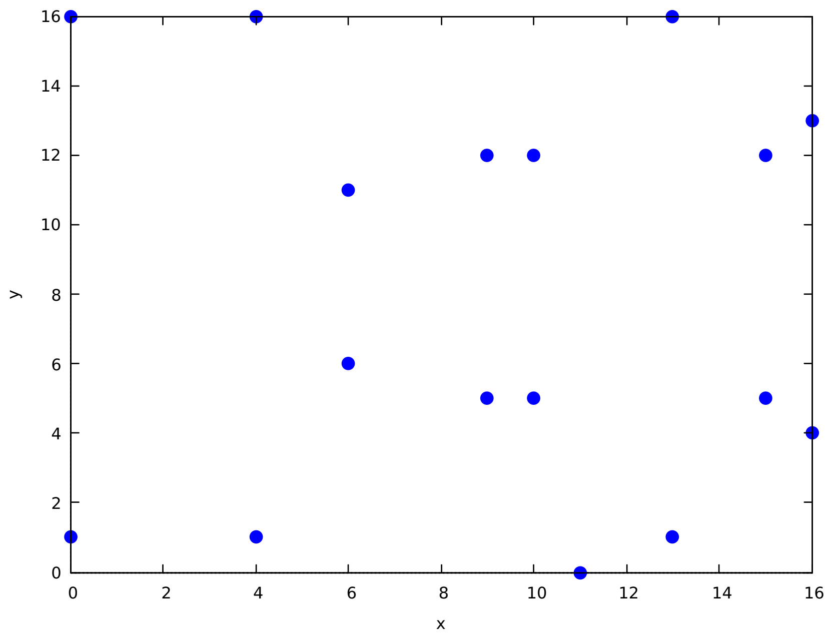
$$[[0,1],[0,4],[2,1],[2,4],[3,1],[3,4],[4,2],[4,3],\infty] \quad (\% \text{ o3})$$

(% i4) ec_order(1,1,5);

$$9 \quad (\% \text{ o4})$$

Note that the output is a list of pair lists. Each pair is a single point on the curve and the single list notation makes it easy to load into other Maxima functions. For example, to plot the points on the curve $y^2 = x^3 + x + 1 \pmod{17}$ we could use the following commands.

(% i4) pts:ec_points(1,1,17)$
(% i5) plot2d([discrete,pts],[style,points]);



We have also included a function that will do the same thing in one step.

```
ec_plot(b,c,n):=block([pts],
    pts:ec_points(b,c,n),
    plot2d([discrete,pts],[style,points])
)$
```

So the command `ec_plot(1,1,17)` will produce the same graph. We can check if a point is on a given curve using,

```
ec_pointOnCurve(b,c,n,pt):=block([lhsv,rhsv],
lhsv:mod(pt[2]^2, n),
rhsv:mod(pt[1]^3+b*pt[1]+c, n),
if (lhsv=rhsv) then true else false)$
```

In this function the point we are checking needs to be an ordered pair in a list, just like the output of the point generators. For example,

(% i9) ec_pointOnCurve(1,1,5,[3,1]);

$$\text{true} \tag{\% o9}$$

(% i10) ec_pointOnCurve(1,1,5,[1,1]);

$$\text{false} \tag{\% o10}$$

We can also find points on a curve given either their x or y coordinate. The following functions will return a point on the curve if one exists or "none" if there is no point on the curve with the given x or y coordinate. Note that these are brute force algorithms so use moduli of a moderate size.

```
ec_pointWithX(b,c,n,x):=block([y,r],r:none,
for y:0 thru n-1 do (
        if (ec_pointOnCurve(b,c,n,[x,y])) then (r:[x,y], y:n)
),r)$

ec_pointWithY(b,c,n,y):=block([x,r],r:none,
for x:0 thru n-1 do (
        if (ec_pointOnCurve(b,c,n,[x,y])) then (r:[x,y], x:n)
),r)$
```

For example, if we wanted to find points on $y^2 = x^3 + x + 1 \pmod 5$ we could use the following commands.

(% i3) ec_pointWithX(1,1,5,2);

$$[2, 1] \tag{\% o3}$$

(% i4) ec_pointWithX(1,1,5,1);

$$none \tag{\% o4}$$

(% i5) ec_pointWithY(1,1,5,1);

$$[0, 1] \tag{\% o5}$$

(% i6) ec_pointWithY(1,1,5,0);

$$none \tag{\% o6}$$

In Elliptic Curve Cryptography it is common to select the linear term and modulus of the curve and a particular point you want on the curve and then calculate the constant term from this information. While this is a simple calculation we created a function to do this.

```
ec_generateCurveConstant(b,n,pt):=block(
    mod(pt[2]^2-(pt[1]^3+b*pt[1]), n)
)$
```

For example, say we wanted the point $(7657, 74389)$ to be on the curve with linear term 3284 and modulus 3263561.

(% i7) ec_generateCurveConstant(3284,3263561,[7657,74389]);

$$1388410 \qquad (\% o7)$$

(% i8) ec_pointOnCurve(3284,1388410,3263561,[7657,74389]);

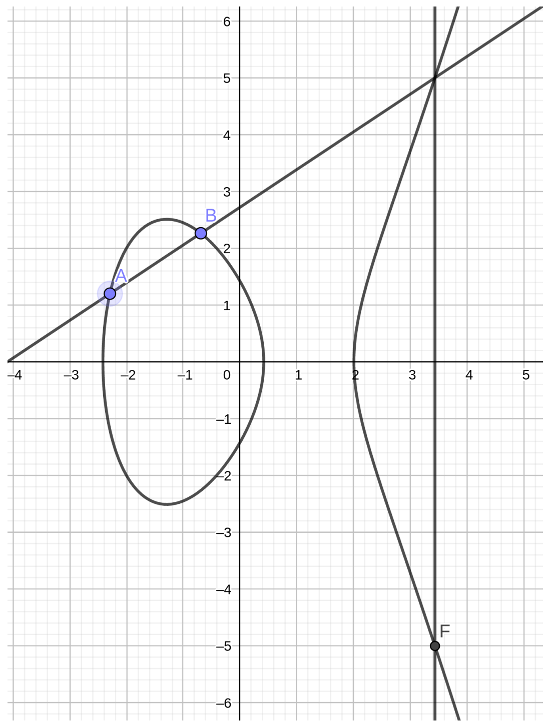
$$\text{true} \qquad (\% o8)$$

We find that the curve $y^2 = x^3 + 3284x + 1388410 \pmod{3263561}$ does the trick.

2.7.2 Arithmetic on an Elliptic Curve

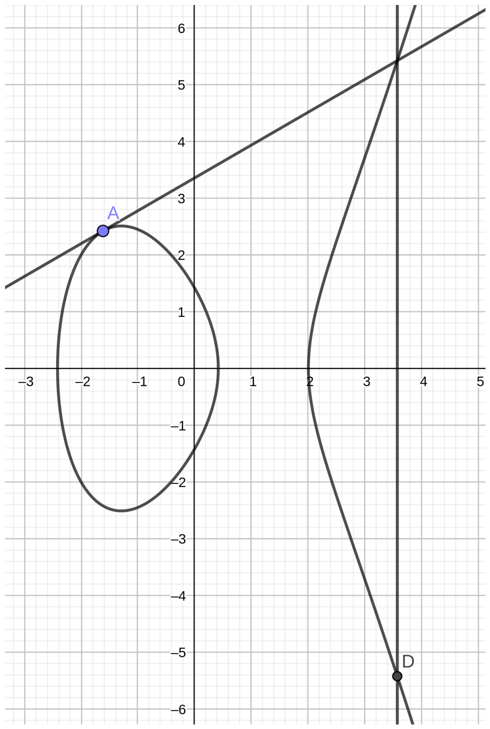
If you have studied elliptic curves you know that there is a method to add two points on an elliptic curve to obtain a third point on the curve. In fact, if you have studied group theory you know that this point addition defines an abelian group structure on the curve. In the case of finite groups this structure is sometimes cyclic. Although we will be dealing with curve modulo n we will briefly discuss the addition law geometrically for elliptic curves in R^2. The addition law in this case has a nice geometrical interpretation that also sheds some light on the formulas.

To add two points on an elliptic curve A and B where $A \neq B$ you first draw a straight line through the two points, this will intersect the curve in another point. Then reflect this point over the x axis to obtain the sum of A and B. So in the diagram on the right we have $A + B = F$.



In the case where $A = B$, in other words we want to calculate $2A$ we take the tangent line to the elliptic curve at A, this will intersect the curve in another point. Then reflect this point over the x axis to obtain $2A$. So in the diagram on the right we have $2A = D$.

In the cases where the line through A and B is vertical or if the tangent line in vertical when calculating $2A$ the sum is the point at infinity, ∞. If we translate this geometric description into algebraic formulas we have the following Addition Law on elliptic curves.



Let E be given by $y^2 = x^3 + bx + c$ and let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then $P_1 + P_2 = P_3 = (x_3, y_3)$ where

$$\begin{aligned} x_3 &= m^2 - x_1 - x_2 \\ y_3 &= m(x_1 - x_3) - y_1 \end{aligned}$$

and

$$m = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2 \\ \frac{3x_1^2 + b}{2y_1} & \text{if } P_1 = P_2 \end{cases}$$

If the slope m is infinite, then $P_3 = \infty$. There is one additional law: $\infty + P = P$ for all points P.

Although these equations were developed using continuous curves and derivatives the same formulas work for the discrete case of finite curves over a modulus. The tricky point here is that in the derivation of m we have either $x_2 - x_1$ or $2y_1$ in the denominator. So if we are working modulo n these values need to have multiplicative inverses modulo n. If they do not have a multiplicative inverse modulo n then the greatest common divisor between them and n is greater then 1 and in some cases this will lead to a factorization of n.

We have created four Maxima functions to do some arithmetic operations. The first is a point addition function. This function will return the sum of the input points if it exists and if not it will return “Error”.

```
ec_pointAdd(b,c,n,p1,p2):=block([a,m,invy,x,y,err],err:false,
if (p1=inf) then return(p2),
if (p2=inf) then return(p1),
if (p1=Error) then err:true,
if (p2=Error) then err:true,
if (err) then a:0 else (
if (p1=p2) then (
                if (mod(2*p1[2],n) =  0 ) then return(inf),
                if (gcd(2*p1[2],n) > 1) then err:true,
                invy:power_mod(2*p1[2],-1,n),
                m:mod((3*p1[1]^2+b)*invy,n)
            ) else (
                if (mod(p1[1],n)=mod(p2[1], n)) then return(inf),
                if (gcd(p1[1]-p2[1],n) > 1) then err:true,
                invy:power_mod(p1[1]-p2[1],-1,n),
                m:mod((p1[2]-p2[2])*invy,n)
            ),
x:mod(m^2-p1[1]-p2[1], n),
y:mod(m*(p1[1]-x)-p1[2], n),
a:[x,y]),
if (err) then Error else a)$
```

For example, say we wanted to add the two points $(2, 1)$ and $(4, 2)$ on the elliptic curve $y^2 = x^3 + x + 1 \pmod 5$. We see that the result is the point $(3, 1)$. Additionally, $(2, 1) + (2, 4) = \infty$ and $2 \cdot (3, 1) = (0, 1)$. Also, if we were to add the two points $(1, 3)$ and $(1771, 705)$ on the elliptic curve $y^2 = x^3 + 4x + 4 \pmod{2773}$. We see that the result is an error. This is because the GCD of $x_2 - x_1 - 1770$ and the modulus 2773 is 59 and hence 1770 is not invertible modulo 2773. The added information is that 59 is a nontrivial factor of 2773. This also shows that if the modulus is not prime (that is the base structure is not a field) then the resulting curve with the addition law does not form a group structure, addition is not closed. It is precisely this fact that is the driver of Lenstra's Elliptic Curve Factorization algorithm.

(% i3) ec_points(1,1,5);

$$[[0, 1], [0, 4], [2, 1], [2, 4], [3, 1], [3, 4], [4, 2], [4, 3]] \qquad (\%\ o3)$$

(% i4) ec_pointAdd(1,1,5,[2,1],[4,2]);

$$[3, 1] \qquad (\%\ o4)$$

(% i5) ec_pointAdd(1,1,5,[2,1],[2,4]);

$$\infty \qquad (\%\ o5)$$

(% i6) ec_pointAdd(1,1,5,[3,1],[3,1]);

$$[0, 1] \qquad (\%\ o6)$$

(% i9) ec_pointAdd(4,4,2773,[1,3],[1771,705]);

$$\mathit{Error} \qquad (\%\ o9)$$

We have created two functions for doing scalar multiplication, the first calculates $t \cdot P$ and the second calculates $t! \cdot P$. The scalar multiple function uses a binary decomposition of the scalar and hence is very fast but the factorial scalar multiple needs to run through each scalar multiple and can be slow for large values of t.

```
ec_pointScalarMult(b,c,n,t,p1):=block([pt,w],pt:inf,w:true,
    if (t < 0) then (t:-t, p1[2]:mod(-p1[2],n)),
    while (w)  do (
        if (mod(t,2)=1) then pt:ec_pointAdd(b,c,n,pt,p1),
        p1:ec_pointAdd(b,c,n,p1,p1),
        t:floor(t/2),
        if (t=0) then w:false
    ),pt)$

ec_pointFactorialScalarMult(b,c,n,t,p1):=block([i,pt],pt:p1,
    for i:2 thru t do (pt:ec_pointScalarMult(b,c,n,i,pt)),
pt)$
```

For example, say we wanted to calculate $5 \cdot (13, 4)$, $738956431 \cdot (13, 4)$, $5! \cdot (13, 4)$, and $20! \cdot (13, 4)$ on the curve $y^2 = x^3 + 2x + 3 \pmod{17}$. The following commands will do these calculations.

(% i3) ec_pointScalarMult(2,3,17,5,[13,4]);

$$[9, 6] \tag{\% o3}$$

(% i4) ec_pointScalarMult(2,3,17,738956431,[13,4]);

$$[5, 11] \tag{\% o4}$$

(% i5) ec_pointFactorialScalarMult(2,3,17,5,[13,4]);

$$[3, 6] \tag{\% o5}$$

(% i6) ec_pointFactorialScalarMult(2,3,17,20,[13,4]);

$$\infty \tag{\% o6}$$

As we pointed out above, elliptic curves with prime modulus form a group structure. In group theory the order of an element is of some importance. We have included another brute force algorithm to calculate the order of a point on the elliptic curve.

```
ec_pointOrder(b,c,n,pt):=block([p,i,r,w],w:true,i:1,
while (w)  do (
        p:ec_pointScalarMult(b,c,n,i,pt),
        if (p=inf) then (r:i, w:false),
        i:i+1
),r)$
```

For example, calculate the order of $(13, 4)$ on the curve $y^2 = x^3 + 2x + 3 \pmod{17}$ we do the following.

(% i7) ec_pointOrder(2,3,17,[13,4]);

$$22 \tag{\% o7}$$

2.8 CryptDS.mac

The Maxima scripts that were discussed in this section are all in the `CryptDS.mac` file that can be found on my web site. To load all of the functions download the `CryptDS.mac` file then in Maxima,

1. Select File > Load Package from the main menu.
2. Navigate to the CryptDS.mac file.
3. Select it and click Open.

At this point all of the functions will be loaded into the Maxima session.

2.8.1 CryptDS.mac Code

```
from_cf(L):=ratsimp(cfdisrep(L))$

from_cf_n(L, n):=ratsimp(cfdisrep(makelist(L[i], i, 1, n)))$

rref(a):=block([r,c,pc,pcf],[r,c]:matrix_size(a),a:echelon(a),
for i:r thru 2 step -1 do (
pc:0,pcf:false,
for j:1 thru c do (
if (a[i,j]=1 and pcf=false) then (pc:j,pcf:true)),
if pcf then (for j:1 thru i-1 do (a:rowop(a,j,i,a[j,pc])))),
a)$

mat_mod_inverse(M, n):=block(
[TEMPMAT, DET, GCD, INVDET, MADJ, MADJINVDET],
TEMPMAT:mod(M, n),
DET:mod(determinant(TEMPMAT), n),
GCD:gcd(DET, n),
if GCD # 1 then return (false),
INVDET:inv_mod(DET, n),
MADJ:adjoint(TEMPMAT),
MADJINVDET:INVDET*MADJ,
mod(MADJINVDET, n)
)$

mod_echelon(a,n):=block([r,c,k,pc,rn,cn,m,pcf,zpos1,zpos2,cm],
[r,c]:matrix_size(a),a:mod(a,n),
pc:1,
for i:1 thru r do (
if (pc > c) then return(),
pcf:false,
for j:i thru r do (
k:a[j,pc],
if (k#0 and gcd(k,n)=1) then (a:rowswap(a,i,j),pcf:true,return())),
if pcf then (
ik:inv_mod(k,n),
for cn:1 thru c do (a[i,cn]:mod(ik*a[i,cn],n)),
for rn:i+1 thru r do (
m:a[rn,pc],
for cn:1 thru c do (a[rn,cn]:mod(-a[i,cn]*m+a[rn,cn],n)))),
pc:pc+1),
for j:1 thru r-1 do (
cm:false,
for i:1 thru r-1 do (
zpos1:c+1,zpos2:c+1,
for cn:1 thru c do (if (a[i,cn]#0 and zpos1=c+1) then (zpos1:cn)),
for cn:1 thru c do (if (a[i+1,cn]#0 and zpos2=c+1) then (zpos2:cn)),
if zpos1 > zpos2 then (a:rowswap(a,i,i+1),cm:true)),
if not cm then return()),
a)$

mod_rref(a,n):=block([r,c,k,pc,rn,cn,m,pcf,npc,zpos1,zpos2,cm],
[r,c]:matrix_size(a),a:mod(a,n),
pc:1,
for i:1 thru r do (
if (pc > c) then return(),
pcf:false,
for j:i thru r do (
k:a[j,pc],
if (k#0 and gcd(k,n)=1) then (a:rowswap(a,i,j),pcf:true,return())),
if pcf then (
ik:inv_mod(k,n),
for cn:1 thru c do (a[i,cn]:mod(ik*a[i,cn],n)),
for rn:i+1 thru r do (
m:a[rn,pc],
for cn:1 thru c do (a[rn,cn]:mod(-a[i,cn]*m+a[rn,cn],n)))),
pc:pc+1),
for i:r thru 2 step -1 do (
pcf:false,npc:false,
for j:1 thru c do (
k:a[i,j],
if (k#0) then (if (k=1) then (pc:j,pcf:true) else npc:true),
if (pcf or npc) then return()),
if pcf then (
for rn:i-1 thru 1 step -1 do (
m:a[rn,pc],
for cn:1 thru c do (a[rn,cn]:mod(-a[i,cn]*m+a[rn,cn],n))))),
for j:1 thru r-1 do (
cm:false,
for i:1 thru r-1 do (
zpos1:c+1,zpos2:c+1,
for cn:1 thru c do (if (a[i,cn]#0 and zpos1=c+1) then (zpos1:cn)),
for cn:1 thru c do (if (a[i+1,cn]#0 and zpos2=c+1) then (zpos2:cn)),
if zpos1 > zpos2 then (a:rowswap(a,i,i+1),cm:true)),
if not cm then return()),
a)$

ec_points(b,c,n):=block([a,lhsv,rhsv],a:[],
for x:0 thru n-1 do (
for y:0 thru n-1 do (
lhsv:mod(y^2, n),rhsv:mod(x^3+b*x+c, n),
if (lhsv=rhsv) then (a:append(a,[[x,y]])))),
a)$

ec_allpoints(b,c,n):=block([a,t],a:ec_points(b,c,n),t:inf,
a:append(a,[t]),
a)$

ec_pointAdd(b,c,n,p1,p2):=block([a,m,invy,x,y,err],err:false,
if (p1=inf) then return(p2),
if (p2=inf) then return(p1),
if (p1=Error) then err:true,
if (p2=Error) then err:true,
if (err) then a:0 else (
if (p1=p2) then (
                if (mod(2*p1[2],n) =  0 ) then return(inf),
                if (gcd(2*p1[2],n) > 1) then err:true,
                invy:power_mod(2*p1[2],-1,n),
                m:mod((3*p1[1]^2+b)*invy,n)
```

```
            ) else (
                if (mod(p1[1],n)=mod(p2[1], n)) then return(inf),
                if (gcd(p1[1]-p2[1],n) > 1) then err:true,
                invy:power_mod(p1[1]-p2[1],-1,n),
                m:mod((p1[2]-p2[2])*invy,n)
            ),
x:mod(m^2-p1[1]-p2[1], n),
y:mod(m*(p1[1]-x)-p1[2], n),
a:[x,y]),
if (err) then Error else a)$

ec_pointScalarMult(b,c,n,t,p1):=block([pt,w],pt:inf,w:true,
    if (t < 0) then (t:-t, p1[2]:mod(-p1[2],n)),
    while (w)  do (
        if (mod(t,2)=1) then pt:ec_pointAdd(b,c,n,pt,p1),
        p1:ec_pointAdd(b,c,n,p1,p1),
        t:floor(t/2),
        if (t=0) then w:false
    ),pt)$

ec_pointOnCurve(b,c,n,pt):=block([lhsv,rhsv],
lhsv:mod(pt[2]^2, n),
rhsv:mod(pt[1]^3+b*pt[1]+c, n),
if (lhsv=rhsv) then true else false)$

ec_pointFactorialScalarMult(b,c,n,t,p1):=block([i,pt],pt:p1,
    for i:2 thru t do (pt:ec_pointScalarMult(b,c,n,i,pt)),
pt)$

ec_order(b,c,n):=block([a,lhsv,rhsv,x,y],a:1,
for x:0 thru n-1 do (
for y:0 thru n-1 do (
lhsv:mod(y^2, n),rhsv:mod(x^3+b*x+c, n),
if (lhsv=rhsv) then (a:a+1))),
a)$

ec_pointWithX(b,c,n,x):=block([y,r],r:none,
for y:0 thru n-1 do (
        if (ec_pointOnCurve(b,c,n,[x,y])) then (r:[x,y], y:n)
),r)$

ec_pointWithY(b,c,n,y):=block([x,r],r:none,
for x:0 thru n-1 do (
        if (ec_pointOnCurve(b,c,n,[x,y])) then (r:[x,y], x:n)
),r)$

ec_generateCurveConstant(b,n,pt):=block(
    mod(pt[2]^2-(pt[1]^3+b*pt[1]), n)
)$

ec_pointOrder(b,c,n,pt):=block([p,i,r],
for i:1 thru 10^100 do (
        p:ec_pointScalarMult(b,c,n,i,pt),
        if (p=inf) then (r:i, i:10^1000)
),r)$

ec_plot(b,c,n):=block([pts],
    pts:ec_points(b,c,n),
    plot2d([discrete,pts],[style,points])
)$
```

Chapter 3

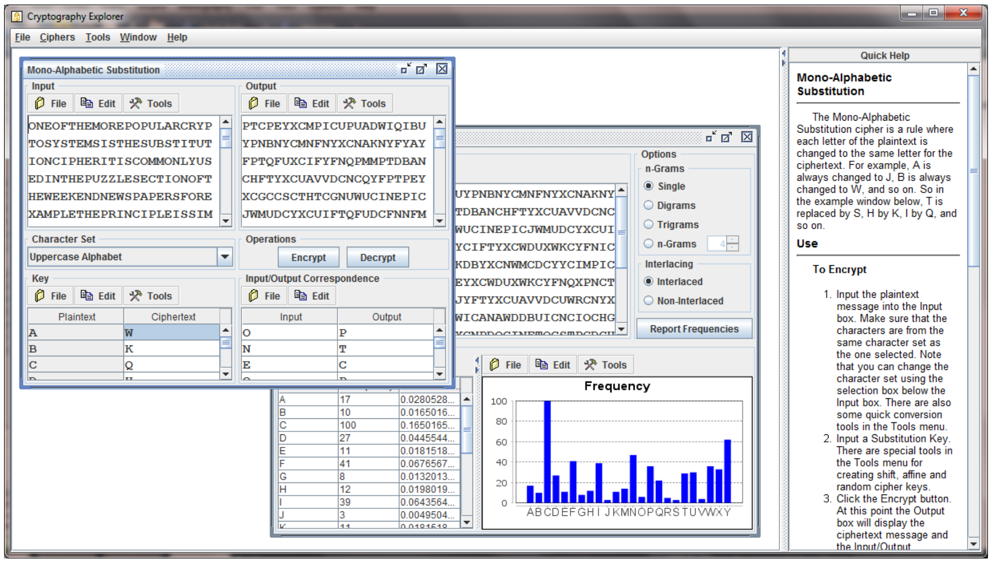
Introduction to Cryptography Explorer

3.1 What is Cryptography Explorer?

Cryptography Explorer is a tool that was developed for the investigation of cryptography and cryptanalysis. It was written mainly to ease the investigation of classical cryptography methods but it also contains features for modern ciphers as well as tools for investigating integer factorization and discrete logarithm calculations.

3.2 Introduction

The main window to the Cryptography Explorer program is pictured below.



```
ONEOFTHEMOREPOPULARCRYP
TOSYSTEMSISTHESUBSTITUT
IONCIPHERITISCOMMONLYUS
EDINTHEPUZZLESECTIONOFT
HEWEEKENDNEWSPAPERSFORE
XAMPLETHEPRINCIPLEISSIM
```

```
PTCPEYXCMPICUPUADWIQIBU
YPNBNYCMNFNYXCNAKNYFYAY
FPTQFUXCIFYFNQPMMPTDBAN
CHFTYXCUAVVDCNCQYFPTPEY
XCGCCSCTHTCGNUWUCINEPIC
JWMUDCYXCUIFTQFUDCFNNFM
```

```
UYPNBNYCMNFNYXCNAKNY
TDBANCHFTYXCUAVVDCNC
WUCINEPICJWMUDCYXCUI
YCIFTYXCWDUXWKCYFNIC
KDBYXCNWMCDCYYCIMPIC
EYXCWDUXWKCYFNQXPNCT
JYFTYXCUAVVDCUWRCNYX
WICANAWDDBUICNCIOCHG
```

Plaintext	Ciphertext
A	W
B	K
C	Q

Input	Output
O	P
N	T
E	C

A	17	0.0280528...
B	10	0.0165016...
C	100	0.1650165...
D	27	0.0445544...
E	11	0.0181518...
F	41	0.0676567...
G	8	0.0132013...
H	12	0.0198019...
I	39	0.0643564...
J	3	0.0049504...

Mono-Alphabetic Substitution

The Mono-Alphabetic Substitution cipher is a rule where each letter of the plaintext is changed to the same letter for the ciphertext. For example, A is always changed to J, B is always changed to W, and so on. So in the example window below, T is replaced by S, H by K, I by Q, and so on.

Use

To Encrypt

1. Input the plaintext message into the Input box. Make sure that the characters are from the same character set as the one selected. Note that you can change the character set using the selection box below the Input box. There are also some quick conversion tools in the Tools menu.
2. Input a Substitution Key. There are special tools in the Tools menu for creating shift, affine and random cipher keys.
3. Click the Encrypt button. At this point the Output box will display the ciphertext message and the Input/Output

Figure 3.1: Cryptography Explorer Main Window

It has a multiple document interface where each cipher and analysis tool has its own child window. There is a standard help system that can be invoked from the Help menu. There is also a quick help bar to the right that displays quick help information for the currently selected cipher or analysis tool. All cipher and analysis tools can be invoked from the main menu of the program. In this getting started guide we will go over each of the tool windows, functions, and options.

Input and Output Boxes

The Input and Output boxes are the same for all cipher and analysis tools in the program. The toolbar at the top of these is really a drop-down menu system. In general, input boxes are editable and output boxes are not editable. With input boxes the standard keystrokes for copy and paste are available, and for output boxes the keystroke for copy is available.

Input boxes have the following menu options. In the Tools menu some of the quick conversion options may not be visible, if that type of conversion is not commonly needed for the cipher or cipher analysis tool. All input box menus contain a tool option of Convert Text which allows the user to select any text conversion currently available in the program.

File —

New: Clears the input box.

Open: Opens a text file and places the contents in the input box.

Save As: Saves the current contents of the input box to a text file.

Print: Prints the current contents of the input box to the selected printer.

Print Preview: Prints the current contents of the input box to the print preview display.

Edit —

Copy: Copies the selected text to the clipboard.

Copy All: Copies the current contents of the input box to the clipboard.

Paste: Pastes the contents of the clipboard to the input box.

Undo: Undoes the last edit.

Redo: Redoes the last edit.

Tools —

Convert to Uppercase: Converts all alphabetic characters to uppercase.

Remove Whitespace: Removes all whitespace in the text, spaces, line brakes, tabs, ...

Remove Punctuation: Removes all punctuation in the text.

Remove Numbers: Removes all numbers in the text.

Remove Double Characters: Removes double characters from the text. This is used primarily in the Playfair cipher and will place a character, usually X, between double characters. For example, FOOD would be converted to FOXOD.

Convert J to I: This will replace all J's with I's and all j's with i's. This is used primarily in the Playfair and ADFGX ciphers.

Convert Text: This opens the text conversion dialog for you to select a special conversion. The program offers many standard conversions of textual information, so you will want to see if the program will automatically do a conversion before you edit the text by hand.

Statistics: Opens a small message box containing character counts and word counts.

Output boxes have the following menu options.

File —

Clear: Clears the input box.

Save As: Saves the current contents of the input box to a text file.

Print: Prints the current contents of the input box to the selected printer.

Print Preview: Prints the current contents of the input box to the print preview display.

Edit —

Copy: Copies the selected text to the clipboard.

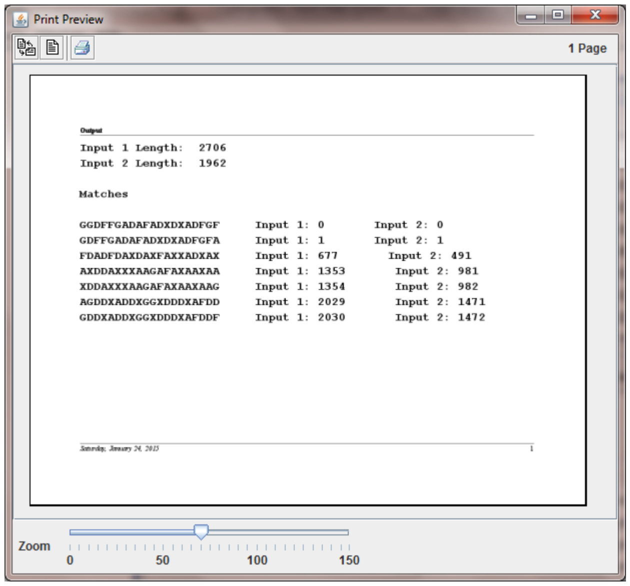
Copy All: Copies the current contents of the input box to the clipboard.

Tools —

Statistics: Opens a small message box containing character counts and word counts.

Print Preview

The Print Preview dialog boxes are the same for all cipher and analysis tools in the program. The toolbar at the top has three tools, the first toggles the page layout between portrait and landscape, the second open a page layout dialog that allows you to select the paper size and set the margins, and the third opens a printer dialog box that allows you to select a printer and print the document. There is a zoom bar at the bottom of the dialog that will change the amount of zoom of the preview images.



```
Output
Input 1 Length:  2706
Input 2 Length:  1962

Matches

GGDFFGADAFADXDXADFGF    Input 1: 0       Input 2: 0
GDFFGADAFADXDXADFGFA    Input 1: 1       Input 2: 1
FDADFDAXDAXFAXXADXAX    Input 1: 677       Input 2: 491
AXDDAXXXAAGAFAXAAXAA    Input 1: 1353       Input 2: 981
XDDAXXXAAGAFAXAAXAAG    Input 1: 1354       Input 2: 982
AGDDXADDXGGXDDDXAFDD    Input 1: 2029       Input 2: 1471
GDDXADDXGGXDDDXAFDDF    Input 1: 2030       Input 2: 1472

Saturday, January 24, 2015                                   1
```

Figure 3.2: Print Preview

Bar Charts

The charts used throughout the program are produced by the same system and hence all have a similar appearance and options. All of them have a popup menu that can be invoked by right-clicking on the chart. This popup menu will allows you to save the chart ad a PNG image, copy the chart to the system clipboard, and to print the chart to a printer. They also have the ability to display the value of the bar by hovering the cursor over the desired bar, as shown below.

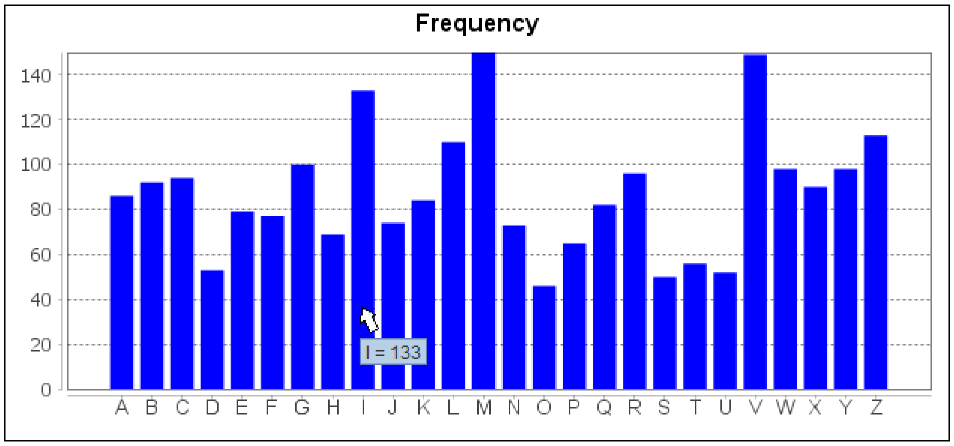


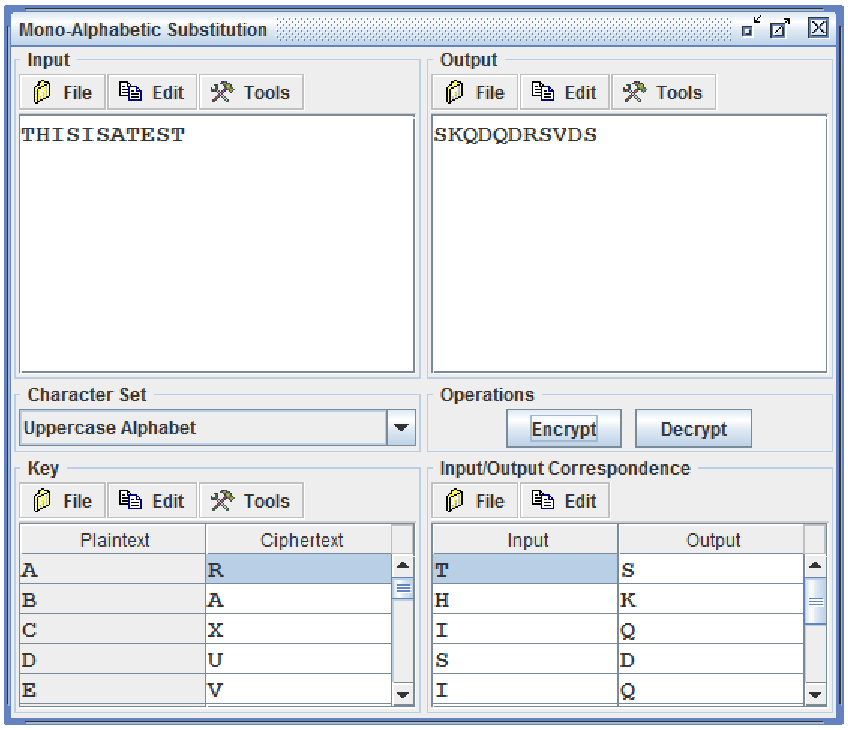
Figure 3.3: Bar Charts

3.3 Ciphers

3.3.1 Monoalphabetic Substitution

The Monoalphabetic Substitution cipher is a rule where each letter of the plaintext is changed to the same letter for the ciphertext. For example, A is always changed to J, B is always changed to W, and so on. So in the example window below, T is replaced by S, H by K, I by Q, and so on.

The Monoalphabetic Substitution window is for creating a simple substitution cipher is below. The upper half of the window contains the input and output boxes, each with their own toolbar/menu. The bottom half of the window contains the options for the cipher and the Input/Output Correspondence.



Input: THISISATEST

Output: SKQDQDRSVDS

Character Set: Uppercase Alphabet

Plaintext	Ciphertext
A	R
B	A
C	X
D	U
E	V

Input	Output
T	S
H	K
I	Q
S	D
I	Q

Figure 3.4: Monoalphabetic Substitution Cipher Tool

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box. Make sure that the characters are from the same character set as the one selected. Note that you can change the character set using the selection box below the Input box. There are also some quick conversion tools in the Tools menu.

2. Input a Substitution Key. There are special tools in the Tools menu for creating shift, affine and random cipher keys.
3. Click the Encrypt button. At this point the Output box will display the ciphertext message and the Input/Output Correspondence table will show the encryption character by character.

To Decrypt —

1. Input the ciphertext message into the Input box. Make sure that the characters are from the same character set as the one selected. Note that you can change the character set using the selection box below the Input box. There are also some quick conversion tools in the Tools menu.
2. Input a Substitution Key. There are special tools in the Tools menu for creating shift, affine and random cipher keys.
3. Click the Decrypt button. At this point the Output box will display the plaintext message and the Input/Output Correspondence table will show the encryption character by character.

Options

- In the lower left quarter of the window is the substitution that will be used for either encoding or decoding and a selection for the character set to use for the substitution. The character sets are as follows:

  Uppercase Alphabet: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

  Uppercase Alphabet with Numbers: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9

  Uppercase & Lowercase Alphabet: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c e d f g h i j k l m n o p q r s t u v w x y z

  Uppercase & Lowercase Alphabet with Numbers: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c e d f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9

  Keyboard Characters: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c e d f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9 ! @ # $ % ˆ & * ( ) + - = [ ] { } — ; ' : , . / < > ?

  User Defined Language: This will open an Open dialog box that will allow you to select a user defined language. Information on creating a user defined language can be found in the User Defined Language Creator tool section.

- The substitution grid has a toolbar with the following options.

  File —

  New: Clears the key grid.

Open: Opens a substitution key file and loads it into the key grid.

Save As: Saves the current key to a substitution key file.

Print: Prints the current key to the selected printer.

Print Preview: Prints the current key to the print preview display.

Edit —

Copy: Copies the entire substitution key grid to the clipboard.

Copy as LaTeX (tabular): Copies the entire substitution key grid to the clipboard using the syntax for the LaTeX tabular environment.

Copy as LaTeX (longtable): Copies the entire substitution key grid to the clipboard using the syntax for the LaTeX longtable environment. ¡/UL¿

Tools —

Create Shift Key: Opens a dialog box to allow the user to select the amount of shift. When the user finishes the shift selection the program will populate the Ciphertext column with the shift key.

Create Affine Key: Opens a dialog box to allow the user to select the multiplier and shift. When the user finishes the input the program will populate the Ciphertext column with the affine key.

Create Random Key: This will populate the Ciphertext column with a random key.

Create Random Kama-Sutra Key: This will populate the Ciphertext column with a random Kama-Sutra key, where each letter is paired with another letter.

- The Input/Output Correspondence grid has a toolbar with the following options.

File —

Save As: Saves the current Input/Output Correspondence grid to a text file.

Print: Prints the current Input/Output Correspondence grid to the selected printer.

Print Preview: Prints the current Input/Output Correspondence grid to the print preview display.

Edit —

Copy: Copies the entire Input/Output Correspondence grid to the clipboard.

Copy as LaTeX (tabular): Copies the entire Input/Output Correspondence grid to the clipboard using the syntax for the LaTeX tabular environment.

Copy as LaTeX (longtable): Copies the entire Input/Output Correspondence grid to the clipboard using the syntax for the LaTeX longtable environment.

- The Encrypt and Decrypt buttons work as follows.

- The Encrypt and Decrypt buttons will, of course, apply the substitution cipher to the input and place the result in the output.
- Encryption will substitute the right hand column for the left. In other words, the the substitution will go from left to right.
- Decryption will substitute the left hand column for the right. In other words, the the substitution will go from right to left.

Notes

- The user can input the substitution by hand as well as use the tools.
- The program is not limited to a one-to-one correspondence between plaintext and ciphertext characters. For example, if there is a character that is not assigned a substitution the program will place an underscore at that position to indicate that the character still needs to be assigned. Likewise, if a character is assigned more than one substitution the program will randomly select one of the options for each of those characters.

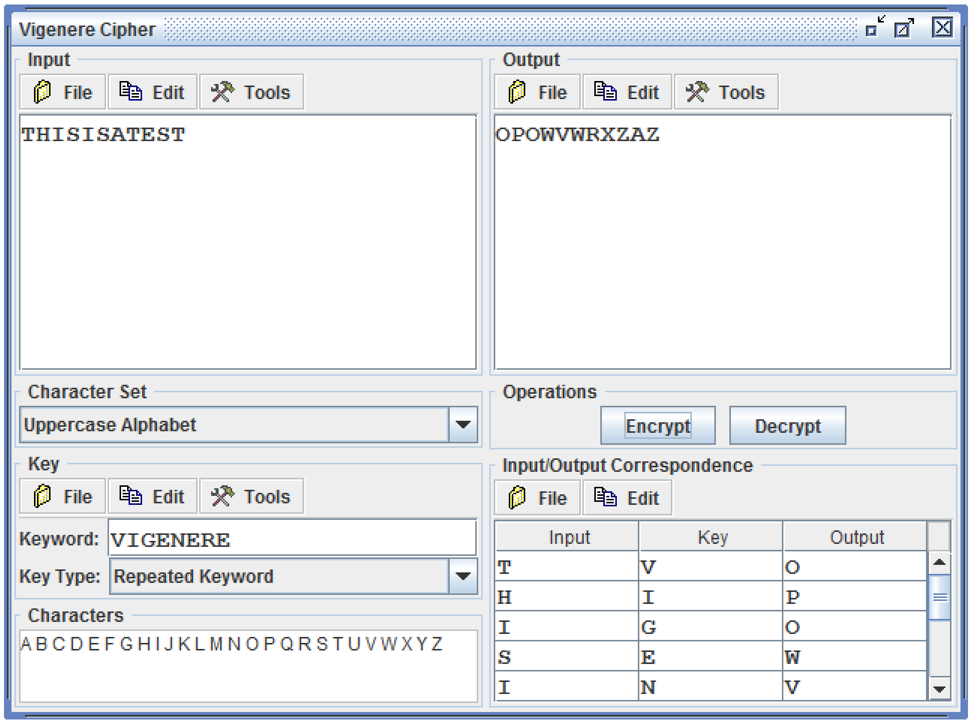
3.3.2 Vigenère

The Vigenère cipher is a method of encrypting alphabetic text by using a series of different Caesar ciphers based on the letters of a keyword. It is a simple form of polyalphabetic substitution. The Vigenère cipher was invented by Giovan Battista Bellaso in 1553 but was later misattributed to Blaise de Vigenère in the 19th century. It was a very strong cipher for the time and was used for several centuries, in fact, it was used by the Confederate Army in the Civil War, even when more secure methods were known.

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box. Make sure that the characters are from the same character set as the one selected. Note that you can change the character set using the selection box below the Input box. There are also some quick conversion tools in the Tools menu.
2. Input a Keyword. The keyword will determine the shift amounts for each position of the plaintext. The keyword must also be from the same character set as the one selected.
3. Select the Key Type. The key type determines how the key is extended to fit the size of the message. The Repeated Keyword option is the classical Vigenère cipher that simply repeats the keyword enough times to cover the message. The Plaintext Autokey option places the plaintext message after the keyword and thus uses the plaintext as the shifts after the keyword is done. The Ciphertext Autokey



THISISATEST

OPOWVWRXZAZ

Character Set: Uppercase Alphabet

Keyword: VIGENERE

Key Type: Repeated Keyword

Characters: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Input	Key	Output
T	V	O
H	I	P
I	G	O
S	E	W
I	N	V

Figure 3.5: Vigenère Cipher Tool

option places the ciphertext after the keyword and thus uses the ciphertext as the shifts after the keyword is done.

4. Click the Encrypt button. At this point the Output box will display the ciphertext message and the Input/Output Correspondence table will show the encryption and key character by character.

To Decrypt —

1. Input the ciphertext message into the Input box. Make sure that the characters are from the same character set as the one selected. Note that you can change the character set using the selection box below the Input box. There are also some quick conversion tools in the Tools menu.
2. Input a Keyword. The keyword will determine the shift amounts for each position of the ciphertext. The keyword must also be from the same character set as the one selected.
3. Select the Key Type. The key type determines how the key is extended to fit the size of the message. The Repeated Keyword option is the classical Vigenère cipher that simply repeats the keyword enough times to cover the message. The Plaintext Autokey option places the plaintext message after the keyword and thus uses the plaintext as the shifts after the keyword is done. The Ciphertext Autokey option places the ciphertext after the keyword and thus uses the ciphertext as the shifts after the keyword is done.

4. Click the Decrypt button. At this point the Output box will display the plaintext message and the Input/Output Correspondence table will show the encryption and key character by character.

Options

- In the lower left quarter of the window is the key that will be used for either encoding or decoding and a selection for the character set to use for the cipher. The character sets are as follows:

  Uppercase Alphabet: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

  Uppercase Alphabet with Numbers: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9

  Uppercase & Lowercase Alphabet: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c e d f g h i j k l m n o p q r s t u v w x y z

  Uppercase & Lowercase Alphabet with Numbers: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c e d f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9

  Keyboard Characters: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c e d f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9 ! @ # $ % ˆ & * ( ) + - = [ ] { } — ; ' : , . / < > ?

  User Defined Language: This will open an Open dialog box that will allow you to select a user defined language. Information on creating a user defined language can be found in the User Defined Language Creator tool section.

- The Keyword has a toolbar with the following options.

  File —

  New: Clears the keyword.

  Open: Opens a keyword file and loads it into the keyword.

  Save As: Saves the current keyword to a keyword file.

  Print: Prints the current keyword to the selected printer.

  Print Preview: Prints the current keyword to the print preview display.

  Edit —

  Copy: Copies the keyword to the clipboard.

  Paste: Pastes the keyword from the clipboard.

  Tools —

  Convert to Uppercase: Converts all alphabetic characters to uppercase.

  Remove Whitespace: Removes all whitespace in the text, spaces, line brakes, tabs, ...

Remove Punctuation: Removes all punctuation in the text.

- The Input/Output Correspondence grid has a toolbar with the following options.

  File —

  Save As: Saves the current Input/Output Correspondence grid to a text file.

  Print: Prints the current Input/Output Correspondence grid to the selected printer.

  Print Preview: Prints the current Input/Output Correspondence grid to the print preview display.

  Edit —

  Copy: Copies the entire Input/Output Correspondence grid to the clipboard.

  Copy as LaTeX (tabular): Copies the entire Input/Output Correspondence grid to the clipboard using the syntax for the LaTeX tabular environment.

  Copy as LaTeX (longtable): Copies the entire Input/Output Correspondence grid to the clipboard using the syntax for the LaTeX longtable environment.

- The Encrypt and Decrypt buttons will, of course, apply the Vigenère cipher to the input and place the result in the output.
  - Encryption will do shifts in the positive direction.
  - Decryption will do shifts in the negative direction.

- The key type determines how the key is extended to fit the size of the message.

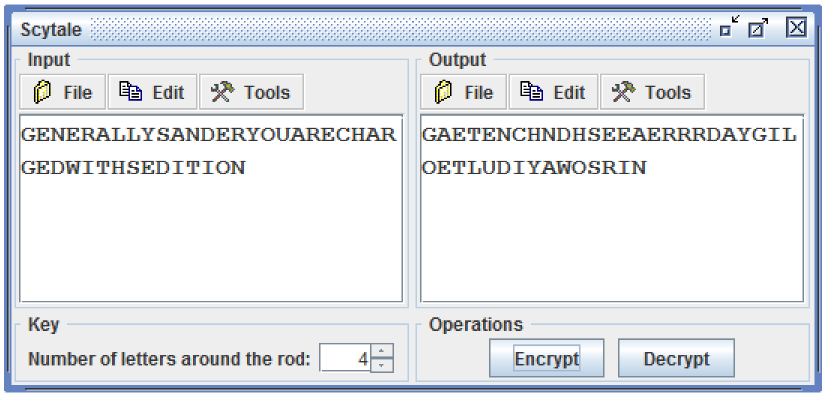
  Repeated Keyword: The Repeated Keyword option is the classical Vigenère cipher that simply repeats the keyword enough times to cover the message.

  Plaintext Autokey: The Plaintext Autokey option places the plaintext message after the keyword and thus uses the plaintext as the shifts after the keyword is done.

  Ciphertext Autokey: The Ciphertext Autokey option places the ciphertext after the keyword and thus uses the ciphertext as the shifts after the keyword is done.

3.3.3 Scytale

The Scytale cipher consisted of a tapered wooden staff around which a strip of parchment (leather or papyrus were also used) was spirally wrapped, layer upon layer. The secret message was written on the parchment lengthwise down the staff. Then the parchment was unwrapped and sent. By themselves, the letters on the parchment were disconnected and made no sense until rewrapped around a staff of equal proportions, at which time the letters would realign to once again make sense.



```
GENERALLYSANDERYOUARECHAR
GEDWITHSEDITION
```

```
GAETENCHNDHSEEAERRRDAYGIL
OETLUDIYAWOSRIN
```

Number of letters around the rod: 4

Figure 3.6: Scytale Cipher Tool

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box.
2. Input the number of letters that are written around the rod before coming back to the starting position.
3. Click the Encrypt button. At this point the Output box will display the ciphertext message.

To Decrypt —

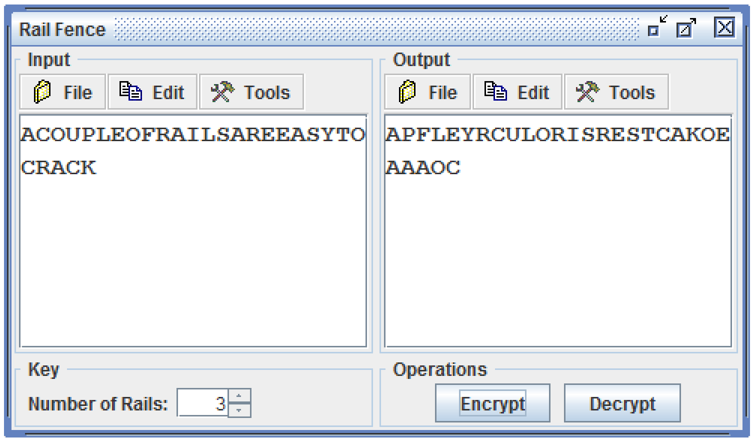
1. Input the ciphertext message into the Input box.
2. Input the number of letters that are written around the rod before coming back to the starting position.
3. Click the Decrypt button. At this point the Output box will display the plaintext message.

3.3.4 Rail Fence

The Rail Fence cipher, like the Scytale cipher, is a transposition cipher. There are several ways that the rail fence can be set up, we use the most common method found in the literature, also known as the zig-zag cipher. Say we are using 3 rails and we encrypt the message A COUPLE OF RAILS ARE EASY TO CRACK. Removing the white-space (which is not necessary) we get ACOUPLEOFRAILSAREEASYTOCRACK, now we zig-zag the message on three rails or rows of a grid.

A				P				F				L				E				Y				R			
	C		U		L		O		R		I		S		R		E		S		T		C		A		K
		O				E				A				A				A				O				C	

We then write the ciphertext as the rails or rows in order from left to right, to get APFLEYRCULORISRESTCAKOEAAAOC.



```
ACOUPLEOFRAILSAREEASYTO
CRACK
```

```
APFLEYRCULORISRESTCAKOE
AAAOC
```

Number of Rails: 3

Figure 3.7: Rail Fence Cipher Tool

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box.
2. Input the number of rails being used.
3. Click the Encrypt button. At this point the Output box will display the ciphertext message.

To Decrypt —

1. Input the ciphertext message into the Input box.
2. Input the number of rails being used.
3. Click the Decrypt button. At this point the Output box will display the plaintext message.

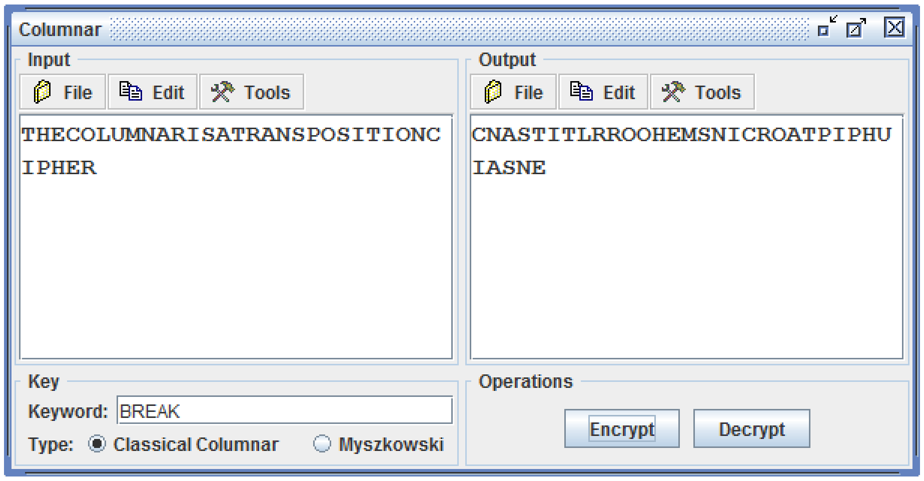
3.3.5 Columnar

The Columnar Cipher is a symmetric transposition cipher that was used through the 1950's either by itself, multiple times, or in conjunction with substitution techniques.

The program offers the user two methods, the classical columnar and the Myszkowski method. For the classical columnar the keyword cannot have any repeated letters in it. The keyword characters become the headers of columns, the message is written left to right and from top to bottom in these columns. Finally, the columns are read in alphabetical order to create the ciphertext. For example, say our keyword is BREAK and the message is THECOLUMNARISATRANSPOSITIONCIPHER. The grid is set up as follows,

B	**R**	**E**	**A**	**K**
T	H	E	C	O
L	U	M	N	A
R	I	S	A	T
R	A	N	S	P
O	S	I	T	I
O	N	C	I	P
H	E	R		

Then the columns are read in alphabetical order, so we get CNASTI TLRROOH EMSNICR OATPIP HUIASNE, and then removing spaces gives us the ciphertext of CNASTITLRROOHEMSNICROATPIPHUIASNE.



```
THECOLUMNARISATRANSPOSITIONC
IPHER
```

```
CNASTITLRROOHEMSNICROATPIPHU
IASNE
```

Key
Keyword: BREAK
Type: [✓ Classical Columnar] Myszkowski
Operations

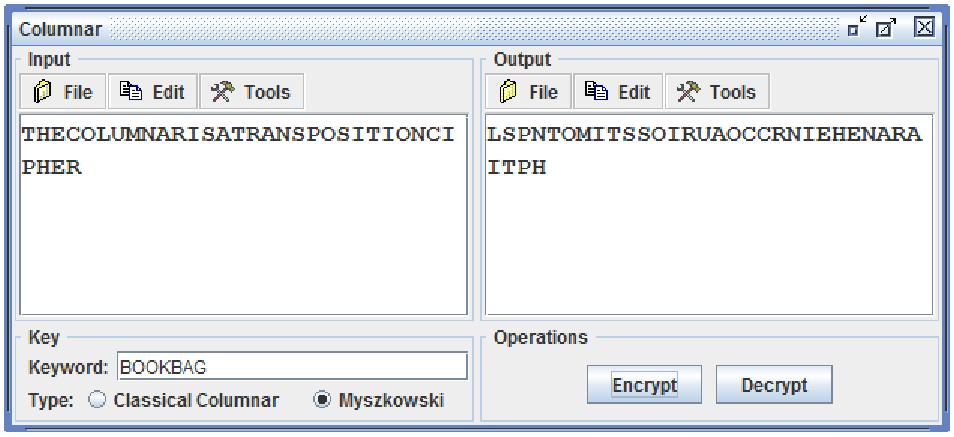
Figure 3.8: Columnar Cipher Tool: Classical Mode

A variant of the classical columnar cipher was developed by Émile Victor Théodore Myszkowski in 1902. With the Myszkowski method, duplicate characters in the keyword are allowed. In the case of duplications, the ciphertext is written left to right between the duplicate columns. With unique letters the column is read as with the classical method. For example, say our keyword is BOOKBAG and the message is again THECOLUMNARISATRANSPOSITIONCIPHER. The grid is set up as follows,

B	**O**	**O**	**K**	**B**	**A**	**G**
T	H	E	C	O	L	U
M	N	A	R	I	S	A
T	R	A	N	S	P	O
S	I	T	I	O	N	C
I	P	H	E	R		

The first column to read is the A column, and there is only one of these, so we get LSPN. Next is the B columns, with two of these we read each row and get, TO, MI, TS, SO, and IR, making TOMITSSOIR. Then the G column, UAOC, then the K column CRNIE. Finally, the O column which gives, HE, NA, RA, IT, and PH, making HENARAITPH. The final ciphertext would be LSPNTOMITSSOIRUAOCCRNIEHENARAITPH.

Decryption of either method is simply done in reverse.



THECOLUMNARISATRANSPOSITIONCI
PHER

LSPNTOMITSSOIRUAOCCRNIEHENARA
ITPH

Keyword: BOOKBAG
Type: ○ Classical Columnar ● Myszkowski

Figure 3.9: Columnar Cipher Tool: Myszkowski Mode

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box.
2. Input a Keyword and select the type of columnar algorithm.
3. Click the Encrypt button. At this point the Output box will display the ciphertext.

To Decrypt —

1. Input the ciphertext message into the Input box.
2. Input a Keyword and select the type of columnar algorithm.
3. Click the Decrypt button. At this point the Output box will display the plaintext.

Options

- The Type specifies the algorithm that is used, either the classical columnar or the Myszkowski method.

Notes

- The keyword need not be only upper-case letters. The columns are ordered by the character's ASCII number and hence any keyboard character can be used in the keyword.

Conventionally, the keyword is all upper-case alphabetic letters, but it is not required. A note about using ASCII numbers for ordering is that A and a have different ASCII numbers so those columns would not be considered to have the same letter.

3.3.6 Two Square

The Two Square Cipher, like the Playfair cipher, is a digram substitution symmetric encryption technique.

There are several slightly different ways to implement this cipher. The one we use equates I and J in order to get 26 letters down to 25, to fit in the 5 X 5 grid. So the plaintext cannot have any J's in it. There is a tool in the input toolbar that will automatically convert J to I. Also like the Playfair cipher, the number of characters in the plaintext must be even, since the technique uses a digram substitution. This tool also supports both vertical and horizontal alignment of the 5 X 5 grids.

The Two Square cipher uses two 5 X 5 grids of letters arranged either vertically or horizontally. The 5 X 5 grids are the key matrices. The program allows the user to input the key matrices either by inputting the matrix or by inputting a key word. If the user inputs a keyword then the matrix is created using the same method as with the Playfair cipher. The keyword is altered by changing all J's to I's and then all repeated letters are removed, so the keyword FOOD is replaced with FOD and the keyword EXAMPLE is replaced by EXAMPL. The matrix is then formed by placing the keyword at the beginning and then filling the remainder of the matrix with the rest of the alphabet letters in order. So the keywords, TWO and SQUARE produce the matrices,

T	W	O	A	B
C	D	E	F	G
H	I	K	L	M
N	P	Q	R	S
U	V	X	Y	Z

and

S	Q	U	A	R
E	B	C	D	F
G	H	I	K	L
M	N	O	P	T
V	W	X	Y	Z

Then when placed in the larger grid, it becomes either,

T	W	O	A	B
C	D	E	F	G
H	I	K	L	M
N	P	Q	R	S
U	V	X	Y	Z

S	Q	U	A	R
E	B	C	D	F
G	H	I	K	L
M	N	O	P	T
V	W	X	Y	Z

or

T	W	O	A	B	S	Q	U	A	R
C	D	E	F	G	E	B	C	D	F
H	I	K	L	M	G	H	I	K	L
N	P	Q	R	S	M	N	O	P	T
U	V	X	Y	Z	V	W	X	Y	Z

To encrypt find the positions of the digram characters, the first from either the top or left grid and the second from the right or lower grid. Using these two positions, create a rectangle, and read off the letters at the other two vertices, left or top followed by right or lower. These are your ciphertext letters. If the digram is in the same column or the same row the digram is unchanged, so some digrams will be the same in the plaintext the the ciphertext.

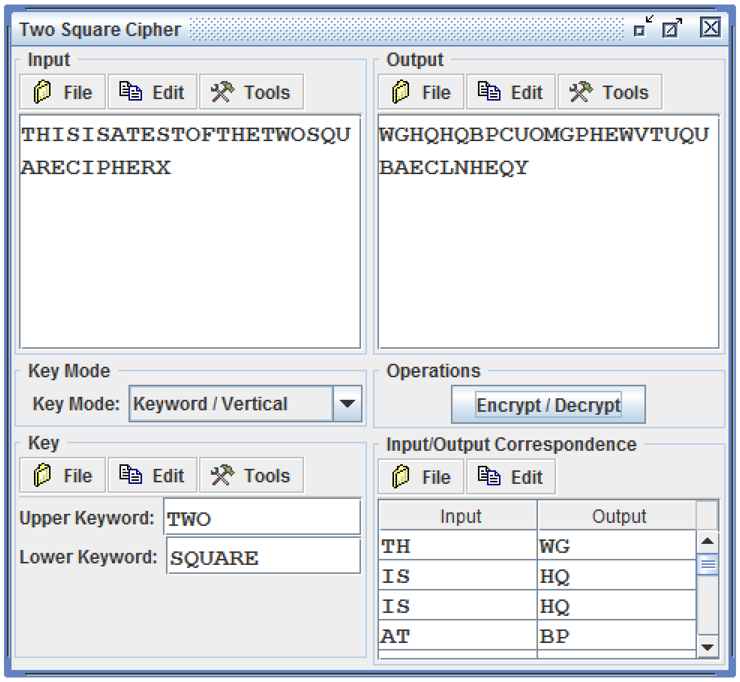
So in vertical mode, TH becomes WG, IS becomes HQ, AD becomes AD, and so on. In horizontal mode, TH becomes HQ, IS becomes WG, TR becomes TR, and so on. The decryption process is exactly like the encryption process.

In key matrix mode, the keyword input is replaced with two 5 X 5 grids.

How to Use the Tool

To Encrypt or Decrypt —

1. Input the plaintext (or ciphertext) message into the Input box. Make sure that the characters are all uppercase and all J's are converted to I's. Note that the Input toolbar has an option for this conversion.
2. Select either the Keyword or Key Matrix mode or operation.
3. Input a Keyword or Matrix.
4. Click the Encrypt/Decrypt button. At this point the Output box will display the ciphertext (or plaintext) message and the Input/Output Correspondence table will show the encryption/decryption character pair by character pair.



Input: THISISATESTOFTHETWOSQUARECIPHERX

Output: WGHQHQBPCUOMGPHEWVTUQUBAECLNHEQY

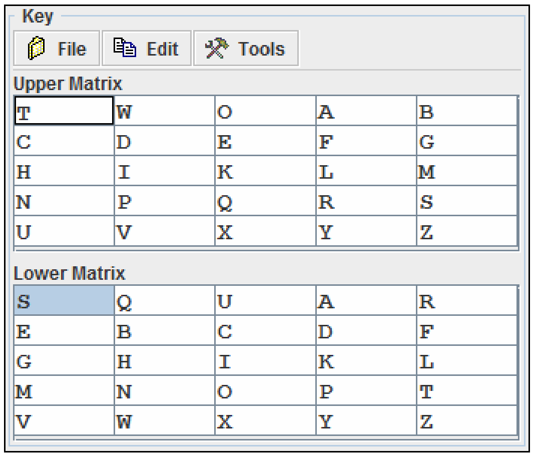
Key Mode: Keyword / Vertical

Upper Keyword: TWO

Lower Keyword: SQUARE

Input	Output
TH	WG
IS	HQ
IS	HQ
AT	BP

Figure 3.10: Two Square Cipher Tool: Keyword Mode



Upper Matrix

T	W	O	A	B
C	D	E	F	G
H	I	K	L	M
N	P	Q	R	S
U	V	X	Y	Z

Lower Matrix

S	Q	U	A	R
E	B	C	D	F
G	H	I	K	L
M	N	O	P	T
V	W	X	Y	Z

Figure 3.11: Two Square Cipher Tool: Key Matrix Mode

Options

- The Key Mode can be set to either Keyword or Key Matrix mode and either vertical or horizontal alignment. With the Keyword mode, the user inputs two keywords, one for the upper right grid and one for the lower left grid. The program automatically converts the keyword to uppercase, ignores and repeated characters and changes any J's to I's. These conversions are done internally, so the user does not see any change in the keyword. The altered keyword is then used to fill out the 5 X 5 grids by starting on row one left to right and moving to subsequent rows when necessary. When the characters of the keyword have been used, the remaining cells in the grid are filled in with the missing letters going in alphabetical order.

- The Key Matrix mode allows the user to create the matrix by hand. This mode comes in handy when one was trying to break a Two Square cipher. In this mode the user needs to enter the characters into each cell of the table.

- In Keyword mode the Key menu has the following options.

  File —

  New: Clears the keywords.

  Open: Opens a keyword file and loads it into the keywords.

  Save As: Saves the current keywords to a keyword file.

  Print: Prints the current keywords to the selected printer.

  Print Preview: Prints the current keywords to the print preview display.

  Edit —

  Copy: Copies the keywords to the clipboard.

  Paste: Pastes the keywords from the clipboard.

  Tools —

  Convert to Uppercase: Converts all alphabetic characters to uppercase.

  Remove Whitespace: Removes all whitespace in the text, spaces, line brakes, tabs, ...

  Remove Punctuation: Removes all punctuation in the text.

- In Key Matrix mode the Key menu has the following options.

  File —

  New: Clears the matrices.

  Open: Opens a key matrix file and loads it into the two tables.

  Save As: Saves the current matrices to a key matrix file.

  Print: Prints the current matrices to the selected printer.

  Print Preview: Prints the current matrices to the print preview display.

  Edit —

  Copy: Copies the entire key grids to the clipboard.

  Copy as LaTeX (tabular): Copies the entire key grids to the clipboard using the syntax for the LaTeX tabular environment.

  Copy as LaTeX (longtable): Copies the entire key grids to the clipboard using the syntax for the LaTeX longtable environment.

  Tools —

Convert to Uppercase: Converts all matrix cells to uppercase.

Check Matrices: Checks the validity of the current tables and displays a message of either valid or invalid.

- The Input/Output Correspondence grid has a toolbar with the following options.

  File —

  Save As: Saves the current Input/Output Correspondence grid to a text file.

  Print: Prints the current Input/Output Correspondence grid to the selected printer.

  Print Preview: Prints the current Input/Output Correspondence grid to the print preview display.

  Edit —

  Copy: Copies the entire Input/Output Correspondence grid to the clipboard.

  Copy as LaTeX (tabular): Copies the entire Input/Output Correspondence grid to the clipboard using the syntax for the LaTeX tabular environment.

  Copy as LaTeX (longtable): Copies the entire Input/Output Correspondence grid to the clipboard using the syntax for the LaTeX longtable environment.

Notes

- There are several slightly different ways to implement this cipher. The one we use does the following.
  - We equate I and J in order to get 26 letters down to 25, so the plaintext cannot have any J's in it.
  - As with all Playfair ciphers, when the plaintext is broken into blocks of 2, so the size of the message must be even, you may need to pad a message with a character.

3.3.7 Four Square

The Four Square Cipher, like the Playfair cipher, is a digram substitution symmetric encryption technique.

There are several slightly different ways to implement this cipher. The one we use equates I and J in order to get 26 letters down to 25, to fit in the 5 X 5 grid. So the plaintext cannot have any J's in it. There is a tool in the input toolbar that will automatically convert J to I. Also like the Playfair cipher, the number of characters in the plaintext must be even, since the technique uses a digram substitution.

The Four Square cipher uses four 5 X 5 grids of letters arranged in a 2 X 2 grid. The upper left and lower right 5 X 5 grids are the alphabet, in order with I = J, reading left to right and top to bottom. Specifically,

a	b	c	d	e
f	g	h	i	k
l	m	n	o	p
q	r	s	t	u
v	w	x	y	z

The upper right and lower left 5 X 5 grids are the key matrices. The program allows the user to input the key matrices either by inputting the matrix or by inputting a key word. If the user inputs a keyword then the matrix is created using the same method as with the Playfair cipher. The keyword is altered by changing all J's to I's and then all repeated letters are removed, so the keyword FOOD is replaced with FOD and the keyword EXAMPLE is replaced by EXAMPL. The matrix is then formed by placing the keyword at the beginning and then filling the remainder of the matrix with the rest of the alphabet letters in order. So the keywords, FOUR and SQUARE produce the matrices

F	O	U	R	A
B	C	D	E	G
H	I	K	L	M
N	P	Q	S	T
V	W	X	Y	Z

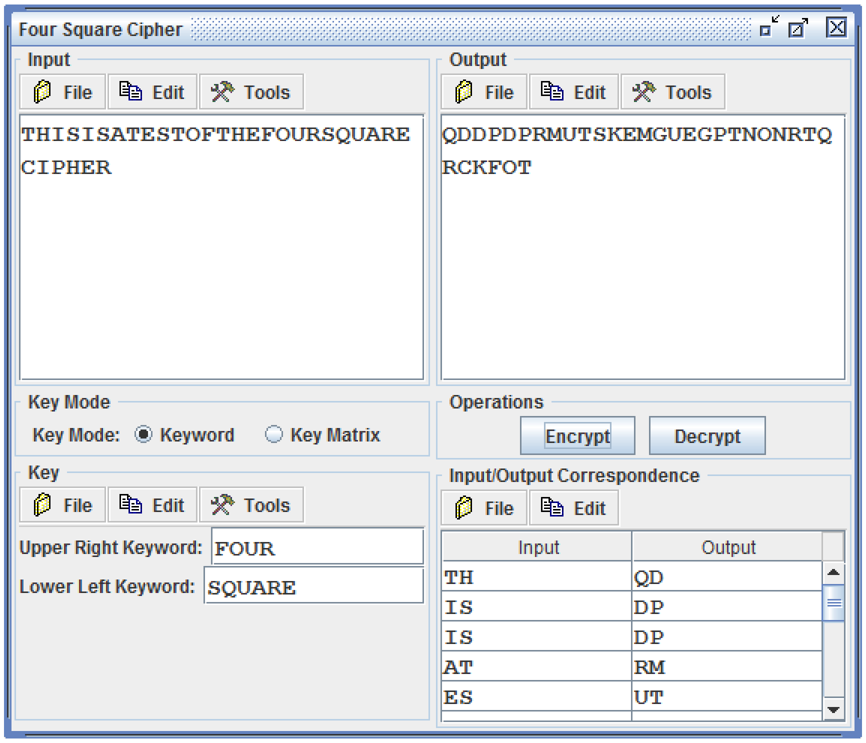
and

S	Q	U	A	R
E	B	C	D	F
G	H	I	K	L
M	N	O	P	T
V	W	X	Y	Z

Then when placed in the larger grid, it becomes,

a	b	c	d	e	F	O	U	R	A
f	g	h	i	k	B	C	D	E	G
l	m	n	o	p	H	I	K	L	M
q	r	s	t	u	N	P	Q	S	T
v	w	x	y	z	V	W	X	Y	Z
S	Q	U	A	R	a	b	c	d	e
E	B	C	D	F	f	g	h	i	k
G	H	I	K	L	l	m	n	o	p
M	N	O	P	T	q	r	s	t	u
V	W	X	Y	Z	v	w	x	y	z

To encrypt find the positions of the digram characters in the plain alphabet grids, upper left for the first letter and lower right for the second. Using these two positions, create a



Input

```
THISISATESTOFTHEFOURSQUARE
CIPHER
```

Output

```
QDDPDPRMUTSKEMGUEGPTNONRTQ
RCKFOT
```

Key Mode: [✓ Keyword] Key Matrix

Upper Right Keyword: FOUR

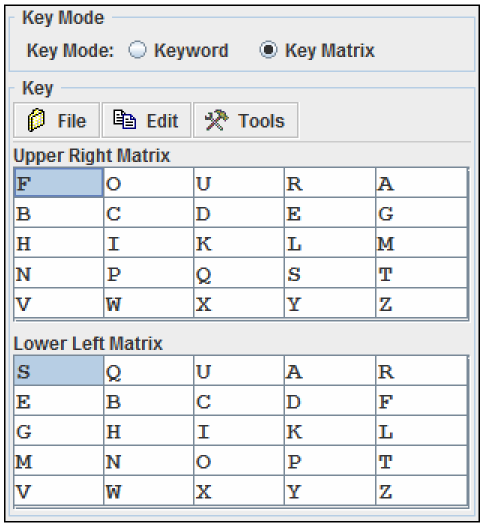
Lower Left Keyword: SQUARE

Input	Output
TH	QD
IS	DP
IS	DP
AT	RM
ES	UT

Figure 3.12: Four Square Cipher Tool: Keyword Mode

rectangle, and read off the letters at the other two vertices, upper right followed by lower left. These are your ciphertext letters. So TH becomes QD, IS becomes DP, and so on. To decrypt, the process is simply reversed, the input digram is found in the key matrices, create the rectangle, and read off the plaintext from the alphabet grids.

In key matrix mode, the keyword input is replaced with two 5 X 5 grids.



Key Mode: Keyword [✓ Key Matrix]

Upper Right Matrix

F	O	U	R	A
B	C	D	E	G
H	I	K	L	M
N	P	Q	S	T
V	W	X	Y	Z

Lower Left Matrix

S	Q	U	A	R
E	B	C	D	F
G	H	I	K	L
M	N	O	P	T
V	W	X	Y	Z

Figure 3.13: Four Square Cipher Tool: Matrix Mode

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box. Make sure that the characters are all uppercase and all J's are converted to I's. Note that the Input toolbar has an option for this conversion.
2. Select either the Keyword or Key Matrix mode or operation.
3. Input a Keyword or Matrix.
4. Click the Encrypt button. At this point the Output box will display the ciphertext message and the Input/Output Correspondence table will show the encryption character pair by character pair.

To Decrypt —

1. Input the ciphertext message into the Input box. Make sure that the characters are all uppercase and all J's are converted to I's. Note that the Input toolbar has an option for this conversion.
2. Select either the Keyword or Key Matrix mode or operation.
3. Input a Keyword or Matrix.
4. Click the Decrypt button. At this point the Output box will display the plaintext message and the Input/Output Correspondence table will show the decryption character pair by character pair.

Options

- The Key Mode can be set to either Keyword or Key Matrix mode. With the Keyword mode, the user inputs two keywords, one for the upper right grid and one for the lower left grid. The program automatically converts the keyword to uppercase, ignores and repeated characters and changes any J's to I's. These conversions are done internally, so the user does not see any change in the keyword. The altered keyword is then used to fill out the 5 X 5 grids by starting on row one left to right and moving to subsequent rows when necessary. When the characters of the keyword have been used, the remaining cells in the grid are filled in with the missing letters going in alphabetical order.
- The Key Matrix mode allows the user to create the matrix by hand. This mode comes in handy when one was trying to break a Four Square cipher. In this mode the user needs to enter the characters into each cell of the table.
- In Keyword mode the Key menu has the following options.

  File —

  New: Clears the keywords.

Open: Opens a keyword file and loads it into the keywords.

Save As: Saves the current keywords to a keyword file.

Print: Prints the current keywords to the selected printer.

Print Preview: Prints the current keywords to the print preview display.

Edit —

Copy: Copies the keywords to the clipboard.

Paste: Pastes the keywords from the clipboard.

Tools —

Convert to Uppercase: Converts all alphabetic characters to uppercase.

Remove Whitespace: Removes all whitespace in the text, spaces, line brakes, tabs, ...

Remove Punctuation: Removes all punctuation in the text.

- In Key Matrix mode the Key menu has the following options.

File —

New: Clears the matrices.

Open: Opens a key matrix file and loads it into the two tables.

Save As: Saves the current matrices to a key matrix file.

Print: Prints the current matrices to the selected printer.

Print Preview: Prints the current matrices to the print preview display.

Edit —

Copy: Copies the entire key grids to the clipboard.

Copy as LaTeX (tabular): Copies the entire key grids to the clipboard using the syntax for the LaTeX tabular environment.

Copy as LaTeX (longtable): Copies the entire key grids to the clipboard using the syntax for the LaTeX longtable environment.

Tools —

Convert to Uppercase: Converts all matrix cells to uppercase.

Check Matrices: Checks the validity of the current tables and displays a message of either valid or invalid.

- The Input/Output Correspondence grid has a toolbar with the following options.

File —

Save As: Saves the current Input/Output Correspondence grid to a text file.

Print: Prints the current Input/Output Correspondence grid to the selected printer.

Print Preview: Prints the current Input/Output Correspondence grid to the print preview display.

Edit —

Copy: Copies the entire Input/Output Correspondence grid to the clipboard.

Copy as LaTeX (tabular): Copies the entire Input/Output Correspondence grid to the clipboard using the syntax for the LaTeX tabular environment.

Copy as LaTeX (longtable): Copies the entire Input/Output Correspondence grid to the clipboard using the syntax for the LaTeX longtable environment.

Notes

- There are several slightly different ways to implement this cipher. The one we use does the following.
  - We equate I and J in order to get 26 letters down to 25, so the plaintext cannot have any J's in it.
  - As with all Playfair ciphers, when the plaintext is broken into blocks of 2, so the size of the message must be even, you may need to pad a message with a character.

3.3.8 Playfair

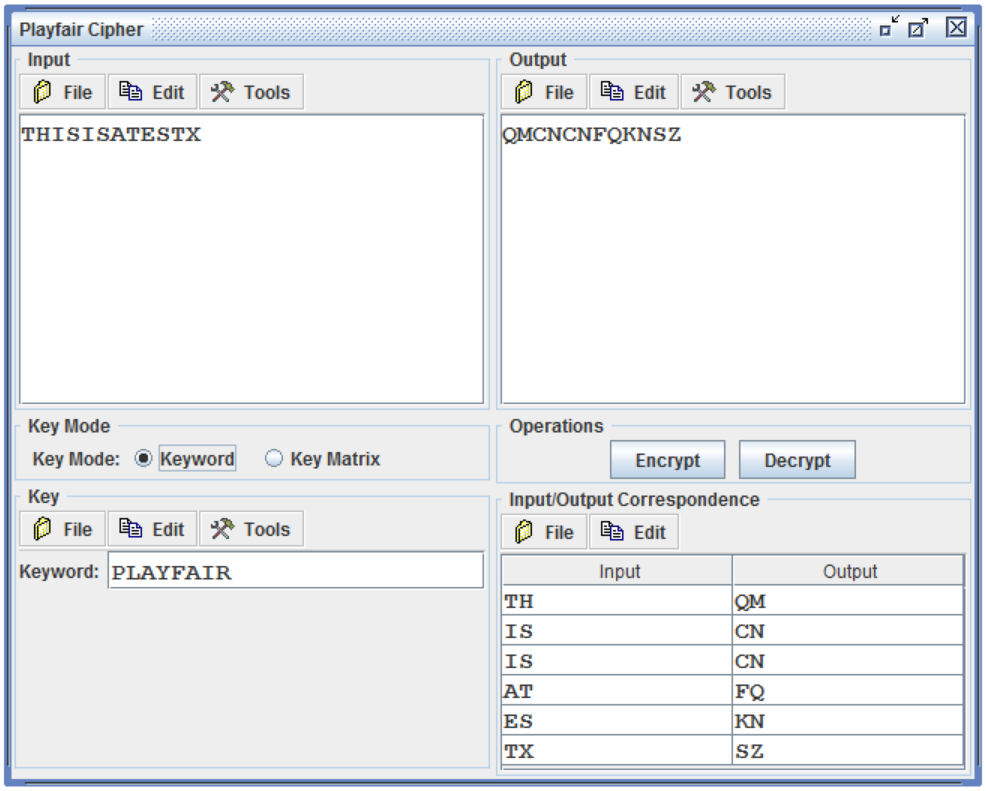
The Playfair cipher is a symmetric encryption technique and was the first digram substitution cipher. It was invented in 1854 by Charles Wheatstone, but was given the name of Lord Playfair (Lyon Playfair) who promoted the use of the cipher. The Playfair cipher was used as a field cipher by British forces in the Second Boer War and in World War I and by the British and Australians during World War II.

There are several slightly different ways to implement this cipher. The one we use equates I and J in order to get 26 letters down to 25, to fit in the 5 X 5 grid. So the plaintext cannot have any J's in it. There is a tool in the input toolbar that will automatically convert J to I. Also, as with all Playfair ciphers, when the plaintext is broken into blocks of 2, there cannot be duplicate characters. There is another tool in the input menu that will remove these automatically by placing an X between the double letters. For example, FOOD would be converted to FOXOD. While it is true that not all duplicate letters need to be replaced, this tool will replace all repeated characters, whether or not the duplication would show up in the same block of 2.

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box. Make sure that the characters are all uppercase, no 2-block duplicate letters, and all J's are converted to I's.



THISISATESTX

QMCNCNFQKNSZ

Key Mode: [✓ Keyword] Key Matrix

Keyword: PLAYFAIR

Input	Output
TH	QM
IS	CN
IS	CN
AT	FQ
ES	KN
TX	SZ

Figure 3.14: Playfair Cipher Tool: Keyword Mode

Note that the Input toolbar has options in the Tools menu for these standard conversions.

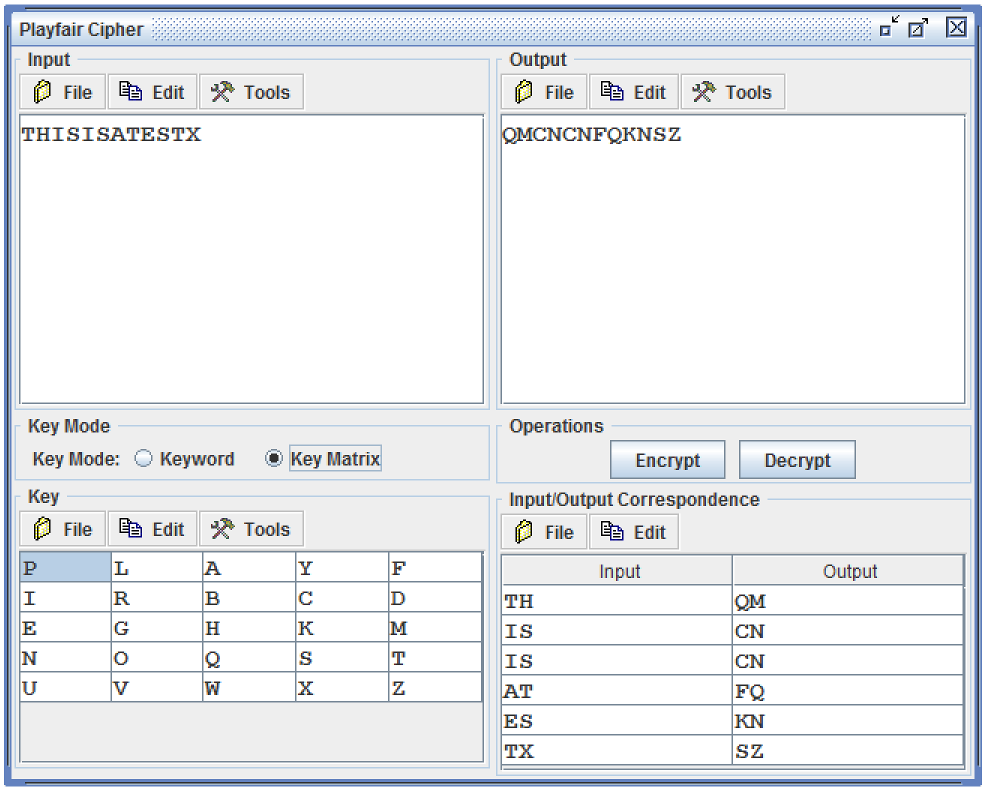
2. Select either the Keyword or Key Matrix mode or operation.
3. Input a Keyword or Matrix.
4. Click the Encrypt button. At this point the Output box will display the ciphertext message and the Input/Output Correspondence table will show the encryption character pair by character pair.

To Decrypt —

1. Input the ciphertext message into the Input box. Make sure that the characters are all uppercase, no 2-block duplicate letters, and all J's are converted to I's.
2. Select either the Keyword or Key Matrix mode or operation.
3. Input a Keyword or Matrix.
4. Click the Decrypt button. At this point the Output box will display the plaintext message and the Input/Output Correspondence table will show the decryption character pair by character pair.

Options

- The Key Mode can be set to either Keyword or Key Matrix mode. With the Keyword mode, as pictured above, the user inputs a single keyword, The program automatically converts the keyword to uppercase, ignores and repeated characters and changes any J's to I's. These conversions are done internally, so the user does not see any change in the keyword. The altered keyword is then used to fill out the Playfair 5 X 5 grid by starting on row one left to right and moving to subsequent rows when necessary. When the characters of the keyword have been used, the remaining cells in the grid are filled in with the missing letters going in alphabetical order. So in the example pictured above, the keyword PLAYFAIR would be converted to PLAYFIR and then P L A Y F would be in the first row and I R would be in the first two cells in the second row.
- The Key Matrix mode (pictured below) allows the user to create the matrix by hand. This mode comes in handy when one was trying to break a Playfair cipher. In this mode the user needs to enter the characters into each cell of the table. In the example below, the table is the same here as it would be for the keyword PLAYFAIR.



THISISATESTX

QMCNCNFQKNSZ

Key Mode: Keyword [✓ Key Matrix]

P	L	A	Y	F
I	R	B	C	D
E	G	H	K	M
N	O	Q	S	T
U	V	W	X	Z

Input	Output
TH	QM
IS	CN
IS	CN
AT	FQ
ES	KN
TX	SZ

Figure 3.15: Playfair Cipher Tool: Matrix Mode

- In Keyword mode the Key menu has the following options.

  File —

New: Clears the keyword.

Open: Opens a keyword file and loads it into the keyword.

Save As: Saves the current keyword to a keyword file.

Print: Prints the current keyword to the selected printer.

Print Preview: Prints the current keyword to the print preview display.

Edit —

Copy: Copies the keyword to the clipboard.

Paste: Pastes the keyword from the clipboard.

Tools —

Convert to Uppercase: Converts all alphabetic characters to uppercase.

Remove Whitespace: Removes all whitespace in the text, spaces, line brakes, tabs, ...

Remove Punctuation: Removes all punctuation in the text.

- In Key Matrix mode the Key menu has the following options.

File —

New: Clears the matrix cells.

Open: Opens a key matrix file and loads it into the table.

Save As: Saves the current matrix to a key matrix file.

Print: Prints the current matrix to the selected printer.

Print Preview: Prints the current matrix to the print preview display.

Edit —

Copy: Copies the entire key grid to the clipboard.

Copy as LaTeX (tabular): Copies the entire key grid to the clipboard using the syntax for the LaTeX tabular environment.

Copy as LaTeX (longtable): Copies the entire key grid to the clipboard using the syntax for the LaTeX longtable environment.

Tools —

Convert to Uppercase: Converts all matrix cells to uppercase.

Check Playfair Matrix: Checks the validity of the current table and displays a message of either valid or invalid.

- The Input/Output Correspondence grid has a toolbar with the following options.

File —

Save As: Saves the current Input/Output Correspondence grid to a text file.

Print: Prints the current Input/Output Correspondence grid to the selected printer.

Print Preview: Prints the current Input/Output Correspondence grid to the print preview display.

Edit —

Copy: Copies the entire Input/Output Correspondence grid to the clipboard.

Copy as LaTeX (tabular): Copies the entire Input/Output Correspondence grid to the clipboard using the syntax for the LaTeX tabular environment.

Copy as LaTeX (longtable): Copies the entire Input/Output Correspondence grid to the clipboard using the syntax for the LaTeX longtable environment.

- The Encrypt and Decrypt buttons work as follows.
  - The Encrypt and Decrypt buttons will, of course, apply the Playfair cipher to the input and place the result in the output.
    - Encryption will apply the forward method for the cipher, that is, right and down for same row and column of character pairs.
    - Decryption will apply the backward method for the cipher, that is, left and up for same row and column of character pairs.

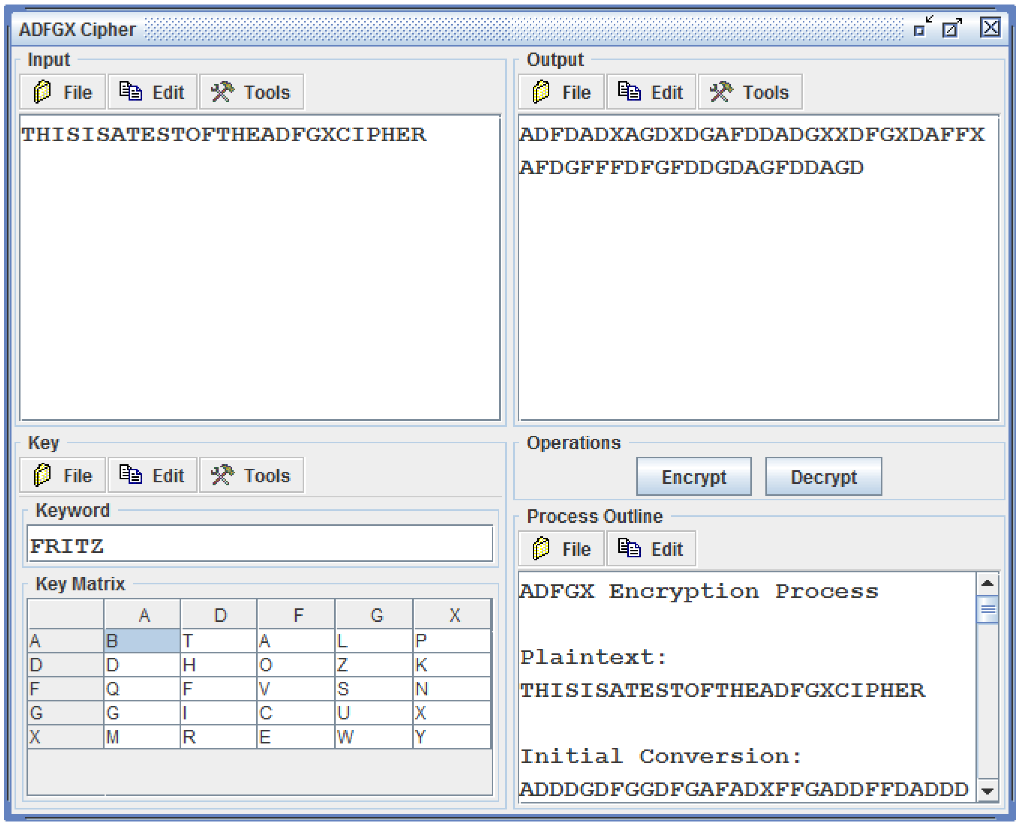
Notes

- There are several slightly different ways to implement this cipher. The one we use does the following.
  - We equate I and J in order to get 26 letters down to 25, so the plaintext cannot have any J's in it.
  - As with all Playfair ciphers, when the plaintext is broken into blocks of 2, there cannot be duplicate characters.
- With the Keyword mode, the user inputs a single keyword, The program automatically converts the keyword to uppercase, ignores and repeated characters and changes any J's to I's. These conversions are done internally, so the user does not see any change in the keyword. The altered keyword is then used to fill out the Playfair 5 X 5 grid by starting on row one left to right and moving to subsequent rows when necessary. When the characters of the keyword have been used, the remaining cells in the grid are filled in with the missing letters going in alphabetical order. For example, the keyword PLAYFAIR would be converted to PLAYFIR and then P L A Y F would be in the first row and I R would be in the first two cells in the second row.

3.3.9 ADFGX

The ADFGX Cipher was invented by Colonel Fritz Nebel in March of 1918, in June of that year the letter V was introduced, making it the ADFGVX Cipher. The ADFGVX cipher was a field cipher that was used by the German Army on the Western Front throughout World War I.

The ADFGX and ADFGVX ciphers got their name from the only five or six letters that show up in the ciphertext: A, D, F, G and X or A, D, F, G, V and X. The reason these particular letters were chosen was because they sound very different from each other when transmitted by Morse code. This was done to reduce the operator error in the transmission of the message, and was probably the first time that coding theory ideas were used in cryptography.



```
THISISATESTOFTHEADFGXCIPHER
```

```
ADFDADXAGDXDGAFDDADGXXDFGXDAFFX
AFDGFFFDFGFDDGDAGFDDAGD
```

Keyword: FRITZ

Key Matrix

	A	D	F	G	X
A	B	T	A	L	P
D	D	H	O	Z	K
F	Q	F	V	S	N
G	G	I	C	U	X
X	M	R	E	W	Y

```
ADFGX Encryption Process

Plaintext:
THISISATESTOFTHEADFGXCIPHER

Initial Conversion:
ADDDGDFGGDFGAFADXFFGADDFFDADDD
```

Figure 3.16: ADFGX Cipher Tool

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box. Make sure that the characters are all uppercase and all J's are converted to I's. Note that the Input toolbar has options in the Tools menu for these standard conversions.
2. Input a Keyword. The keyword must consist of all uppercase letters with no duplications.

3. Input a Key Matrix. The key matrix must consist of all uppercase letters excluding J, as with the playfair cipher we use the version of the ADFGX which equates I and J.
4. Click the Encrypt button. At this point the Output box will display the ciphertext message and the Process Outline box will show the encryption process step by step.

To Decrypt —

1. Input the ciphertext message into the Input box.
2. Input a Keyword. The keyword must consist of all uppercase letters with no duplications.
3. Input a Key Matrix. The key matrix must consist of all uppercase letters excluding J, as with the playfair cipher we use the version of the ADFGX which equates I and J.
4. Click the Decrypt button. At this point the Output box will display the plaintext message and the Process Outline box will show the decryption process step by step.

Options

- The Key menu has the following options.

  File —

  New: Clears the keyword and matrix.

  Open: Opens a key file and loads it into the keyword and matrix.

  Save As: Saves the current key to a file.

  Print: Prints the current key to the selected printer.

  Print Preview: Prints the current key to the print preview display.

  Edit —

  Copy Keyword: Copies the keyword to the clipboard.

  Copy Key Matrix: Copies the key matrix to the clipboard.

  Copy Key Matrix As LaTeX: Copies the key matrix to the clipboard using LaTeX syntax.

  Paste Keyword: Pastes the keyword from the clipboard.

  Paste Key Matrix: Pastes the key matrix from the clipboard. Specifically, the contents of the clipboard are treated as a tab-delimited grid and the paste begins in the upper left corner of the table.

  Tools —

  Convert to Uppercase: Converts all alphabetic characters to uppercase.

Remove Whitespace: Removes all whitespace in the text, spaces, line brakes, tabs, ...

Remove Punctuation: Removes all punctuation in the text.

Check Key: Checks if the keyword and matrix are valid.

- The Process Outline has a toolbar with the following options.

  File —

  Save As: Saves the current Process Outline to a text file.

  Print: Prints the current Process Outline to the selected printer.

  Print Preview: Prints the current Process Outline to the print preview display.

  Edit —

  Copy: Copies the selected portion of the entire Process Outline to the clipboard.

  Copy All: Copies the entire Process Outline to the clipboard.

- The Encrypt and Decrypt buttons work as follows.
  - The Encrypt and Decrypt buttons will, of course, apply the ADFGX cipher to the input and place the result in the output.
    - Encryption will apply the forward method for the cipher.
    - Decryption will apply the backward method for the cipher.

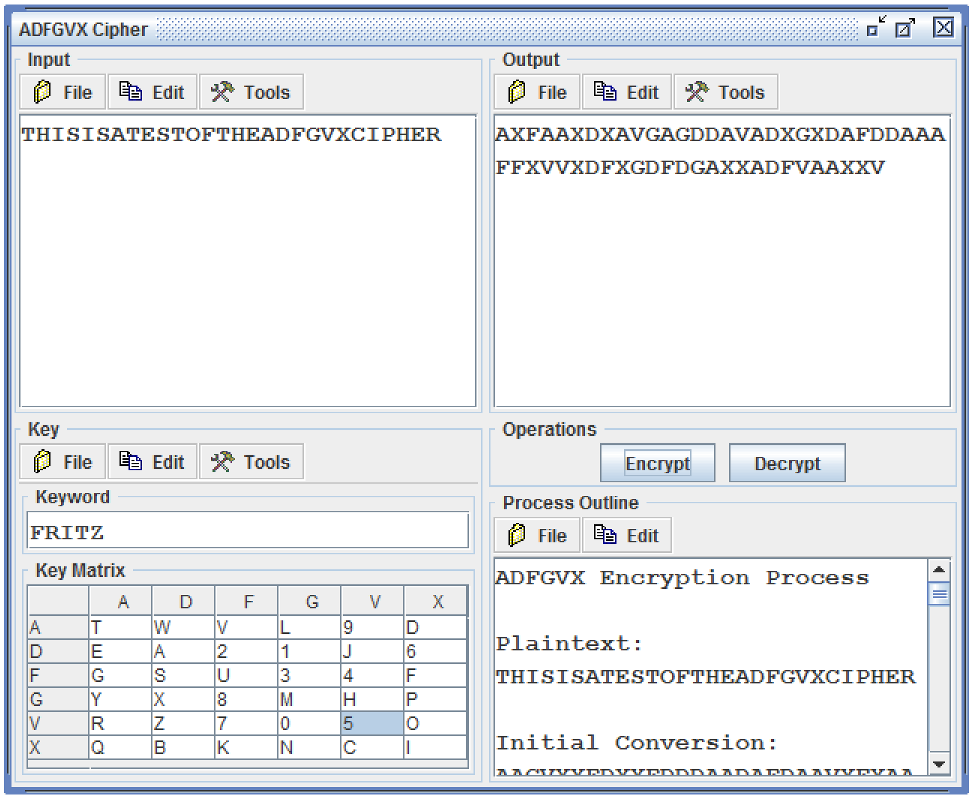
Notes

- We equate I and J in order to get 26 letters down to 25, so the plaintext cannot have any J's in it.

3.3.10 ADFGVX

The ADFGX Cipher was invented by Colonel Fritz Nebel in March of 1918, in June of that year the letter V was introduced, making it the ADFGVX Cipher. The ADFGVX cipher was a field cipher that was used by the German Army on the Western Front throughout World War I.

The ADFGX and ADFGVX ciphers got their name from the only five or six letters that show up in the ciphertext: A, D, F, G and X or A, D, F, G, V and X. The reason these particular letters were chosen was because they sound very different from each other when transmitted by Morse code. This was done to reduce the operator error in the transmission of the message, and was probably the first time that coding theory ideas were used in cryptography.



```
THISISATESTOFTHEADFGVXCIPHER
```

```
AXFAAXDXAVGAGDDAVADXGXDAFDDAAA
FFXVVXDFXGDFDGAXXADFVAAXXV
```

Keyword: FRITZ

Key Matrix

	A	D	F	G	V	X
A	T	W	V	L	9	D
D	E	A	2	1	J	6
F	G	S	U	3	4	F
G	Y	X	8	M	H	P
V	R	Z	7	0	5	O
X	Q	B	K	N	C	I

```
ADFGVX Encryption Process

Plaintext:
THISISATESTOFTHEADFGVXCIPHER

Initial Conversion:
```

Figure 3.17: ADFGVX Cipher Tool

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box. Make sure that the characters are all uppercase. Note that the Input toolbar has an option in the Tools menu for this conversion.
2. Input a Keyword. The keyword must consist of all uppercase letters with no duplications.
3. Input a Key Matrix. The key matrix must consist of all uppercase letters and single digits.
4. Click the Encrypt button. At this point the Output box will display the ciphertext message and the Process Outline box will show the encryption process step by step.

To Decrypt —

1. Input the ciphertext message into the Input box.
2. Input a Keyword. The keyword must consist of all uppercase letters with no duplications.

3. Input a Key Matrix. The key matrix must consist of all uppercase letters and single digits.
4. Click the Decrypt button. At this point the Output box will display the plaintext message and the Process Outline box will show the decryption process step by step.

Options

- The Key menu has the following options.

  File —

  New: Clears the keyword and matrix.

  Open: Opens a key file and loads it into the keyword and matrix.

  Save As: Saves the current key to a file.

  Print: Prints the current key to the selected printer.

  Print Preview: Prints the current key to the print preview display.

  Edit —

  Copy Keyword: Copies the keyword to the clipboard.

  Copy Key Matrix: Copies the key matrix to the clipboard.

  Copy Key Matrix As LaTeX: Copies the key matrix to the clipboard using LaTeX syntax.

  Paste Keyword: Pastes the keyword from the clipboard.

  Paste Key Matrix: Pastes the key matrix from the clipboard. Specifically, the contents of the clipboard are treated as a tab-delimited grid and the paste begins in the upper left corner of the table.

  Tools —

  Convert to Uppercase: Converts all alphabetic characters to uppercase.

  Remove Whitespace: Removes all whitespace in the text, spaces, line brakes, tabs, ...

  Remove Punctuation: Removes all punctuation in the text.

  Check Key: Checks if the keyword and matrix are valid.

- The Process Outline has a toolbar with the following options.

  File —

  Save As: Saves the current Process Outline to a text file.

  Print: Prints the current Process Outline to the selected printer.

  Print Preview: Prints the current Process Outline to the print preview display.

  Edit —

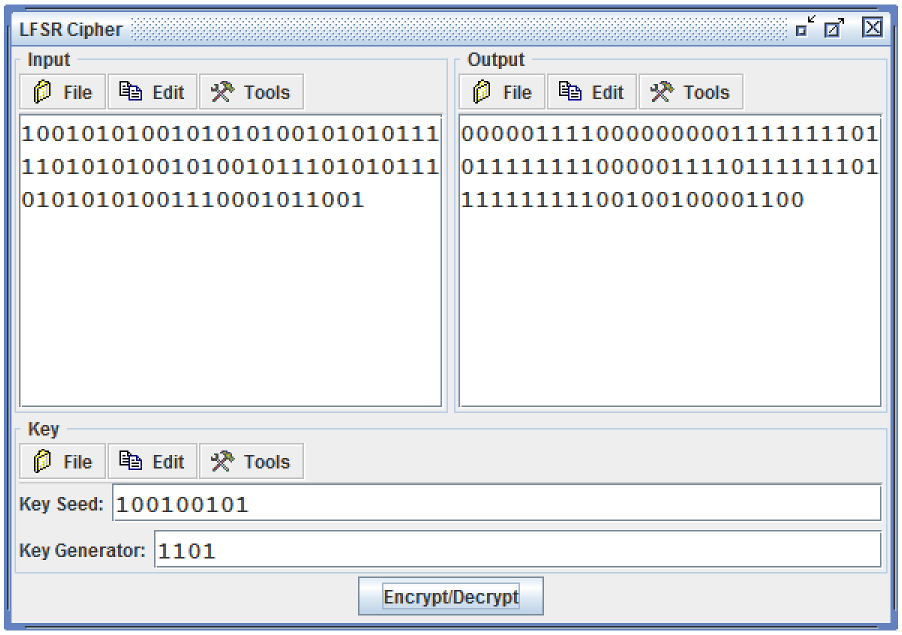
  Copy: Copies the selected portion of the entire Process Outline to the clipboard.

Copy All: Copies the entire Process Outline to the clipboard.

- The Encrypt and Decrypt buttons work as follows.
  - The Encrypt and Decrypt buttons will, of course, apply the ADFGX cipher to the input and place the result in the output.
    - Encryption will apply the forward method for the cipher.
    - Decryption will apply the backward method for the cipher.

3.3.11 Linear Feedback Shift Register (LFSR)

The LFSR Cipher is a binary stream cipher.



```
1001010100101010100101010111
1101010100101001011101010111
01010101001110001011001
```

```
0000011110000000001111111101
0111111110000011110111111101
11111111100100100001100
```

Key Seed: 100100101

Key Generator: 1101

Figure 3.18: Linear Feedback Shift Register Cipher Tool

How to Use the Tool

To Encrypt or Decrypt —

1. Input the binary plaintext message into the Input box. The input must be a binary (0 and 1) string. Note that the Input toolbar has options in the Tools menu for conversions from text to numbers, including binary representations of ASCII character values.
2. Input a Key Seed. The key seed must be a binary string.
3. Input a Key Generator. The key generator must be a binary string.
4. Click the Encrypt/Decrypt button.

Options

- The Key menu has the following options.

  File —

  New: Clears the key inputs.

  Open: Opens a key file and loads it into the key inputs.

  Save As: Saves the current key to a file.

  Print: Prints the current key to the selected printer.

  Print Preview: Prints the current key to the print preview display.

  Edit —

  Copy: Copies the key to the clipboard.

  Tools —

  Remove Whitespace: Removes all whitespace in the text, spaces, line brakes, tabs, ...

  Remove Punctuation: Removes all punctuation in the text.

- The Encrypt/Decrypt button applies the LFSR algorithm to the input and key and places the result in the output.

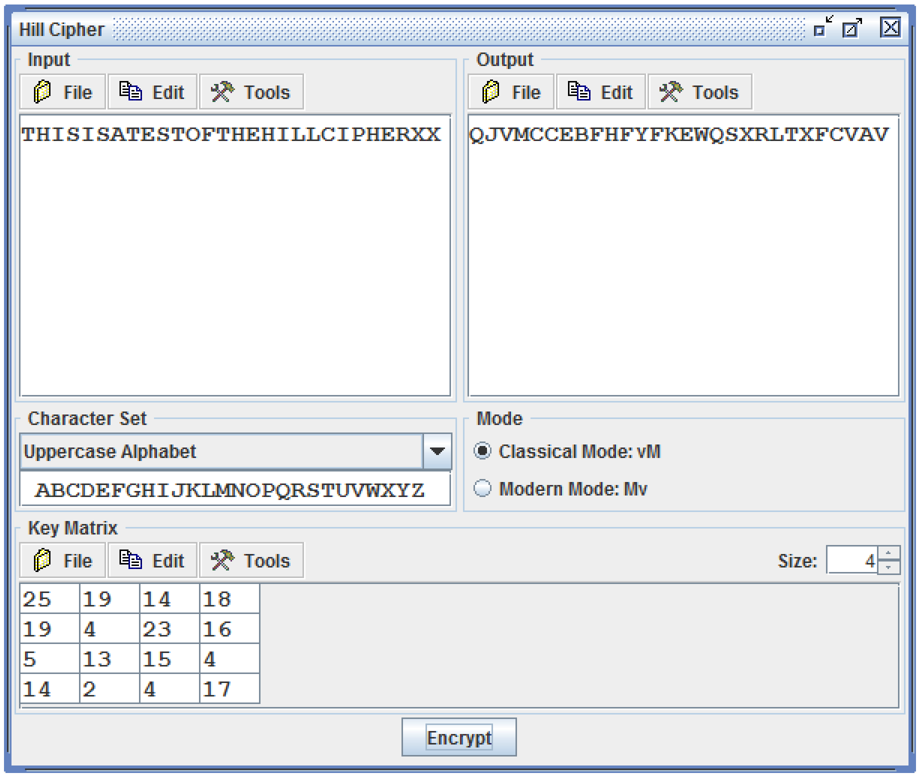
Notes

- The LFSR cipher simply takes the binary plaintext message and XOR's each bit of the plaintext with the corresponding bit of the key. So if the message is 100101 and the key is 110111 then the ciphertext is 010010.

- The key is produced by a seed and a generator string. The seed is taken verbatim as the beginning of the key. After the seed is exhausted, the generator string will generate more bits to the key until the length of the key matches the length of the length of the plaintext message. The method of generation is as follows. The key generator (which is given in binary form) is placed on the right of the key so that the last bit of the key is in the same position as the last bit of the generator. If the generator has a 1 in a position then the value of key in that position is taken. All taken bits are then added modulo 2 and the result is the next bit in the key. This process is then continued for the next bit of the key and so on. For example, if the seed if 110111 and the generator is 101 then the key will be 11011101001...

3.3.12 Hill

The Hill Cipher was developed by Lester Hill in 1929. Lester Hill was a professor at Hunter College in New York City and first published this method in the American Mathematical Monthly with his article *Cryptography in an Algebraic Alphabet*. Although it seems that this method was not used much in practice, it marked the transition cryptography made from a mainly linguistic practice to a mathematical discipline. Prior to World War II most cryptographic and cryptanalysis methods centered around replacing characters in a message with different characters (using one or more alphabets) and mixing up or rearranging the message. Hence the code breakers were primarily people who were highly trained in linguistics, could speak several languages, and were good puzzle solvers. With the invention of the Enigma machine, used by the German's in World War II, cryptanalysis of these ciphertexts required advanced mathematics and an enormous amount of computation, far beyond that of a single person or group of people.



THISISATESTOFTHEHILLCIPHERXX

QJVMCCEBFHFYFKEWQSXRLTXFCVAV

Character Set: Uppercase Alphabet

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Mode: [✓ Classical Mode: vM] Modern Mode: Mv

Size: 4

25	19	14	18
19	4	23	16
5	13	15	4
14	2	4	17

Figure 3.19: Hill Cipher Tool

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box. Make sure that the characters are from the same character set as the one selected. Note that you can change the character set using the selection box below the Input box. There are also some quick conversion tools in the Tools menu. Note that since the Hill cipher is a block cipher with block size the size of the key matrix, the number of characters in the input must be a multiple of the number of rows (and hence columns) of

the matrix. In some cases you may need to pad the input with extra characters to apply the cipher.

2. Input a Key Matrix. The key matrix must consist of numbers greater than or equal to 0 and less than the size of your character set, since the Hill cipher will do all calculations modulo the size of the character set.
3. Click the Encrypt button. At this point the Output box will display the ciphertext message.

To Decrypt —

1. Input the ciphertext message into the Input box. Make sure that the characters are from the same character set as the one selected. Note that you can change the character set using the selection box below the Input box. There are also some quick conversion tools in the Tools menu. Note that since the Hill cipher is a block cipher with block size the size of the key matrix, the number of characters in the input must be a multiple of the number of rows (and hence columns) of the matrix. In some cases you may need to pad the input with extra characters to apply the cipher.
2. Input a Key Matrix. The key matrix must be the inverse, modulo the size of the character set, of the encryption matrix. The tool will not automatically invert the matrix that is displayed, you need to use the Modular Matrix Calculator to find the inverse.
3. Click the Encrypt button. At this point the Output box will display the plaintext message.

Options

- In the lower left quarter of the window is the key matrix that will be used for encoding and a selection for the character set to use for the cipher. The character sets are as follows:

  Uppercase Alphabet: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

  User Defined Language: This will open an Open dialog box that will allow you to select a user defined language. Information on creating a user defined language can be found in the section on the *User Defined Language Creator*.

- The Mode is the way that the message vectors are multiplied by the encryption matrix. Classical mode uses the same process as Lester Hill used back in 1929. It translates the plaintext message to row vectors $\mathbf{v}$, and then multiplies the vector on the right by the encryption matrix M, that is, it computes $\mathbf{v}M = \mathbf{w}$. The vector $\mathbf{w}$ is then translated back to the character set as the ciphertext. Modern mode follows the current linear algebra practice of treating a linear transformation more as a function, basically the modern mode simply reverses the multiplication. In modern mode the plaintext message is converted into column vectors $\mathbf{v}$ and then multiplied on the left by the encryption matrix M, that is, $M\mathbf{v} = \mathbf{w}$. Since matrix multiplication is not commutative these methods will produce different ciphertexts on the same message with the same key matrix.

- The Key Matrix menu has the following options. Also, to change the size of the matrix, there is a size selection to the right of the menu.

  File —

  New: Clears the entries of the key matrix.

  Open: Opens a key file and loads it into the matrix.

  Save As: Saves the current key to a file.

  Print: Prints the current key to the selected printer.

  Print Preview: Prints the current key to the print preview display.

  Edit —

  Copy: Copies the key matrix to the clipboard.

  Copy as LaTeX (tabular): Copies the key matrix to the clipboard using LaTeX syntax and the tabular environment.

  Copy as LaTeX (array): Copies the key matrix to the clipboard using LaTeX syntax and the array environment.

  Copy to Mathematica Syntax: Copies the key matrix to the clipboard using Mathematica syntax.

  Copy to Maxima Syntax: Copies the key matrix to the clipboard using Maxima syntax.

  Tools —

  Create Random Matrix: Creates a random matrix using entries in the range of 0 to one less then the size of the character set.

  Check for Inverse Matrix: Checks if the matrix has an inverse modulo the size of the character set.

- The Encrypt button will apply the Hill cipher to the input. In the encryption process, the program will

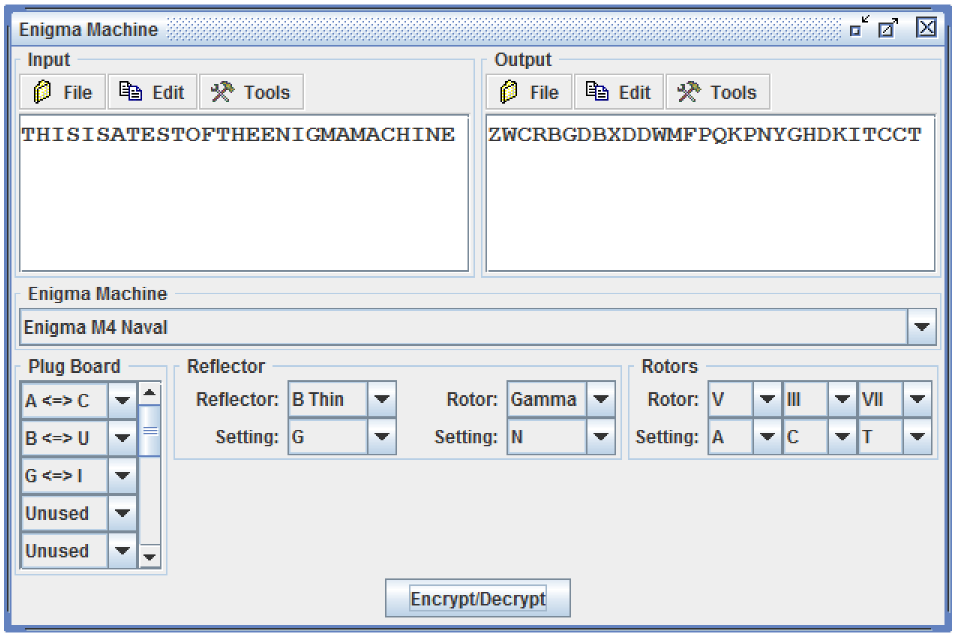
  1. Block the input into blocks of size n (where the key matrix is n X n).
  2. Translate each of these blocks into a row vector v. The correspondence of letters to numbers is done by the position of the letter in the character set. So if the character set is the uppercase alphabet (A B C D E F G H I J K L M N O P Q R S T U V W X Y Z) then A = 0, B = 1, C = 2, and so on. If, in the other hand, the character set is rtdfi then r = 0, t = 1, d = 2, and so on and the calculations are done modulo 5 as opposed to modulo 26 in the case of the uppercase alphabet.
  3. Apply the matrix on the right, that is w = vM.
  4. Translate w back to letters for the output.

Notes

- With the Hill cipher, decryption is the same as encryption except that you use the inverse of the encryption matrix (modulo n).
- The encryption matrix need not be invertible to apply the encryption.
- To decrypt a message with the Hill cipher key matrix that is needed is the inverse (modulo the size of the character set) of the encryption matrix. This tool will not find the inverse of the encryption matrix automatically, you will need to use the modular matrix calculator to calculate the inverse.

3.3.13 Enigma

The Enigma Machine was Germany's encryption device throughout WWII. This simulator is designed to simulate four versions of the Enigma Machine, the Enigma I, the M3 Army, the M3 Naval and the M4 Naval.



Input: THISISATESTOFTHEENIGMAMACHINE

Output: ZWCRBGDBXDDWMFPQKPNYGHDKITCCT

Enigma Machine: Enigma M4 Naval

Plug Board: A <=> C, B <=> U, G <=> I, Unused, Unused

Reflector: Reflector: B Thin, Setting: G; Rotor: Gamma, Setting: N

Rotors: Rotor: V, III, VII; Setting: A, C, T

Figure 3.20: Enigma Machine Simulation Tool

How to Use the Tool

To Encrypt and Decrypt —

1. Input the message into the Input box. Make sure that the characters are all uppercase letters. There are some quick conversion tools in the Tools menu.

2. Select the type of Enigma machine you want to use, the Enigma I, the M3 Army, the M3 Naval or the M4 Naval. When the selection of the machine is made the Rotor and Reflector options will change to match that type of machine.
3. Set the plug board options. Each cable in the plug board is represented by a drop-down list of the form `A <=> B`, `A <=> C`, and so on. So a setting of `D <=> M` would represent a patch between the letters D and M. Note that there are more cables available in this simulator then there were in the original machines. Also, none of the cables can have duplicate listings, so selecting `A <=> B` for one cable and `A <=> C` for another will produce an error, as will having cables `A <=> B` and `B <=> C`.
4. Set the rotors and reflector settings. The rotor and reflector settings are also done by drop-down lists. The top selection is the rotor or reflector to be used and the bottom selection is the character setting of the rotor. On some of the Enigma models the rotors were actually labeled with numbers 1-26 instead of letters, but the usual letter number correspondence applies. The program will not allow a duplication of rotors in the machine, hence the rotors must all be different.
5. Click the Encrypt/Decrypt button. At this point the Output box will display the ciphertext message.

Notes

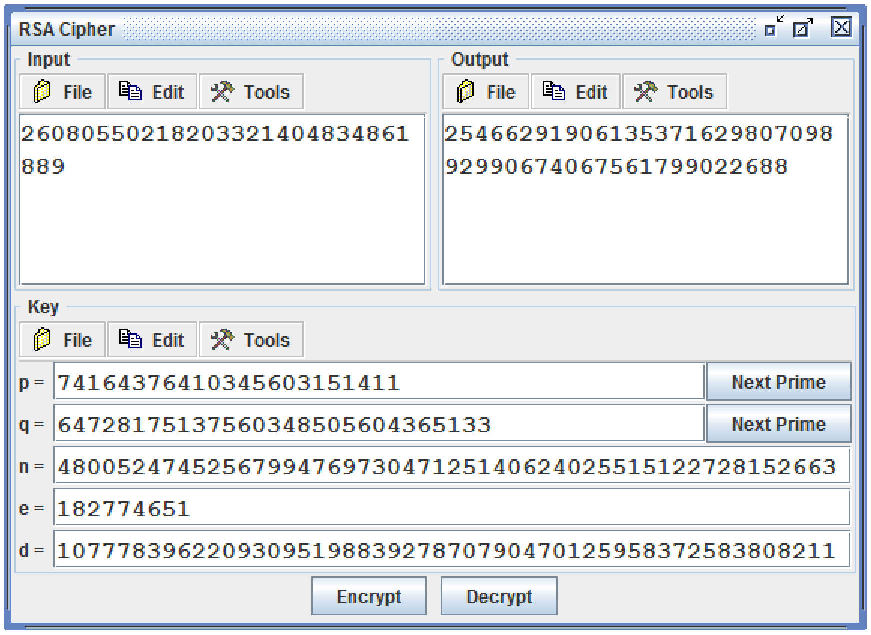
- The rotor and reflector wirings are as follows,
  - Rotor I: Substitution: EKMFLGDQVZNTOWYHXUSPAIBRCJ with Notch at Q.
  - Rotor II: Substitution: AJDKSIRUXBLHWTMCQGZNPYFVOE with Notch at E.
  - Rotor III: Substitution: BDFHJLCPRTXVZNYEIWGAKMUSQO with Notch at V.
  - Rotor IV: Substitution: ESOVPZJAYQUIRHXLNFTGKDCMWB with Notch at J.
  - Rotor V: Substitution: VZBRGITYUPSDNHLXAWMJQOFECK with Notch at Z.
  - Rotor VI: Substitution: JPGVOUMFYQBENHZRDKASXLICTW with Notches at M and Z.
  - Rotor VII: Substitution: NZJHGRCXMYSWBOUFAIVLPEKQDT with Notches at M and Z.
  - Rotor VIII: Substitution: FKQHTLXOCBJSPDZRAMEWNIUYGV with Notches at M and Z.
  - Rotor Beta: Substitution: LEYJVCNIXWPBQMDRTAKZGFUHOS.

- Rotor Gamma: Substitution: FSOKANUERHMBTIYCWLQPZXVGJD.
- Reflector A: Substitution: EJMZALYXVBWFCRQUONTSPIKHGD.
- Reflector B: Substitution: YRUHQSLDPXNGOKMIEBFZCWVJAT.
- Reflector B Thin: Substitution: ENKQAUYWJICOPBLMDXZVFTHRGS.
- Reflector C Thin: Substitution: RDOBJNTKVEHMLFCWZAXGYIPSUQ.

3.3.14 RSA

The RSA algorithm was developed by three professors at MIT in 1977, Ron Rivest, Adi Shamir, and Leonard Adlemen, their initials give the algorithm its name. This was one of the first algorithms to implement the concept of public-key cryptography. The actual concept of public-key cryptography was discovered by Whitfield Diffe and Martin Hellman just one year earlier.

As a historical note, Rivest, Shamir and Adlemen were not the first mathematicians to discover this technique. In 1973, Clifford Cocks, a British mathematician and cryptographer at the Government Communications Headquarters (GCHQ), had developed an equivalent system.



```
26080550218203321404834861
889
```

```
25466291906135371629807098
92990674067561799022688
```

p = 74164376410345603151411

q = 64728175137560348505604365133

n = 4800524745256799476973047125140624025515122728152663

e = 182774651

d = 1077783962209309519883927870790470125958372583808211

Figure 3.21: RSA Cipher Tool

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box. This must be in the form of a single number or a set of numbers separated by spaces which are less than the modulus $n = pq$. If a number in the list is larger than n, it will first be reduced modulo n, hence most likely losing the information it held. There are numerous options in the Tools menu for converting text to numbers.
2. Input the public key of n and e. Alternatively, if you are creating the public and private keys for the system you can input the primes, p and q and the encryption exponent e, then select the option from the Tools menu for the key to generate d and n. The input boxes for p and q have options for selecting the next probable prime greater than the one input.
3. Click the Encrypt button. At this point the Output box will display the ciphertext message, in this case the number $c = m^e \pmod{n}$. If several numbers were input then there will be one number output for each of the inputs.

To Decrypt —

1. Input the ciphertext message into the Input box. This must be in the form of a single number or a set of numbers separated by spaces which are less than the modulus $n = pq$. If a number in the list is larger than n, it will first be reduced modulo n, hence most likely losing the information it held.
2. Input the private key of n and d. Alternatively, if you know p and q you can input these and the encryption exponent e, then select the option from the Tools menu for the key to generate d and n.
3. Click the Decrypt button. At this point the Output box will display the plaintext message, in this case the number $m = c^d \pmod{n}$. If several numbers were input then there will be one number output for each of the inputs.

Options

- The Key menu has the following options.

  File —

  New: Clears the entries of the key.

  Open: Opens a key file and loads it into the entries.

  Save As: Saves the current key to a file.

  Print: Prints the current key to the selected printer.

  Print Preview: Prints the current key to the print preview display.

  Edit —

  Copy: Copies the key to the clipboard.

  Tools —

Generate n and d from p, q and e: Given p, q and e the program will generate n and d. If p, q, and e do not have the necessary properties the program will produce the appropriate error message.

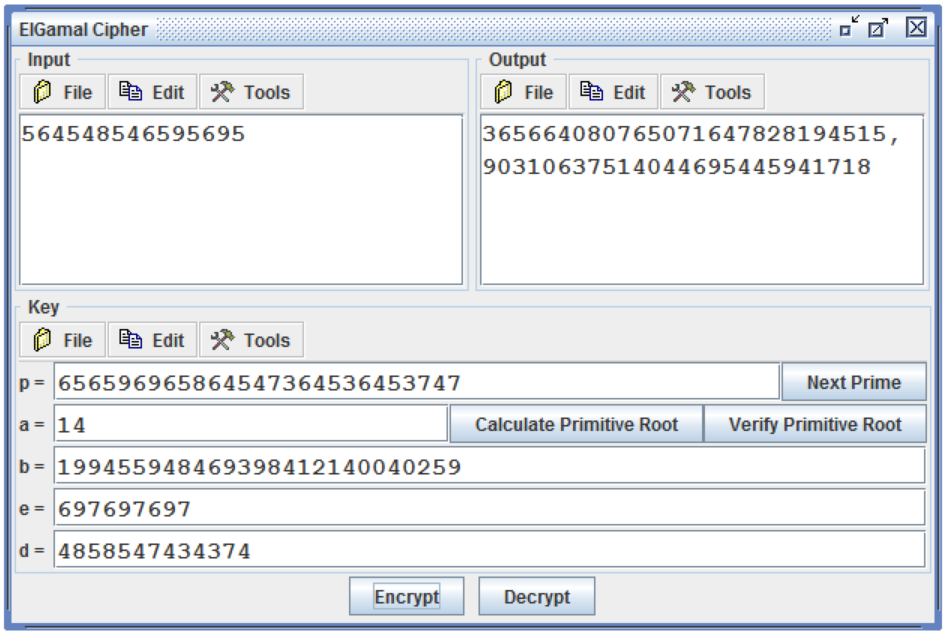
- The Encrypt and Decrypt buttons will apply the RSA algorithm to the input using the current key. Encrypt will use e as the exponent and Decrypt will use d as the exponent.

3.3.15 ElGamal

The ElGamal public key algorithm was developed by Taher ElGamal in 1985 and is a system whose security relies on the difficulty of computing discrete logarithms. The encryption and decryption process is as follows. The public portion of the key is a triple (p, a, b) where p is a prime, a is a primitive root of p and $b = a^d \pmod{p}$, where d is a private decryption exponent.

To encrypt a message m, where m is a number between 0 and $p-1$, a private exponent e is selected and the two values r and t are calculated, $r = a^e \pmod{p}$ and $t = b^e \cdot m \pmod{p}$. The pair (r, t) is the ciphertext.

To decrypt a message, one calculates $tr^{-e} = m \pmod{p}$.



Input: 564548546595695

Output: 365664080765071647828194515,
90310637514044695445941718

p = 656596965864547364536453747

a = 14

b = 199455948469398412140040259

e = 697697697

d = 4858547434374

Figure 3.22: ElGamal Cipher Tool

How to Use the Tool

To Encrypt —

1. Input the plaintext message into the Input box. This must be in the form of a single number which is less than the modulus p. If a number in the list is larger than p, it will first be reduced modulo p, hence most likely losing the information it held. There are numerous options in the Tools menu for converting text to numbers.
2. Input the public key of p, a and b. Also input an encryption exponent e.

   If you are creating the public and private keys for the system you can input the prime p, the decryption exponent d, and a primitive root a of p, then select the option from the Tools menu for the key to generate b. The input box for p has an option for selecting the next probable prime greater then the one input. The input box for a has the options to generate a primitive root and to check if the input number is a primitive root.
3. Click the Encrypt button. At this point the Output box will display the ciphertext message, which is the pair (r, t). The pair is output without parentheses but with a comma separating r and t.

To Decrypt —

1. Input the ciphertext message into the Input box, this must be of the same form as the encryption output, that is, two numbers separated by a comma. If either number in the list is larger than p, it will first be reduced modulo p, hence most likely losing the information it held.
2. Input the private decryption exponent d.
3. Click the Decrypt button. At this point the Output box will display the plaintext message.

Options

- The Key menu has the following options.

  File —

  New: Clears the entries of the key.

  Open: Opens a key file and loads it into the entries.

  Save As: Saves the current key to a file.

  Print: Prints the current key to the selected printer.

  Print Preview: Prints the current key to the print preview display.

  Edit —

  Copy: Copies the key to the clipboard.

  Tools —

  Generate b from a, d and p: Given a, d and p the program will generate b.

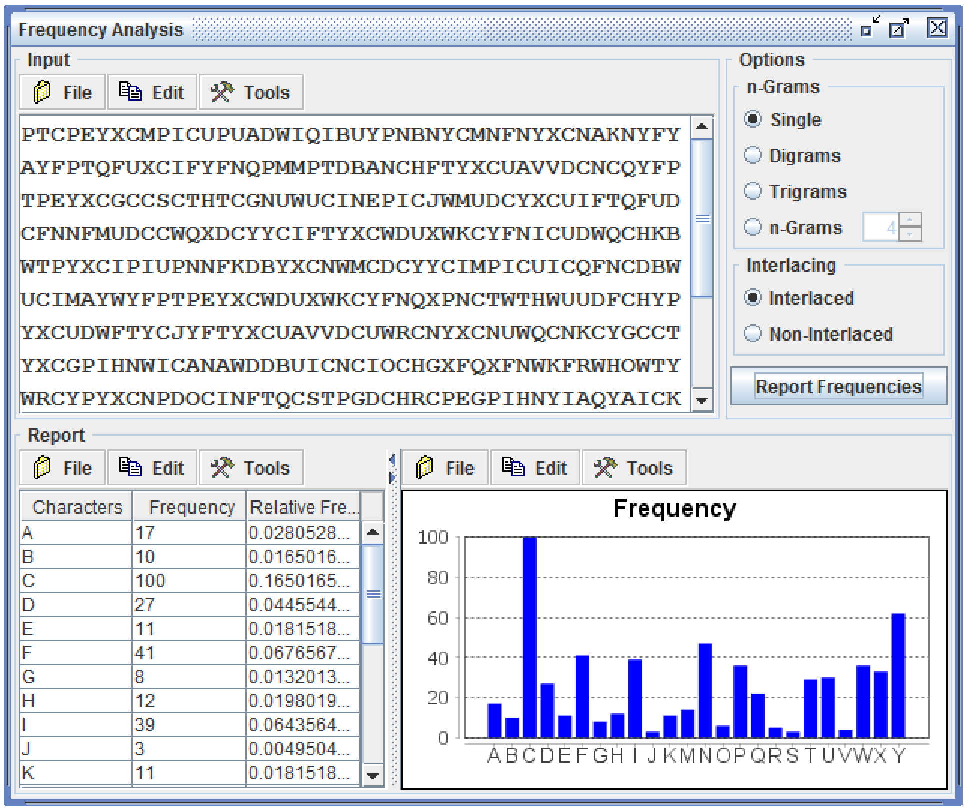
Notes

- Both of the options to calculate a primitive root and to verify a primitive root rely on factoring $p - 1$. This can be a lengthy process for some large primes p.

3.4 Text and Stream Analysis

3.4.1 Frequency Analysis

The Frequency Analysis window will analyze text for character, digram, trigram, or n-gram frequencies. There is also an option to interlace or not interlace the blocks of characters.



```
PTCPEYXCMPICUPUADWIQIBUYPNBNYCMNFNYXCNAKNYFY
AYFPTQFUXCIFYFNQPMMPTDBANCHFTYXCUAVVDCNCQYFP
TPEYXCGCCSCTHTCGNUWUCINEPICJWMUDCYXCUIFTQFUD
CFNNFMUDCCWQXDCYYCIFTYXCWDUXWKCYFNICUDWQCHKB
WTPYXCIPIUPNNFKDBYXCNWMCDCYYCIMPICUICQFNCDBW
UCIMAYWYFPTPEYXCWDUXWKCYFNQXPNCTWTHWUUDFCHYP
YXCUDWFTYCJYFTYXCUAVVDCUWRCNYXCNUWQCNKCYGCCT
YXCGPIHNWICANAWDDBUICNCIOCHGXFQXFNWKFRWHOWTY
WRCYPYXCNPDOCINFTQCSTPGDCHRCPEGPIHNYIAQYAICK
```

n-Grams: [✓ Single] Digrams Trigrams n-Grams 4

Interlacing: [✓ Interlaced] Non-Interlaced

Characters	Frequency	Relative Fre...
A	17	0.0280528...
B	10	0.0165016...
C	100	0.1650165...
D	27	0.0445544...
E	11	0.0181518...
F	41	0.0676567...
G	8	0.0132013...
H	12	0.0198019...
I	39	0.0643564...
J	3	0.0049504...
K	11	0.0181518...

Figure 3.23: Frequency Analysis Tool

How to Use the Tool

1. Input the ciphertext into the Input box.
2. Select the analysis option, either single characters, digrams, trigrans, or n-grams.
3. Select the interlacing option of either interlaced or not interlaced.

4. Click on the Report Frequencies button. At this point the frequencies and relative frequencies will be displayed in the report table and the same data will be displayed in the report bar chart.

Options

- The frequency reporting options are as follows. The reporting is case sensitive, so A and a are seen as two distinct characters.

  Single: Single will report the frequencies and relative frequencies of each character in the input.

  Digrams: This will report all consecutive character pairs in the input.

  Trigrams: This will report all consecutive character triples in the input.

  n-Grams: This will report all consecutive blocks of n characters in the input. The user can select between 4 and 10 characters per block.

- The interlacing options are as follows,

  Interlaced: This will report all n-gram frequencies taking all consecutive blocks of n characters. For example, with digrams the input of QWERTY would count QW, WE, ER, RT, and TY.

  Non-Interlaced: This will report all n-gram frequencies taking consecutive blocks of n characters with no overlap. For example, with digrams the input of QWERTY would count QW, ER, and TY.

- The Report Table has a toolbar with the following options.

  File —

  Save As: Saves the current report table to a text file.

  Print: Prints the current report table to the selected printer.

  Print Preview: Prints the current report table to the print preview display.

  Edit —

  Copy: Copies the entire report table to the clipboard.

  Copy as LaTeX (tabular): Copies the entire report table to the clipboard using the syntax for the LaTeX tabular environment.

  Copy as LaTeX (longtable): Copies the entire report table to the clipboard using the syntax for the LaTeX longtable environment.

  Tools —

  Sort by Character: Sorts the table (and bar chart) using the character column of the table and alphabetical order.

Sort Ascending by Frequency: Sorts the table (and bar chart), from lowest to highest, using the frequency column of the table.

Sort Descending by Frequency: Sorts the table (and bar chart), from highest to lowest, using the frequency column of the table.

- The Report Bar Chart has a toolbar with the following options.

File —

Save As: Saves the current chart to a png file.

Print: Prints the current chart to the selected printer.

Print Preview: Prints the current chart to the print preview display.

Edit —

Copy: Copies the chart to the clipboard.

Tools —

Toggle Frequency and Relative Frequency: Toggles the display between reporting frequencies and relative frequencies.

Plot All Data: Plots all of the data in the chart, that is, each frequency is plotted as a single bar.

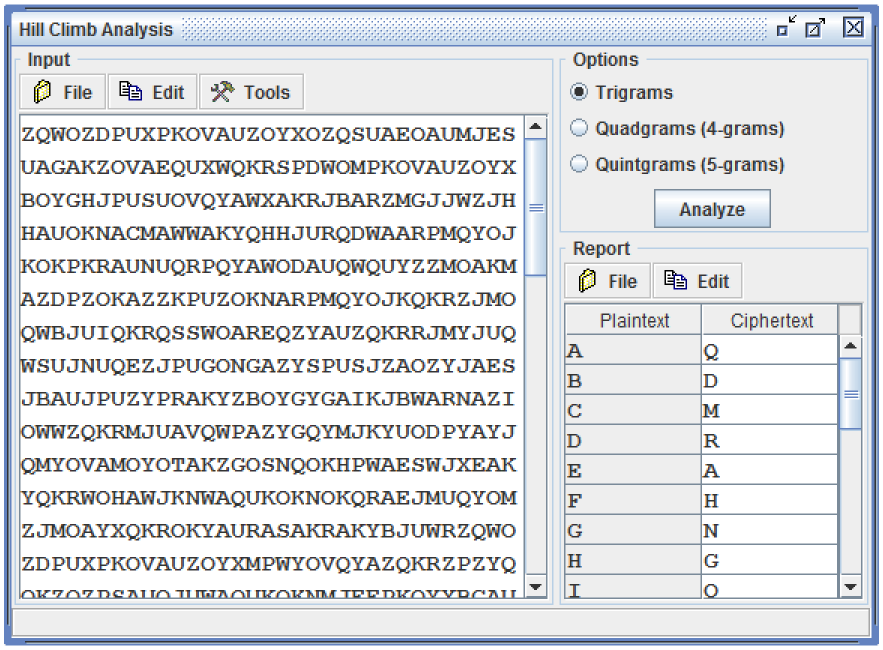
Plot Range of Data: Plots a selected range of the data. When the user selects this option a small dialog box will appear asking for the number of data items to plot. When the user selects the number of items the graph is changed to have only that number of bars showing. Also there will appear a slider tool at the bottom of the chart that will allow the user to slide the graph left and right, in order to see the entire set of data.

3.4.2 Hill Climb Analysis

The Hill Climb Analysis window will apply the hill climbing algorithm for substitution ciphers to the input text. When the analysis is finished, the best guess to the substitution cipher key will be displayed in the report grid.

How to Use the Tool

1. Input the ciphertext into the Input box.
2. Select the analysis option, either using the trigram, quadgram or quintgram data files.
3. Click on the Analyze button. The process may take several seconds to complete, depending on the size of the ciphertext, which data set you choose to use and the number of passes that are made in the analysis.



```
ZQWOZDPUXPKOVAUZOYXOZQSUAEOAUMJES
UAGAKZOVAEQUXWQKRSPDWOMPKOVAUZOYX
BOYGHJPUSUOVQYAWXAKRJBARZMGJJWZJH
HAUOKNACMAWWAKYQHHJURQDWAARPMQYOJ
KOKPKRAUNUQRPQYAWODAUQWQUYZZMOAKM
AZDPZOKAZZKPUZOKNARPMQYOJKQKRZJMO
QWBJUIQKRQSSWOAREQZYAUZQKRRJMYJUQ
WSUJNUQEZJPUGONGAZYSPUSJZAOZYJAES
JBAUJPUZYPRAKYZBOYGYGAIKJBWARNAZI
OWWZQKRMJUAVQWPAZYGQYMJKYUODPYAYJ
QMYOVAMOYOTAKZGOSNQOKHPWAESWJXEAK
YQKRWOHAWJKNWAQUKOKNOKQRAEJMUQYOM
ZJMOAYXQKROKYAURASAKRAKYBJUWRZQWO
ZDPUXPKOVAUZOYXMPWYOVQYAZQKRZPZYQ
```

[✓ Trigrams] Quadgrams (4-grams) Quintgrams (5-grams)

Plaintext	Ciphertext
A	Q
B	D
C	M
D	R
E	A
F	H
G	N
H	G
I	O

Figure 3.24: Hill Climb Analysis Tool

Options

- The data set options are as follows.

  Trigrams: This is a data set of 17,556 trigrams and frequencies taken from 4,274,127,909 English text trigrams.

  Quadgrams: This is a data set of 389,373 quadgrams and frequencies taken from 4,224,127,912 English text quadgrams.

  Quintgrams: This is a data set of 4,354,914 quintgrams and frequencies taken from 4,174,127,916 English text quintgrams.

- The Report Table has a toolbar with the following options.

  File —

  Save As: Saves the current report table to a text file.
  Print: Prints the current report table to the selected printer.
  Print Preview: Prints the current report table to the print preview display.

  Edit —

  Copy: Copies the entire report table to the clipboard.
  Copy as LaTeX (tabular): Copies the entire report table to the clipboard using the syntax for the LaTeX tabular environment.
  Copy as LaTeX (longtable): Copies the entire report table to the clipboard using the syntax for the LaTeX longtable environment.

Notes

- There are many ways to implement a hill climbing algorithm on a substitution cipher. This implementation uses the following algorithm.

  1. The ciphertext is first frequency analyzed using single characters and assigned a preliminary key by frequency. The most frequent letter assigned to E, the next assigned to T, then A, then O, and so on.
  2. At this point the hill climbing step starts. The fitness measure of the single character frequency substitution is calculated.
  3. The program will then begin transposing the substitution key entries, starting with A and B, then A and C, down to A and Z, then B and C, down to B and Z, and so on until Y and Z. After each transposition is done, the ciphertext is converted to a possible plaintext and the fitness measure is calculated on that possible plaintext. If this measure is larger than the previous one, the new substitution key is used and if not, the transposed characters are reassigned to their original positions.
  4. Once all of the possible transpositions are done, we consider that a single pass. If the fitness measure after a pass is larger than the fitness measure before the pass the program will make another pass. If the fitness measure does not increase during a pass, the program will consider the current substitution key the best guess and display that key.

  The fitness measure is calculated by taking the sum of the logarithms (base 10) of the probabilities of each of the trigrams, quadgrams, or quintgrams in the converted ciphertext.

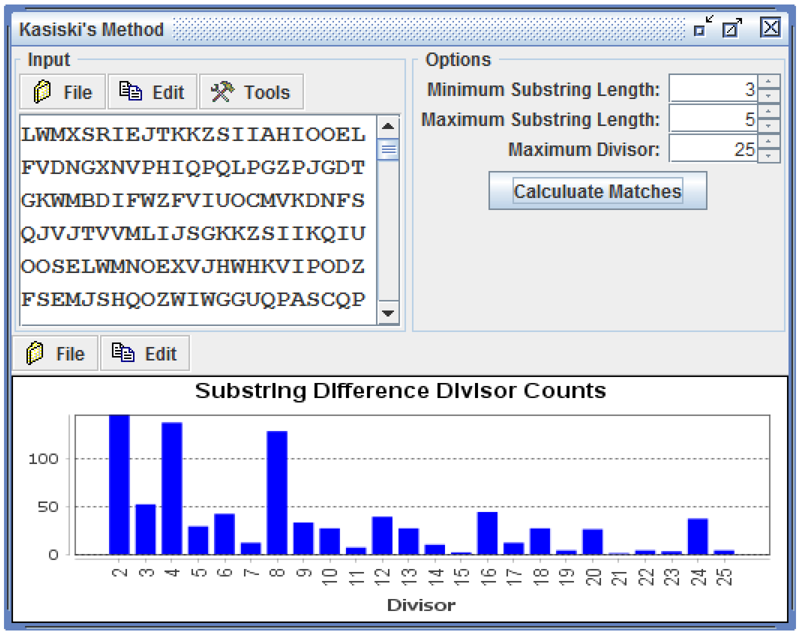
- Depending on your ciphertext, using a different data set may produce better results. In some cases, using trigrams may get closer to the substitution key than using quadgrams or quintgrams.

- Since the hill climbing algorithm may take several passes, this process might, on average, take a few seconds to complete. In most cases, unless you have an extraordinarily long ciphertext to analyze, the process will only take a couple seconds. Nonetheless, we have placed the algorithm in a worker thread of execution so that the program is not locked out during the process. At the bottom of the window is a status bar that displays the current progress of the algorithm. The display shows the current pass, the percentage of that pass that is complete, the fitness measure of the current best key being examined, and on the far right is the elapsed time of the algorithm.

3.4.3 Kasiski's Method

Kasiski's Method, which is also known as Kasiski's Test or Kasiski examination was developed by Friedrich Kasiski[2] in 1863 but it seems to have been independently discovered by Charles Babbage[1] as early as 1846.

The Kasiski's Method window will report the divisor counts of the distances between equal substrings of a given length of the input. This is one of the possible tools for determining the length of a Vigenère cipher keyword.



```
LWMXSRIEJTKKZSIIAHIOOEL
FVDNGXNVPHIQPQLPGZPJGDT
GKWMBDIFWZFVIUOCMVKDNFS
QJVJTVVMLIJSGKKZSIIKQIU
OOSELWMNOEXVJHWHKVIPODZ
FSEMJSHQOZWIWGGUQPASCQP
```

Minimum Substring Length: 3
Maximum Substring Length: 5
Maximum Divisor: 25

Figure 3.25: Kasiski's Method Analysis Tool

How to Use the Tool

1. Input the ciphertext into the Input box.
2. Select the minimum and maximum substring length to be tested.
3. Select the maximum divisor to be tested for each substring occurrence difference.
4. Click on the Calculate Matches button. At this point the number of divisors of the differences between the substring matches will be displayed in the report bar chart.

Options

- The Report Bar Chart has a toolbar with the following options.

  File —

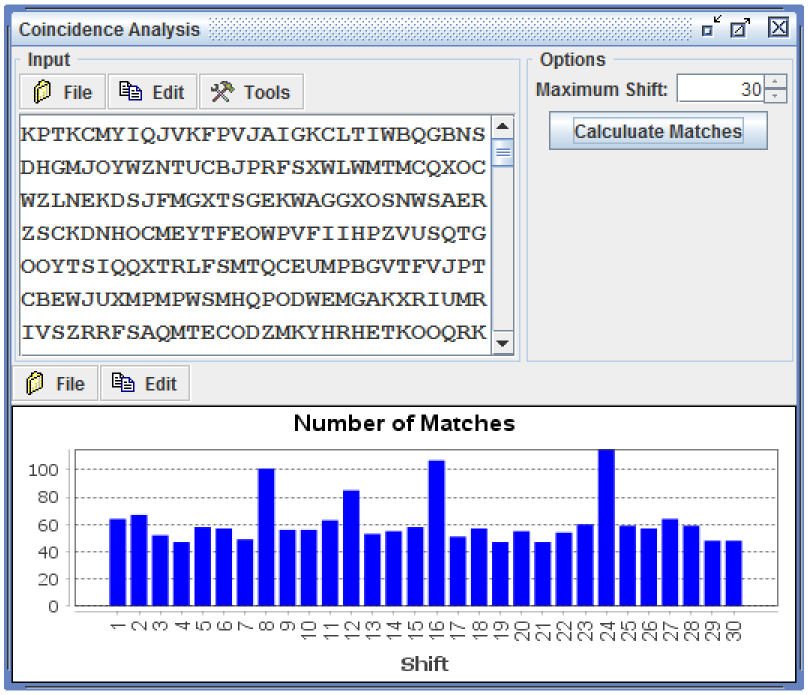
  Save As: Saves the current chart to a png file.
  Print: Prints the current chart to the selected printer.
  Print Preview: Prints the current chart to the print preview display.

  Edit —

  Copy: Copies the chart to the clipboard.

3.4.4 Coincidence Analysis

The Coincidence Analysis window will report the number of matches of characters in the same position between the original text and shifts of that text. This is one of the possible tools for determining the length of a Vigenère cipher keyword.



Maximum Shift: 30

```
KPTKCMYIQJVKFPVJAIGKCLTIWBQGBNS
DHGMJOYWZNTUCBJPRFSXWLWMTMCQXOC
WZLNEKDSJFMGXTSGEKWAGGXOSNWSAER
ZSCKDNHOCMEYTFEOWPVFIIHPZVUSQTG
OOYTSIQQXTRLFSMTQCEUMPBGVTFVJPT
CBEWJUXMPMPWSMHQPODWEMGAKXRIUMR
IVSZRRFSAQMTECODZMKYHRHETKOOQRK
```

Figure 3.26: Coincidence Analysis Tool

How to Use the Tool

1. Input the ciphertext into the Input box.
2. Select the maximum shift.
3. Click on the Calculate Matches button. At this point the number of matches between shifts of 1 and the maximum will be displayed in the report bar chart.

Options

- The Report Bar Chart has a toolbar with the following options.

  File —

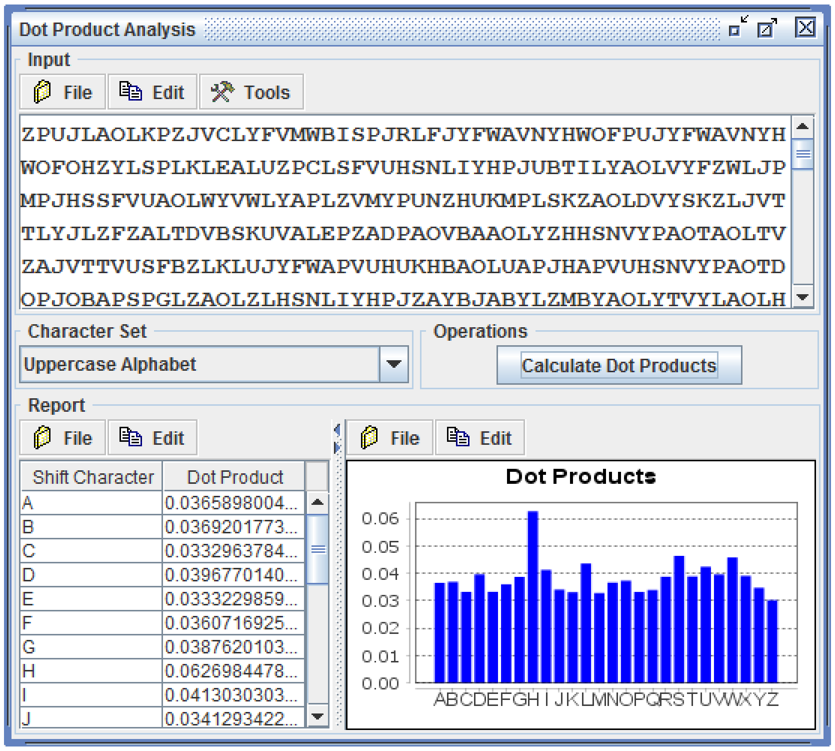
  > **Save As:** Saves the current chart to a png file.
  > **Print:** Prints the current chart to the selected printer.
  > **Print Preview:** Prints the current chart to the print preview display.

Edit —

Copy: Copies the chart to the clipboard.

3.4.5 Dot Product Analysis

The Dot Product Analysis window will analyze dot products between the relative frequency counts of the given text and shifts of the relative frequencies of characters in the selected language. If the user selects the uppercase alphabet as the character set the program will use the relative frequencies of characters in the English language and if the user selects a user defined language the relative frequencies of that language will be used. This is one numeric technique for determining if a shift cipher was used and determining the shift.



```
ZPUJLAOLKPZJVCLYFVMWBISPJRLFJYFWAVNYHWOFPUJYFWAVNYH
WOFOHZYLSPLKLEALUZPCLSFVUHSNLIYHPJUBTILYAOLVYFZWLJP
MPJHSSFVUAOLWYVWLYAPLZVMYPUNZHUKMPLSKZAOLDVYSKZLJVT
TLYJLZFZALTDVBSKUVALEPZADPAOVBAAOLYZHHSNVYPAOTAOLTV
ZAJVTTVUSFBZLKLUJYFWAPVUHUKHBAOLUAPJHAPVUHSNVYPAOTD
OPJOBAPSPGLZAOLZLHSNLIYHPJZAYBJABYLZMBYAOLYTVYLAOLH
```

Character Set: Uppercase Alphabet

Shift Character	Dot Product
A	0.0365898004...
B	0.0369201773...
C	0.0332963784...
D	0.0396770140...
E	0.0333229859...
F	0.0360716925...
G	0.0387620103...
H	0.0626984478...
I	0.0413030303...
J	0.0341293422...

Figure 3.27: Dot Product Analysis Tool

How to Use the Tool

1. Input the ciphertext into the Input box.
2. Select the character set to use.
3. Click on the Calculate Dot Products button. At this point the dot products of all possible shifts will be displayed in the report table and the same data will be displayed in the report bar chart.

Options

- The character sets are as follows:

  Uppercase Alphabet: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

  User Defined Language: This will open an Open dialog box that will allow you to select a user defined language. Information on creating a user defined language can be found in the section on the *User Defined Language Creator*.

- The Report Table has a toolbar with the following options.

  File —

  Save As: Saves the current report table to a text file.

  Print: Prints the current report table to the selected printer.

  Print Preview: Prints the current report table to the print preview display.

  Edit —

  Copy: Copies the entire report table to the clipboard.

  Copy as LaTeX (tabular): Copies the entire report table to the clipboard using the syntax for the LaTeX tabular environment.

  Copy as LaTeX (longtable): Copies the entire report table to the clipboard using the syntax for the LaTeX longtable environment.

- The Report Bar Chart has a toolbar with the following options.

  File —

  Save As: Saves the current chart to a png file.

  Print: Prints the current chart to the selected printer.

  Print Preview: Prints the current chart to the print preview display.

  Edit —

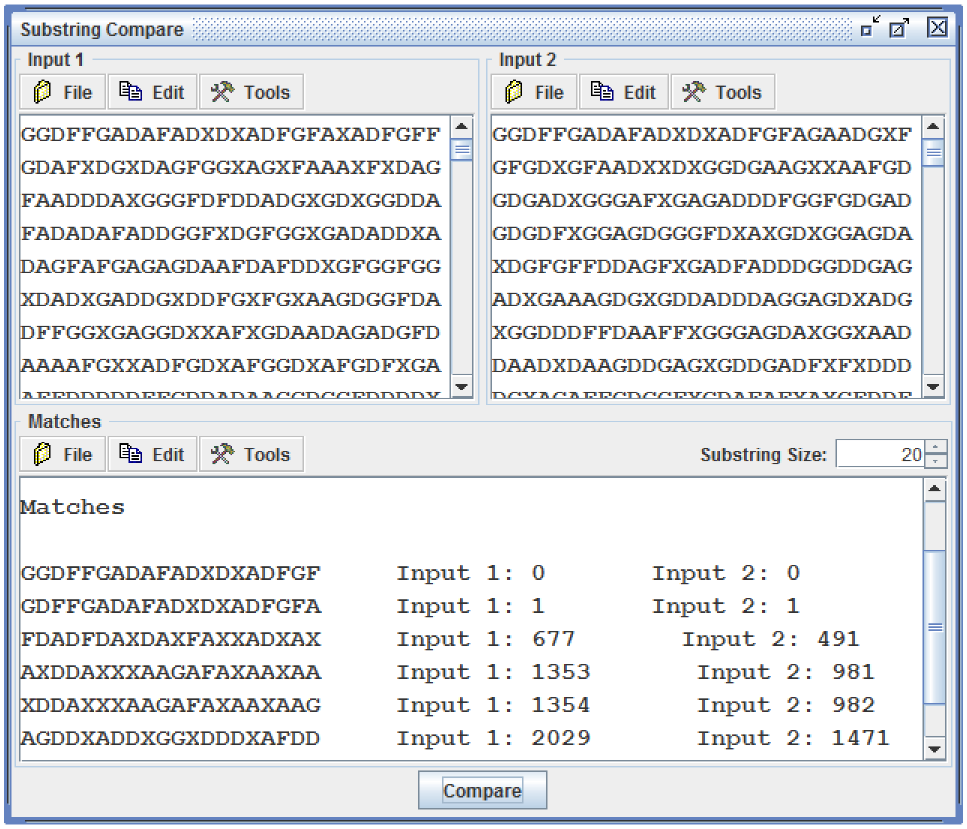
  Copy: Copies the chart to the clipboard.

3.4.6 Substring Compare

The Substring Compare window finds matches between two strings and reports the positions of those matches. This is primarily a tool for cracking ADFGX and ADFGVX ciphers.

How to Use the Tool

1. Input the ciphertext of the two encryptions into the two Input boxes.
2. Select the Substring Size to use.
3. Click on the Compare button at the bottom of the window. At this point all matches of substrings of the given size will appear in the Matches output box.



Input 1:

```
GGDFFGADAFADXDXADFGFAXADFGFF
GDAFXDGXDAGFGGXAGXFAAAXFXDAG
FAADDDAXGGGFDFDDADGXGDXGGDDA
FADADAFADDGGFXDGFGGXGADADDXA
DAGFAFGAGAGDAAFDAFDDXGFGGFGG
XDADXGADDGXDDFGXFGXAAGDGGFDA
DFFGGXGAGGDXXAFXGDAADAGADGFD
AAAAFGXXADFGDXAFGGDXAFGDFXGA
```

Input 2:

```
GGDFFGADAFADXDXADFGFAGAADGXF
GFGDXGFAADXXDXGGDGAAGXXAAFGD
GDGADXGGGAFXGAGADDDFGGFGDGAD
GDGDFXGGAGDGGGFDXAXGDXGGAGDA
XDGFGFFDDAGFXGADFADDDGGDDGAG
ADXGAAAGDGXGDDADDDAGGAGDXADG
XGGDDDFFDAAFFXGGGAGDAXGGXAAD
DAADXDAAGDDGAGXGDDGADFXFXDDD
```

Substring Size: 20

```
Matches


GGDFFGADAFADXDXADFGF     Input 1: 0        Input 2: 0
GDFFGADAFADXDXADFGFA     Input 1: 1        Input 2: 1
FDADFDAXDAXFAXXADXAX     Input 1: 677        Input 2: 491
AXDDAXXXAAGAFAXAAXAA     Input 1: 1353        Input 2: 981
XDDAXXXAAGAFAXAAXAAG     Input 1: 1354        Input 2: 982
AGDDXADDXGGXDDDXAFDD     Input 1: 2029        Input 2: 1471
```

Figure 3.28: Substring Comparison Tool

3.4.7 LFSR Cipher Analysis

The LFSR Cipher Analysis window is for determining the key recurrence relation given a portion of the key. This tool has facilities for calculating the determinants (modulo 2) of square matrices produced by the consecutive digit stream and for modulo 2 reduction of a resultant matrix of specified size.

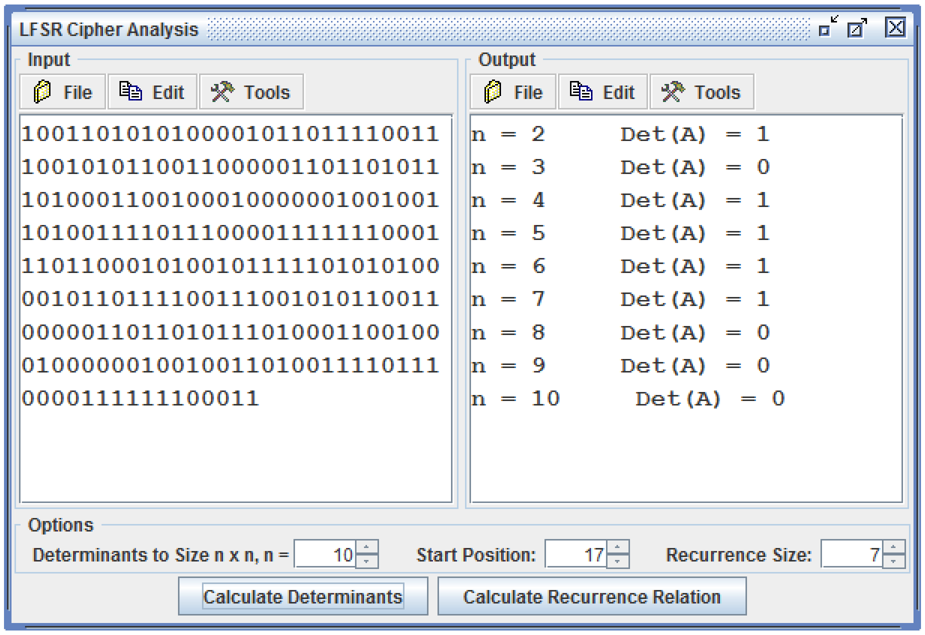
How to Use the Tool

Determining the Relation Generator Length —

1. Input the key stream into the Input box.
2. Select the maximum determinant size and the starting position in the stream.
3. Click on the Calculate Determinants button. At this point you will see a list of determinants from 2 X 2 to n X n, where n is the maximum size that was selected.

Determining the Relation Generator String —

1. After the above analysis, input the recurrence size.

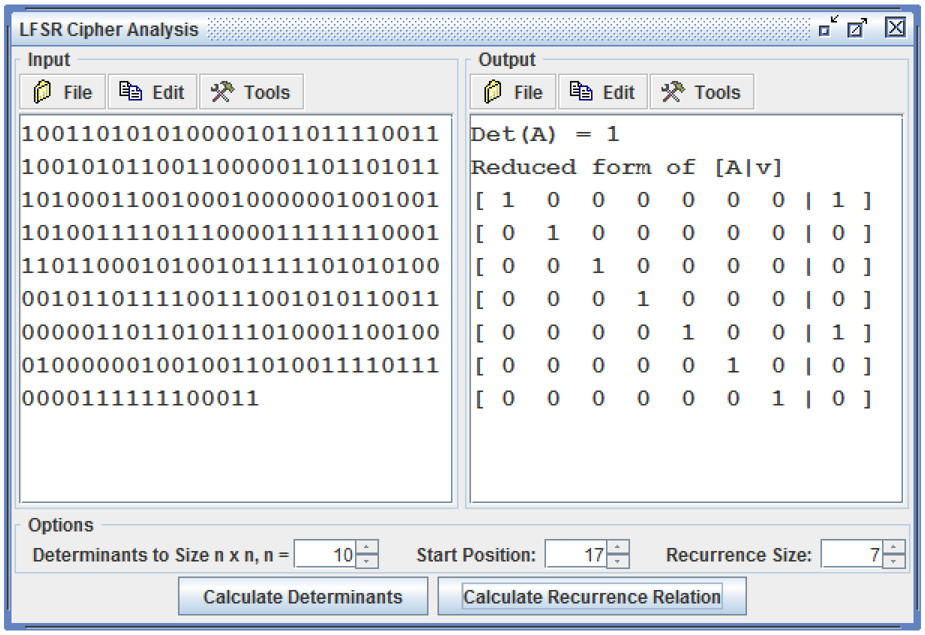


```
1001101010100001011011110011
1001010110011000001101101011
1010001100100010000001001001
1010011110111000011111110001
1101100010100101111101010100
0010110111100111001010110011
0000011011010111010001100100
0100000010010011010011110111
0000111111100011
```

```
n = 2     Det(A) = 1
n = 3     Det(A) = 0
n = 4     Det(A) = 1
n = 5     Det(A) = 1
n = 6     Det(A) = 1
n = 7     Det(A) = 1
n = 8     Det(A) = 0
n = 9     Det(A) = 0
n = 10     Det(A) = 0
```

Determinants to Size n x n, n = 10 Start Position: 17 Recurrence Size: 7

Figure 3.29: LFSR Analysis Tool: Determinants



```
1001101010100001011011110011
1001010110011000001101101011
1010001100100010000001001001
1010011110111000011111110001
1101100010100101111101010100
0010110111100111001010110011
0000011011010111010001100100
0100000010010011010011110111
0000111111100011
```

```
Det(A) = 1
Reduced form of [A|v]
[ 1  0  0  0  0  0  0 | 1 ]
[ 0  1  0  0  0  0  0 | 0 ]
[ 0  0  1  0  0  0  0 | 0 ]
[ 0  0  0  1  0  0  0 | 0 ]
[ 0  0  0  0  1  0  0 | 1 ]
[ 0  0  0  0  0  1  0 | 0 ]
[ 0  0  0  0  0  0  1 | 0 ]
```

Determinants to Size n x n, n = 10 Start Position: 17 Recurrence Size: 7

Figure 3.30: LFSR Analysis Tool: Relation

2. Click on the Calculate Recurrence Relation button. The output will display the determinant of the coefficient matrix as well as the reduced augmented matrix. If the size is correct the relation bits (coefficients) will be in the augment, in order.

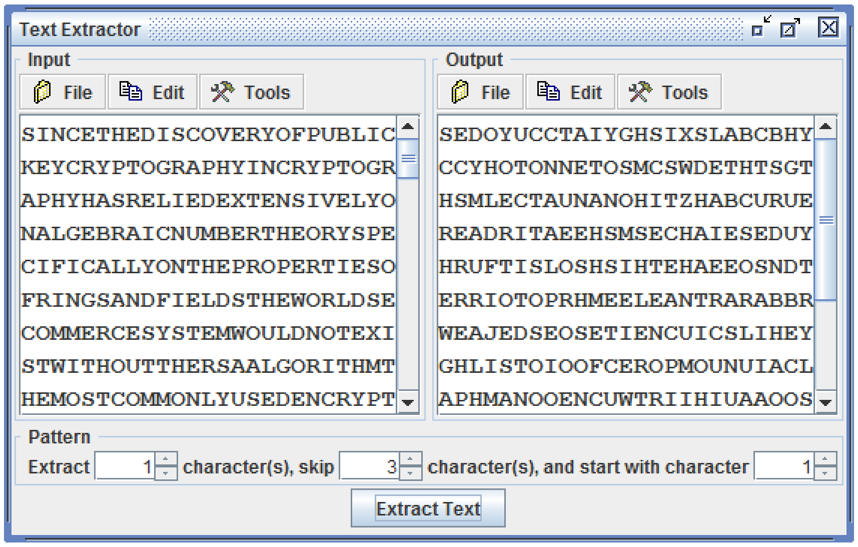
Notes

- The starting position is the bit where the extraction will begin. This number should be set to skip the probable size of the key seed, since the seed may not follow the relation.

3.5 Text Tools

3.5.1 Text Extractor

The Text Extractor window is for taking some input text and extracting a substring using a predefined pattern. The pattern is defined by extracting X characters, then skipping Y characters starting at character Z.



```
SINCETHEDISCOVERYOFPUBLIC
KEYCRYPTOGRAPHYINCRYPTOGR
APHYHASRELIEDEXTENSIVELYO
NALGEBRAICNUMBERTHEORYSPE
CIFICALLYONTHEPROPERTIESO
FRINGSANDFIELDSTHEWORLDSE
COMMERCESYSTEMWOULDNOTEXI
STWITHOUTTHERSAALGORITHMT
HEMOSTCOMMONLYUSEDENCRYPT
```

```
SEDOYUCCTAIYGHSIXSLABCBHY
CCYHOTONNETOSMCSWDETHTSGT
HSMLECTAUNANOHITZHABCURUE
READRITAEEHSMSECHAIESEDUY
HRUFTISLOSHSIHTEHAEEOSNDT
ERRIOTOPRHMEELEANTRARABBR
WEAJEDSEOSETIENCUICSLIHEY
GHLISTOIOOFCEROPMOUNIACL
APHMANOOENCUWTRIIHIUAAOOS
```

Extract 1 character(s), skip 3 character(s), and start with character 1

Figure 3.31: Text Extractor Tool

How to Use the Tool

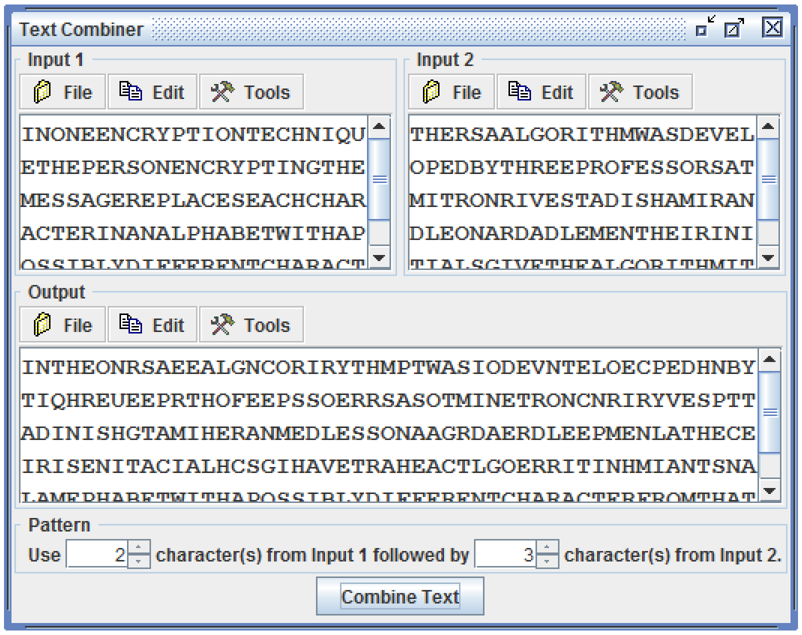
1. Input the text you wish to extract from into the Input box.
2. Select the extraction pattern.
3. Click on the Extract Text button. At this point the number of matches between shifts of 1 and the maximum will be displayed in the report bar chart.

Notes

- The extraction pattern is defined by extracting X characters, then skipping Y characters starting at character Z. For example, if the input string is CRYPTOGRAPHY and the pattern is to extract 1 character, skip 3 starting at 2 then the extracted text would be ROP.

3.5.2 Text Combiner

The Text Combiner window will take two input strings and combine them into one string. The combining pattern is of the form, taking X characters from one string and then Y characters from the second, and so on. If there are any characters left in one of the strings the remainder is placed at the end.



```
INONEENCRYPTIONTECHNIQU
ETHEPERSONENCRYPTINGTHE
MESSAGEREPLACESEACHCHAR
ACTERINANALPHABETWITHAP
OSSIBLYDIFFERENTCHARACT
```

```
THERSAALGORITHMWASDEVEL
OPEDBYTHREEPROFESSORSAT
MITRONRIVESTADISHAMIRAN
DLEONARDADLEMENTHEIRINI
TIALSGIVETHEALGORITHMIT
```

```
INTHEONRSAEEALGNCORIRYTHMPTWASIODEVNTELOECPEDHNBY
TIQHREUEEPRTHOFEEPSSOERRSASOTMINETRONCNRIRYVESPTT
ADINISHGTAMIHERANMEDLESSONAAGRDAERDLEEPMENLATHECE
IRISENITACIALHCSGIHAVETRAHEACTLGOERRITINHMIANTSNA
LAMEPHABETWITHAPOSSIBLYDIFFERENTCHARACTERFROMTHAT
```

Use 2 character(s) from Input 1 followed by 3 character(s) from Input 2.

Figure 3.32: Text Combiner Tool

How to Use the Tool

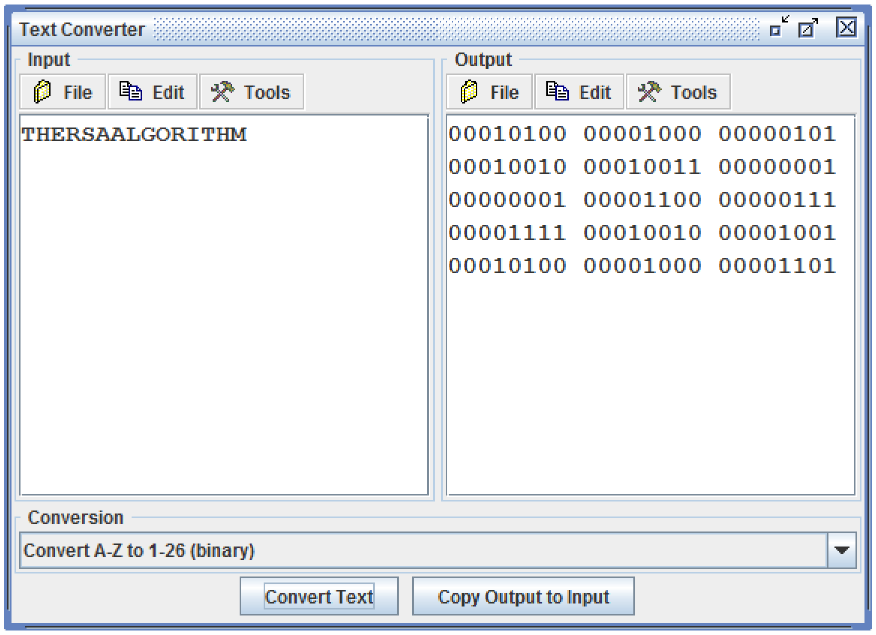
1. Input the two texts you wish to combine into the two Input boxes.
2. Select the combining pattern.
3. Click on the Combine Text button.

Notes

- The combining pattern is of the form, taking X characters from one string and then Y characters from the second, and so on. If there are any characters left in one of the strings the remainder is placed at the end. For example, if the input 1 is INONEENCRYPTION and input 2 is THERSAALGORITHM and the pattern is two from input 1 followed by 3 from input two the result of the combine would be INTHEONRSAEEALGNCORIRYTHMPTION.

3.5.3 Text Converter

The Text Converter is a simple conversion program that will convert strings into other strings. All conversions that can be done here are also possible through the Tools menu in each input box.



```
THERSAALGORITHM
```

```
00010100 00001000 00000101
00010010 00010011 00000001
00000001 00001100 00000111
00001111 00010010 00001001
00010100 00001000 00001101
```

Conversion: Convert A-Z to 1-26 (binary)

Figure 3.33: Text Converter Tool

How to Use the Tool

1. Input the text you wish to convert into the Input box.
2. Select the conversion.
3. Click on the Convert Text button.
4. If you wish to do several conversions, there is a button that will copy the text from the output box into the input box.

Options

Many of the options for text conversion are fairly obvious but we list the options below.

Convert to UPPERCASE: Converts the input box contents to uppercase.

Convert to lowercase: Converts the input box contents to lowercase.

Remove White Space: Removes all whitespace from the input box contents. Spaces, returns, tabs, etc. are removed.

Remove Punctuation: Removes all punctuation marks from the input box contents.

Remove Numbers: Removes all numbers from the input box contents.

Change J to I: Replaces all occurrences of J with I, in the input box contents. The case of the j's is not altered, so if the j is lowercase it will be replaced with a lowercase i and if the J is uppercase it will be replaced with an uppercase I.

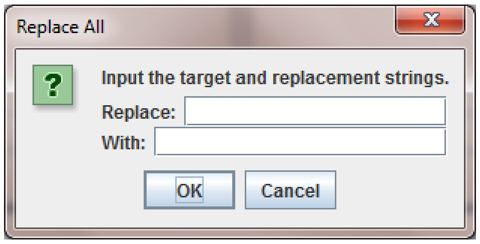
Remove Double Characters: This will place an X between any double characters and if the double character is XX it will place a Z between them. So OO would be changed to OXO and XX to XZX.

Convert Spaces to Line Breaks: Converts all spaces in the input box contents to line breaks.

Convert Line Breaks to Spaces: Converts all line breaks in the input box contents to spaces.

Convert White Space to Single Spaces: Converts all white space in the input box contents to a single space.

Replace All...: Replaces every occurrence of the Replace target string with the With string.


Replace All

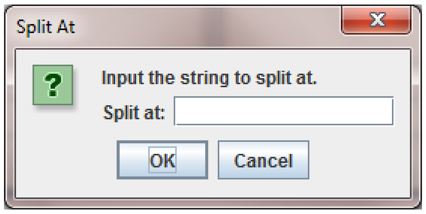
Input the target and replacement strings.

Replace:

With:

Figure 3.34: Replace All Dialog

Split At...: Splits the contents of the input box at the Split At string. The split at string is removed from the input box contents and replaced with line breaks.


Split At

Input the string to split at.

Split at:

Figure 3.35: Split All Dialog

Convert A-Z to 0-25: Converts the uppercase A–Z to the numbers 0–25. If the input box contents have characters not in the A–Z range the program will generate an error.

Convert A-Z to 00-25: Converts the uppercase A–Z to the numbers 00–25, that is, it will use two characters per letter. If the input box contents have characters not in the A–Z range the program will generate an error.

Convert 0-25 to A-Z: Converts the numbers 0–25 to the letters A–Z. The numbers must be separated by a space. This will work for numbers in the range of 00–25 as well, that is, using two digits for each number.

Convert A-Z to 1-26: Converts the uppercase A–Z to the numbers 1–26. If the input box contents have characters not in the A–Z range the program will generate an error.

Convert A-Z to 01-26: Converts the uppercase A–Z to the numbers 01–26, that is, it will use two characters per letter. If the input box contents have characters not in the A–Z range the program will generate an error.

Convert 1-26 to A-Z: Converts the numbers 1–26 to the letters A–Z. The numbers must be separated by a space. This will work for numbers in the range of 01–26 as well, that is, using two digits for each number.

Convert A-Z to 0-25 (binary): Converts the uppercase A–Z to the numbers 0–25, in binary form, using 8 bits per letter. If the input box contents have characters not in the A–Z range the program will generate an error.

Convert 0-25 (binary) to A-Z: Converts binary numbers in the range of 0–25 to the letters A–Z.

Convert A-Z to 1-26 (binary): Converts the uppercase A–Z to the numbers 1–26, in binary form, using 8 bits per letter. If the input box contents have characters not in the A–Z range the program will generate an error.

Convert 1-26 (binary) to A-Z: Converts binary numbers in the range of 1–26 to the letters A–Z.

Convert Text to ASCII: Converts each character of the text in the input box to the character's ASCII number.

Convert ASCII to Text: Converts each ASCII number in the input box to the ASCII character. Each ASCII number must be separated by a space.

Convert Text to ASCII Stream Using 3 Decimal Numbers/Character: Converts each character of the text in the input box to the character's ASCII number, but uses three digits for each number, that is, smaller numbers are padded with 0's.

Convert ASCII Stream to Text Using 3 Decimal Numbers/Character: Converts each three digit ASCII number in the input box to the ASCII character. Each ASCII number must be separated by a space.

Convert Text to ASCII (binary): Converts each character of the text in the input box to the character's ASCII number, in binary, using 8 bits per number.

Convert ASCII (binary) to Text: Converts each 8-bit binary number to its respective ASCII character. The binary numbers must be separated by a space.

Convert Decimal to Binary: Converts a decimal number to binary.

Convert Binary to Decimal: Converts a binary number to a decimal.

Convert Decimal to Octal: Converts a decimal number to octal.

Convert Octal to Decimal: Converts an octal number to decimal.

Convert Decimal to Hexadecimal: Converts a decimal number to hexadecimal.

Convert Hexadecimal to Decimal: Converts a hexadecimal number to decimal.

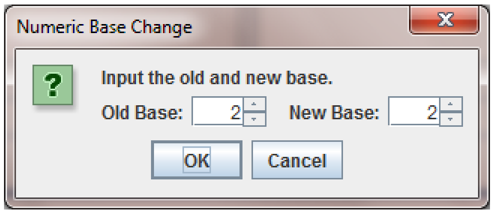
Convert Binary to Octal: Converts a binary number to octal.

Convert Octal to Binary: Converts an octal number to binary.

Convert Binary to Hexadecimal: Converts a binary number to hexadecimal.

Convert Hexadecimal to Binary: Converts a hexadecimal number to binary.

Change Numeric Base...: This will open up a dialog box that allows the user to select an old base and a new base, the old base represents the base of the current number and the new base is the converted base. When the old base is larger than 10, the input can have the letters A, B, C, D, E, and F to represent "digits" of 10, 11, 12, 13, 14, and 15 respectively.


Numeric Base Change

Input the old and new base.

Old Base: 2 New Base: 2

Figure 3.36: Base Change Dialog

Convert Text to Morse Code: Converts the letters A–Z to Morse Code. The Morse Code representation is uses - and ..

Convert Morse Code to Text: Converts the Morse Code represented as - and . to the letters A–Z.

Break Binary Stream into Blocks of 8: Breaks a binary stream of 0's and 1's onto blocks of 8. If the input is not 0's and 1's the program will generate an error.

Break Binary Stream into Blocks of 16: Breaks a binary stream of 0's and 1's onto blocks of 16. If the input is not 0's and 1's the program will generate an error.

Break Binary Stream into Blocks of 32: Breaks a binary stream of 0's and 1's onto blocks of 32. If the input is not 0's and 1's the program will generate an error.

Break Binary Stream into Blocks of 64: Breaks a binary stream of 0's and 1's onto blocks of 64. If the input is not 0's and 1's the program will generate an error.

Break Binary Stream into Blocks of 128: Breaks a binary stream of 0's and 1's onto blocks of 128. If the input is not 0's and 1's the program will generate an error.

Break Binary Stream into Blocks of 256: Breaks a binary stream of 0's and 1's onto blocks of 256. If the input is not 0's and 1's the program will generate an error.

Break Binary Stream into Blocks of 512: Breaks a binary stream of 0's and 1's onto blocks of 512. If the input is not 0's and 1's the program will generate an error.

Break Binary Stream into Blocks of 1024: Breaks a binary stream of 0's and 1's onto blocks of 1024. If the input is not 0's and 1's the program will generate an error.

Break Binary Stream into Blocks of n...: Breaks a binary stream of 0's and 1's onto blocks of size n. If the input is not 0's and 1's the program will generate an error. When selected a dialog box will open allowing the user to select the block size.

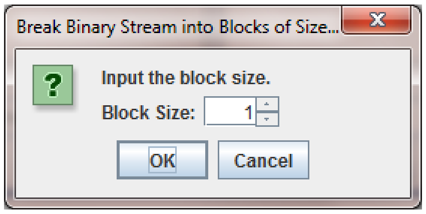


Figure 3.37: Binary Stream Block Dialog

Break ASCII Stream into Blocks of 3: Breaks a number stream onto blocks of 3. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 6: Breaks a number stream onto blocks of 6. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 9: Breaks a number stream onto blocks of 9. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 12: Breaks a number stream onto blocks of 12. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 15: Breaks a number stream onto blocks of 15. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 18: Breaks a number stream onto blocks of 18. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 21: Breaks a number stream onto blocks of 321. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 24: Breaks a number stream onto blocks of 24. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 27: Breaks a number stream onto blocks of 27. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 30: Breaks a number stream onto blocks of 30. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 45: Breaks a number stream onto blocks of 45. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 60: Breaks a number stream onto blocks of 60. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of 90: Breaks a number stream onto blocks of 90. If the input is not a stream of numbers with no spaces the program will generate an error.

Break ASCII Stream into Blocks of n...: Breaks a number stream onto blocks of size n. If the input is not a stream of numbers with no spaces the program will generate an error. When selected a dialog box will open allowing the user to select the block size.

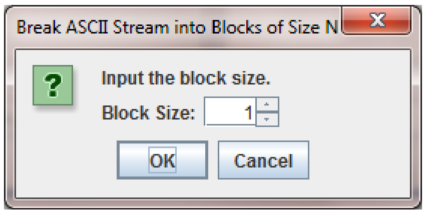


Figure 3.38: ASCII Stream Block Dialog

Break Character Stream into Blocks of 1: Breaks a character stream onto blocks of 1.

Break Character Stream into Blocks of 2: Breaks a character stream onto blocks of 2.

Break Character Stream into Blocks of 3: Breaks a character stream onto blocks of 3.

Break Character Stream into Blocks of 4: Breaks a character stream onto blocks of 4.

Break Character Stream into Blocks of 5: Breaks a character stream onto blocks of 5.

Break Character Stream into Blocks of 10: Breaks a character stream onto blocks of 10.

Break Character Stream into Blocks of 15: Breaks a character stream onto blocks of 15.

Break Character Stream into Blocks of 20: Breaks a character stream onto blocks of 20.

Break Character Stream into Blocks of 25: Breaks a character stream onto blocks of 25.

Break Character Stream into Blocks of 30: Breaks a character stream onto blocks of 30.

Break Character Stream into Blocks of 40: Breaks a character stream onto blocks of 40.

Break Character Stream into Blocks of 50: Breaks a character stream onto blocks of 50.

Break Character Stream into Blocks of 75: Breaks a character stream onto blocks of 75.

Break Character Stream into Blocks of 80: Breaks a character stream onto blocks of 80.

Break Character Stream into Blocks of 100: Breaks a character stream onto blocks of 100.

Break Character Stream into Blocks of n...: Breaks a character stream onto blocks of size n. When selected a dialog box will open allowing the user to select the block size.

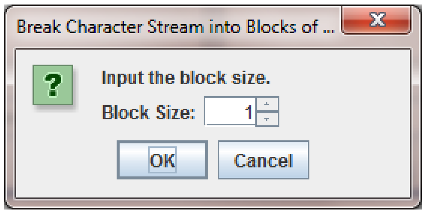
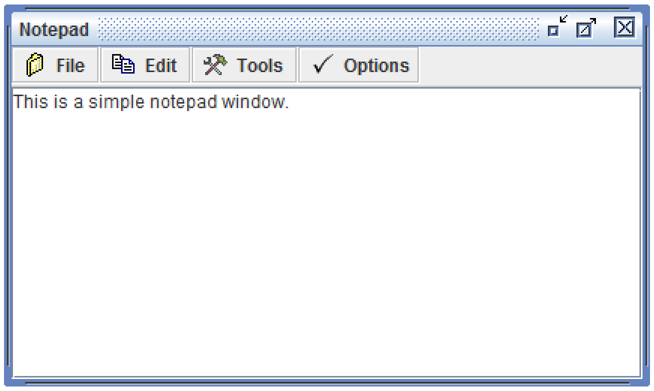


Figure 3.39: Character Stream Block Dialog

3.5.4 Notepad

The Notepad window is a simple text editing window. The editing window is a standard Input box, with the usual file, editing and text conversion tools. The notepad also has one other option in the Options menu, to toggle the wrap mode of the box between wrapping at word breaks and wrapping at the character level.

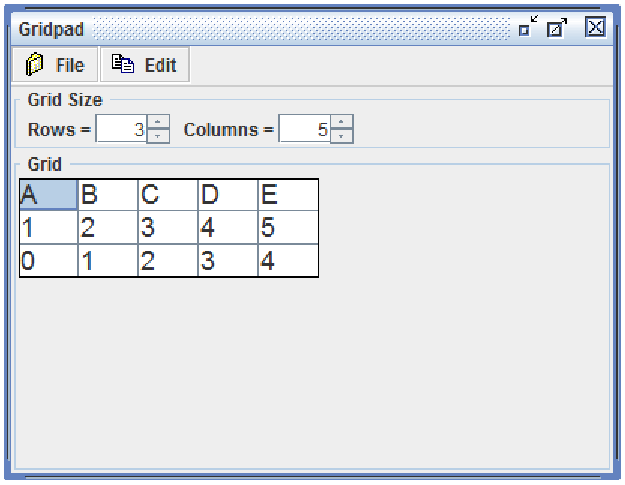


This is a simple notepad window.

Figure 3.40: Notepad

3.5.5 Gridpad

The Gridpad is essentially a grid-based notepad, it is simply a place for the user to input grid or spreadsheet type data into a convenient grid tool. This tool is not a spreadsheet, there are no numeric tools available in this grid, it is simply to ease the display of grid data.



Rows = 3 Columns = 5

A	B	C	D	E
1	2	3	4	5
0	1	2	3	4

Figure 3.41: Gridpad

How to Use the Tool

1. Select the size of the desired grid.
2. Input the desired data.
3. Select any desired options from the menu.

Options

- The toolbar with the following options.

  File —

  - **New:** Removes the current data from the grid and resizes the grid to 3 X 3.
  - **Open:** Opens a tab delimited text file and loads the data into the grid.
  - **Save As:** Saves the current grid to a text file.
  - **Print:** Prints the current grid to the selected printer.
    - **Grid Only:** Prints the contents of the grid.
    - **Row One as Header:** Prints the contents of the grid using the first row as a header row.
    - **Row One and Column One as Headers:** Prints the contents of the grid using the first row as a header row and the first column as a header column.
  - **Print Preview:** Prints the current grid to the print preview display.
    - **Grid Only:** Prints the contents of the grid.
    - **Row One as Header:** Prints the contents of the grid using the first row as a header row.
    - **Row One and Column One as Headers:** Prints the contents of the grid using the first row as a header row and the first column as a header column.

  Edit —

  - **Copy:** Copies the grid to the clipboard.
  - **Copy as LaTeX (tabular, No Header):** Copies the entire grid to the clipboard using the syntax for the LaTeX tabular environment. The tabular code treats all grid entries equally, no data is used as a header.
  - **Copy as LaTeX (longtable, No Header):** Copies the entire grid to the clipboard using the syntax for the LaTeX longtable environment. The longtable code treats all grid entries equally, no data is used as a header.
  - **Copy as LaTeX (tabular, With Header):** Copies the entire grid to the clipboard using the syntax for the LaTeX tabular environment. The first row of the grid is treated as a header row, and is bold faced.

Copy as LaTeX (longtable, With Header): Copies the entire grid to the clipboard using the syntax for the LaTeX longtable environment. The first row of the grid is treated as a header row, it is bold faced and it is set up to repeat on subsequent pages if the longtable is broken between pages.

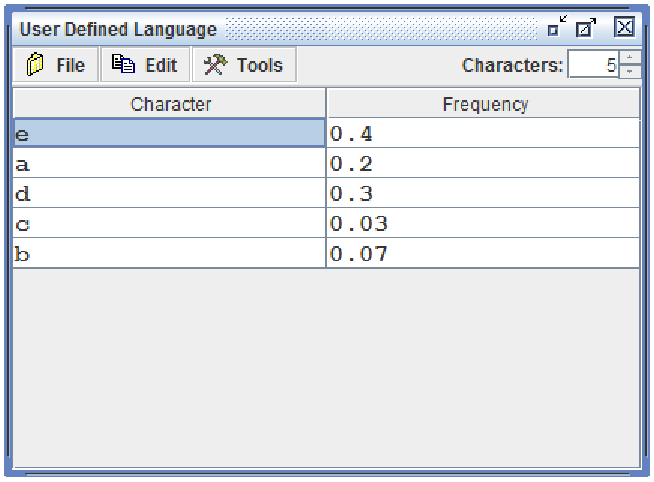
Transpose Grid: This transposes the grid, it makes rows into columns and columns into rows.

Notes

- The menu options are discussed in the help system for this program. The grid can copy and paste to and from a standard spreadsheet and word processor by using the standard keyboard keys (Ctrl+C and Ctrl+V).

3.5.6 User Defined Language Creator

The User Defined Language tool is for creating a language with letter frequencies that can be used in place of English in the program. Not all ciphers lend themselves easily to changing the language but some do. The Monoalphabetic substitution, Vigenère and Hill ciphers allow user defined languages as does the dot product analysis tool. This feature allows the user to experiment with the cryptographic concepts without the aid of a familiar language. The user could also use this tool to add languages to the program, such as French, German, and Spanish.



Characters: 5

Character	Frequency
e	0.4
a	0.2
d	0.3
c	0.03
b	0.07

Figure 3.42: User Defined Language Creator Tool

How to Use the Tool

1. Select the number of characters in the language.

2. Type in a language character in the left column and the corresponding relative frequency of the character in the right hand column.

Options

- The toolbar with the following options.

  File —

    New: Clears the language table.

    Open: Loads a user defined language file into the grid.

    Save As: Saves the current grid contents to a file.

    Print: Prints the current grid to the selected printer.

    Print Preview: Prints the current grid to the print preview display.

  Edit —

    Copy: Copies the grid to the clipboard.

    Paste: Pastes the contents of the clipboard into the grid.

    Copy as LaTeX (tabular): Copies the entire grid to the clipboard using the syntax for the LaTeX tabular environment.

    Copy as LaTeX (longtable): Copies the entire grid to the clipboard using the syntax for the LaTeX longtable environment.

  Tools —

    Convert Characters to Uppercase: Changes all of the characters in the left hand column to uppercase.

    Sort by Character: Sorts the table using the character column of the table and alphabetical order.

    Sort Ascending by Frequency: Sorts the table, from lowest to highest, using the frequency column of the table.

    Sort Descending by Frequency: Sorts the table, from highest to lowest, using the frequency column of the table.

    Check if Language is Valid: Checks to see if the current contents of the grid represent a valid language. For the language to be valid there, each character must be a single character, the character   may not be used, there can be no duplications of letters, and the frequencies must all be numbers.

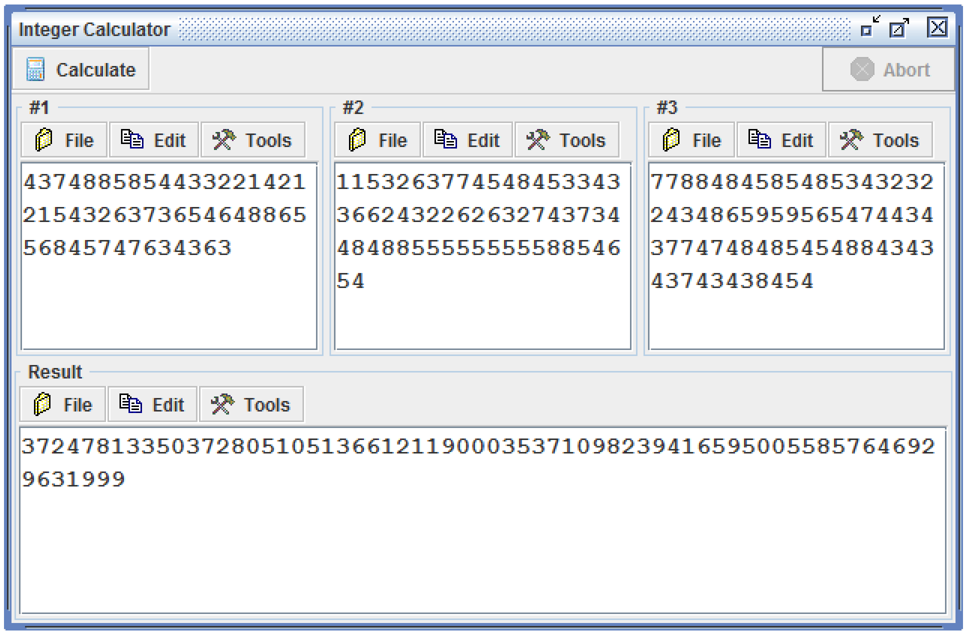
Notes

- In a valid language, each character must be a single character, the character   may not be used, there can be no duplications of letters, and the frequencies must all be numbers.
- If the frequencies do not add up to one, the tools will adjust the frequencies to add to one. If all the frequencies are 0, the program will adjust them to be equal.

3.6 Calculators

3.6.1 Integer Calculator

The Integer Calculator is a simple infinite precision integer arithmetic tool.



```
#1
43748858544332214212154326373654648865
56845747634363

#2
11532637745484533433662432262632743734
484885555555588546
54

#3
77884845854853432322434865959565474434
377474848545488434343743438454

Result
372478133503728051051366121190003537109823941659500558576469
29631999
```

Figure 3.43: Integer Calculator

How to Use the Tool

1. Input the numbers into the three input boxes, #1, #2, and #3,
2. Select the operation from the Calculate menu at the top of the window.

Calculate Menu Options

Arithmetic —

#1 + #2 Adds the numbers from input box #1 and #2.

#1 - #2 Subtracts the number in input box #2 from #1.

#1 * #2 Multiplies the numbers from input box #1 and #2.

#1 ^ #2 Raises the number from input box #1 to the power of #2. Since this is not a modular operation if the program suspects that the calculation will be too large for computation, it will display a warning.

#1 / #2 Divides the number from input box #1 by #2. This is an integer operation, so 4/3 will result in 1.

Rem(#1, #2) Finds the remainder when the number from input box #1 is divided by #2.

Modular Arithmetic —

#1 + #2 Mod (#3) Adds the numbers from input box #1 and #2 modulo #3.

#1 - #2 Mod (#3) Subtracts the number in input box #2 from #1 modulo #3.

#1 * #2 Mod (#3) Multiplies the numbers from input box #1 and #2 modulo #3.

#1 ˆ #2 Mod (#3) Raises the number from input box #1 to the power of #2 modulo #3.

#1 / #2 Mod (#3) Divides the number from input box #1 by #2 modulo #3. If input #2 is not invertible modulo #3 the program will display an error.

#1 Mod (#3) Takes input #1 modulo #3.

#2 Mod (#3) Takes input #2 modulo #3.

GCD —

GCD These commands find the GCD of the numbers listed in the menu option.

GCD of List These option will calculate the GCD of the list of elements in the selected input box. A list in this calculator is a set of numbers separated by commas. For example, the list 12, 4, 16, 200 will return 4 as its GCD.

Factorial —

Factorial Calculates the factorial of the number in the selected input box. If the program suspects that the result is too large it will display a warning. Also if the input is larger than 2147483647, the program will not do the operation.

Double Factorial Calculates the double factorial of the number in the selected input box. If the program suspects that the result is too large it will display a warning. Also if the input is larger than 2147483647, the program will not do the operation.

Square Root —

Square Root Calculates the square root of the number in the selected input box. The output is a decimal number.

Floor of Square Root Calculates the floor of the square root of the number in the selected input box. The output is an integer.

Chinese Remainder Theorem — The Chinese Remainder Theorem requires two lists, one of residues and the other of moduli. Lists in this program are simply numbers separated by commas. For example, 23, 45, 67 is a list of three integers. So for the Chinese Remainder Theorem if the residue list is 1, 2, 3 and the modulus list is 5, 7, 11 then the program will compute $x \pmod{5 \cdot 7 \cdot 11}$ that satisfies $x = 1 \pmod{5}$, $x = 2 \pmod{7}$, $x = 3 \pmod{11}$.

Jacobi Symbol — Finds the Jacobi Symbol of a over b, that is, $\left(\frac{a}{b}\right)$, where a and b are the numbers in the chosen input boxes.

Primes —

Is Prime Tests if the number in the selected input box is prime or composite. If the result is a probable prime, the probability that the number is composite is less than 2^{-100}.

Next Prime Calculates the next probable prime number greater than the one in the selected input box. The probability that the number is composite is less than 2^{-100}.

Previous Prime Calculates the next probable prime number less than the one in the selected input box. The probability that the number is composite is less than 2^{-100}.

Semiprime Calculates the semiprime number composed of the next probable primes of the two selected input boxes.

Number of Primes Returns the number of prime numbers less than or equal to the selected input box. The input can be at most 2,000,000,000 for this operation.

Nth Prime Returns the n^{th} prime number. The input can be at most 100,000,000 for this operation.

Totient — Returns the Euler Totient (Euler Phi) of the number in the selected input box. The calculation of the totient requires the factorization of the number and hence can be lengthy operations.

Primitive Root —

Primitive Root Calculates the smallest primitive root modulo the number in the selected input box. Calculation of a primitive root requires the factorization of the totient of the number and hence can be lengthy operations.

Is Primitive Root Checks if the selected input box number is a primitive root modulo the number in the other selected input box. Verification of a primitive root requires the factorization of the totient of the number and hence can be lengthy operations.

Factor — Returns the factorization of the number in the selected input box. Factorization of a number can be a lengthy operation.

Discrete Logarithm —

Pohlig-Hellman/Pollard Rho This option solves the discrete logarithm problem using the Pohlig-Hellman reduction with the Pollard Rho method. Discrete logarithm problems can be lengthy operations.

Index Calculus This option solves the discrete logarithm problem using the index calculus method with the selected prime base size. Discrete logarithm problems can be lengthy operations.

Index Calculus (Prime Base Size = n) This option solves the discrete logarithm problem using the index calculus method with a prime base size that is input by the user. Discrete logarithm problems can be lengthy operations.

Evaluate — Evaluates the numeric expression in the selected input box. The syntax for these expressions is given in the below subsection.

Options

- Several options in the calculate menu require lengthy derivations and in these cases the calculation will be done in its own thread and the Abort option will be available to cancel the operation if desired.

Notes

- The Chinese Remainder Theorem requires two lists, one of residues and the other of moduli. Lists in this program are simply numbers separated by commas. For example, 23, 45, 67 is a list of three integers. So for the Chinese Remainder Theorem if the residue list is 1, 2, 3 and the modulus list is 5, 7, 11 then the program will compute $x \pmod{5 \cdot 7 \cdot 11}$ that satisfies $x = 1 \pmod 5$, $x = 2 \pmod 7$, and $x = 3 \pmod{11}$.
- The GCD calculations that operate on lists require a list of numbers separated by commas. For example, if 225, 25, 55 is in an input box then the GCD on that list will return 5.
- Totients, primitive roots and factoring all require the factoring of a number and hence can be lengthy operations.
- The calculator also has the ability to evaluate simple algebraic integer expressions. The syntax of the expressions is discussed below. To evaluate the expression of any of the three cells simply select the evaluate option.

Expression Syntax

Arithmetic operations use the standard characters (+, −, ⋆, / and ˆ) and juxtaposition is not supported. So for example, the expression 2 3 would cause a syntax error but 2 * 3 would return 6. Grouping is done with parentheses (). The modulus operator % is also available, so 23 % 7 would return 2. Note that division is integer division, so 23/7 would return 3.

The calculator also accepts a few functions, discussed below. The system is case insensitive, so nextprime, Nextprime, NextPrime, or NeXtpRIme all return the next prime.

factorial(n) returns the factorial of n, that is, $n!$.

doublefactorial(n) returns the double factorial of n, that is, $n!!$.

pi(n) returns the number of primes less than or equal to n. The value of n can be at most 2,000,000,000.

nextprime(n) returns the next probable prime greater than n.

previousprime(n) returns the next probable prime less than n.

semiprime(n, m) returns the semiprime created from the next probable prime greater than n and the next probable prime greater than m.

totient(n) returns the number of integers less then n that are relatively prime to n.

eulerphi(n) returns the number of integers less then n that are relatively prime to n.

primitiveroot(n) returns the smallest positive primitive root of n. Here, n must be prime.

mod(a, n) returns $a \pmod{n}$.

gcd(a, b) returns the greatest common divisor of a and b.

gcd(a, b, c, ...) returns the greatest common divisor of a, b, c, $\ldots$.

jacobi(a, b) returns the jacobi symbol of a over b. Here, b most be positive and odd.

powermod(a, e, n) returns $a^e \pmod{n}$.

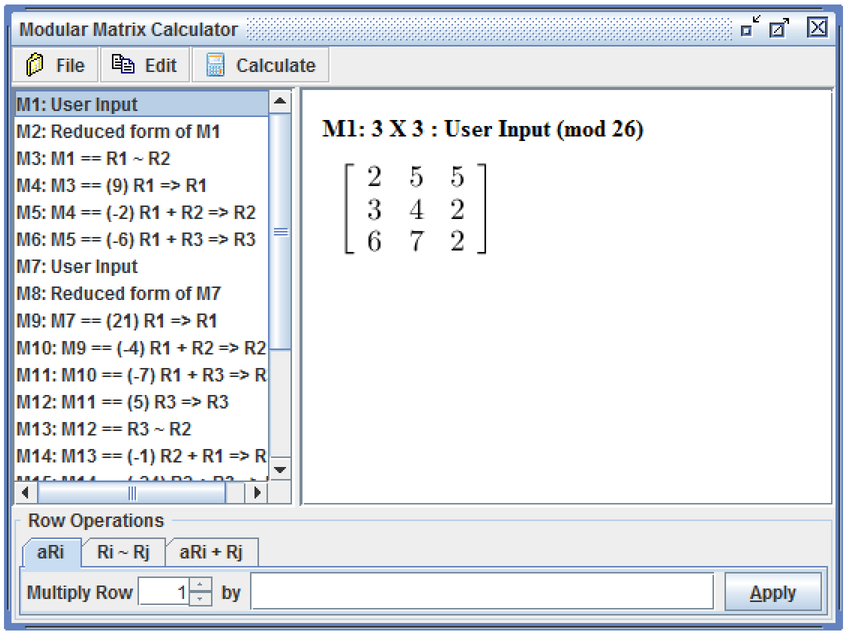
chrem(a1, n1, a2, n2, ... ar, nr) returns the value x that satisfies the congruences $x = a_1 \pmod{n}_1$, $x = a_2 \pmod{n}_2$, $\ldots$, $x = a_r \pmod{n}_r$. The list of residues and moduli may be as long as the user chooses.

chineseremainder(a1, n1, a2, n2, ... ar, nr) returns the value x that satisfies the congruences $x = a_1 \pmod{n}_1$, $x = a_2 \pmod{n}_2$, $\ldots$, $x = a_r \pmod{n}_r$. The list of residues and moduli may be as long as the user chooses.

3.6.2 Modular Matrix Calculator

The Modular Matrix Calculator is a simple matrix manipulator and arithmetic tool. Each matrix has an associated modulus for all calculations. Reduction, inverses and matrix arithmetic is done over the associated modulus. Two matrices can only be added, subtracted or multiplied if they have the same modulus, and appropriate sizes.

The list on the left is the current set of matrices that are in the workspace. The panel on the right is the currently selected matrix from the list. The panel at the bottom is a row operation panel that gives the user a quick interface to do row operations on the currently selected matrix. This panel is hidden when the calculator is started but can toggled on and off from the calculate menu.



M1: User Input
M2: Reduced form of M1
M3: M1 == R1 ~ R2
M4: M3 == (9) R1 => R1
M5: M4 == (-2) R1 + R2 => R2
M6: M5 == (-6) R1 + R3 => R3
M7: User Input
M8: Reduced form of M7
M9: M7 == (21) R1 => R1
M10: M9 == (-4) R1 + R2 => R2
M11: M10 == (-7) R1 + R3 => R
M12: M11 == (5) R3 => R3
M13: M12 == R3 ~ R2
M14: M13 == (-1) R2 + R1 => R

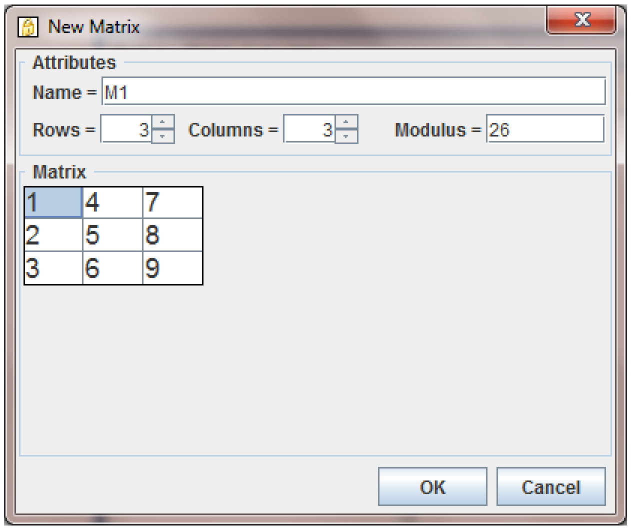
M1: 3 X 3 : User Input (mod 26)

$$\begin{bmatrix} 2 & 5 & 5 \\ 3 & 4 & 2 \\ 6 & 7 & 2 \end{bmatrix}$$

Figure 3.44: Modular Matrix Calculator

How to Use the Tool

1. To input a matrix, select Edit > New Matrix. . . from the menu. At this point the matrix input/edit dialog box will appear.



Name = M1

Rows = 3 Columns = 3 Modulus = 26

1	4	7
2	5	8
3	6	9

Figure 3.45: New Matrix Dialog

The program will select a matrix name of the form M### where the ### is a number that has not been used for another matrix in the workspace. You can change the name but it must be unique to the workspace, no two matrices can share the same name. Then select the size of the matrix, the maximum size for this program is 100 rows and 100 columns. Next input the modulus, the modulus must be an integer but is not restricted in size. Finally, input the matrix entries into the matrix grid and click on the OK button.

At this point the matrix will be loaded into the workspace. The program will mod all the entries by the modulus before loading it into the workspace.

2. Select an operation from the Calculate menu at the top of the window, the options are discussed below.

Menu Options

File —

New: Clears the current workspace.

Open: Opens a workspace file.

Save As: Saves a workspace file.

Save As LaTeX: Saves the contents of the workspace to a LaTeX file.

Print: Prints the current workspace to the selected printer.

Print Preview: Opens the print preview window with the current workspace.

Edit —

New Matrix: Opens the new matrix dialog box allowing the user to input a new matrix.

Edit Matrix: Opens the edit matrix dialog box allowing the user to edit the currently selected matrix. When the user clicks OK, a new matrix will be loaded into the workspace, the original matrix will remain unaltered. This allows the user to make a copy of the current matrix by simply selecting to edit the matrix and then clicking OK. The edit matrix dialog box will also be invoked by double-clicking the matrix name and description in the workspace list on the left.

Copy Matrix: Copies the contents of the current matrix to the clipboard. The copy is done in a tab-delimited format so that it can be pasted into a spreadsheet or into any another grid in this program.

Copy Matrix to LaTeX: Copies the current matrix to a LaTeX array environment.

Copy Matrix to Mathematica Syntax: Copies the current matrix to Mathematica syntax.

Copy Matrix to Maxima Syntax: Copies the current matrix to Maxima syntax.

Copy Workspace to LaTeX: Copies the entire workspace to LaTeX.

Calculate —

Show/Hide Row Operations Panel: This toggles the row operations panel at the bottom of the window. If there is no matrix in the workspace the panel will remain hidden. The row operations panel has three tabs, one for each of the three standard row operations. Once the operation type is selected, fill in the parameters needed and select Apply. At this point a new matrix will be added to the workspace which is the current matrix with the operation applied to it.

Reduce: Reduces the matrix as far as it can. Since all calculations are done over a modulus, the result may not look like a reduced matrix over the real numbers. For example, if there are no invertible elements in a column the program will move to the next column to find any possible reductions.

Add: This will bring up a dialog box allowing the user to select the two matrices to add. The selection is done by matrix name in two drop-down boxes.

Subtract: This will bring up a dialog box allowing the user to select the two matrices to subtract. The selection is done by matrix name in two drop-down boxes.

Negate: Will negate the current matrix, by the matrix modulus.

Multiply: This will bring up a dialog box allowing the user to select the two matrices to multiply. The selection is done by matrix name in two drop-down boxes.

Scalar Multiply: This will bring up a dialog box allowing the user to input the scalar to multiply by. The scalar must be an integer.

Invert: This will invert the current matrix, under the modulus.

Power: This will bring up a dialog box allowing the user to select an integer power between -100 and 100.

Transpose: This will transpose the current matrix.

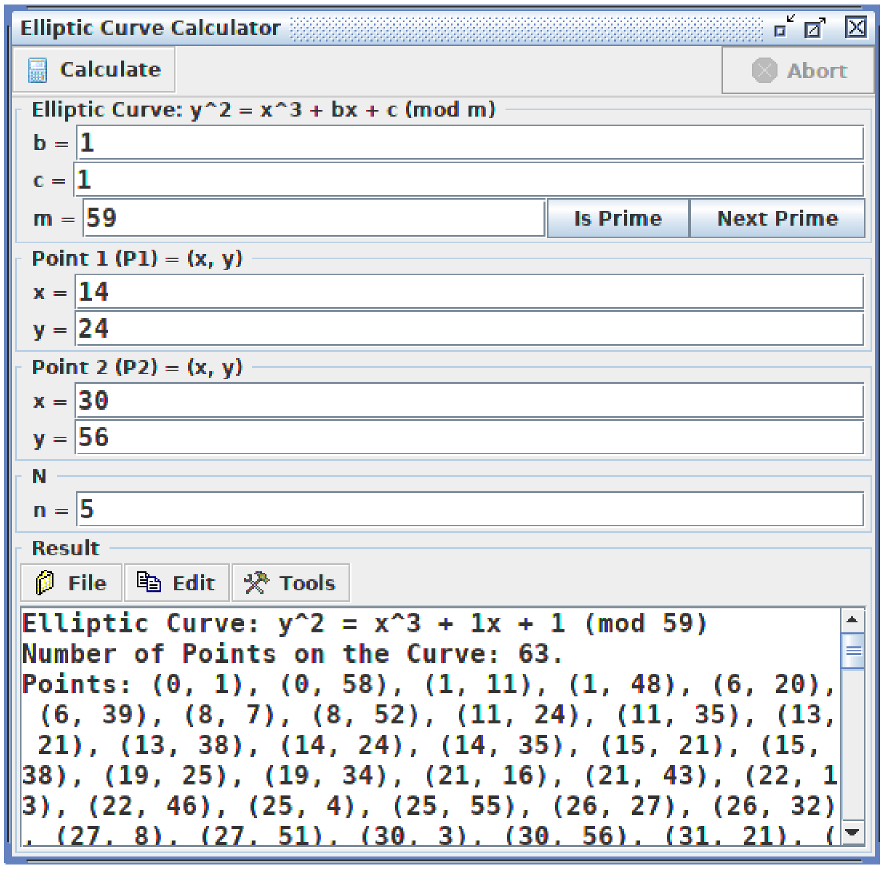
Notes

- When an operation is done on a matrix the original matrix is not altered, instead a new matrix is loaded into the workspace.
- As with any operation in linear algebra, if the matrix sizes are not compatible for the selected operation you will get an error.

3.6.3 Elliptic Curve Calculator

The Elliptic Curve Calculator is a simple tool to aid in some calculations involving points on an elliptic curve. This tool uses the reduced form of an elliptic curve

$$y^2 = x^3 + bx + c \pmod{m}$$



b = 1
c = 1
m = 59

Point 1 (P1): x = 14, y = 24

Point 2 (P2): x = 30, y = 56

n = 5

```
Elliptic Curve: y^2 = x^3 + 1x + 1 (mod 59)
Number of Points on the Curve: 63.
Points: (0, 1), (0, 58), (1, 11), (1, 48), (6, 20),
 (6, 39), (8, 7), (8, 52), (11, 24), (11, 35), (13,
 21), (13, 38), (14, 24), (14, 35), (15, 21), (15,
38), (19, 25), (19, 34), (21, 16), (21, 43), (22, 1
3), (22, 46), (25, 4), (25, 55), (26, 27), (26, 32)
, (27, 8), (27, 51), (30, 3), (30, 56), (31, 21), (
```

Figure 3.46: Modular Matrix Calculator

How to Use the Tool

Input the parameters for the elliptic curve, specifically the linear term, the constant term, and modulus. Input the coordinates for one or two points on the curve and/or a constant n. Select the operation from the Calculate menu at the top of the window.

Menu Options

Calculate —

P1 + P2: Adds the two points P1 and P2 on the curve. Note that this does not check if the points are on the curve.

n * P1: Multiplies n times P1. Note that this does not check if the point P1 is on the curve.

n! * P1: Multiplies $n!$ times P1. Note that this does not check if the point P1 is on the curve.

n * P2: Multiplies n times P2. Note that this does not check if the point P2 is on the curve.

n! * P2: Multiplies $n!$ times P2. Note that this does not check if the point P2 is on the curve.

Is P1 on the Curve?: Determines if the point P1 is on the curve.

Is P2 on the Curve?: Determines if the point P2 is on the curve.

Is m prime?: Determines if m is probably prime or composite.

Order of P1: Determines the order of the point P1. Note that this does not check if the point P1 is on the curve.

Order of P2: Determines the order of the point P2. Note that this does not check if the point P2 is on the curve.

Generate elliptic curve using b, m, and P1: Finds the constant term needed for the point P1 to be on the curve given the values of the linear term and modulus.

Generate elliptic curve using b, m, and P2: Finds the constant term needed for the point P2 to be on the curve given the values of the linear term and modulus.

Generate point on the elliptic curve using x value of P1: Finds a point on the elliptic curve with the same x coordinate as P1, if one exists.

Generate point on the elliptic curve using x value of P2: Finds a point on the elliptic curve with the same x coordinate as P2, if one exists.

Generate point on the elliptic curve using y value of P1: Finds a point on the elliptic curve with the same y coordinate as P1, if one exists.

Generate point on the elliptic curve using y value of P2: Finds a point on the elliptic curve with the same y coordinate as P2, if one exists.

Generate a random point on the elliptic curve: Finds a random point on the elliptic curve.

Generate 10 random points on the elliptic curve: Finds 10 random points on the elliptic curve. Note that there may be repeated points in this list.

Find the number of points on the elliptic curve: Returns the number of points on the elliptic curve.

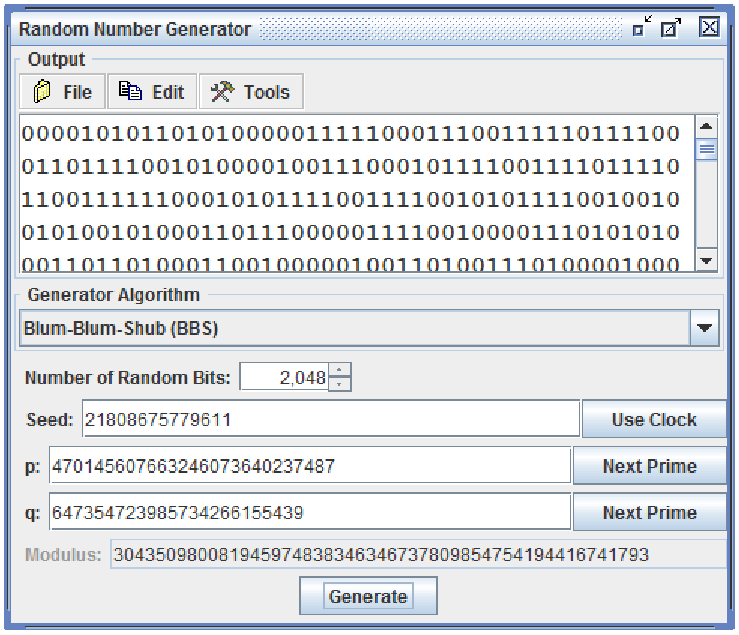
Generate all points on the elliptic curve: Returns a list of all the points on the elliptic curve. A list of Mathematica style points and Maxima style points are returned as well for loading into these computer algebra systems.

Notes

- Several options in the calculate menu require lengthy derivations and in these cases the calculation will be done in its own thread and the Abort option will be available to cancel the operation if desired.

3.6.4 Random Number Generator

The Random Number Generator will create either a list of random numbers using either a linear congruential algorithm or Java's built in Random class, or a stream of random binary digits using the Blum-Blum-Shub algorithm.



```
00001010110101000001111100011100111110111100
01101111001010000100111000101111001111011110
110011111110001010111100111100101011110010010
01010010100011011100000111100100001110101010
001101110100011001000001001101001110100001000
```

Generator Algorithm: Blum-Blum-Shub (BBS)

Number of Random Bits: 2,048

Seed: 21808675779611

p: 470145607663246073640237487

q: 647354723985734266155439

Modulus: 304350980081945974838346346737809854754194416741793

Figure 3.47: Random Number Generator

How to Use the Tool

1. Select the generator algorithm you wish to use, linear congruential, Java's built-in random number generator, or the Blum-Blum-Shub algorithm.

2. Input the number of random numbers (or random bits) you wish to generate.

3. Input the parameters for the method, this will depend on the method chosen.

4. Click the Generate button to generate the numbers or bits.

Options

Linear Congruential: For the Linear Congruential algorithm you will need to supply a seed to the sequence, an adder, multiplier and modulus. For the seed, you can click on the Use Clock button to get a nanosecond representation of the current time.

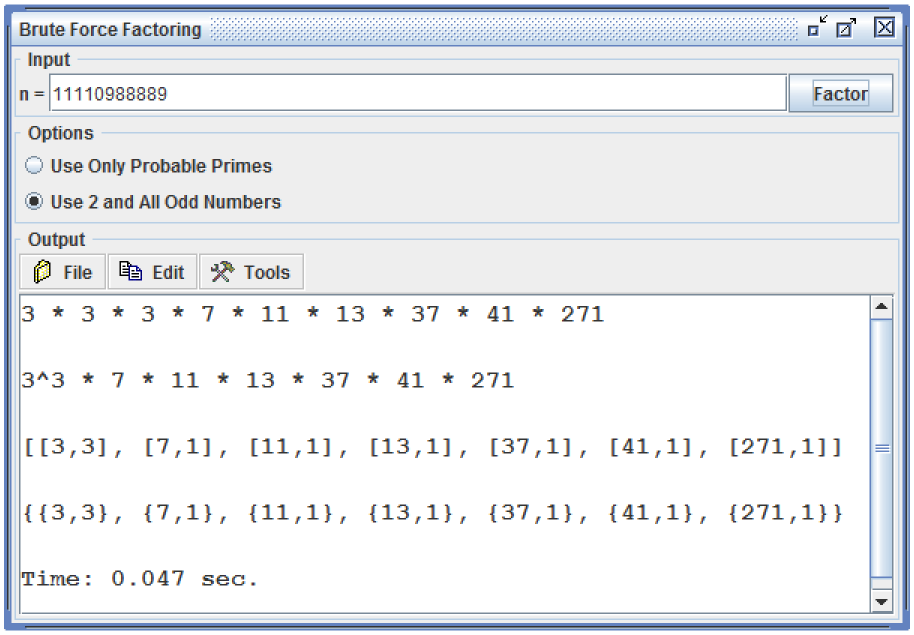
Java's Random Class: The built-in random number generator in Java (the language this program was written in) also uses a linear congruential algorithm but keeps a constant adder and multiplier, so you need only supply the seed and modulus. As with the linear congruential algorithm you can click on the Use Clock button to get a nanosecond representation of the current time for the seed. Also, when using Java's random number generator, the modulus can be at most 9,223,372,036,854,775,807. If you use a modulus larger than this the program will automatically convert it to this maximum.

Blum-Blum-Shub Algorithm: The Blum-Blum-Shub Algorithm is for the generation of random bits. For this algorithm you must supply the seed, again you can use the clock, and two primes p and q. Each of the prime input boxes has an option for generating the next probable prime larger then the number currently in the box. Note that these buttons will find the next prime that is congruent to 3 (mod 4). For this algorithm, the modulus is pq, so you do not need to input it, it will be calculated when you click on the Generate button.

3.7 Factoring Tools

3.7.1 Brute Force Factoring

The Brute Force Factoring tool will factor an integer into its prime factor decomposition using the brute force method of trial division.



n = 11110988889

( ) Use Only Probable Primes
(●) Use 2 and All Odd Numbers

```
3 * 3 * 3 * 7 * 11 * 13 * 37 * 41 * 271

3^3 * 7 * 11 * 13 * 37 * 41 * 271

[[3,3], [7,1], [11,1], [13,1], [37,1], [41,1], [271,1]]

{{3,3}, {7,1}, {11,1}, {13,1}, {37,1}, {41,1}, {271,1}}

Time: 0.047 sec.
```

Figure 3.48: Brute Force Factoring Tool

How to Use the Tool

1. Input the number to be factored into the Input box.

2. Select the type of trial division,
   - Use Only Probable Primes — will calculate the next probable prime for trial division.
   - Use 2 and All Odd Numbers — will use 2, 3, 5, 7, 9, 11, 13, 15, 17, 19, . . . for the trial division.
3. Click the Factor button.

Options

Use Only Probable Primes: This will calculate the next probable prime for trial division.

Use 2 and All Odd Numbers: This will use 2, 3, 5, 7, 9, 11, 13, 15, 17, 19, ... for the trial division.

Status Bar Information

Since factoring methods could take some time to complete, there is status bar information that shows you if the method is making progress on factoring the number. On the far right is the elapsed time and on the left is a display of the current trial divisor being tested.

Notes

- When a divisor is found the program will take the quotient and apply brute force factoring to it.
- After a new quotient is calculated it is checked to be a probable prime, if it is, the process is finished and the program will report the results.
- The output of the factorization is in 4 forms. The first is an expanded multiplication, the second is a product expression with powers, the third is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger command. For example, when factoring 11110988889 the output is,

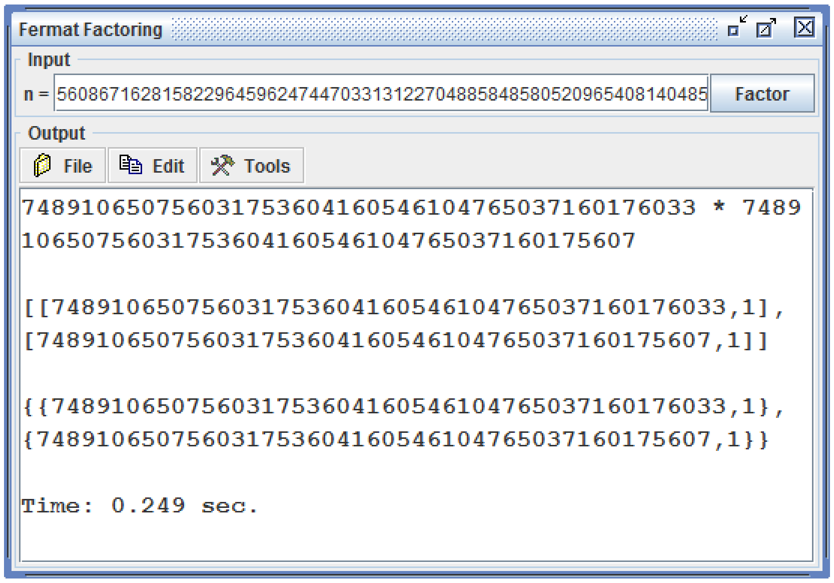
```
3 * 3 * 3 * 7 * 11 * 13 * 37 * 41 * 271
3^3 * 7 * 11 * 13 * 37 * 41 * 271
[[3,3], [7,1], [11,1], [13,1], [37,1], [41,1], [271,1]]
{{3,3}, {7,1}, {11,1}, {13,1}, {37,1}, {41,1}, {271,1}}
Time: 0.047 sec.
```

- At the bottom of the output box is the amount of time used for the factoring operation.
- Factoring integers can be a lengthy process, the factoring operation is done in a separate thread so that the user can use other facilities of the program while a factoring operation is being completed.

- When the factoring operation is invoked, the Factor button will be changed to an Abort button. To end the process simply click on the Abort button.

3.7.2 Fermat Factoring

The Fermat Factoring algorithm simply computes $n + i^2$, for $i = 1, 2, 3, \ldots$ until the result is a perfect square. Then $n + i^2 = d^2$, so $n = d^2 - i^2 = (d + i)(d - i)$.



```
n = 56086716281582296459624744703313122704885848580520965408140485
```

```
748910650756031753604160546104765037160176033 * 7489
10650756031753604160546104765037160175607

[[748910650756031753604160546104765037160176033,1],
[748910650756031753604160546104765037160175607,1]]

{{748910650756031753604160546104765037160176033,1},
{748910650756031753604160546104765037160175607,1}}

Time: 0.249 sec.
```

Figure 3.49: Fermat Factoring Tool

How to Use the Tool

1. Input the number to be factored into the Input box.
2. Click the Factor button.

Status Bar Information

Since factoring methods could take some time to complete, there is status bar information that shows you if the method is making progress on factoring the number. On the far right is the elapsed time and on the left is a display of the current difference being tested.

Notes

- The method does not find a complete prime factorization of n, even if it finds a factor. So the factors in the output could still be composite.

- The output of the factorization is in 3 forms. The first is as a product, the second is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger command. For example, when factoring 5371384963127189 the output is,

```
73290023 * 73289443
[[73290023,1], [73289443,1]]
{{73290023,1}, {73289443,1}}
Time: 0.047 sec.
```

- At the bottom of the output box is the amount of time used for the factoring operation.
- Factoring integers can be a lengthy process, the factoring operation is done in a separate thread so that the user can use other facilities of the program while a factoring operation is being completed.
- When the factoring operation is invoked, the Factor button will be changed to an Abort button. To end the process simply click on the Abort button.

3.7.3 Pollard $p-1$ Factoring

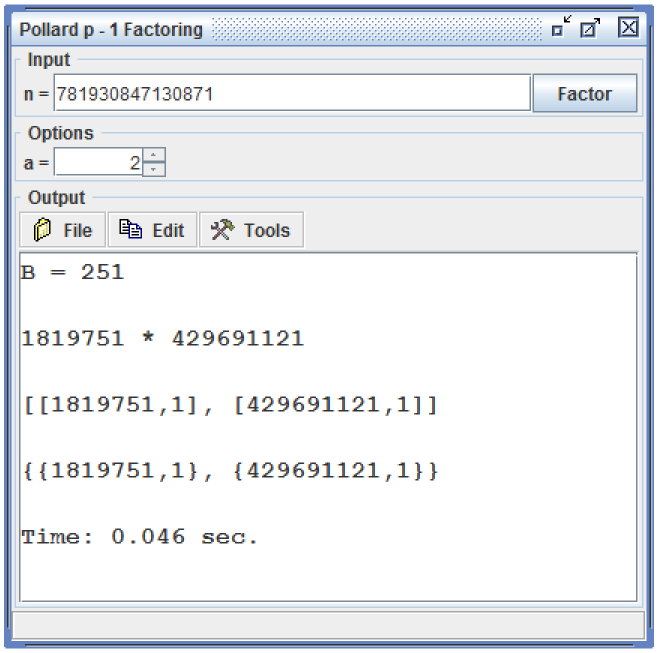
The Pollard $p-1$ Factoring algorithm simply computes $b = a^{B!} \pmod{n}$ and then finds $d = \gcd(b-1, n)$. If $1 < d < n$ then we have a non-trivial factor of n. This tool makes a slight alteration of the algorithm by iterating until a factorization if found or the user aborts the calculation. At each iteration, B is incremented by one, b is updated by computing $\left(a^{(B-1)!}\right)^B \pmod{n}$ and then finally computing d. The iteration stops when a factor is found.

How to Use the Tool

1. Input the number to be factored into the Input box.
2. Input the base, a, to be used in the calculation of b.
3. Click the Factor button. If the process is successful, the output box will contain the smallest value of B that succeeded in factoring n, the factorization $n = d \cdot q$ in three forms, and the amount of time the process took to find a factor of n.

Status Bar Information

Since factoring methods could take some time to complete, there is status bar information that shows you if the method is making progress on factoring the number. On the far right is the elapsed time and on the left is a display of the current iteration.



```
B = 251

1819751 * 429691121

[[1819751,1], [429691121,1]]

{{1819751,1}, {429691121,1}}

Time: 0.046 sec.
```

Figure 3.50: Pollard $p - 1$ Factoring Tool

Notes

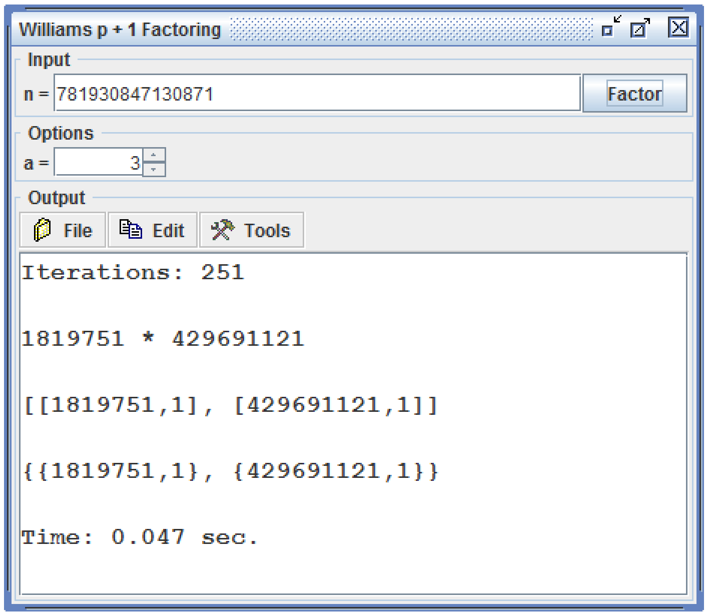
- The method does not find a complete prime factorization of n, even if it finds a factor. So the factors in the output could still be composite.
- The range for the base, a, is 2 to 1,000,000.
- The output of the factorization is in 3 forms. The first is as a product, the second is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger command. For example, when factoring 781930847130871 the output is,

```
B = 251
1819751 * 429691121
[[1819751,1], [429691121,1]]
{{1819751,1}, {429691121,1}}
Time: 0.046 sec.
```

- At the bottom of the output box is the amount of time used for the factoring operation.
- Factoring integers can be a lengthy process, the factoring operation is done in a separate thread so that the user can use other facilities of the program while a factoring operation is being completed.
- When the factoring operation is invoked, the Factor button will be changed to an Abort button. To end the process simply click on the Abort button.

3.7.4 Williams P + 1 Factoring

The Williams $p+1$ Factoring algorithm simply computes the sequence V_i defined as, $V_0 = 2$, $V_1 = a$, and $V_i = a \cdot V_{i-1} - V_{i-2} \pmod{n}$. For each $V_{k!}$, we compute $d = \gcd(V_{k!} - 2, n)$, if $1 < d < n$ we have found a non-trivial factor of n.



n = 781930847130871

a = 3

```
Iterations: 251

1819751 * 429691121

[[1819751,1], [429691121,1]]

{{1819751,1}, {429691121,1}}

Time: 0.047 sec.
```

Figure 3.51: Williams $p+1$ Factoring Tool

How to Use the Tool

1. Input the number to be factored into the Input box.
2. Input the starting value of the sequence, a.
3. Click the Factor button.

Status Bar Information

Since factoring methods could take some time to complete, there is status bar information that shows you if the method is making progress on factoring the number. On the far right is the elapsed time and on the left is a display of the current iteration.

Notes

- The method does not find a complete prime factorization of n, even if it finds a factor. So the factors in the output could still be composite.
- The range for the base, a, is 3 to 1,000,000.

- The output of the factorization is in 3 forms. The first is as a product, the second is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger command. For example, when factoring 781930847130871 the output is,

```
Iterations: 251
1819751 * 429691121
[[1819751,1], [429691121,1]]
{{1819751,1}, {429691121,1}}
Time: 0.266 sec.
```

- At the bottom of the output box is the amount of time used for the factoring operation.
- Factoring integers can be a lengthy process, the factoring operation is done in a separate thread so that the user can use other facilities of the program while a factoring operation is being completed.
- When the factoring operation is invoked, the Factor button will be changed to an Abort button. To end the process simply click on the Abort button.
- The implementation of this algorithm uses an iterative building scheme to calculate $V_{k!}$, so the output of 251 iterations means that the program found a non-trivial factor at $V_{251!}$.

3.7.5 Pollard Rho Factoring

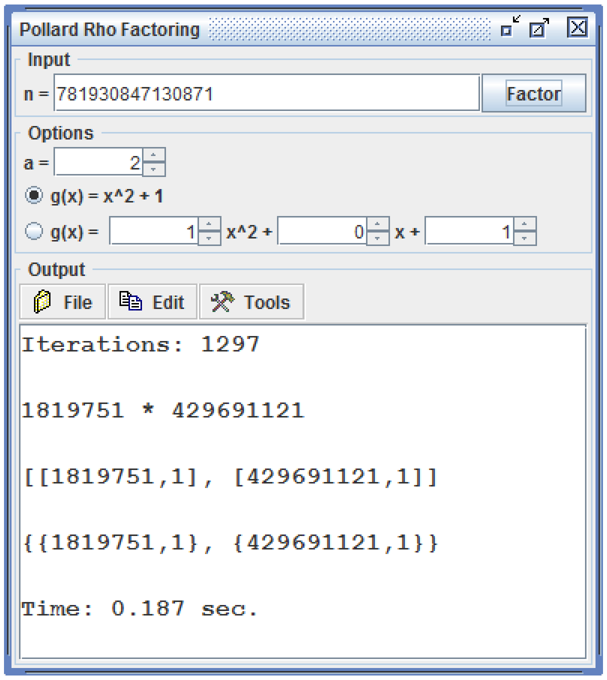
The Pollard Rho Factoring algorithm simply computes two sequences, $\{x_i\}$ and $\{y_i\}$ defined as $x_0 = y_0 = a$, $x_i = g(x_{i-1}))$ and $y_i = g(g(y_{i-1}))$, where $g(x) = x^2 + 1 \pmod{n}$ or $g(x) = a \cdot x^2 + b \cdot x + c \pmod{n}$, depending on your selection of options. For each i, we compute $d = \gcd(|x_i - y_i|, n)$, if $1 < d < n$ we have found a non-trivial factor of n. If, on the other hand, $d = n$ the method fails.

How to Use the Tool

1. Input the number to be factored into the Input box.
2. Input the starting value of the sequences, a.
3. Click the Factor button.

Options

- The base a is the starting value of both the x and y sequences, the range for the base is 2 to 1,000,000.



n = 781930847130871

a = 2

[✓ g(x) = x^2 + 1]

g(x) = 1 x^2 + 0 x + 1

```
Iterations: 1297

1819751 * 429691121

[[1819751,1], [429691121,1]]

{{1819751,1}, {429691121,1}}

Time: 0.187 sec.
```

Figure 3.52: Pollard Rho Factoring Tool

- The program allows you to select between using $g(x) = x^2 + 1 \pmod{n}$ or $g(x) = a \cdot x^2 + b \cdot x + c \pmod{n}$ as the sequence generating function. With the more general quadratic function you may select the coefficients of the terms to be any integer between $-1,000,000$ to $1,000,000$.

Status Bar Information

Since factoring methods could take some time to complete, there is status bar information that shows you if the method is making progress on factoring the number. On the far right is the elapsed time and on the left is a display of the current iteration.

Notes

- The method does not find a complete prime factorization of n, even if it finds a factor. So the factors in the output could still be composite.
- The range for the base, a, is 2 to 1,000,000.
- The each of the $g(x)$ coefficients range from $-1,000,000$ to $1,000,000$.
- The output of the factorization is in 3 forms. The first is as a product, the second is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger command. For example, when factoring 781930847130871 the output is,

```
Iterations: 1297
1819751 * 429691121
[[1819751,1], [429691121,1]]
{{1819751,1}, {429691121,1}}
Time: 0.046 sec.
```

- At the bottom of the output box is the amount of time used for the factoring operation.
- The iteration count is for each iteration of the computation of the next x, next y and the GCD of the difference with n. Hence each iteration consists of three evaluations of g and a GCD.
- Factoring integers can be a lengthy process, the factoring operation is done in a separate thread so that the user can use other facilities of the program while a factoring operation is being completed.
- When the factoring operation is invoked, the Factor button will be changed to an Abort button. To end the process simply click on the Abort button.

3.7.6 Brent's Method Factoring

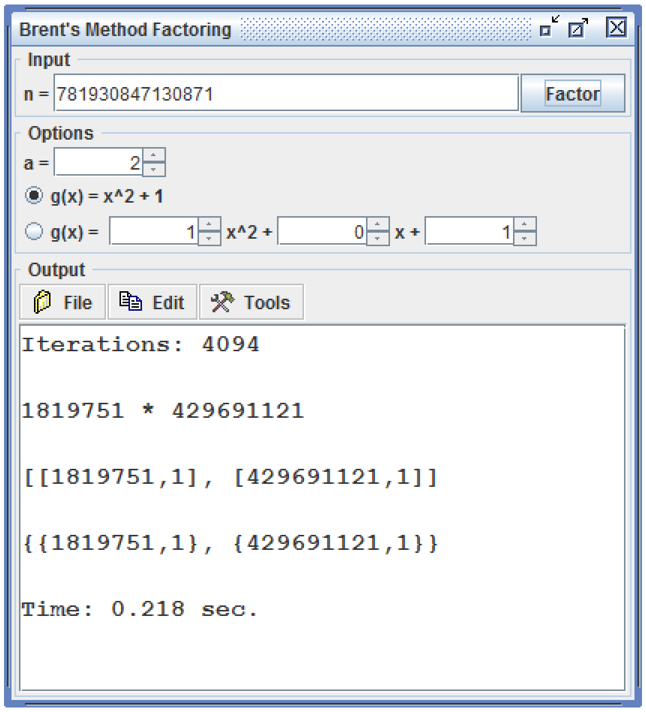
The Brent's Method Factoring algorithm simply computes the sequence, $\{x_i\}$ defined as $x_0 = a$, $x_i = g(x_{i-1})$, where $g(x) = x^2 + 1 \pmod{n}$ or $g(x) = a \cdot x^2 + b \cdot x + c \pmod{n}$, depending on your selection of options. For each i, we compute $d = \gcd(|x_i - x_j|, n)$, where j is the last subscript that is a power of 2, if $1 < d < n$ we have found a non-trivial factor of n. If, on the other hand, $d = n$, we backtrack through the last power of two subsequence. Here we will either find a non-trivial factor or the method will fail. The algorithm was coded directly from the algorithm P_2'' given on pages 182–183 of Brent's original 1980 paper.

How to Use the Tool

1. Input the number to be factored into the Input box.
2. Input the starting value of the sequences, a.
3. Click the Factor button.

Options

- The base a is the starting value of the x sequence, the range for the base is 2 to 1,000,000.
- The program allows you to select between using $g(x) = x^2 + 1 \pmod{n}$ or $g(x) = a \cdot x^2 + b \cdot x + c \pmod{n}$, as the sequence generating function. With the more general quadratic function you may select the coefficients of the terms to be any integer between $-1,000,000$ to $1,000,000$.



n = 781930847130871

a = 2

[✓ g(x) = x^2 + 1]

g(x) = 1 x^2 + 0 x + 1

```
Iterations: 4094

1819751 * 429691121

[[1819751,1], [429691121,1]]

{{1819751,1}, {429691121,1}}

Time: 0.218 sec.
```

Figure 3.53: Brent's Method Factoring Tool

Status Bar Information

Since factoring methods could take some time to complete, there is status bar information that shows you if the method is making progress on factoring the number. On the far right is the elapsed time and on the left is a display of the current iteration.

Notes

- The method does not find a complete prime factorization of n, even if it finds a factor. So the factors in the output could still be composite.
- The range for the base, a, is 2 to 1,000,000.
- The each of the $g(x)$ coefficients range from $-1,000,000$ to $1,000,000$.
- The output of the factorization is in 3 forms. The first is as a product, the second is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger command. For example, when factoring 781930847130871 the output is,

```
Iterations: 1297
1819751 * 429691121
[[1819751,1], [429691121,1]]
{{1819751,1}, {429691121,1}}
```

```
Time: 0.046 sec.
```

- At the bottom of the output box is the amount of time used for the factoring operation.
- The iteration count is for each evaluation of the $g(x)$ function. The algorithm also uses Brent's progressive reduction for calculating the GCD, so each iteration does not include a complete GCD calculation.
- Factoring integers can be a lengthy process, the factoring operation is done in a separate thread so that the user can use other facilities of the program while a factoring operation is being completed.
- When the factoring operation is invoked, the Factor button will be changed to an Abort button. To end the process simply click on the Abort button.

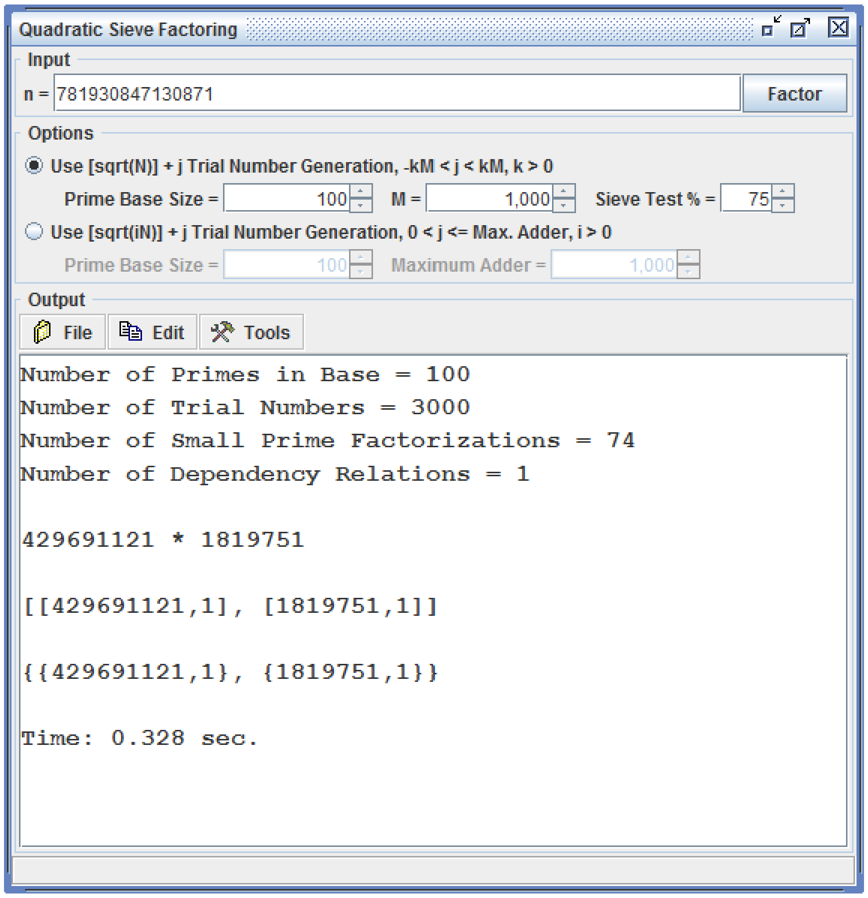
3.7.7 Quadratic Sieve Factoring

The Quadratic Sieve Factoring algorithm attempts to find two numbers x and y with $1 < \gcd(x-y, n) < n$. This is done by finding a set of values x such that $x^2 \pmod{n}$ factors completely using only small prime numbers (the set of small primes is called the Prime Base). Then combinations of these are then multiplied together to get an expression of the form $x^2 \equiv y^2 \pmod{n}$. If x is not congruent to y or $-y$ modulo n the $\gcd(x-y, n)$ will produce a non-trivial factor of n. As a small example, say we have four primes in our prime base, p, q, r and s. Say we find three numbers a, b, and c with $a^2 = p^3rs^2$, $b^2 = pq^3rs^5$, $c^2 = q^7s$ all modulo n. Then $(abc)^2 = p^4q^{10}r^2s^8 = (p^2q^5rs^4)^2$. We would let $x = abc$ and $y = p^2q^5rs^4$, then if x is not y or $-y$ modulo n we calculate $\gcd(x-y, n)$ to obtain a non-trivial factor.

This tool has two modes to find possible numbers whose squares will produce a small prime factorizations. The first follows the classical quadratic sieve method. A nice description of the algorithm can be found in Robert Silverman's 1987 paper *The Multiple Polynomial Quadratic Sieve* which was published in the journal Mathematics of Computation, Volume 48, Issue 177, pages 329–339.

We will not go into the entire algorithm here, please see the Silverman paper, but we will give a quick outline. The program uses a sieving method to quickly pick out values x that are likely to produce small prime factorizations of the values $(x + \lfloor\sqrt{n}\rfloor)^2 - n$. It then computes $(x + \lfloor\sqrt{n}\rfloor)^2 - n$ and then factors the number, using the brute force method on the primes in the prime base. If the factorization completes, the number and factorization are then taken to the next stage to determine if a dependency, and then a factor, is found.

The second method actually skips the sieving step of the classical algorithm. It considers numbers of the form, $\lfloor\sqrt{in} + j\rfloor$, i is called the multiplier and j is called the adder. When the program runs it starts i and j at one, and increments j by one for each new trial number. When j exceeds the maximum adder number, which is an option in this program, it resets j to 1 and increments i by 1. Each of the numbers are then squared and factored, using the brute force method on the primes in the prime base. If the factorization completes, the number and factorization are then taken to the next stage to determine if a dependency, and then a factor, is found.



n = 781930847130871

(● Use [sqrt(N)] + j Trial Number Generation, -kM < j < kM, k > 0) Prime Base Size = 100, M = 1,000, Sieve Test % = 75

Use [sqrt(iN)] + j Trial Number Generation, 0 < j <= Max. Adder, i > 0 Prime Base Size = 100, Maximum Adder = 1,000

```
Number of Primes in Base = 100
Number of Trial Numbers = 3000
Number of Small Prime Factorizations = 74
Number of Dependency Relations = 1

429691121 * 1819751

[[429691121,1], [1819751,1]]

{{429691121,1}, {1819751,1}}

Time: 0.328 sec.
```

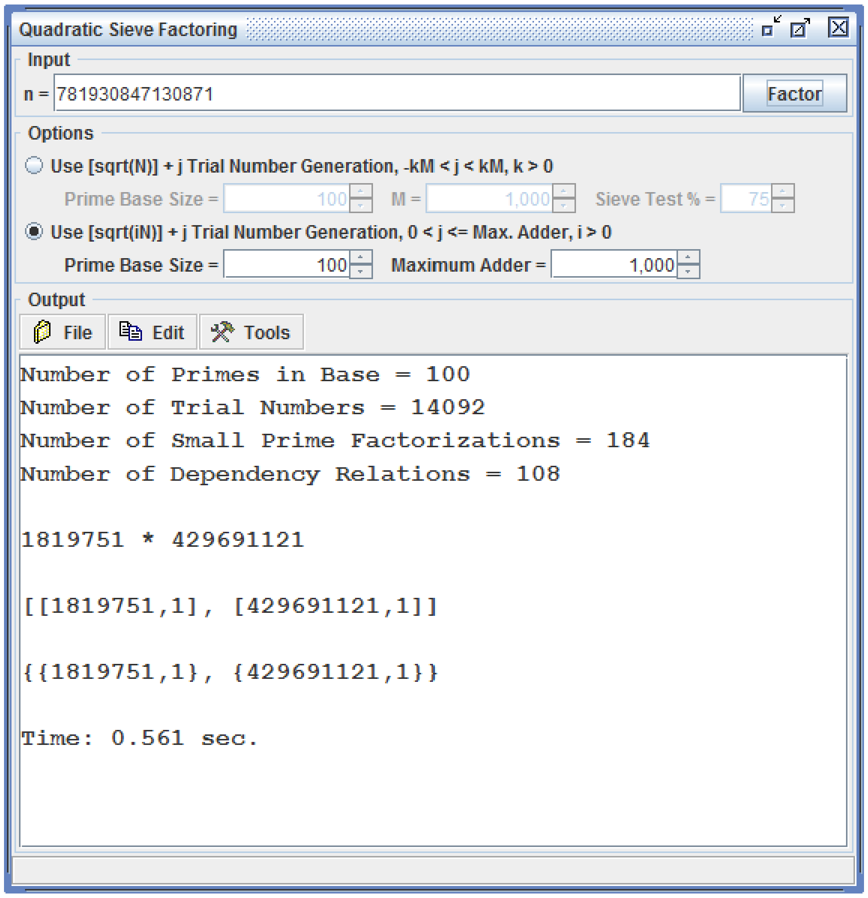
Figure 3.54: Quadratic Sieve Factoring Tool: Classical Sieve

How to Use the Tool

1. Input the number to be factored into the Input box.
2. Select the mode of operation for the sieve. The `[sqrt(N)]+j` is the classical method and the `[sqrt(iN)]+j` is the method that skips the sieving step.
3. Input the bound on the size of the prime base. This number represents the number of primes in the prime base. Also input the other parameters the method needs, these will be discussed below.
4. Click the Factor button.

Options

- In classical mode you need to specify M and the sieve test %. M is the size of the sieving array used to pretest factorizations. The program will start with the interval $[-M, M]$, if more small prime factorizations are needed it will increase the interval to $[-2M, 2M]$, then $[-3M, 3M]$ and so on until a factorization of n is found. Each



n = 781930847130871

Options

○ Use [sqrt(N)] + j Trial Number Generation, -kM < j < kM, k > 0

Prime Base Size = 100 M = 1,000 Sieve Test % = 75

● Use [sqrt(iN)] + j Trial Number Generation, 0 < j <= Max. Adder, i > 0

Prime Base Size = 100 Maximum Adder = 1,000

```
Number of Primes in Base = 100
Number of Trial Numbers = 14092
Number of Small Prime Factorizations = 184
Number of Dependency Relations = 108

1819751 * 429691121

[[1819751,1], [429691121,1]]

{{1819751,1}, {429691121,1}}

Time: 0.561 sec.
```

Figure 3.55: Quadratic Sieve Factoring Tool: Alternative Method

entry position of the sieving array represents a number of the form $(x + \lfloor\sqrt{n}\rfloor)^2 - n$ and the value in that entry represents a scale of the likelihood of the number having a small prime factorization. Theoretically, if an entry has a particular value then the associated number will factor into small primes. In practice, using this number will miss a considerable number of small prime factorizations, so we also test positions that are slightly less than the theoretical bound. So the Sieve Test % is the % of the theoretical bound we will accept as having a possible small prime factorization and send it on to the factorization routine. The larger this number is the more small prime factorizations that you will miss, slowing down the progress toward finding a factor of n and setting this number too low will attempt to factor more numbers that will not have a small prime factorization, again making the program do more work and slowing the progress toward factoring n. Empirically, a setting around 75 to 80 seems to produce the best times on the numbers we tested.

- In the sieve skip mode you need to specify the maximum adder, j, that is used. When j exceeds this value it is reset to 1 and the value of i is increased by 1.

- The prime base is the number of primes used in the small prime factorizations. For the non-classical method this is simply the first so many primes, that is, if the prime base size is set to 100 it will use the first 100 primes. With the classical method, we can skip several primes that will not be of use to us and use all the primes p such that the Jacobi symbol of n and p is 1.

- The prime base size, the adder bound and the sieve array size M are all in the range of 2 to 1,000,000. The Sieve Test % is in the range of 0 to 100.

Status Bar Information

Since factoring methods could take some time to complete, there is status bar information that shows you if the method is making progress on factoring the number. On the far right is the elapsed time and on the left is a display of the number of trial numbers used, the number of small prime factorizations, and the number of mod 2 dependency relations found.

Notes

- The method does not find a complete prime factorization of n, even if it finds a factor. So the factors in the output could still be composite.
- The prime base size and the adder bound are both in the range of 2 to 1,000,000.
- The output of the factorization is in 3 forms. The first is as a product, the second is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger command. For example, when factoring 781930847130871 the output is,

```
Number of Primes in Base = 53
Number of Trial Numbers = 25685
Number of Small Prime Factorizations = 115
Number of Dependency Relations = 74
429691121 * 1819751
[[429691121,1], [1819751,1]]
{{429691121,1}, {1819751,1}}
Time: 0.188 sec.
```

- At the bottom of the output box is the amount of time used for the factoring operation.
- The output also has several benchmark numbers to let you know how much work was done in each area of the algorithm.

  Number of Primes in Base: Tells you the size of the prime base used for the factorizations.

Number of Trial Numbers: Tells you the total number of numbers that the program factored using the prime base. If the number factored within the prime base the factorization is taken to the next stage of the algorithm, if it did not factor within the prime base the next trial number is calculated and the process continues.

Number of Small Prime Factorizations: Tells you the number of factorizations that were completed within the prime base. Each of these are then checked with the current set of independent factorizations to find a mod two dependency among the exponents of the factorizations. If the relation turns out to be dependent, the final stage of the algorithm is done, and if the new factorization is independent of the current ones it is added to the independent set and the process resumes with the next trial number. Dependency relations are calculated using Gaussian elimination on the matrix constructed from the exponent parity vectors of the small prime factorizations.

Number of Dependency Relations: This is the number of modulo 2 exponent dependencies that were found. For each of these, x and y are calculated from the dependency relation, x and y are checked if they are equal or opposite mod n and finally, if applicable, the $\gcd(x - y, n)$ is calculated to find a non-trivial factor of n.

- Factoring integers can be a lengthy process, the factoring operation is done in a separate thread so that the user can use other facilities of the program while a factoring operation is being completed.
- When the factoring operation is invoked, the Factor button will be changed to an Abort button. To end the process simply click on the Abort button.

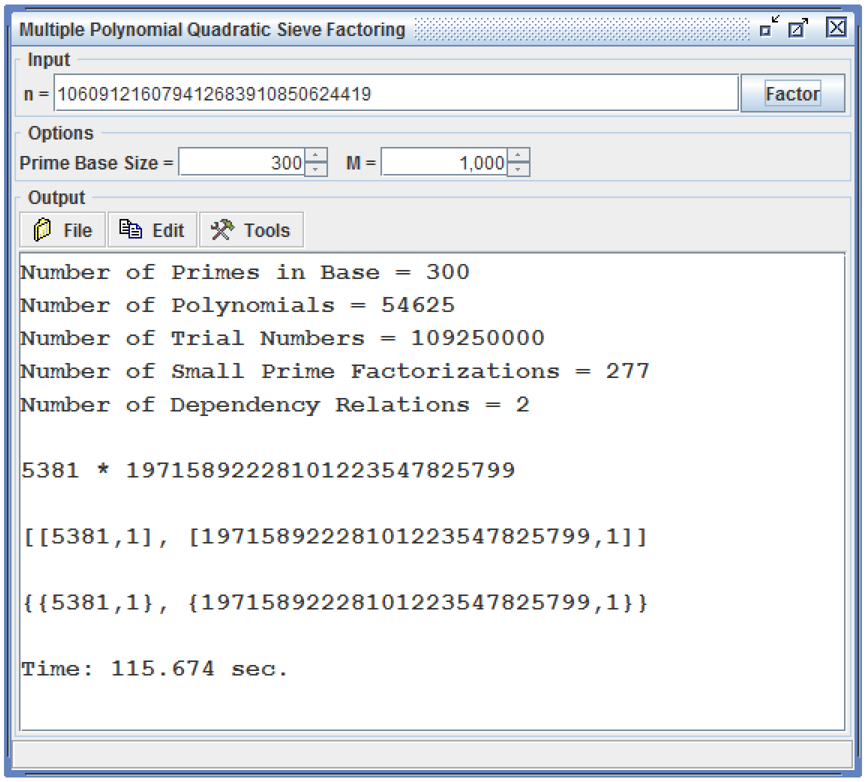
3.7.8 Multiple Polynomial Quadratic Sieve Factoring

The Multiple Polynomial Quadratic Sieve algorithm, like the Quadratic Sieve Factoring algorithm, attempts to find two numbers x and y with $1 < \gcd(x - y, n) < n$. This is done by finding a set of values x such that $x^2 \pmod{n}$ factors completely using only small prime numbers (the set of small primes is called the Prime Base). Then combinations of these are then multiplied together to get an expression of the form $x^2 \equiv y^2 \pmod{n}$. If x is not congruent to y or $-y$ modulo n the $\gcd(x - y, n)$ will produce a non-trivial factor of n. As a small example, say we have four primes in our prime base, p, q, r and s. Say we find three numbers a, b, and c with $a^2 = p^3rs^2$, $b^2 = pq^3rs^5$, $c^2 = q^7s$ all modulo n. Then $(abc)^2 = p^4q^{10}r^2s^8 = (p^2q^5rs^4)^2$. We would let $x = abc$ and $y = p^2q^5rs^4$, then if x is not y or $-y$ modulo n we calculate $\gcd(x - y, n)$ to obtain a non-trivial factor.

A nice description of the algorithm can be found in Robert Silverman's 1987 paper *The Multiple Polynomial Quadratic Sieve* which was published in the journal Mathematics of Computation, Volume 48, Issue 177, pages 329–339.

We will not go into the entire algorithm here, please see the Silverman paper, but we will give a quick outline. The program uses a sieving method to quickly pick out values x

that are likely to produce small prime factorizations of the values $(ax^2 + bx + c)^2$ modulo n. It then computes $(ax^2 + bx + c)^2 \pmod{n}$ and then factors the number, using the brute force method on the primes in the prime base. If the factorization completes, the number and factorization are then taken to the next stage to determine if a dependency, and then a factor, is found.



n = 106091216079412683910850624419

Prime Base Size = 300   M = 1,000

```
Number of Primes in Base = 300
Number of Polynomials = 54625
Number of Trial Numbers = 109250000
Number of Small Prime Factorizations = 277
Number of Dependency Relations = 2

5381 * 19715892228101223547825799

[[5381,1], [19715892228101223547825799,1]]

{{5381,1}, {19715892228101223547825799,1}}

Time: 115.674 sec.
```

Figure 3.56: Multiple Polynomial Quadratic Sieve Factoring Tool

How to Use the Tool

1. Input the number to be factored into the Input box.
2. Select the size of the prime base, this number represents the number of primes in the prime base.
3. Select the length of the sieving interval $[-M, M]$.
4. Click the Factor button.

Options

- M is the size of the sieving array used to pretest factorizations.

- The prime base is the number of primes used in the small prime factorizations. Not all consecutive primes are used here. We can skip several primes that will not be of use to us and use all the primes p such that the Jacobi symbol of n and p is 1.
- The prime base size and the sieve array size M are all in the range of 2 to 1,000,000.

Status Bar Information

Since factoring methods could take some time to complete, there is status bar information that shows you if the method is making progress on factoring the number. On the far right is the elapsed time and on the left is a display of the number of polynomials used, the number of trial numbers used, the number of small prime factorizations, and the number of mod 2 dependency relations found.

Notes

- The method does not find a complete prime factorization of n, even if it finds a factor. So the factors in the output could still be composite.
- The output of the factorization is in 3 forms. The first is as a product, the second is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger command. For example, when factoring 781930847130871 the output is,

```
Number of Primes in Base = 100
Number of Polynomials = 194
Number of Trial Numbers = 388000
Number of Small Prime Factorizations = 90
Number of Dependency Relations = 3
429691121 * 1819751
[[429691121,1], [1819751,1]]
{{429691121,1}, {1819751,1}}
Time: 2.122 sec.
```

- At the bottom of the output box is the amount of time used for the factoring operation.
- The output also has several benchmark numbers to let you know how much work was done in each area of the algorithm.

  Number of Primes in Base: Tells you the size of the prime base used for the factorizations.

  Number of Trial Numbers: Tells you the total number of numbers that the program factored using the prime base. If the number factored within the prime base the factorization is taken to the next stage of the algorithm, if it did not factor within the prime base the next trial number is calculated and the process continues.

Number of Small Prime Factorizations: Tells you the number of factorizations that were completed within the prime base. Each of these are then checked with the current set of independent factorizations to find a mod two dependency among the exponents of the factorizations. If the relation turns out to be dependent, the final stage of the algorithm is done, and if the new factorization is independent of the current ones it is added to the independent set and the process resumes with the next trial number. Dependency relations are calculated using Gaussian elimination on the matrix constructed from the exponent parity vectors of the small prime factorizations.

Number of Dependency Relations: This is the number of modulo 2 exponent dependencies that were found. For each of these, x and y are calculated from the dependency relation, x and y are checked if they are equal or opposite mod n and finally, if applicable, the $\gcd(x - y, n)$ is calculated to find a non-trivial factor of n.

- Factoring integers can be a lengthy process, the factoring operation is done in a separate thread so that the user can use other facilities of the program while a factoring operation is being completed.
- When the factoring operation is invoked, the Factor button will be changed to an Abort button. To end the process simply click on the Abort button.

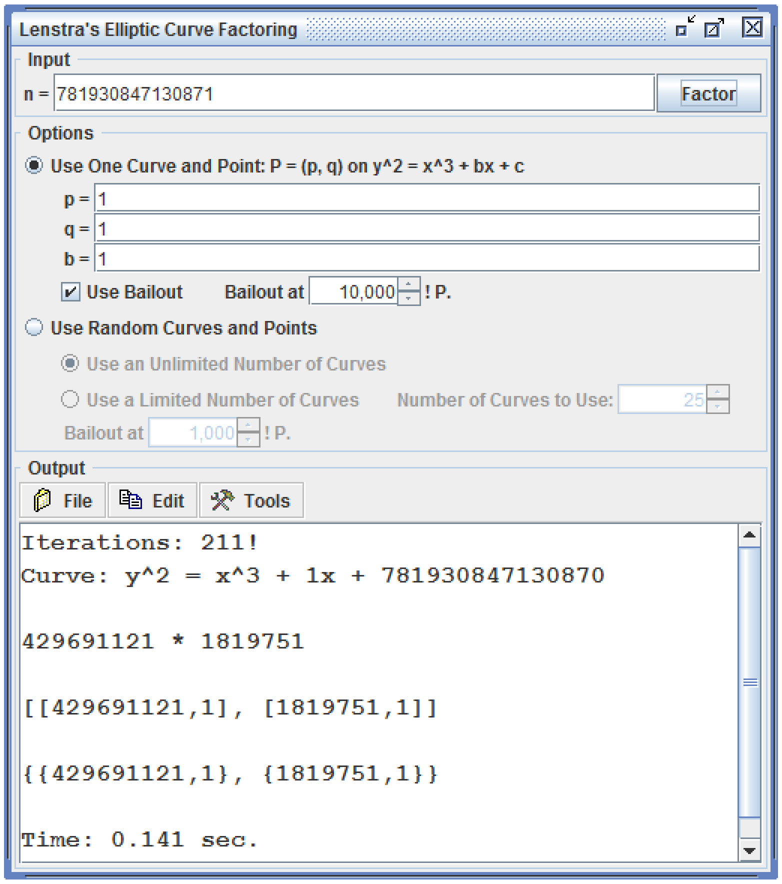
3.7.9 Lenstra's Elliptic Curve Factoring

The Lenstra's Elliptic Curve Factoring algorithm takes an elliptic curve of the form $y^2 = x^3 + bx + c \pmod{n}$ and a point P on the curve, we then calculate $m!P$ for increasingly larger m. During the calculation of the slope, there may be a case where the slope cannot be calculated due to a non-invertible number modulo n. When this occurs, the GCD of this number and n is not 1 and hence could be a non-trivial factor of n. If so the program returns this GCD, g, and $\frac{n}{g}$.

The Lenstra's Elliptic Curve Factoring tool has two modes of operation, the first, is where the user can select a single elliptic curve and point on that curve and run the algorithm with that point and curve. In practice, one usually selects several random elliptic curves but this mode allows the user to experiment with particular curves and points. The second mode, is to allow the computer to select any number of random elliptic curves and run the algorithm on each.

How to Use the Tool

1. Input the number to be factored into the Input box.
2. Select the mode of operation and the parameters for that mode, these will be discussed further below.
3. Click the Factor button.



n = 781930847130871

Use One Curve and Point: P = (p, q) on y^2 = x^3 + bx + c [✓]

p = 1

q = 1

b = 1

Use Bailout [✓] Bailout at 10,000 ! P.

Use Random Curves and Points

Use an Unlimited Number of Curves

Use a Limited Number of Curves Number of Curves to Use: 25

Bailout at 1,000 ! P.

```
Iterations: 211!
Curve: y^2 = x^3 + 1x + 781930847130870

429691121 * 1819751

[[429691121,1], [1819751,1]]

{{429691121,1}, {1819751,1}}

Time: 0.141 sec.
```

Figure 3.57: Lenstra's Elliptic Curve Factoring Tool

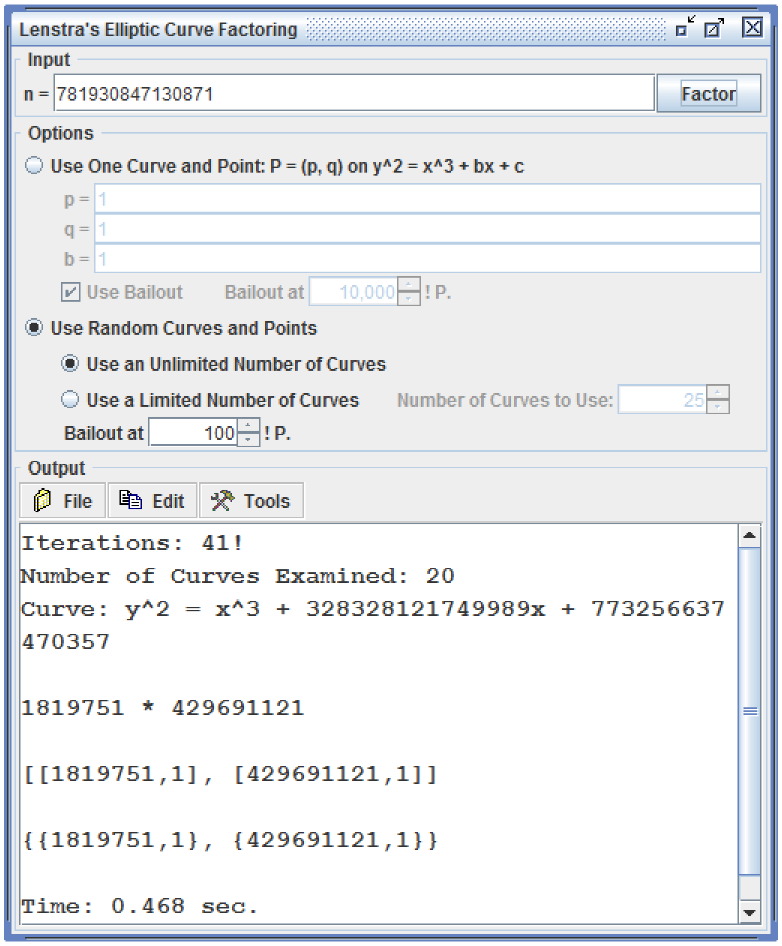
Options

Single Curve Mode —

- The point on the curve is $P = (p, q)$, the numbers p and q must be input by the user.
- The linear coefficient, b, of the elliptic curve, $y^2 = x^3 + bx + c \pmod{n}$, must be input by the user. The program will take this information and calculate the value of c so that the point P is on the curve. Note that if the calculated curve has multiple roots you will get a warning dialog that will allow you to either continue with the calculation or terminate the calculation.
- The user can specify if a bailout is to be used and the size of the bailout. If the user uses the bailout, when the bailout iteration is exceeded and no factor is found the algorithm will halt with a failure message.

Random Curve Mode —

- In this mode the program will generate random curves and points for the algorithm. The user can select to allow the program to run indefinitely or to limit the



n = 781930847130871

( ) Use One Curve and Point: P = (p, q) on y^2 = x^3 + bx + c

p = 1
q = 1
b = 1

[✓] Use Bailout   Bailout at 10,000 ! P.

(●) Use Random Curves and Points

(●) Use an Unlimited Number of Curves

( ) Use a Limited Number of Curves   Number of Curves to Use: 25

Bailout at 100 ! P.

```
Iterations: 41!
Number of Curves Examined: 20
Curve: y^2 = x^3 + 328328121749989x + 773256637
470357

1819751 * 429691121

[[1819751,1], [429691121,1]]

{{1819751,1}, {429691121,1}}

Time: 0.468 sec.
```

Figure 3.58: Lenstra's Elliptic Curve Factoring Tool

number of curves it uses.

- In this mode a bailout must be given. When the bailout iteration is exceeded for a curve the program will continue (restart) with another point and curve.

Status Bar Information

Since factoring methods could take some time to complete, there is status bar information that shows you if the method is making progress on factoring the number. On the far right is the elapsed time and on the left is a display of the curve number and iteration.

Notes

- The method does not find a complete prime factorization of n, even if it finds a factor. So the factors in the output could still be composite.
- The bailout values are in the range of 5 to 1,000,000.

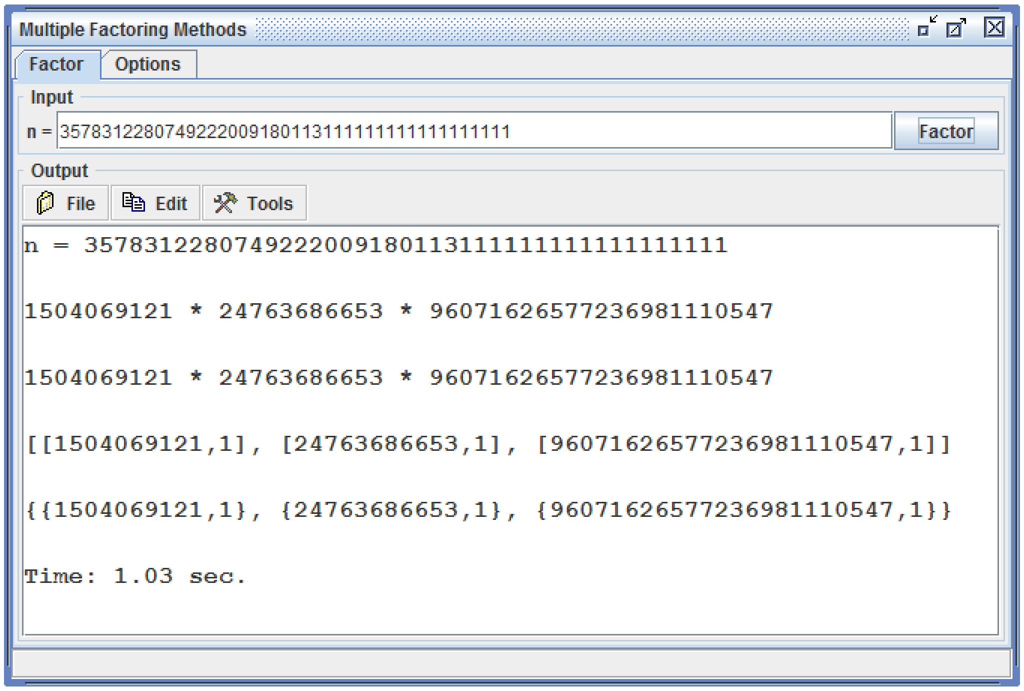
- The number of curves has a range of 1 to 1,000,000.
- The output of the factorization is in 3 forms. The first is as a product, the second is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger command. For example, when factoring 781930847130871 the output is,

```
Iterations: 619!
Number of Curves Examined: 3
Curve: y^2 = x^3 + 325191091533488x + 443604025114081
1819751 * 429691121
[[1819751,1], [429691121,1]]
{{1819751,1}, {429691121,1}}
Time: 0.234 sec.
```

- At the bottom of the output box is the amount of time used for the factoring operation.
- The output also has several benchmark numbers to let you know how much work was done by the algorithm.
  - In one curve mode the output includes the number of iterations the program did with the curve and the curve itself.
  - In random curve mode, the output includes the number of curves that were examined along with the last curve used and the number of iterations needed for that curve to factor n.
- Factoring integers can be a lengthy process, the factoring operation is done in a separate thread so that the user can use other facilities of the program while a factoring operation is being completed.
- When the factoring operation is invoked, the Factor button will be changed to an Abort button. To end the process simply click on the Abort button.
- Calculation of $m!P$ is done by the progression of calculations, $2P$, $3(2P)$, $4(3!P)$, $5(4!P)$, $\ldots$, $m((m-1)!P)$.

3.7.10 Multiple Factoring Methods

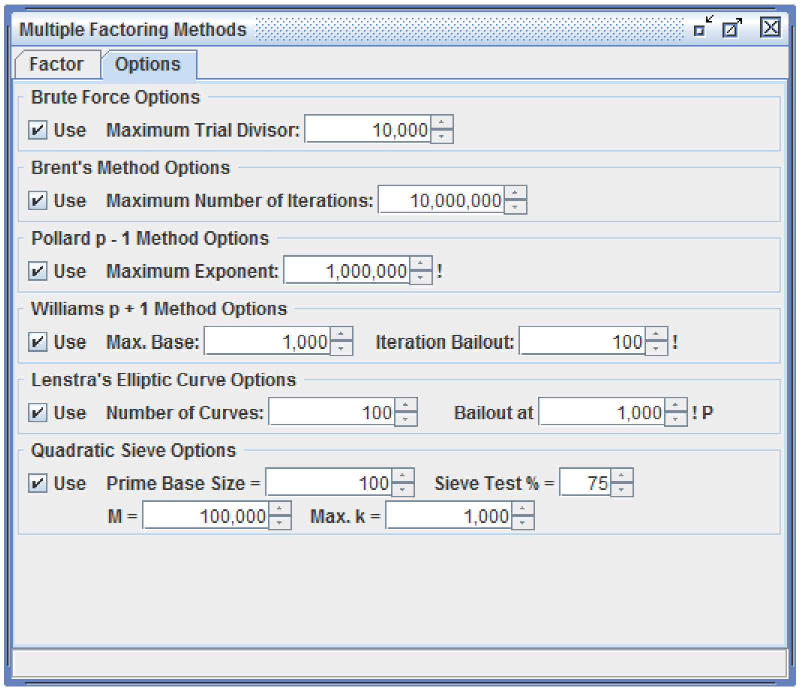
The Multiple Factoring Methods tool will factor an integer into its prime factor decomposition using a combination of various methods. The methods used can be found on the Options tab, they are applied in order of their listing. This tool allows the user to experiment with different factoring methods in combination with each other. It can be used as a general integer factoring tool but it is not as efficient as those found in most available computer algebra systems, such as Mathematica, Maxima, or Maple.



n = 3578312280749222009180113111111111111111

```
n = 3578312280749222009180113111111111111111

1504069121 * 24763686653 * 96071626577236981110547

1504069121 * 24763686653 * 96071626577236981110547

[[1504069121,1], [24763686653,1], [96071626577236981110547,1]]

{{1504069121,1}, {24763686653,1}, {96071626577236981110547,1}}

Time: 1.03 sec.
```

Figure 3.59: Multiple Factoring Methods Tool



Brute Force Options
✓ Use Maximum Trial Divisor: 10,000

Brent's Method Options
✓ Use Maximum Number of Iterations: 10,000,000

Pollard p - 1 Method Options
✓ Use Maximum Exponent: 1,000,000 !

Williams p + 1 Method Options
✓ Use Max. Base: 1,000 Iteration Bailout: 100 !

Lenstra's Elliptic Curve Options
✓ Use Number of Curves: 100 Bailout at 1,000 ! P

Quadratic Sieve Options
✓ Use Prime Base Size = 100 Sieve Test % = 75
M = 100,000 Max. k = 1,000

Figure 3.60: Multiple Factoring Methods Tool Options

How to Use the Tool

1. Input the number to be factored into the Input box.

2. Select the Options tab and alter the method options, if desired.
3. Click the Factor button.

Options

- All Methods can be either selected or deselected form use by checking or unchecking the Use check box in the method's option section.
- When any method finds a factor, the new factors are checked to see if they are prime or composite. If a factor is prime (that is probably prime with failure probability less than 2^{-100}) they are stored for final output and if they are composite the method is restarted on all composite factors.
- If a complete factorization is found at any time the current method halts and all subsequent methods are skipped.

Method Options

Brute Force Options —

Maximum Trial Divisor: This is the largest number that is used as a trial divisor before moving onto the next method.

Brent's Method Options —

Maximum Number of Iterations: This is the largest number of iterations that will be done using Brent's method.

Pollard $p - 1$ Method Options —

Maximum Exponent: This is the largest exponent used for Pollard's $p - 1$ method. This number is a factorial and corresponds to the number or iterations done in the process.

Williams $p + 1$ Method Options —

Max. Base: This is the largest number used for the seed of the Lucas sequence. The program will try each seed from 3 to this number.

Iteration Bailout: This is the largest number of terms of the Lucas sequence that will be calculated. This number is a factorial and corresponds to the number or iterations done in the process.

Lenstra's Elliptic Curve Options —

Number of Curves: This is the largest number of randomly generated elliptic curves used in the process.

Bailout at: This is the largest number of points that will be calculated for each curve. This number is a factorial and corresponds to the number or iterations done in the process.

Quadratic Sieve Options —

Prime Base: This is the number of primes in the prime base for the sieve. This tool uses the classical sieve, so all primes have Jacobi symbol 1 with n.

M: This is the size of the sieving array.

Max. k: This is the maximum multiple of M that is used for the the sieving array. So the sieve array starts at $[-M, M]$ and proceeds up to $[-kM, kM]$ until a factor is found.

Sieve Test %: This is the percentage of the theoretical bound used to determine a possible small prime factorization.

Status Bar Information

Since each factoring method could take some time to complete, there is status bar information that shows you if the method is making progress on factoring the number. Each method has different information displayed but all methods have the elapsed time to the right and `PF: ##  CF: ##` on the left. The PF stands for prime factors and is the number of prime factors found up to this point. The CF stands for composite factors and is the number of composite factors found up to this point. The other information is in the center of the status bar and depends on the current method being used.

Brute Force: Displays the current trial divisor being tested.

Brent's Method: Displays the current iteration.

Pollard $p - 1$ Method: Displays the current iteration.

Williams $p + 1$ Method: Displays the current base and iteration on that base.

Lenstra's Elliptic Curve: Displays the curve number and iteration.

Quadratic Sieve: Displays the number of trial numbers used (TN), the number of small prime factorizations (SPF), and the number of mod 2 dependency relations found (DR).

Notes

- If there is a composite number that was not able to be factored using the selected methods with the selected parameters it will be enclosed in parentheses in the final output.

- The output of the factorization is in 4 forms. The first is an expanded multiplication, the second is a product expression with powers, the third is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger command. For example, when factoring 11110988889 the output is,

```
3 * 3 * 3 * 7 * 11 * 13 * 37 * 41 * 271
3^3 * 7 * 11 * 13 * 37 * 41 * 271
[[3,3], [7,1], [11,1], [13,1], [37,1], [41,1], [271,1]]
{{3,3}, {7,1}, {11,1}, {13,1}, {37,1}, {41,1}, {271,1}}
Time: 0.047 sec.
```

- At the bottom of the output box is the amount of time used for the factoring operation.
- Factoring integers can be a lengthy process, the factoring operation is done in a separate thread so that the user can use other facilities of the program while a factoring operation is being completed.
- When the factoring operation is invoked, the Factor button will be changed to an Abort button. To end the process simply click on the Abort button.

3.8 Discrete Logarithm Tools

3.8.1 Brute Force Discrete Logarithm

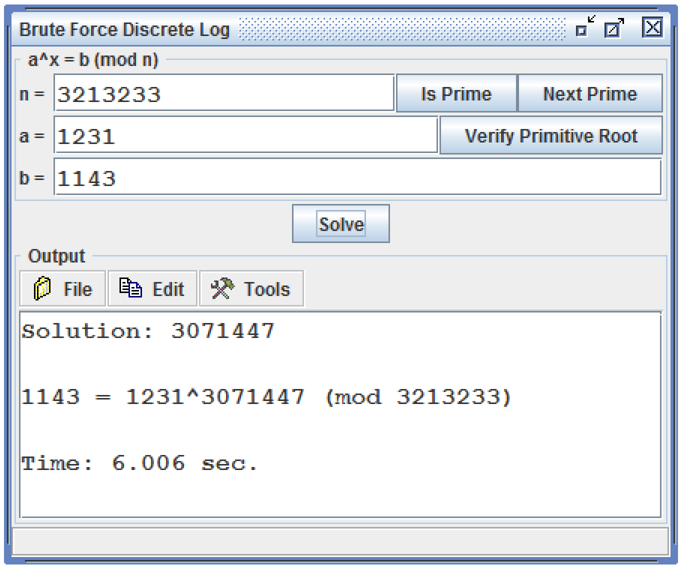
The Brute Force Discrete Logarithm tool will solve the discrete log problem of $b = a^x \pmod{n}$ given integers a, b, and n. The method simply tries all possible exponents x until a solution is found or it is determined that a solution does not exist.

How to Use the Tool

1. Input the numbers a, b, and n.
2. Click the Solve button.

Options

- The modulus, n, has options for determining the next probable prime and for testing if n is a probable prime. The value of n need not be prime for the program to run.
- The base, a, has the option to determine if it is a primitive root modulo n. In this case, n, must be a prime number. The determination of a primitive root depends on the factorization of the totient of n, and hence can be a lengthy process if n is a large prime number. The value of a need not be a primitive root for the program to run.



n = 3213233

a = 1231

b = 1143

```
Solution: 3071447

1143 = 1231^3071447 (mod 3213233)

Time: 6.006 sec.
```

Figure 3.61: Brute Force Discrete Logarithm Tool

Status Bar Information

Since discrete log methods could take some time to complete, there is status bar information that shows you if the method is making progress. On the far right is the elapsed time and on the left is a display of the current exponent being tested.

Notes

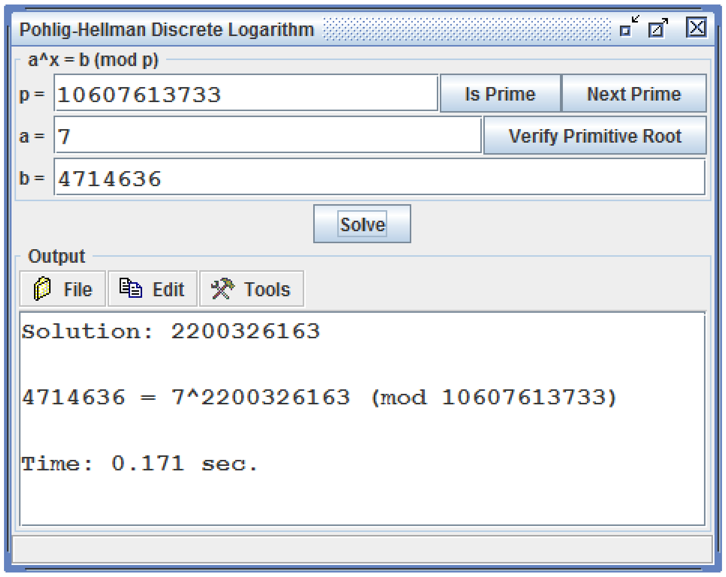
- Solving discrete log problems can be a lengthy process, the operation is done in a separate thread so that the user can use other facilities of the program while the operation is being completed.
- When the solving operation is invoked, the Solve button will be changed to an Abort button. To end the process simply click on the Abort button.

3.8.2 Pohlig-Hellman Discrete Logarithm

The Pohlig-Hellman Discrete Logarithm tool uses the Pohlig-Hellman algorithm to solve the discrete log problem of $b = a^x \pmod{p}$ given integers a, b, and p. For this tool, the modulus n must be prime and the base a must be a primitive root modulo p. Recall that the Pohlig-Hellman algorithm is efficient if $p - 1$ factors into a product of small primes.

How to Use the Tool

1. Input the numbers a, b, and p.
2. Click the Solve button.



p = 10607613733
a = 7
b = 4714636

```
Solution: 2200326163

4714636 = 7^2200326163 (mod 10607613733)

Time: 0.171 sec.
```

Figure 3.62: Pohlig-Hellman Discrete Logarithm Tool

Options

- The modulus, p, has options for determining the next probable prime and for testing if p is a probable prime. The value of p must be prime for the program to run.
- The base, a, has the option to determine if it is a primitive root modulo p. In this case, p, must be a prime number. The determination of a primitive root depends on the factorization of the totient of p, and hence can be a lengthy process if p is a large prime number. The value of a must be a primitive root for the program to run.

Status Bar Information

Since discrete log methods could take some time to complete, there is status bar information that shows you if the method is making progress. On the far right is the elapsed time. On the left is a display similar to the following.

```
Prime #3/5: 121259   Round: #2/3   Exponent: 23442
```

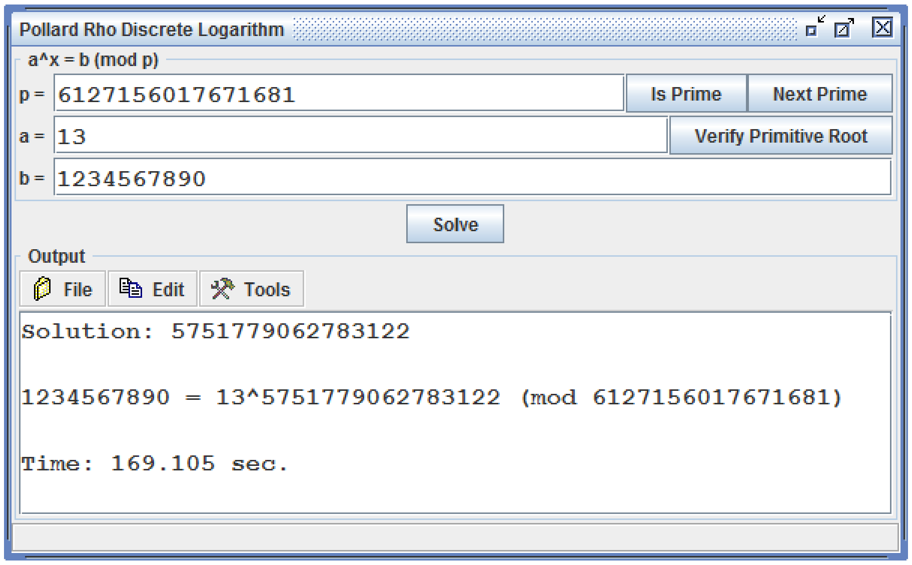
This means that there are 5 distinct primes in the factorization of $p-1$, and the program is currently working on the third prime, which is 121259. The Round indicates that in the factorization of $p-1$ there was a factor of 121259^3, so the program must run through 3 total rounds and it is currently on the second one. In that round, the exponent multiplier is currently 23442. Recall from the Pohlig-Hellman algorithm that each round might need to go through all the exponents up to the size of the prime being considered. So if the prime is large, $p-1$ did not factor into small primes, and the Pohlig-Hellman algorithm may not find a solution in a reasonable amount of time.

Notes

- In the status bar, the primes that are listed are sorted into increasing order. So in the above example, the two remaining primes are larger than 121259.
- Solving discrete log problems can be a lengthy process, the operation is done in a separate thread so that the user can use other facilities of the program while the operation is being completed.
- When the solving operation is invoked, the Solve button will be changed to an Abort button. To end the process simply click on the Abort button.

3.8.3 Pollard Rho Discrete Logarithm

The Pollard Rho Discrete Logarithm tool uses the Pollard Rho algorithm for discrete logs to solve the discrete log problem of $b = a^x \pmod{p}$ given integers a, b, and p. For this tool, the modulus p must be prime and the base a should be a primitive root modulo p, although the program will not force this second criteria.



p = 6127156017671681

a = 13

b = 1234567890

```
Solution: 5751779062783122

1234567890 = 13^5751779062783122 (mod 6127156017671681)

Time: 169.105 sec.
```

Figure 3.63: Pollard Rho Discrete Logarithm Tool

How to Use the Tool

1. Input the numbers a, b, and p.
2. Click the Solve button.

Options

- The modulus, p, has options for determining the next probable prime and for testing if p is a probable prime. The value of p must be prime for the program to run.
- The base, a, has the option to determine if it is a primitive root modulo p. In this case, p, must be a prime number. The determination of a primitive root depends on the factorization of the totient of p, and hence can be a lengthy process if p is a large prime number.

Status Bar Information

Since discrete log methods could take some time to complete, there is status bar information that shows you if the method is making progress. On the far right is the elapsed time. On the left is a display of the current iteration.

Notes

- Solving discrete log problems can be a lengthy process, the operation is done in a separate thread so that the user can use other facilities of the program while the operation is being completed.
- When the solving operation is invoked, the Solve button will be changed to an Abort button. To end the process simply click on the Abort button.

3.8.4 Pohlig-Hellman with Pollard Rho Discrete Logarithm

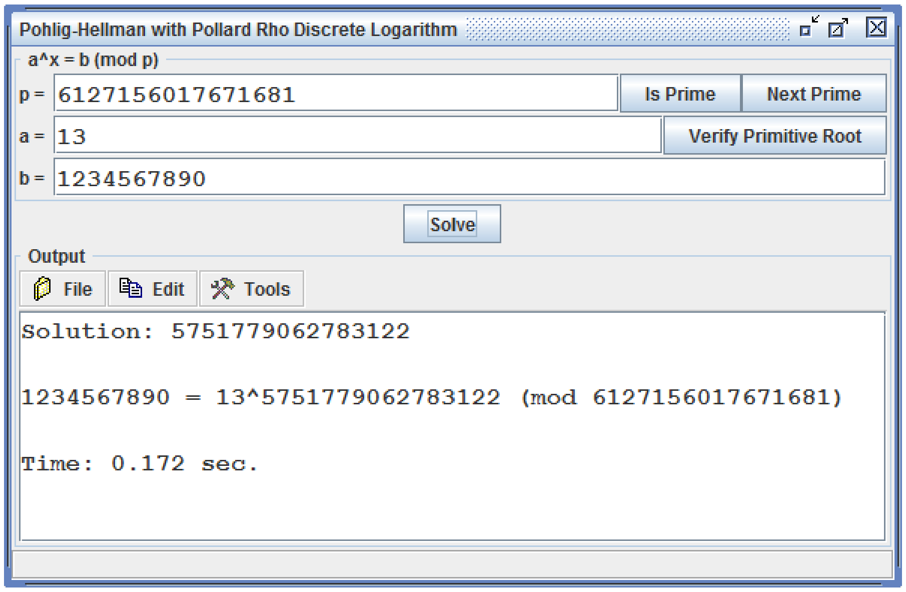
The Pohlig-Hellman with Pollard Rho Discrete Logarithm tool uses the Pohlig-Hellman algorithm to solve the discrete log problem of $b = a^x \pmod{p}$ given integers a, b, and p. In the classical Pohlig-Hellman algorithm a brute force algorithm is used to solve smaller discrete logarithm problems. In the version that brute force portion is replaced with the faster Pollard Rho algorithm. For this tool, the modulus p must be prime and the base a must be a primitive root modulo p. Recall that the Pohlig-Hellman algorithm is efficient if $p - 1$ factors into a product of small primes.

How to Use the Tool

1. Input the numbers a, b, and p.
2. Click the Solve button.

Options

- The modulus, p, has options for determining the next probable prime and for testing if p is a probable prime. The value of p must be prime for the program to run.



p = 6127156017671681
a = 13
b = 1234567890

```
Solution: 5751779062783122

1234567890 = 13^5751779062783122 (mod 6127156017671681)

Time: 0.172 sec.
```

Figure 3.64: Pohlig-Hellman with Pollard Rho Discrete Logarithm Tool

- The base, a, has the option to determine if it is a primitive root modulo p. In this case, p, must be a prime number. The determination of a primitive root depends on the factorization of the totient of p, and hence can be a lengthy process if p is a large prime number. The value of a must be a primitive root for the program to run.

Status Bar Information

Since discrete log methods could take some time to complete, there is status bar information that shows you if the method is making progress. On the far right is the elapsed time. On the left is a display similar to the following.

```
Prime #3/5: 121259    Round: #2/3    Exponent: 23442
```

or

```
Prime #3/5: 121259    Round: #2/3    Iteration: 23442
```

This means that there are 5 distinct primes in the factorization of $p-1$, and the program is currently working on the third prime, which is 121259. The Round indicates that in the factorization of $p-1$ there was a factor of 121259^3, so the program must run through 3 total rounds and it is currently on the second one. In that round, the exponent multiplier or Pollard Rho iteration is currently 23442. If the prime is less than 10,000 brute force is used for the smaller logarithms and if it is larger then the Pollard Rho algorithm is used. If the prime is large, and $p-1$ did not factor into relatively small primes, the Pohlig-Hellman algorithm may not find a solution in a reasonable amount of time.

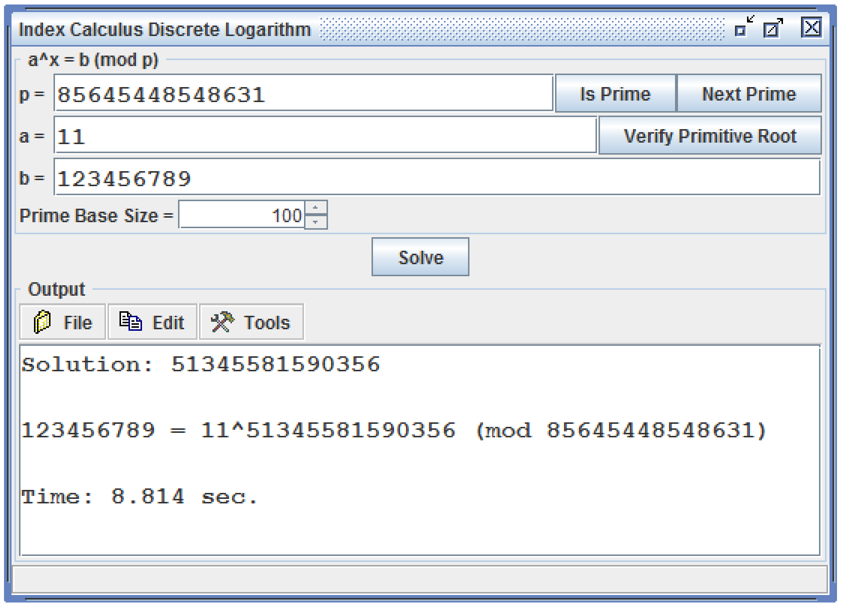
Notes

- In the status bar, the primes that are listed are sorted into increasing order. So in the above example, the two remaining primes are larger than 121259.
- Solving discrete log problems can be a lengthy process, the operation is done in a separate thread so that the user can use other facilities of the program while the operation is being completed.
- When the solving operation is invoked, the Solve button will be changed to an Abort button. To end the process simply click on the Abort button.

3.8.5 Index Calculus Discrete Logarithm

The Index Calculus Discrete Logarithm tool uses the Index Calculus algorithm to solve the discrete log problem of $b = a^x \pmod{p}$ given integers a, b, and p. For this tool, the modulus p must be prime and the base a should be a primitive root modulo p, although the program will not force this second criteria.

The program will first use small prime factorizations and solving of the linear systems of the exponents to find the discrete logarithms base a of each prime in the prime base modulo p. Then it will calculate $a^r \cdot b \pmod{p}$ for various r until the number factors in the prime base and then using the small prime logarithms it will calculate the x that solves $b = a^x \pmod{p}$.



p = 85645448548631
a = 11
b = 123456789
Prime Base Size = 100

```
Solution: 51345581590356

123456789 = 11^51345581590356 (mod 85645448548631)

Time: 8.814 sec.
```

Figure 3.65: Index Calculus Discrete Logarithm Tool

How to Use the Tool

1. Input the numbers a, b, and p.
2. Input the size of the prime base.
3. Click the Solve button.

Options

- The modulus, p, has options for determining the next probable prime and for testing if p is a probable prime. The value of p must be prime for the program to run.
- The base, a, has the option to determine if it is a primitive root modulo p. In this case, p, must be a prime number. The determination of a primitive root depends on the factorization of the totient of p, and hence can be a lengthy process if p is a large prime number.

Status Bar Information

Since discrete log methods could take some time to complete, there is status bar information that shows you if the method is making progress. On the far right is the elapsed time. On the left is a display of the current exponent and the number of small prime factorizations found.

Notes

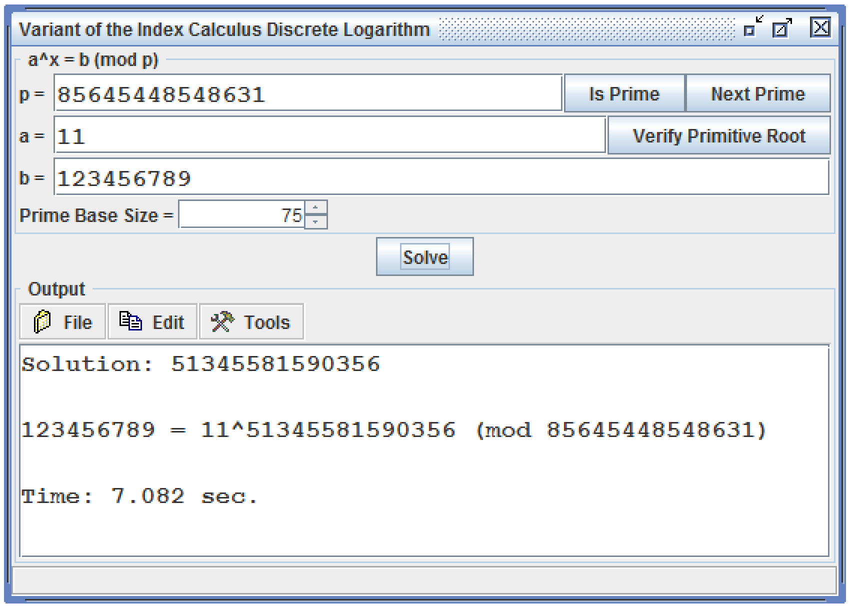
- Solving discrete log problems can be a lengthy process, the operation is done in a separate thread so that the user can use other facilities of the program while the operation is being completed.
- When the solving operation is invoked, the Solve button will be changed to an Abort button. To end the process simply click on the Abort button.

3.8.6 Variant of the Index Calculus Discrete Logarithm

The Index Calculus Discrete Logarithm tool uses the Index Calculus algorithm to solve the discrete log problem of $b = a^x \pmod{p}$ given integers a, b, and p. For this tool, the modulus p must be prime and the base a should be a primitive root modulo p, although the program will not force this second criteria.

This method uses the same process as the classical index calculus algorithm except that it rearranges the process and in most cases it is able to solve the discrete logarithm using fewer steps, and hence faster. In the worst case, it may run slower than the classical index calculus algorithm due to the overhead in checking for partial solutions and matrix resizing.

The program will first calculate $a^r \cdot b \pmod{p}$ for various r until the number factors in the prime base. The program will then use small prime factorizations of $a^s \pmod{p}$, for increasing values of s, and solving of the linear systems of the exponents to find the discrete logarithms base a of each prime in the prime base modulo p. During this process, the program will check for any new logarithm solutions to the primes in the prime base. Once the logarithms of the primes in the factorization of $a^r \cdot b \pmod{p}$ are found the program can quickly calculate the solution to $b = a^x \pmod{p}$ without the solutions to the remaining primes in the prime base and it does so. If a new prime base logarithm is found but the program does not have enough information to calculate the final solution, the program will store the new solution and reduce the size of the linear system, making the reduction phase of the algorithm faster.



p = 85645448548631

a = 11

b = 123456789

Prime Base Size = 75

```
Solution: 51345581590356

123456789 = 11^51345581590356 (mod 85645448548631)

Time: 7.082 sec.
```

Figure 3.66: Variant of the Index Calculus Discrete Logarithm Tool

How to Use the Tool

1. Input the numbers a, b, and p.
2. Input the size of the prime base.
3. Click the Solve button.

Options

- The modulus, p, has options for determining the next probable prime and for testing if p is a probable prime. The value of p must be prime for the program to run.
- The base, a, has the option to determine if it is a primitive root modulo p. In this case, p, must be a prime number. The determination of a primitive root depends on the factorization of the totient of p, and hence can be a lengthy process if p is a large prime number.

Status Bar Information

Since discrete log methods could take some time to complete, there is status bar information that shows you if the method is making progress. On the far right is the elapsed time. On the left is a display of the current exponent and the number of small prime factorizations found.

Notes

- Solving discrete log problems can be a lengthy process, the operation is done in a separate thread so that the user can use other facilities of the program while the operation is being completed.
- When the solving operation is invoked, the Solve button will be changed to an Abort button. To end the process simply click on the Abort button.

Bibliography

[1] O. I. Franksen. *Mr. Babbage's Secret: the Tale of a Cipher—and APL.* Prentice Hall, 1985.

[2] F. W. Kasiski. *Die Geheimschriften und die Dechiffrir-Kunst.* E. S. Mittler und Sohn, Berlin, 1863.

[3] Maxima. Maxima, a Computer Algebra System. `http://maxima.sourceforge.net/`, 2014. [Online; accessed January 2, 2014].

[4] Wolfram. Mathematica. `http://www.wolfram.com/`, 2014. [Online; accessed January 2, 2014].

Index