Cryptography Notes

Technology Guide on Using Mathematica, Maxima, and Cryptography Explorer

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 supplement for a first course in cryptography aimed at the third or fourth year undergraduate mathematics major. We assume a basic understanding of linear algebra, discrete mathematics and modular arithmetic. This document is also intended to be a technology guide for the same class, concentrating on the packages, Mathematica, Maxima, and Cryptography Explorer. It is not intended to be a textbook but to accompany one. It is also not intended to be a general reference for Mathematica or Maxima. Although there are appendices for getting started with Mathematica and Maxima, these concentrate on the background needed for cryptographic applications.

Although these materials were designed for mathematics majors, there are segments that are accessible for those in a survey course in mathematics and for high school students. Specifically, the first chapter on the background material for cryptography and the chapter on classical methods, up to the section on the Linear Feedback Shift Register. From there through the remainder of the notes we assume the student has seen some elementary linear algebra.

Cryptography, like nearly all areas of mathematics, has both a theoretical side and an application side. Both of these play off of one another to advance the field. In some cases, the theoretical mathematics involved produces applications that quickly become computationally infeasible, and hence we do not have the computational power to explore "real world" scenarios. But even in these cases we can use simpler examples that portray the ideas behind the methods and help us understand the important points in the question of data security.

Exploration is one of the best teachers of mathematics. Working through the mathematics is an important part of learning mathematics but if it stops there we miss out on so much of the whole experience. While a concept is being earned we need to look at numerous examples to get a feel for the extent and limitations of the concept. It also helps to make the theoretical concepts more concrete. When I was an undergraduate, (yes this is an old guy talking) we did not have the technological tools, at least they were not readily available, to do more involved experimentation. Nonetheless, the experimentation that we could do back then helped me understand the more esoteric aspects of the material. With technology, we have the ability to bring in more substantial examples that are nearly impossible to do by hand. If you decide to use these notes, either in part or in full, please take the time to work through the examples, and make up your own examples.

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 then 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/).[77]

> 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/).[33]

> 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.asp

Edition
July 26, 2019

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

Contents

1 Cryptography **1**
- 1.1 Introduction & Some Definitions . . . 1
- 1.2 Classical and Modern Cryptography . . . 1
- 1.3 Alice, Bob & Eve . . . 2
- 1.4 Keys and Key Spaces . . . 3
- 1.5 Kerckhoffs' Principle . . . 3
- 1.6 Codes verses Ciphers . . . 4
- 1.7 Public verses Symmetric Key . . . 8
- 1.8 Stream verses Block Ciphers . . . 9
- 1.9 Attacks on a Cryptographic Method . . . 10
- 1.10 The Main Objectives of Cryptography . . . 11

2 Classical Cryptography **13**
- 2.1 Introduction . . . 13
- 2.2 Monoalphabetic Substitution . . . 15
  - 2.2.1 Definitions and History . . . 15
  - 2.2.2 Cryptanalysis . . . 27
  - 2.2.3 Cryptography Explorer . . . 40
- 2.3 Vigenère . . . 47
  - 2.3.1 Definitions and History . . . 47
  - 2.3.2 Cryptanalysis . . . 52
  - 2.3.3 Cryptography Explorer . . . 74
- 2.4 Scytale . . . 88
  - 2.4.1 Definitions and History . . . 88
  - 2.4.2 Cryptanalysis . . . 89
  - 2.4.3 Cryptography Explorer . . . 89
- 2.5 Rail Fence . . . 91
  - 2.5.1 Definitions and History . . . 91
  - 2.5.2 Cryptanalysis . . . 91
  - 2.5.3 Cryptography Explorer . . . 92
- 2.6 Columnar . . . 93
  - 2.6.1 Definitions and History . . . 93
  - 2.6.2 Cryptanalysis . . . 94

2.6.3 Cryptography Explorer 102
2.7 Playfair 106
2.7.1 Definitions and History 106
2.7.2 Cryptanalysis 112
2.7.3 Cryptography Explorer 114
2.8 Two Square 118
2.8.1 Definitions and History 118
2.8.2 Cryptanalysis 122
2.8.3 Cryptography Explorer 123
2.9 Four Square 127
2.9.1 Definitions and History 127
2.9.2 Cryptanalysis 131
2.9.3 Cryptography Explorer 133
2.10 ADFGX and ADFGVX 136
2.10.1 Definitions and History 136
2.10.2 Cryptanalysis 140
2.10.3 Cryptography Explorer 152
2.11 Linear Feedback Shift Register (LFSR) 159
2.11.1 Definitions and History 159
2.11.2 Cryptanalysis 164
2.11.3 Cryptography Explorer 173
2.11.4 Mathematica 177
2.11.5 Maxima 177
2.12 One Time Pad 179
2.12.1 Definitions and History 179
2.12.2 Cryptography Explorer 186
2.12.3 Examples Using Cryptography Explorer 193
2.13 The Hill Cipher 197
2.13.1 Introduction 197
2.13.2 History 197
2.13.3 Encryption with the Hill Cipher 198
2.13.4 Decryption with the Hill Cipher 200
2.13.5 Breaking the Hill Cipher 202
2.13.6 The Hill Cipher from a Linear Algebra Point of View 204
2.13.7 Cryptography Explorer 209
2.13.8 Mathematica 225
2.13.9 Maxima 242
2.13.10 Exercises 274
2.14 The Hill Cipher Classical Approach 287
2.14.1 Introduction 287
2.14.2 History 287
2.14.3 Encryption with the Hill Cipher 288
2.14.4 Decryption with the Hill Cipher 291

2.14.5 Breaking the Hill Cipher . . . 292
2.14.6 The Hill Cipher from a Linear Algebra Point of View . . . 295
2.14.7 Cryptography Explorer . . . 300
2.14.8 Mathematica . . . 317
2.14.9 Maxima . . . 334
2.14.10 Exercises . . . 367
2.15 Cipher Machines . . . 380
2.15.1 Introduction . . . 380
2.15.2 Enigma . . . 380
2.15.3 The Red and Purple . . . 385
2.15.4 M-209 . . . 386
2.15.5 Cryptography Explorer . . . 387

3 Modern Cryptography **390**
3.1 Introduction . . . 390
3.2 RSA . . . 395
3.2.1 History . . . 395
3.2.2 The RSA Algorithm . . . 395
3.2.3 Examples . . . 398
3.2.4 Applications . . . 416
3.3 More to Come . . . 418

A Introduction to Mathematica **419**
A.1 What is Mathematica? . . . 419
A.2 The User Interface . . . 420
A.3 Basic Calculations . . . 425
A.3.1 Numeric Calculations . . . 425
A.3.2 Algebra . . . 427
A.3.3 Execution Timing . . . 429
A.4 Defining Functions . . . 430
A.5 Some Discrete Mathematics & Number Theory Commands . . . 432
A.5.1 Modulus . . . 432
A.5.2 Power Calculations with a Modulus . . . 432
A.5.3 Greatest Common Divisor . . . 433
A.5.4 Extended Greatest Common Divisor . . . 433
A.5.5 Least Common Multiple . . . 434
A.5.6 Chinese Remainder Theorem . . . 434
A.5.7 Functions for Primes . . . 435
A.5.8 Jacobi and Legendre Symbols . . . 436
A.5.9 Continued Fractions . . . 437
A.5.10 Solving Equations . . . 440
A.5.11 Factoring . . . 441
A.5.12 Factoring Polynomials . . . 442
A.5.13 Euler Totient Function . . . 442

A.5.14 Primitive Roots and Element Orders . . . 443
A.5.15 Discrete Logarithms . . . 444
A.6 Vectors and Matrices . . . 445
A.6.1 Defining a Matrix and a Vector . . . 445
A.6.2 Matrix Arithmetic . . . 449
A.6.3 Matrix Reduction . . . 451
A.6.4 Modular Matrix Operations . . . 452
A.7 Elliptic Curves . . . 454
A.7.1 Points on an Elliptic Curve . . . 455
A.7.2 Arithmetic on an Elliptic Curve . . . 459
A.8 CryptDSEC.nb . . . 462
A.8.1 CryptDSEC.nb Code . . . 463

B Introduction to Maxima 464
B.1 What is Maxima? . . . 464
B.2 The User Interface . . . 466
B.3 Basic Calculations . . . 469
B.3.1 Numeric Calculations . . . 469
B.3.2 Algebra . . . 471
B.3.3 Execution Timing . . . 473
B.4 Defining Functions . . . 473
B.5 Some Discrete Mathematics & Number Theory Commands . . . 477
B.5.1 Modulus . . . 477
B.5.2 Power Calculations with a Modulus . . . 477
B.5.3 Inverse Calculations with a Modulus . . . 478
B.5.4 Greatest Common Divisor . . . 478
B.5.5 Extended Greatest Common Divisor . . . 479
B.5.6 Greatest Common Divisor of Several Numbers . . . 479
B.5.7 Least Common Multiple . . . 480
B.5.8 Chinese Remainder Theorem . . . 480
B.5.9 Functions for Primes . . . 480
B.5.10 Jacobi and Legendre Symbols . . . 482
B.5.11 Continued Fractions . . . 483
B.5.12 Solving Equations . . . 486
B.5.13 Modular Square Roots and Cube Roots . . . 488
B.5.14 Factoring . . . 489
B.5.15 Factoring Polynomials . . . 489
B.5.16 Euler Totient Function . . . 490
B.5.17 Primitive Roots . . . 490
B.5.18 Discrete Logarithms . . . 491
B.5.19 Order of an Element . . . 492
B.6 Vectors and Matrices . . . 493
B.6.1 Defining a Matrix . . . 493

B.6.2 Matrix Arithmetic . . . 500
B.6.3 Matrix Reduction . . . 502
B.6.4 Modular Matrix Operations . . . 504
B.7 Elliptic Curves . . . 514
B.7.1 Points on an Elliptic Curve . . . 514
B.7.2 Arithmetic on an Elliptic Curve . . . 517
B.8 CryptDS.mac . . . 521
B.8.1 CryptDS.mac Code . . . 521

C Introduction to Cryptography Explorer 524
C.1 What is Cryptography Explorer? . . . 524
C.2 Introduction . . . 524
C.3 Ciphers . . . 528
C.3.1 Monoalphabetic Substitution . . . 528
C.3.2 Vigenère . . . 531
C.3.3 Scytale . . . 534
C.3.4 Rail Fence . . . 535
C.3.5 Columnar . . . 536
C.3.6 Two Square . . . 539
C.3.7 Four Square . . . 543
C.3.8 Playfair . . . 548
C.3.9 ADFGX . . . 552
C.3.10 ADFGVX . . . 555
C.3.11 Linear Feedback Shift Register (LFSR) . . . 558
C.3.12 Hill . . . 559
C.3.13 Enigma . . . 563
C.3.14 RSA . . . 565
C.3.15 ElGamal . . . 567
C.4 Text and Stream Analysis . . . 569
C.4.1 Frequency Analysis . . . 569
C.4.2 Hill Climb Analysis . . . 571
C.4.3 Kasiski's Method . . . 573
C.4.4 Coincidence Analysis . . . 575
C.4.5 Dot Product Analysis . . . 576
C.4.6 Substring Compare . . . 577
C.4.7 LFSR Cipher Analysis . . . 578
C.5 Text Tools . . . 580
C.5.1 Text Extractor . . . 580
C.5.2 Text Combiner . . . 581
C.5.3 Text Converter . . . 582
C.5.4 Notepad . . . 589
C.5.5 Gridpad . . . 589
C.5.6 User Defined Language Creator . . . 591

C.6 Calculators 593
C.6.1 Integer Calculator 593
C.6.2 Modular Matrix Calculator 597
C.6.3 Elliptic Curve Calculator 600
C.6.4 Random Number Generator 603
C.7 Factoring Tools 604
C.7.1 Brute Force Factoring 604
C.7.2 Fermat Factoring 606
C.7.3 Pollard $p - 1$ Factoring 607
C.7.4 Williams P + 1 Factoring 609
C.7.5 Pollard Rho Factoring 610
C.7.6 Brent's Method Factoring 612
C.7.7 Quadratic Sieve Factoring 614
C.7.8 Multiple Polynomial Quadratic Sieve Factoring 618
C.7.9 Lenstra's Elliptic Curve Factoring 621
C.7.10 Multiple Factoring Methods 624
C.8 Discrete Logarithm Tools 628
C.8.1 Brute Force Discrete Logarithm 628
C.8.2 Pohlig-Hellman Discrete Logarithm 629
C.8.3 Pollard Rho Discrete Logarithm 631
C.8.4 Pohlig-Hellman with Pollard Rho Discrete Logarithm 632
C.8.5 Index Calculus Discrete Logarithm 634
C.8.6 Variant of the Index Calculus Discrete Logarithm 635

D Steganography 638
D.1 Some Examples from Antiquity 638
D.1.1 Wax Tablets 638
D.1.2 Scytale 639
D.1.3 Messages Written on the Messenger's Body 639
D.1.4 Messages on the Printed Page 640
D.2 The 20th Century 642
D.2.1 Invisible Ink 642
D.2.2 The Microdot 643
D.3 Image Steganography 643

E A Quick Reminder on Modular Arithmetic 648
E.1 Definition 648
E.2 Arithmetic 649
E.3 Matrices 650

Bibliography 655

Index 660

List of Figures

1.1 Alice, Bob & Eve . . . 2
1.2 Symmetric-Key Cryptography . . . 8
1.3 Public-Key Cryptography . . . 8

2.1 Pigpen Cipher . . . 15
2.2 Pigpen Cipher Ring . . . 16
2.3 Dancing Men . . . 17
2.4 Dancing Men: Sherlock Holmes At The Museum Of London . . . 17
2.5 English Letter Relative Frequencies Sorted by Frequency . . . 28
2.6 English Letter Relative Frequencies Sorted by Letter . . . 28
2.7 Cryptography Explorer Monoalphabetic Substitution Cipher Tool . . . 40
2.8 Cryptography Explorer Monoalphabetic Substitution Cipher Tool Displaying Partial Decryption . . . 42
2.9 Frequency Analysis Tool . . . 43
2.10 Frequency Analysis Tool . . . 44
2.11 Hill Climb Analysis Tool . . . 45
2.12 Frequency Analysis of Vigenère Example . . . 50
2.13 English Letter Relative Frequencies Sorted by Letter . . . 50
2.14 Kasiskis Method on the Vigenère Example . . . 55
2.15 Coincidence Analysis on a Vigenère Encryption . . . 56
2.16 Coincidence Analysis on the Vigenère Example . . . 57
2.17 Coincidence Analysis on a Vigenère Encryption of Three Character Alphabet . . . 59
2.18 Frequency Analysis of Shakespeare's Sonnet #18 . . . 61
2.19 Kasiski's Method on Vigenère Example of Shakespeare's Sonnet #18 . . . 62
2.20 Coincidence Analysis on Vigenère Example of Shakespeare's Sonnet #18 . . . 62
2.21 Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #1 . . . 63
2.22 Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #2 . . . 64
2.23 Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #3 . . . 64
2.24 Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #4 . . . 64
2.25 Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #5 . . . 65
2.26 Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #6 . . . 65
2.27 Dot Product Analysis on a Shift Cipher . . . 69
2.28 Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #1 . . . 70
2.29 Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #2 . . . 70

2.30 Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #3 . . . 71
2.31 Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #4 . . . 71
2.32 Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #5 . . . 71
2.33 Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #6 . . . 71
2.34 Vigenère Cipher Tool . . . 75
2.35 Kasiski's Method Analysis Tool . . . 78
2.36 Coincidence Analysis Tool . . . 79
2.37 Frequency Analysis Tool . . . 80
2.38 Dot Product Analysis Tool . . . 81
2.39 Text Extractor Tool . . . 82
2.40 Kasiski's Method Dialog on Vigenère Example of Shakespeare's Sonnet #18 83
2.41 Coincidence Analysis Dialog on Vigenère Example of Shakespeare's Sonnet #18 84
2.42 Text Extractor Dialog on Vigenère Example of Shakespeare's Sonnet #18 . . 85
2.43 Frequency Analysis Dialog on Vigenère Example of Sonnet #18: Subtext #1 86
2.44 Dot Product Analysis Dialog on Vigenère Example of Sonnet #18: Subtext #1 87
2.45 Skytale . . . 88
2.46 Scytale Cipher Tool . . . 90
2.47 Rail Fence Cipher Tool . . . 92
2.48 Columnar Cipher Tool: Classical Mode . . . 103
2.49 Columnar Cipher Tool: Myszkowski Mode . . . 103
2.50 Substring Comparison Tool . . . 105
2.51 Wheatstone Bridge . . . 106
2.52 Holmes Stereoscope . . . 106
2.53 Stereoscopic Movie Camera . . . 107
2.54 Playfair Cipher Tool: Keyword Mode . . . 115
2.55 Playfair Cipher Tool: Matrix Mode . . . 116
2.56 Two Square Cipher Tool: Keyword Mode . . . 124
2.57 Two Square Cipher Tool: Key Matrix Mode . . . 125
2.58 Four Square Cipher Tool: Keyword Mode . . . 133
2.59 Four Square Cipher Tool: Matrix Mode . . . 134
2.60 ADFGX Cipher Tool . . . 153
2.61 ADFGVX Cipher Tool . . . 154
2.62 Substring Comparison Tool . . . 155
2.63 Hill Climb Analysis Tool . . . 156
2.64 Linear Feedback Shift Register Cipher Tool . . . 173
2.65 LFSR Analysis Tool: Determinants . . . 174
2.66 LFSR Analysis Tool: Relation . . . 175
2.67 Vernam Patent 1,310,719 . . . 181
2.68 Vigenère Cipher Tool . . . 187
2.69 Linear Feedback Shift Register Cipher Tool . . . 189
2.70 Random Number Generator . . . 190
2.71 Text Converter Tool . . . 191
2.72 Character Stream Block Dialog . . . 192

2.73 Hill Cipher Tool . . . 210
2.74 Modular Matrix Calculator . . . 213
2.75 New Matrix Dialog . . . 213
2.76 Integer Calculator . . . 216
2.77 Hill Cipher Tool . . . 301
2.78 Modular Matrix Calculator . . . 303
2.79 New Matrix Dialog . . . 304
2.80 Integer Calculator . . . 306
2.81 The Enigma . . . 380
2.82 Enigma Layout: Image taken from CrypTool 2.0 Enigma Presentation View[41] 381
2.83 The Enigma Rotor Assembly . . . 382
2.84 Exploded View of Enigma Rotor Assembly . . . 383
2.85 The Bombe . . . 384
2.86 Part of the Purple Machine . . . 385
2.87 The M-209 Machine . . . 386
2.88 Enigma Machine Simulation Tool . . . 387

3.1 Symmetric-Key Cryptography . . . 390
3.2 Three-Pass Protocol . . . 391
3.3 One-Pass Protocol . . . 392
3.4 Public-Key Cryptography . . . 392
3.5 Shamir, Rivest and Adleman[1] . . . 395
3.6 RSA Cipher Tool . . . 403
3.7 Text Converter Tool . . . 405
3.8 Integer Calculator . . . 408
3.9 Encryption Information for Amazon.com . . . 416
3.10 Encryption Method for Amazon.com . . . 416

A.1 User Interface to Mathematica 10.0 . . . 420
A.2 User Interface to Mathematica 10.0 with Commands . . . 421
A.3 Mathematica Palette . . . 423
A.4 User Interface to Mathematica 10.0 Command Completion Interface . . . 424

B.1 User Interface to Maxima . . . 466
B.2 User Interface to Maxima with Commands . . . 467

C.1 Cryptography Explorer Main Window . . . 524
C.2 Print Preview . . . 527
C.3 Bar Charts . . . 527
C.4 Monoalphabetic Substitution Cipher Tool . . . 528
C.5 Vigenère Cipher Tool . . . 532
C.6 Scytale Cipher Tool . . . 535
C.7 Rail Fence Cipher Tool . . . 536
C.8 Columnar Cipher Tool: Classical Mode . . . 537
C.9 Columnar Cipher Tool: Myszkowski Mode . . . 538

C.10 Two Square Cipher Tool: Keyword Mode . . . 541
C.11 Two Square Cipher Tool: Key Matrix Mode . . . 541
C.12 Four Square Cipher Tool: Keyword Mode . . . 545
C.13 Four Square Cipher Tool: Matrix Mode . . . 545
C.14 Playfair Cipher Tool: Keyword Mode . . . 549
C.15 Playfair Cipher Tool: Matrix Mode . . . 550
C.16 ADFGX Cipher Tool . . . 553
C.17 ADFGVX Cipher Tool . . . 556
C.18 Linear Feedback Shift Register Cipher Tool . . . 558
C.19 Hill Cipher Tool . . . 560
C.20 Enigma Machine Simulation Tool . . . 563
C.21 RSA Cipher Tool . . . 565
C.22 ElGamal Cipher Tool . . . 567
C.23 Frequency Analysis Tool . . . 569
C.24 Hill Climb Analysis Tool . . . 572
C.25 Kasiski's Method Analysis Tool . . . 574
C.26 Coincidence Analysis Tool . . . 575
C.27 Dot Product Analysis Tool . . . 576
C.28 Substring Comparison Tool . . . 578
C.29 LFSR Analysis Tool: Determinants . . . 579
C.30 LFSR Analysis Tool: Relation . . . 579
C.31 Text Extractor Tool . . . 580
C.32 Text Combiner Tool . . . 581
C.33 Text Converter Tool . . . 582
C.34 Replace All Dialog . . . 583
C.35 Split All Dialog . . . 583
C.36 Base Change Dialog . . . 585
C.37 Binary Stream Block Dialog . . . 586
C.38 ASCII Stream Block Dialog . . . 587
C.39 Character Stream Block Dialog . . . 588
C.40 Notepad . . . 589
C.41 Gridpad . . . 589
C.42 User Defined Language Creator Tool . . . 591
C.43 Integer Calculator . . . 593
C.44 Modular Matrix Calculator . . . 598
C.45 New Matrix Dialog . . . 598
C.46 Modular Matrix Calculator . . . 601
C.47 Random Number Generator . . . 603
C.48 Brute Force Factoring Tool . . . 604
C.49 Fermat Factoring Tool . . . 606
C.50 Pollard $p - 1$ Factoring Tool . . . 608
C.51 Williams $p + 1$ Factoring Tool . . . 609
C.52 Pollard Rho Factoring Tool . . . 611

C.53 Brent's Method Factoring Tool . . . 613
C.54 Quadratic Sieve Factoring Tool: Classical Sieve . . . 615
C.55 Quadratic Sieve Factoring Tool: Alternative Method . . . 616
C.56 Multiple Polynomial Quadratic Sieve Factoring Tool . . . 619
C.57 Lenstra's Elliptic Curve Factoring Tool . . . 622
C.58 Lenstra's Elliptic Curve Factoring Tool . . . 623
C.59 Multiple Factoring Methods Tool . . . 625
C.60 Multiple Factoring Methods Tool Options . . . 625
C.61 Brute Force Discrete Logarithm Tool . . . 629
C.62 Pohlig-Hellman Discrete Logarithm Tool . . . 630
C.63 Pollard Rho Discrete Logarithm Tool . . . 631
C.64 Pohlig-Hellman with Pollard Rho Discrete Logarithm Tool . . . 633
C.65 Index Calculus Discrete Logarithm Tool . . . 634
C.66 Variant of the Index Calculus Discrete Logarithm Tool . . . 636

D.1 A wooden wax tablet with bronze stylus and eraser, originating from Egypt circa 600 AD. . . . 638
D.2 Skytale . . . 639
D.3 Lead Type . . . 640
D.4 Microdot . . . 643
D.5 Image Steganography . . . 643
D.6 Zoom in of a portion of the cat image. . . . 644
D.7 Primary Additive Colors . . . 644
D.8 (r, g, b) Color Values . . . 645
D.9 Colors $(248, 207, 170)$ and $(249, 206, 171)$ . . . 645

List of Tables

1.1 Morse Code 5

2.1 English Alphabet Numeric Coding 0–25 14
2.2 English Alphabet Numeric Coding 1–26 14
2.8 English Letter Relative Frequencies[5] 27
2.14 Vigenère Cipher Shift Coding 47
2.29 ADFGVX Morse Code 136
2.37 English Alphabet Numeric Coding 0–25 199
2.38 English Alphabet Numeric Coding 0–25 289

A.1 Brackets in Mathematica 423

B.1 Brackets in Maxima 468

D.2 Color Conversion Example (Encoding) 646
D.3 Color Conversion Example (Decoding) 646
D.4 Color Conversion Chart 647

Chapter 1

Cryptography

1.1 Introduction & Some Definitions

Let's start out with a few definitions,

Definition 1: *Cryptography is defined to be the process of creating ciphers such that when applied to a message it hides the meaning of the message.*

Definition 2: *Cryptanalysis is the process of breaking the cipher and discovering the meaning of the message.*

Definition 3: *Cryptology is the study of both Cryptography and Cryptanalysis.*

The terms Cryptography and Cryptology have become synonymous over the years. If we look at the definition of Cryptography a little closer, note the phrase "hides the *meaning* of the message". It does not say that it hides the *existence* of the message. So one of our assumptions in cryptography is that everyone knows that we are sending an encoded message. We make no attempt to try to conceal the fact that the message exists. The hiding of a message is known as Steganography,

Definition 4: *Steganography is the art of concealing a message, that is, its existence. In modern times this has become the practice of concealing a file, message, image, or video within another (seemingly harmless) file, message, image, or video.*

This set of notes does not concentrate on steganographic methods but there is a short discussion of some types of steganography later on.

1.2 Classical and Modern Cryptography

Classical cryptography refers to encryption methods that are no longer in use today because they are no longer secure. *Modern cryptography* refers to encryption methods that are currently in use. The division between classical and modern is a little disputed among the professionals but the primary force between the two is the invention of the computer. The classical methods were strong enough to withstand attacks by humans but are easily broken with high-speed computers, hence the reason they are not in use today.

There are many classical methods and there is a rich history of cryptography and cryptanalysis, in Chapter 2 on Classical Cryptography we will look at several of these methods and a little of their history. This list is not comprehensive, by far. The methods we discuss were simply chosen to exemplify some of the main concepts for the classical methods or to point out some historical milestones in cryptography.

If you wish to explore classical techniques further, please consider starting with the following references. For a good, and mostly non-mathematical, introduction to cryptography please take some time to read *The Code Book* by Simon Singh [55] or Fred Wrixon's book *Codes Ciphers & Other Cryptic & Clandestine Communication* [78]. A more encyclopedic treatment of classical cryptography can be found in David Kahn's book, *The Codebreakers: The Story of Secret Writing* [26] or his more recent book *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet* [27]. For something more mathematical, but very readable, please see the first couple chapters of *An Introduction to Cryptography with Coding Theory* by Wade Trappe and Lawrence Washington [62] or *An Introduction to Cryptography* by Richard A. Mollin [36].

1.3 Alice, Bob & Eve

Three people you need to know in the world of cryptography are Alice, Bob and Eve. Alice is the sender of the message, Bob is the receiver of the message and Eve is the eavesdropper. There are many other members to this family but for now, Alice, Bob and Eve will do. The figure pictured on the right is the classical setup of Symmetric-Key Cryptography, we will discuss the difference between Symmetric-Key Cryptography and Public-Key Cryptography shortly but they both have the same basic setup. The difference in them is in the way the keys are managed. Alice wants to send Bob a message, but she wants this message to be private so that only she and Bob know what the message says. The problem is that Alice cannot be certain that the transmission of the message will not be intercepted by someone else, Eve. So Alice takes her message which is human readable and we call plaintext, encrypts it with some encryption algorithm and an encryption key to get a non human readable message which we call ciphertext, and then sends the ciphertext to Bob. Bob takes the ciphertext and using a decryption algorithm and a decryption key decrypts the message back into plaintext and reads the message. Eve, having intercepted the message, has only the ciphertext and not the plaintext message. So if Eve wants to read the actual message, she needs to "crack the code" and decrypt the ciphertext.

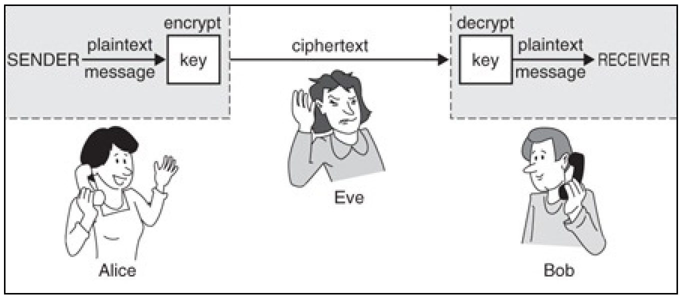


Figure 1.1: Alice, Bob & Eve

1.4 Keys and Key Spaces

In all cryptographic systems, there is an encryption algorithm and an encryption key as well as a decryption algorithm and decryption key. In some cases the encryption and decryption algorithms are identical and other times they are different. The same can be said for the encryption and decryption keys. We will see examples of these later on in the notes.

Definition 5: *A Key to a cryptographic system or algorithm specifies the particular transformation of plaintext into ciphertext during the encryption process, or vice versa during the decryption process.*

In the next section we will see that one of the underlying assumptions in cryptography is that our opponent knows what algorithm we are using to encrypt and to decrypt the message, this is Kerckhoffs' Principle. So the strength of our encryption and decryption system must be in the strength of the key. That is, the strength in either keeping the key secret or in the strength in not being able to find the key from other information, such as the ciphertext.

Even if the encryption and decryption algorithms and the encryption and decryption keys are different, they are obviously related, they must be inverse operations of each other or the system will not work. The decryption process must undo the encryption process. Also, if one knows the algorithm being used, as we assume, and if they have the encryption or the decryption key then they will be able to calculate the other key. So it is essential that either the keys are kept secret or that the process of calculating one key from the other is too difficult.

Every cryptographic algorithm has many keys that can be used with it. In most cases the number of keys is enormous and in some the number of keys is actually infinite. The set of all keys associated with a cryptographic algorithm is called the algorithm's *key space.*

Definition 6: *The Key Space of a cryptographic algorithm is the set of all encryption and decryption keys that can be used with the algorithm.*

On method of attacking a cryptographic system which always works, in theory, is a *brute-force* attack. In a brute-force attack the attacker simply tries all possible keys in the key space to see which one returns an understandable message. If you have a small key space then a brute-force attack is possible but if your key space is very large a brute-force attack is foiled. This does not imply that simply having a large key space will prevent someone from breaking your system, there may be other ways to attack the system, all it says is that a large key space is necessary.

1.5 Kerckhoffs' Principle

In the early 1800's the telegraph was invented, which allowed the electronic transmission of information over long distances. By the late 1800's this invention had matured to the point where it could be used practically for military purposes. Auguste Kerckhoffs actually Jean-Guillaume-Hubert-Victor-François-Alexandre-Auguste Kerckhoffs von Nieuwenhof (1835–1903) was a Dutch linguist and cryptographer. He was a professor of German at the at the École des Hautes Études Commerciales in Paris and in 1883 published *La Cryptographie Militaire* (*Military Cryptography*) in *le Journal des Sciences Militaires* (*Journal of Military Science*). In this article, he examined the current use of cryptography in the French army, specifically on the *field ciphers* that were used, and proposed many improvements. In this, he came up with six principles of practical cipher design,

1. The system should be, if not theoretically unbreakable, unbreakable in practice.
2. The design of a system should not require secrecy, and compromise of the system should not inconvenience the correspondents.
3. The key should be memorable without notes and should be easily changeable.
4. The cryptograms should be transmittable by telegraph.
5. The apparatus or documents should be portable and operable by a single person.
6. The system should be easy, neither requiring knowledge of a long list of rules nor involving mental strain.

This was written in 1883, and clearly not all of these principles would hold true today. The second one was restated by Claude Shannon in his 1949 paper *Communication Theory of Secrecy Systems* in the *Bell System Technical Journal* [54] as “the enemy knows the system being used”. This has become known as Kerckhoffs’ Principle.

Kerckhoffs’ Principle

In assessing the security of a cryptosystem, one should always assume the enemy knows the method being used.

1.6 Codes verses Ciphers

Most people when asked what the difference is between a code and a cipher is would probably say that they are the same. There is, however, a difference between the two.

Definition 7: *A Code is where words, phrases or letter combinations are replaced with other words, phrases or symbols, called codewords.*

Definition 8: *A Cipher is where every string of characters is encrypted, that is changed, by some algorithm.*

Probably the most famous code is Morse Code, where each letter or number is replaced by a series of dots and dashes that are converted to and from telegraph beeps.

Another example is the code-book used by the U.S. Navy in World War II. In a Navy communication, the message would first be coded before it was encrypted. For example, the Airfield at San Island, Midway was code named ALCATRAZ, Cuba was code named

Table 1.1: Morse Code

Letter	Code	Letter	Code	Letter	Code	Letter	Code
A	.-	J	.---	S	...	1	.----
B	-...	K	-.-	T	-	2	..---
C	-.-.	L	.-..	U	..-	3	...--
D	-..	M	--	V	...-	4	....-
E	.	N	-.	W	.--	5	.....
F	..-.	O	---	X	-..-	6	-....
G	--.	P	.--.	Y	-.--	7	--...
H	....	Q	--.-	Z	--..	8	---..
I	..	R	.-.	0	-----	9	----.

LANDSHARK and Onslow, Australia was code named PAINO. Most, but not all, of the Navy codewords represented places.

In a cipher, we simply use an algorithm to replace or encode any string of characters. A cipher does not use any linguistic structure of the language and it is applicable to any string of characters, even if it is meaningless. For example, GHKFTIYFKGK can be easily encoded with any encryption method but you will probably not find it in any code-book.

Although codes and ciphers are distinctly different they are frequently used together. So even if the enemy can decrypt the message, they would need the code-book to understand the entire message.

Along these lines, there is an area of mathematics called *coding theory* that deals with the science of altering a message so that when it is sent over a noisy channel we can correct any errors that may be encountered. That is, we assume that some of the message gets corrupted during transit and find mathematical ways to correct these errors. With modern cryptographic methods the alteration of a single bit could effect, and possibly destroy, the entire message. So the incorporation of these types of coding methods help to ensure that the ciphertext that makes it to the receiver is the correct ciphertext. We do not discuss coding theory in this set of notes but we will give a few examples here.

Example 1: Say we are sending a simple YES or NO message, we could do this by sending a 0 for no and a 1 for yes. Say we want to send YES. If we simply send the single bit 1 and the message is corrupted to a 0 the receiver will get and interpret the incorrect message of NO. Instead, say we send five copies of the message, that is we send the 5 bits 11111. Now if there is a single error in transmission, for example to, 11101 then the receiver would say, well there are four 1's and only one 0, so the message was probably 1 (or YES). In fact, if there were two transmission errors there would still be three 1's and only two 0's, so again the receiver would suspect that the actual message was 1 (or YES). It would take three or more errors of the five 1's before the receiver would decode the message incorrectly. Also, if there was one to four errors in transmission the receiver would know that there was an error in the transmission. It would take errors in all five bits for the receiver not to suspect that there was an error in the transmission. So with this simple repetition code we can detect up to four errors and correct up to two errors. Δ

The above code is, of course, fictitious but it does point out the ability to both detect errors and to correct errors in transmission by adding extra information to the transmission. In the above example, we added five times the information to be sent, which does not make this code very efficient. Sending 5 bits instead of one is no big deal, in fact, with digital transmissions we would be sending far more bits in a single packet of transmission, but if we are talking about sending petabytes of information, this code would become very cumbersome. Here are some examples of codes that are used in our everyday life.

Example 2: ASCII stands for American Standard Code for Information Interchange. It's a 7-bit character code where every single bit represents a unique character. In the ASCII system A is number 65, which in binary is 1000001. If the third bit is corrupted we would receive the number 1010001 which is 81 and corresponds to Q, so A was altered to Q with this single bit. Now if we add a bit to the end of these numbers that is the sum of the digits modulo 2, then A would be 10000010 and Q would be 10100011. Now if the third bit of A was corrupted we would have received 10100010, which we know is an error in transmission since if the first 7 were correct the eighth would be a 1 and not a 0. So we have detected an error. Unfortunately, with this simple system we could not correct the error since an alteration of any one of the transmitted bits would result in an error. Furthermore, if there had been two errors in the transmission, say to 10110010, we could not determine that this was a two-error A or a correct Y. So this code can detect a single error but not correct any. We call this last bit a parity bit or a check sum. Check sums are frequently used to verify the correctness of a transmission. In cases like this, when an error is detected the receiver contacts the sender and asks for a retransmission of the message. So these are usually implemented when there is two-way communication between the sender and receiver. Δ

Example 3: ISBN-10 stands for the 10 digit ISBN number given to any published book. ISBN stands for the International Standard Book Number, this form of book identification was adopted in 1970 and was replaced by a 13 digit version ISBN-13 in 2007. The 10 digit ISBN is broken down as follows,

1. The first two digits are the group or country identifier which identifies a national or geographic grouping of publishers.
2. The next four digits is a publisher identifier which identifies a particular publisher within a group.
3. The next three digits are the title identifier which identifies a particular title or edition of a title.
4. The last digit is a check digit which validates the ISBN.

The check digit is calculated so that when you take 10 times the first digit, 9 times the second, 8 times the third, down to 1 times the final (check) digit and then add them all up we get a number that is a multiple of 11. So for the ISBN 0131862391 (Wade Trappe and

Lawrence C. Washington. *Introduction to Cryptography with Coding Theory.* Prentice Hall, Upper Saddle River, NJ 07458, 2nd edition, 2006.) we have

$$0 \cdot 10 + 1 \cdot 9 + 3 \cdot 8 + 1 \cdot 7 + 8 \cdot 6 + 6 \cdot 5 + 2 \cdot 4 + 3 \cdot 3 + 9 \cdot 2 + 1 \cdot 1 = 154 = 11 \cdot 14$$

Another way to say this, using a little higher mathematics is that this weighted sum must add up to 0 modulo 11. Here, $154 \equiv 0 \pmod{11}$.

To calculate the check digit from the other information is fairly easy. Say we are working with the same ISBN number and we, the publisher, have determined the title number then we have 01-3186-239 so far. Replacing the last product in the above sum with c we get

$$0 \cdot 10 + 1 \cdot 9 + 3 \cdot 8 + 1 \cdot 7 + 8 \cdot 6 + 6 \cdot 5 + 2 \cdot 4 + 3 \cdot 3 + 9 \cdot 2 + c = 153 + c$$

Now using modular arithmetic (mod 11) we have

$$\begin{aligned} 153 + c &\equiv 0 \pmod{11} \\ 10 + c &\equiv 0 \pmod{11} \\ c &\equiv -10 \pmod{11} \\ c &\equiv 1 \pmod{11} \end{aligned}$$

So our check digit is 1, as expected, and the full ISBN is 0131862391. Since we are working modulo 11 we can have residues between 0 and 10, so there is a possibility that our check digit is a 10. In this case we use an X, so you may see the final digit of an ISBN-10 number being X, such as *Foundations of Cryptography: Volume 2, Basic Applications* by Oded Goldreich, ISBN-10: 052111991X.

The ISBN is another example of an error detection code that can detect the error in a single digit or the transposition of two digits. The ISBN can correct a single digit omission error, that is if we know the position of the missing digit we can calculate what it needs to be. Say for example, that we received the following ISBN number `0131_62391` where the underscore represents the position of the missing digit. If we set up our check equation like above, replacing the missing digit with a c, we have,

$$0 \cdot 10 + 1 \cdot 9 + 3 \cdot 8 + 1 \cdot 7 + c \cdot 6 + 6 \cdot 5 + 2 \cdot 4 + 3 \cdot 3 + 9 \cdot 2 + 1 \cdot 1 = 106 + 6c$$

Now setting up our modular equation,

$$\begin{aligned} 106 + 6c &\equiv 0 \pmod{11} \\ 7 + 6c &\equiv 0 \pmod{11} \\ 6c &\equiv -7 \pmod{11} \\ 6c &\equiv 4 \pmod{11} \\ c &\equiv 6^{-1} \cdot 4 \pmod{11} \\ c &\equiv 2 \cdot 4 \pmod{11} \\ c &\equiv 8 \pmod{11} \end{aligned}$$

Which is the missing digit. Δ

Error correcting codes are extremely common in today's world, satellite transmissions, bar codes, QR codes, ... all have error correction built into them. There are many good texts on coding theory, to name a few that are good for the undergraduate student, please see *Introduction to Coding Theory* by Ron Roth [50], *Introduction to Cryptography with Coding Theory* by Wade Trappe and Lawrence Washington [62] or *A First Course in Coding Theory* by Raymond Hill [23]. For a more advanced mathematical treatment of coding theory, have a look at *Introduction to Coding Theory* by J. H. van Lint [75]

1.7 Public verses Symmetric Key

In symmetric-key cryptography, the traditional method and the only one used before 1977, Alice and Bob would share an encryption and decryption key that only the two of them knew. When Alice wanted to send a message (plaintext) to Bob she would use the key to encrypt the message into ciphertext, she would then send the ciphertext to Bob where he would use the decryption key to decrypt the message back into plaintext and read what Alice had to say. In all transmissions we assume that a third person, Eve, could and does intercept the message. Now Eve does not have the key, she only has the ciphertext. It is her job to break the code either by finding the key and decrypting the message or by finding the meaning of the message without finding the entire key.

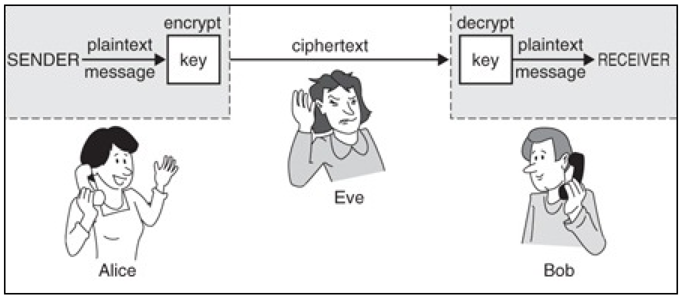


Figure 1.2: Symmetric-Key Cryptography

In public-key cryptography the setup is slightly different. There is no need to exchange a key between Alice and Bob before any messages are sent. In this scheme, Bob creates an encryption key and a decryption key. He keeps the decryption key private, he is the only one who knows the decryption key. He then publishes the encryption key on the Internet so that everyone knows the encryption key, including Alice and Eve. When Alice wants to send Bob a message she uses the encryption key Bob published, encrypts the message and sends it to Bob. Bob then uses the decryption key to decrypt the message. Now Eve has both the ciphertext of the message and the encryption key. The decryption key can, of course, be calculated

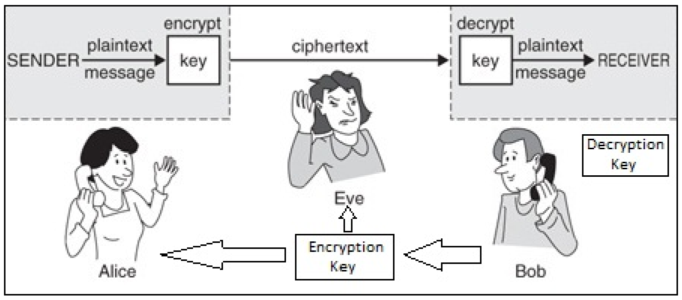


Figure 1.3: Public-Key Cryptography

mathematically from the encryption key, they must be inverses of each other in some sense. So what is stopping Eve from calculating the decryption key from the encryption key? We will look at this closer later on but the short answer is that whatever encryption/decryption process we have devised it has the feature that calculating the decryption key having only the encryption key is *Computationally Infeasible*. A process is computationally infeasible if, using the current technology, it would take too long to complete the calculation. As an example, when you make a credit card purchase on the Internet today your credit card number is encrypted using the RSA algorithm, which we will discuss later. The key to this algorithm is simply a couple numbers, one that is usually about 600 digits in length. In order to calculate the decryption key you need to factor this 600 digit number, or do something that is computationally equivalent to factoring the number. Using the fastest computers available with the fastest factoring algorithms known today (outside the walls of the NSA) it would take billions of years to factor the 600 digit number.

We will revisit this later, multiplication is an example of what is called a one-way function. A *one-way function* is one that is easy to do in one direction but to do its inverse is computationally infeasible. It is easy to multiply two 300 digit numbers together to get a 600 digit number, one can even do this by hand with enough patience, but to take a 600 digit number that is the product of two 300 digit numbers and factor it is extremely time-consuming. We will look at other examples of one-way functions in Chapter 3.

1.8 Stream verses Block Ciphers

Most cryptographic system fall into one of two categories of operation, block ciphers or stream ciphers.

Definition 9: *A Block Cipher is a cryptographic system that breaks the plaintext message into strings, called blocks, of fixed length n and enciphers (and usually transmits) one block at a time. The size n of a block is called the blocklength.*

The definition of a stream cipher can get a bit complicated but for now we will take a less mathematical and more intuitive approach.

Definition 10: *A Stream Cipher is a cryptographic system that encodes and decodes messages one character, number or bit at a time.*

For example, say we wanted to send the message CRYPTOGRAPHYISCOOL. A block cipher might break this message into blocks of length 6 to get CRYPTO GRAPHY ISCOOL, encrypt each of the three blocks separately and then transmit the three ciphertexts. The receiver will decrypt each of the three separate transmissions to get the original message. A stream cypher would encrypt each letter separately, the C then the R, and so on, and then send the encrypted scheme.

1.9 Attacks on a Cryptographic Method

Eve's job is to attack the system. She could have any one (or more) of several goals in mind, the most common are,

1. Read the message. This is simply finding out what Alice has said to Bob.
2. Find the key. If she can generate the key, then she can decrypt any message that was sent from Alice to Bob.
3. Alter Alice's message into another message. This would be done in a way that Bob would think that Alice had sent the altered message. So Eve could alter a love letter to Bob into a Dear Bob letter.
4. Masquerade as Alice. In this case, Eve would communicate with Bob in such a way that Bob thinks Eve is Alice.

The first two are called passive attacks and the last two are called active attacks. An *active attack* on a communications system is one in which the attacker changes the communication. They may create, forge, alter, replace, block or reroute messages. A *passive attack*, on the other hand, is one in which the attacker only eavesdrops. They read messages they are not supposed to see, but they do not alter the messages.

With these goals in mind, there are four standard or classical attacks on a cryptographic method, They are,

Ciphertext Only: Eve has only a copy of the ciphertext. This is the most difficult situation since it is the one where Eve has the least information. On the other hand, since we assume that any ciphertext is sent over a non-secure channel, we may assume that Eve has a copy of the ciphertext.

Known Plaintext: Eve has a copy of the ciphertext and the corresponding plaintext from the message or another message she suspects was encrypted with the same key. Obviously, she would not have the plaintext of the entire message or there would be no reason to decrypt the message, unless the goal was the key and not simply the message. For example, suppose that Alice always started her letters to Bob with "Dear Bob," then Eve knows the first several letters of message and has the corresponding ciphertext. This is called a *crib*, a portion of the plaintext message. On the surface it seems that knowing only seven letters would be of little use to Eve, but in many cases this would be enough information to construct the entire key. Even with very complicated ciphers, like the German Enigma, this amount of information was useful.

In one particular case, during World War II, shortly after 6 AM every morning the Germans sent an encrypted radio transmission for the day's weather. So it was certain that the word WETTER, the German word for weather, would be in that transmission. Also, with the consistency of military-type transmissions it was not too difficult to figure out where that word would be in each transmission. Also in World War II there was a German outpost in the Sahara Desert that every day sent the same message, that there was nothing new to report. In World War II, the primary method of encryption by the German forces was the Enigma, which had a different key for each day, so having a daily transmission crib was helpful in cracking the Enigma.

Chosen Plaintext: Eve temporally gains access to the encryption machine, she carefully chooses some plaintext messages, sends them through the machine to obtain the corresponding ciphertexts. Now she can do a known plaintext attack. If she chooses her plaintext messages well she will have an easier time in finding the key. This method could be broken into two separate methods, the other being an *Adaptive Chosen Plaintext* attack where the choice of plaintext messages is dependent on previously received plaintexts.

Chosen Ciphertext: Eve temporally gains access to the decryption machine, carefully chooses some ciphertexts, sends them through the machine to obtain the corresponding plaintexts. Now she can do a known plaintext attack. Again, if she chooses her ciphertext messages well she will have an easier time in finding the key. This method could be broken into two separate methods, the other being an *Adaptive Chosen Ciphertext* attack where the choice of ciphertext messages is dependent on previously received ciphertexts.

Different cryptographic methods have their own particular strengths and weaknesses. So some attacks work better for certain methods than others. As we mentioned before, the brute force attack is one that always works, in theory, but as long as the key space is large it will not work in practice.

1.10 The Main Objectives of Cryptography

Cryptography is not simply an academic curiosity of how to alter messages or information so that it cannot be read by an eavesdropper. It is about communication and information security in the real world. Cryptography has four main objectives:

Confidentiality of the Message Only the authorized recipient should be able to extract the content of the cipher. In addition, obtaining information about the content of the message (such as a statistical distribution of certain characters) should not be possible.

Message Integrity The recipient must be able to determine if the message was altered during transmission.

Authentication of the Sender The recipient should be able to identify the sender and verify if it was him who sent the message.

Irrevocability of the Sender It should not be possible to deny the authorship of the message.

Confidentiality of the message and message integrity are fairly self-explanatory. The sender and receiver both want to be certain that their communication is known only to them and that the recieved message was, in fact, the sent message.

Authentication of the sender is fairly self-explanatory as well. The receiver of the message wants to make sure that the message was not forged by a third party masquerading as the sender. Many cryptographic methods have the ability to attach or incorporate digital signatures to the transmission. These signatures usually take the form of an extra step or two in the algorithm, with possibly another key, an authentication key.

Irrevocability of the sender is where the sender cannot say that they did not send the message. This is closely related to authentication but slightly different. As an example, say that you digitally sign a document, such as a PDF document, and send it in an email. The recipient, upon seeing your signature, knows that the document is from you, and you cannot deny that you wrote the document. Another example where this and authentication is essential is in credit card transactions and gambling over the Internet.

Not all cryptographic systems achieve all of these objectives. In practical applications we usually use different algorithms to achieve these different goals. Furthermore, you may not want or care about some of these objectives for every transmission of a message. For example, in some cases you may want the sender of the transmission to be anonymous, or once you have authenticated the sender you may not care if they deny sending the message.

Chapter 2

Classical Cryptography

2.1 Introduction

Classical cryptography refers to encryption methods that are no longer in use today because they are no longer secure. *Modern cryptography* refers to encryption methods that are currently in use. The division between classical and modern is a little disputed among the professionals but the primary force between the two is the invention of the computer. The classical methods were strong enough to withstand attacks by humans but are easily broken with high-speed computers, hence the reason they are not in use today.

There are many classical methods and there is a rich history of cryptography and cryptanalysis, we will look at several of these methods and a little of their history in this chapter. This list is not comprehensive, by far. The methods we discuss were simply chosen to exemplify some of the main concepts for the classical methods or to point out some historical milestones in cryptography.

If you wish to explore classical techniques further, please consider starting with the following references. For a good, and mostly non-mathematical, introduction to cryptography please take some time to read *The Code Book* by Simon Singh [55] or Fred Wrixon's book *Codes Ciphers & Other Cryptic & Clandestine Communication* [78]. A more encyclopedic treatment of classical cryptography can be found in David Kahn's book, *The Codebreakers: The Story of Secret Writing* [26] or his more recent book *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet* [27]. For something more mathematical, but very readable, please see the first couple chapters of *An Introduction to Cryptography with Coding Theory* by Wade Trappe and Lawrence Washington [62] or *An Introduction to Cryptography* by Richard A. Mollin [36].

Classical cryptographic methods usually use either substitution, transposition or a combination of both. In the following definitions we use the term *unit* of plaintext or ciphertext, this refers to the smallest block of the message that is encoded or decoded. For example, if we are substituting each letter of the plaintext with a letter of ciphertext then our unit is a single character. On the other hand, if we encrypt each digram (pair of letters), as in the Playfair cipher, our unit would be digrams. We could also be encoding sets of three, four, . . . letters as in the Hill cipher, in which case the units would be the blocks we were using.

Most classical ciphers worked on messages that were written in a particular language, so they worked on letters or combinations of letters. It is possible that a message unit takes another form, such as, a bit (0 or 1) or numbers.

Definition 11: *A Substitution Cipher is a method where each plaintext unit is replaced with a ciphertext unit. Decryption is done by inverting the substitution.*

Definition 12: *A Transposition Cipher rearranges the units of the message in a different order, but the units themselves are left unchanged. Decryption is done by inverting the rearrangement.*

With classical substitution ciphers and ciphers that have a substitution component, we sometimes code the letters to numbers first so that we can do arithmetic operations on them. In many case these numeric values are then translated back to letters. The two most common codings are the 0–25 and 1–26, shown below.

Table 2.1: English Alphabet Numeric Coding 0–25

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M
Code	0	1	2	3	4	5	6	7	8	9	10	11	12
Letter	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Code	13	14	15	16	17	18	19	20	21	22	23	24	25

Table 2.2: English Alphabet Numeric Coding 1–26

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M
Code	1	2	3	4	5	6	7	8	9	10	11	12	13
Letter	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Code	14	15	16	17	18	19	20	21	22	23	24	25	26

The 0–25 encoding is used much more frequently since it fits better into modular calculations, which are used extensively in both classical and modern cryptographic methods.

With some classical methods and most modern methods, a more sophisticated method of coding the English message into numeric values is needed, we will discuss those when the time comes.

2.2 Monoalphabetic Substitution

2.2.1 Definitions and History

Definition 13: *A Monoalphabetic Substitution cipher is where each plaintext letter is replaced by one cyphertext symbol.*

The monoalphabetic substitution cipher was one of the first ciphers to be used and hence its origins are difficult to trace. We do, however, know about some of the historical milestones.

It appears that the first military use of a substitution cipher was by Julius Caesar[61]. In what is now commonly known as the Caesar Shift cipher, or just the Caesar cipher. According to Suetonius, Caesar simply shifted each letter in the message by three places. So A became D, B became E, and so on. When we reach the end of the alphabet we wrap around to the beginning.[56]

Atbash is a simple substitution cipher that reverses all of the characters in the alpahbet, so in English Z would be substituted for A, Y for B, X for C, and so on to A for Z. Atbash goes back to biblical times, it was used in several places in the Book of Jeremiah. It has been associated with the methodologies of Jewish mysticism's interpretations of Hebrew religious texts as in the Kabbalah.[24]

One of the earliest descriptions of encryption by substitution appears in the Kama-sutra, a text written in the 4th century A.D. by the Brahmin scholar Vatsyayana, but based on manuscripts dating back to the 4th century B.C. The Kama-sutra recommends that women should study 64 arts, including cooking, dressing, massage and the preparation of perfumes. The list also includes some less obvious arts, including conjuring, chess, bookbinding and carpentry. Number 45 on the list is mlecchita-vikalpa, the art of secret writing, advocated in order to help women conceal the details of their liaisons. One of the recommended techniques involves randomly pairing letters of the alphabet, and then substituting each letter in the original message with its partner.[56]

The Pigpen Cipher was used by Freemasons in the 18th Century to keep their records private. The cipher does not substitute one letter for another; rather it substitutes each letter for a symbol. The alphabet is written in the grids shown, and then each letter is enciphered by replacing it with a symbol that corresponds to the portion of the pigpen grid that contains the letter.[56] There are several different ways to implement the Pigpen Cipher, depending on which order you fill the four grids. As you can see from the images below, the cipher ring uses a different formulation then does our grid.

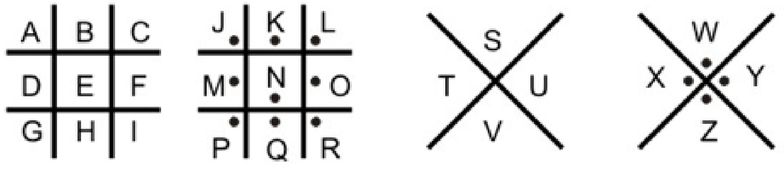


Figure 2.1: Pigpen Cipher

So the alphabet using the Pigpen Cipher would be

⌋⊔⌊⊐□⊏⌉⊓⌈ ⌋̇⊔̇⌊̇⊐̇⊡⊏̇⌉̇⊓̇⌈̇ V><Λ V̇>̇<̇Λ̇

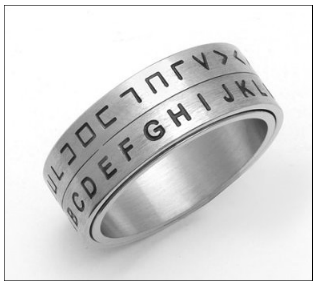


Figure 2.2: Pigpen Cipher Ring

No discussion of substitution ciphers would be complete without mentioning the great fictional character Sherlock Holmes in *The Adventures of the Dancing Men*, by Sir Arthur Conan Doyle. Although this is a purely fictional use of a substitution cipher, it is well-known, and signifies one of the earlier uses of cryptography in general entertainment literature.

In the story, Mr. Hilton Cubitt hires Sherlock Holmes to decipher what appear to be simple stick figures on a piece of paper, figures that resemble dancing men. When Mr. Cubitt visits Holmes and Watson at 221B Baker Street he explains that he had been married to his wife, Elsie Patrick, for about a year. Also that she is an American and he knew little about her past before she moved to England. In fact, Elsie had requested that he never ask about her past, a past that she wanted to forget. She had recently received an upsetting letter from the United States and after reading it she quickly burned it. Shortly after that the dancing men figures were found written on a window sill in chalk. Cubitt told Elsie of the markings, which upset Elsie further, but he had washed them off before Elsie saw them. Shortly after that, a piece of paper with more dancing figures appeared on a sundial in the garden. When Cubitt found it and showed it to Elsie, she fainted.

Holmes tells Cubitt to return to his home, Ridling Thorpe Manor in Norfolk, and send home any more dancing men figures that appear. Cubitt does so on several occasions. After Holmes see the last message that Cubitt sends to him, he and Watson rush off to Ridling Thorpe Manor. When they get there they are too late, Mr. Cubitt is dead and Elsie is in serious condition, both from gunshot wounds. It seemed that Elsie had shot and killed her husband and then attempted to commit suicide.

Holmes investigates and discovers the existence of a third person at the scene of the crime. Further investigation leads him to a Mr. Abe Slaney who is currently residing at Elriges Farm. In a telegram response to Holmes's inquiry to the New York police force regarding Mr. Slaney, he finds out that Slaney is "the most dangerous crook in Chicago". Holmes sends a message, encoded with dancing men, to Slaney at Elriges Farm. Slaney, thinking that Elsie sent the message, arrives at Ridling Thorpe Manor only to discover Holmes, Watson and Inspector Martin waiting for him. Abe is arrested, and he confesses to everything, as criminals in Doyle's stories usually do. Abe gets life in prison and Elsie eventually recovers from her injuries.

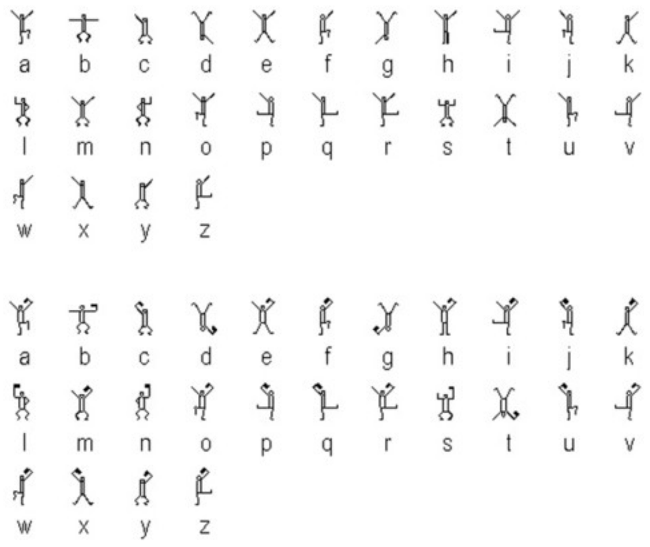


Figure 2.3: Dancing Men

Figure 2.4: Dancing Men: Sherlock Holmes At The Museum Of London

From a cryptographic point of view Holmes uses two standard techniques, frequency analysis and a suspected crib. A crib is a piece of plaintext that the analyzer either knows is in the ciphertext or suspects is in the ciphertext. Since these messages were so disturbing to Cubitt's wife Elsie, Holmes suspects that her name is in the ciphertext somewhere. Holmes also knows a bit about frequency analysis, that is, English language letters show up with different frequencies, E being the most frequent. Lucky for Holmes that Elsie begins and ends with an E. Of course, Holmes solves the substitution cipher with very little information. Normally it would take a bit more ciphertext to make the conclusions he made, but then again he is Sherlock Holmes and this is to be expected. Holmes also does more than simple solve the cipher, he takes an active attack when he sends a message to Slaney, masquerading as Elsie.

The first known explanation of frequency analysis, and hence the first use of cryptanalysis, was in the 9th century by Al-Kindi, in *A Manuscript on Deciphering Cryptographic Messages.*[2] A thorough study of the Qur'an showed that Arabic had a characteristic letter frequency, like most other European languages. Its use spread to Europe by the time of the Renaissance and in 1474, Cicco Simonetta wrote a manual on deciphering encryptions of Latin and Italian texts.[26]

Starting at about the time of the Renaissance, substitution ciphers were usually combined with some type of coding. Names of people or locations were given code names before encryption so even if the code was broken the key people and locations were still a mystery. A combination of coding and substitution ciphers is sometimes referred to as a nomenclator or nomenclature.

Caesar Shift

The Caesar Shift cipher is where each plaintext letter is replaced by the letter k letters further along in the alphabet. When we reach the end of the alphabet we wrap around to the beginning.

To encrypt a message we simply shift each character down the alphabet by k places, wrapping around at the end of the alphabet. Decryption is the same process except that we move each character up k places.

Example 4: Say we are working with just the uppercase English alphabet and we let our shift $k = 3$. Then A would get cent to D, B to E, and so on up to Z that is sent to C.

Plaintext	**Ciphertext**
A	D
B	E
C	F
D	G
E	H
F	I
G	J
H	K
I	L
J	M
K	N
L	O
M	P
N	Q
O	R
P	S
Q	T
R	U
S	V
T	W

Plaintext	**Ciphertext**
U	X
V	Y
W	Z
X	A
Y	B
Z	C

So if we wish to encrypt the message, ATTACKATDAWN, A is sent D, T to W and so on to get, DWWDFNDWGDZQ. For decryption, D goes to A, W to T, and so on. Δ

If we consider the keyspace of this cipher, it is not very large. There can be only as many shifts as there are characters in the language. So for English, using only the uppercase alphabet, we get a keyspace of size 26, specifically, $k = 0, 1, 2, \ldots, 25$. One can argue that a zero shift is not a very intelligent possibility and hence our keyspace is only of size 25. In either case, this is a very small keyspace and using a brute force attack is a perfectly fine approach to the cryptanalysis of a Caesar shift cipher.

Here we are speaking of a ciphertext only attack. If all we have is the ciphertext then we could try all possible shifts until the message shows itself. What about the other types of attacks?

Ciphertext Only: Brute-force on the 26 keys.

Known Plaintext: Here all Eve needs is a single character correspondence. Since all the shifts are the same, if she finds one shift sh has them all.

Chosen Plaintext: Here all Eve needs to do is send A into the cipher machine, the output will show her the shift amount. Of course, any letter will do, no particular need for sending A.

Chosen Ciphertext: Here again all Eve needs to do is send A into the decipher machine, the output will show her the shift amount. Again, any letter will do.

It is fairly clear that the Caesar shift cipher is not a very secure encryption technique, but at the time of Caesar, most of the population was illiterate in the first place. So sending an unencrypted message was nearly as secure.

Affine Substitution

The Affine Substitution cipher is where we apply an affine transformation to the characters of the message. An affine transformation, in general, is a linear transformation followed by a shift. When we apply this idea to a substitution cipher we get the following.

1. Code the plaintext message using a 0–25 scheme, or 0–n if you are working with a language that does not have 26 characters.

2. Choose a multiplier m and a shift k. The value m must be chosen so that $\gcd(m, 26) = 1$, or in general, $\gcd(m, n) = 1$.
3. For each plaintext number in the message stream, p, replace it by, $m \cdot p + k \pmod{26}$, or in general, $m \cdot p + k \pmod{n}$.
4. Code the numbers back to letters using the 0–25 scheme, or 0–n, for the ciphertext message.

To decrypt the message we need to reverse the process, if we let c represent the ciphertext number that was produced from the plaintext number p, then,

$$\begin{aligned} c &\equiv m \cdot p + k \pmod{n} \\ c - k &\equiv m \cdot p \pmod{n} \\ m^{-1}(c - k) &\equiv p \pmod{n} \end{aligned}$$

So the formula for p is, $m^{-1}(c - k) \pmod{n}$, where m^{-1} represents the inverse of m modulo n, so in the case of English, modulo 26. If you are not familiar with modular inverses, consult a textbook on number theory, classical cryptography or look in the appendices of these notes for a description of how to calculate them. This will also explain why we must have $\gcd(m, n) = 1$.

1. Code the ciphertext message using a 0–25 scheme, or 0–n if you are working with a language that does not have 26 characters.
2. For each ciphertext number in the message stream, c, replace it by, $m^{-1}(c - k) \pmod{26}$, or in general, $m^{-1}(c - k) \pmod{n}$.
3. Code the numbers back to letters using the 0–25 scheme, or 0–n, for the plaintext message.

Example 5: If we choose our multiplier $m = 5$ and our shift $k = 3$ then our affine transformation is $5 \cdot p + 3 \pmod{26}$. So A, which is 0, is sent to 3, which is D. The character B, which is 1, is sent to 8, which is I, C to N, and so on.

Plaintext	**Ciphertext**
A	D
B	I
C	N
D	S
E	X
F	C
G	H
H	M
I	R

Plaintext	Ciphertext
J	W
K	B
L	G
M	L
N	Q
O	V
P	A
Q	F
R	K
S	P
T	U
U	Z
V	E
W	J
X	O
Y	T
Z	Y

So our message ATTACKATDAWN would be encrypted as DUUDNBDUSDJQ. For decryption, we could simply use the above table, but if we wanted to calculate the decryption formula, $5^{-1}(c-3) \pmod{26}$ we do the following. Since $\gcd(5, 26) = 1$ we know that 5 has an inverse modulo 26. That is, there is a number t with $1 \equiv 5 \cdot t \pmod{26}$. The value of t is the inverse of 5. Either using the extended Euclidean algorithm or simple trial and error, we see that $t = 21$, so our decryption formula is $21 \cdot (c-3) \pmod{26}$. Δ

If we consider the size of the keyspace here, it is larger than the Caesar shift, but it would still be manageable by brute force, even without a calculation device. Before we discuss the keyspace we should consider what a key to this cipher looks like. The two options open to Alice and Bob are the choices of m and k, so a key would be an ordered pair of the multiplier and the shift, (m, k).

The number of possible m values is the Euler Totient, or Euler Phi function on 26, $\phi(26) = 12$. This is easy to calculate since m cannot be even nor can it be 13, this leaves 12 possibilities. There are 26 possibilities for k which gives a grand total of $12 \cdot 26 = 312$ possible keys.

So for a ciphertext only attack, brute force is a little more painful but with a little time it is doable, even by hand. What if we had some more information, for example, what if we knew a couple character correspondences.

Example 6: Say we intercept an affine cipher LSVVEUVKA, and from some extra espionage we also know that the first two letters of the plain text are IN. So I encrypts to L and

N encrypts to S. This gives us the following modular equations,

$$\begin{aligned} 11 &\equiv m \cdot 8 + k \pmod{26} \\ 18 &\equiv m \cdot 13 + k \pmod{26} \end{aligned}$$

If we subtract the first congruence from the second we get

$$7 \equiv m \cdot 5 \pmod{26}$$

So

$$7 \cdot 5^{-1} \equiv 7 \cdot 21 \equiv 17 \equiv m \pmod{26}$$

So we know that $m = 17$. If we substitute this into the first equation and solve for k we get,

$$11 - 17 \cdot 8 \equiv 5 \equiv k \pmod{26}$$

Hence our key is $(17, 5)$, using this we can decrypt LSVVEUVKA into the plaintext message INEEDHELP, which most of my students and several of my colleagues will tell you is a very true message. Δ

What made this possible to calculate so easily is the fact that after we took the difference between the two congruences, we got a number times m that was invertible modulo 26, that is, 5. what if this did not happen? Lets consider the following example.

Example 7: Say we intercept an affine cipher LSVVEUVKA, and from some extra espionage we also know that the L came from an I and the V came from an E. So I encrypts to L and E encrypts to V.

This gives us the following modular equations,

$$\begin{aligned} 11 &\equiv m \cdot 8 + k \pmod{26} \\ 21 &\equiv m \cdot 4 + k \pmod{26} \end{aligned}$$

If we subtract the second congruence from the first we get

$$16 \equiv m \cdot 4 \pmod{26}$$

Since 4 is not invertible modulo 26, we cannot proceed as we did in the last example. If we try to brute force solve the last congruence we see that we get the two solutions $m = 4, 17$. Since 4 is not invertible modulo 26 and 17 is, the value of m is 17. From there a substitution will find $k = 5$ as it did in the previous example. So even though it was a little more difficult it was better than brute force on 312 keys. Another note, there are more procedural ways to solve linear congruences when you cannot invert needed numbers. Δ

So in summary, we could approach the cryptanalysis of the affine cipher as follows,

Ciphertext Only: Brute-force on the 312 keys.

Known Plaintext: Here all Eve needs are two different character correspondences, she would then proceed as in the above examples.

Chosen Plaintext: Here all Eve needs to do is send two different letters into the cipher machine, the output along with the above examples will enable her to find the key. She should take care to choose two letters that will make the calculations of m and k easy.

Chosen Ciphertext: Here again, all Eve needs to do is send two different letters into the decipher machine, the output along with the above examples will enable her to find the key.

Kama-Sutra Cipher

The Kama-Sutra cipher is where we pair up letters and we do encoding and decoding by switching the letters in each pair.

Example 8: Say we do the following pairing of letters,

Plaintext	Ciphertext
A	G
B	L
C	Y
D	O
E	V
F	W
H	T
I	Z
J	Q
K	N
M	X
P	S
R	U

When we encode, or decode, we will replace all A's with G and G's with A, B's with L and L's with B, and so on. So with this pairing ATTACKATDAWN would be encrypted as GHHGYNGHOGFK. Δ

The keyspace here is much larger than in our previous examples, much too large for a brute force attack. The size of the keyspace depends on the implementation of the cipher. I have seen several different algorithms for the pairing of letters, the two most common methods are as follows.

1. Pair A with any of the remaining 25 letters. Pair the next unused letter with any of the 23 remaining letters. Then pair the next unused letter with any of the 21 remaining letters, and so on. For example, we could pair A and C, then B and R, then D and J, and so on.

2. Take the first 13 letters and pair them randomly with the last 13 letters. So A could

be paired with W but not with F.

As for the keyspace, in the first case when we pair A with a letter, we have 25 choices, the next letter to be paired has only 23 choices since A is used, A's pair is used and the next letter is used. The third choice has 21 possible letters to choose from since we have used two pairs and another letter. So for pairing algorithm number 1 we have the following number of keys,

$$25 \cdot 23 \cdot 21 \cdot 19 \cdot 17 \cdot 15 \cdot 13 \cdot 11 \cdot 9 \cdot 7 \cdot 5 \cdot 3 \cdot 1 = 7905853580625$$

This is known as a double factorial, where you start with a number and multiply every second number until you get down to 1 or 2. So the above calculation is 25!!. Both Mathematica and Maxima have a built-in function for the double factorial and both use the `25!!` syntax.

If we use the second algorithm for pairing, then A has 13 choices for a pair, B has 12, C has 11, and so on. So in this case, the keyspace size is 13! = 6227020800.

With either algorithm, brute force checking of 6,227,020,800 or 7,905,853,580,625 keys would not be feasible. We will discuss frequency analysis later in this section which is a general method for doing a ciphertext only attack on a Kama-Sutra cipher. The other attacks re fairly obvious.

Ciphertext Only: Frequency analysis, to be discussed later.

Known Plaintext: If there are enough character correspondences Eve will be able to fill out the key. If there are enough correspondences, Eve could also guess at the remaining ones by substituting what she knows into the ciphertext and looking for recognizable words.

Chosen Plaintext: Eve could send in the 26 letters and she will get the entire key.

Chosen Ciphertext: As with the chosen plaintext attack, Eve could send in the 26 letters and she will get the entire key.

Random Substitution

This is the most general substitution method. Here A is assigned to any of the 26 possible letters, including A. B is assigned to any of the 25 remaining letters, and so on.

Example 9: Consider the following random substitution cipher.

Plaintext	**Ciphertext**
A	O
B	K
C	G
D	U
E	S
F	Y
G	A

Plaintext	**Ciphertext**
H	V
I	E
J	B
K	L
L	P
M	N
N	M
O	I
P	H
Q	T
R	X
S	F
T	Z
U	J
V	Q
W	D
X	W
Y	C
Z	R

So here, A is encoded as O, B as K and so on. Decryption is simply done in the reverse order, A is decoded as G, B as J and so on. So our message ATTACKATDAWN is encoded as OZZOGLOZUODM. Δ

The size of the keyspace here is easy to calculate, A has 26 possible choices, B has 25, C has 24 and so on. So in all there are 26! = 403291461126605635584000000 possible keys. This is far too many for a brute force attack, even for a modern computer. If a computer could check 1,000,000,000 keys every second, which is well outside the capabilities of a personal computer, it would take a little over 12,779,885,129 years to check every key in the keyspace.

As we pointed out above, there is a method called frequency analysis that will allows us to crack a general substitution cipher if we have enough ciphertext to work with. But what if there was no short cut method? What if we really had to check every possible key? In that case, a simple substitution cipher would not be breakable by a personal computer and it would be an enormous undertaking even for a supercomputer. Let's think about what would need to be done for a moment. The computer would need to generate a key, decode the ciphertext with that key and then check the decoded message for recognizable words. The first steps are quick but the last step of checking for recognizable words can be very time consuming. You would need to take blocks of the decoded message and search a database of words in your language for a match, even with fast searching algorithms this could take a long time. So even for a supercomputer, 1,000,000,000 keys per second may be an overestimate.

Another point to note here, although frequency analysis can get around brute force key checking for the substitution cipher, what if we devised a cryptographic method that had a large keyspace and did not have an alternative form of attack? We would need to check all of the keys, and if the keyspace was large enough, this would be computationally unfeasible.

Keyword Substitution

Keyword substitution does not seem to have been used much with substitution ciphers but it was generally used for later ciphers such as the Vigenére, Playfair, and ADFGVX ciphers. A keyword is simply an easy to remember word that defines some portion of the key to the cipher, the rest of the key is created by some algorithm using the remaining letters. The way it would work for the substitution cipher is as follows.

Example 10: We would choose a keyword, say CAESAR. We would create the substitution key by sending A to C, B to A, C to E, D to S, and E to R. We would skip the repeated A in CAESAR since we do not want to send two different characters to the same letter. We would then fill out the rest of the key by using the unused letters in order. So F is sent to B, G to D, and so on.

Plaintext	Ciphertext
A	C
B	A
C	E
D	S
E	R
F	B
G	D
H	F
I	G
J	H
K	I
L	J
M	K
N	L
O	M
P	N
Q	O
R	P
S	Q
T	T
U	U
V	V
W	W
X	X
Y	Y
Z	Z

So our message of ATTACKATDAWN would be encrypted as CTTCEICTSCWL. Δ

As you can see from the above example, this type of key construction does not look very good. The end of the key does not change many of the letters and in the middle we are looking at a simple shift. These are two huge weaknesses to the key and probably why this method was not widely implemented. Although, it is easier to remember a keyword than it is 26 random letters.

Again, frequency analysis will work for a ciphertext only attack, in this case it may be easier since we have a good idea what the end of the key will look like.

The keyspace size is difficult to estimate. We can use any word in the language but we also might use the name of a location, say FRANCE, or the name of a famous person, etc. Also, some words will produce the same key, such as fed and feed. Suffice it to say, the keyspace will be too large for a brute force attack.

2.2.2 Cryptanalysis

Say we know that we have a substitution cipher and all we have is the ciphertext, no other information to go on. We don't even know what type of substitution algorithm was used. The extra bit of information is actually buried in the language itself.

Modern languages, especially character based languages, reuse characters to create words, and this reuse of characters is not uniform. For example, in English, the letter E is used most often and significantly more often than any other letter. With a monoalphabetic substitution, if E was sent to W then we would expect that W would show up most often in the ciphertext, and with enough ciphertext to go on, it should show up about the same percentage of the time. If we take this a bit further, pairs of letters, called digrams have different frequencies as well, as do triples of letters, called trigrams. Below is a chart, and bar chart, of the relative frequencies of letters in standard English text. Below that is a list of digrams and trigrams in order of their frequencies. After that is a list of the most common words in English text, also in order of frequency.

Table 2.8: English Letter Relative Frequencies[5]

Letter	e	t	a	o	i	n	s	h	r	d	l	c	u
Freq. %	12.7	9.1	8.2	7.5	7	6.7	6.3	6.1	6	4.3	4	2.8	2.8

Letter	m	w	f	g	y	p	b	v	k	j	q	x	z
Freq. %	2.4	2.3	2.2	2	2	1.9	1.5	1	0.8	0.2	0.1	0.1	0.1

Digram Frequency:
th he in er an re on at en nd st or te es is ha ou it to ed ti ng ar se al nt as le ve of me hi ea ne de co ro ll ri li ra io be el ch ic ce ta ma ur om ho et no ut si ca la il fo us pe ot ec lo di ns ge ly ac wi wh tr ee so un rs wa ow id ad ai ss pr ct we mo ol em nc rt sh po ie ul im ts am ir yo fi os pa ni ld sa ay ke mi na oo su do ig ev gh bl if tu av pl wo ry bu

Trigram Frequency:
the ing and ion ent hat her tio tha for ter ere his you thi ate ver all ati ith rea con wit are

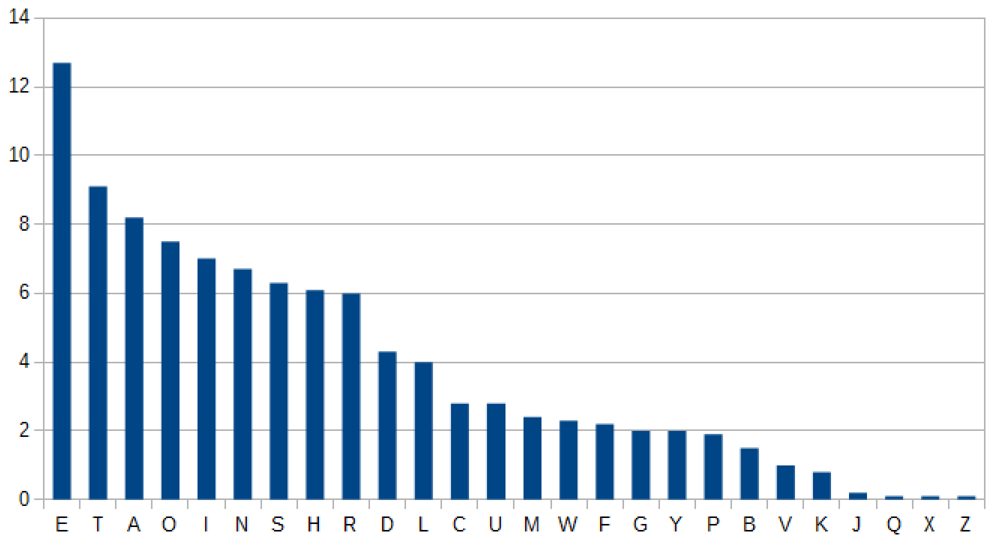


Figure 2.5: English Letter Relative Frequencies Sorted by Frequency

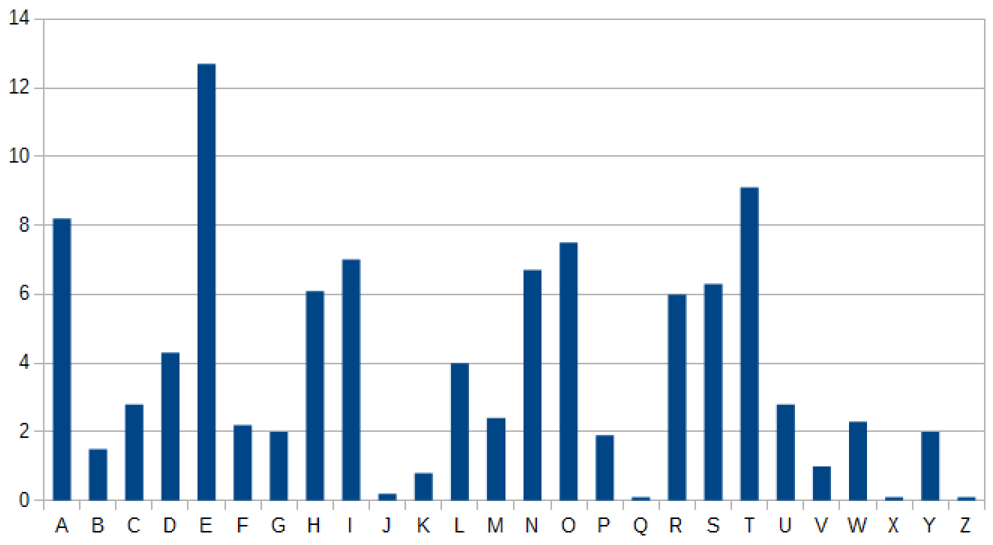


Figure 2.6: English Letter Relative Frequencies Sorted by Letter

ers int nce sta not eve res ist ted ons ess ave ear out ill was our men pro com est ome one ect ive tin hin hav ght but igh ore ain str oul per sti ine uld ste tur man oth oun rom ble nte ove ind han hou whi fro use der ame ide ort und rin cti ant hen end tho art red lin

Word Frequency:
the of to and a in that is i it for as with you on was be he this not have are at if but by from his or they an which we all said one had will my so has their more there no what were when would your her can been she out who some do about me up new him other them time than into like only now its then may any how could me two our very these end first just people

after get also even most should return over such many see well know much struct good before same long way because make those think must where down int here being us little did last

So how do we use this information? The idea is to find the letter correspondence for the key. The first step would be to take the ciphertext and count the frequencies of all the letters. Assuming that we have a monoalphabetic substitution, these frequencies should line up relatively well with the above charts. The more ciphertext we have the better the frequencies should match, in general.

If you look at the relative frequency charts of English letters you will see that E is well above the rest, but then when we hit T, A, O, I, N, S, H and R. These form a gradually decreasing sequence of letters, it is not too difficult to have a plaintext that mixes these up a bit, especially if you sre not working with a lot of ciphertext. From there on out we have letters that appear less than 5% of the time. So using just single letters will usually only get us a letter or two, three or four if we are very lucky.

Once we have E and T, or at least we think we do, we start looking at digrams and trigrams. For example, the letter H is not a high frequency character so frequency analysis on single characters will probably not pick it out. On the other hand, TH and HE are the two most frequent digrams, so if we see a letter that is paired with the ciphertext for T or the ciphertext for E, then it is probably the ciphertext for H. Furthermore, THE is the most frequent trigram, again, given that we know T and or E we can probably pick out H. Also, I and N both have a relative frequency close to 7% and the digram IN is common, so looking at a high frequency digram with each of the two characters being at about 7% relative frequency could get us a couple more letters. The same is true for A, N and AN. Another observation is that both RE and ER are common digrams, with E probably known, R should be easy to spot.

Once we have a handful of possibilities we can start playing a guess and check game, something I like to call playing Wheel of Fortune. I would assume that most readers have seen the game show Wheel of Fortune, but if not, contestants try to solve a short word puzzle by guessing at letters in the puzzle. When a new letter in the puzzle is guessed, all occurrences of that letter are shown, leaving a partial solution to the puzzle. At first, contestants choose high frequency consonants like T, S, N, H, and R. They then buy vowels, yes they cost money, usually E is bought first followed by A, O, or I, depending on the known puzzle contents. When the puzzle has a sufficient number of letters in it the player can opt to solve the puzzle.

We do essentially the same thing here, we will fill in the letters we know, or think we know, and then try to form words by guessing at what letters might be until we build the key and decipher the message.

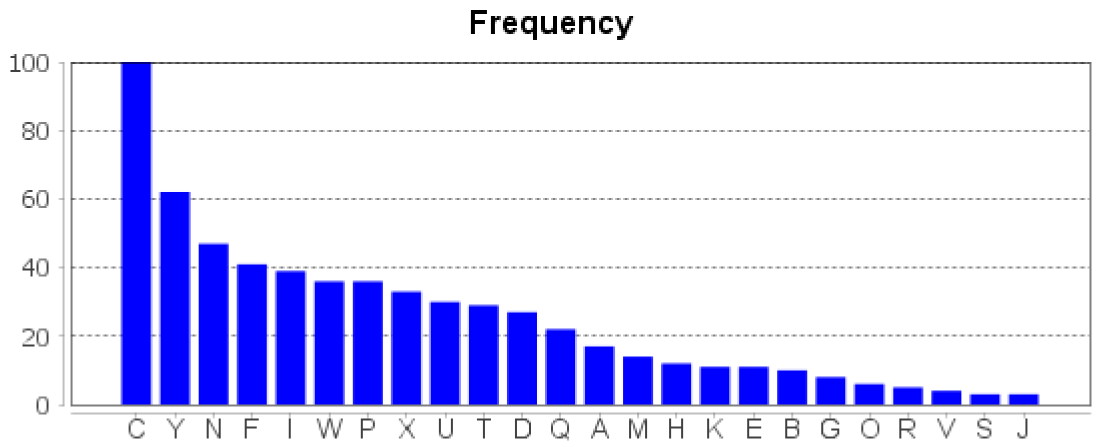
Example 11: Say we intercepted the following message, and we know that it was encrypted with a substitution cipher, but we do not know what type of substitution was used. After trying shifts we have concluded that it is not a Caesar shift cipher.

```
PTCPEYXCMPICUPUADWIQIBUYPNBNYCMNFNYXCNAKNYFYAYFPTQ
FUXCIFYFNQPMMPTDBANCHFTYXCUAVVDCNCQYFPTPEYXCGCCSCT
HTCGNUWUCINEPICJWMUDCYXCUIFTQFUDCFNNFMUDCCWQXDCYYC
```

```
IFTYXCWDUXWKCYFNICUDWQCHKBWTPYXCIPIUPNNFKDBYXCNWMC
DCYYCIMPICUICQFNCDBWUCIMAYWYFPTPEYXCWDUXWKCYFNQXPN
CTWTHWUUDFCHYPYXCUDWFTYCJYFTYXCUAVVDCUWRCNYXCNUWQC
NKCYGCCTYXCGPIHNWICANAWDDBUICNCIOCHGXFQXFNWKFRWHOW
TYWRCYPYXCNPDOCINFTQCSTPGDCHRCPEGPIHNYIAQYAICKCQPM
CNOCIBANCEADXPGCOCIYPFTQICWNCNCQAIFYBFYFNKCYYCIYPP
MFYYXCNUWQCNYXCNXFEYWTHWEEFTCQFUXCINWICCJWMUDCNPEN
AKNYFYAYFPTQFUXCINYXCOFRCTCICWTHXFDDQFUXCINWICTPYN
FTQCYXCBUCIMAYCKDPQSNPEDCYYCINIWYXCIYXWTPTCDCYYCIW
YWYFMC
```

Analyzing the ciphertext we get the following information. First note that there are over 600 characters in this ciphertext, which under normal circumstances, is sufficient for frequency analysis. The chart below shows that it is likely that $E \longrightarrow C$ and that $T \longrightarrow Y$. Looking at the top 10 digrams and trigrams as well, we see that in the digrams XC and YX are the top two and in the trigrams we have YXC topping the list, and is way out in front. This supports that C ad Y are E and T, and furthermore it strongly suggests that $H \longrightarrow X$. Also looking at the digrams, I and N follow C fairly often, and XCN, XCI and CIN are the next three trigrams as well. This would suggest that I or N was R, N, or S. Since HER is a common trigram, I or N is probably R. Since IC is also a prominent digram, I being R is more probable. If we put in just these four letters and decrypt what we can, we have.

```
__E__THE__RE______R_R__T____TE____THE____T_T_T____
__HER_T____________E___THE_____E_E_T_____THE_EE_E_
__E_____ER___RE_____ETHE_R______E_______EE__H_ETTE
R__THE___H__ET__RE____E_______THER_R_________THE___E
_ETTER__RE_RE___E____ER__T_T_____THE___H__ET___H__
E_________E_T_THE_____TE_T__THE_____E___E_THE____E
__ET_EE_THE__R___RE_________RE_ER_E__H__H__________
_T__ET_THE____ER____E_____E__E____R__TR__T_RE_E___
E__ER___E___H__E_ERT____RE__E_E__R_T__T___ETTERT__
__TTHE____E_THE_H__T________E___HER__REE_____E____
___T_T_T______HER_THE___E_ERE___H______HER__RE__T_
___ETHE__ER__TE__________ETTER_R_THERTH___E_ETTER_
T_T__E
```



Characters	Frequency	Relative Frequency
XC	23	0.03801652892561983
YX	20	0.03305785123966942
CI	19	0.03140495867768595
CN	14	0.023140495867768594
YF	12	0.019834710743801654
IC	12	0.019834710743801654
DC	11	0.01818181818181818
CY	11	0.01818181818181818
FT	9	0.01487603305785124
CU	8	0.013223140495867768

Characters	Frequency	Relative Frequency
YXC	19	0.03145695364238411
XCN	6	0.009933774834437087
XCI	6	0.009933774834437087
CIN	6	0.009933774834437087
QFU	5	0.008278145695364239
CYY	5	0.008278145695364239
DCY	5	0.008278145695364239
YCI	5	0.008278145695364239
YYC	5	0.008278145695364239
YFN	4	0.006622516556291391

If we look at the end of the message

Input	Output
I	R
W	

Input	**Output**
Y	T
X	H
C	E
I	R
Y	T
X	H
W	
T	

It appears that this phrase is `RATHER THAN`, which would fit for W being A and T being N. Substituting these in we get,

```
_NE__THE__RE_____AR_R__T____TE____THE____T_T_T__N_
__HER_T_______N____E__NTHE_____E_E_T__N__THE_EE_EN
_NE___A_ER___RE_A___ETHE_R_N____E_______EEA_H_ETTE
R_NTHEA__HA_ET__RE__A_E___AN_THER_R________THE_A_E
_ETTER__RE_RE___E__A_ER__TAT__N__THEA__HA_ET___H__
ENAN_A____E_T_THE__A_NTE_T_NTHE_____E_A_E_THE__A_E
__ET_EENTHE__R__ARE___A____RE_ER_E__H__H__A___A__A
NTA_ET_THE____ER__N_E_N___E__E____R__TR__T_RE_E___
E__ER___E___H__E_ERT__N_REA_E_E__R_T__T___ETTERT__
__TTHE__A_E_THE_H__TAN_A___NE___HER_AREE_A___E____
___T_T_T__N___HER_THE___ENEREAN_H______HER_AREN_T_
_N_ETHE__ER__TE_________ETTER_RATHERTHAN_NE_ETTERA
TAT__E
```

A little further up from the end we now see, `WICTPY` going to `AREN_T`, which would suggest that P is O.

Input	**Output**
W	A
I	R
C	E
T	N
P	
Y	T

Putting this in we have,

```
ONEO_THE_ORE_O___AR_R__TO___TE____THE____T_T_T_ON_
__HER_T___O__ON____E__NTHE_____E_E_T_ONO_THE_EE_EN
_NE___A_ER__ORE_A___ETHE_R_N____E_______EEA_H_ETTE
R_NTHEA__HA_ET__RE__A_E___ANOTHEROR_O______THE_A_E
```

```
_ETTER_ORE_RE___E__A_ER__TAT_ONO_THEA__HA_ET___HO_
ENAN_A____E_TOTHE__A_NTE_T_NTHE_____E_A_E_THE__A_E
__ET_EENTHE_OR__ARE___A____RE_ER_E__H__H__A___A__A
NTA_ETOTHE_O__ER__N_E_NO__E__EO__OR__TR__T_RE_E_O_
E__ER___E___HO_E_ERTO_N_REA_E_E__R_T__T___ETTERTOO
__TTHE__A_E_THE_H__TAN_A___NE___HER_AREE_A___E_O__
___T_T_T_ON___HER_THE___ENEREAN_H______HER_ARENOT_
_N_ETHE__ER__TE__O___O__ETTER_RATHERTHANONE_ETTERA
TAT__E
```

From the very beginning, in fact letter 5 we see that F was sent to E. Also a little further down there is a `YFPT` that translates into `T_ON`, suggesting that F is I. With these in we have.

```
ONEOFTHE_ORE_O___AR_R__TO___TE__I_THE____TIT_TION_
I_HERITI__O__ON____E_INTHE_____E_E_TIONOFTHE_EE_EN
_NE___A_ER_FORE_A___ETHE_RIN_I__EI__I___EEA_H_ETTE
RINTHEA__HA_ETI_RE__A_E___ANOTHEROR_O__I___THE_A_E
_ETTER_ORE_RE_I_E__A_ER__TATIONOFTHEA__HA_ETI__HO_
ENAN_A___IE_TOTHE__AINTE_TINTHE_____E_A_E_THE__A_E
__ET_EENTHE_OR__ARE___A____RE_ER_E__HI_HI_A_I_A__A
NTA_ETOTHE_O__ER_IN_E_NO__E__EOF_OR__TR__T_RE_E_O_
E__ER___EF__HO_E_ERTOIN_REA_E_E__RIT_ITI__ETTERTOO
_ITTHE__A_E_THE_HIFTAN_AFFINE_I_HER_AREE_A___E_OF_
___TIT_TION_I_HER_THE_I_ENEREAN_HI___I_HER_ARENOT_
IN_ETHE__ER__TE__O___OF_ETTER_RATHERTHANONE_ETTERA
TATI_E
```

The last several letters `ATATI_E` would suggest that the blank letter is an M, making M go to M. Putting this in we have `OMMON` at the end of the first line, suggesting that Q is C. Putting these in gives us,

```
ONEOFTHEMORE_O___ARCR__TO___TEM_I_THE____TIT_TIONC
I_HERITI_COMMON____E_INTHE_____E_ECTIONOFTHE_EE_EN
_NE___A_ER_FORE_AM__ETHE_RINCI__EI__IM__EEACH_ETTE
RINTHEA__HA_ETI_RE__ACE___ANOTHEROR_O__I___THE_AME
_ETTERMORE_RECI_E__A_ERM_TATIONOFTHEA__HA_ETI_CHO_
ENAN_A___IE_TOTHE__AINTE_TINTHE_____E_A_E_THE__ACE
__ET_EENTHE_OR__ARE___A____RE_ER_E__HICHI_A_I_A__A
NTA_ETOTHE_O__ER_INCE_NO__E__EOF_OR__TR_CT_RE_ECOM
E__ER___EF__HO_E_ERTOINCREA_E_EC_RIT_ITI__ETTERTOO
MITTHE__ACE_THE_HIFTAN_AFFINECI_HER_AREE_AM__E_OF_
___TIT_TIONCI_HER_THE_I_ENEREAN_HI__CI_HER_ARENOT_
INCETHE__ERM_TE__OC__OF_ETTER_RATHERTHANONE_ETTERA
TATIME
```

Note, right after the first `TION` there is `CI_HER`, suggesting that U is P. When we put that in we have a first line starting with `ONEOFTHEMOREPOP__AR` suggesting that the two missing letters are U and L, making A the letter U and D the letter L. Also, right after that we have `CR_PTO`, suggesting that B is Y. Now things are really taking shape.

```
ONEOFTHEMOREPOPULARCRYPTO_Y_TEM_I_THE_U__TITUTIONC
IPHERITI_COMMONLYU_E_INTHEPU__LE_ECTIONOFTHE_EE_EN
_NE__PAPER_FORE_AMPLETHEPRINCIPLEI__IMPLEEACHLETTE
RINTHEALPHA_ETI_REPLACE__YANOTHERORPO__I_LYTHE_AME
LETTERMOREPRECI_ELYAPERMUTATIONOFTHEALPHA_ETI_CHO_
ENAN_APPLIE_TOTHEPLAINTE_TINTHEPU__LEPA_E_THE_PACE
__ET_EENTHE_OR__AREU_UALLYPRE_ER_E__HICHI_A_I_A__A
NTA_ETOTHE_OL_ER_INCE_NO_LE__EOF_OR__TRUCTURE_ECOM
E__ERYU_EFULHO_E_ERTOINCREA_E_ECURITYITI__ETTERTOO
MITTHE_PACE_THE_HIFTAN_AFFINECIPHER_AREE_AMPLE_OF_
U__TITUTIONCIPHER_THE_I_ENEREAN_HILLCIPHER_ARENOT_
INCETHEYPERMUTE_LOC__OFLETTER_RATHERTHANONELETTERA
TATIME
```

On the first line, we have `_Y_TEM` suggesting that N is S. Putting this in we see the word SUBSTITUTION if we assign K to B. And there is a `COMMONLYUSE_` suggesting that H is D.

```
ONEOFTHEMOREPOPULARCRYPTOSYSTEMSISTHESUBSTITUTIONC
IPHERITISCOMMONLYUSEDINTHEPU__LESECTIONOFTHE_EE_EN
DNE_SPAPERSFORE_AMPLETHEPRINCIPLEISSIMPLEEACHLETTE
RINTHEALPHABETISREPLACEDBYANOTHERORPOSSIBLYTHESAME
LETTERMOREPRECISELYAPERMUTATIONOFTHEALPHABETISCHOS
ENANDAPPLIEDTOTHEPLAINTE_TINTHEPU__LEPA_ESTHESPACE
SBET_EENTHE_ORDSAREUSUALLYPRESER_ED_HICHISABI_AD_A
NTA_ETOTHESOL_ERSINCE_NO_LED_EOF_ORDSTRUCTUREBECOM
ES_ERYUSEFULHO_E_ERTOINCREASESECURITYITISBETTERTOO
MITTHESPACESTHESHIFTANDAFFINECIPHERSAREE_AMPLESOFS
UBSTITUTIONCIPHERSTHE_I_ENEREANDHILLCIPHERSARENOTS
INCETHEYPERMUTEBLOC_SOFLETTERSRATHERTHANONELETTERA
TATIME
```

We are coming down the home stretch now, we have `PU__LE` and the ciphertext letters here are VV so we know that V is Z. The `NE_SPAPERS` suggests that G is W. Now right before that we have `WEE_END`, suggesting that S is K. The `FORE_AMPLE` right after that suggests that J is X.

```
ONEOFTHEMOREPOPULARCRYPTOSYSTEMSISTHESUBSTITUTIONC
IPHERITISCOMMONLYUSEDINTHEPUZZLESECTIONOFTHEWEEKEN
```

```
DNEWSPAPERSFOREXAMPLETHEPRINCIPLEISSIMPLEEACHLETTE
RINTHEALPHABETISREPLACEDBYANOTHERORPOSSIBLYTHESAME
LETTERMOREPRECISELYAPERMUTATIONOFTHEALPHABETISCHOS
ENANDAPPLIEDTOTHEPLAINTEXTINTHEPUZZLEPA_ESTHESPACE
SBETWEENTHEWORDSAREUSUALLYPRESER_EDWHICHISABI_AD_A
NTA_ETOTHESOL_ERSINCEKNOWLED_EOFWORDSTRUCTUREBECOM
ES_ERYUSEFULHOWE_ERTOINCREASESECURITYITISBETTERTOO
MITTHESPACESTHESHIFTANDAFFINECIPHERSAREEXAMPLESOFS
UBSTITUTIONCIPHERSTHE_I_ENEREANDHILLCIPHERSARENOTS
INCETHEYPERMUTEBLOCKSOFLETTERSRATHERTHANONELETTERA
TATIME
```

The `PUZZLEPA_ES` suggests that R is G. And finally, the `AD_ANTAGE` suggests that O is V. Putting these in gives us the plaintext.

ONE OF THE MORE POPULAR CRYPTOSYSTEMS IS THE SUBSTITUTION CIPHER IT IS COMMONLY USED IN THE PUZZLE SECTION OF THE WEEKEND NEWSPAPERS FOR EXAMPLE THE PRINCIPLE IS SIMPLE EACH LETTER IN THE ALPHABET IS REPLACED BY ANOTHER OR POSSIBLY THE SAME LETTER MORE PRECISELY A PERMUTATION OF THE ALPHABET IS CHOSEN AND APPLIED TO THE PLAINTEXT IN THE PUZZLE PAGES THE SPACES BETWEEN THE WORDS ARE USUALLY PRESERVED WHICH IS A BIG ADVANTAGE TO THE SOLVER SINCE KNOWLEDGE OF WORD STRUCTURE BECOMES VERY USEFUL HOWEVER TO INCREASE SECURITY IT IS BETTER TO OMIT THE SPACES THE SHIFT AND AFFINE CIPHERS ARE EXAMPLES OF SUBSTITUTION CIPHERS THE VIGENERE AND HILL CIPHERS ARE NOT SINCE THEY PERMUTE BLOCKS OF LETTERS RATHER THAN ONE LETTER AT A TIME

In summary, the key we constructed is below, since Q and J were not used in the plaintext, there is no need to include them here. This plaintext is an excerpt from the text, *Introduction to Cryptography with Coding Theory* by Wade Trappe and Lawrence C. Washington, in their section on substitution ciphers.

Plaintext	**Ciphertext**
A	W
B	K
C	Q
D	H
E	C
F	E
G	R
H	X
I	F
J	

Plaintext	**Ciphertext**
K	S
L	D
M	M
N	T
O	P
P	U
Q	
R	I
S	N
T	Y
U	A
V	O
W	G
X	J
Y	B
Z	V

One should also note that we did get a little lucky at the beginning with our first choices. Recall that we were fairly sure that I or N was R, N, or S, and since HER is a common trigram, I or N is probably R. Then we noted that since IC is also a prominent digram in the ciphertext, I being R is more probable. This turned out to be the right choice. If we had made an incorrect choice it would have become apparent when we got further along and words were not showing themselves. In this case we would backtrack to our choice for R and continue from there. Δ

Hill Climbing

In the above discussion and example, we used frequency analysis to find the key to a substitution cipher. In that analysis, we used the frequencies of single characters and a few of the more frequent digrams and trigrams. From these we also used word patterns to guess certain characters when frequency analysis did not supply enough information. These are the methods that would have been used a century or more ago, methods that can be completed by hand.

One question we might ask is, what would happen if we took into account the probabilities all of the digrams or trigrams in the English language, or better yet, consider quadgrams (4-grams) and quintgrams (5-grams)? Using this much information is clearly beyond the capability of a human but, of course, we can enlist the power of the computer. Going this far is clearly not necessary, since we can solve a substitution cipher by using single character frequencies and a little information on digram and trigram frequencies, given enough ciphertext. So why might we bother with a new technique? In general, even though a problem has been solved, finding other solutions to the problem can provide insight into the mathematical workings behind the problem and can provide insight into different but related problems.

One might be inclined to invoke the old adage "why reinvent the wheel", but frankly I am very glad that someone did, I would hate to think of what my gas mileage would be if my car had stone tires.

Hill Climbing is a technique that takes more than single character frequencies and digrams into account in its analysis, it is couched firmly in probability theory and does not rely on word patterns so there is no need for human intervention and guessing at characters to fill out the words. The downside is that it does require the analysis of a large amount of information and thus the computer is a necessity.

In the hill climbing technique we take a piece of text and we calculate the probability that the text is in English, or whatever language we are using. One way to do this is to take a list of all possible trigrams, AAA, AAB, AAC, . . . and calculate frequencies of each of these trigrams in the English language, in much the same was as we got single character frequencies. We would take a large amount of English text, remove all punctuation and spaces, convert the characters to uppercase to make the analysis case insensitive and start counting. There are 17,576 possible trigrams using the English alphabet, so doing this by hand is not possible, even though some trigrams like QQQ could be skipped. In a similar fashion, we could examine the 456,976 possible quadgrams and the 11,881,376 quintgrams and so on. Once we have this data, we can calculate the probability of AAA, AAB, . . . showing up in a random bit of English text, we will denote these probabilities as $p(AAA)$, $p(AAB)$, ... These would be calculated empirically as well, simply talking the frequency of the trigram and dividing it bu the total number of trigrams that were examined, that is, $p(AAA) = \frac{freq(AAA)}{n}$, where $freq(AAA)$ denotes the frequency of AAA in the examined text and n is the total number of trigrams in the examined text. If a trigram does not show up in our list of frequencies, like QQQ would have no occurrences, we do not associate a probability of 0, we give it a very small probability, usually about $\frac{1}{100}$ of a single occurrence is sufficient, so $p(QQQ) = \frac{1}{100n}$.

Now if we have a piece of text, let's assume that the text is not ciphertext but plaintext, and we want to find the probability of the text being English using trigram analysis. For example, let's say our text is, "HILLCLIMBINGISATECHNIQUE", which is in English. We break the text into trigrams, HIL, ILL, LLC, LCL, . . . , QUE. The probability that the text is English would be the probability that HIL is English, and ILL is English, and LLC is English, . . . , and QUE is English. From elementary probability theory, we know that when we want the probability of event A and event B happening, $p(A \cap B) = p(A) \cdot P(B)$, as long as events A and B are independent. It is certainly true that having AAA in some text and AAB in some text is not independent, if AAA is in some text then the existence of the last two A's makes it possible for AAB to be in the text. On the other hand, if there are no occurrences of AA in the text then it is impossible for AAB to be in the text. So theoretically put trigram events are not independent, but as is commonly done, we will ignore this difficulty since the dependencies do not significantly alter the final results. So assuming that we have independence, the probability that our short text is in English is,

$$p(HILLCL \ldots QUE) = p(HIL) \cdot p(ILL) \cdot p(LLC) \cdots p(QUE)$$

Note that all of these probabilities are fairly small. It is true that $p(THE)$ is relatively large but consider the number of times that LLC or CHN occur in English text. So the product of these probabilities is very small. Furthermore, consider a bit of text that is 1000 characters long, there are 998 trigrams in that text, and so the probability is a product of 998 very small numbers. As mathematicians, really small and really big numbers do not bother us, but we need to consider the computer. Now if we are using a computer algebra system that deals with exact arithmetic, then again this is no problem, but these systems tend to be a bit slow for this type of application. If we use a program that is written in a modern language, like C++ or Java, then there are limits to the how small and how big a stored number can be. For example, in C++ and Java, a double precision decimal number must be larger than about 10^{-308}. For any decent size text the products of the probabilities will easily go below this number, and subsequently be converted to 0 by the program, making the calculation useless. So instead of looking at the probability we consider the logarithm of the probability. We can use any base we want for the logarithm, but 10 is a common choice. When we do this, our calculation is,

$$\begin{aligned}\log(p(HILLCL\ldots QUE)) &= \log(p(HIL)\cdot p(ILL)\cdot p(LLC)\cdots p(QUE)) \\ &= \log(p(HIL)) + \log(p(ILL)) + \cdots + \log(p(QUE))\end{aligned}$$

Using this, our numbers will be more manageable in size and if we precalculate the logarithms, using addition in place of multiplication will increase the speed of the computation. This sum of logarithms is called a fitness measure, which we will denote as $fm(T)$ where T simply represents some text. So in general,

$$fm(T) = \sum_{t\in T} \log(p(t))$$

where T is some text and t is a trigram in T. If we go back to probabilities for a second, the larger the probability, the better the text fits the English language, and hence the more likely that the text is actually English. Since the logarithm base 10 is an increasing function, the fitness measure will increase if and only if the probability increases. So the larger the fitness measure is, the better the text fits English, and hence the more likely the text is English. This fitness measure can be used with quadgrams and quintgrams and n-grams in the same manner as long as we have a database of the n-gram probabilities.

So, how can we use this calculation to solve a substitution cipher? What we would want is the key that produces the largest fitness measure, but this simply brings us back to a brute force method and with $26! = 403,291,461,126,605,635,584,000,000$ possible keys we clearly need another approach.

There are many ways to implement a hill climb in a more efficient manner, in many places in the literature it is suggested that you start with a random key, calculate the fitness measure of that key (probably will not be too high), then start an iteration in which at each pass you transpose two key entries and recalculate the fitness measure, if the new fitness measure is higher you repeat with the new key and if it is smaller you go back to the old key and repeat with another pair. It is suggested that this iteration be done a fixed number of times that is predetermined by the user. The pair of entries that are transposed are usually chosen at random.

Another method that we have found to work particularly well, and usually better than the above method is as follows.

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.

A few things to note about this algorithm. First we do not predetermine the number of transpositions to be done ahead of time. This does run the risk, in a worst case scenario, of having to do many passes before the fitness measure does not increase. In the very worst case, we would be looking at something equivalent to a brute force attack. Fortunately, on the ciphertexts we tested, the number of passes was usually 3–4 and in some cases 5–6. The fact that we start with a key produced from a single character frequency analysis and not a random key helps reduce the number of needed passes.

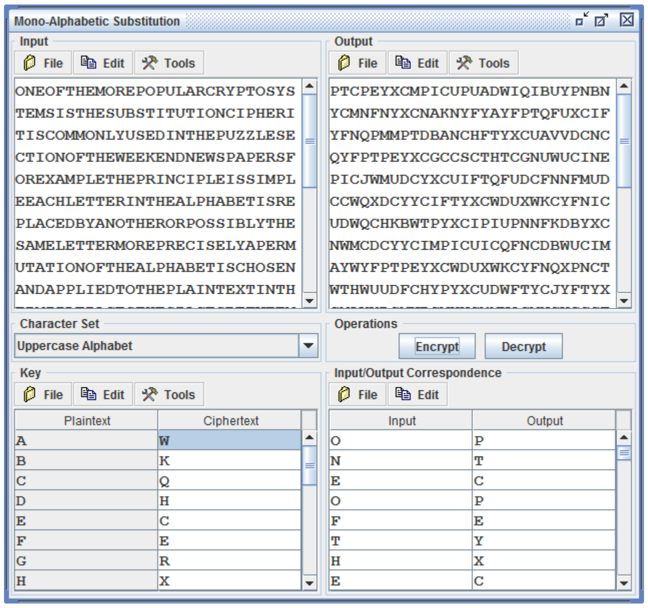
Second, in each pass we check all possible transpositions of two key letters, since there are only 325 of these each pass can be done by the machine in a short amount of time. This also guarantees that there is not a duplication of a transposition, as would possibly be the case if they are selected randomly and it guarantees that all transpositions are tried.

Third, let's think about what needs to be done in each pass. The algorithm does 325 transpositions, for each of these it creates a possible plaintext and then it recalculates the fitness measure. If the ciphertext has, say 1000 characters in it, then the program will need to find the probabilities of about 1000 trigrams, quadgrams, or quintgrams for each transposition. This is a total of 325,000 searches of the trigrams, quadgrams, or quintgrams databases. The trigram database will not be very large, around 17,576 entries. On the other hand, the quadgrams database can have up to 456,976 entries and the quintgrams database could possibly have 11,881,376 entries. So doing a linear search of these databases would not be a very smart implementation, since for the quintgrams database that would require an average of 1,930,723,600,000 string comparisons for each pass, even for a fast computer this will take a very long time. So if we want to go as far as quintgram analysis, we need to sort the databases so that we can use a binary search. A binary search for each pass of the quintgram analysis would require only 7,800,000 string comparisons, which is much more reasonable.

2.2.3 Cryptography Explorer

The Cryptography Explorer program has a tool for the monoalphabetic substitution cipher, a tool for doing frequency analysis, and a tool for doing a hill climbing analysis.

Monoalphabetic Substitution Cipher Tool



```
ONEOFTHEMOREPOPULARCRYPTOSYS
TEMSISTHESUBSTITUTIONCIPHERI
TISCOMMONLYUSEDINTHEPUZZLESE
CTIONOFTHEWEEKENDNEWSPAPERSF
OREXAMPLETHEPRINCIPLEISSIMPL
EEACHLETTERINTHEALPHABETISRE
PLACEDBYANOTHERORPOSSIBLYTHE
SAMELETTERMOREPRECISELYAPERM
UTATIONOFTHEALPHABETISCHOSEN
ANDAPPLIEDTOTHEPLAINTEXTINTH
```

```
PTCPEYXCMPICUPUADWIQIBUYPNBN
YCMNFNYXCNAKNYFYAYFPTQFUXCIF
YFNQPMMPTDBANCHFTYXCUAVVDCNC
QYFPTPEYXCGCCSCTHTCGNUWUCINE
PICJWMUDCYXCUIFTQFUDCFNNFMUD
CCWQXDCYYCIFTYXCWDUXWKCYFNIC
UDWQCHKBWTPYXCIPIUPNNFKDBYXC
NWMCDCYYCIMPICUICQFNCDBWUCIM
AYWYFPTPEYXCWDUXWKCYFNQXPNCT
WTHWUUDFCHYPYXCUDWFTYCJYFTYX
```

Character Set: Uppercase Alphabet

Plaintext	Ciphertext
A	W
B	K
C	Q
D	H
E	C
F	E
G	R
H	X

Input	Output
O	P
N	T
E	C
O	P
F	E
T	Y
H	X
E	C

Figure 2.7: Cryptography Explorer Monoalphabetic Substitution Cipher Tool

We will start with the Cipher tool. 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.

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 —

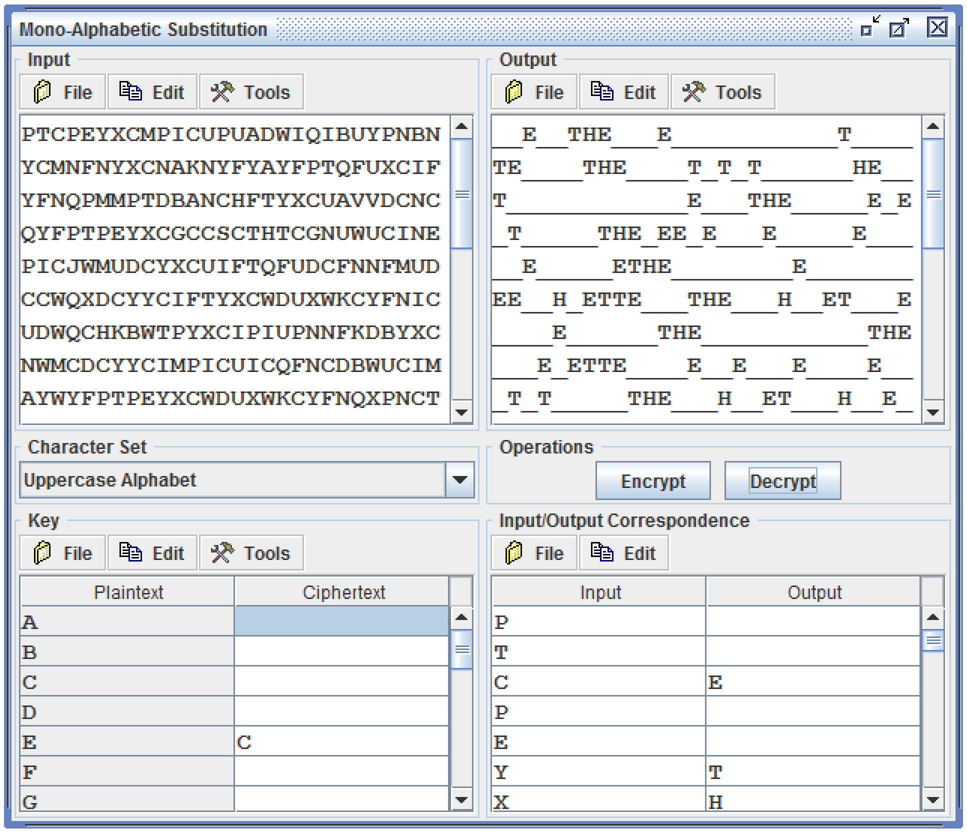
1. Input the ciphertext message into the Input box. Make sure that the characters are from the same character set as the one selected. Note that you can change the character set using the selection box below the Input box. There are also some quick conversion tools in the Tools menu.
2. Input a Substitution 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.

When you change the character set you will see the Plaintext/Ciphertext grid change accordingly. Although the classical monoalphabetic cipher is set up as a one-to-one correspondence between the plaintext characters and the ciphertext characters, the program is not limited to a one-to-one correspondence between plaintext and ciphertext characters. 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 to more than one substitution the program will randomly select one of the options for each of those characters. For example, if we had WT in the ciphertext column beside A, then whenever an encryption is done and the letter A is encountered, the program will randomly generate a 0 or 1, with equal probability. If a 0 is generated the program will use W for the A and if a 1 is generated then the program will use a T for the A. Likewise, if there are three or more letters associated with a single letter the program will randomly generate a choice with equal probability. The same can be done for decryption. For example, if we place an E in the ciphertext column beside both R and H, when we click the decrypt button and an E is encountered then the program will choose either R or H for the decryption, again with equal probability.

In the above example, the ciphertext that we started with was placed in the input box in the window, and all of the typewriter font text from there on was created by the program and copied from the output box. Similarly, the input/output correspondence grids were taken from the Input/Output Correspondence grid in the lower right of the window. So, for example, Once we had determined what we thought T, H, and E were, we placed Y in the ciphertext side of the key chart beside T, X in the ciphertext side of the key chart beside H, and C in the ciphertext side of the key chart beside E, and clicked the Decrypt button. At that point, all of the occurrences of Y, X and C in the input were changed to T, H, and E respectively. Since all of the other letters were still unassigned they were given blanks.

Frequency Analysis Tool

The Cryptography Explorer program also has a tool for 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.



Input:

```
PTCPEYXCMPICUPUADWIQIBUYPNBN
YCMNFNYXCNAKNYFYAYFPTQFUXCIF
YFNQPMMPTDBANCHFTYXCUAVVDCNC
QYFPTPEYXCGCCSCTHTCGNUWUCINE
PICJWMUDCYXCUIFTQFUDCFNNFMUD
CCWQXDCYYCIFTYXCWDUXWKCYFNIC
UDWQCHKBWTPYXCIPIUPNNFKDBYXC
NWMCDCYYCIMPICUICQFNCDBWUCIM
AYWYFPTPEYXCWDUXWKCYFNQXPNCT
```

Output:

```
__E__THE___E___________T____
TE____THE_____T_T_T_____HE__
T____________E___THE_____E_E
_T_____THE_EE_E___E_____E___
__E_____ETHE________E_______
EE__H_ETTE___THE___H__ET___E
____E______THE___________THE
___E_ETTE____E__E___E____E__
_T_T_____THE___H__ET___H__E_
```

Character Set: Uppercase Alphabet

Key:

Plaintext	Ciphertext
A	
B	
C	
D	
E	C
F	
G	

Input/Output Correspondence:

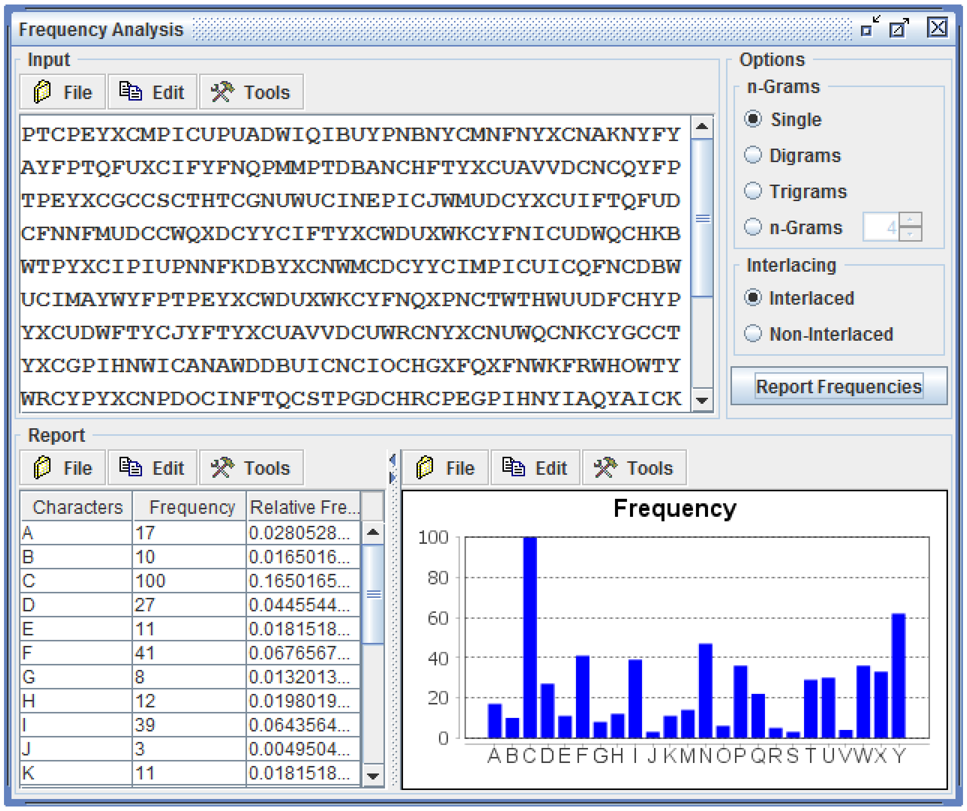
Input	Output
P	
T	
C	E
P	
E	
Y	T
X	H

Figure 2.8: Cryptography Explorer Monoalphabetic Substitution Cipher Tool Displaying Partial Decryption

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.

The above image is displaying the previous example using single character analysis. The image below is displaying the same text using a digram analysis. A couple things to notice here. Under the tools menu above the report grid is an option to sort the results, which we did to get the most frequent digrams at the top. This option also sorts the results in the bar chart on the right. When this is done the bar chart is set to its default of showing all the data. With digrams, trigrams, and n-grams there are usually too many bars in this mode to see things clearly. Under the tools menu above the chart is an option to show a range of data. When you select this the program will ask you for the number of data items to



```
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...

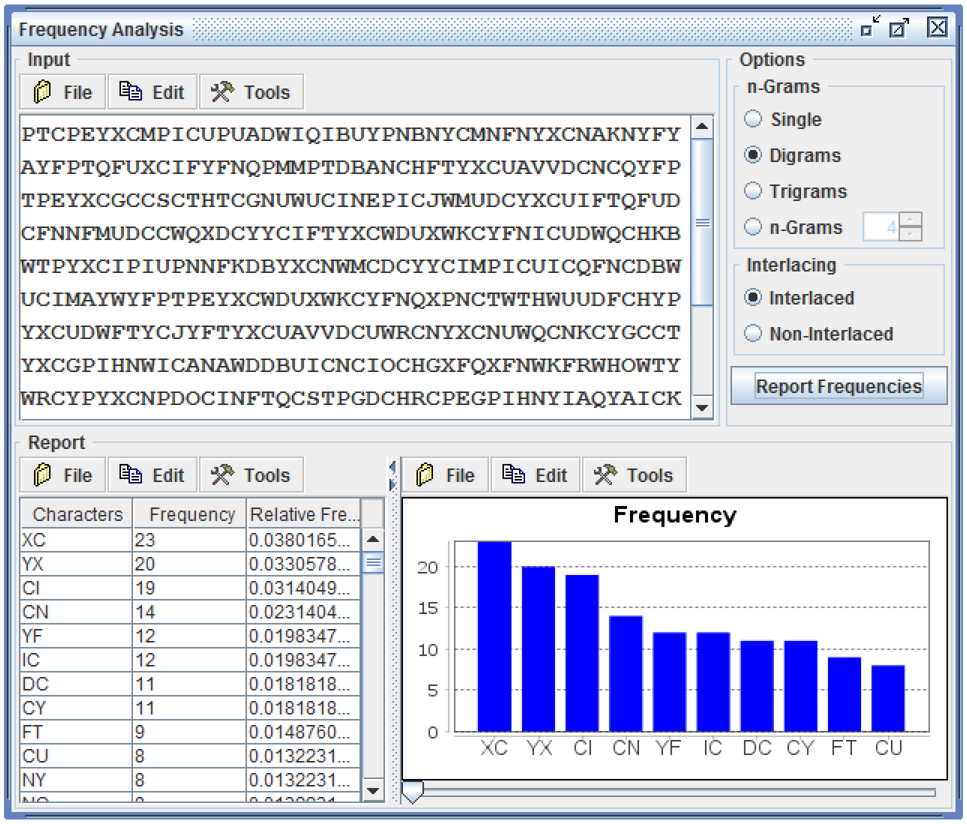
Figure 2.9: Frequency Analysis Tool

display, select the number you want and click OK. At that point the bar chart will redisplay the chart and a slider will appear at the bottom of the chart. We did this, selecting to view 10 items. The slider can be used to shift the graph to view the other items.

The frequency reporting options are fairly self-explanatory. Single will report the frequencies and relative frequencies of each character in the input. Digrams will report all consecutive character pairs, trigrams will report all consecutive character triples, and n-grams will report all consecutive blocks of n characters in the input. The user can select between 4 and 10 characters per block.

Interlaced 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 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.

There are options for coping the grid and chart to the clipboard, printing the results and saving the information to a file. For more information on these tools, please see the appendix on the Cryptography Explorer program.



```
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...
XC	23	0.0380165...
YX	20	0.0330578...
CI	19	0.0314049...
CN	14	0.0231404...
YF	12	0.0198347...
IC	12	0.0198347...
DC	11	0.0181818...
CY	11	0.0181818...
FT	9	0.0148760...
CU	8	0.0132231...
NY	8	0.0132231...

Figure 2.10: Frequency Analysis Tool

Hill Climb Analysis Tool

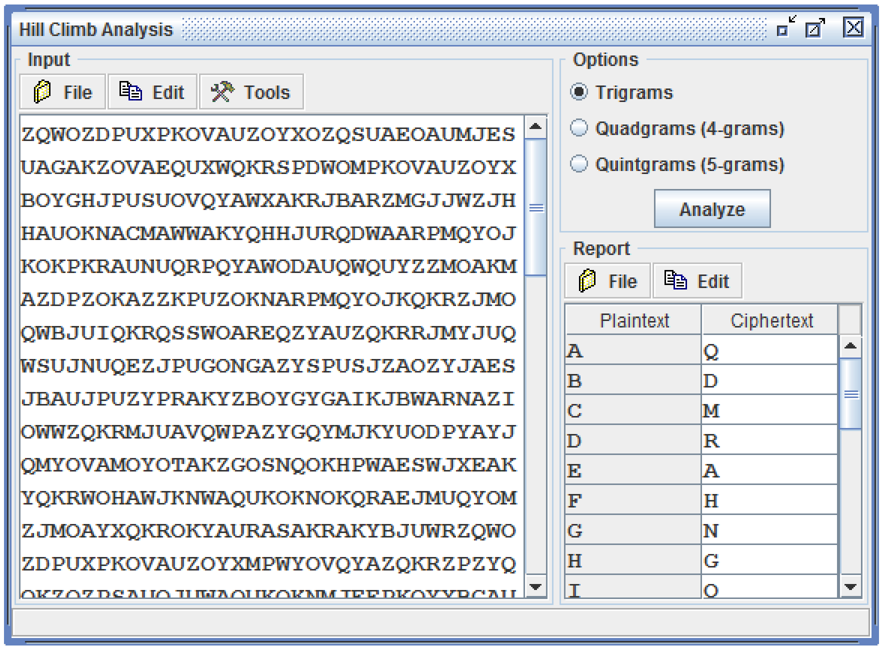
The Hill Climb Analysis window will apply the hill climbing algorithm for substitution ciphers to the input text. When the analysis is finished, the best guess to the substitution cipher key will be displayed in the report grid.

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.

Options

- The data set options are as follows.



```
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 2.11: Hill Climb Analysis Tool

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.

2.3 Vigenère

2.3.1 Definitions and History

The Vigenère cipher was developed in the mid sixteenth century by Blaise de Vigenère. When Vigenère was twenty six years old he went on a two-year diplomatic mission to Rome. There is where he was first exposed to the world of cryptography. Building on the work of Leon Battista Alberti(1404–1472) (often called the Father of Western Cryptography), Johannes Trithemius, and Giovanni Porta, Vigenère developed several cryptographic methods and steganographic techniques. In 1585 he published *Traictè des Chiffres*, which contained his contributions to the field. Vigenère developed much more sophisticated ciphers then the standard repeated keyword cipher we will discuss here. We will briefly discuss some of the autokey ciphers that Vigenère published in the Traictè des Chiffres, which can be more difficult to break than the classical repeated keyword.

Alberti was one the leading figures of the Renaissance; a painter, composer, poet and philosopher. He was also the author of the first scientific analysis of perspective, a treatise on the housefly, and a funeral oration for his dog. He is probably best known as an architect, having designed Rome's first Trevi Fountain and having written "De Re Aedificatoria", the first printed book on architecture, which acted as a catalyst for the transition from Gothic to Renaissance design.[56]

The Vigenère cipher is a polyalphabetic cipher which is a variation of the Caesar shift cipher. As the name implies it uses several different alphabetic substitutions in place of the one, as in the monoalphabetic ciphers.

The Vigenère cipher was one of those "quantum leaps" in cryptography, of which there are several in history, where it seems that a completely secure method of encryption had been found. The Vigenère cipher was considered to be unbreakable, and it achieved the title of the *chiffre indéchiffrable.* It was not until 1863, nearly 300 years after its creation, when F. W. Kasiski found a method for cryptanalyzing the cipher and later the method of coincidence analysis made the determination of the keyword length even easier. We will discuss both methods later in this section.

Definition 14: *The Vigenère cipher encryption is implemented as follows,*

1. *A keyword is chosen for the cipher, this is the key to both encryption and decryption.*
2. *Each letter is converted to a shift value using the standard A–Z as 0–25, as in the chart below. The keyword then becomes a shift vector.*

Table 2.14: Vigenère Cipher Shift Coding

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M
Shift	0	1	2	3	4	5	6	7	8	9	10	11	12
Letter	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Shift	13	14	15	16	17	18	19	20	21	22	23	24	25

For example, if the keyword is VIGENERE then the shift vector is $(21, 8, 6, 4, 13, 4, 17, 4)$.

3. *The plaintext message is written out character for character and the keyword (or shift vector) is written below it, character for character and the keyword is repeated as many times as is needed to cover the message.*

4. *Each character in the plaintext message is shifted by the amount as its corresponding keyword letter to create the ciphertext.*

The decryption process is the same except that the shift done in the last step is reversed to take the cuphertext back to the plaintext.

Repeated Keyword

The process outlined in the definition is called the Repeated Keyword formulation of the Vigenère cipher, since the keyword is repeated to obtain enough shifts for the entire message. As usual in cryptography, repetition tends to be a weakness and we will see how this repetition can be used to easily crack the Vigenère cipher. We will do a quick example of encryption and decryption.

Example 12: Let's consider the following plaintext,

```
THEVIGENERECIPHERISAMETHODOFENCRYPTINGALPHABETICTE
XTBYUSINGASERIESOFDIFFERENTCAESARCIPHERSBASEDONTHE
LETTERSOFAKEYWORDITISASIMPLEFORMOFPOLYALPHABETICSU
BSTITUTION
```

We will encrypt this message with the keyword ENCRYPT, the shift vector for the word ENCRYPT is $(4, 13, 2, 17, 24, 15, 19)$. To encrypt the message we write the plaintext in the far left column, the key column repeats the word ENCRYPT through the message. The third column shows the ciphertext after the character shift. For the first character, T (19), is shifted by E (4), to X (23). The second character H (7) is shifted by N (13) to U (20), then E (4) by C (2) to G (6). The character V (21) is shifted by R (17) to M ($12 = 38 \pmod{26}$).

Plaintext	Key	Ciphertext
T	E	X
H	N	U
E	C	G
V	R	M
I	Y	G
G	P	V
E	T	X
N	E	R
E	N	R
R	C	T
E	R	V

Plaintext	**Key**	**Ciphertext**
C	Y	A
I	P	X
P	T	I
⋮	⋮	⋮
U	N	H
T	C	V
I	R	Z
O	Y	M
N	P	C

Continuing the process through the entire message gives us the following ciphertext,

```
XUGMGVXRRTVAXILRTZQPFIGJFBDYIAEIWEMMAIRJEAEOGKGRMI
KVSWJLMAIRQTKMRUFDSBJSGICCMGNGJYGVMCJVPHUEFGUMCMLR
NVRIXVFQWYZXCJQIBXMMFCJGBIPRHFPBHJCQCWPETUCSCIBGFW
SQIBXHVZMC
```

One should note a couple things about this encryption. If we look closely at the partial encryption grid above we will note that the letter E encrypts to G, X, R, and V, also T encrypts to X and V. So we no longer have a one-to-one correspondence between plaintext letters and ciphertext letters. A single plaintext letter can encrypt to several different ciphertext letters, this is because a single plaintext letter can match up with different keyword letters and hence be shifted by different amounts. For the same reason, two plaintext letters can be encrypted to the same ciphertext letter.

At the time that the Vigenère cipher was being used, the only known cryptanalysis was frequency analysis, which depended on a one-to-one correspondence between plaintext letters and ciphertext letters to work. The Vigenère cipher does not have this property, in general, and hence the only known cryptanalysis technique was rendered useless. This is what made the Vigenère cipher unbreakable for nearly three centuries.

If we attempt to do frequency analysis on the ciphertext from this example we get the following frequency chart.

As you can see from the frequency chart of the ciphertext from the example and the English Letter Relative Frequencies chart, the distribution is clearly different. We no longer have a letter with a substantially higher frequency, there are more higher frequency characters, that are closer together, and the lower frequency characters are not a prominent. All of this is due to the mixing that the Vigenère cipher creates.

Δ

Autokeys

When we look at the cryptanalysis of the Vigenère cipher we will see that the weakness comes from repeating the keyword over and over again. This is what Kasiski's method

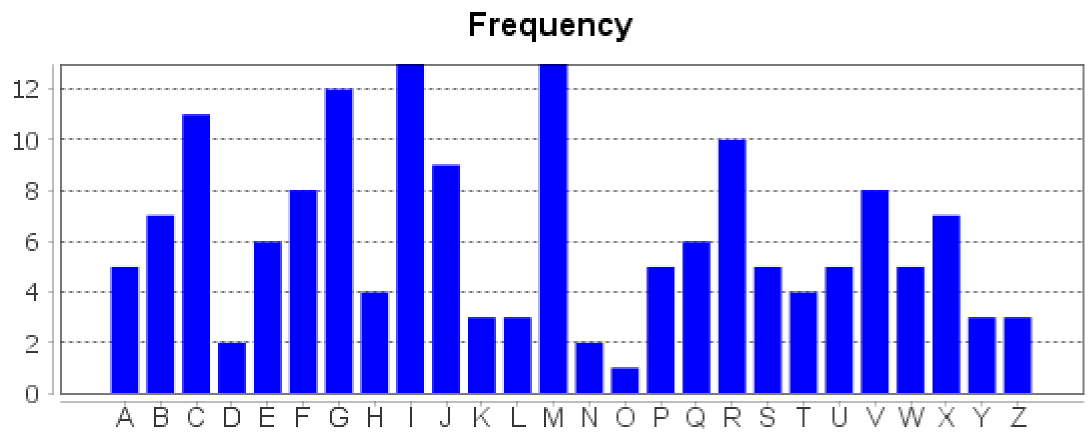


Figure 2.12: Frequency Analysis of Vigenère Example

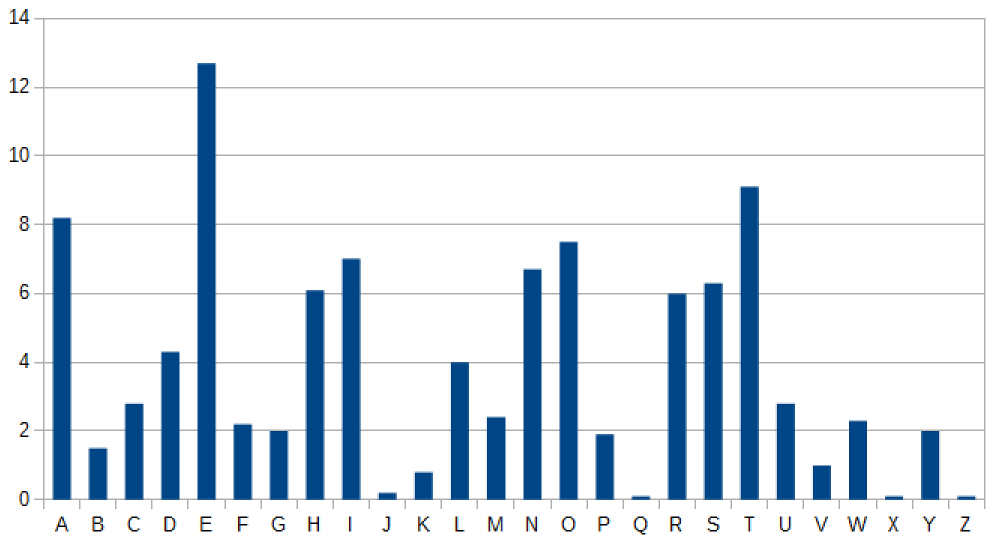


Figure 2.13: English Letter Relative Frequencies Sorted by Letter

and the method of coincidence analysis exploits to determine the possible lengths of the keyword. In general, in cryptography, any repetition in the encryption process tends to create a weakness. Even very small repetitions can be fatal, as in the case of the German Enigma cipher of World War II.

In the Traictè des Chiffres, Vigenère also discussed autokey methods that did not repeat the keyword, although he did not know at the time that the keyword repetition would turn out to be a crippling weakness to his cipher.

The two most common types of Vigenère autokey ciphers are the plaintext autokey and the ciphertext autokey. The plaintext autokey cipher uses the keyword first but then instead of repeating the keyword you use the plaintext message to fill out the key. Similarly, with the ciphertext autokey, you first use the keyword and then you use the ciphertext message to fill out the key. With the ciphertext autokey, you are building the ciphertext at the same time you are using it as a key, but since you are at least one letter ahead (since the keyword has at least one letter) you will always have the next ciphertext letter you need for the key.

As an example, we will redo the above example using both the plaintext and ciphertext autokeys.

Example 13: As before we will use the following plaintext with the keyword ENCRYPT.

```
THEVIGENERECIPHERISAMETHODOFENCRYPTINGALPHABETICTE
XTBYUSINGASERIESOFDIFFERENTCAESARCIPHERSBASEDONTHE
LETTERSOFAKEYWORDITISASIMPLEFORMOFPOLYALPHABETICSU
BSTITUTION
```

Here we will use a plaintext autokey encryption process. We reproduce only the fist several lines of the Plaintext/Ciphertext correspondence grid. Note that after the word ENCRYPT, the key contains the plantext message.

Plaintext	**Key**	**Ciphertext**
T	E	X
H	N	U
E	C	G
V	R	M
I	Y	G
G	P	V
E	T	X
N	T	G
E	H	L
R	E	V
E	V	Z
C	I	K
I	G	O
P	E	T
H	N	U
E	E	I
R	R	I
I	E	M
⋮	⋮	⋮

The final ciphertext for the encryption is,

```
XUGMGVXGLVZKOTUIIMUIBLXYWVORIGJFBDYMAIRJEAIOKTTRAE
YXUGWLMKZBQYJQRYOXHZNJWFJQBHFIJEEVKPLWRJDIHLHFFUHW
PHHGXYWZJTDIPOCWDSXGOOJLUITWFGZYDQTTZPMZUWOMCTTRZU
CWMQVMNJGG
```

The first seven letters of the ciphertext are the same as before, as expected, and then the sequence changes. Δ

Example 14: As before we will use the following plaintext with the keyword ENCRYPT.

```
THEVIGENERECIPHERISAMETHODOFENCRYPTINGALPHABETICTE
XTBYUSINGASERIESOFDIFFERENTCAESARCIPHERSBASEDONTHE
LETTERSOFAKEYWORDITISASIMPLEFORMOFPOLYALPHABETICSU
BSTITUTION
```

Here we will use a ciphertext autokey encryption process. We reproduce only the fist several lines of the Plaintext/Ciphertext correspondence grid. Note that after the word ENCRYPT, the key contains the ciphertext message.

Input	**Key**	**Output**
T	E	X
H	N	U
E	C	G
V	R	M
I	Y	G
G	P	V
E	T	X
N	X	K
E	U	Y
R	G	X
E	M	Q
C	G	I
I	V	D
P	X	M
⋮	⋮	⋮

The final ciphertext for the encryption is,

```
XUGMGVXKYXQIDMRCOYADYVVVMDRDZIXDBGWHVDDMVDHWHWUXWL
TAXSROTGGXKVFBKYLPYNGPCCTLGIPGUTCIQENYKUJQWRBYHCXA
CFRAGOSQKRKKMOEBUSDUGETCESFKJHTQGKZXSRQRZGXTVJZBYR
UNCHUSKCBP
```

Again, the first seven letters of the ciphertext are the same as before, as expected, and then the sequence changes. Δ

2.3.2 Cryptanalysis

The cryptanalysis of the Vigenère cipher is very interesting and quite cleaver. When you are looking at these classical techniques of encryption and wondering why it took a bunch of brilliant minds several centuries to come up with something that seems so easy to do, you need to also investigate the mathematical landscape of the time. For example, substitution ciphers had been around since the first century, at the very least, yet it took about 900 years before anyone discovered the concept that eventually showed the weakness of the cipher. It then took another 500 years for the concept to spread and mature into a cryptanalysis technique. During that time, the mathematical foundations existed but no one put the right pieces in the right order to create a code breaking technique.

The mathematics that was known at the time of Caesar was very limited, and most of the general population was not only illiterate but they were mathematically illiterate as well. The Greeks had laid down the foundations of geometry about 500 years earlier and there was an algorithmic understanding of some mathematical processes that were developed by the Egyptians and the Babylonians, but there was no real concept of what we now consider to be high school algebra. Nor was there anything that was even approaching the most basic ideas in statistics, where we would commonly put frequency analysis. In short, the mathematics, and linguistics, needed to break a monoalphabetic cipher did not exist during the majority of the time the cipher was used. And even when it did, the cipher was still used successfully for a large number of years, until the concept of frequency analysis became mature enough to be a common attack.

Keep this idea in mind when we look at other classical techniques and even modern techniques. Our modern ciphers are secure for now, but will they succumb to faster computers or "different" computers, or will it be faster algorithms we already know or are derivable from our current knowledge of mathematics, or is the mathematics we need not yet discovered?

As for the Vigenère cipher, it withstood 300 years of cryptanalysis. And it did so in an age when there was a higher percentage of literate and mathematically literate people. What is clever about the method is that once the length of the keyword is known, the problem is reduced to a simple frequency analysis, in fact just a shift analysis. For example, say we know that the keyword has 5 letters in it. This means that if we take letters 1, 6, 11, 16, 21, 26, ... from the ciphertext message, all of these characters would have been shifted by the same amount, the first letter of the keyword. So if this set of letters is large enough we can do frequency analysis on them. In fact, since we know that all of these letters were simply shifted we can look for the most frequent letter in the set, this one probably came from an E, and take the shift amount from E to get the keyword letter. We would then extract letters 2, 7, 12, 17, 22, 27, ... from the ciphertext and do the same analysis to get the second letter in the keyword, and so on. If we were correct in our determination of the keyword length and if the ciphertext is long enough, this process should work.

The second stage is a simple frequency analysis, which of course existed in the mid sixteenth century. So the question is, did the mathematics to determine the probable keyword length exist in the mid sixteenth century?

Repeated Keyword

As we discussed above, the method works in two stages. First find the probable length of the keyword and then find the keyword itself, letter by letter.

There are two main ways to determine the keyword length, Kasiski's method and the method of coincidence analysis.

Kasiski's Method

Kasiski's Method, which is also known as Kasiski's Test or Kasiski examination was developed by Friedrich Kasiski[28] in 1863 but it seems to have been independently discovered by Charles Babbage[14] as early as 1846.

The idea behind Kasiskis method is the observation that repeated portions of plaintext enciphered with the same part of the keyword must result in identical ciphertext patterns. Therefore, the number of symbols between the start of repeated ciphertext patterns should be a multiple of the keylength. There will, of course be some coincidences that will through this off from a prefect fit but the longer the matching substrings are the more confidence we would have in the distances being a multiple of the keylength.

To apply Kasiskis method to some ciphertext we would set a length or range of lengths of the substrings to look for then we would find all equal substrings of these lengths. When a match is found we would find the distance between the two equal substrings, this should be a multiple of the keylength. So we would find the factors of this length and compare then to factors of the other lengths of matching substrings. For example, say we set the substring length to 3, and we find that there are two occurrences of ABC in the ciphertext and the number of characters between the two A's is 35. This would suggest that the length of the keyword is either 5, 7 or 35. One is also a divisor 35 by having a single character Vigenère keyword is unlikely and results in a simple Caesar shift. Also, 35 is a bit long but we will not discount that yet. Let's say we also find a repeated XYZ that has 21 characters between the X's. This would suggest that the length of the keyword is either 3, 7 or 21. With these two findings we would get the gut feeling that we were looking at a keyword of length 7, the most frequent divisor of the lengths. We will most likely find many repeated substrings in the ciphertext, if we keep a count of all the divisors of all the difference lengths the most frequent ones will probably be a divisor of the keyword size.

The smaller the substring size we use is the more matches we will find, but also more of these matches will be coincidences and not have anything to do with the keyword length. The larger the substring size the fewer matches we will get but even less of these will be due to simple coincidences. So a good practice is to use moderately sized substring lengths. We would not go below size 3 and on the upper end, it would depend how much ciphertext you have. If you have a long string of ciphertext you might get lucky with longer substrings. Furthermore, you should try different lengths of substrings or an interval of substring lengths.

Example 15: If we apply Kasiskis method to the ciphertext we got from the repeated keyword encryption using the keyword ENCRYPT,

```
XUGMGVXRRTVAXILRTZQPFIGJFBDYIAEIWEMMAIRJEAEOGKGRMI
KVSWJLMAIRQTKMRUFDSBJSGICCMGNGJYGVMCJVPHUEFGUMCMLR
NVRIXVFQWYZXCJQIBXMMFCJGBIPRHFPBHJCQCWPETUCSCIBGFW
SQIBXHVZMC
```

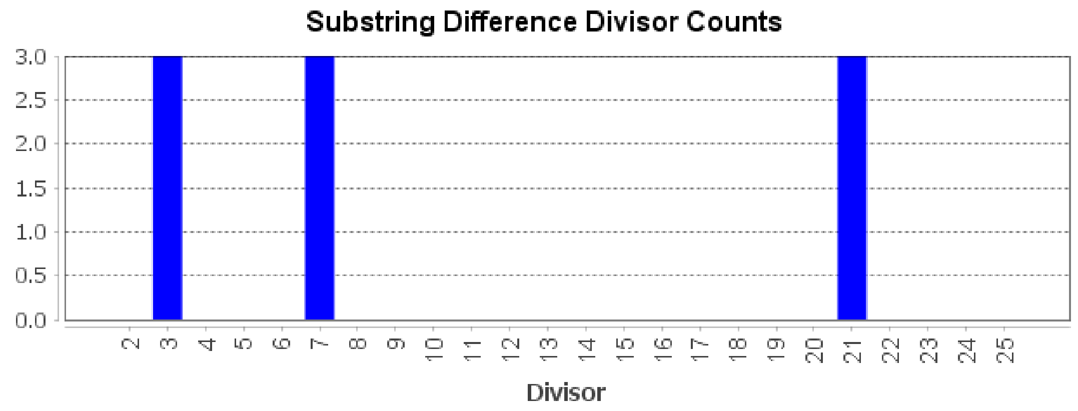


Figure 2.14: Kasiskis Method on the Vigenère Example

We get the following divisor counts of the substring matches of lengths 3, 4, and 5.

What this graph is representing is with substring lengths of 3, 4, or 5, there are three matches that are 21 characters apart. Hence the three divisor bars at 3, 7, and 21. There are also 3 matches at a distance of 37, but a 37 character keyword is unlikely. This 37 length was probably just a coincidence. Increasing the lengths of the substrings, up to length 100, does not change this.

So we would suspect that the keyword length was 3 or 7, or possibly 21, although 21 characters is getting a little long. We know that the keyword is ENCRYPT for this example, having length 7, but if we did not know, as we wouldn't, we would proceed to the second stage and try lengths 3 and 7. If both of these fail then length 21 might be a possibility. With the short ciphertext here, if the keyword was length 21 we would probably have a lot of trouble in the second stage.

Δ

Kasiskis Method could have been done, or discovered, in the mid sixteenth century. There is very little mathematics involved, simply counting lengths and finding divisors. But it took Kasiski to put it all together in the mid nineteenth century.

Coincidence Analysis

Coincidence Analysis tends to be a bit more telling about the length of a repeated Vigenère keyword. The idea here is that we take the ciphertext on one line and the same ciphertext shifted by a couple characters on the line below it and look to see if there are any matches between the ciphertext and the shifted ciphertext. For example, consider the following ciphertext from a repeated keyword Vigenère encryption. So that we do not confuse the shift of the ciphertext with the Vigenère shifts we will call the ciphertext shift a displacement.

Displacement of 2, one match.

```
CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
  CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
```

```
                              *
```

Displacement of 3, two matches.

```
CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
   CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
          *               *
```

Displacement of 4, five matches.

```
CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
    CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
         *          *       *              *   *
```

Displacement of 5, one match.

```
CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
     CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
                                     *
```

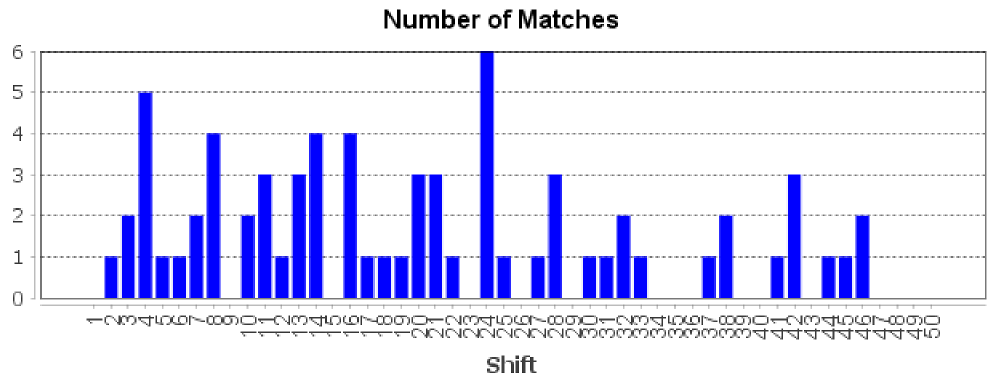


Figure 2.15: Coincidence Analysis on a Vigenère Encryption

If we continue this through a displacement of 50 we get the above bar chart of matches. Notice at a displacement of 4 we get a spike in the number of matches, as well as at 8, 14, 16, and 24. The spikes are where we have possible multiples of the keyword length. From the chart, it looks like the length of the keyword is probably 4. The spike at 14 and the lack of a spike at 2 would suggest that the spike at 14 is a coincidence. Also the lack of spikes at 12 and 20 would not concern us too much given the spikes at 4, 8, 16, and 24. The ciphertext contains only 54 characters so after 24 there is less than half the ciphertext being considered for the test, which will substantially decrease the match counts.

Example 16: If we apply Coincidence Analysis to the ciphertext we got from the repeated keyword encryption using the keyword ENCRYPT,

```
XUGMGVXRRTVAXILRTZQPFIGJFBDYIAEIWEMMAIRJEAEOGKGRMI
KVSWJLMAIRQTKMRUFDSBJSGICCMGNGJYGVMCJVPHUEFGUMCMLR
NVRIXVFQWYZXCJQIBXMMFCJGBIPRHFPBHJCQCWPETUCSCIBGFW
SQIBXHVZMC
```

We get the following match counts,

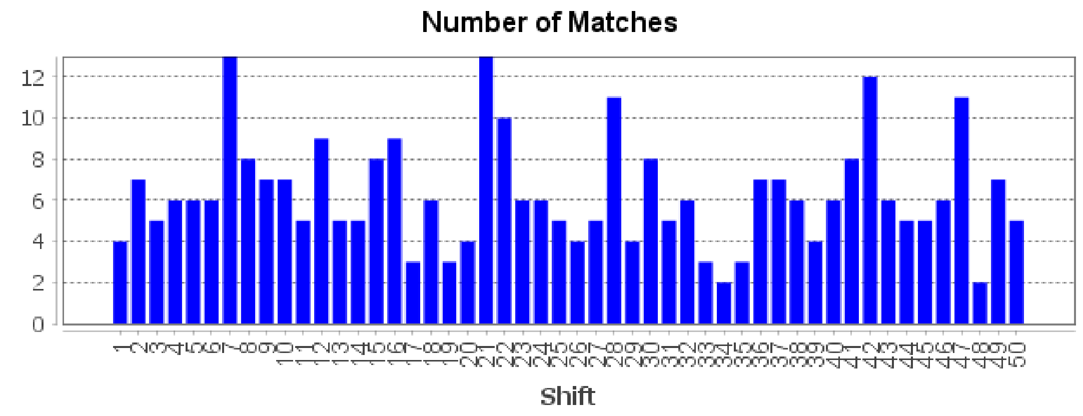


Figure 2.16: Coincidence Analysis on the Vigenère Example

The tallest spikes are at displacements of 7, 21, 28, 42, and 47. The GCD of the first four numbers is 7 and the last number, 47, would appear to be a fluke. The number 47 is prime, so if it had anything to do with the keyword length, the keyword would have 47 characters to it and it is unlikely in that event that we would have the other spikes at numbers that are all multiples of 7. This would suggest that our keyword has length 7, as we know it does. In this case though, there are no other clear possibilities, as there was with Kasiskis method.

Δ

It is very unlikely that the Coincidence Analysis method could have been conceived of in the mid sixteenth century. This method came after Kasiskis method, so it clearly was not discovered then, but our point is that the mathematics that makes this method work was not known at that time. Later on in this section, when we discuss the second stage of the cryptanalysis, we will talk about dot product analysis which relies on the same observation.

To illustrate the concept before we apply it, let's say we have an alphabet with three letters A, B, and C. We also have relative letter frequencies in our language, as we do in English, and for our language they are 0.1, 0.7 and 0.2. So A shows up 10% of the time, B shows up 70% of the time and C shows up 20% of the time, on average. Let's let A_0 represent the relative frequency vector for our language, so $A_0 = (0.1, 0.7, 0.2)$. Let's also let A_1 be the relative frequency vector for our language displaced by 1, so $A_1 = (0.2, 0.1, 0.7)$, and A_2 be the relative frequency vector for our language displaced by 2, so $A_2 = (0.7, 0.2, 0.1)$. In these cases we wrap the relative frequencies around. Now consider the dot products of A_0 with A_0, A_1, and A_2.

$A_0 \cdot A_0$	$0.1 \cdot 0.1 + 0.7 \cdot 0.7 + 0.2 \cdot 0.2$	0.54
$A_0 \cdot A_1$	$0.1 \cdot 0.2 + 0.7 \cdot 0.1 + 0.2 \cdot 0.7$	0.23
$A_0 \cdot A_2$	$0.1 \cdot 0.7 + 0.7 \cdot 0.2 + 0.2 \cdot 0.1$	0.23

Notice that $A_0 \cdot A_0$ is larger than the other two dot products. This is because in this product the larger relative frequencies match up with larger relative frequencies, whereas with the displacement vectors these larger relative frequencies matched up with smaller relative frequencies and produced a smaller sum. If you have taken a course in vector calculus this should be familiar to you, one commonly sees there that the dot product of two unit vectors is a maximum when the vectors are pointing in the same direction, that is, they are the same vector.

So how does this apply to Coincidence Analysis? When we displace the ciphertext by the length of the keyword, or a multiple of that length, then the characters are lining up with the same Vigenère shift. From our short example, when we had 5 matches with a shift of 4, which was the length of the keyword, the C and R were shifted the same amount during encryption. As were the A and P, and the L and T, and so on.

```
CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
    CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
         *          *       *              *   *
```

Since each column is using the same shift, any matches in the plaintext using this same displacement will be encrypted to the same ciphertext letter. As shown below, with the actual plaintext below the matches marks.

```
CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
    CPTDRALENAEZOBSPLWTYLQOPWKPLWIWJBQDZOBSPEQRPWMCPLQASNZ
         *          *       *              *   *
THISISATESTOFTHECOINCIDENCEANALYSISOFTHEVIGENERECIPHER
    THISISATESTOFTHECOINCIDENCEANALYSISOFTHEVIGENERECIPHER
```

Let's put this into more mathematical terms using our three character alphabet above, that is, A, B, and C with $A_0 = (0.1, 0.7, 0.2)$. Let's also assume that we have done a Vigenère cipher on some text from that language and got, `ACABBABCBBCBBAABCBBC`. Now apply coincidence analysis to the ciphertext.

Displacement of 1, five matches.

```
ACABBABCBBCBBAABCBBC
 ACABBABCBBCBBAABCBBC
    *    *  * *   *
```

Displacement of 2, five matches.

```
ACABBABCBBCBBAABCBBC
  ACABBABCBBCBBAABCBBC
  *   * *  *     *
```

Displacement of 3, nine matches.

```
ACABBABCBBCBBAABCBBC
   ACABBABCBBCBBAABCBBC
     **   ****  *   **
```

Displacement of 4, three matches.

```
ACABBABCBBCBBAABCBBC
    ACABBABCBBCBBAABCBBC
        *   *  *
```

Coincidence analysis would suggest a keyword length of three. Continuing with the same message gives us the following bar chart, which also suggests a keyword length of three, with spikes at 6 and 9.

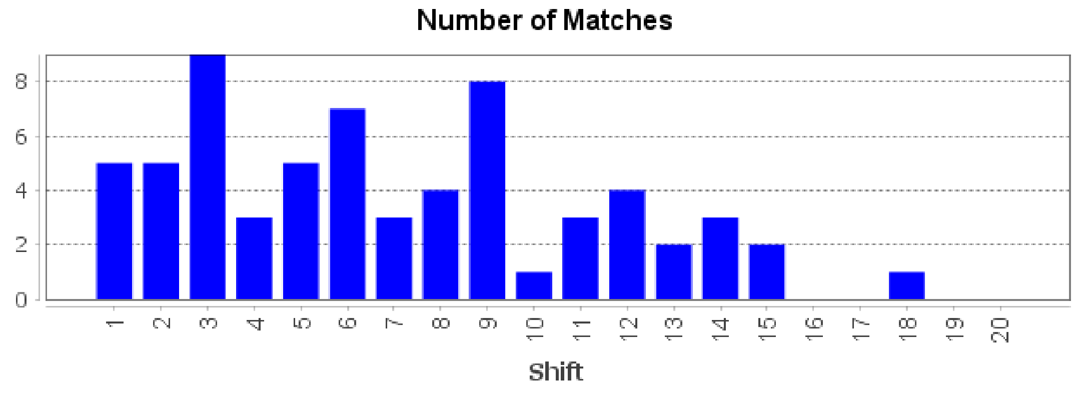


Figure 2.17: Coincidence Analysis on a Vigenère Encryption of Three Character Alphabet

Let's look at the displacement of 3 a little closer.

```
ACABBABCBBCBBAABCBBC
   ACABBABCBBCBBAABCBBC
     **   ****  *   **
```

What is the probability that we have a match in the ciphertext when we are displaced by the keyword length or a multiple of it? For this to happen we would have to have a match in the plaintext. So to get a match of two A's in the ciphertext, as the first match column above, we would have needed to have an A and an A in the plaintext in those positions (with no Vigenère shift applied) or a B and a B in those positions (with a Vigenère shift of 2) or a C and a C in those positions (with a Vigenère shift of 1). Remember with probabilities, and's are multiplications and or's are addition. So the probability of this happening is

$$0.1 \cdot 0.1 + 0.7 \cdot 0.7 + 0.2 \cdot 0.2 = 0.54 = A_0 \cdot A_0$$

Now let's look at the case where the displacement is not the keyword length or a multiple of it.

So to get a match of two A's in the ciphertext, we would have needed to have an A in the plaintext (with no Vigenère shift) and either a B or a C (with a Vigenère shift of 1 or 2) or we would have a B match up with an A or C (again with different Vigenère shifts) or we would have a C match up with either an A or B (again with different Vigenère shifts). So the probability of this happening is, or is close to,

$$0.1 \cdot 0.2 + 0.7 \cdot 0.1 + 0.2 \cdot 0.7 = 0.23 = A_0 \cdot A_1$$

or

$$0.1 \cdot 0.7 + 0.7 \cdot 0.2 + 0.2 \cdot 0.1 = 0.23 = A_0 \cdot A_2$$

So the probability of a match at displacements of the length or multiples of the length is greater, and hence the expected value of matches is greater. This is why Coincidence Analysis works, it is simply probabilities, driven by the character frequencies in the language.

Determining the Keyword

Once we have determined the keyword length, or at least have it narrowed down to a couple possibilities, we can determine the keyword fairly easily. First we extract all of the characters that are shifted by the same amount, for example, when we did coincidence analysis on the following ciphertext, we came to the conclusion that the keyword probably had length 7. So we will extract 7 sub texts. The first will be starting at character 1 and taking every 7th letter. The second will be starting at character 2 and taking every 7th letter, and so on.

```
XUGMGVXRRTVAXILRTZQPFIGJFBDYIAEIWEMMAIRJEAEOGKGRMI
KVSWJLMAIRQTKMRUFDSBJSGICCMGNGJYGVMCJVPHUEFGUMCMLR
NVRIXVFQWYZXCJQIBXMMFCJGBIPRHFPBHJCQCWPETUCSCIBGFW
SQIBXHVZMC
```

Subtext #1: `XRLIIMEIMMJGMELVCMPJTGX`
Subtext #2: `URRGAAOKARSNCFRFJFRCUFH`
Subtext #3: `GTTJEIGVIUGGJGNQQCHQCWV`
Subtext #4: `MVZFIRKSRFIJVUVWIJFCSSZ`
Subtext #5: `GAQBWJGWQDCYPMRYBGPWCQM`
Subtext #6: `VXPDEERJTSCGHCIZXBBPIIC`
Subtext #7: `XIFYMAMLKBMVUMXXMIHEBB`

We know that each of these sets of letters came from a Caesar shift, so we can simply treat each as its own separate ciphertext and attempt to find the shift. One thing to note here is that each of these seven sets does not contain very many letters, only 22 or 23. This is not much information to go on, in general we would like to have a bit more text. In effect, for the Vigenère cipher, the longer the keyword the better since the subtext lengths will be the length of the ciphertext divided by the keyword length. In the above example, the ciphertext has 160 letters, if we went with a keyword of length 3, we would have subtexts of lengths 53 and 54, giving more information for finding the word itself. On the other hand, if we went with a keyword of length 10, the subtexts would be of length 16, making it even more difficult then it is here.

If we have enough ciphertext to work with we can use the one of the following two methods, either we can do a frequency analysis, really just a shift analysis, on each of the subtexts or we can use a dot product method similar to the discussion of why coincidence analysis works. The dot product method tends to work better than shift analysis when the subtexts are short, as in the example above. We will begin with a shift analysis example using a little more ciphertext.

Example 17: Let's take as an example, Shakespeare's Sonnet #18 as our plaintext.[53]

> Shall I compare thee to a summer's day?
> Thou art more lovely and more temperate:
> Rough winds do shake the darling buds of May,
> And summer's lease hath all too short a date:
> Sometime too hot the eye of heaven shines,
> And often is his gold complexion dimmed,
> And every fair from fair sometime declines,
> By chance, or nature's changing course untrimmed:
> But thy eternal summer shall not fade,
> Nor lose possession of that fair thou ow'st,
> Nor shall death brag thou wand'rest in his shade,
> When in eternal lines to time thou grow'st,
> So long as men can breathe or eyes can see,
> So long lives this, and this gives life to thee.

Since this is a 400 year old sonnet, we would not expect it to conform exactly to modern English letter frequencies, but is it fairly close, as the following chart shows.

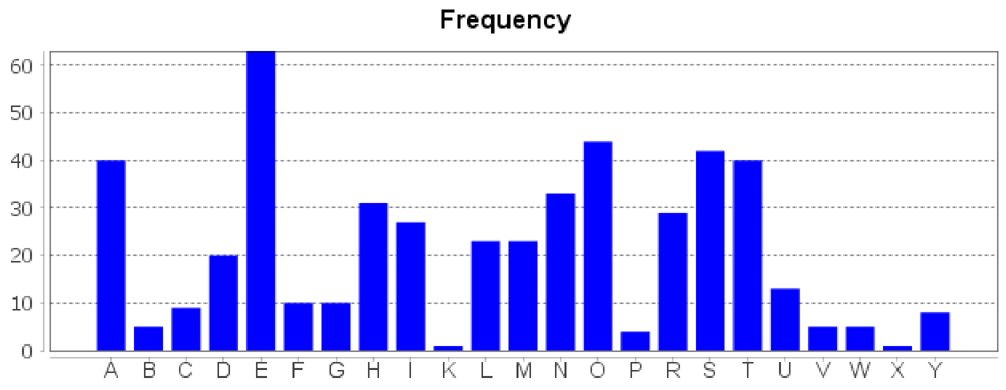


Figure 2.18: Frequency Analysis of Shakespeare's Sonnet #18

We will now encrypt this with the Vigenère cipher using the keyword SONNET.

```
KVNYPBUCZCEKWHURIMGOFHQFWFFQERLVBHEKLABEIEGJRYCTFR
```

```
ZBVXLSZCIKSHRESNYVJVRWKRBFLTCSGUIWSFYVRZTIQFSYEOLN
RWKIZZIKKZRNWXZOGUEEDHBBWAGFGNHTLSFBQXLWZRXHGVBGXA
WSLRSYZSNIIGKVVAILSBQBJMWBVFLBKUBYHVGACYIQACAQMFES
QNRWWJRECYSWESVHETNVVLGARGMFWRRPPBFSFOCVZOAPIHJBNG
YKWGPUEGYWATGHMFFRYGLFVZQXVPHGXAQSGRVGSZFHQFWFFUEE
DBBGJTVSABVEGGRCSLKSFFMHFCSGLTLTNVVMZCHBALLBBEWASZ
YQITLVOEEZLVBHATFRERWMABUVWLZOQRAAWBVAIMWFANPEABRF
XHLWZRXAGITESPKHFBPHFUNFQXFQNAFKWOGUIHJSLRWVSBFRIL
GZBAKEAJRFXAAGNAHMZWFTMOWGYVJXLCGUIX
```

If we assume that we do not know the keyword length and attack this with Kasiski's method and coincidence analysis we get the following two charts.

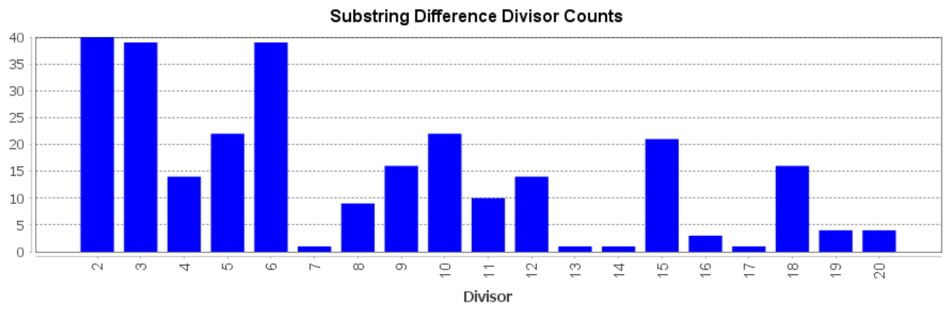


Figure 2.19: Kasiski's Method on Vigenère Example of Shakespeare's Sonnet #18

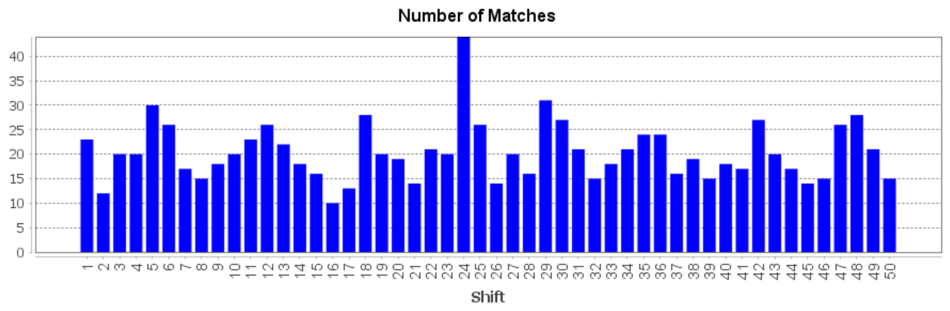


Figure 2.20: Coincidence Analysis on Vigenère Example of Shakespeare's Sonnet #18

Although it is difficult to push the fact that we know that the length of the keyword is 6 out of our mind, let's try to look at these charts as if we did not know this. Kasiski's method would suggest that the length was probably 6, due to the spike at 6 and the spikes at the divisors of 6. Small spikes at 12 and 18 would also support this conclusion. Although, there are spikes at 10 and 15, and one about the same height at 5. So a length of 5 might be a possibility, but the 2, 3, and 6 spikes would certainly count against that. Looking at the coincidence analysis chart, there is an overwhelming spike at 24, suggesting that the length is 24 or a multiple of 24 or a divisor of 24. Since a keyword length of 24 is approaching long we would probably think that this the keyword length is a divisor of 24. Looking at the coincidence analysis chart a little more closely, the shifts (really displacements) that produce 25 or more matches are, 5, 6, 12, 18, 25, 29, 30, 42, 47, and 48. Although 5, 25, and 30 are all multiples of 5, we have 6, 12, 18, 30, 42, and 48 that are all multiples of 6. The 29 and 47 are most likely flukes. All in all, even if we did not know the answer, we would have to say the most likely candidate for the length of the keyword would be 6. If we are wrong, it will show up later when we cannot get any recognizable plaintext from the ciphertext using a length of 6. At which point we would give 5 (or 3 or 2) a try.

At this point we will extract our subtexts, each have size of 81 characters, which should not be too bad to analyze.

```
KUWGWLLGFLSYKCSTEKKZDGLLGWZKSWKGAEWSEGWFZJWYMLVQSW
DVGKFLZLSLLFAZWWALGKFFWJSGAAZWL

VCHOFVAJRSHVRSFIOIZOHFSWVSSVBBUACSJWTARSOBGWFFPSZF
BSGSCTCBZVVRBOBFBWIHUQOSBZJGWGC

NZUFFBBRZZRJBGYQLZRGBGFZBLNVQVBCAQRENRRFANPAFVHGFF
BARFSNHBYOBEUQVARZTFNNGLFBRNFYG

YCRHQHEYBCEVFUVFNZNUBNBRGRIABFYYQNESVGPOPGUTRZGRHU
GBCFGVBEQEHRVRANFREBFAURRAFATVU

PEIQEEICVISRLIRSRIWEWHQXXSIIJLHIMRCVVMPCIYEGYQXVQE
JVSMLVAWIEAWWAIPXXSPQFIWIKXHMJI

BKMFRKETXKNWTWZYWKXEATXHAYGLMBVQFWYHLFBVHKGHGXAGFE
TELHTMLATZTMLAMEHAPHXKHVLEAMOXX
```

The frequency analysis charts are below.

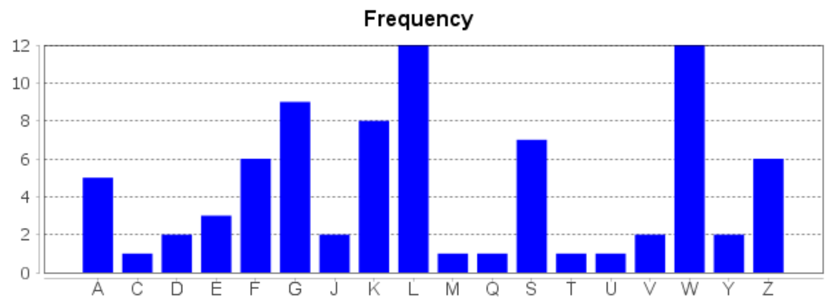


Figure 2.21: Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #1

From the analysis of the first subtext, we have our largest spikes at L and W. This would suggest that E was sent to either L or to W. If it was sent to L then the Caesar shift would have been $11 - 4 = 7$ which would correspond to a first letter of H. If it was sent to W then

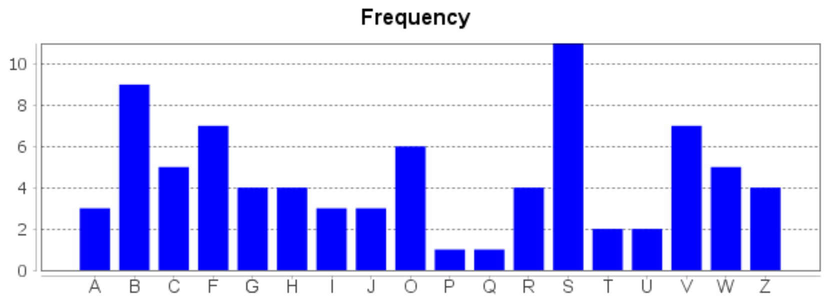


Figure 2.22: Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #2

the Caesar shift would have been $22 - 4 = 18$ which would correspond to a first letter of S. We know the answer, so we know that S is the first letter but if we did not then either would be a possibility at this point.

With subtext number 2, the largest spikes are at B and S. This would suggest that E was sent to B or S. If it was sent to B then the shift would be $1 - 4 = -3 \equiv 23 \pmod{26}$ corresponding to X, an unlikely choice for a second letter. If it was sent to S then the shift would be $18 - 4 = 14$ corresponding to O, a much more likely choice for a second letter.

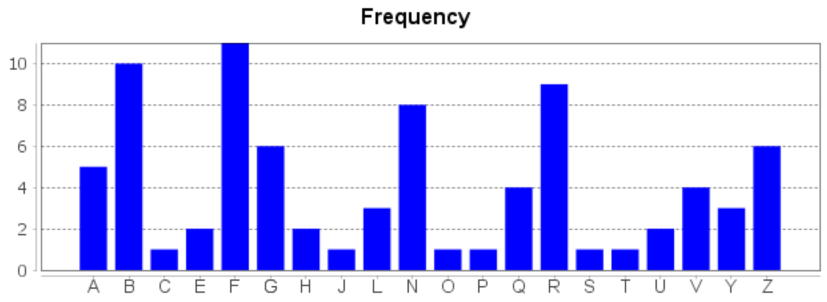


Figure 2.23: Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #3

With subtext number 3, the largest spikes are at B, F, and R, N would be a possibility as well. This would suggest that the shift letter was X, B, or N, with J as a possibility.

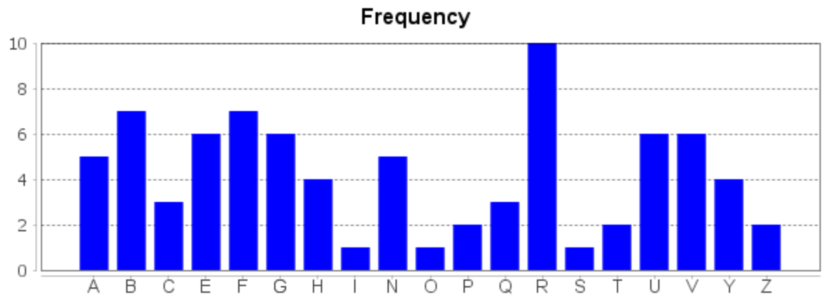


Figure 2.24: Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #4

With subtext number 4, there is an overwhelming spike at R. This would suggest that the shift letter was N. If this is a fluke and N does not fit into keyword we would reexamine this chart and probably take the next 6 most frequent letters.

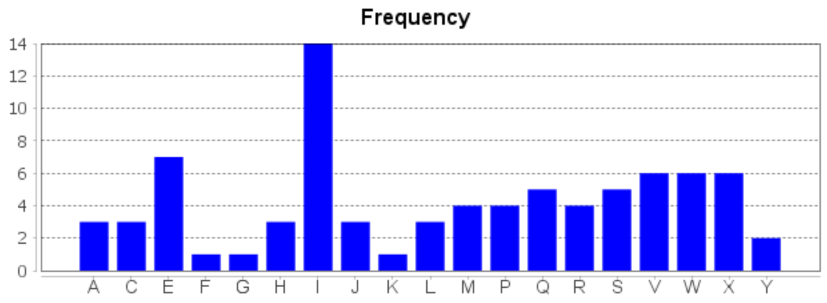


Figure 2.25: Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #5

With subtext number 5, there is an overwhelming spike at I. This would suggest that the shift letter was E. Again, if this is a fluke we would reexamine this chart later.

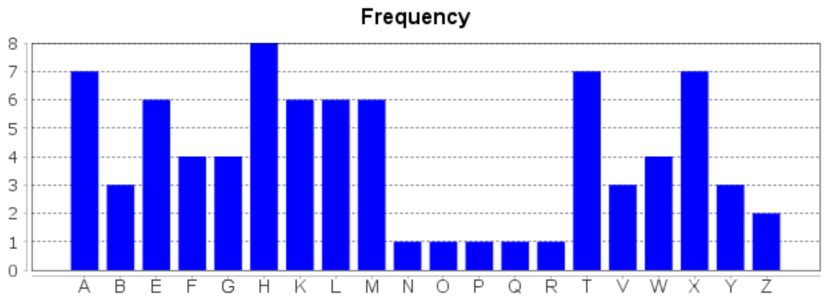


Figure 2.26: Frequency Analysis on Vigenère Example of Sonnet #18: Subtext #6

Finally, with subtext number 6, there are many spikes that are close together. The most frequent would be H, A, T, and X, with E, K, L, and M close behind. This would suggest the shift letters, in order of probability, of D, W, P, and T with A, G, H, and I close behind.

If we put the possibilities in a chart by letter we get the following, ignoring the second string of possibilities for the last letter for now.

		X			D
S	X	B	N	E	W
H	O	N			P
		J			T

If we assume that the keyword is an English word or phrase, as is common, the character for position two is most likely not an X and having X of J as a third character is unlikely. Also, ending a word with W is not very common. This would give us,

S H	O	B N	N	E	D P T

Assuming we are looking for an English word and not an abbreviation or set of abbreviations the keyword of SONNET appears to be our choice. At this point we would use the keyword SONNET with the ciphertext and decipher the message. If we found that we did not get an understandable message we could go back and change a letter to one of the other candidates. One nice thing about the Vigenère cipher is that if only one or two keyword characters are incorrect you can usually tell that from the decryption. There will be most of a recognizable word with a letter out of place. In fact, the placement of the wrong letter will show you which letter of the keyword is incorrect.

Δ

One of the downfalls to the above method is that we put all of our stock in where a single letter, E, was sent. We did not look at what would happen with the other letters. For example, say we considered the first letter again, it was probably an S or an H. If it was the H then E would have been sent to L, in this case, Z would have been sent to G, which has a high frequency from the subtext #1 chart, meaning that Z would have been a high frequency character in the plaintext, which is not very likely. Thus taking two letters into consideration gives a strong likelihood of S being the first letter. What would be better is if we could take all 26 letters into account, this is exactly what dot product analysis does.

Dot Product Analysis is a technique that takes the relative frequencies of all the characters in a set of text and determines the most likely Caesar shift that was done on that set of characters. More specifically, we take the relative letter frequencies for English and make a vector out of them (A–Z), we will call this vector A_0, recall the letter frequencies for English,

Letter	e	t	a	o	i	n	s	h	r	d	l	c	u
Freq. %	12.7	9.1	8.2	7.5	7	6.7	6.3	6.1	6	4.3	4	2.8	2.8

Letter	m	w	f	g	y	p	b	v	k	j	q	x	z
Freq. %	2.4	2.3	2.2	2	2	1.9	1.5	1	0.8	0.2	0.1	0.1	0.1

So,

$$\begin{aligned} A_0 = (&0.082, .015, 0.028, 0.043, 0.127, 0.022, 0.02, 0.61, 0.07, 0.002, 0.008, 0.04, 0.024,\\ &0.067, 0.075, 0.019, 0.001, 0.06, 0.063, 0.091, 0.028, 0.01, 0.023, 0.001, 0.02, 0.001)\end{aligned}$$

We will also define the shifts of this vector, let A_1 be a shift of 1, that is, A goes to B, B to C and so on to Z goes to A, so the relative frequency of A must be in the spot for B, B's relative frequency in the spot for C and so on to Z's relative frequency in the spot for A. So,

$$\begin{aligned} A_1 = (&0.001, 0.082, .015, 0.028, 0.043, 0.127, 0.022, 0.02, 0.61, 0.07, 0.002, 0.008, 0.04,\\ &0.024, 0.067, 0.075, 0.019, 0.001, 0.06, 0.063, 0.091, 0.028, 0.01, 0.023, 0.001, 0.02)\end{aligned}$$

Let A_2 be a shift of 2, and so on,

$$A_2 = (0.02, 0.001, 0.082, .015, 0.028, 0.043, 0.127, 0.022, 0.02, 0.61, 0.07, 0.002, 0.008, 0.04, 0.024, 0.067, 0.075, 0.019, 0.001, 0.06, 0.063, 0.091, 0.028, 0.01, 0.023, 0.001)$$

Now take some ciphertext that is a result of a Caesar shift and create its relative frequency vector, call it W. Since the ciphertext is a shift, then the relative frequency vector W should be close to one of the shift vectors, A_0, A_1, A_2, , …, A_{25}. If we then take the dot product of W with each shift vector, that is, calculate $W \cdot A_0$, $W \cdot A_1$, …, $W \cdot A_{25}$, the largest dot product will tell us what the shift probably was. Recall from the discussion of why coincidence analysis works, the shift vector that produces the largest dot product is most probably the shift that the plaintext had undergone to create the cyphertext.

Example 18: Take the sonnet and do a shift of 7 on it, this is the same as a Vigenère cipher with the keyword H, and we get the following,

```
ZOHSSPJVTWHYLAOLLAVHZBTTLYZKHFAOVBHYATVYLSVCLSFHUK
TVYLALTWLYHALYVBNODPUKZKVZOHRLAOLKHYSPUNIBKZVMTHFH
UKZBTTLYZSLHZLOHAOHSSAVVZOVYAHKHALZVTLAPTLAVVOVAAO
LLFLVMOLHCLUZOPULZHUKVMALUPZOPZNVSKJVTWSLEPVUKPTTL
KHUKLCLYFMHPYMYVTMHPYZVTLAPTLKLJSPULZIFJOHUJLVYUHA
BYLZJOHUNPUNJVBYZLBUAYPTTLKIBAAOFLALYUHSZBTTLYZOHS
SUVAMHKLUVYSVZLWVZZLZZPVUVMAOHAMHPYAOVBVDZAUVYZOHS
SKLHAOIYHNAOVBDHUKYLZAPUOPZZOHKLDOLUPULALYUHSSPULZ
AVAPTLAOVBNYVDZAZVSVUNHZTLUJHUIYLHAOLVYLFLZJHUZLLZ
VSVUNSPCLZAOPZHUKAOPZNPCLZSPMLAVAOLL
```

If we calculate the relative frequencies of each character in this ciphertext we get the following table.

Characters	Frequency	Relative Frequency
A	40	0.0823045267489712
B	13	0.026748971193415638
C	5	0.0102880658436214
D	5	0.0102880658436214
E	1	0.00205761316872428
F	8	0.01646090534979424
H	40	0.0823045267489712
I	5	0.0102880658436214
J	9	0.018518518518518517
K	20	0.0411522633744856
L	63	0.12962962962962962
M	10	0.0205761316872428
N	10	0.0205761316872428

Characters	**Frequency**	**Relative Frequency**
O	31	0.06378600823045268
P	27	0.05555555555555555
R	1	0.00205761316872428
S	23	0.047325102880658436
T	23	0.047325102880658436
U	33	0.06790123456790123
V	44	0.09053497942386832
W	4	0.00823045267489712
Y	29	0.059670781893004114
Z	42	0.08641975308641975

Notice that L has a relative frequency of 0.1296, very close to the 12.7% that E has. This would suggest that E was sent to L making the shift 7, which is the case. Putting this aside, let's calculate the dot products. Convert the above chart to the vector W, so $W = (0.082, 0.027, \ldots, 0.002)$. Now calculate $W \cdot A_0$, $W \cdot A_1$, $\ldots$, $W \cdot A_{25}$. We will identify the shift with the shift character, so a shift of 0 corresponds to the vector A_0 and the letter A, a shift of 1 corresponds to the vector A_1 and the letter B, and so on. With this association, we have the following dot products,

Shift Character	**Dot Product**
A	0.03911316872427983
B	0.03574074074074074
C	0.03307613168724279
D	0.04285802469135802
E	0.03375925925925926
F	0.03134362139917696
G	0.04232921810699588
H	0.06670987654320988
I	0.03951234567901235
J	0.02895884773662551
K	0.033913580246913586
L	0.04499382716049382
M	0.03319135802469136
N	0.035711934156378605
O	0.03969958847736625
P	0.03324691358024691
Q	0.03319753086419753
R	0.03993621399176955
S	0.04465020576131687
T	0.03969547325102881
U	0.042265432098765425
V	0.0408724279835391

Shift Character	Dot Product
W	0.04136625514403293
X	0.037197530864197526
Y	0.032
Z	0.03466049382716049

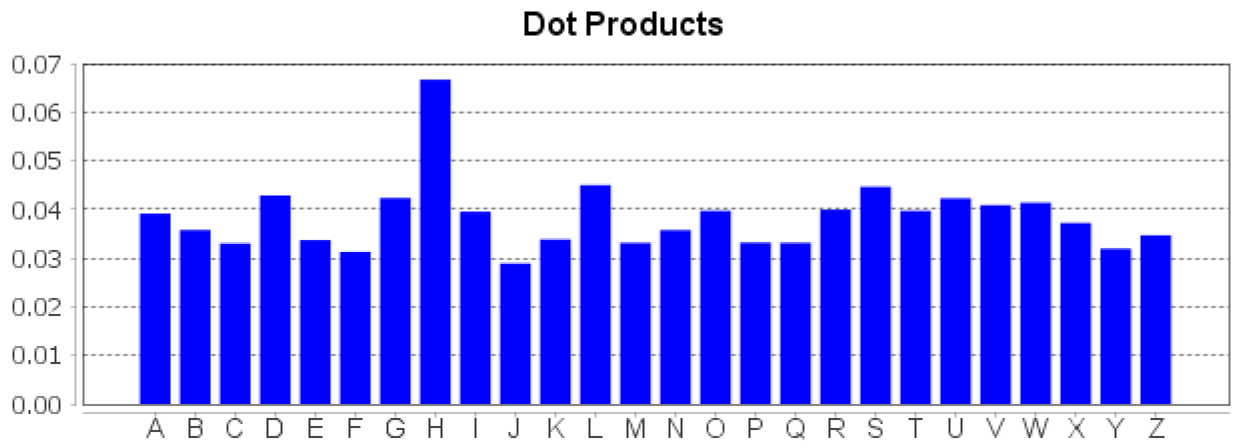


Figure 2.27: Dot Product Analysis on a Shift Cipher

It is clear that the largest dot product, by far, is at the character H, corresponding to a shift of 7.

Δ

The nice thing about this method is that it takes all 26 letters into account, so it is usually very obvious from dot product analysis what the shift amount was. Furthermore, since it does take all 26 letters into account, this method tends to work better for smaller amounts of ciphertext. This comes in handy when finding the keyword to a Vigenère cipher, since the subtexts can be relatively short. To illustrate, we will redo the Vigenère cipher we did on Sonnet #18 with keyword SONNET.

Example 19: Recall that the ciphertext was the following and that the keyword was SONNET.

```
KVNYPBUCZCEKWHURIMGOFHQFWFFQERLVBHEKLABEIEGJRYCTFR
ZBVXLSZCIKSHRESNYVJVRWKRBFLTCSGUIWSFYVRZTIQFSYEOLN
RWKIZZIKKZRNWXZOGUEEDHBBWAGFGNHTLSFBQXLWZRXHGVBGXA
WSLRSYZSNIIGKVVAILSBQBJMWBVFLBKUBYHVGACYIQACAQMFES
QNRWWJRECYSWESVHETNVVLGARGMFWRRPPBFSFOCVZOAPIHJBNG
YKWGPUEGYWATGHMFFRYGLFVZQXVPHGXAQSGRVGSZFHQFWFFUEE
DBBGJTVSABVEGGRCSLKSFFMHFCSGLTLTNVVMZCHBALLBBEWASZ
YQITLVOEEZLVBHATFRERWMABUVWLZOQRAAWBVAIMWFANPEABRF
XHLWZRXAGITESPKHFBPHFUNFQXFQNAFKWOGUIHJSLRWVSBFRIL
GZBAKEAJRFXAAGNAHMZWFTMOWGYVJXLCGUIX
```

We will also assume that we have done the same analysis on the keyword length and have decided to use a length of 6 characters. The subtexts are below. Also recall that the frequency analysis we did was inconclusive for several letters.

```
KUWGWLLGFLSYKCSTEKKZDGLLGWZKSWKGAEWSEGWFZJWYMLVQSW
DVGKFLZLSLLFAZWWALGKFFWJSGAAZWL

VCHOFVAJRSHVRSFIOIZOHFSWVSSVBBUACSJWTARSOBGWFFPSZF
BSGSCTCBZVVRBOBFBWIHUQOSBZJGWGC

NZUFFBBRZZRJBGYQLZRGBGFZBLNVQVBCAQRENRRFANPAFVHGFF
BARFSNHBYOBEUQVARZTFNNGLFBRNFYG

YCRHQHEYBCEVFUVFNZNUBNBRGRIABFYYQNESVGPOPGUTRZGRHU
GBCFGVBEQEHRVRANFREBFAURRAFATVU

PEIQEEICVISRLIRSRIWEWHQXXSIIJLHIMRCVVMPCIYEGYQXVQE
JVSMLVAWIEAWWAIPXXSPQFIWIKXHMJI

BKMFRKETXKNWTWZYWKXEATXHAYGLMBVQFWYHLFBVHKGHGXAGFE
TELHTMLATZTMLAMEHAPHXKHVLEAMOXX
```

Using these, we create all of the dot product charts.

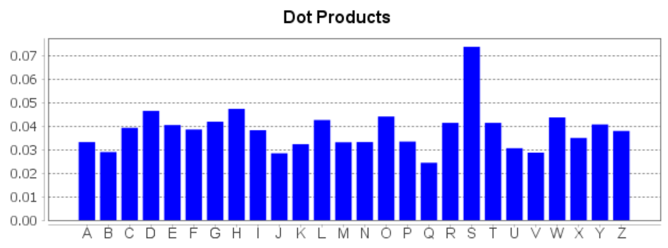


Figure 2.28: Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #1

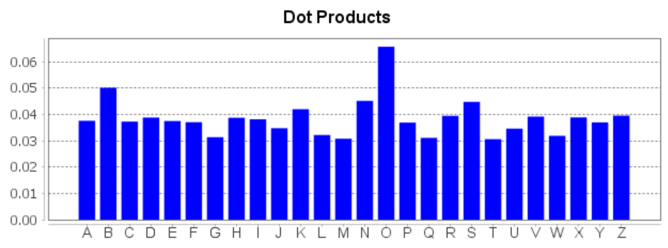


Figure 2.29: Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #2

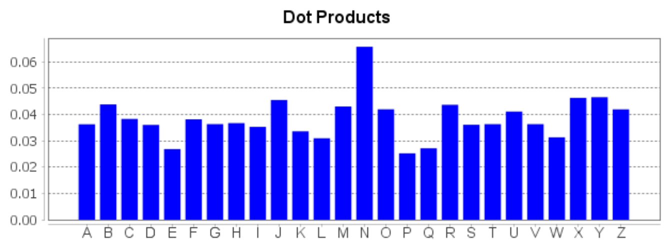


Figure 2.30: Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #3

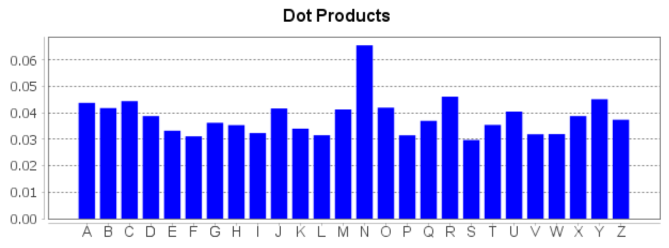


Figure 2.31: Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #4

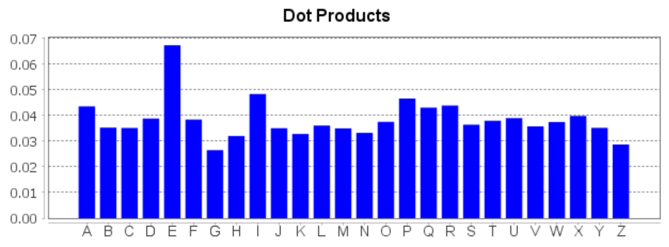


Figure 2.32: Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #5

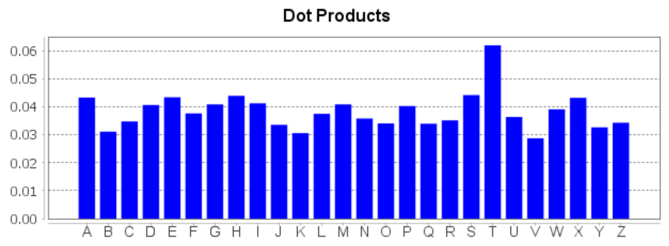


Figure 2.33: Dot Product Analysis on Vigenère Example of Sonnet #18: Subtext #6

Taking the longest bar in each case gives us SONNET as the keyword. In fact, in all of the charts, there is no bar that is even close to the tallest one. Again this is due to taking all 26 character into account instead of just one or two.

Δ

To illustrate the power of dot product analysis lets go back to a previous example where the subtexts we got after extraction were short.

Example 20: As before we will use the following plaintext with the keyword ENCRYPT.

```
THEVIGENERECIPHERISAMETHODOFENCRYPTINGALPHABETICTE
XTBYUSINGASERIESOFDIFFERENTCAESARCIPHERSBASEDONTHE
LETTERSOFAKEYWORDITISASIMPLEFORMOFPOLYALPHABETICSU
BSTITUTION
```

The final ciphertext for the encryption is,

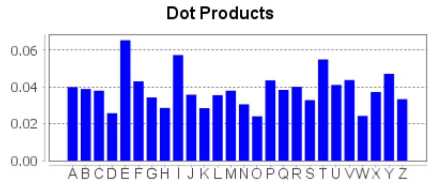
```
XUGMGVXKYXQIDMRCOYADYVVVMDRDZIXDBGWHVDDMVDHWHWUXWL
TAXSROTGGXKVFBKYLPYNGPCCTLGIPGUTCIQENYKUJQWRBYHCXA
CFRAGOSQKRKKMOEBUSDUGETCESFKJHTQGKZXSRQRZGXTVJZBYR
UNCHUSKCBP
```

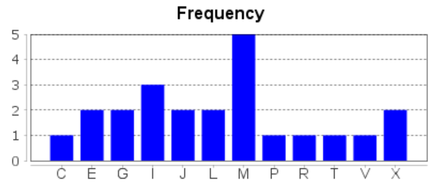
We have already gone through a coincidence analysis of this ciphertext and determined that the keyword length was most likely 7. Extracting our subtexts gives us the following, as before there are only 22 or 23 characters in each of these subtexts.

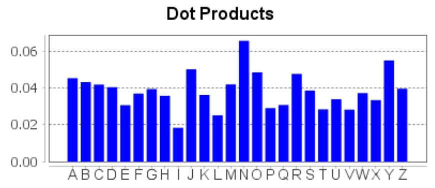
Subtext #1: `XRLIIMEIMMJGMELVCMPJTGX`
Subtext #2: `URRGAAOKARSNCFRFJFRCUFH`
Subtext #3: `GTTJEIGVIUGGJGNQQCHQCWV`
Subtext #4: `MVZFIRKSRFIJVUVWIJFCSSZ`
Subtext #5: `GAQBWJGWQDCYPMRYBGPWCQM`
Subtext #6: `VXPDEERJTSCGHCIZXBBPIIC`
Subtext #7: `XIFYMAMLKBMVUMXXMIHEBB`

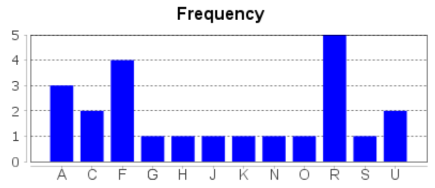
Now we will display the frequency analysis charts and the dot product analysis charts for each of these subtexts.

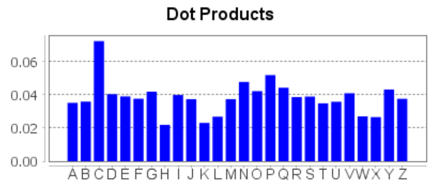
Dot Products | **Frequency Analysis**

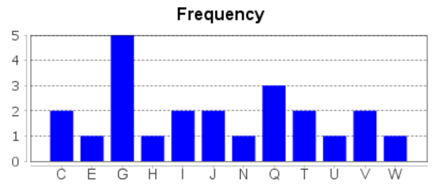




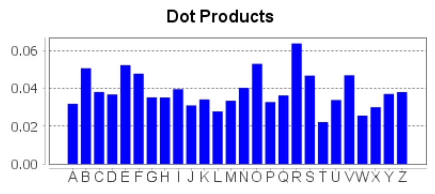


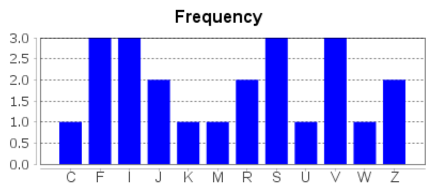


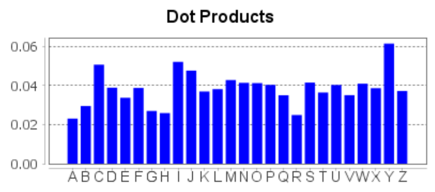


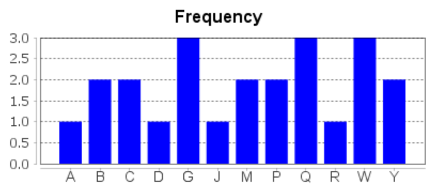


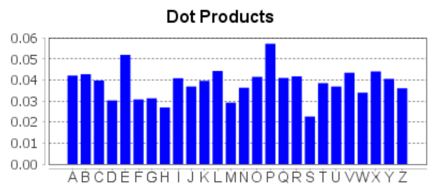
Dot Products | **Frequency Analysis**

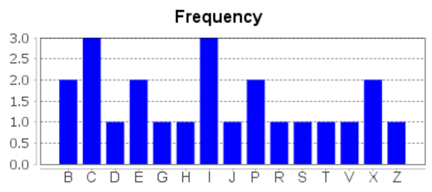


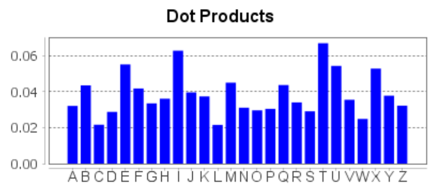


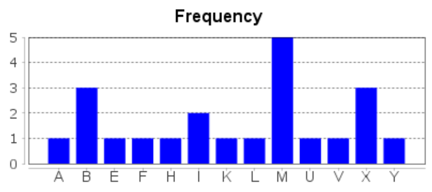












If we blindly take the largest dot product shift characters we get ENCRYPT as our keyword, the correct keyword. The frequency analysis charts are far less conclusive.

Δ

Final Remarks on the Cryptanalysis of the Vigenère Cipher

We have spent a lot of time on a cipher that has not been used for over a century, why? Several reasons, first, it is a nice example of a cipher that could have potentially been broken as soon as it was created, but it took 300 years for people to put the right pieces together. Kasiski's method used very simple observations and little mathematics, and using letter frequencies to do a shift analysis was easy and known as well. In fact, it was the knowledge of frequency analysis that necessitated the creation of a new, more powerful, cipher like the Vigenère.

Second, although the mathematics needed to break a Vigenère cipher existed at the time, newer and better methods were developed that could not have been conceived of in the mid sixteenth century, namely coincidence analysis and dot product analysis. Although a dot product is easy to calculate, its development did not occur until about the mid nineteenth century, and at that, it was not called a dot product or scalar product, those names are relatively modern. The concept of the dot product or scalar product came out of the study of complex numbers and the quaternions.[8] In the mid sixteenth century mathematicians were still having difficulties with negative numbers, let alone complex numbers which were usually dismissed as having no meaning. At about the same time the solution to the cubic equation was discovered which first gave a meaning to complex numbers, but it would take over a century after that before they were really investigated and understood.[11] In fact, some of Euler's early works in the beginning to mid eighteenth century dealt with optimizations that would eventually be recognized as the dot product maximization behind the coincidence analysis and dot product analysis techniques.[51]

The attacks we have discussed have been ciphertext only attacks. The other attacks all have some type of crib that is either available or is acquired in some fashion. If we have a crib and know where the correspondence is located then a simple subtraction will give us the letter shifts. With enough of these we could establish the keyword.

In general, when we discuss attacks on a system we will usually not discuss each type of attack but simply state that if we have acquired some specific information we can attack the system by applying this method, or given that we have nothing but the ciphertext we can attack the system by this method.

Another thing to note here is that using Kasiski's method and frequency analysis is an attack that could be done fairly easily by hand but once we get to dot product analysis the computations become a little more cumbersome. They can still be done by hand, even without a calculator, but the time to crack the cipher would be much longer. Nonetheless, this added calculation complexity tends to give better, more definitive, results.

We really have not discussed the keyspace or brute force attacks on this system. The keyspace is essentially infinite. Any word or phrase or in general any string of characters can be used as a keyword. There is no restriction on its size. Hence a brute force attack would be out of the question. Even if we used Kasiski's method or coincidence analysis and determined that the length of the keyword was n, a brute force attack would take the examination of 26^n keys. It would not take a very long keyword to make this a difficult problem.

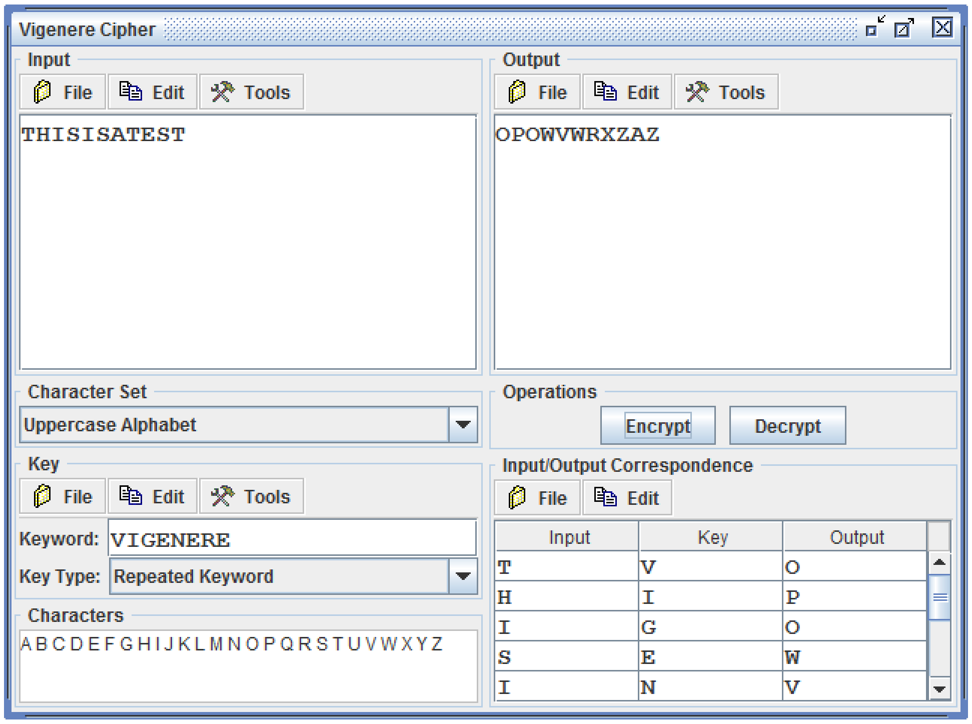
If the keyword of a Vigenère cipher is the same length as the message then there is no repetition of the keyword in the ciphertext, which drives Kasiski's method and coincidence analysis, and frankly stops our analysis method dead in its tracks. Using a long keyword is not common but if it is the same length as the message then we have a system that is close to a one-time-pad. We will look at one-time-pads in more detail later, but if we chose a *random* keyword that is the same length as our message then we would be incapable of breaking a Vigenère cipher using that keyword, outside of a brute force attack.

2.3.3 Cryptography Explorer

There are quite a few tools we need to have at our disposal to use and break a Vigenère cipher. First we will look at a tool to encrypt and decrypt the cipher, then wee will look at some tools for cryptanalysis of the cipher, these include tools for, Kasiski's method, coincidence analysis, frequency analysis, dot product analysis, and a tool for extracting every n^{th} character to more easily build the subtexts.

Vigenère Cipher Tool

Cryptography Explorer has a built-in tool for encryption and decryption of a Vigenère cipher, select Ciphers > Vigenère Cipher from the main menu and the following dialog will appear.



Input: THISISATEST

Output: 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 2.34: Vigenère Cipher Tool

The upper half of the dialog is the input and output windows. Each has file and printing options, editing options and tools. The input box has several quick text conversion tools as well as an option for doing all conversions the program allows. The lower right quarter of the window contains the encrypt and decrypt buttons as well as an input/output correspondence grid. The input/output correspondence grid has file saving and printing options as well as options to copy the grid's contents to the clipboard in either text format or several types of LaTeX table formats.

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 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.

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 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 Vigenre 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.

You can find a more detailed description of this tool and the options that are available for each of the parts of the tool in the appendix on the Cryptography Explorer program.

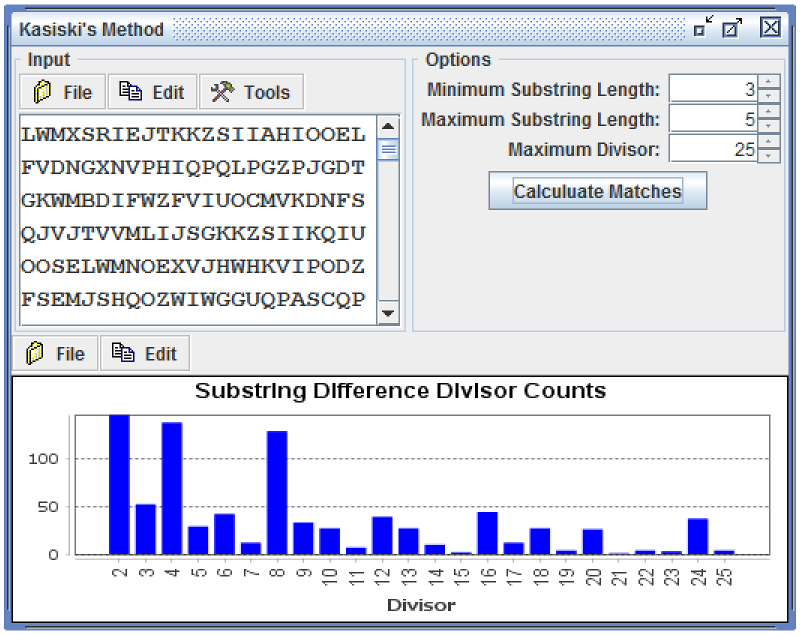
Kasiski's Method Tool

The Kasiski's Method window will report the divisor counts of the distances between equal substrings of a given length of the input. The upper left quarter of the dialog is a standard input box, with file loading and saving options, printing options, copy and paste options, and text conversion options. The upper right quarter has options for what length substrings to analyze and the maximum divisor of the distances to track. The lower half of the dialog is a chart of the divisor counts for each integer from 2 to the maximum divisor that you set.

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.

You can find a more detailed description of this tool and the options that are available for each of the parts of the tool in the appendix on the Cryptography Explorer program.



```
LWMXSRIEJTKKZSIIAHIOOEL
FVDNGXNVPHIQPQLPGZPJGDT
GKWMBDIFWZFVIUOCMVKDNFS
QJVJTVVMLIJSGKKZSIIKQIU
OOSELWMNOEXVJHWHKVIPODZ
FSEMJSHQOZWIWGGUQPASCQP
```

Minimum Substring Length: 3
Maximum Substring Length: 5
Maximum Divisor: 25

Figure 2.35: Kasiski's Method Analysis Tool

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. The upper left portion of the dialog contains a standard input box, with file loading and saving options, printing options, copy and paste options, and text conversion options. The upper right portion has an option to set the maximum shift to do on the text. The lower half of the dialog is a chart of the number of matches of characters for each shift from 1 to the maximum shift that you set.

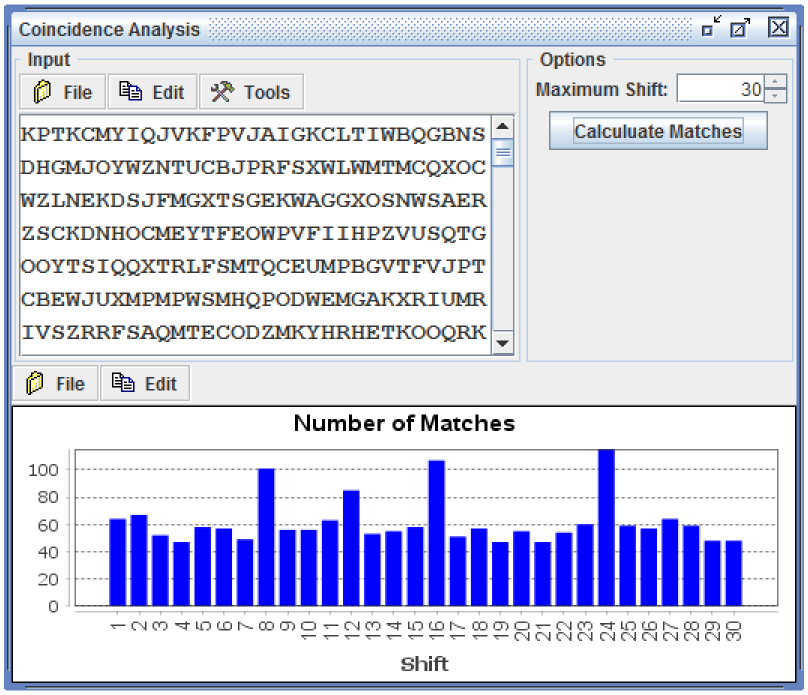
To use the tool,

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.

You can find a more detailed description of this tool and the options that are available for each of the parts of the tool in the appendix on the Cryptography Explorer program.

Frequency Analysis

If you have already red about the Frequency Analysis tool in the monoalphabetic substitution section you can skip this discussion. The Frequency Analysis window will analyze text for character, digram, trigram, or n-gram frequencies. There is also an option to interlace or not



Maximum Shift: 30

```
KPTKCMYIQJVKFPVJAIGKCLTIWBQGBNS
DHGMJOYWZNTUCBJPRFSXWLWMTMCQXOC
WZLNEKDSJFMGXTSGEKWAGGXOSNWSAER
ZSCKDNHOCMEYTFEOWPVFIIHPZVUSQTG
OOYTSIQQXTRLFSMTQCEUMPBGVTFVJPT
CBEWJUXMPMPWSMHQPODWEMGAKXRIUMR
IVSZRRFSAQMTECODZMKYHRHETKOOQRK
```

Figure 2.36: Coincidence Analysis Tool

interlace the blocks of characters. For using this method to find the keyword of a Vigenère cipher, you will only need to use single character analysis, so we will not discuss the other options of this tool here.

The upper half of the dialog contains a standard input box, with file loading and saving options, printing options, copy and paste options, and text conversion options. Also in the upper half of the dialog, on the right, are the analysis options, you will keep these as Single and Interlaced (Interlaced and Non-Interlaced are equivalent in Single mode). The lower half of the dialog is a report section that displays a grid of characters, frequencies and relative frequencies along with a chart of the information. You can find a more detailed description of this tool and the options that are available for each of the parts of the tool in the appendix on the Cryptography Explorer program.

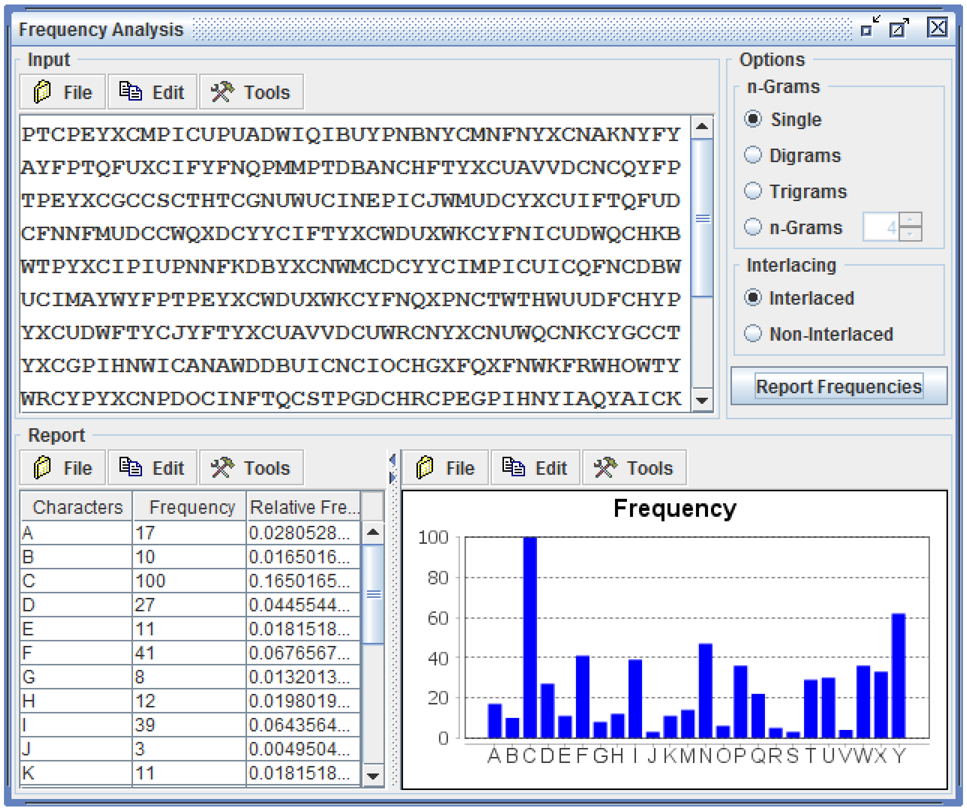
To use the tool,

1. Input the ciphertext into the Input box.

2. Select the analysis option, either single characters, digrams, trigrans, or n-grams. Again, for the Vigenère cipher you would select single characters.

3. Select the interlacing option of either interlaced or not interlaced. For the Vigenère cipher you will be in single character mode so this option is irrelevant.

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.



```
PTCPEYXCMPICUPUADWIQIBUYPNBNYCMNFNYXCNAKNYFY
AYFPTQFUXCIFYFNQPMMPTDBANCHFTYXCUAVVDCNCQYFP
TPEYXCGCCSCTHTCGNUWUCINEPICJWMUDCYXCUIFTQFUD
CFNNFMUDCCWQXDCYYCIFTYXCWDUXWKCYFNICUDWQCHKB
WTPYXCIPIUPNNFKDBYXCNWMCDCYYCIMPICUICQFNCDBW
UCIMAYWYFPTPEYXCWDUXWKCYFNQXPNCTWTHWUUDFCHYP
YXCUDWFTYCJYFTYXCUAVVDCUWRCNYXCNUWQCNKCYGCCT
YXCGPIHNWICANAWDDBUICNCIOCHGXFQXFNWKFRWHOWTY
WRCYPYXCNPDOCINFTQCSTPGDCHRCPEGPIHNYIAQYAICK
```

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 2.37: Frequency Analysis Tool

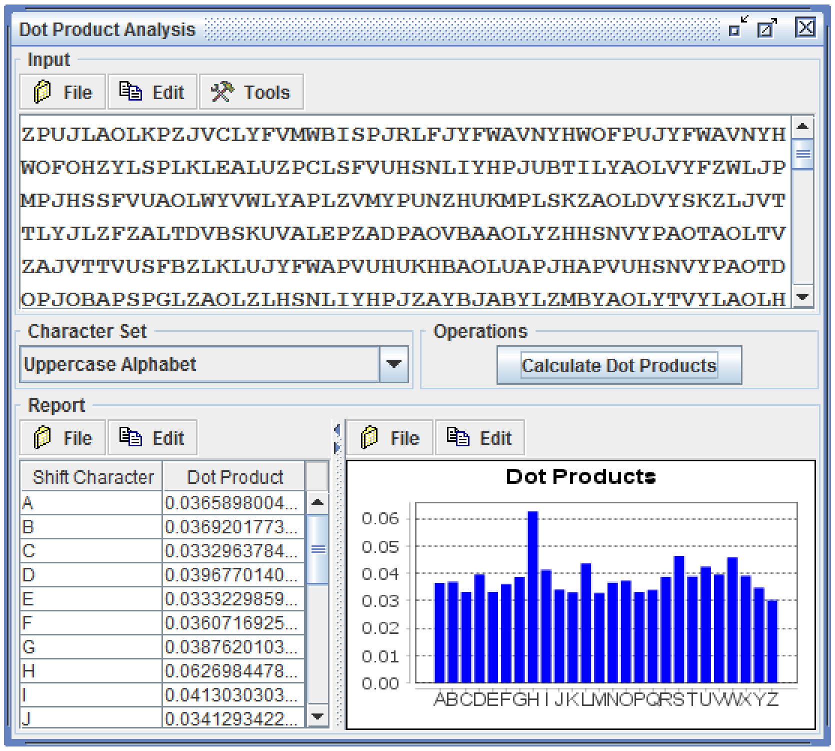
Dot Product Analysis

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.

The upper half of the dialog contains a standard input box, with file loading and saving options, printing options, copy and paste options, and text conversion options. There is also an option to set the character set to be used, either the uppercase English alphabet or a user defined language is available. The lower half of the dialog is a report section that displays a grid of dot product values and a chart of the same data. The shifts are displayed as the character of the shift and not as a numeric shift value.

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



```
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 2.38: Dot Product Analysis Tool

possible shifts will be displayed in the report table and the same data will be displayed in the report bar chart.

You can find a more detailed description of this tool and the options that are available for each of the parts of the tool in the appendix on the Cryptography Explorer program.

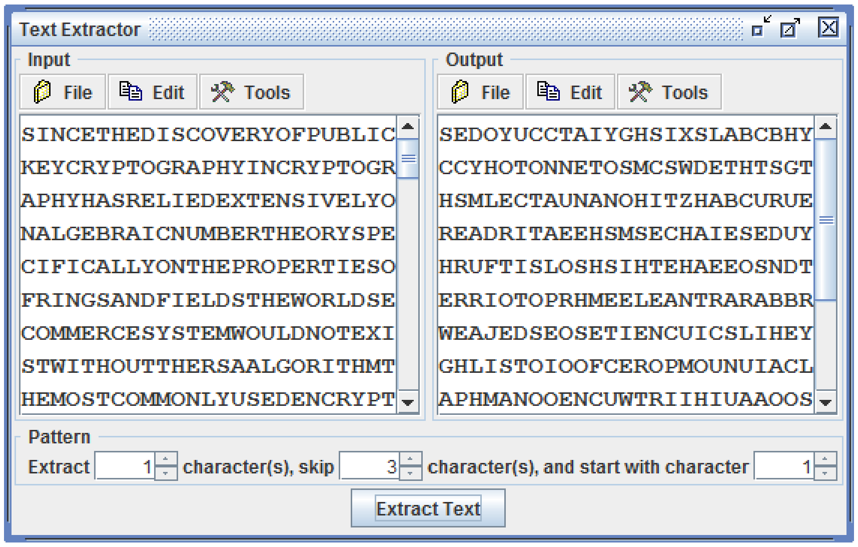
Text Extractor

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.

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.

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



```
SINCETHEDISCOVERYOFPUBLIC
KEYCRYPTOGRAPHYINCRYPTOGR
APHYHASRELIEDEXTENSIVELYO
NALGEBRAICNUMBERTHEORYSPE
CIFICALLYONTHEPROPERTIESO
FRINGSANDFIELDSTHEWORLDSE
COMMERCESYSTEMWOULDNOTEXI
STWITHOUTTHERSAALGORITHMT
HEMOSTCOMMONLYUSEDENCRYPT
```

```
SEDOYUCCTAIYGHSIXSLABCBHY
CCYHOTONNETOSMCSWDETHTSGT
HSMLECTAUNANOHITZHABCURUE
READRITAEEHSMSECHAIESEDUY
HRUFTISLOSHSIHTEHAEEOSNDT
ERRIOTOPRHMEELEANTRARABBR
WEAJEDSEOSETIENCUICSLIHEY
GHLISTOIOOFCEROPMOUNUIACL
APHMANOOENCUWTRIIHIUAAOOS
```

Extract 1 character(s), skip 3 character(s), and start with character 1

Figure 2.39: Text Extractor Tool

pattern is to extract 1 character, skip 3 starting at 2 then the extracted text would be ROP. You can find a more detailed description of this tool and the options that are available for each of the parts of the tool in the appendix on the Cryptography Explorer program.

Since there are a lot of tools involved in the cryptanalysis of the Vigenère cipher, we will go through the Sonnet #18 example using the Cryptography Explorer program.

Example 21: Recall that Shakespeare's Sonnet #18 was our plaintext.[53]

> Shall I compare thee to a summer's day?
> Thou art more lovely and more temperate:
> Rough winds do shake the darling buds of May,
> And summer's lease hath all too short a date:
> Sometime too hot the eye of heaven shines,
> And often is his gold complexion dimmed,
> And every fair from fair sometime declines,
> By chance, or nature's changing course untrimmed:
> But thy eternal summer shall not fade,
> Nor lose possession of that fair thou ow'st,
> Nor shall death brag thou wand'rest in his shade,
> When in eternal lines to time thou grow'st,
> So long as men can breathe or eyes can see,
> So long lives this, and this gives life to thee.

Encrypting this with the Vigenère cipher using the keyword SONNET gives,

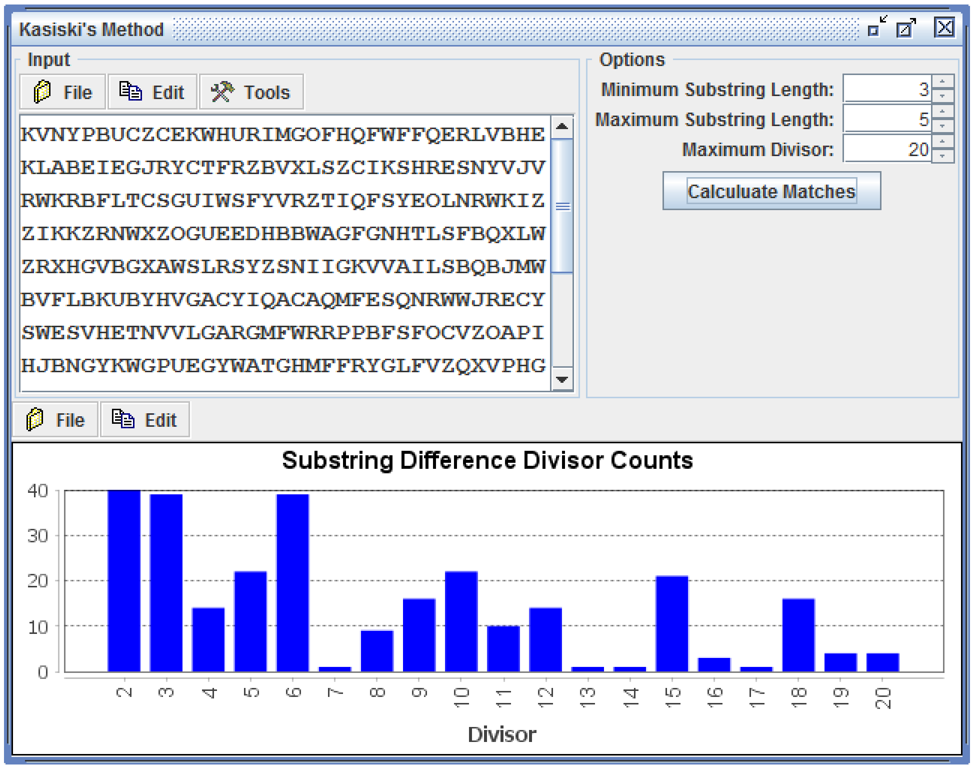
```
KVNYPBUCZCEKWHURIMGOFHQFWFFQERLVBHEKLABEIEGJRYCTFR
ZBVXLSZCIKSHRESNYVJVRWKRBFLTCSGUIWSFYVRZTIQFSYEOLN
RWKIZZIKKZRNWXZOGUEEDHBBWAGFGNHTLSFBQXLWZRXHGVBGXA
```

```
WSLRSYZSNIIGKVVAILSBQBJMWBVFLBKUBYHVGACYIQACAQMFES
QNRWWJRECYSWESVHETNVVLGARGMFWRRPPBFSFOCVZOAPIHJBNG
YKWGPUEGYWATGHMFFRYGLFVZQXVPHGXAQSGRVGSZFHQFWFFUEE
DBBGJTVSABVEGGRCSLKSFFMHFCSGLTLTNVVMZCHBALLBBEWASZ
YQITLVOEEZLVBHATFRERWMABUVWLZOQRAAWBVAIMWFANPEABRF
XHLWZRXAGITESPKHFBPHFUNFQXFQNAFKWOGUIHJSLRWVSBFRIL
GZBAKEAJRFXAAGNAHMZWFTMOWGYVJXLCGUIX
```

Determining the Keyword Length

Recall that we can determine the keyword length by applying either Kasiski's method or coincidence analysis. We will go through both methods.

For Kasiski's method, select Tools > Text & Stream Analysis > Kasiski's Method from the main menu. Copy the ciphertext to the input box, leave the options as they are, minimum substring length of 3, maximum substring length of 5 and maximum divisor of 20. Click the Calculate Matches button and you should see the following,



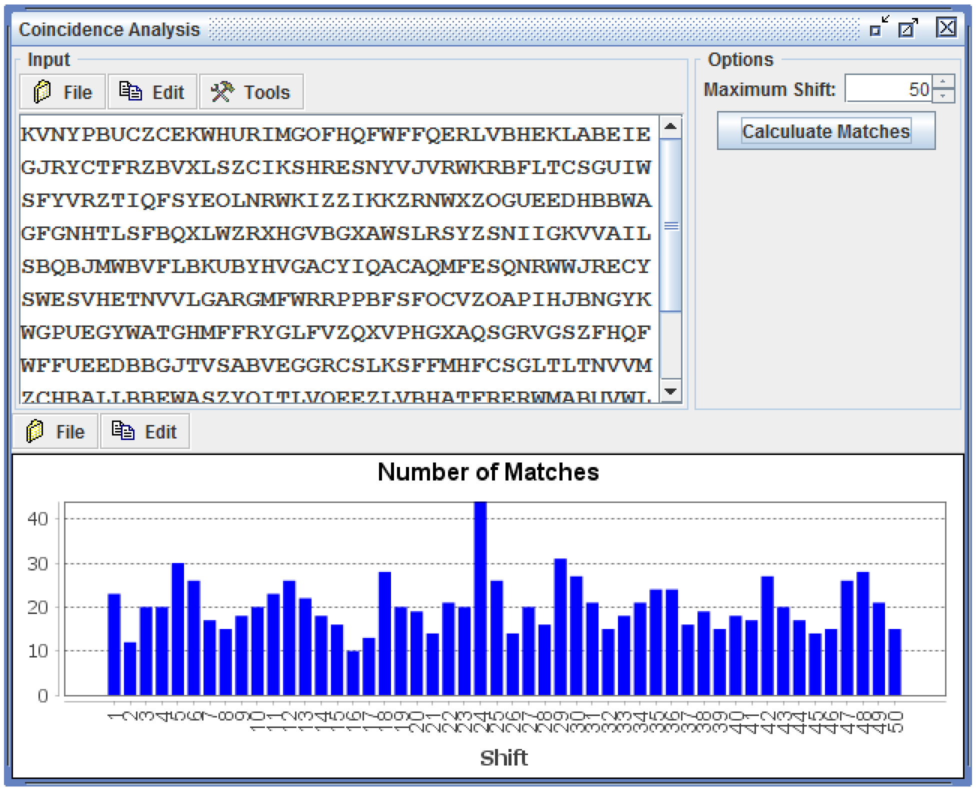
Minimum Substring Length: 3
Maximum Substring Length: 5
Maximum Divisor: 20

```
KVNYPBUCZCEKWHURIMGOFHQFWFFQERLVBHE
KLABEIEGJRYCTFRZBVXLSZCIKSHRESNYVJV
RWKRBFLTCSGUIWSFYVRZTIQFSYEOLNRWKIZ
ZIKKZRNWXZOGUEEDHBBWAGFGNHTLSFBQXLW
ZRXHGVBGXAWSLRSYZSNIIGKVVAILSBQBJMW
BVFLBKUBYHVGACYIQACAQMFESQNRWWJRECY
SWESVHETNVVLGARGMFWRRPPBFSFOCVZOAPI
HJBNGYKWGPUEGYWATGHMFFRYGLFVZQXVPHG
```

Figure 2.40: Kasiski's Method Dialog on Vigenère Example of Shakespeare's Sonnet #18

You will notice that this is the same bar chart that we saw in the previous example. I usually like to play a little with the options to see if anything changes. Moving the maximum substring length to 6 or 7 changes the chart but only in a minor way and above that there are no changes. Also, increasing the maximum divisor does not give us any information we did not already have. As in our previous example we would conclude that the most likely choice for the length of the keyword is 6 with the outside chance of a length of 5.

To apply coincidence analysis to the ciphertext, select Tools > Text & Stream Analysis > Coincidence Analysis from the main menu. Copy the ciphertext to the input box, leave the maximum shift at 50, and click the Calculate Matches button. At this point you should see the image below.



Maximum Shift: 50

Calculate Matches

```
KVNYPBUCZCEKWHURIMGOFHQFWFFQERLVBHEKLABEIE
GJRYCTFRZBVXLSZCIKSHRESNYVJVRWKRBFLTCSGUIW
SFYVRZTIQFSYEOLNRWKIZZIKKZRNWXZOGUEEDHBBWA
GFGNHTLSFBQXLWZRXHGVBGXAWSLRSYZSNIIGKVVAIL
SBQBJMWBVFLBKUBYHVGACYIQACAQMFESQNRWWJRECY
SWESVHETNVVLGARGMFWRRPPBFSFOCVZOAPIHJBNGYK
WGPUEGYWATGHMFFRYGLFVZQXVPHGXAQSGRVGSZFHQF
WFFUEEDBBGJTVSABVEGGRCSLKSFFMHFCSGLTLTNVVM
ZCHBALLBBEWASZYOITLVOEEZLVBHATERERWMABUVWL
```

Figure 2.41: Coincidence Analysis Dialog on Vigenère Example of Shakespeare's Sonnet #18

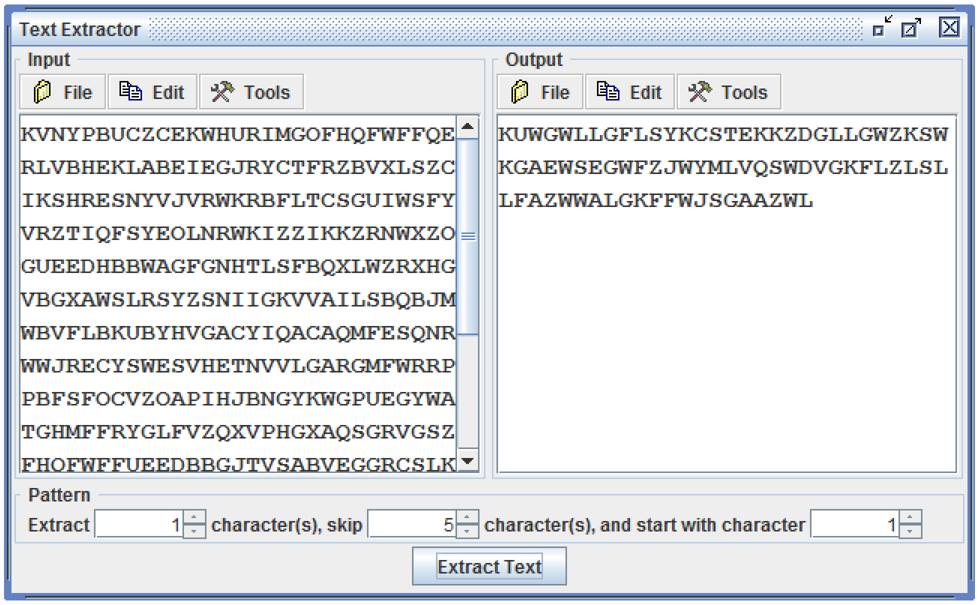
Again, I usually like to play with the options to see if there is any more information I can milk out of the data. If we increase our maximum shift up to 100 we get additional spikes at 54, 60, 78, 84, and 96, all multiples of 6, which further supports our guess of a length 6 keyword.

Determining the Keyword

At this point we will extract our subtexts. Open up the text extractor by selecting Tools > Text Tools > Text Extractor from the main menu. Copy the entire ciphertext into the input box. Since we want to extract every sixth character we will want to extract 1 character then skip 5 characters, for example, if we take the beginning of the ciphertext from this example, the underlined characters are the 1^{st}, 7^{th}, 13^{th}, ...and the number of characters between them is 5, the amount we want to skip.

KVNYPBUCZCEKWHURIMGOFHQFWFFQ. . . .

To get all 6 subtexts we will want to start at the first character for the first subtext, the second character for the second and so on. So for the first subtext, we will set the extract to 1, the skip to 5 and the start to 1, then click Extract Text. When you do you will see the first extraction in the output box, below. Copy and paste the output into a text editor or the Notepad tool under the Tools menu. From here all you need to do is change the start character to 2 and click the Extract Text button again, you now have the second subtext, copy it some where safe. Continue with start characters of 3, 4, 5, and finally 6. At this point you have all 6 subtexts below.



```
KVNYPBUCZCEKWHURIMGOFHQFWFFQE
RLVBHEKLABEIEGJRYCTFRZBVXLSZC
IKSHRESNYVJVRWKRBFLTCSGUIWSFY
VRZTIQFSYEOLNRWKIZZIKKZRNWXZO
GUEEDHBBWAGFGNHTLSFBQXLWZRXHG
VBGXAWSLRSYZSNIIGKVVAILSBQBJM
WBVFLBKUBYHVGACYIQACAQMFESQNR
WWJRECYSWESVHETNVVLGARGMFWRRP
PBFSFOCVZOAPIHJBNGYKWGPUEGYWA
TGHMFFRYGLFVZQXVPHGXAQSGRVGSZ
FHOFWFFUEEDBBGJTVSABVEGGRCSLK
```

```
KUWGWLLGFLSYKCSTEKKZDGLLGWZKSW
KGAEWSEGWFZJWYMLVQSWDVGKFLZLSL
LFAZWWALGKFFWJSGAAZWL
```

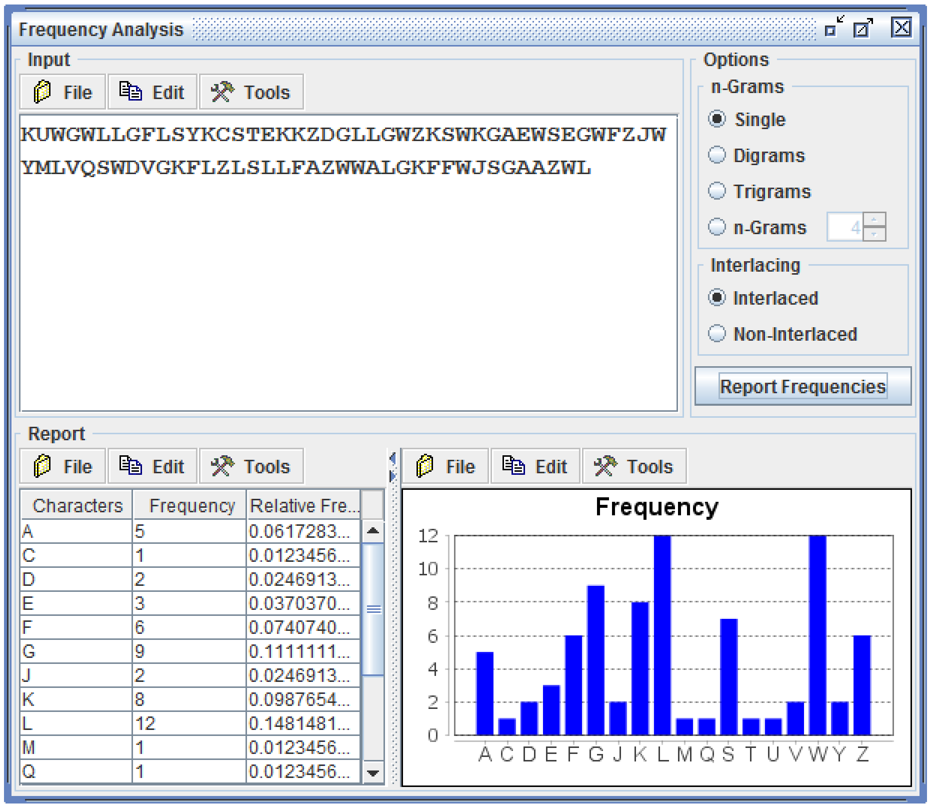
Extract 1 character(s), skip 5 character(s), and start with character 1

Figure 2.42: Text Extractor Dialog on Vigenère Example of Shakespeare's Sonnet #18

```
KUWGWLLGFLSYKCSTEKKZDGLLGWZKSWKGAEWSEGWFZJWYMLVQSW
DVGKFLZLSLLFAZWWALGKFFWJSGAAZWL

VCHOFVAJRSHVRSFIOIZOHFSWVSSVBBUACSJWTARSOBGWFFPSZF
BSGSCTCBZVVRBOBFBWIHUQOSBZJGWGC

NZUFFBBRZZRJBGYQLZRGBGFZBLNVQVBCAQRENRRFANPAFVHGFF
BARFSNHBYOBEUQVARZTFNNGLFBRNFYG

YCRHQHEYBCEVFUVFNZNUBNBRGRIABFYYQNESVGPOPGUTRZGRHU
GBCFGVBEQEHRVRANFREBFAURRAFATVU

PEIQEEICVISRLIRSRIWEWHQXXSIIJLHIMRCVVMPCIYEGYQXVQE
JVSMLVAWIEAWWAIPXXSPQFIWIKXHMJI

BKMFRKETXKNWTWZYWKXEATXHAYGLMBVQFWYHLFBVHKGHGXAGFE
TELHTMLATZTMLAMEHAPHXKHVLEAMOXX
```

Now we have the choice of doing a frequency analysis to determine the shift for each subtext or we could do the dot product analysis to determine the shift. We will show how to do both methods here.

We will start out with the frequency analysis technique. Open the Frequency Analysis tool by selecting Tools > Text & Stream Analysis > Frequency Analysis from the main menu. Copy the first subtext into the input box. With the Vigenère cipher we will always use Single and Interlaced for the options. Click on the Report Frequencies button and you will see the dialog below. Note that the bar chart is the same as the one from the previous example.



```
KUWGWLLGFLSYKCSTEKKZDGLLGWZKSWKGAEWSEGWFZJW
YMLVQSWDVGKFLZLSLLFAZWWALGKFFWJSGAAZWL
```

n-Grams: [✓ Single] Digrams Trigrams n-Grams 4

Interlacing: [✓ Interlaced] Non-Interlaced

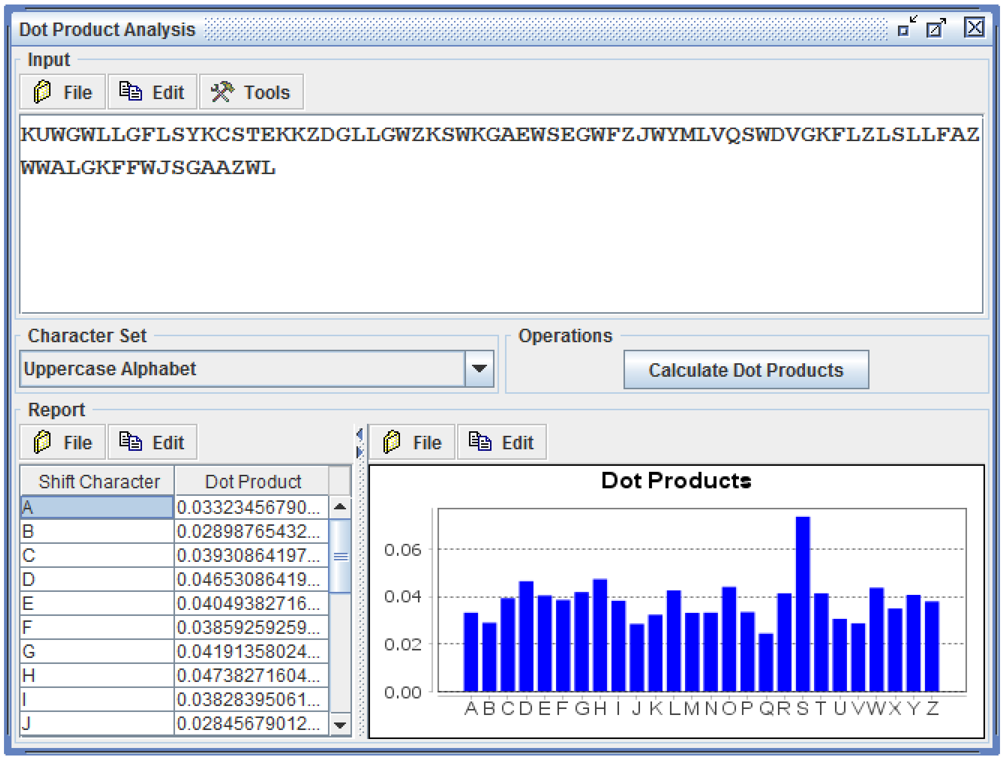
Characters	Frequency	Relative Fre...
A	5	0.0617283...
C	1	0.0123456...
D	2	0.0246913...
E	3	0.0370370...
F	6	0.0740740...
G	9	0.1111111...
J	2	0.0246913...
K	8	0.0987654...
L	12	0.1481481...
M	1	0.0123456...
Q	1	0.0123456...

Figure 2.43: Frequency Analysis Dialog on Vigenère Example of Sonnet #18: Subtext #1

Repeat this process with the second subtext, then the third and so on to get all of the charts we produced in the previous example. From here, we would analyze the charts as we did in the above example and come up with or letter possibilities for each of the six character positions.

S H	O	B N	N	E	D P T

To apply the dot product analysis to the subtexts, select Tools > Text & Stream Analysis > Dot Product Analysis from the main menu. Copy the first subtext into the input box, since our character set it the English alphabet we will leave that option as its default. Click on the Calculate Dot Products button and you will see the following dialog.



KUWGWLLGFLSYKCSTEKKZDGLLGWZKSWKGAEWSEGWFZJWYMLVQSWDVGKFLZLSLLFAZ
WWALGKFFWJSGAAZWL

Character Set: Uppercase Alphabet

Shift Character	Dot Product
A	0.03323456790...
B	0.02898765432...
C	0.03930864197...
D	0.04653086419...
E	0.04049382716...
F	0.03859259259...
G	0.04191358024...
H	0.04738271604...
I	0.03828395061...
J	0.02845679012...

Figure 2.44: Dot Product Analysis Dialog on Vigenère Example of Sonnet #18: Subtext #1

The chart in the dialog is the same as in the previous example, telling us that S is most likely the first character in our keyword. To get the other subtext charts, and hence keyword letters, copy the second subtext, third subtext, and so on into the input box and recalculate the dot products. As with the previous example, the charts will show that the keyword is fairly conclusively SONNET.

At this point we would go back to the Vigenère Cipher tool, copy the entire ciphertext into the input box, input the keyword SONNET with the repeated keyword type, and click on the Decrypt button. We should then see an English masterpiece in the output box. If we didn't, then we would need to go back to the analysis we did and look at other possible keywords and possibly key lengths. △

2.4 Scytale

2.4.1 Definitions and History

The scytale was an ancient tool for creating a transposition cipher. The way it worked is that the person who was encoding the message would wrap a strip of parchment or leather around a staff of a certain diameter and then write the message along the length of the staff, as pictured on the right. When the parchment or leather was unwound from the staff the letters would be seen lengthwise on the strip but in a nonsensical order. When the message was delivered the recipient would wind the message around a staff of the same diameter and then read the message.

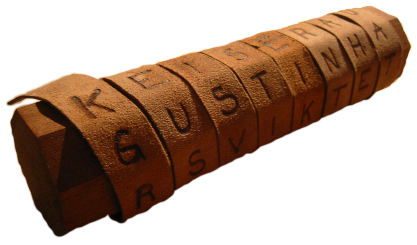


Figure 2.45: Skytale

Steganography also comes into the mix by the way that the scytale was transferred from sender to receiver. If the messenger had carried a long strip of parchment with letters on it, it was a bit of a giveaway that it was a scytale. Furthermore the scytale was not very difficult to break if you knew how it worked. One way would be to simply try different staffs of different diameters until the message could be read, but the staff was not even necessary. The interceptor of the message just needed to line up the characters with different skips until they discovered the message. So when the message was written on a strip of leather, the messenger would ware the leather strip as a belt with the lettering on the inside.

The scytale dates back to the 5th century BC. One account of its use was in 404 BC, five messengers were sent from Persia to Sparta with a message to Lysander that Pharnabazus of Persia was planning to attack Sparta. Only one of the five messengers made it to Lysander of Sparta, bloody and battered, the messenger handed Lysander his belt, Lysander wrapped the belt around his scytale to read the warning. Lysander prepared for the attack and was successful in driving back Pharnabazus's army.[55]

The first paragraph defines the way that the scytale worked, but we will formally define it here.

Definition 15: *The scytale is a transposition cipher. To encode a message the encoder would wrap a strip of parchment or leather around a staff of a certain diameter and then write the message along the length of the staff. The parchment or leather was then unwound from the staff. To decode the message, the recipient would wind the message parchment or leather around a staff of the same diameter and then read the message.*

Example 22: If we were to encrypt the message,

```
THISISATESTOFTHESCYTALECIPHER
```

using a staff diameter that permitted 4 letters to be written around the rod it would encrypt to,

```
TESIHSCPITYHSOTEIFARSTL␣AHE␣TEC
```

Δ

2.4.2 Cryptanalysis

The scytale is fairly easy to crack with a ciphertext only attack. There can only be a few options for the number of letters that can be written around the staff, the cryptanalyst would simply position the parchment or leather so that there would be 2 letters around, then 3 letters around, then 4 then 5 and so on until a recognizable message was seen. If we do not have the physical parchment we can simply write the letters in a table down each column with the number of rows being the number of letters around.

Example 23: Take our ciphertext message from the previous example,

```
TESIHSCPITYHSOTEIFARSTL␣AHE␣TEC
```

If we start with two letters around we get,

```
T S H C I Y S T I A S L A E T C
E I S P T H O E F R T   H   E
```

which does not produce anything understandable. Now try three letters around,

```
T I C T S E A T A   C
E H P Y O I R L H T
S S I H T F S   E E
```

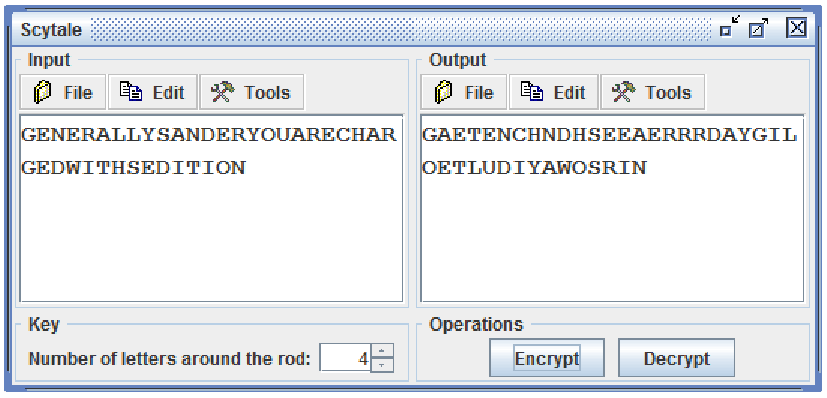
which also does not produce anything understandable. Now try four letters around,

```
T H I S I S A T
E S T O F T H E
S C Y T A L E C
I P H E R
```

which does give us an understandable message. Also note that the positions of the spaces showed us that using 2 or 3 letters around was not correct. Since we know that the A in AHE, just after the first space, must appear on the first line, and that there are 24 characters (including the space) before the A, the number of letters around the rod must be a divisor of 24. In our example here, that did not save us any work, but if 4 had not worked, the divisor observation would have told us that 5 would not work either and hence we would proceed to 6 rows. Δ

2.4.3 Cryptography Explorer

The Cryptography Explorer program has a tool for encryption and decryption using the Scytale cipher, select Ciphers > Scytale from the main menu and a dialog box similar to the one below will appear. The left portion is a standard input box, with file and print options, editing options and the text conversion tools. The right side is a standard output box, with file and print options, and some editing options. At the bottom, there is an option to specify the number of letters around the rod and the Encrypt and Decrypt buttons.



```
GENERALLYSANDERYOUARECHAR
GEDWITHSEDITION
```

```
GAETENCHNDHSEEAERRRDAYGIL
OETLUDIYAWOSRIN
```

Number of letters around the rod: 4

Figure 2.46: Scytale Cipher Tool

Since the only option is the number of letters around the rod, the user can also use this tool as a scytale cryptanalysis tool by placing the ciphertexxt in the input box, selecting 2, 3, 4, . . . letters around the rod and clicking Decrypt for each number selection until an understandable message is obtained.

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.

2.5 Rail Fence

2.5.1 Definitions and History

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.

Definition 16: *To encrypt a message with the Rail Fence cipher, first select the number of rails and make that the number of rows of a grid. Start in the upper left cell of the grid and write the message in a diagonal zig-zag pattern, one letter per cell. More specifically, begin in the upper left, for each letter move to the right one cell and down one cell. When you reach the last row proceed by moving up one cell and right one cell for each letter until you reach the first row and then repeat the downward direction. Continue until the message is completely in the grid. Rewrite the message by reading off the first row then the second row, and so on, ignoring blank cells.*

To decrypt a message with the Rail Fence cipher, first select the number of rails and make that the number of rows of a grid. Start in the upper left cell of the grid and mark the cells in the same diagonal zig-zag pattern as explained above in the encryption directions, one mark per cell, continue until you have as many marks as letters in the ciphertext. Start in the first row and place the letters of the ciphertext in each of the marked cells, one letter per cell. When the end of the first row is reached, continue with the second row and then the third row and so on until the ciphertext message is all in the grid. Now read the message by following the zig-zag pattern.

Example 24: 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. Δ

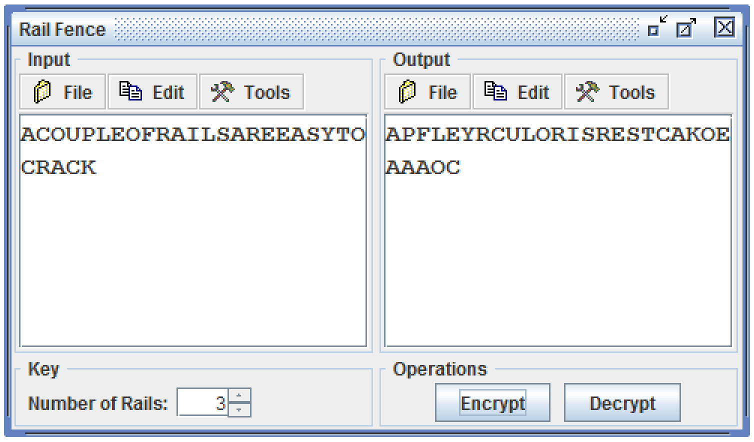
There is not much to say about this cipher historically, it appears to never to have been seriously used. The most common reference to it use is a "playground" cipher. Grade school children commonly use the cipher with two rails to encrypt messages to their friends.

2.5.2 Cryptanalysis

The cryptanalysis of the rail fence cipher is similar to that of the scytale cipher. There are only a few possibilities to the number of rails that would be used. In fact, if the number of rails is equal to the number of characters in the message, the ciphertext is the same as the plaintext. As with the scytale cipher, we would simply try to decrypt the ciphertext with 2 rails, then 3 rails, then 4, then 5, and so on until we got an understandable message.

2.5.3 Cryptography Explorer

The Cryptography Explorer program has a tool for the encryption and decryption of messages using the rail fence cipher, select Ciphers > Rail Fence from the main menu and a dialog box similar to the one below will appear. The left portion is a standard input box, with file and print options, editing options and the text conversion tools. The right side is a standard output box, with file and print options, and some editing options. At the bottom, there is an option to specify the number of rails and the Encrypt and Decrypt buttons.



ACOUPLEOFRAILSAREEASYTO
CRACK

APFLEYRCULORISRESTCAKOE
AAAOC

Number of Rails: 3

Figure 2.47: Rail Fence Cipher Tool

Since the only option is the number of rails, the user can also use this tool as a rail fence cryptanalysis tool by placing the ciphertexxt in the input box, selecting 2, 3, 4, . . . tails and clicking Decrypt for each number selection until an understandable message is obtained.

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.

2.6 Columnar

2.6.1 Definitions and History

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. Used alone it is a fairly weak cipher but when combined with other encryption techniques it is more formidable. For example, the ADFGX and ADFGVX ciphers used by the German military in World War I, is a combination of a substitution cipher and the columnar cipher.

During World War I and World War II, a variant of the columnar was used, called the Double Transposition Cipher. The Double Transposition Cipher was simply two applications of the columnar cipher using either the same key for both rounds or different keys for each round. While the extra shuffling of characters strengthened the cipher considerably, it was still very weak compared to the cipher machines like the German Enigma, the M-209 used by the US military, and Japan's Purple that were more commonly used in World War II.

There are several ways to set up a columnar cipher but the two that seem to be used the most are what we will call the classical columnar method and a variant developed by Émile Victor Théodore Myszkowski in 1902, which we will call the Myszkowski method.

Definition 17: *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.*

Example 25: 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. △

Definition 18: *The Myszkowski method allows duplicate characters in the keyword. 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.*

Example 26: 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. Δ

Decryption of either method is simply done in reverse.

2.6.2 Cryptanalysis

A brute force attack on a classical columnar cipher can be done, and would be relatively quick to do by computer as long as the keyword was not too long.

You would simply start with a keyword with two letters, say AB, and take all possible permutations of the letters as possible keywords. So for length 2 words the possibilities would be AB and BA. Simply decrypt the message using each possible keyword. If both of these fail, we move on to keywords of length three, there are 6 possible keywords of this length, ABC, ACB, BAC, BCA, CAB, and CBA. Again try each until a message appears. Failing this move on to 4 characters, and so on. As long as the keyword that was used is fairly small this should not take too long to solve, at least by computer. Since the number of letter permutations of a word of length n is $n!$, if the keyword is long then this will be difficult to do even with the machine.

Furthermore, an implementation of this could also use a hill climbing technique similar to the one discussed in the monoalphabetic substitution section. With each permutation, a fitness measure would be calculated, the user could designate the supposed keyword size or a set of possible sizes, the program would then return the result from the highest observed fitness measure.

We could also use this fitness measure to hopefully reduce the number of permutations that are needed to be examined. One such method would be to calculate a fitness measure for a keyword with two letters transposed, if the transposed word attains a higher fitness measure then we proceed with the new keyword and if not we would proceed with the old keyword. For example, say we were looking at a keyword of length 7. We would start with ABCDEFG, use it to decipher the the ciphertext and then calculate a fitness measure for the possible plaintext. Then we would transpose two of the characters, say to BACDEFG, use it to decipher the ciphertext and calculate a fitness measure for the new word. If the fitness measure increases we proceed to do more transposes with BACDEFG and if not we would proceed to do other transposes with ABCDEFG. There are 5040 permutations on 7 letters but there are only 21 transpositions, so even if we processed all the transpositions, calling that a single pass, then even if we had to do 5 passes before we settled in on a keyword, it is still nearly a fiftieth of the work required to check all permutations.

A general algorithm for this procedure would look a lot like the hill climbing method when applied to the monoalphabetic substitution cipher.

1. For each keyword length in the set of possible lengths, do the following.

2. Start with a keyword of the length to be tested, in general, a random keyword could be used as well as a stock word like ABC.... Use this keyword to decrypt the ciphertext and then calculate the fitness measure of the possible plaintext.

3. The program will then begin transposing the keyword entries, starting with A and B, then A and C, and so on, then B and C, and so on until the last two letters have been transposed. 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 keyword 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.

As we pointed out in the historical discussion, the columnar method was combined with a substitution method during World War I by the German military to create the ADFGX and ADFGVX ciphers. The way we broke these ciphers was to take advantage of the misuse of the cipher, and that was in the repetition in the messages. Many messages started with the same information, such as a title, location of transmission, the day's date, and so on. If two messages are encrypted with the same keyword and the messages begin with the same phrase then we can use matching substrings between the two messages to determine the beginnings of the columns in the grid. This will tell us the size of the keyword, as well as tell us the length of each of the columns, which in turn will possibly give us some information on the arrangement of the columns so that we do not need to try all permutations of the keyword letters. As we will see in the example below, this process can be accomplished by hand in a reasonable amount of time.

Example 27: We will do an example of using the substring comparison to determine the size of the keyword and the lengths of the columns. Our assumption here is that we have two ciphertexts whose plaintexts begin with the same phrase and that both were encrypted with the same keyword. We will take our example from the *Hitchhiker's Guide to the Galaxy* by Douglas Adams.[1]

The first plaintext will be,

> Far out in the uncharted backwaters of the unfashionable end of the western spiral arm of the Galaxy lies a small unregarded yellow sun.

> Orbiting this at a distance of roughly ninety-two million miles is an utterly insignificant little blue green planet whose ape-descended life forms are so amazingly primitive that they still think digital watches are a pretty neat idea.

Once converted and encrypted we get the following ciphertext.

```
FINEWONOEHEROAIAEDWBTANONTLLNLGATEPWPEIMONIEHLKAHA
YDUERCREHLFSPREYSNDLONATFLTIMSESIILEEESDOEZPITTIIT
RTTOHAAEHSBOESAHXAURLNISSOHEMNITNFLBENSEEFRAYTASHG
AAEAANCDAFFNNERAFLELGYSIHDCUIWIEUYNNLGLHENFSAGMTEL
DLEPNERTHBTTAADWNLTASLAEUTIIEGNOOSTIITERAODDEAMLIH
YTIWSREATUTKSUIETTIMGLMREORGTARYYLIARICTUNTACLRSIR
VTINTCETI
```

The second plaintext will be,

> Far out in the uncharted backwaters of the unfashionable end of the western spiral arm of the Galaxy lies a small unregarded yellow sun.
>
> Many were increasingly of the opinion that they'd all made a big mistake in coming down from the trees in the first place. And some said that even the trees had been a bad move, and that no one should ever have left the oceans.

Once converted and encrypted we get the following ciphertext.

```
FINEWONOEHEROAIAEDWNISOPTEMIAOOMEHTAEHNEBAATSEVTAU
ERCREHLFSPREYSNDLMRELEOTLASNGRTNRCODVTAAVHNLHFCOHA
AEHSBOESAHXAURLNERGHITAEIINFEIIASIEEHNOTOUREOANCDA
FFNNERAFLELGYSYNIFIHYAGKMWTEEPNSATEEDNNHVEHNRTHBTT
AADWNLTASLAEUWCNTNADDMEINHSFLDATHSEMDOOELESTUTKSUI
ETTIMGLMREOAEAYONHLBTCDORTSEMTERDBEAEDATE
```

The cryptography explorer program has a tool for finding substring matches between two input ciphertexts, we will discuss this option later in this section and in the appendix at the end of the notes. Although we are resorting to a computer at this point, the same analysis could be done by hand.

Using this feature and searching for substrings of 18 characters in length we get the following output. Note that the first character is at position 0, this was done to make the determination of the column lengths a little easier to calculate.

```
Input 1 Length:  309
Input 2 Length:  291

Matches

FINEWONOEHEROAIAED     Input 1: 0         Input 2: 0
```

```
INEWONOEHEROAIAEDW      Input 1: 1          Input 2: 1
UERCREHLFSPREYSNDL      Input 1: 52         Input 2: 49
OHAAEHSBOESAHXAURL      Input 1: 103        Input 2: 97
HAAEHSBOESAHXAURLN      Input 1: 104        Input 2: 98
ANCDAFFNNERAFLELGY      Input 1: 154        Input 2: 145
NCDAFFNNERAFLELGYS      Input 1: 155        Input 2: 146
RTHBTTAADWNLTASLAE      Input 1: 206        Input 2: 194
THBTTAADWNLTASLAEU      Input 1: 207        Input 2: 195
TUTKSUIETTIMGLMREO      Input 1: 258        Input 2: 243
```

The output here says that the substring FINEWONOEHEROAIAED was at position 0 in both inputs, the substring UERCREHLFSPREYSNDL started at position 52 in the first text and at position 49 in the second. Since, the two ciphertexts are of different lengths it makes sence that the column lengths will be different. It also tells us that in the first ciphertext the first column is of length $52 - 0 = 52$. We also have the substring OHAAEHSBOESAHXAURL starting at position 103 in the first text. So the third column starts at position 103 and the second column has length $103 - 52 = 51$. Notice that there are 5 jumps in positions in the first text (as well as the second) and each jump is 51 or 52 characters. This is a very good indication that there are 6 letters in the keyword. We will concentrate on the first ciphertext.

Continuing our analysis, we see that the column starts are at positions 0, 52, 103, 154, 206, and 258, signifying 6 columns and a keyword of length 6. Furthermore the column lengths are 52, 51, 51, 52, 52, and 51. Taking the ciphertext and breaking the lines at these positions we have the following.

```
FINEWONOEHEROAIAEDWBTANONTLLNLGATEPWPEIMONIEHLKAHAYD
UERCREHLFSPREYSNDLONATFLTIMSESIILEEESDOEZPITTIITRTT
OHAAEHSBOESAHXAURLNISSOHEMNITNFLBENSEEFRAYTASHGAAEA
ANCDAFFNNERAFLELGYSIHDCUIWIEUYNNLGLHENFSAGMTELDLEPNE
RTHBTTAADWNLTASLAEUTIIEGNOOSTIITERAODDEAMLIHYTIWSREA
TUTKSUIETTIMGLMREORGTARYYLIARICTUNTACLRSIRVTINTCETI
```

Rewriting the rows as columns we get,

F	U	O	A	R	T
I	E	H	N	T	U
N	R	A	C	H	T
E	C	A	D	B	K
W	R	E	A	T	S
O	E	H	F	T	U
N	H	S	F	A	I
O	L	B	N	A	E
E	F	O	N	D	T
H	S	E	E	W	T
E	P	S	R	N	I

R	R	A	A	L	M
O	E	H	F	T	G
A	Y	X	L	A	L
I	S	A	E	S	M
A	N	U	L	L	R
E	D	R	G	A	E
D	L	L	Y	E	O
W	O	N	S	U	R
B	N	I	I	T	G
T	A	S	H	I	T
A	T	S	D	I	A
N	F	O	C	E	R
O	L	H	U	G	Y
N	T	E	I	N	Y
T	I	M	W	O	L
L	M	N	I	O	I
L	S	I	E	S	A
N	E	T	U	T	R
L	S	N	Y	I	I
G	I	F	N	I	C
A	I	L	N	T	T
T	L	B	L	E	U
E	E	E	G	R	N
P	E	N	L	A	T
W	E	S	H	O	A
P	S	E	E	D	C
E	D	E	N	D	L
I	O	F	F	E	R
M	E	R	S	A	S
O	Z	A	A	M	I
N	P	Y	G	L	R
I	I	T	M	I	V
E	T	A	T	H	T
H	T	S	E	Y	I
L	I	H	L	T	N
K	I	G	D	I	T
A	T	A	L	W	C
H	R	A	E	S	E
A	T	E	P	R	T
Y	T	A	N	E	I
D			E	A	

Since the columns with 51 characters must be on the right, this is a consequence of the columnar encryption method, we can rearrange the columns so that the three of length 52 are the first three and the three columns that are of length 51 are the last three columns.

F	A	R	T	U	O
I	N	T	U	E	H
N	C	H	T	R	A
E	D	B	K	C	A
W	A	T	S	R	E
O	F	T	U	E	H
N	F	A	I	H	S
O	N	A	E	L	B
E	N	D	T	F	O
H	E	W	T	S	E
E	R	N	I	P	S
R	A	L	M	R	A
O	F	T	G	E	H
A	L	A	L	Y	X
I	E	S	M	S	A
A	L	L	R	N	U
E	G	A	E	D	R
D	Y	E	O	L	L
W	S	U	R	O	N
B	I	T	G	N	I
T	H	I	T	A	S
A	D	I	A	T	S
N	C	E	R	F	O
O	U	G	Y	L	H
N	I	N	Y	T	E
T	W	O	L	I	M
L	I	O	I	M	N
L	E	S	A	S	I
N	U	T	R	E	T
L	Y	I	I	S	N
G	N	I	C	I	F
A	N	T	T	I	L
T	L	E	U	L	B
E	G	R	N	E	E
P	L	A	T	E	N
W	H	O	A	E	S
P	E	D	C	S	E
E	N	D	L	D	E

I	F	E	R	O	F
M	S	A	S	E	R
O	A	M	I	Z	A
N	G	L	R	P	Y
I	M	I	V	I	T
E	T	H	T	T	A
H	E	Y	I	T	S
L	L	T	N	I	H
K	D	I	T	I	G
A	L	W	C	T	A
H	E	S	E	R	A
A	P	R	T	T	E
Y	N	E	I	T	A
D	E	A			

At this point it is fairly simple, looking at the first row or two we simply want to rearrange them so that we see some English words. The first three already spell FAR and the last three can be arranged to OUT. Even if it was not as easy as this, there are $3! = 6$ ways to permute three letters, and hence columns. So there would be 36 possible ways to permute the first three columns and the second three columns, one of these combinations will produce our plaintext. Since the text is only transposed and there is no substitution on top of it, we would rarely need to resort to trying all possible permutations.

F	A	R	O	U	T
I	N	T	H	E	U
N	C	H	A	R	T
E	D	B	A	C	K
W	A	T	E	R	S
O	F	T	H	E	U
N	F	A	S	H	I
O	N	A	B	L	E
E	N	D	O	F	T
H	E	W	E	S	T
E	R	N	S	P	I
R	A	L	A	R	M
O	F	T	H	E	G
A	L	A	X	Y	L
I	E	S	A	S	M
A	L	L	U	N	R
E	G	A	R	D	E
D	Y	E	L	L	O
W	S	U	N	O	R

B	I	T	I	N	G
T	H	I	S	A	T
A	D	I	S	T	A
N	C	E	O	F	R
O	U	G	H	L	Y
N	I	N	E	T	Y
T	W	O	M	I	L
L	I	O	N	M	I
L	E	S	I	S	A
N	U	T	T	E	R
L	Y	I	N	S	I
G	N	I	F	I	C
A	N	T	L	I	T
T	L	E	B	L	U
E	G	R	E	E	N
P	L	A	N	E	T
W	H	O	S	E	A
P	E	D	E	S	C
E	N	D	E	D	L
I	F	E	F	O	R
M	S	A	R	E	S
O	A	M	A	Z	I
N	G	L	Y	P	R
I	M	I	T	I	V
E	T	H	A	T	T
H	E	Y	S	T	I
L	L	T	H	I	N
K	D	I	G	I	T
A	L	W	A	T	C
H	E	S	A	R	E
A	P	R	E	T	T
Y	N	E	A	T	I
D	E	A			

Note that we have deciphered the ciphertext without actually finding the keyword. This is always an option for the cryptanalyst, finding the meaning without the need to find the key.

Now that the first ciphertext has been deciphered, we would like to decipher the second, and any others that may have been encoded with the same key. Now we cannot figure out the actual keyword that was used but then again we do not need to. Since the keyword is of length 6, we can use the numbers 123456 as the characters for the keyword. Look back to the chart we got right after we broke the text into columns. The top line was FUOART and we wanted to rearrange it to FAROUT. If we take FUOART and label the characters 123456 then write the corresponding number below each character in FAROUT, we get 145326.

```
FUOART    FAROUT
123456    145326
```

If we had a keyword with more than 10 letters we could, in the same manner, use the alphabet. For example, if we used ABCDEF in place of 123456, and applied the same method we get the keyword, ADECBF.

```
FUOART    FAROUT
ABCDEF    ADECBF
```

Obviously, neither 145326 nor ADECBF was the keyword used but it does not matter. These work just as well as the actual keyword.

One thing to keep in mind is that in this example we used a lot of the same plaintext at the beginning. Historically, while this method was used, the duplicated texts were not nearly as lengthy. They were more like a couple words, or title, or location of a transmission. In the case where the duplicated texts are not as long we would be searching for smaller substring duplications, 18 is a good size substring. When we look at smaller substrings we may encounter false findings, duplications that do not the start of columns. Remember that all of the columns are of the same length, give or take a single character. So finding jumps at even intervals that are (close to) divisors of the length of the text is a good indication of a column break. Δ

The above example assumes that the encryption was done using a classical columnar cipher, if a Myszkowski method had been used then the cryptanalysis of the cipher becomes more difficult since the repeated character columns are dealt with differently than the unique letter columns. If one were trying to automate this, the algorithm would need to be altered to account for duplicate letters and the use of the fitness measure would greatly facilitate the finding of possible keywords.

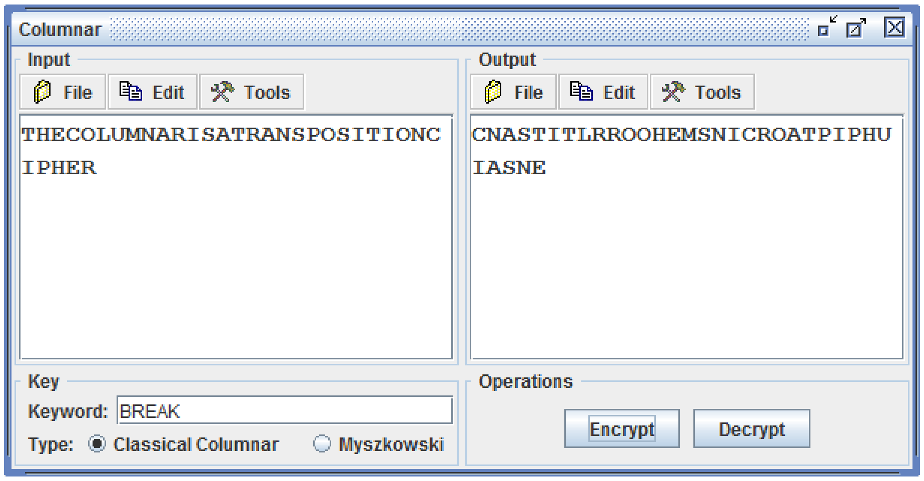
2.6.3 Cryptography Explorer

The Cryptography Explorer program has a tool for the encryption and decryption using a columnar cipher, and it has a tool to do substring comparisons to aid in the breaking of a columnar cipher.

Columnar Cipher Tool

Select Ciphers > Columnar from the main menu and the following will appear. At the top of the dialog are the input and output boxes, as with most of the cipher tools in the Cryptography Explorer program. There are standard file, printing, and editing options found with the input and output boxes as well as text conversion options for the input boxes. At the bottom are the buttons to invoke the encryption and decryption processes as well as an input for the keyword and a selection to use either the classical columnar or to use Myszkowski's method.

The program does not have an option for doing a double transposition cipher, although, you can open two columnar cipher windows and simply copy the output of the first into the input of the second.



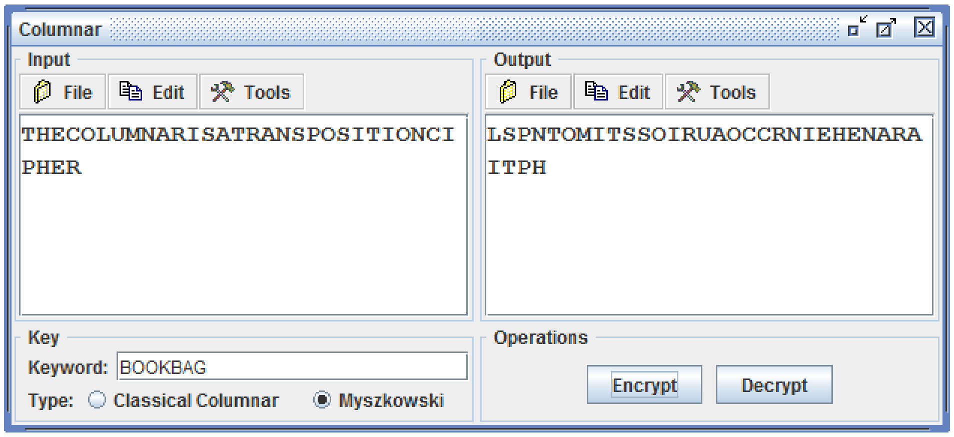
THECOLUMNARISATRANSPOSITIONCIPHER

CNASTITLRROOHEMSNICROATPIPHUIASNE

Keyword: BREAK

Type: [● Classical Columnar] ○ Myszkowski

Figure 2.48: Columnar Cipher Tool: Classical Mode



THECOLUMNARISATRANSPOSITIONCIPHER

LSPNTOMITSSOIRUAOCCRNIEHENARAITPH

Keyword: BOOKBAG

Type: ○ Classical Columnar [● Myszkowski]

Figure 2.49: Columnar Cipher Tool: Myszkowski Mode

Use of the tool,

To Encrypt —

1. Input the plaintext message into the Input box.
2. Input a Keyword and select the type of columnar algorithm, either the classical columnar or the Myszkowski method.

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, either the classical columnar or the Myszkowski method.
3. Click the Decrypt button. At this point the Output box will display the plaintext.

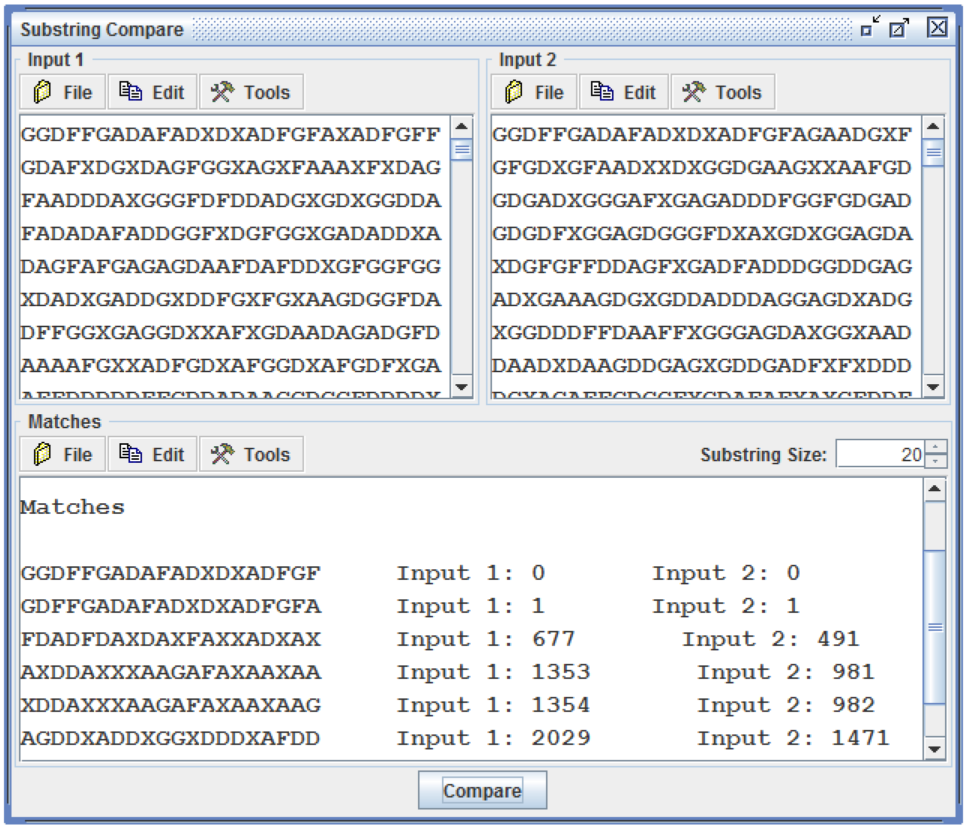
Note that the keyword need not be only uppercase letters. The columns are ordered by the character's ASCII number and hence any keyboard character can be used in the keyword. Conventionally, the keyword is all uppercase alphabetic letters, but it is not required. A note about using ASCII numbers for ordering is that A and a have different ASCII numbers so those columns would not be considered to have the same letter. Historically, some applications of the columnar cipher used numbers in place of letters for the keyword, for example 42835 may be the keyword. Since the program uses ASCII values for the characters, using numbers in place of letters works equally well. The ASCII order of numbers is 0123456789, and not 1234567890, as is the layout on the standard keyboard.

Substring Compare Tool

The Substring Compare window finds matches between two strings and reports the positions of those matches. This is primarily a tool for cracking columnar ciphers and the 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 2.50: Substring Comparison Tool

2.7 Playfair

2.7.1 Definitions and History

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.

Charles Wheatstone was an English scientist and inventor who lived between 1802 and 1875. He is probably most well known for the invention of the Wheatstone Bridge, an electrical circuit used to measure an unknown electrical resistance by balancing two legs of a bridge circuit, shown on the right. Wheatstone also made major contributions to the fields of acoustics, electricity, optics, cryptography, chronography, and was a major contributor to the development of the telegraph.[6][39]

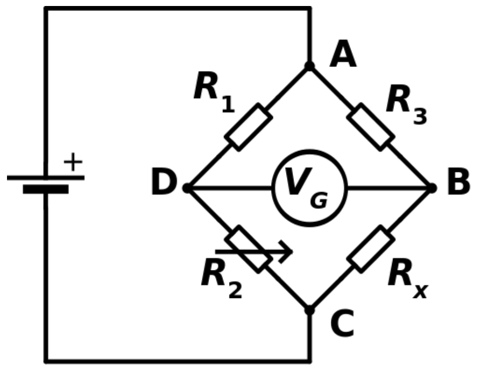


Figure 2.51: Wheatstone Bridge

Wheatstone's biggest contribution to optics was in the invention of the stereoscope.[19] The Stereoscope was a method for viewing three-dimensional images from a two-dimensional medium. The concept is still in use today in the production of 3-D images and movies. Take two photographs of the same image that are a couple inches apart, that is, the image photographed and then photographed again with the camera moved horizontally by a few inches so that the line if sight of the second image is in parallel with the line of sight of the first. If these two images are mounted side by side and viewed through prismatic lenses or colored filters the viewer will see a three dimensional rendition of the image.

Oliver Wendell Holmes built a cheaper hand-held version of Wheatstone's stereoscope in 1861 that became a household item by the turn of the century. It was constructed of wood or in some cases tin with two prismatic lenses and a stand to hold a stereo card. The same technology was used in the 1960's with the invention of the View-Master, which you may have had as a kid, as well as the old red/blue glasses and the newer polarized versions used to view 3-D movies. In fact, this technology is also incorporated into virtual-reality headsets.

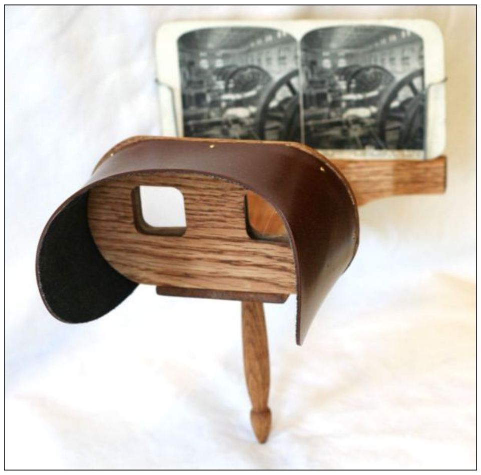


Figure 2.52: Holmes Stereoscope

Lyon Playfair or Baron Playfair, (1818–1898) was a scientist as well as Wheatstone but also had a career in politics. Playfair held many high-ranking posts including, Member of Parliament, Chairman of Ways and Means, Postmaster General, and Vice-President of the

Committee on Education. During his political career he advocated the use of sending secure messages and to that end promoted Wheatstone's cipher. At first, the British Foreign Office rejected the cipher, stating that it was too complex. Wheatstone then offered to demonstrate that three out of four grade school children from a local school could master the cipher in less than 15 minutes. The Under Secretary of the Foreign Office responded with, "That is very possible, but you could never teach it to attachés."

There are several slightly different ways to implement this cipher. The one we will be using equates I and J in order to get 26 letters down to 25, so that the alphabet will fit in a 5×5 grid. So the plaintext cannot have any J's in it.

Figure 2.53: Stereoscopic Movie Camera

As we stated above, this is a digram cipher, so the plaintext will be broken into blocks of two characters. One stipulation on these blocks is that there can be no repeated characters, so if we encounter a repeated character we must insert a dummy character, such as X. For example, say we had the word COMMITTEE in our plaintext and that as we were breaking the plaintext into blocks of two a block started at the beginning of the word. Then we would have the following blocks, CO MM IT TE E$\underline{\text{X}}$ (where the $\underline{\text{X}}$ just represents the first character of the next word). Note that the TT and EE were split due to their position but the MM was not. So we must insert a character between the MM, giving COMXMITTEE and subsequently CO MX MI TT EE when we block the letters. Now the MM is no longer a problem but due to the insertion the TT must now be split, this gives us COMXMITXTEE and subsequently CO MX MI TX TE E$\underline{\text{X}}$. Again the extra character has automatically split the EE and as long as the next word does not begin with E, we are finished blocking the word COMMITTEE. If the C in COMMITTEE had been blocked with the last character of the previous word then we would have, $\underline{\text{X}}$C OM MI TT EE, which would have been converted to, $\underline{\text{X}}$C OM MI TX TE E$\underline{\text{X}}$. Sometimes this double character problem is taken care of in a slightly different manner. Instead of worrying about which pair needs to be altered and which do not and to avoid this continual backtracking, sometimes the encrypter will simply split all repeated characters, so COMMITTEE would be altered to COMXMITXTEXE, and in the very unlikely event of XX in the plaintext we would change it to XZX or something equivalent.

Also since this is a digram cipher, we must have an even number of characters in the plaintext. Hence we may need to pad the plaintext with a character at the end, again X is commonly used but any letter that does not produce a double letter at the end would work as well.

Once the plaintext is prepared, we need to construct a key. Usually the key is given as a single word or short phrase. The word is altered as follows, first all J's are converted to I's, then any duplication of characters is removed. At this point the word is written in a 5×5 grid, starting in the upper left cell and moving across the first row, then the second and so on until the characters of the keyword are exhausted. The remaining cells in the grid are filled with the remaining letters of the alphabet (again, I and J are the same) 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. From there, we continue with B, C, D, and so on. When finished we get the following grid,

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

These grids are called Polybius Squares. Polybius square ciphers were common at the beginning of the 20th century and used by both sides in World War I.

When encrypting the plaintext there are three different cases, depending on the digram that is being encrypted.

1. If the letters of the plaintext diagram are not in the same row or column. In this case we use the plaintext diagram character positions as the corners of a rectangle and encrypt the digram with the other two corners, using the entry in the same row as the first plaintext character for the first ciphertext character and using the entry in the same row as the second plaintext character for the second ciphertext character. For example, if we encrypt TH using the above grid, we find the T and H in the grid, shown in red, and we take the other two corners, shown in blue. The Q is on the same row as T so Q is the first ciphertext character and M is on the same row as H so it becomes the second ciphertext character. Hence TH is encrypted as QM.

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

   Similarly, HT would encrypt as MQ, LS would encrypt as YO, GD as MR and so on.

2. If the letters of the plaintext diagram are in the same row. In this case we take the letters directly to the right of the plaintext characters, wrapping around to the far left character of the same row if the plaintext character is at the far right. For example, to encrypt NS, which is on row 4, we shift the N to O and the S to T, so NS encrypts to OT.

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

Similarly, SN would encrypt as TO, OT would encrypt as QN, HM as KE and so on.

3. If the letters of the plaintext diagram are in the same column. In this case we take the letters directly below the plaintext characters, wrapping around to the top character of the same column if the plaintext character is at the bottom of the column. For example, to encrypt LO, which is on column 2, we shift the L to R and the O to V, so LO encrypts to RV.

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

Similarly, OL would encrypt as VR, RV would encrypt as GL, AW as BA and so on.

Considering our above discussion there is probably no need to formally define the Playfair cipher but we will do so for the sake of completeness.

Definition 19: *The Playfair cipher is a digram substitution cipher using a* 5×5 *polybius square with the following rules.*

Polybius Square Setup

The polybius square is a 5×5 *grid of characters such that each cell contains a single character from the alphabet, excluding J, since all J's are changed to I in the plaintext. Although any configuration of this type is acceptable, the grid is usually set up using a keyword in the following manner. Take the keyword and convert all of the J's to I's, then remove any duplication of characters. Write the word is in the* 5×5 *grid by starting in the upper left cell and moving across the first row, then the second and so on until the characters of the altered keyword are exhausted. The remaining cells in the grid are filled with the remaining letters of the alphabet (excluding J) in alphabetical order.*

Plaintext Setup

To prepare the plaintext for encryption we do the following.

1. *Convert all characters to uppercase, remove all spaces, punctuation, and numbers.*

2. *Change all J's to I's.*

3. *Break the plaintext into blocks of two characters, if a double character pair is encountered insert an X between the double characters, if the double character is XX insert a Z between the two. If needed, add an X or a Z to the end of the plaintext to make the number of characters in the plaintext even, use whichever character does not create a double character at the end.*

Encryption Procedure

Encrypt each plaintext digram using the 5×5 *polybius square as follows. The process breaks into three different cases depending on the positions of the plaintext characters in the polybius square.*

1. *If the letters of the plaintext diagram are not in the same row or column. In this case we use the plaintext diagram character positions as the corners of a rectangle and encrypt the digram with the other two corners, using the entry in the same row as the first plaintext character for the first ciphertext character and using the entry in the same row as the second plaintext character for the second ciphertext character.*
2. *If the letters of the plaintext diagram are in the same row. In this case we take the letters directly to the right of the plaintext characters, wrapping around to the far left character of the same row if the plaintext character is at the far right.*
3. *If the letters of the plaintext diagram are in the same column. In this case we take the letters directly below the plaintext characters, wrapping around to the top character of the same column if the plaintext character is at the bottom of the column.*

Finally, we would concatenate all of the ciphertext digrams together.

Decryption Procedure

The decryption process is simply a reversal of the encryption process. Specifically, to decrypt we take each ciphertext digram and use the 5×5 *polybius square as follows. The process breaks into three different cases depending on the positions of the plaintext characters in the polybius square.*

1. *If the letters of the ciphertext diagram are not in the same row or column. In this case we use the ciphertext diagram character positions as the corners of a rectangle and encrypt the digram with the other two corners, using the entry in the same row as the first ciphertext character for the first plaintext character and using the entry in the same row as the second ciphertext character for the second plaintext character.*
2. *If the letters of the ciphertext diagram are in the same row. In this case we take the letters directly to the left of the ciphertext characters, wrapping around to the far right character of the same row if the ciphertext character is at the far left.*
3. *If the letters of the ciphertext diagram are in the same column. In this case we take the letters directly above the ciphertext characters, wrapping around to the bottom character of the same column if the ciphertext character is at the top of the column.*

Finally, we would concatenate all of the plaintext digrams together.

Example 28: We will take one of my favorite passages from the Hitchhiker's Guide to the Galaxy, which was unfortunately cut from the 2005 Touchstone Pictures movie staring Martin Freeman, Zooey Deschanel, Sam Rockwell and Bill Nighy. Evidently the writers did not like it as much as I do.

```
"But the plans were on display..."
"On display? I eventually had to go down to the cellar to find them."
"That's the display department."
"With a torch."
"Ah, well the lights had probably gone."
"So had the stairs."
"But look, you found the notice didn't you?"
"Yes," said Arthur, "yes I did. It was on display in the bottom of a
locked filing cabinet stuck in a disused lavatory with a sign on the
door saying Beware of the Leopard."
```

First, we convert the plaintext to uppercase and remove spaces and punctuation.

```
BUTTHEPLANSWEREONDISPLAYONDISPLAYIEVENTUALLYHADTOG
ODOWNTOTHECELLARTOFINDTHEMTHATSTHEDISPLAYDEPARTMEN
TWITHATORCHAHWELLTHELIGHTSHADPROBABLYGONESOHADTHES
TAIRSBUTLOOKYOUFOUNDTHENOTICEDIDNTYOUYESSAIDARTHUR
YESIDIDITWASONDISPLAYINTHEBOTTOMOFALOCKEDFILINGCAB
INETSTUCKINADISUSEDLAVATORYWITHASIGNONTHEDOORSAYIN
GBEWAREOFTHELEOPARD
```

There are no J's in the plaintext so we do not need to change these to I's. We then remove all double characters, we have simply placed an X between any double characters even though for some it would not have been necessary.

```
BUTXTHEPLANSWEREONDISPLAYONDISPLAYIEVENTUALXLYHADT
OGODOWNTOTHECELXLARTOFINDTHEMTHATSTHEDISPLAYDEPART
MENTWITHATORCHAHWELXLTHELIGHTSHADPROBABLYGONESOHAD
THESTAIRSBUTLOXOKYOUFOUNDTHENOTICEDIDNTYOUYESXSAID
ARTHURYESIDIDITWASONDISPLAYINTHEBOTXTOMOFALOCKEDFI
LINGCABINETSTUCKINADISUSEDLAVATORYWITHASIGNONTHEDO
XORSAYINGBEWAREOFTHELEOPARD
```

At this point there are an odd number of characters in the plaintext so we pad the plaintext with an X at the end.

```
BUTXTHEPLANSWEREONDISPLAYONDISPLAYIEVENTUALXLYHADT
OGODOWNTOTHECELXLARTOFINDTHEMTHATSTHEDISPLAYDEPART
MENTWITHATORCHAHWELXLTHELIGHTSHADPROBABLYGONESOHAD
THESTAIRSBUTLOXOKYOUFOUNDTHENOTICEDIDNTYOUYESXSAID
ARTHURYESIDIDITWASONDISPLAYINTHEBOTXTOMOFALOCKEDFI
LINGCABINETSTUCKINADISUSEDLAVATORYWITHASIGNONTHEDO
XORSAYINGBEWAREOFTHELEOPARDX
```

Break the above text into digrams.

```
BU TX TH EP LA NS WE RE ON DI SP LA YO ND IS PL AY IE VE NT UA LX LY
HA DT OG OD OW NT OT HE CE LX LA RT OF IN DT HE MT HA TS TH ED IS PL
AY DE PA RT ME NT WI TH AT OR CH AH WE LX LT HE LI GH TS HA DP RO BA
BL YG ON ES OH AD TH ES TA IR SB UT LO XO KY OU FO UN DT HE NO TI CE
DI DN TY OU YE SX SA ID AR TH UR YE SI DI DI TW AS ON DI SP LA YI NT
HE BO TX TO MO FA LO CK ED FI LI NG CA BI NE TS TU CK IN AD IS US ED
LA VA TO RY WI TH AS IG NO NT HE DO XO RS AY IN GB EW AR EO FT HE LE
OP AR DX
```

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

Using the above polybius square, we encrypt the pliantext digrams to get the following.

```
IW SZ QM NI AY OT UH IG QO IR NY AY LS TI CN LA YF EN UG ON WP YV AF
QB MZ VO TR QV ON QN KG IK YV AY DO TL EU MZ KG TZ QB NT QM MI CN LA
YF IM LY DO EG ON UB QM FQ VG BK BQ UH YV FO KG PR HK NT QB IF GV HB
RA LK QO KN QG FB QM KN QF RB QC ZN RV VS SC NV LT PU MZ KG OQ ND IK
IR IT SF NV PK XY QY RI LB QM VI PK NC IR IR QZ YQ QO IR NY AY PC ON
KG RQ SZ NQ GT PY RV KS MI PD PR OE BY CR UN NT NZ KS EU FB CN XN MI
AY WL NQ CL UB QM YQ RE OQ ON KG RT VS CO YF EU HR HU LB GN DZ KG PG
NL LB CZ
```

Concatenating the digrams together gives us the following ciphertext.

```
IWSZQMNIAYOTUHIGQOIRNYAYLSTICNLAYFENUGONWPYVAFQBMZ
VOTRQVONQNKGIKYVAYDOTLEUMZKGTZQBNTQMMICNLAYFIMLYDO
EGONUBQMFQVGBKBQUHYVFOKGPRHKNTQBIFGVHBRALKQOKNQGFB
QMKNQFRBQCZNRVVSSCNVLTPUMZKGOQNDIKIRITSFNVPKXYQYRI
LBQMVIPKNCIRIRQZYQQOIRNYAYPCONKGRQSZNQGTPYRVKSMIPD
PROEBYCRUNNTNZKSEUFBCNXNMIAYWLNQCLUBQMYQREOQONKGRT
VSCOYFEUHRHULBGNDZKGPGNLLBCZ
```

Δ

2.7.2 Cryptanalysis

Certainly if we had a large enough crib we could use the correspondence to build the polybius square or at least most of it. We will concentrate, instead, on ciphertext only attack strategies. The idea is to create the polybius square from just the ciphertext digrams. Since we are working with digrams, we will need a lot more ciphertext to break a Playfair cipher then we did for a monoalphabetic substitution cipher. With our setup of the Playfair cipher there are 600 possible digrams whereas with a monoalphabetic substitution cipher there are only 26 single characters.

If we are attacking this by hand, here are a few things to keep in mind when constructing the square.

1. If the polybius square was created using a keyword, as is usually the case, then the last row will probably contain the last 5 characters in the alphabet or at least most of them, in alphabetical order.

2. Doing a digram frequency analysis, with enough ciphertext, may produce some digram correspondences, for the more frequent digrams like th, he, in, er, an, re, on, at, en, nd, st, or, te, es, and is.

3. Since RE and ER are two common digrams, if we find in our ciphertext that a pair of digrams are frequent and reversed then these are likely to correspond to E and R. The same is true for TH and HT although HT is not as frequent.

4. Character duplication between two ciphertext digrams or between a ciphertext digram and a probable plaintext digram creates a, so to speak, pivot position that can help in setting the locations of the paired characters.

5. If you have deduced, in some manner, some of the plaintext. If there is a letter in common between the plaintext and the ciphertext then you know that the letters were on the same row or the same column and adjacent to each other.

6. If you know something about the message you can use that information to make educated guesses about the contents. For example, although this does not happen too much in modern times, many of these encryptions came from telegrams, which commonly ended with the word STOP. This gave you two easy digrams. Also, if the telegram was military in origin, there was probably a time given in the text somewhere. Considering that the military tends to get started early you could expect many of those times to start with a 0, such as 06:00 hours. The numbers would be spelled out since there are no places for digits in the polybius square. This would produce a large number of words ZERO, so looking for a large quadgram frequency may be helpful, especially since we are fairly sure where Z is positioned in the square.

For a much more detailed description of how to create a Playfair key from ciphertext please see chapter 7 of the United States Army Field Manual 34-40-2.[63]

If we invoke the computer we have another option. Brute force would not be a smart option since there are still $25! = 15,511,210,043,330,985,984,000,000$ keys in the keyspace. On the other hand, we could do a hill climbing algorithm similar to the one discussed in the section on monoalphabetic substitution. One possible algorithm for this is the following.

1. Start by constructing a random 5×5 polybius square or one that has the last 5 alphabetic characters in the last row, in order.

2. Use this square to decrypt the ciphertext and calculate a fitness measure for the decryption.
3. Now begin the hill climbing process. Take all possible transpositions of entries in the polybius square, transpose cell $(1, 1)$ and $(1, 2)$, then $(1, 1)$ and $(1, 3)$, down to $(1, 1)$ and $(5, 5)$, then $(1, 2)$ and $(1, 3)$, and so on, there are 300 of these in total. Each time a transposition is made use the new polybius square to decrypt the message and calculate a fitness measure on the new text. If the fitness measure increases, use the new polybius square and continue the process and if the fitness measure does not increase, revert to the previous polybius square and continue the process. Once we have tried all possible transpositions we will call this a single pass.
4. When we are done with a pass we compare the fitness measure before the pass was made with the fitness measure after the pass was made and if the fitness measure increased during the pass we make another pass. If the fitness measure did not change during the pass then we stop the process and report the last polybius square as our best guess to the key matrix.

2.7.3 Cryptography Explorer

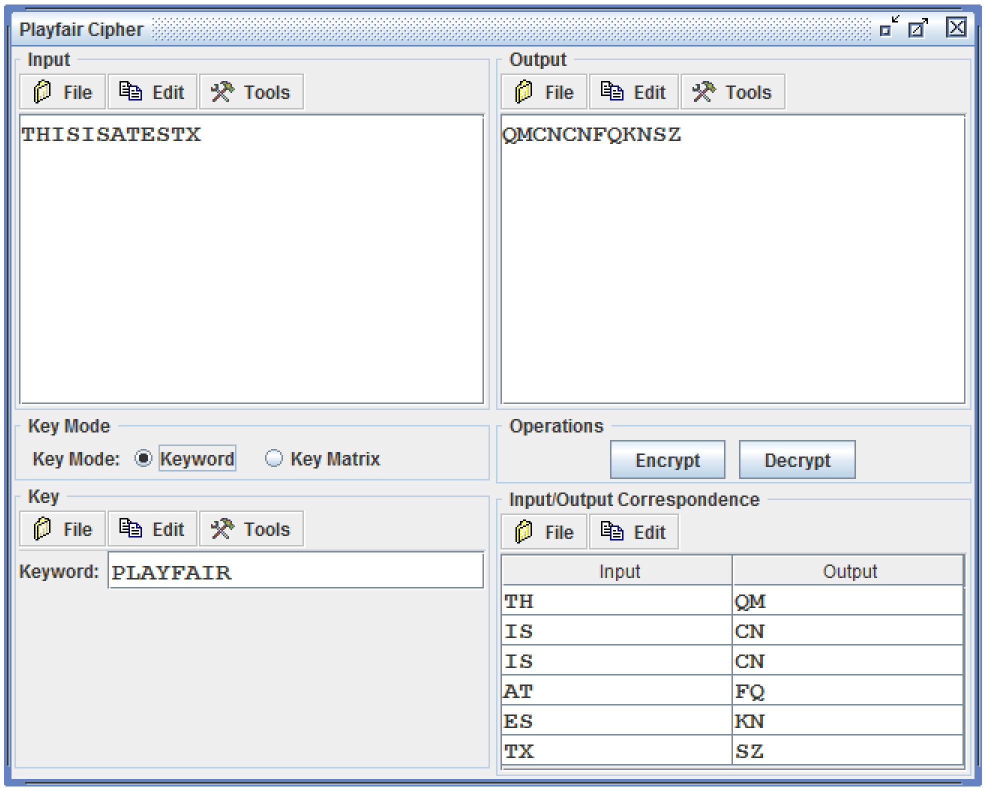
The Cryptography Explorer program has a built-in tool for encryption and decryption using the Playfair cipher. To invoke the tool, select Ciphers > Playfair from the main menu.

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 5×5 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. 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.



Input: THISISATESTX

Output: QMCNCNFQKNSZ

Key Mode: [✓ Keyword] Key Matrix

Keyword: PLAYFAIR

Input	Output
TH	QM
IS	CN
IS	CN
AT	FQ
ES	KN
TX	SZ

Figure 2.54: Playfair Cipher Tool: Keyword Mode

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.

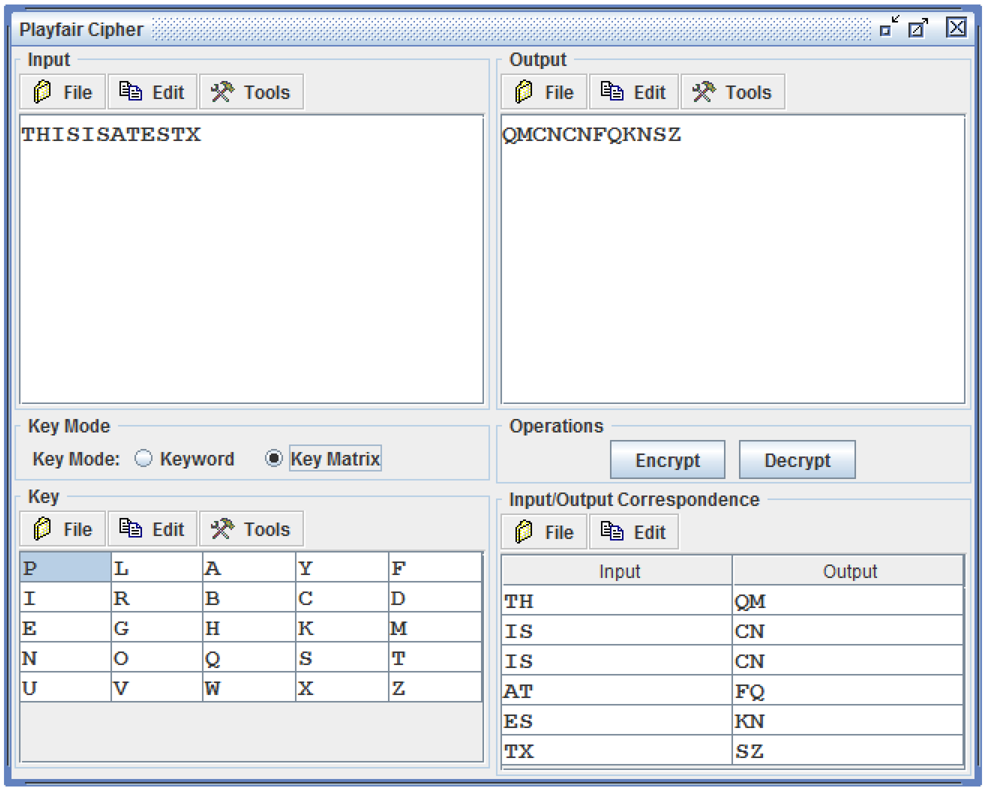
To Decrypt —

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 × 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.



Input: THISISATESTX

Output: 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 2.55: Playfair Cipher Tool: Matrix Mode

You can find a more detailed description of the options for this tool in the appendix on the Cryptography Explorer program.

In the cryptanalysis section of this chapter we discussed that one of the possible methods to get a plaintext/ciphertext correspondence of digrams is by digram frequency analysis. If you read the monoalphabetic substitution section you know that the Cryptography Explorer program has a tool for frequency analysis. We will not discuss it again her but refer the reader to the monoalphabetic substitution section or the frequency analysis tool section in the appendix on the Cryptography Explorer program. One note we would like to make here is that since the plaintext and ciphertext are blocked into blocks of two characters, that are by design not to overlap, when you are using the frequency analysis tool on Playfair ciphertext you should set the interlacing option to Non-Interlaced.

Example 29: If we take the ciphertext from the example above,

```
IWSZQMNIAYOTUHIGQOIRNYAYLSTICNLAYFENUGONWPYVAFQBMZ
VOTRQVONQNKGIKYVAYDOTLEUMZKGTZQBNTQMMICNLAYFIMLYDO
EGONUBQMFQVGBKBQUHYVFOKGPRHKNTQBIFGVHBRALKQOKNQGFB
QMKNQFRBQCZNRVVSSCNVLTPUMZKGOQNDIKIRITSFNVPKXYQYRI
LBQMVIPKNCIRIRQZYQQOIRNYAYPCONKGRQSZNQGTPYRVKSMIPD
PROEBYCRUNNTNZKSEUFBCNXNMIAYWLNQCLUBQMYQREOQONKGRT
VSCOYFEUHRHULBGNDZKGPGNLLBCZ
```

and do a frequency analysis on the digrams in a non-interlaced mode we get the following higher frequency counts.

Characters	**Frequency**	**Relative Frequency**
KG	7	0.042682926829268296
QM	6	0.036585365853658534
ON	5	0.03048780487804878
IR	5	0.03048780487804878
AY	5	0.03048780487804878

Going back to the polybius square for this example,

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

We see that the correspondence between the higher frequency digrams in the ciphertext and the plaintext they came from is as follows.

Ciphertext	**Plaintext**
KG	HE
QM	TH
ON	NT
IR	DI
AY	LA

Although, NT, DI, and LA are not high on the digram list, we have identified TH and HE. Δ

2.8 Two Square

2.8.1 Definitions and History

Historically, the Four Square cipher, which was developed by Félix Marie Delastelle, came before the two square cipher. The two square and four square ciphers are very similar in construction and use but the two square cipher is a little easier to set up. In Félix Delastelle's 1902 book *Traité Élémentaire de Cryptographie*[9], he presents methods that are very close to the two square method we discuss here, the main difference is that one of his polybius squares (either the top or right) is the ordered alphabet square that is used in the upper left and lower right of the four square method.

The Two Square cipher, like the Playfair cipher and Four Square 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 5×5 grid. So the plaintext cannot have any J's in it. Also like the Playfair cipher, the number of characters in the plaintext must be even, since the technique uses a digram substitution, so a dummy character (like X) might need to be added to the end. Unlike the Playfair cipher, the plaintext can have double letter digrams, so there is no need to separate these with an X as we did with Playfair.

The Two Square cipher uses two 5×5 grids of letters, these grids are arranged either vertically or horizontally with each other. The 5×5 grids are the key matrices which are both polybius squares similar to those constructed for the Playfair cipher. We construct these squares the same way we did for the Playfair cipher. We could either just fill in the two grids or use a pair of keywords. If we use a keyword, then the polybius squares are filled in just like they were with the Playfair cipher.

Usually the keywords are given as single words or short phrases. The word or phrase is altered as follows, first all J's are converted to I's, then any duplication of characters is removed. At this point the word is written in a 5×5 grid, starting in the upper left cell and moving across the first row, then the second and so on until the characters of the keyword are exhausted. The remaining cells in the grid are filled with the remaining letters of the alphabet (again, I and J are the same) 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. From there, we continue with B, C, D, and so on.

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.

Formally, we could define the two square cipher as follows.

Definition 20: *The Two Square cipher is a digram substitution cipher using two* 5×5 *polybius squares with the following rules.*

Polybius Square Setup

Each of the two polybius squares is a 5×5 grid of characters such that each cell contains a single character from the alphabet, excluding J, since all J's are changed to I in the plaintext. Although any configuration of this type is acceptable, the grid is usually set up using a keyword in the following manner. Take the keyword and convert all of the J's to I's, then remove any duplication of characters. Write the word is in the 5×5 grid by starting in the upper left cell and moving across the first row, then the second and so on until the characters of the altered keyword are exhausted. The remaining cells in the grid are filled with the remaining letters of the alphabet (excluding J) in alphabetical order. The two polybius squares are then arranged either vertically or horizontally. In the vertical arrangement the upper 5×5 grid is the top square and the lower one is the bottom square. In the horizontal arrangement, you have left and right squares respectively.

Plaintext Setup

To prepare the plaintext for encryption we do the following.

1. *Convert all characters to uppercase, remove all spaces, punctuation, and numbers.*
2. *Change all J's to I's.*
3. *Break the plaintext into blocks of two characters. If needed, add an X to the end of the plaintext to make the number of characters in the plaintext even.*

Encryption Procedure

Encrypt each plaintext digram using the 5×5 polybius squares as follows. The process breaks into two cases depending on the positions of the plaintext characters in the polybius squares.

1. *If the letters of the plaintext diagram are not in the same row or column. In this case we use the plaintext diagram character positions as the corners of a rectangle and encrypt the digram with the other two corners, using the entry in the same row as the first plaintext character for the first ciphertext character and using the entry in the same row as the second plaintext character for the second ciphertext character.*
2. *If the letters of the plaintext diagram are in the same row or column, the plaintext digram is unaltered.*

Finally, we would concatenate all of the ciphertext digrams together.

Decryption Procedure

The decryption process is simply a reversal of the encryption process. Specifically, to decrypt we take each ciphertext digram and use the 5×5 polybius squares as follows. The process again breaks into two cases depending on the positions of the plaintext characters in the polybius square.

1. *If the letters of the ciphertext diagram are not in the same row or column. In this case we use the ciphertext diagram character positions as the corners of a rectangle and encrypt the digram with the other two corners, using the entry in the same row as the first ciphertext character for the first plaintext character and using the entry in the same row as the second ciphertext character for the second plaintext character.*

2. *If the letters of the ciphertext diagram are in the same row or column, then the ciphertext digram is unaltered.*

Finally, we would concatenate all of the plaintext digrams together.

Example 30: Let's take an example from Act 3, Scene 1 of *The Life and Death of Julies Caesar* by William Shakespeare.[52]

```
Doth not Brutus bootless kneel?
Speak, hands for me!
Et tu, Brute!
Liberty! Freedom! Tyranny is dead!
Run hence, proclaim, cry it about the streets.
```

Converting this to a usable plaintext gives us the following, Notice the X at the end to make the number of characters even.

```
DOTHNOTBRUTUSBOOTLESSKNEELSPEAKHANDSFORMEETTUBRUTE
LIBERTYFREEDOMTYRANNYISDEADRUNHENCEPROCLAIMCRYITAB
OUTTHESTREETSX
```

Breaking this into digrams,

```
DO TH NO TB RU TU SB OO TL ES SK NE EL SP EA KH AN DS FO RM EE TT UB
RU TE LI BE RT YF RE ED OM TY RA NN YI SD EA DR UN HE NC EP RO CL AI
MC RY IT AB OU TT HE ST RE ET SX
```

If we use the keywords TWO and SQUARE we get the following polybius squares.

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

We will use a vertical placement of the two grids with this example, so our combined square is

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

Following our encryption rules, the plaintext digrams are converted to the following ciphertext digrams.

```
EN WG QM WE QA OS PF OO BG CU RL NE GI RT FU II WP CQ EP NP CC BM VE
QA TE KK TF SP ZD ND FC TO AV RA PM XK RF FU GQ VM HE QE FO QP GG OK
KF RY MN WD OU BM HE ST ND GO QZ
```

Giving a final ciphertext of,

```
ENWGQMWEQAOSPFOOBGCURLNEGIRTFUIIWPCQEPNPCCBMVEQATE
KKTFSPZDNDFCTOAVRAPMXKRFFUGQVMHEQEFOQPGGOKKFRYMNWD
OUBMHESTNDGOQZ
```

Δ

Example 31: Had we instead used a horizontal arrangement in the above example, our combined square would have been,

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

The ciphertext digrams in this case would be,

```
PC HQ NO CQ AO TU GN QU HR OE MP CM KF SP OD KH RQ WE RC RM EE NR CW
AO CS LI GS RT FZ FM ED QS UA AP NN LX GP OD WF NW CG CO QD RO HF LU
GI YP PL FQ OU NR CG ST FM QF ZO
```

Giving a ciphertext of,

```
PCHQNOCQAOTUGNQUHROEMPCMKFSPODKHRQWERCRMEENRCWAOCS
LIGSRTFZFMEDQSUAAPNNLXGPODWFNWCGCOQDROHFLUGIYPPLFQ
OUNRCGSTFMQFZO
```

Δ

2.8.2 Cryptanalysis

The cryptanalysis of the two square cipher is similar to that of the Playfair cipher. This should make sense since they are both digram ciphers using polybius squares. The big difference is that with the two square cipher we have twice as many letters to place and we cannot use the polybius square off of itself as we can with the Playfair cipher. On the other hand, the two square cipher has a substantial weakness. When the letters of a digram are in the same row or column the digram is unchanged, this is called a transparency, you can see the plaintext through the ciphertext. With enough ciphertext, there is a good chance that you will get several transparencies close to each other which will outline words enough to guess at the non-transparencies that are around them. Certainly if we had a large enough crib we could use the correspondence to build the polybius squares or at least most of them.

If we are attacking this by hand, here are a few things to keep in mind when constructing the squares. These are very similar to those for breaking the Playfair cipher.

1. If the polybius squares were created using a keyword, as is usually the case, then the last row will probably contain the last 5 characters in the alphabet or at least most of them, in alphabetical order.

2. Doing a digram frequency analysis, with enough ciphertext, may produce some digram correspondences, for the more frequent digrams like th, he, in, er, an, re, on, at, en, nd, st, or, te, es, and is.

3. Look for possible transparencies, these not only help in constructing a crib but also signify characters that are either on the same row, in horizontal alignment, or in the same column, in the vertical alignment.

4. Character duplication between two ciphertext digrams or between a ciphertext digram and a probable plaintext digram creates a, so to speak, pivot position that can help in setting the locations of the paired characters.

5. If you have deduced, in some manner, some of the plaintext. If there is a letter in common between the plaintext and the ciphertext then you know that the letters were on the same row or the same column and adjacent to each other.

6. If you know something about the message you can use that information to make educated guesses about the contents. For example, although this does not happen too much in modern times, many of these encryptions came from telegrams, which commonly ended with the word STOP. This gave you two easy digrams. Also, if the telegram was military in origin, there was probably a time given in the text somewhere. Considering that the military tends to get started early you could expect many of those times to start with a 0, such as 06:00 hours. The numbers would be spelled out since there are no places for digits in the polybius square. This would produce a large number of words ZERO, so looking for a large quadgram frequency may be helpful, especially since we are fairly sure where Z is positioned in the square.

For a much more detailed description of how to create a Playfair key from ciphertext please see chapter 7 of the United States Army Field Manual 34-40-2.[63]

If we invoke the computer we have another option. Brute force would still not be a smart option since there are $(25!)^2 \approx 2.4 \times 10^{50}$ keys in the keyspace. On the other hand, we could do a hill climbing algorithm similar to the one discussed in the section on monoalphabetic substitution. This is essentially the same algorithm as the hill climbing procedure for the Playfair cipher, as would be expected. The only difference is that we need to process two squares simultaneously. One possible algorithm for this is the following.

1. Start by constructing two random 5×5 polybius squares or ones that has the last 5 alphabetic characters in the last row, in order. Arrange these in either the vertical or horizontal alignment, depending on the assumed setup of the encryption. If the alignment is not known, then run the algorithm on both vertical and horizontal alignments and take the one with the best fitness measure.

2. Use this combined square to decrypt the ciphertext and calculate a fitness measure for the decryption.

3. Now begin the hill climbing process. Take all possible transpositions of entries in the top or left polybius square, transpose cell $(1, 1)$ and $(1, 2)$, then $(1, 1)$ and $(1, 3)$, down to $(1, 1)$ and $(5, 5)$, then $(1, 2)$ and $(1, 3)$, and so on, there are 300 of these in total. Each time a transposition is made use the new polybius square to decrypt the message and calculate a fitness measure on the new text. If the fitness measure increases, use the new polybius square and continue the process and if the fitness measure does not increase, revert to the previous polybius square and continue the process.

   Now take all possible transpositions of entries in the bottom or right polybius square, transpose cell $(1, 1)$ and $(1, 2)$, then $(1, 1)$ and $(1, 3)$, down to $(1, 1)$ and $(5, 5)$, then $(1, 2)$ and $(1, 3)$, and so on, there are 300 of these in total. Each time a transposition is made use the new polybius square to decrypt the message and calculate a fitness measure on the new text. If the fitness measure increases, use the new polybius square and continue the process and if the fitness measure does not increase, revert to the previous polybius square and continue the process.

   Once we have tried all possible transpositions on both polybius squares we will call this a single pass.

4. When we are done with a pass we compare the fitness measure before the pass was made with the fitness measure after the pass was made and if the fitness measure increased during the pass we make another pass. If the fitness measure did not change during the pass then we stop the process and report the last polybius square as our best guess to the key matrix.

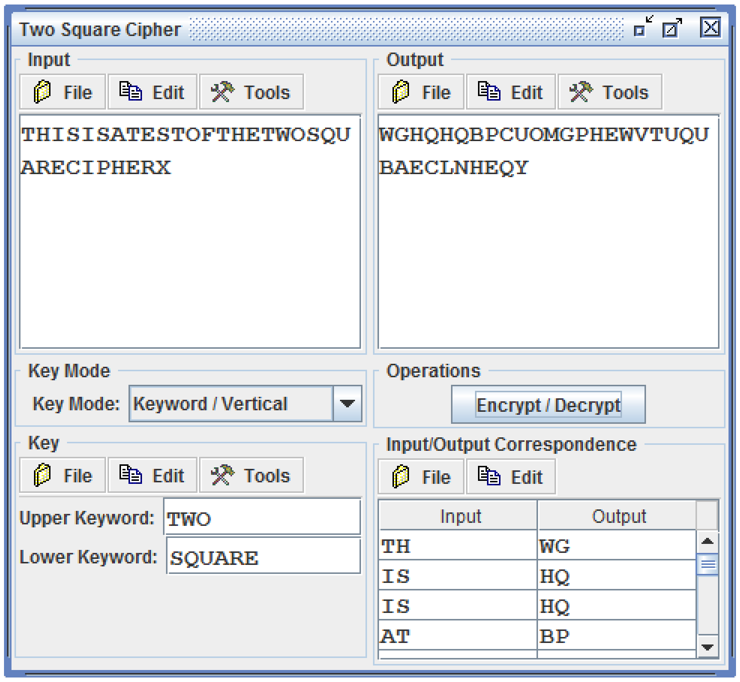
2.8.3 Cryptography Explorer

The Cryptography Explorer program has a built-in tool for encryption and decryption using the Two Square cipher. To invoke the tool, select Ciphers > Two Square from the main menu.

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 5×5 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 5×5 grids.

The Two Square cipher uses two 5×5 grids of letters arranged either vertically or horizontally. The 5×5 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.

The program has options for using either a keyword or creating the matrix by hand. It also has options for aligning the two polybius squares either vertically or horizontally.



```
THISISATESTOFTHETWOSQU
ARECIPHERX
```

```
WGHQHQBPCUOMGPHEWVTUQU
BAECLNHEQY
```

Key Mode: Keyword / Vertical

Upper Keyword: TWO

Lower Keyword: SQUARE

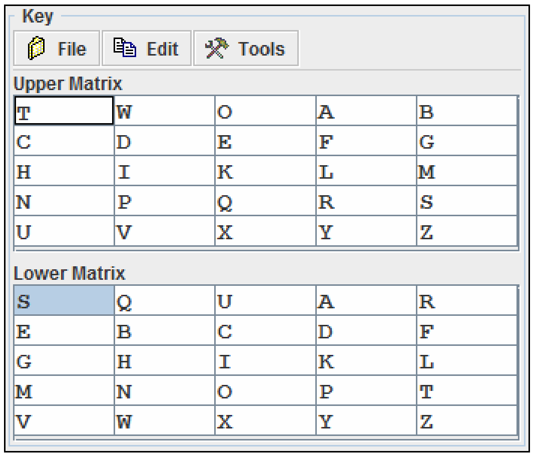
Input	Output
TH	WG
IS	HQ
IS	HQ
AT	BP

Figure 2.56: Two Square Cipher Tool: Keyword Mode

In key matrix mode, the keyword input is replaced with two 5×5 grids.

How to Use the Tool

To Encrypt or Decrypt —



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 2.57: Two Square Cipher Tool: Key Matrix Mode

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.

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 5×5 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.

You can find a more detailed description of the options for this tool in the appendix on the Cryptography Explorer program.

In the cryptanalysis section of this chapter we discussed that one of the possible methods to get a plaintext/ciphertext correspondence of digrams is by digram frequency analysis. If you read the monoalphabetic substitution section you know that the Cryptography Explorer program has a tool for frequency analysis. We will not discuss it again here but refer the reader to the monoalphabetic substitution section or the frequency analysis tool section in the appendix on the Cryptography Explorer program. One note we would like to make here is that since the plaintext and ciphertext are blocked into blocks of two characters, that are by design not to overlap, when you are using the frequency analysis tool on two square ciphertext you should set the interlacing option to Non-Interlaced.

2.9 Four Square

2.9.1 Definitions and History

The Four Square cipher was developed by Félix Marie Delastelle. Delastelle was not an academic, or a diplomat, or in the military, the types of careers that would lend themselves to the study of cryptography. He worked on the docks at a shipping port for the majority of his life, and took up cryptography as a hobby. After he retired in 1900, at age 60, he wrote a book on cryptography, *Traité Élémentaire de Cryptographie*[9], which was published a few months after his death in 1902. In his book he presented several of the cryptographic techniques that he had created. Of these, the most notable were the Four Square ciphers and two fractionation ciphers, the bifid cipher and the three-dimensional version of it called the trifid cipher.[26]

A Fractionation Cipher is where a single character or symbol in the plaintext is given at least two bits of information that determines the letter, such as A = 01, B = 02, . . . , Z = 26, and these bits are separated and combined in a different way to produce different letters for the ciphertext. In the bifid cipher, each letter is given two numbers, a row and column number, and in the trifid cipher each letter is given three numbers, a row, column, and level.

Example 32: We will do a quick example of the bifid cipher. With this cipher we create a 5×5 polybius square which could be randomly assigned or created from a keyword as with the Playfair cipher. In this case we label the rows and columns with the numbers 1–5. As we have done in the past, we equate I and J so that the alphabet fits into a 5×5 grid. For example,

	1	2	3	4	5
1	B	G	W	K	Z
2	Q	P	N	D	S
3	I	O	A	X	E
4	F	C	L	U	M
5	T	H	Y	V	R

Now say that we want to encrypt the message "flee at once", we first change all characters to uppercase and remove all spaces to get FLEEATONCE. Create a grid with three rows and as many columns as letters in the plaintext. Place the plaintext in the first row, one character per cell. On the second row put the row number of the position of the character, and on the third row put the column number of the position of the character. So since F is in row 4 column 1, we put a 4 in row two under the F and a 1 in row three under the F.

F	L	E	E	A	T	O	N	C	E
4	4	3	3	3	5	3	2	4	3
1	3	5	5	3	1	2	3	2	5

Now read the second and third rows left to right to get, 44333532431355312325, and divide the string into blocks of two numbers to get, 44 33 35 32 43 13 55 31 23 25. Finally, read these as row/column positions and substitute the character in that position. For example, in row 4 column 4 is U, in row 3 column 3 is A, in row 3 column 5 is E, and so on.

44	33	35	32	43	13	55	31	23	25
U	A	E	O	L	W	R	I	N	S

The ciphertext is this new set of letters, UAEOLWRINS.

Δ

The Four Square Cipher, like the Playfair cipher and Two Square 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 5×5 grid. So the plaintext cannot have any J's in it. 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 5×5 grids of letters arranged in a 2×2 grid. The upper left and lower right 5×5 grids are the alphabet, in order with I and J representing the same cell, 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 5×5 grids are the key matrices, the standard polybius squares as with the Playfair and two square ciphers. We construct these squares the same way we did for the Playfair cipher. We could either just fill in the two grids randomly or we could use a pair of keywords. If we use a keyword, then the polybius squares are filled in just like they were with the Playfair cipher.

Usually the keywords are given as single words or short phrases. The word or phrase is altered as follows, first all J's are converted to I's, then any duplication of characters is removed. At this point the word is written in a 5×5 grid, starting in the upper left cell and moving across the first row, then the second and so on until the characters of the keyword are exhausted. The remaining cells in the grid are filled with the remaining letters of the alphabet (again, I and J are the same) 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. From there, we continue with B, C, D, and so on.

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 rectangle, and read off the letters at the other two vertices, upper right followed by lower left. These are your ciphertext letters.

Formally, we could define the four square cipher as follows.

Definition 21: *The Four Square cipher is a digram substitution cipher using two 5×5 polybius squares with the following rules.*

Polybius Square Setup

Each of the two polybius squares is a 5×5 grid of characters such that each cell contains a single character from the alphabet, excluding J, since all J's are changed to I in the plaintext. Although any configuration of this type is acceptable, the grid is usually set up using a keyword in the following manner. Take the keyword and convert all of the J's to I's, then remove any duplication of characters. Write the word is in the 5×5 grid by starting in the upper left cell and moving across the first row, then the second and so on until the characters of the altered keyword are exhausted. The remaining cells in the grid are filled with the remaining letters of the alphabet (excluding J) in alphabetical order.

The two polybius squares are then placed in the 2×2 grid, in the upper right and lower left, along with the alphabet squares in the upper left and lower right.

Plaintext Setup

To prepare the plaintext for encryption we do the following.

1. *Convert all characters to uppercase, remove all spaces, punctuation, and numbers.*
2. *Change all J's to I's.*
3. *Break the plaintext into blocks of two characters. If needed, add an X to the end of the plaintext to make the number of characters in the plaintext even.*

Encryption Procedure

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 rectangle, and read off the letters at the other two vertices, upper right followed by lower left. These are your ciphertext letters. Finally, we would concatenate all of the ciphertext digrams together.

Decryption Procedure

To decrypt find the positions of the digram characters in the key grids, upper right for the first letter and lower left for the second. Using these two positions, create a rectangle, and read off the letters at the other two vertices, upper left followed by lower right. These are your plaintext letters. Finally, we would concatenate all of the plaintext digrams together.

Example 33: For this example we will use my second line from Douglas Adams's *The Hitchhikers Guide to the Galaxy.* If you have never read the book, you should if you like British humor. I will not delve into the plot here but there is a guide book *The Hitchhikers Guide to the Galaxy* which is a pun off of the guide books of the 1970's, like *Frommer's Europe from \$85 a Day.* The guide told you all the important thinks you needed to know when hitchhiking through the universe. In the guide, there is an entry on alcohol, which states that the best drink in existence is the Pan Galactic Gargle Blaster. Then it describes the effect of the Pan Galactic Gargle Blaster as, It says that the effect of a Pan Galactic Gargle Blaster is like having your brains smashed out by a slice of lemon wrapped round a large gold brick.

"The Guide also tells you on which planets the best Pan Galactic Gargle Blasters are mixed, how much you can expect to pay for one and what voluntary organizations exist to help you rehabilitate afterwards." But I am getting a bit off topic here. We will encrypt the above quote using the four square cipher with the keywords FOUR and SQUARE. So the keywords, FOUR and SQUARE produce the polybius squares,

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

Taking the plaintext, we convert it to uppercase, and even though we do not need to do this here, we convert all J's to I's and pad with an X at the end if the number of characters is odd.

```
ITSAYSTHATTHEEFFECTOFAPANGALACTICGARGLEBLASTERISLI
KEHAVINGYOURBRAINSSMASHEDOUTBYASLICEOFLEMONWRAPPED
ROUNDALARGEGOLDBRICK
```

Break the text stream into blocks of two characters.

```
IT SA YS TH AT TH EE FF EC TO FA PA NG AL AC TI CG AR GL EB LA ST ER
IS LI KE HA VI NG YO UR BR AI NS SM AS HE DO UT BY AS LI CE OF LE MO
NW RA PP ED RO UN DA LA RG EG OL DB RI CK
```

Encrypt the digrams, note that IT encrypts to EP, SA to NU and so on.

```
EP NU XP QD RM QD AR BE UR SK BS HR IC FG US SD OC OM BH OR HS SO OT
DP LE GR BU YE IC YK PT ON RE KO PI UM GU RK ST RW UM LE AU HD MS LH
IX NQ ML RR SH QL FA HS PB OF HK OA SB AC
```

Concatenating this produces the ciphertext.

```
EPNUXPQDRMQDARBEURSKBSHRICFGUSSDOCOMBHORHSSOOTDPLE
GRBUYEICYKPTONREKOPIUMGURKSTRWUMLEAUHDMSLHIXNQMLRR
SHQLFAHSPBOFHKOASBAC
```

Δ

2.9.2 Cryptanalysis

The cryptanalysis of the four square cipher is similar to that of the Playfair cipher. This should make sense since they are both digram ciphers using polybius squares. The big difference is that with the four square cipher we have twice as many letters to place and we cannot use the polybius square off of itself as we can with the Playfair cipher. Also, the four square cipher does not have the weakness of transparencies as does the two square cipher. But the four square cipher does have a new weakness. Although it is not required that the upper left and lower right squares are the alphabet squares it is a common choice. If this is the case, then there will be duplications in some of the ciphertext letters. In general, any word segment of the form XYZY where the characters for X and Z are on the same row of the alphabet squares will produce a ciphertext of the form ABAC for some character A. Similarly, any word segment of the form XYXZ where the characters for Y and Z are on the same row of the alphabet squares will produce a ciphertext of the form BACA for some character A. Finding these patterns in the ciphertext can help in determining the plaintext correspondences.

If we are attacking this by hand, here are a few things to keep in mind when constructing the squares. These are very similar to those for breaking the Playfair cipher.

1. If the polybius squares were created using a keyword, as is usually the case, then the last row will probably contain the last 5 characters in the alphabet or at least most of them, in alphabetical order.
2. Doing a digram frequency analysis, with enough ciphertext, may produce some digram correspondences, for the more frequent digrams like th, he, in, er, an, re, on, at, en, nd, st, or, te, es, and is.

3. If the upper left and lower right squares are the alphabet squares. Look for the same ciphertext letters that are two spaces apart, that is, of the form, ABAC or BACA. These would have come from plaintext letters of the form XYZY and XYXZ respectively.

4. If you know something about the message you can use that information to make educated guesses about the contents. For example, although this does not happen too much in modern times, many of these encryptions came from telegrams, which commonly ended with the word STOP. This gave you two easy digrams. Also, if the telegram was military in origin, there was probably a time given in the text somewhere. Considering that the military tends to get started early you could expect many of those times to start with a 0, such as 06:00 hours. The numbers would be spelled out since there are no places for digits in the polybius square. This would produce a large number of words ZERO, so looking for a large quadgram frequency may be helpful, especially since we are fairly sure where Z is positioned in the square.

For a much more detailed description of how to create a Playfair key from ciphertext please see chapter 7 of the United States Army Field Manual 34-40-2.[63]

If we invoke the computer we have another option. Brute force would still not be a smart option since there are $(25!)^2 \approx 2.4 \times 10^{50}$ keys in the keyspace. On the other hand, we could do a hill climbing algorithm similar to the one discussed in the section on monoalphabetic substitution. This is essentially the same algorithm as the hill climbing procedure for the Playfair cipher, as would be expected. The only difference is that we need to process two squares simultaneously. One possible algorithm for this is the following.

1. Start by constructing two random 5×5 polybius squares or ones that has the last 5 alphabetic characters in the last row, in order. Arrange these in the 2×2 grid in the upper right and lower left positions.

2. Use this combined square to decrypt the ciphertext and calculate a fitness measure for the decryption.

3. Now begin the hill climbing process. Take all possible transpositions of entries in the top right polybius square, transpose cell $(1, 1)$ and $(1, 2)$, then $(1, 1)$ and $(1, 3)$, down to $(1, 1)$ and $(5, 5)$, then $(1, 2)$ and $(1, 3)$, and so on, there are 300 of these in total. Each time a transposition is made use the new polybius square to decrypt the message and calculate a fitness measure on the new text. If the fitness measure increases, use the new polybius square and continue the process and if the fitness measure does not increase, revert to the previous polybius square and continue the process.

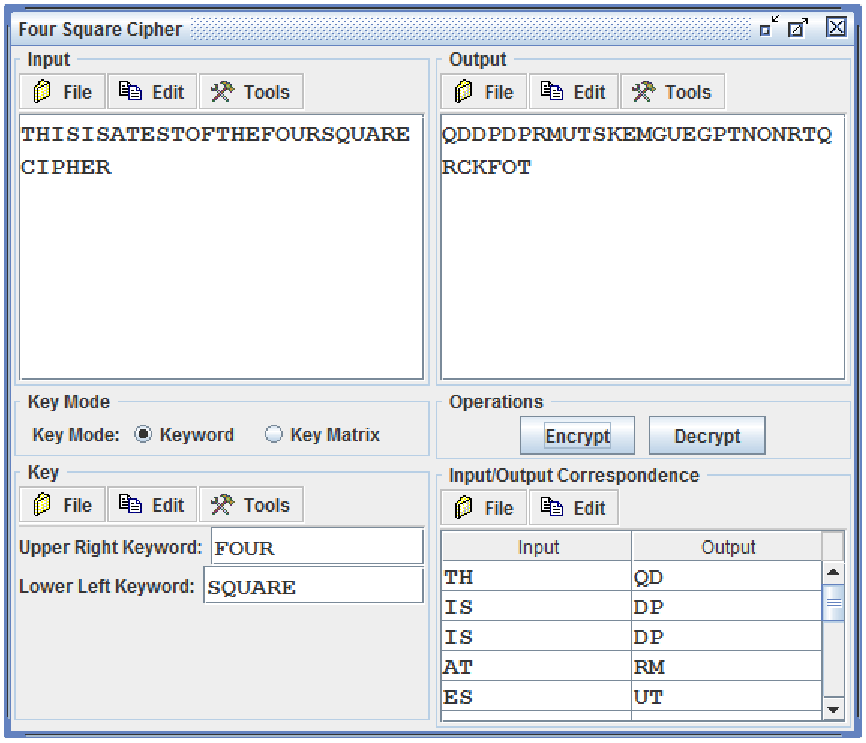
   Now take all possible transpositions of entries in the bottom left polybius square, transpose cell $(1, 1)$ and $(1, 2)$, then $(1, 1)$ and $(1, 3)$, down to $(1, 1)$ and $(5, 5)$, then $(1, 2)$ and $(1, 3)$, and so on, there are 300 of these in total. Each time a transposition is made use the new polybius square to decrypt the message and calculate a fitness measure on the new text. If the fitness measure increases, use the new polybius square and continue the process and if the fitness measure does not increase, revert to the previous polybius square and continue the process.

Once we have tried all possible transpositions on both polybius squares we will call this a single pass.

4. When we are done with a pass we compare the fitness measure before the pass was made with the fitness measure after the pass was made and if the fitness measure increased during the pass we make another pass. If the fitness measure did not change during the pass then we stop the process and report the last polybius square as our best guess to the key matrix.

2.9.3 Cryptography Explorer

The Cryptography Explorer program has a built-in tool for encryption and decryption using the Four Square cipher. To invoke the tool, select Ciphers > Four Square from the main menu.



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

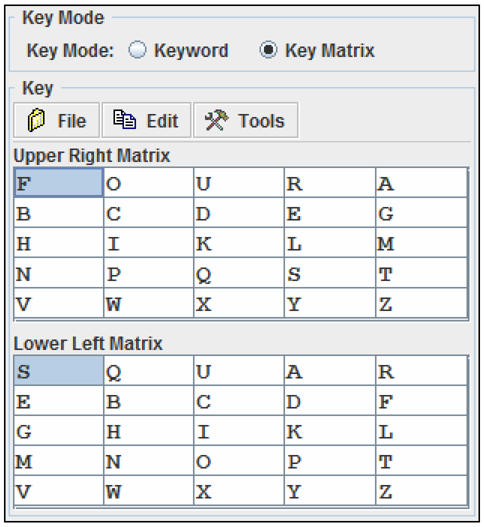
Figure 2.58: Four Square Cipher Tool: Keyword Mode

In key matrix mode, the keyword input is replaced with two 5×5 grids.

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



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 2.59: Four Square Cipher Tool: Matrix Mode

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 5×5 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.

You can find a more detailed description of the options for this tool in the appendix on the Cryptography Explorer program.

In the cryptanalysis section of this chapter we discussed that one of the possible methods to get a plaintext/ciphertext correspondence of digrams is by digram frequency analysis. If you read the monoalphabetic substitution section you know that the Cryptography Explorer program has a tool for frequency analysis. We will not discuss it again here but refer the reader to the monoalphabetic substitution section or the frequency analysis tool section in the appendix on the Cryptography Explorer program. One note we would like to make here is that since the plaintext and ciphertext are blocked into blocks of two characters, that are by design not to overlap, when you are using the frequency analysis tool on two square ciphertext you should set the interlacing option to Non-Interlaced.

2.10 ADFGX and ADFGVX

2.10.1 Definitions and History

The ADFGX Cipher was invented by Colonel Fritz Nebel in March of 1918, March 5 was the first time that an ADFGX cipher was intercepted. Then 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.

Table 2.29: ADFGVX Morse Code

Letter	**Code**
A	.–
D	–..
F	..–.
G	––.
V	...–
X	–..–

From a naive point of view, it is surprising that they added V, since V and X are so close, then again I am not a telegraphist.

The ADFGX and ADFGVX ciphers are a combination of a substitution cipher and a transposition cipher, specifically a classical columnar transposition cipher. The Germans believed the ADFGVX cipher was unbreakable, they were of course wrong.[18] Although it is a formidable cipher to break by hand, it can be and it was done.

There are two parts to the key for these ciphers, a keyword, which is used for the columnar transposition portion, and a polybius square which is used for the substitution portion of the cipher. If we are only using the letters ADFGX then the polybius square is a 5×5 grid of alphabetic letters where we combine I and J, really we convert J to I, just as we have done with the Playfair, two square, and four square ciphers. On the other hand, if we are using the letters ADFGVX then the polybius square is a 6×6 grid. This gives us enough room for all the letters of the alphabet, with no deletions, and all the single digit numbers, 0–9. The polybius square can be randomly generated, which seems to have been how it was usually used in World War I, or we can use another keyword and generate the square in the same manner as we did for the Playfair, two square, and four square ciphers. One reason that the squares for this cipher were random and not generated from a keyword is that the ADFGX cipher was only used for a short time, most of the time the ADFGVX cipher was used. If one uses a keyword for the 6×6 grid, then not only does the end of the grid contain most of the final letters in the alphabet it also contains most, if not all, of the numbers. This would be far too much information and would make the determination of square mush easier.

Since this cipher has several components to it we will do a detailed example before we give a formal definition. We will use the same example as we did for the four square cipher.

Example 34: For this example we will use my second line from Douglas Adams's *The Hitchhikers Guide to the Galaxy.* If you have never read the book, you should if you like British humor. I will not delve into the plot here but there is a guide book *The Hitchhikers Guide to the Galaxy* which is a pun off of the guide books of the 1970's, like *Frommer's Europe from $85 a Day.* The guide told you all the important thinks you needed to know when hitchhiking through the universe. In the guide, there is an entry on alcohol, which states that the best drink in existence is the Pan Galactic Gargle Blaster. Then it describes the effect of the Pan Galactic Gargle Blaster as,

> It says that the effect of a Pan Galactic Gargle Blaster is like having your brains smashed out by a slice of lemon wrapped round a large gold brick.

"The Guide also tells you on which planets the best Pan Galactic Gargle Blasters are mixed, how much you can expect to pay for one and what voluntary organizations exist to help you rehabilitate afterwards." But I am getting a bit off topic here. We will encrypt the above quote using the ADFGX cipher with a keyword of DOUGLAS and a polybius square of

	A	D	F	G	X
A	C	V	Q	X	G
D	F	H	W	Z	S
F	E	N	P	M	B
G	Y	I	O	D	T
X	U	K	L	R	A

Note that with the polybius squares for this cipher we label the rows and columns with A, D, F, G, and X. For the ADFGVX cipher we would do the same with the six rows and columns. Now take the plaintext, convert it to uppercase, change all the J's to I's, and remove all of the spaces. Doing so gives us,

```
ITSAYSTHATTHEEFFECTOFAPANGALACTICGARGLEBLASTERISLI
KEHAVINGYOURBRAINSSMASHEDOUTBYASLICEOFLEMONWRAPPED
ROUNDALARGEGOLDBRICK
```

For each letter in the plaintext, find its position on the grid and then replace the plaintext character with the letter pair of the row letter followed by the column letter. For example, I is in the fourth row and the second column, so we replace it by GD. Similarly, we replace T by GX, S by DX and so on. Doing so gives us the following,

```
GDGXDXXXGADXGXDDXXGXGXDDFAFADADAFAAAGXGFDAXXFFXXFD
AXXXXFXXAAGXGDAAAXXXXGAXXFFAFXXFXXDXGXFAXGGDDXXFGD
XDFADDXXADGDFDAXGAGFXAXGFXXGXXGDFDDXDXFGXXDXDDFAGG
GFXAGXFXGAXXDXXFGDAAFAGFDAXFFAFGGFFDDFXGXXFFFFFAGG
XGGFXAFDGGXXXFXXXGAXFAAXGFXFGGFXXGGDAAXD
```

At this point the substitution portion of the cipher is finished. What we do now is a classical columnar transposition cipher on the above string of ADFGX characters, using our keyword DOUGLAS. Recall, in the classical columnar transposition cipher, we have as many columns as there are letters in the keyword. We write the plaintext, for us this will be the intermediate text, out as rows, left to right. Then we sort the columns so that the keyword is in alphabetical order. Finally, we read down the columns to create the ciphertext.

```
D O U G L A S            A D G L O S U
-------------            -------------
G D G X D X X            X G X D D X G
X G A D X G X            G X D X G X A
D D X X G X G            X D X G D G X
X D D F A F A            F X F A D A D
D A D A F A A            A D A F A A D
A G X G F D A            D A G F G A X
X X F F X X F            X X F X X F F
D A X X X X F            X D X X A F X
X X A A G X G            X X A G X G A
D A A A X X X            X D A X A X A
X G A X X F F            F X X X G F A
A F X X F X X            X A X F F X X
D X G X F A X            A D X F X X G
G G D D X X F            X G D X G F D
G D X D F A D            A G D F D D X
D X X A D G D            G D A D X D X
F D A X G A G            A F X G D G A
F X A X G F X            F F X G X X A
X G X X G D F            D X X G G F X
D D X D X F G            F D D X D G X
X X D X D D F            D X X D X F D
A G G G F X A            X A G F G A G
G X F X G A X            A G X G X X F
X D X X F G D            G X X F D D X
A A F A G F D            F A A G A D F
A X F F A F G            F A F A X G F
G F F D D F X            F G D D F X F
G X X F F F F            F G F F X F X
F A G G X G G            G F G X A G G
```

```
F X A F D G G            G F F D X G A
X X X F X X X            X X F X X X X
G A X F A A X            A G F A A X X
G F X F G G F            G G F G F F X
X X G G D A A            A X G D X A G
X D                        X     D
```

So our final ciphertext is

```
XGXFADXXXXFXAXAGAFDFDXAGFFFFGGXAGAGXDXDAXDXDXADGGD
FFXDXAGXAAGGFFXGGXXXDXFAGFXAAXXXDDAXXXDXGXXAFDFGFF
FFGDXGAFFXXGXXFFXFDGGGXDFGFGADFXDXAGDDGDDAGXAXAGFX
GDXDXGDXGXDAXFXAXXXAFXDXXGAAAFFGXFXXFDDGXFGFAXDDGXF
GGXXFAGAXDDXFXAAAXGDXXAAXXDGFXFFFXGAXXXG
```

Δ

Decryption is simply done in reverse. We would use the keyword to do a columnar decryption process on the ciphertext to obtain the intermediate text, then break the intermediate text into blocks of two and use the square grid to convert the letter pairs back to a plaintext message.

We can define it formally as follows.

Definition 22:

The ADFGX and ADFGVX ciphers are a combination of a monogram to digram substitution and a classical columnar transposition cipher. Both ciphers require a keyword and either a 5×5 polybius square for the ADFGX or a 6×6 polybius square for the ADFGVX.

Plaintext Setup

If the cipher is the ADFGX cipher then the plaintext must be all uppercase alphabetic letters and all of the J's must be converted to I's so that we use only 25 distinct characters. If the cipher is the ADFGVX cipher then the plaintext must be all uppercase characters and single digit numbers 0–9, in this case we do not convert J to I.

Polybius Square Setup

If the cipher is the ADFGX cipher then the polybius square is a 5×5 grid that contains all of the uppercase alphabetic letters, excluding J. If the cipher is the ADFGVX cipher then the polybius square is a 6×6 grid that contains all of the uppercase alphabetic letters and the digits 0–9. In either case, we label each row and column with A, D, F, G, X, or A, D, F, G, V, X, in that order. The polybius square can be set up using a keyword, in the same manner as with the Playfair cipher, or the cells can be assigned characters randomly. For the ADFGX cipher, using a keyword is acceptable, although it tends to weaken the cipher. For the ADFGVX cipher, unless a significant number of digits are used in the keyword, the use of a keyword is discouraged since it substantially weakens the cipher.

Substitution Step

For each plaintext letter, find its position in the polybius square, and then replace the character with the row label character followed by the column label character. At this point *we have an intermediate form of the ciphertext that has only the letters A, D, F, G, X, and possibly V.*

Transposition Step

At this point we apply a classical columnar transposition cipher to the intermediate form of the ciphertext. 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.

Decryption of the ADFGX and ADFGVX is simply the reverse of the encryption process, do a decryption using the classical columnar cipher with the keyword, then use the polybius square on the row/column digrams of the intermediate form to obtain the plaintext.

2.10.2 Cryptanalysis

The ADFGX cipher was cryptanalysed by French Army Lieutenant Georges Painvin and the cipher was broken in early April 1918. Painvin used similar methods to break the ADFGVX cipher in early June 1918, almost immediately after the cipher was launched.[40]

Painvin's method for solving the cipher is very close to the method we will examine later on in this section. It relied on having two ciphertexts that were encrypted with the same key and had plaintexts that started with the same couple words. By the way that the cipher is applied to the plaintext, having the same words at the beginning meant that the starts of the columns in the transposition step are the same between the two ciphertexts. This gave the analyst an idea of the length of the keyword and the lengths of the columns in the transposition step. Knowing this, and the column arrangement, reduced the cipher to a monoalphabetic substitution cipher, which is relatively easy to solve.

Painvin did not have the luxury of a computer at his disposal, and breaking a monoalphabetic cipher numerous times for each possible column arrangement was too time consuming. So Painvin attacked the polybius square in a different manner. In the case where the keyword is of even length, each column contains either a row label or a column label, and not a mixture of the two. In this case, he did frequency analysis on the row label columns and the column label columns. This told him the combined frequencies of the row or column letters. For example, with the square we used in the above example,

	A	D	F	G	X
A	C	V	Q	X	G
D	F	H	W	Z	S
F	E	N	P	M	B
G	Y	I	O	D	T
X	U	K	L	R	A

The frequency of the row label F would be the combined frequencies of E, N, P, M, and B. The frequency of the column label G would be the combined frequencies of X, Z, M, D, and R.[18] This also gave an indication as to which columns represented row labels and which were column labels. After that, he was able to pair up the labels and use frequency analysis to complete the grid.

The cryptanalysis of the ADFGX and ADFGVX ciphers that we will discuss is very similar to the way that Lieutenant Georges Painvin broke the cipher back in 1918. If you read the section on the columnar transposition cipher you know that one of the keys to breaking the cipher was the determination of the lengths of the columns, this gave you a good indication of the length of the keyword and in the cases where the columns have different lengths you could significantly reduce the size of the keyspace.

When we cryptanalyzed the columnar cipher, the way we found the lengths of the columns was by assuming that we had two different ciphertexts that were both encrypted with the same keyword, and in our case here the same polybius square, and that both of the encrypted plaintexts started out the same way. Then we looked for substrings of different lengths that were the same between both the ciphertexts, these indicated the beginning of columns in the columnar table. Once we have the column lengths, we needed to look at the permutations of these columns in order to get an English message.

For the ADFGX and ADFGVX we will do the same thing, take two ciphertexts that we are confident were encrypted with the same key and their plaintexts begin with the same few words. We will use the same substring method to determine the size of the keyword and the lengths of the columns. Now we have a small problem. When we start shifting the columns around and doing the columnar decryption we get back that intermediate string of ADFGX or ADFGVX characters. We need the polybius square to convert these back to alphabetic characters, but we do not have it. The nifty thing is that we do not need it. Let's say, for the sake of argument, that we have guessed the correct arrangement of the columns. What would happen if we simply used,

	A	D	F	G	X
A	A	B	C	D	E
D	F	G	H	I	K
F	L	M	N	O	P
G	Q	R	S	T	U
X	V	W	X	Y	Z

to convert the ADFGX or ADFGVX characters back to the English alphabet? Well, we would probably get gibberish since it is highly unlikely that this was the polybius square that was used. A better question is, what would the relationship be between this gibberish string we got and the plaintext we want? The only difference is that this polybius square is a character rearrangement of the one we want, hence the gibberish string is simply a character rearrangement of the plaintext. So the gibberish string is a monoalpahbetic substitution of the plaintext, in fact, the same monoalphabetic substitution that will convert the above polybius square to the one that was actually used.

This gives us a complete plan of attack for deciphering an ADFGX and ADFGVX ciphers.

1. Take two ciphertexts that we are confident were encrypted with the same key and their plaintexts begin with the same few words. Do a substring comparison between the two

to determine the length of the keyword and the lengths of the columns in the columnar grid. If there is some doubt to the keyword length we can usually narrow it down to several possibilities, in which case we do the following with each possibility.

2. Take a column arrangement from the set of possible column arrangements, in some cases you may need to look at all possible permutations of the columns but in others the column lengths will allow you to significantly reduce this number of possibilities. With this arrangement, use the columnar decryption to produce the intermediate string of ADFGX or ADFGVX characters.

3. Use the polybius square,

	A	D	F	G	X
A	A	B	C	D	E
D	F	G	H	I	K
F	L	M	N	O	P
G	Q	R	S	T	U
X	V	W	X	Y	Z

   to convert the ADFGX or ADFGVX character string to alphabetic characters.

4. Do a monoalphabetic substitution cryptanalysis on this alphabetic string. If successful, you have the decryption of the ciphertext and a key for transforming the above polybius square into the one that was used in the encryption process. On the other hand, if the monoalphabetic substitution process fails to produce the plaintext you would go back to the column arrangement step and use another arrangement.

   One thing to note about this step, since both ciphertexts were encrypted with the same key, once they are both converted to the English alphabet, they can be combined so that frequency analysis will have a better chance of producing the substitution key.

Example 35: Let's say that we have intercepted the following two ciphertexts,

Ciphertext 1:

```
FFXXGAXAFXDFDGXDDGDFDAGXXAXAGAXXFXGXDGFXGAGXXGDFDF
GGGDGFDFGDXFGGFXXAAXFGXXXFADXGFXXXGDXAXXXAAAXAFDXX
XAXXAFXADFDDDDAAXXDXXDGAXXFAFFGDGFAGAGFGAFGGAFDXXX
AXXAFXAXXXXDXDAAAFXXXFADXGFXXXGXDAXDXFGGDXGXAFXAXD
AAAFFXFGFDXXXXFXDDXAXAGAXXFXGXGDGGXGFDFGDFFXXGAXAF
XDFGXXDAFGXFXFDXDGXFXXAGXFFXFFXXGXXFGGXA
```

Ciphertext 2:

```
FFXXGAXAFXDFDDAGFXGFXGXFDAFXFDDFDFFXXGAXAFXDFDXFAG
GFAGXXFFAGFGGGFFDXAXFXGXXGXGFFGGGFFGXDFXGGXGXXGXXF
```

```
ADXGFXXXGXAAGXXXFXFFFDDFDXXAAXXXFADXGFXXXGXADADFAD
FADGAXFXAGDXGDXFAXGFXDGXFXGXXDFAAAXDDXAAAFGXFDXXXA
XXAFXAXDXDAFFAFAADDAFXAXDXXFDXXXAXXAFXAXGGGGFGAAAX
AFAXFAFFADFXDFDXAGXDXDDADXFAFFGDXFXXXXGXGXAXAGAXXF
XGXGFXGXDGXDGGDGDXFGGFFGXAXAGAXXFXGXXFGFDDXDGAFAFG
GXDXFXGDGGXGXGAFDGGDADGDXGXFXGGXFDFDXD
```

Notice that both of these begin with the same string of 13 characters, FFXXGAX-AFXDFD. This is a good indication that the ciphertexts were encrypted with the same key and that the beginning words of both plaintexts are the same. If not, then two different messages with two different keys had to produce the same 6 beginning characters. While this is not impossible, it is unlikely.

If we do a substring comparison on these two ciphertexrs, looking at substrings of length 13, we get the following.

```
Input 1 Length:  290
Input 2 Length:  388

Matches

FFXXGAXAFXDFD      Input 1: 0          Input 2: 0  33
XXXFADXGFXXXG      Input 1: 72  168    Input 2: 129
FDXXXAXXAFXAX      Input 1: 145        Input 2: 194  227
XXFADXGFXXXGX      Input 1: 169        Input 2: 97  130
XAXAGAXXFXGXG      Input 1: 218        Input 2: 291
DFFXXGAXAFXDF      Input 1: 240        Input 2: 32
```

We did this using the Cryptography Explorer program substring compare tool but the same analysis could have been done by hand. Before we start looking at the numbers, lets consider what we would likely see as column starts for different size keywords. Our ciphertext lengths are 290 and 388 respectively, lets look at where the column breaks would be for various sized keyword lengths and see if any match the above data. All we need to do is divide the ciphertext lengths by the keyword lengths, we rounded them here. Doing so gives us the following chart.

	Input 1	**Input 2**
Length	290	388
Length 3	97, 193	129, 259
Length 4	73, 145, 218	97, 194, 291
Length 5	58, 116, 174, 232	78, 155, 233, 310
Length 6	48, 97, 145, 193, 242	65, 129, 194, 259, 323
Length 7	41, 83, 124, 166, 207, 249	55, 111, 166, 222, 277, 333
Length 8	36, 73, 109, 145, 181, 218, 254	49, 97, 146, 194, 243, 291, 340

What we would be looking for in the substring comparison data are numbers around these, that is, plus or minus one since the columns could have different lengths. In addition, we would be looking for these numbers to correspond to the same substring. So if one substring was at position 48 in input 1 and 65 in input two and another substring was at position 145 in input 1 and 194 in input 2 this would suggest that the keyword length is six. On the other hand, if one substring was at position 48 in input 1 and a different substring was at position 65 in input 2 then we may have our doubts on the length being six. Looking at the substring data, which found matches with substrings of length 13, it appears that the length of the keyword is 4. Notice the 145 matching up with the 194 and the 218 matching up with the 291. There is also a 72 in input 1 but it does not match up with a 97 (or so) in input 2, which is a little concerning. On the other hand, the other lengths between 3 and 8 do not seem to fit the data. Normally, one would also look at substring length matches, for example, if we use a length of 12 instead of 13 we get the following.

```
Input 1 Length:  290
Input 2 Length:  388

Matches

FFXXGAXAFXDF     Input 1: 0  241        Input 2: 0  33
FXXGAXAFXDFD     Input 1: 1             Input 2: 1  34
XAXAGAXXFXGX     Input 1: 24  218       Input 2: 291  324
XXXFADXGFXXX     Input 1: 72  168       Input 2: 129
XXFADXGFXXXG     Input 1: 73  169       Input 2: 97  130
FDXXXAXXAFXA     Input 1: 96  145       Input 2: 194  227
DXXXAXXAFXAX     Input 1: 146           Input 2: 195  228
XFADXGFXXXGX     Input 1: 170           Input 2: 98  131
AXAGAXXFXGXG     Input 1: 219           Input 2: 292
DFFXXGAXAFXD     Input 1: 240           Input 2: 32
```

There are many more matches, as we would expect. Notice that now we have a 73 matching up with a 97, which was a concern we had in the last data set. In addition, the numbers are still pointing to a keyword length of 4. So we will go with that assumption, if we are wrong we can come back to this step and make a different choice, but given the numbers from these two data sets it seems unlikely that the keyword is not length 4. One possibility would be if the keyword is longer than 8 and the duplication between the two texts is shorter than we thought. This is possible, but with the beginning duplication of 13 characters it is unlikely.

Assuming that the length of the keyword is 4, let's continue. The first ciphertext is length 290. Dividing 290 by 4 gives us 72.5, also, we could write $290 = 72 \cdot 4 + 2$. This means that two columns will have length 72 and two columns will have length 73. This is actually very good news, it means that the number of possible arrangements of the columns is only 4 and not 24. With the second ciphertext, there are 388 characters which says that there are exactly 97 characters for each column. Although this does not reduce the number of possible column arrangements it does help determine the column breaks for the first ciphertext. Breaking the second ciphertext into its columns gives the following.

A	**B**	**C**	**D**
F	X	F	X
F	X	D	A
X	F	X	X
X	A	X	A
G	D	X	G
A	X	A	A
X	G	X	X
A	F	X	X
F	X	A	F
X	X	F	X
D	X	X	G
F	G	A	X
D	X	X	G
D	A	D	F
A	A	X	X
G	G	D	G
F	X	A	X
X	X	F	D
G	X	F	G
F	F	A	X
X	X	F	D
G	F	A	G
X	F	A	G
F	F	D	D
D	D	D	G
A	D	A	D
F	F	F	X
X	D	X	F
F	X	A	G
D	X	X	G
D	A	D	F
F	A	X	F
D	X	X	G
F	X	F	X
F	X	D	A
X	F	X	X
X	A	X	A
G	D	X	G

A	**B**	**C**	**D**
A	X	A	A
X	G	X	X
A	F	X	X
F	X	A	F
X	X	F	X
D	X	X	G
F	G	A	X
D	X	X	X
X	A	G	F
F	D	G	G
A	A	G	F
G	D	G	D
G	F	F	D
F	A	G	X
A	D	A	D
G	F	A	G
X	A	A	A
X	D	X	F
F	G	A	A
F	A	F	F
A	X	A	G
G	F	X	G
F	X	F	X
G	A	A	D
G	G	F	X
G	D	F	F
F	X	A	X
F	G	D	G
D	D	F	D
X	X	X	G
A	F	D	G
X	A	F	X
F	X	D	G
X	G	X	X
G	F	A	G
X	X	G	A
X	D	X	F
G	G	D	D
X	X	X	G
G	F	D	G
F	X	D	D

A	**B**	**C**	**D**
F	G	A	A
G	X	D	D
G	X	X	G
G	D	F	D
F	F	A	X
F	A	F	G
G	A	F	X
X	A	G	F
D	X	D	X
F	D	X	G
X	D	F	G
G	X	X	X
G	A	X	F
X	A	X	D
G	A	X	F
X	F	G	D
X	G	X	X
G	X	G	D

Breaking the second ciphertext into its columns gives the following. Due to the fact that the beginnings of the columns between the two ciphertexts must be identical, we have that there are 73 characters in the first and third columns and there are 72 characters in the second and fourth columns.

A	**B**	**C**	**D**
F	X	F	X
F	X	D	A
X	F	X	X
X	A	X	A
G	D	X	G
A	X	A	A
X	G	X	X
A	F	X	X
F	X	A	F
X	X	F	X
D	X	X	G
F	G	A	X
D	D	X	G
G	X	X	D
X	A	X	G
D	X	X	G
D	X	D	X

A	**B**	**C**	**D**
G	X	X	G
D	A	D	F
F	A	A	D
D	A	A	F
A	X	A	G
G	A	F	D
X	F	X	F
X	D	X	F
A	X	X	X
X	X	F	X
A	X	A	G
G	A	D	A
A	X	X	X
X	X	G	A
X	A	F	F
F	F	X	X
X	X	X	D
G	A	X	F
X	D	G	G
D	F	X	X
G	D	D	X
F	D	A	D
X	D	X	A
G	D	D	F
A	A	X	G
G	A	F	X
X	X	G	F
X	X	G	X
G	D	D	F
D	X	X	D
F	X	G	X
D	D	X	D
F	G	A	G
G	A	F	X
G	X	X	F
G	X	A	X
D	F	X	X
G	A	D	A
F	F	A	G
D	F	A	X
F	G	A	F

A	**B**	**C**	**D**
G	D	F	F
D	G	F	X
X	F	X	F
F	A	F	F
G	G	G	X
G	A	F	X
F	G	D	G
X	F	X	X
X	G	X	X
A	A	X	F
A	F	X	G
X	G	F	G
F	G	X	X
G	A	D	A
X		D	

Note that we put columns headers on both tables as ABCD since at this point in the decipherment the keyword would have been sorted into alphabetical order. From the first ciphertext above, we know that the longer columns must be the first two and the shorter columns must be the second two. This gives us four possibilities for a keyword, ACBD, CABD, ACDB, and CADB. For each of these, we will decrypt both ciphertexts, using the alphabetically ordered polybius square,

	A	D	F	G	X
A	A	B	C	D	E
D	F	G	H	I	K
F	L	M	N	O	P
G	Q	R	S	T	U
X	V	W	X	Y	Z

and then we will attempt to construct a monoalphabetic substitution to construct the plaintext.

ACBD

Applying the keyword ACBD and the alphabetically ordered polybius square gives the following two decryptions.

```
NZMVZPZAUIAVZUEPLXXZKYLUKIUWZDKYGZUYGCLBFCAYSBZNZH
EZXZAYRAEZYVXCPPZWUCYIKPRKLGZFRHEDSEYXYZRHKWOZKGLT
SEUXQZKPRALOFPLSSHHUZNNCTUSEMTZPZUECEOXTPURAW
```

```
NZMVZPZAUIAVZUEPLXXZKYLUKYGCEERTLZXWSYLPXWQOVOMMGI
AGNPZHLYKYGCPCKYNZMVZPZAUIAVZUEPLXXZKYLUKZYCOIDCTG
SMOEAGQOVAZHLQNCAYUONZQBSUSHLZMTHGZYBOXEMYZUQOYVZH
RRZYROMWLQRWUYSGLPNDSEYCGZPIXIUZUCZBUCYMZUTW
```

Combining these and applying our best guess to the substitution gives the following. While there seem to be word segments forming, this is clearly not the correct arrangement.

```
VEMBERESTUSBETHROLLEDIOTDUTYEZDINETINAOGXASICGEVEW
HELESIKSHEIBLARREYTAIUDRKDONEXKWHZCHILIEKWDYPEDNOJ
CHTLFEDRKSOPXROCCWWTEVVAJTCHMJERETHAHPLJRTKSYVEMBE
RESTUSBETHROLLEDIOTDINAHHKJOELYCIORLYFPBPMMNUSNVRE
WOIDINARADIVEMBERESTUSBETHROLLEDIOTDEIAPUZAJNCMPHS
NFPBSEWOFVASITPVEFGCTCWOEMJWNEIGPLHMIETFPIBEWKKEIK
PMYOFKYTICNORVZCHIANERULUTETAEGTAIMETJY
```

CABD

Applying the keyword CABD and the alphabetically ordered polybius square gives the following two decryptions.

```
NZHVZPZAYIAVZUVPCXPZWYCUWIYWZDWYGZYYGCCBBCAYOBZNZH
VZPZAYIAVZUVPCXPZWYCUIWPIKCGZFIHVDOEUXUZIHWWSZWGCT
OEYXDZWPIACOBPCSOHMUZNNCTUOEHTZPZUVCVOPTXUIAK
```

```
NZHVZPZAYIAVZUVPCXPZWYCUWYGCVEITCZPWOYCPPWDOEOHMGI
AGNPZHCYWYGCXCWYNZHVZPZAYIAVZUVPCXPZWYCUWZUCSIQCTG
OMSEAGDOEAZHCQNCAYYONZDBOUOHCZHTMGZYFOPEHYZUDOUVZH
IRZYIOHWCQIWYYOGCPNDOEUCGZXIPIYZYCZBYCUMZUTW
```

Combining these and applying our best guess to the substitution gives the following. Again, there seem to be word segments forming, but as with our last try, this is clearly not the correct arrangement.

```
GENPETEMSIMPERPTOFTEDSORDISDELDSHESSHOOKKOMSAKEGEN
PETEMSIMPERPTOFTEDSORIDTIJOHEWINPLAURFREINDDBEDHOC
AUSFLEDTIMOAKTOBANYREGGOCRAUNCETERPOPATCFRIMJGENPE
TEMSIMPERPTOFTEDSORDSHOPUICOETDASOTTDLAUANYHIMHGTE
NOSDSHOFODSGENPETEMSIMPERPTOFTEDSORDEROBIVOCHAYBUM
HLAUMENOVGOMSSAGELKARANOENCYHESWATUNSERLARPENIZESI
ANDOVIDSSAHOTGLAUROHEFITISESOEKSORYERCD
```

ACDB

Applying the keyword ACDB and the alphabetically ordered polybius square gives the following two decryptions.

```
NZMEZXZAURAEZYEXLPXZKULYKRUKZQKUGZUUGLLFFLAUSFZNZM
EZXZAURAEZYEXLPXZKULYRKXRWLGZBRMEQSVYPYZRMKKOZKGLT
SVUPQZKXRALSFXLOSMHYZNNLTYSVMTZXZYELESXTPYRAW
```

```
NZMEZXZAURAEZYEXLPXZKULYKUGLEVRTLZXKSULXXKQSVSMHGR
AGNXZMLUKUGLPLKUNZMEZXZAURAEZYEXLPXZKULYKZYLORDLTG
SHOVAGQSVAZMLDNLAUUSNZQFSYSMLZMTHGZUBSXVMUZYQSYEZM
RIZURSMKLDRKUUSGLXNQSVYLGZPRXRUZULZFULYHZYTK
```

Combining these and applying our best guess to the substitution gives the following. We have a winner.

```
PANGALACTICGARGLEBLASTERSITSAYSTHATTHEEFFECTOFAPAN
GALACTICGARGLEBLASTERISLIKEHAVINGYOURBRAINSSMASHED
OUTBYASLICEOFLEMONWRAPPEDROUNDALARGEGOLDBRICKPANGA
LACTICGARGLEBLASTERSTHEGUIDEALSOTELLSYOUONWHICHPLA
NETSTHEBESTPANGALACTICGARGLEBLASTERSAREMIXEDHOWMUC
HYOUCANEXPECTTOPAYFORONEANDWHATVOLUNTARYORGANIZATI
ONSEXISTTOHELPYOUREHABILITATEAFTERWARDS
```

The substitution key that was used is the following.

Plaintext	**Ciphertext**	**Plaintext**	**Ciphertext**
A	Z	N	M
B	P	O	S
C	A	P	N
D	T	Q	C
E	L	R	Y
F	F	S	K
G	E	T	U
H	G	U	V
I	R	V	B
J	J	W	H
K	W	X	D
L	X	Y	Q
M	O	Z	I

Applying this to the alphabetic square,

	A	D	F	G	X
A	A	B	C	D	E
D	F	G	H	I	K
F	L	M	N	O	P
G	Q	R	S	T	U
X	V	W	X	Y	Z

produces the grid,

	A	D	F	G	X
A	C	V	Q	X	G
D	F	H	W	Z	S
F	E	N	P	M	B
G	Y	I	O	D	T
X	U	K	L	R	A

This was the same polybius square as we used in the previous example, although we never used that assumption in our decryption. Also, the repeated text was

```
PANGALACTICGARGLEBLASTERS
```

which is 25 characters in length and hence produced 50 repeated characters in the ciphertext.

Δ

As usual, the computer can help out with the cryptanalysis. The finding of matching substrings can easily be automated and would save the analyzer a lot of work. Also, when attempting to find a substitution from the alphabetically ordered polybius square to the actual one used, the hill climbing method would produce a best guess very quickly, this is in fact what we did in the previous example where we needed to do three substitution keys. Furthermore, since we can automate the columnar cryptanalysis, we could combine it with monoalphabetic hill climbing to automate the entire process.

Another quick note about the example above. When we found the substring matches there were quite a few that were simply coincidences. Since our alphabet is reduced to 5 characters we tend to pick up more matches than we would with a larger alphabet.

If you did not have two ciphertexts with the same beginning string or if your substring analysis did not give you any hint to the length possibilities of the keyword you could do a brute force attack on the keyword in a similar manner as you would do a brute force attack on the keyword of the columnar cipher. The only difference is that when the decryption is made you would have the extra layer of finding a substitution key, either by hand or using hill climbing.

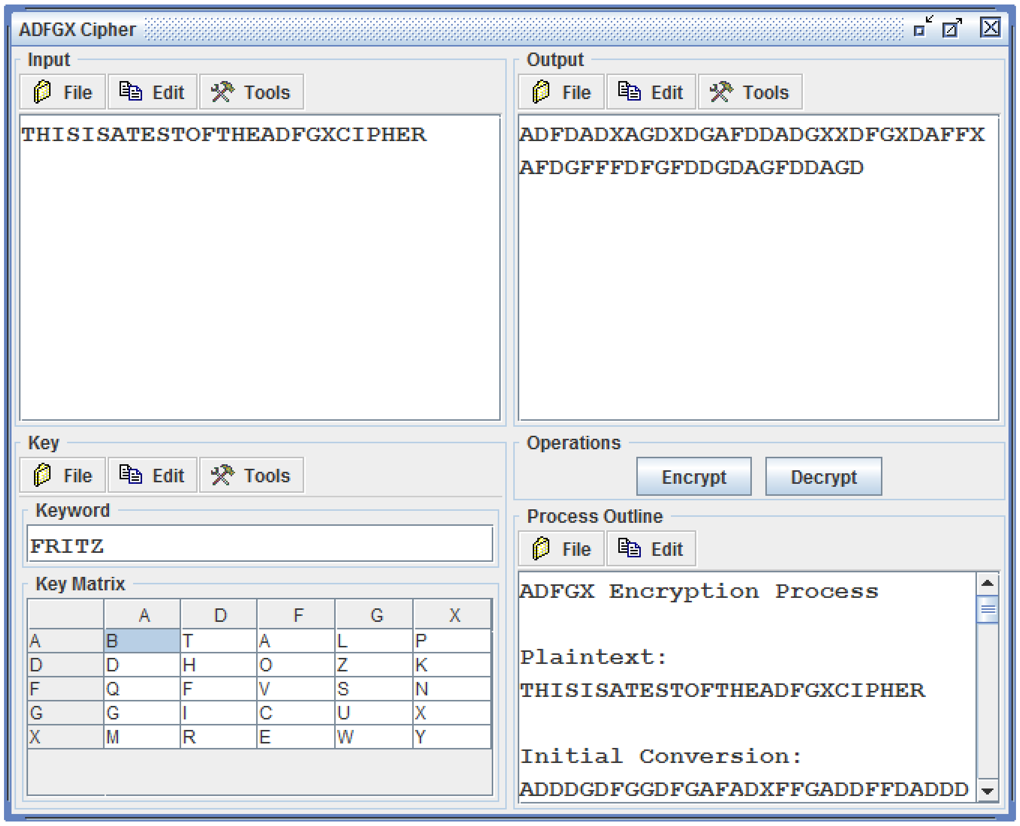
More specifically, start with a keyword with two letters, say AB, and take all possible permutations of the letters as possible keywords. So for length 2 words the possibilities would be AB and BA. Decrypt the message using each possible keyword and apply a hill climbing method to these or some other method for finding the best guess substitution key. If both of these fail, we move on to keywords of length three, there are 6 possible keywords of this length, ABC, ACB, BAC, BCA, CAB, and CBA. Again try each until a message appears. Failing this move on to 4 characters, and so on. As long as the keyword that was used is fairly small this should not take too long to solve, at least by computer. Since the number of letter permutations of a word of length n is $n!$, if the keyword is long then this will be difficult to do even with the machine.

2.10.3 Cryptography Explorer

The Cryptography Explorer program has a tool for the encryption and decryption using an ADFGX cipher and a tool for an ADFGVX. It also has a tool to do substring comparisons and monoalphabetic hill climbing to aid in the breaking of an ADFGX cipher.

ADFGX and ADFGVX Cipher Tools

The tools for encryption and decryption of the ADFGX and ADFGVX ciphers are identical in their options except that the key matrix for the ADFGVX is 6×6 whereas the key matrix for the ADFGX is 5×5, as expected. Both of these tools are under the Ciphers menu of the main menu.



```
THISISATESTOFTHEADFGXCIPHER
```

```
ADFDADXAGDXDGAFDDADGXXDFGXDAFFX
AFDGFFFDFGFDDGDAGFDDAGD
```

Keyword: FRITZ

	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

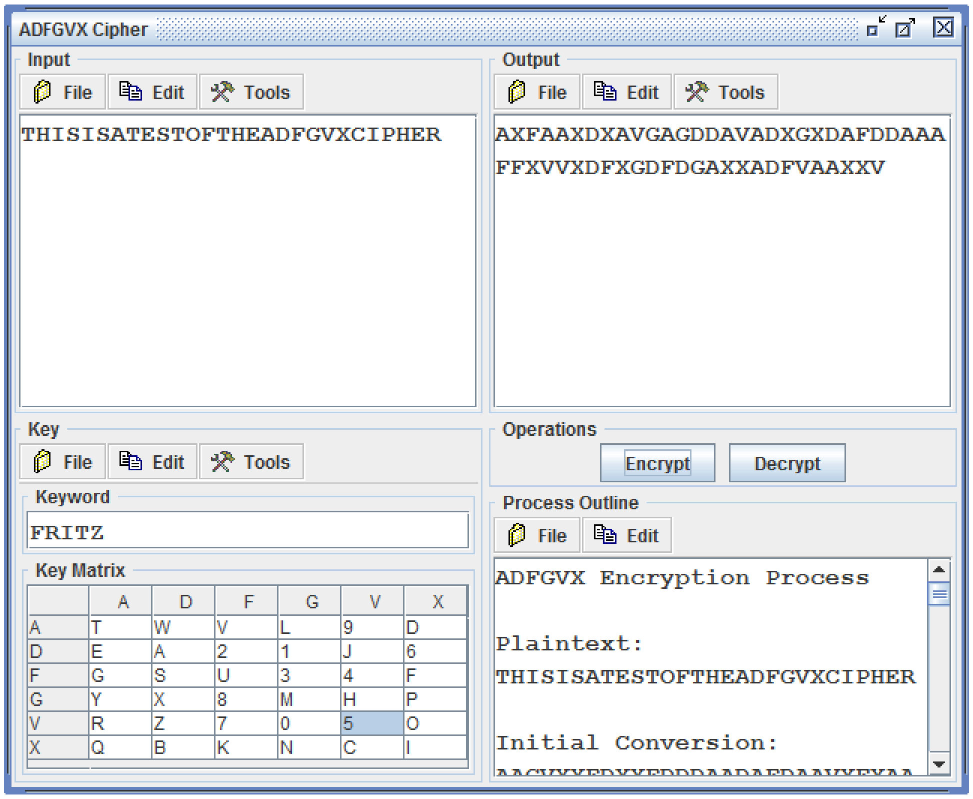
```
ADFGX Encryption Process


Plaintext:
THISISATESTOFTHEADFGXCIPHER

Initial Conversion:
ADDDGDFGGDFGAFADXFFGADDFFDADDD
```

Figure 2.60: 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. If you are running the ADFGX tool you need to also convert all of the J's 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.



```
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 2.61: ADFGVX Cipher Tool

3. Input a Key Matrix. For the ADFGX cipher tool, 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. For the ADFGVX cipher tool, the key matrix must consist of all uppercase letters and single digit numbers.
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 —

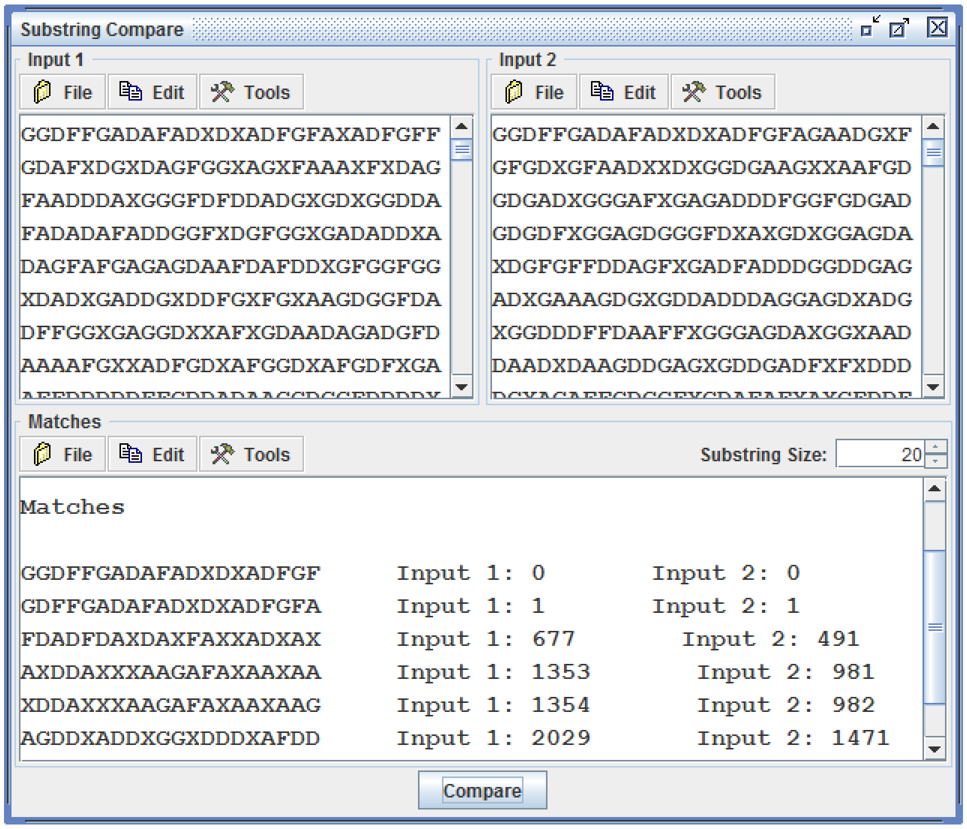
1. Input the ciphertext message into the Input box.
2. Input a Keyword. The keyword must consist of all uppercase letters with no duplications.
3. Input a Key Matrix. For the ADFGX cipher tool, the key matrix must consist of all uppercase letters excluding J, as with the playfair cipher we use the version of the ADFGX which equates I and J. For the ADFGVX cipher tool, the key matrix must consist of all uppercase letters and single digit numbers.
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.

Notes

- With the ADFGX tool, we equate I and J in order to get 26 letters down to 25, so the plaintext cannot have any J's in it.
- Since this cipher is a bit more involved, the tool has a process window in the lower right that will display the results of the main portions of the encryption ir decryption process. It will display the plaintext, the substituted plaintext, the table with the keyword at the top, the table with the sorted keyword, and the ciphertext.

Substring Comparison Tool

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.



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 2.62: Substring Comparison Tool

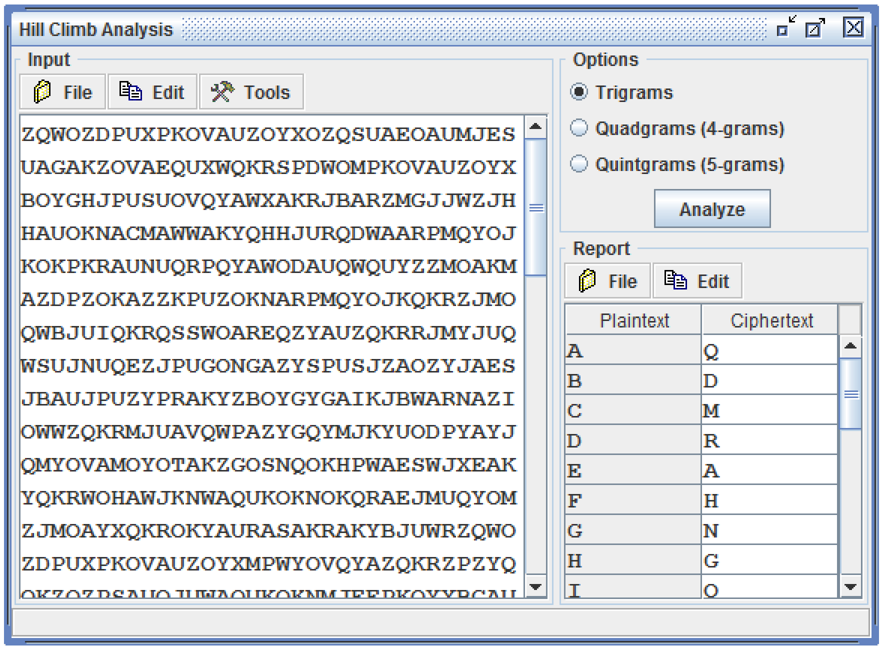
How to Use the Tool

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.

Hill Climbing Tool

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.



```
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 2.63: Hill Climb Analysis Tool

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.

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.

You can find a more detailed description of the options for these tools in the appendix on the Cryptography Explorer program.

One thing to note here is that if you use the hill climbing tool you will only be able to break an ADFGX cipher, since this tool does not include frequencies of single digits in English language literature. Even it it did this would be of little use in most cases. Adding the V was more of a change than just adding 11 characters to the alphabet.

Breaking an ADFGX cipher, even with the help of the Cryptography Explorer program, still requires a bit of work. The Cryptography Explorer program will remove the tedious work but you still need to know which tool to use when. If you go through the cryptanalysis example, you will see the following general method.

1. Use the substring comparison tool on two compatible ciphertexts to determine the length, or possible lengths, of the keyword and the lengths of the columns.

2. From the lengths of the columns and length of the keyword find the possible column permutations. Specifically, start with a keyword ABC. . . assigned to the sorted columns and then from the column length positions create all possible keywords.

3. With each of the keywords, use the ADFGX cipher tool with the keyword and the alphabetic square,

	A	D	F	G	X
A	A	B	C	D	E
D	F	G	H	I	K
F	L	M	N	O	P
G	Q	R	S	T	U
X	V	W	X	Y	Z

   on both ciphertexts to get a pair of "substituted" messages.

4. Combine the two substituted messages (to increase the number of letters for frequency analysis), and then do a hill climbing analysis to create the best guess of a substitution key.

5. Use the combined substituted messages and the best guess of a substitution key in a monoalphabetic cipher tool to decrypt the substituted messages. If an English message appears then you have the correct column arrangement and the correct substitution key. If not then pick the next possible keyword and repeat the process.

2.11 Linear Feedback Shift Register (LFSR)

2.11.1 Definitions and History

The Linear Feedback Shift Register (LFSR) cipher is a binary XOR cipher that simply XOR's the message, bit for bit, with the key. For this type of cipher the message and the key must both be binary streams of 0's and 1's. The XOR operation, which we will denote as $\oplus$, is 1 if the two bits are different and 0 if thee two bits are the same. Specifically,

$$\begin{aligned} 1 \oplus 1 &= 0 \\ 1 \oplus 0 &= 1 \\ 0 \oplus 1 &= 1 \\ 0 \oplus 0 &= 0 \end{aligned}$$

Another way to define the XOR operation is bit-wise addition modulo 2.

$$\begin{aligned} 1+1 &\equiv 0 \pmod{2} \\ 1+0 &\equiv 1 \pmod{2} \\ 0+1 &\equiv 1 \pmod{2} \\ 0+0 &\equiv 0 \pmod{2} \end{aligned}$$

The XOR type ciphers have a number of advantages. First, due to their binary nature, they can be implemented in computer hardware very easily, making their calculation extremely fast so even very large data streams can be processed quickly. Second, the XOR is its own inverse operation, so the same method that encrypts a message can also be used to decrypt the message.

Example 36: Given the message `1001010001010100100101001` and a binary key of the same length, `1100100100111010001101010`, we simply XOR each corresponding bit to obtain the ciphertext.

```
   Message:  1001010001010100100101001
       Key:  1100100100111010001101010
             -------------------------
Ciphertext:  0101110101101110101000011
```

Using the ciphertext and the same key, XORing these together will result in our original message.

```
Ciphertext:  0101110101101110101000011
       Key:  1100100100111010001101010
             -------------------------
   Message:  1001010001010100100101001
```

Δ

These types of ciphers require that the key be the same length, at least, as the message. There are a number of ways that one can generate a key of any given length. One possible method is to generate the bits at random, this will be discussed in the section about one-time-pads. We could also use formulas to generate a key, which is how the LFSR works.

Let x_i denote the i^{th} bit of the key, we could define x_i in terms of the bits before it, for example,

$$x_{i-5} + x_{i-2} + x_{i-1} = x_i$$

where the addition is done modulo 2. More specifically,

$$x_{i-5} + x_{i-2} + x_{i-1} \equiv x_i \pmod{2}$$

We will usually drop the mod 2 at the end since this is the only modulus that is implied and it is necessary. In the above formula, this would say that the i^{th} bit is the sum, modulo 2, of the bit that is one back plus the bit that is two back plus the bit that is 5 back. So if our key ends with 10010, the next bit is 0, giving 100100, then the next bit is 0, giving 1001000, the next bit is 0, giving 10010000, the next bit is 1, giving, 100100001, and so on. So using a starting string of 0's and 1's, called a seed, we can use this recurrence relation to generate a key of any length.

One thing to note is that some textbooks write this a little differently, they start the furthest bit at i and go up from there. So in this notation we would write the above example as,

$$x_i + x_{i+3} + x_{i+4} = x_{i+5}$$

where the addition is done modulo 2. More specifically,

$$x_i + x_{i+3} + x_{i+4} \equiv x_{i+5} \pmod{2}$$

The different notations are, obviously, equivalent.

The *length* of a recurrence relation is the number of bits, or in general terms, that we move back the sequence in order to calculate the next bit. For example, with the relation,

$$x_{i-5} + x_{i-2} + x_{i-1} = x_i$$

or equivalently,

$$x_i + x_{i+3} + x_{i+4} = x_{i+5}$$

we must move back 5 bits in order to get all the information we need to calculate the next bit, so the length of this relation is 5.

Example 37: Say we started with the seed `101010111` and that our recurrence relation is

$$x_{i-6} + x_{i-3} + x_{i-1} = x_i$$

Then the sequence of bits becomes,

```
101010111
1010101110
10101011100
101010111001
1010101110010
10101011100101
101010111001011
```

and so on. Δ

Another way to write these relations is by using a binary coefficient notation, for example,

$$a_{i-n}x_{i-n} + \cdots + a_{i-3}x_{i-3} + a_{i-2}x_{i-2} + a_{i-1}x_{i-1} = x_i$$

where each a_k is either 0 or 1, and as usual, the arithmetic is all done modulo 2. So the relation we used in the above example could be written as

$$1 \cdot x_{i-6} + 0 \cdot x_{i-5} + 0 \cdot x_{i-4} + 1 \cdot x_{i-3} + 0 \cdot x_{i-2} + 1 \cdot x_{i-1} = x_i$$

Specifically, $a_{i-6} = 1$, $a_{i-5} = 0$, $a_{i-4} = 0$, $a_{i-3} = 1$, $a_{i-2} = 0$, $a_{i-1} = 1$. We could also adopt a short-hand notation for this as, 100101. That is, simply write the coefficients in increasing subscript order. This is sometimes called a generator string.

Example 38: We will redo the last example with the new notation. We start with the seed `101010111` and our recurrence relation is

$$x_{i-6} + x_{i-3} + x_{i-1} = x_i$$

that is,

$$1 \cdot x_{i-6} + 0 \cdot x_{i-5} + 0 \cdot x_{i-4} + 1 \cdot x_{i-3} + 0 \cdot x_{i-2} + 1 \cdot x_{i-1} = x_i$$

which has the generator string `100101`. If we place the generator string at the end of the current key string, multiply each column, then add modulo 2, we get the next bit. This is precisely what would be done in our arithmetic formula.

```
101010111
   100101
   ------
   000101 = 0

1010101110
    100101
    ------
    100100 = 0

10101011100
     100101
     ------
     000100 = 1

101010111001
      100101
      ------
      100001 = 0

1010101110010
       100101
       ------
       100000 = 1
```

and so on. Δ

At this point you should be getting the idea of where the name for this cipher came from. The name is really how the key is generated and not the encryption method itself, which is just a bit-wise XOR. The next key bit is generated by a linear equation, fed back into the key and then the relation is shifted by one to repeat the process.

The above examples are just on the generation of the key for an LFSR cipher, we would then XOR this key with the message to obtain the cipher text, or vice-versa. Formally, we define the LFSR as follows.

Definition 23: *The Linear Feedback Shift Register (LFSR) has three components, the plaintext message, a key seed and a key generator.*

The message is any binary string. The key seed is a binary string that starts the key and the key generator is a binary string of coefficients to the linear recurrence relation that generates the key. The key seed must be at least as long as the key generator.

Use the key seed and the recurrence relation defined by the key generator to create a key that is as long as the message. Then XOR the key and message, bit by bit, to obtain the ciphertext. Decryption uses the same process.

Example 39: Say we wish to send the message, SEND HELP. Remove the spaces as usual to get SENDHELP. Now we need to convert this to binary, There are many ways this coding can be done, one method is to do our normal coding of A–Z to 0–25 but in this case we write the numbers 0–25 in binary instead of decimal. If we do that, we convert SENDHELP to 18 4 13 3 7 4 11 15, and then to 00010010 00000100 00001101 00000011 00000111 00000100 00001011 00001111. Removing all of the spaces produces our ciphertext of

```
0001001000000100000011010000001100000111000001000000101100001111
```

We will use the key seed of `101100111` and a key generator of `1001011`. Using these we expand the key to 64 bits to cover the message and then XOR with the pliantext.

```
PT:  0001001000000100000011010000001100000111000001000000101100001111
Key: 1011001111001111001111001111001111001111001111001111001111001111
```

```
     ----------------------------------------------------------------
CT:  1010000111001011001100011111000011001000000111000111110001100000
```

At this point we would send the ciphertext bit stream. With many cryptographic methods we usually like to put the ciphertext in the same form, that is, using the same alphabet, as the plaintext. If one considers the plaintext for the LFSR to be the binary string then both the plaintext and ciphertext use the same alphabet. On the other hand, if we consider SENDHELP to be the plaintext then we have different alphabets for the plaintext and the ciphertext. This is really not a problem as long as the recipient of the message knows that there was an initial coding of the message and what that coding was. In this example, we cannot use the coding method to convert the ciphertext back to English letters since the first 8 bits of the ciphertext are 10100001 which converts to 161, far outside the 0–25 range.
Δ

The LFSR, as presented here, is not a secure cryptographic method, like any other classical method. As we will see in the next section, if we can get enough of the key we can easily calculate the recurrence coefficients and hence reproduce all but the seed of the key. Since the seed is usually not very long, other methods can easily determine the beginning of the message. There are other cryptographic systems that are LFSR-like that offer more security but these usually involve nonlinear recurrence relations.

Although the basic LFSR is insecure as a cryptographic method, its binary form along with its properties does make it and similar methods very useful in numerous methods of data transmission and error detection in data transmissions.[5]

One minor tweak to the LFSR presented here will create the linear congruential algorithm for generating pseudo-random numbers.[70] The linear congruential algorithm, while not cryptographically sound, is the method used by nearly all random number generators found in calculators and programming language libraries. For example, the TI calculator line uses it for its random functions, it is used in the rand function in GCC's C++ compiler and in Java's random function in its Math library.

A cyclic redundancy check is another algorithm that is closely related to the LFSR system. This is used in numerous types of transmission and storage hardware. For example,

- SATA (probably your computer's hard drive), SCSI (hard drives and other types of peripheral devices), and USB 3.0 devices.[67]
- Scramblers for Ethernet devices like the 1000BASE-T (Gigabit Ethernet) and 100BASE-T2.[64]
- PCI Express 3.0, communication bus between a computer's motherboard and expansion cards like a graphics card.
- IEEE 802.11a, wireless scramblers.
- Bluetooth Low Energy Link Layer, short range digital communication.[32]
- Audio and video transmission of television and radio signals in North America, Europe, Australia, parts of Asia.

- Satellite and cellular telephone transmissions.

An LFSR system can also be used to generate pseudo-random noise in transmissions that are effective in communication jamming systems.[44] Also, since LFSR systems are very fast, they can be used to transmit very high-precision timing offsets. One application of this is in the Global Positioning System.[21]

2.11.2 Cryptanalysis

Cryptanalysis of an LFSR amounts to finding the recurrence relation that generates the key. To do so we need a sufficiently long segment of the key. We will see later what sufficiently long really means, but for now let's say we have enough bits of the key sequence. It turns out that we do not need the beginning of the key, we simply need a segment. One question would be how would you get a segment of the key? Having a crib of a segment of plaintext and its corresponding ciphertext would work since all we would need to do is XOR these two together and we have the key used in that segment. A known plaintext or known ciphertext attack would supply us with the needed crib. How we get the segment is not really the important part, but how we take the segment of the key and derive the relation is.

We will walk through an example. Say that we have obtained the following segment of an LFSR key.

```
011001000111101011001000111101011001000111101011001000111 1
```

Since the recurrence relation is a linear relation it would make sense to use techniques from linear algebra to solve the relation. We do not know what the length of the relation is but we do know that the length is not 1. A recurrence relation of length 1 would produce a constant sequence and this one is clearly not. So we can start with length 2. If the relation is length 2, then it would be of the form,

$$b_{i-2}x_{i-2} + b_{i-1}x_{i-1} = x_i$$

where b_{i-1} and b_{i-2} are either 0 or 1. If we let $i = 3$ and substitute the values of the key stream into the recurrence relation we would get,

$$b_{i-2} \cdot 0 + b_{i-1} \cdot 1 = 1$$

and if we let $i = 4$ and substitute the values of the key stream into the recurrence relation we would get,

$$b_{i-2} \cdot 1 + b_{i-1} \cdot 1 = 0$$

We can write this system of equations in matrix form as,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} b_{i-2} \\ b_{i-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

If we then create the augmented matrix for this system,

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

and reduce it modulo 2,

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

This gives the solution of $b_{i-2} = 1$ and $b_{i-1} = 1$, and hence the relation

$$x_{i-2} + x_{i-1} = x_i$$

Unfortunately, this does not work. It is fine for the start of the sequence, that is, the `011` and the `110`, but the next three bits `100` do not follow this relation. So a relation is not of length 2, so we try length 3.

$$b_{i-3}x_{i-3} + b_{i-2}x_{i-2} + b_{i-1}x_{i-1} = x_i$$

Doing the same as above with three equations we get the following system,

$$\begin{aligned} b_{i-3} \cdot 0 + b_{i-2} \cdot 1 + b_{i-1} \cdot 1 &= 0 \\ b_{i-3} \cdot 1 + b_{i-2} \cdot 1 + b_{i-1} \cdot 0 &= 0 \\ b_{i-3} \cdot 1 + b_{i-2} \cdot 0 + b_{i-1} \cdot 0 &= 1 \end{aligned}$$

In matrix form this is,

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_{i-3} \\ b_{i-2} \\ b_{i-1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

and in augmented form,

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

Reducing, modulo 2, gives,

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

This gives the solution of $b_{i-3} = 1$, $b_{i-2} = 1$ and $b_{i-1} = 1$, and hence the relation

$$x_{i-3} + x_{i-2} + x_{i-1} = x_i$$

This works for `0110`, `1100`, and `1001` but not for `0010`. So length 3 does not work so we move to length 4. Since we are probably getting the method by now we will jump to the augmented matrix,

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

Reducing, modulo 2, gives,

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

This gives the solution of $b_{i-4} = 1$, $b_{i-3} = 0$, $b_{i-2} = 0$ and $b_{i-1} = 1$, and hence the relation

$$x_{i-4} + x_{i-1} = x_i$$

Going back to the bit sequence you can see that this relation does in fact work. So for the key segment we have, this relation will generate it. There is always the possibility that we do not have enough of the key to compute the relation used for the ciphertext. Unfortunately, we cannot know this ahead of time. All we can do is derive the relation from what we have and then use that relation on the ciphertext and hope that we get a valid message. In the example we just did, the actual relation may have been of length 100 and not 4, we would not be able to tell this, nor solve it, since we do not have enough bits of the key.

This brings up the question of how many bits do we need? In other words, what is a sufficient number of key bits? Looking at our above example, when we assumed the length of the relation was 2 we needed the first 4 bits of the stream to create the augmented matrix. When we assumed that the relation length was 3 we needed the first 6 bits and when we assumed the relation was of length 4 we needed the first 8 bits. Obviously, this will continue in the same fashion, so to solve a relation of length n we would need $2n$ consecutive bits of the key. So in our above example, we had 58 bits so we could only have solved the relation if the relation had length 29 or less.

Let's bring out a little more linear algebra. Again, using the example above, since each of the following three augmented matrices,

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

had a unique solution, we know that their coefficient matrices,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

are invertible. Hence their determinants are all non-zero, but if we are doing all of our arithmetic modulo 2 the only non-zero number we have is 1, hence,

$$\begin{vmatrix} 0 & 1 \\ 1 & 1 \end{vmatrix} = 1 \quad \begin{vmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{vmatrix} = 1 \quad \begin{vmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{vmatrix} = 1$$

Let's consider the next larger coefficient matrix.

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

If we reduce this matrix, we get the following.

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

Since this did not reduce to the identity matrix, there will not be a unique solution to any system of equations that has this as its coefficient matrix. Hence any system of equations will either have more than one solution (note that I did not say infinitely many) or no solutions at all. This also implies that the determinant of the matrix is 0,

$$\begin{vmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix} = 0$$

Going up to the 6×6 matrix, we have,

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

When reduced we get,

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

Implying again that the determinant of the original matrix is 0.

$$\begin{vmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \end{vmatrix} = 0$$

Clearly we are trying to make a point here. This is not a coincidence, it is a theorem, once the length of the recurrence relation is exceeded the coefficient matrices will all have determinant 0.

Theorem 1: *Let $x_1, x_2, x_3, \dots$ be a sequence produced by an LFSR. For each $n \geq 1$, let*

$$M_n = \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n \\ x_2 & x_3 & x_4 & \cdots & x_{n+1} \\ x_3 & x_4 & x_5 & \cdots & x_{n+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_n & x_{n+1} & x_{n+2} & \cdots & x_{2n-1} \end{bmatrix}$$

Let N be the length of the shortest recurrence relation that generates $x_1, x_2, x_3, \dots$. Then $|M_N| = 1$ and $|M_n| = 0$ for all $n > N$. All calculations are done modulo 2.

Note that this says $|M_N| = 1$ and not $|M_n| = 1$ for all $n \leq N$. It is quite possible that some of the coefficient matrices that are smaller than $N \times N$ will have zero determinants, but we are guaranteed that $|M_N| = 1$ and all larger matrices have determinant 0.

Proof: We will first take care of the case where $n > N$. Since N is the length of the shortest recurrence relation that generates $x_1, x_2, x_3, \dots$, we know that

$$a_1x_1 + \cdots + a_{N-2}x_{N-2} + a_{N-1}x_{N-1} + a_Nx_N = x_{N+1}$$

for $a_i \in \{0, 1\}$. This implies that the $N+1^{st}$ row of M_n is a linear combination of the first N rows. Hence using row reductions on M_n we can generate a zero row (specifically, row $N+1$). This reduced matrix will have determinant 0 and since $|M_n|$ is equal to the determinant of the reduced matrix, $|M_n| = 0$.

Now suppose, by way of contradiction, that $|M_N| = 0$. This means that the null-space of M_N contains a non-zero vector. Thus there is a row vector $\mathbf{v} = (v_1, v_2, \dots, v_N)$ with $\mathbf{v}M_N = \mathbf{0}$. Let,

$$M_{(i)} = \begin{bmatrix} x_{i+1} & x_{i+2} & x_{i+3} & \cdots & x_{i+N} \\ x_{i+2} & x_{i+3} & x_{i+4} & \cdots & x_{i+N+1} \\ x_{i+3} & x_{i+4} & x_{i+5} & \cdots & x_{i+N+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{i+N} & x_{i+N+1} & x_{i+N+2} & \cdots & x_{i+2N-1} \end{bmatrix}$$

Since the a_k values in our recurrence relation are constant we have for all $n \geq N$,

$$a_1x_{n-N+1} + \cdots + a_{N-2}x_{n-2} + a_{N-1}x_{n-1} + a_Nx_n = x_{n+1}$$

and thus,

$$M_{(i)} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} x_{i+1} & x_{i+2} & x_{i+3} & \cdots & x_{i+N} \\ x_{i+2} & x_{i+3} & x_{i+4} & \cdots & x_{i+N+1} \\ x_{i+3} & x_{i+4} & x_{i+5} & \cdots & x_{i+N+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{i+N} & x_{i+N+1} & x_{i+N+2} & \cdots & x_{i+2N-1} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} x_{i+N+1} \\ x_{i+N+2} \\ x_{i+N+3} \\ \vdots \\ x_{i+2N} \end{bmatrix}$$

We will now verify that $\mathbf{v}M_{(i)} = \mathbf{0}$, for all $i \geq 0$. We will do this by induction. If we let $i = 0$ then $\mathbf{v}M_{(0)} = \mathbf{v}M_N = \mathbf{0}$ by our selection of $\mathbf{v}$. Now assume that $\mathbf{v}M_{(i)} = \mathbf{0}$ for some i and we will show that $\mathbf{v}M_{(i+1)} = \mathbf{0}$, which will finish our inductive argument.

$$\mathbf{v} \begin{bmatrix} x_{i+N+1} \\ x_{i+N+2} \\ x_{i+N+3} \\ \vdots \\ x_{i+2N} \end{bmatrix} = \mathbf{v}M_{(i)} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_N \end{bmatrix} = \mathbf{0} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_N \end{bmatrix} = 0$$

The vector,

$$\begin{bmatrix} x_{i+N+1} \\ x_{i+N+2} \\ x_{i+N+3} \\ \vdots \\ x_{i+2N} \end{bmatrix}$$

is the last column of the matrix $M_{(i+1)}$. The other columns of $M_{(i+1)}$ are identical to columns of $M_{(i)}$, so $\mathbf{v}M_{(i+1)} = \mathbf{0}$. Since $\mathbf{v}M_{(i)} = \mathbf{0}$ for all $i \geq 0$, the components of $\mathbf{v}$ give us the coefficients of a recurrence relation that generates the sequence $x_1, x_2, x_3, \ldots$. Specifically,

$$v_1 x_{i+1} + v_2 x_{i+2} + v_3 x_{i+3} + \cdots + v_N x_{i+N} = 0$$

for all i. Since $\mathbf{v} \neq \mathbf{0}$, there is a non-zero entry in $\mathbf{v}$, let m be the largest component of $\mathbf{v}$ that is not 0. Thus, $v_m = 1$ and $v_k = 0$ for all $k > m$. Then our above relation becomes,

$$v_1 x_{i+1} + v_2 x_{i+2} + v_3 x_{i+3} + \cdots + x_{i+m} = 0$$

and hence,

$$v_1 x_{i+1} + v_2 x_{i+2} + v_3 x_{i+3} + \cdots + v_{m-1} x_{i+m-1} = x_{i+m}$$

but this is a recurrence relation that generates the sequence $x_1, x_2, x_3, \ldots$, with length $m - 1 < N$. Since we assumed that the smallest such relation had length N we arrive at our contradiction. Therefore, $|M_N| = 1$. $\square$

What is nice about this theorem is that its conclusion produces an algorithm for finding the LFSR key generation recurrence relation if we have enough bits in the key.

1. Take the key bit stream and create matrices M_n for $n = 2, 3, 4, \ldots$. Recall that if the relation had length one it would be a constant sequence.

2. Find $|M_n|$ for each n.
3. Once there is $|M_n| = 0$ for several consecutive values of n, the last value of n that produced $|M_n| = 1$ is probably the length of the key.
4. Solve the system,
$$M_n \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} x_{n+1} \\ x_{n+2} \\ x_{n+3} \\ \vdots \\ x_{2n} \end{bmatrix}$$
with augmented matrix,
$$\left[\begin{array}{ccccc|c} x_1 & x_2 & x_3 & \cdots & x_n & x_{n+1} \\ x_2 & x_3 & x_4 & \cdots & x_{n+1} & x_{n+2} \\ x_3 & x_4 & x_5 & \cdots & x_{n+2} & x_{n+3} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n & x_{n+1} & x_{n+2} & \cdots & x_{2n-1} & x_{2n} \end{array}\right]$$
to obtain the coefficients of the recurrence relation.

A couple notes to end this section. As you probably already know from a linear algebra class, the determinant calculation is a very computationally costly operation. In fact, one of the more efficient ways to calculate the determinant of a general square matrix is to reduce the matrix, keeping track of row interchanges and row multipliers, then calculating the product of the main diagonal along with the inverses of the multipliers and $(-1)^t$, where t was the number of row interchanges. Since we are working modulo 2, there are no row multipliers and $-1 = 1$ modulo 2, so the determinant calculation is essentially a matrix reduction. Therefore, in our algorithm, we could eliminate the determinant calculations and simply solve the systems,
$$M_n \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} x_{n+1} \\ x_{n+2} \\ x_{n+3} \\ \vdots \\ x_{2n} \end{bmatrix}$$
for increasing n until it was evident that we had the relation coefficients.

Even if the bit stream was taken from the middle of the key we can generate the beginning of the stream as well as the end of the stream, with the exception of the seed, which does not need to conform to the recurrence relation. All we need to do to generate the beginning of the bit stream is to solve the recurrence relation for the smallest subscript. For example, say we had the recurrence relation
$$x_{i-6} + x_{i-3} + x_{i-1} = x_i$$

we can rewrite this as,

$$x_{i-6} = x_{i-3} + x_{i-1} + x_i$$

and then by simply shifting the subscripts we get,

$$x_i = x_{i+3} + x_{i+5} + x_{i+6}$$

which will generate lower subscript bits from the higher subscript bits.

With a deterministic feedback function, like a recurrence relation, the key sequence will repeat at some point, the length of the repetition is called the period of the key. Specifically, if the recurrence relation has length n then there are $2^n - 1$ non-zero strings, we discount the zero stream since this would produce a key that would not alter the message. So the maximum period of a key string using a recurrence relation of length n is $2^n - 1$. In general you would like to use a relation that will produce a key stream with the longest period. One way to do this is to examine the associated polynomial of a recurrence relation. A recurrence relation has an associated polynomial which is constructed as follows. Say we have a recurrence relation of length n,

$$c_0 x_{i-n} + c_1 x_{i-n+1} + c_2 x_{i-n+2} + \cdots + c_{n-1} x_{i-1} = x_i$$

where $c_k \in \{0, 1\}$. Let $i = n$, to get,

$$c_0 x_0 + c_1 x_1 + c_2 x_2 + \cdots + c_{n-1} x_{n-1} = x_n$$

Make the subscripts on the x's into exponents and bring the x^n on the other side of the equal sign.

$$f(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1} + x^n$$

We will not go into all the relationships between the recurrence relation and its associated polynomial here, but we will mention a few properties, without proof. First, if $f(x)$ is irreducible modulo 2, meaning that $f(x)$ does not factor into a product of non-constant polynomials modulo 2, then the period of the bit stream is a divisor of $2^n - 1$. If $f(x)$ is reducible modulo 2 then this may not be the case. One corollary of this is that if $2^n - 1$ is a prime number, prime numbers of this form are called Mersenne primes, and $f(x)$ is irreducible modulo 2, then if the bit stream is not constant then it must have a period of exactly $2^n - 1$, the maximum it can have.

Example 40: Consider the recurrence relation,

$$x_{i-6} + x_{i-5} = x_i$$

Letting $i = 6$ we have

$$x_0 + x_1 = x_6$$

which creates the associated polynomial,

$$f(x) = x^6 + x + 1$$

This polynomial is irreducible modulo 2, and if we use the seed `10010101001001` (randomly generated), we get the following key sequence, note that the following bit stream was taken from a section outside the seed.

```
111111000001000011000101001111010001110010010
110111011001101010111111000001000011000101001
111010001110010010110111011001101010111111000
001000011000101001111010001110010010110111011
001101010111111000001000011000101001111010001
```

Shifting the sequence shows that the period of the stream is $63 = 2^6 - 1$.

```
1111110000010000110001010011110100011100100101101110110011010101111110000001
␣␣␣␣␣␣␣␣␣|␣␣␣␣␣␣␣␣␣|␣␣␣␣␣␣␣␣␣|␣␣␣␣␣␣␣␣␣|␣␣␣␣␣␣␣␣␣|␣␣␣␣␣␣␣␣␣|␣␣␣111111000001
```

Since 63 is not prime we were not guaranteed that the period would be maximal, but in this case it is.

Now consider the recurrence relation,

$$x_{i-6} + x_{i-3} + x_{i-1} = x_i$$

Letting $i = 6$ again gives

$$x_0 + x_3 + x_5 = x_6$$

and thus the associated polynomial is,

$$f(x) = x^6 + x^5 + x^3 + 1$$

In this case the polynomial is not irreducible modulo 2,

$$f(x) = x^6 + x^5 + x^3 + 1 \equiv (x+1)^3(x^3 + x + 1) \pmod 2$$

If we use the same seed as before `10010101001001`, we get the following key sequence, note that the following bit stream was again taken from a section outside the seed.

```
111101100001001111011000010011110110000100111
101100001001111011000010011110110000100111101
100001001111011000010011110110000100111101100
001001111011000010011110110000100111101100001
001111011000010011110110000100111101100001001
```

Shifting the sequence shows that the period of the stream is 14, which is not a divisor of 63.

```
1111011000010011110110000100111101100001001111011000010011110110000100111 10
␣␣␣␣␣␣␣␣␣|␣␣␣␣11110110000100111101100001001111011000010011110110000100111 10
```

Δ

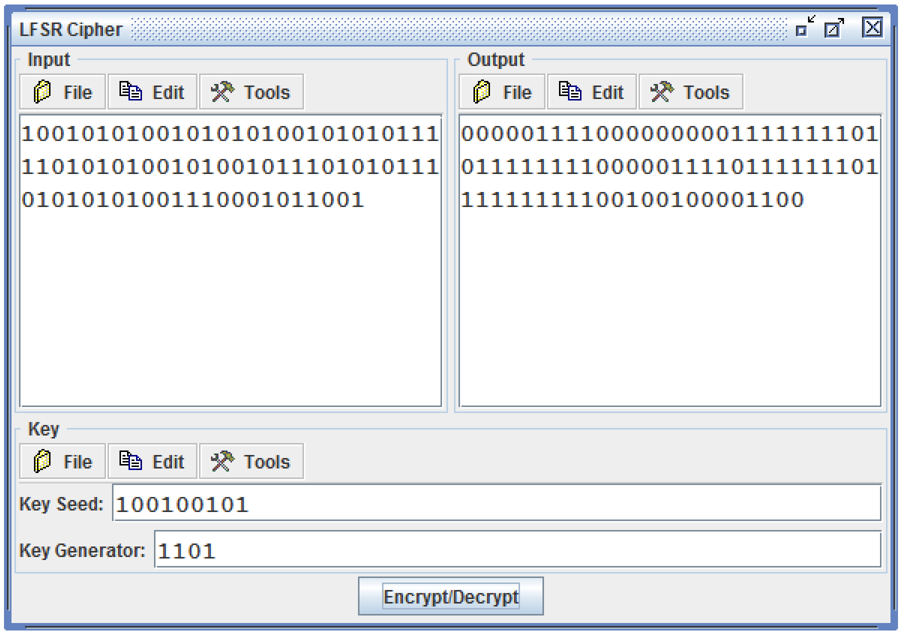
One final note, all of the above work was with an LFSR, a linear recurrence relation. If the recurrence relation is not linear, such as,

$$x_{i-6} + x_{i-5}x_{i-3} + x_{i-2} = x_i$$

then our methods of cryptanalysis do not work. There are methods for analyzing non-linear recurrence relations but we will not go into them here, that would be getting too far off topic. If we were to use a non-linear relation for key generation, the system would be called a Feedback Shift Register (FSR).

2.11.3 Cryptography Explorer

Linear Feedback Shift Register Tool



```
1001010100101010100101010111
1101010100101001011101010111
01010101001110001011001
```

```
0000011110000000001111111101
0111111110000011110111111101
11111111100100100001100
```

Key Seed: 100100101

Key Generator: 1101

Figure 2.64: Linear Feedback Shift Register Cipher 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.

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... In recurrence relation notation, the generator 101 means the relation,

$$x_{i-3} + x_{i-1} = x_i$$

equivalently,

$$x_0 + x_2 = x_3$$

and the generator 100101 means the relation,

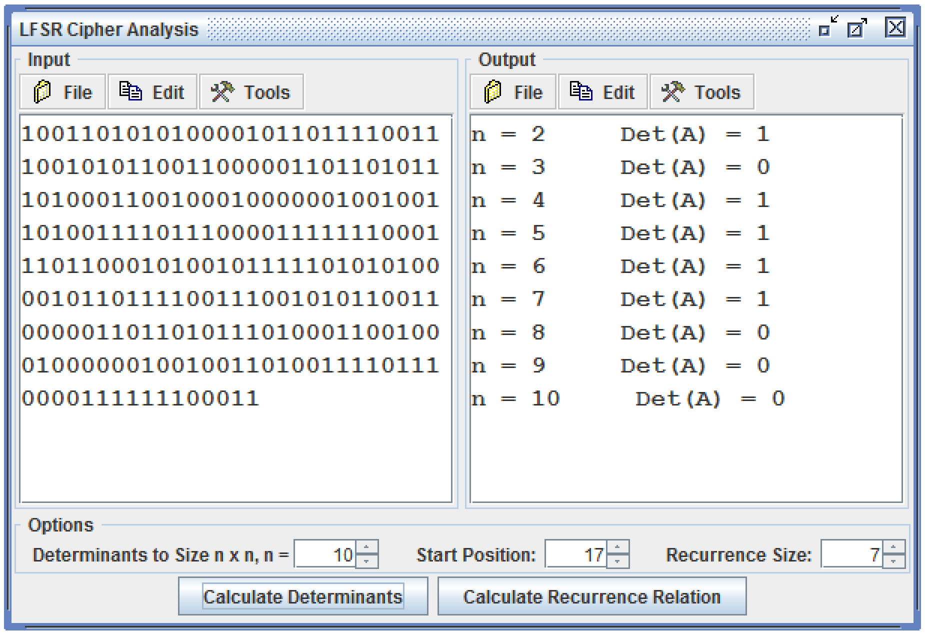
$$x_{i-6} + x_{i-3} + x_{i-1} = x_i$$

equivalently,

$$x_0 + x_3 + x_5 = x_6$$

LFSR Cipher Analysis Tool

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 bit stream and for modulo 2 reduction of a resultant matrix of specified size.

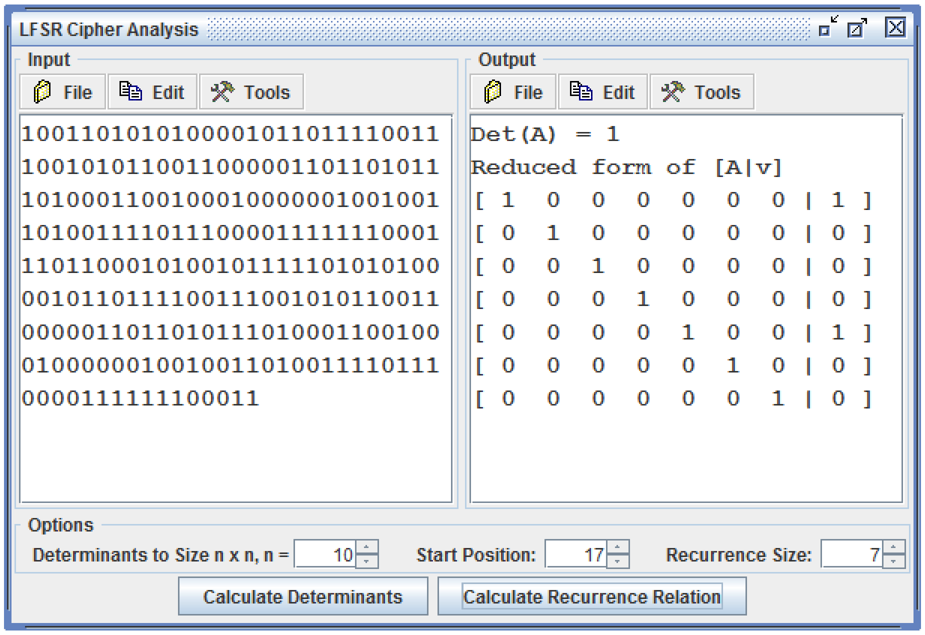


```
1001101010100001011011110011
1001010110011000001101101011
1010001100100010000001001001
1010011110111000011111110001
1101100010100101111101010100
0010110111100111001010110011
0000011011010111010001100100
0100000010010011010011110111
0000111111100011
```

```
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 2.65: 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 2.66: LFSR Analysis Tool: Relation

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 2×2 to $n \times n$, where n is the maximum size that was selected.

Determining the Relation Generator String —

1. After the above analysis, input the recurrence size.
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.

Example 41: Let's say that we have the following bit stream of an LFSR key.

```
111101100001001111011000010011110110000100111
101100001001111011000010011110110000100111101
```

```
10000100111101100001001111011000010011110l100
00100111101100001001111011000010011110l100001
00111101100001001111011000010011110l100001001
```

This is the same period 14 stream that we saw in the previous example. If er load this into the Input box of the analysis tool, set the Determinants to Size n X n to 10 and click on the Calculate Determinants button we would get e following output.

```
n = 2      Det(A) = 0
n = 3      Det(A) = 1
n = 4      Det(A) = 0
n = 5      Det(A) = 1
n = 6      Det(A) = 0
n = 7      Det(A) = 0
n = 8      Det(A) = 0
n = 9      Det(A) = 0
n = 10      Det(A) = 0
```

From the theorem this would say that the length of the relation is 5. If you look back to the example that generated this sequence of bits you will see that the recurrence relation was length 6. This is not a mistake, as may be believed. Just because a relation generates a bit stream does not mean that another relation, which is possibly shorted, can also generate the same bit stream.

Assuming that the length of the relation is 5, we would set the Recurrence Size box to 5 and click the Calculate Recurrence Relation button. Doing so will produce the following output.

```
Det(A) = 1
Reduced form of [A|v]
[ 1  0  0  0  0 | 1 ]
[ 0  1  0  0  0 | 1 ]
[ 0  0  1  0  0 | 1 ]
[ 0  0  0  1  0 | 0 ]
[ 0  0  0  0  1 | 0 ]
```

This would say that the relation, in our binary notation, is `11100`, which could be translated as,

$$x_{i-5} + x_{i-4} + x_{i-3} = x_i$$

If you go back to the LFSR cipher tool, use `11100` as the key generator, `10010101001001` as the key seed, put a string of 100 to 150 0's into the input box and click the Encrypt/Decrypt button you will get the same key as we did using `100101`, which is the way to write the relation.

$$x_{i-6} + x_{i-3} + x_{i-1} = x_i$$

Δ

2.11.4 Mathematica

Although most of what you do with the LFSR will be easier to do with Cryptography Explorer, if you want to factor the associated polynomial modulo 2, in order to gather information about the period of the recurrence relation you will need to use a computer algebra system like Mathematica or Maxima.

In Mathematica, the command to factor a polynomial is `Factor`. Note in the example below, the first input and output is the factorization of $x^6 + x^5 + x^3 + 1$ using integer coefficients, this is not a modulus 2 factorization. If you are familiar with factorization modulo 2 you probably noticed this, since modulo 2, $x^2+1 = (x+1)^2$. To get Mathematica to factor modulo a prime you need to include the the `Modulus` option at the end of the factor command, as we did with the second input.

```
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)
```

2.11.5 Maxima

Although most of what you do with the LFSR will be easier to do with Cryptography Explorer, if you want to factor the associated polynomial modulo 2, in order to gather information about the period of the recurrence relation you will need to use a computer algebra system like Mathematica or Maxima.

In Maxima, the command to factor a polynomial is `factor`. Note in the example below, the first input and output is the factorization of $x^6 + x^5 + x^3 + 1$ using integer coefficients, this is not a modulus 2 factorization. If you are familiar with factorization modulo 2 you probably noticed this, since modulo 2, $x^2+1 = (x+1)^2$. To get Maxima to factor modulo a prime you need to set the `modulus` option to that prime, as we did with the second input. Now when you invoke the factor command you will get a modulo 2 factorization.

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

Note that when you are done and want to go back to non-modulus calculations simply set the `modulus` option to `false`.

2.12 One Time Pad

2.12.1 Definitions and History

Definition 24: *The One Time Pad is an encryption technique that exhibits perfect secrecy. The key to a one time pad is completely random, and when used to encrypt a message it totally masks all information about the message, except possibly the maximum length of the message, and any information about the underlying language of the message. Furthermore, one time pad keys are used only once and are destroyed immediately after they are used.*

Looking at the above definition you may be wondering why we are studying cryptography? Why does the field of cryptography even exist? This one time pad seems to be the holy grail of secret writing, so why are we even considering other techniques, just use this one time pad and be done with it. As you probably guessed, there has got to be some difficulty with this technique, in fact there are several. You may also notice that the definition we gave above is rather vague, it does not suggest an algorithm nor does it give any implementation information. This would lead you to believe that there are either multiple one time pad methods or that an implementation does not exist. There are, in fact, several ways that a one time pad can be implemented and we will discuss a couple here. In general, if any of the methods are misused then the property of perfect secrecy is lost.

Definition 25: *Perfect secrecy is the notion that, given an encrypted message from a perfectly secure encryption system, absolutely nothing will be revealed about the unencrypted message by the ciphertext.*

First let's discuss the history of the one time pad and some of its difficulties. The first published account of a one time pad cipher was by Frank Miller in 1882.[35] Frank Miller was a New York banker who was interested in sending secure telegrams about transactions from one bank to another. His method was very simple and it was essentially a Caesar shift cipher but it was not a shift of the alphabet, it was a shift of words or phrases with the addition of a code word. In his 1882 publication, Mr. Miller compiled a list of 12,300 words and phrases and 1,700 cipher-words, making a total of 14,000 code words. These phrases included locations, New York bank names and addresses, numbers into the millions (no billionaires in 1882), and dates.

Each entry in this massive table consisted of a plaintext column, numeric assignment, and a ciphertext column, for example, two entries from the table are,

Plaintext	**Number**	**Ciphertext**
Extended	04651	Foredated
Gauged	05134	Gentleness

In the one and a half page preface of Mr. Miller's book, he describes the method and the rest of the book is the table of 14,000 words and numbers. The way his cipher worked was that a large set of "random" shift numbers were compiled by one banker and this list was delivered, securely, to another bank or banks. When one banker wanted to send a message to another, they would take the plaintext message, look up each word or phrase in Miller's table from the plaintext column, then write down the corresponding number. Then the banker would use the list of shift numbers and add the shift number to the word number to get another corresponding number. Then they would look up the new number in Miller's table and use the ciphertext word associated with it. For example, say that the plaintext word was Extended and that the shift number was 483. The encoder would start with the word Extended, its number is 04651, then they would add 483 to it to get 05134, then they would look up 05134 and use the associated ciphertext word of Gentleness. In all, Extended was encrypted to Gentleness. The example that Mr. Miller gives in his book is the following.

Example 42: The plaintext message will be "Extended for eight days" and the shift numbers will be 483, 281, 175, and 892.

Extended	for	eight	days
4651	4942	226	3271
483	281	175	892
5134	5223	401	4163
Gentleness	Glaired	Allegro	Fantasia

So the ciphertext would be "Gentleness Glaired Allegro Fantasia". Δ

Decryption is done by simply reversing the process and subtracting the shift numbers instead of adding them.

Example 43: So the ciphertext would be "Gentleness Glaired Allegro Fantasia" and the shift numbers are still 483, 281, 175, and 892.

Gentleness	Glaired	Allegro	Fantasia
5134	5223	401	4163
483	281	175	892
4651	4942	226	3271
Extended	for	eight	days

So the plaintext message will be "Extended for eight days". Δ

Miller goes on to state that,

> If the sender finds that the addition of a key produces a sum greater than the highest "serial number" (14000) in this book, he must deduct said last "serial number" from said sum and count the excess from the first page.
>
> On the other hand, if the receiver finds that the "serial number" of a cipher-word is less than the key which is to unlock it, he must temporarily add to said serial number the highest number in this book and deduct the key from the sum.

He referred to the associated numbers as serial numbers. He is also describing what we now call modular arithmetic (modulo 14000 in his case). He also states,

> Any system which allows a cipher word to be used twice with the same signification is open to detection. ... The lists of "shift-numbers" should be kept by one person in each bank, and from him one or more of the "shift-numbers" may be obtained by such clerks as receive or send telegrams. ... The sender and receiver must each cancel shift-numbers as fast as they are used. ... When a shift-number has been applied, or used, it must be erased from the list and not used again.

which makes the technique a one time pad. He also points out that the shift numbers must be "irregular".

The one time pad was reinvented between 1917 and the early 1920's by Gilbert S. Vernam and Joseph Mauborgne. The one time pad is sometimes referred to as the Vernam cipher or the Vernam-Mauborgne cipher. Vernam patented an electronic cipher machine on July 22, 1919.

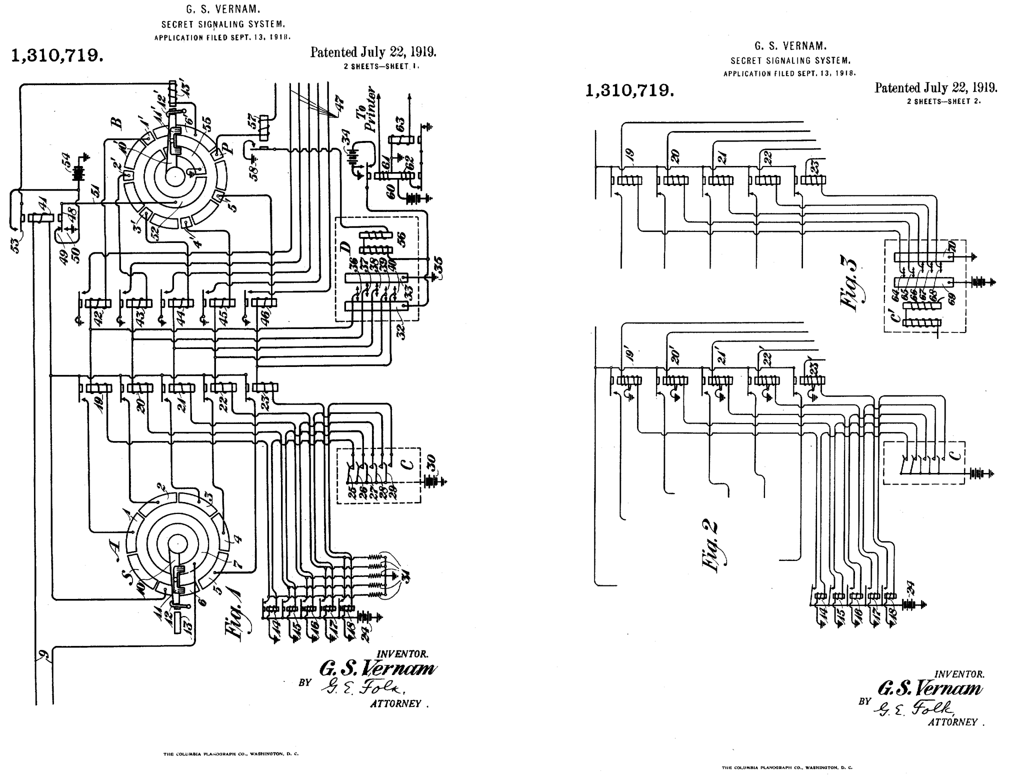


Figure 2.67: Vernam Patent 1,310,719

The machine itself was not a one time pad, but an electronic XORing machine that could XOR the plaintext character with a key character to produce a ciphertext character, in the same way that we use the XOR function as part of the LFSR cipher. Melville Klein of the NSA called this patent "perhaps one of the most important in the history of cryptography."[30] Shortly after the machine was patented, Joseph Mauborgne, a Captain in the US Army Signal Corps, proposed that a paper tape that contained the random information for the key be added to the machine, making it the first automatic one time pad device.

In 1949, Claude Shannon proved that the one-time pad is unbreakable.[54] He also proved that any unbreakable system must have essentially the same characteristics as the one-time pad,

1. The key must be truly random.

2. The key must be at least as large as the plaintext.

3. The key must never be reused in whole or part.

4. The key must be kept secret.

There are several difficulties with implementing a one time pad. Although Shannon proved that the one time pad was unbreakable, he used a theoretical mathematical model to do so. In that model, and his stipulations above, he assumed that the key was “truly random”. This is harder than it may seem. Your computer programs and calculators have the ability to generate random numbers, but none of them are truly random, they are called pseudorandom numbers and in fact they are not random at all, they are completely deterministic. A pseudorandom number sequence is a sequence of numbers that have some of the properties of a truly random number sequence, such as, statistical randomness. In statistical randomness, would stipulate, among other things, that the distribution of the digits be uniform, as well as pairs, triples, quadruples, . . . of digits. Most random (that is pseudorandom) number sequences that are generated by a computer program or by a calculator use the linear congruential algorithm. This algorithm produces a sequence of numbers that are good enough for most simulations that require a random sequence of numbers but it falls short of being usable for cryptographic purposes. A truly random sequence of numbers cannot be produced by a computer, at least not by what we currently call a computer. They can be produced from physical phenomena, such as radioactive decay or quantum spins. Both of these methods are too costly to use in practice.

Another major problem is in the exchange of the key. The security of the one time pad is only as secure as the security of the one time pad exchange. This is called the Key Distribution Problem. We will discuss the Key Distribution Problem further later in the section on public key methods. If we send the one time pad key using another cryptographic method then the security of the one time pad is reduced to the security of the other cryptographic method.

Finally, as with any cryptographic method, the misuse of the method seriously jeopardizes the security of the system. For the one time pad, using the key more than once makes the messages vulnerable. Historically, when keys were reused then one time pads were broken.

Some historical uses of the one time pad are below,

1. The hotline between Moscow and Washington D.C., established in 1963 after the Cuban missile crisis, used teleprinters protected by a commercial one time tape system.

2. The World War II voice scrambler SIGSALY used a form of a one time system.

3. The NSA describes one-time tape systems like SIGTOT and 5-UCO as being used for intelligence traffic until the introduction of the electronic cipher based KW-26 in 1957.[30]
4. The Weimar Republic Diplomatic Service began using the one time pad method in about 1920.
5. The U.S.S.R. adopted one time pads for some purposes by around 1930, and KGB spies are also known to have used pencil and paper one time pads more recently.
6. The British Special Operations Executive used one-time pads in World War II.
7. British one time tape cipher machines included the Rockex and Noreen.
8. The German Stasi Sprach Machine was also capable of using one time tape which East Germany, Russia, and Cuba used to send encrypted messages to their agents.[68]

Two examples of misuse that led to the breaking of one time pad systems are,

1. In World War II, the U.S. Army's Signals Intelligence Service broke a one time pad system used by the German Foreign Office, codenamed GEE. GEE was insecure because the pads were not completely random, that is, the machine used to generate the pads produced predictable output.[13]
2. In 1945, the US discovered that Canberra-Moscow messages were being encrypted first using a code-book and then using a one time pad. However, the one time pad used was the same one used by Moscow for Washington, DC-Moscow messages.

We will look at two more methods for creating one time pad ciphers. One was already discussed as being used by Miller in 1882. We will use a Vigenère Cipher as a base method for one and the LFSR Cipher for the other, very similar to the Vernam-Mauborgne machine.

Using the Vigenère Cipher, we can create a one time pad by creating a random key that is as long as the message.

Example 44: Let's encrypt the message "Nobody expects the Spanish Inquisition"[7] using a one time pad form of the Vigenère Cipher. Removing spaces and converting to uppercase gives us

```
NOBODYEXPECTSTHESPANISHINQUISITION
```

The plaintext is 34 characters in length, so we only need a key that has 34 random characters. As we discussed above, creating truly random numbers cannot be done by a computer, but there are deterministic methods for generating pseudorandom numbers that are cryptographically sound. The Blum-Blum-Shub algorithm is one of these.

The Blum-Blum-Shub Algorithm is for the generation of random bits. It is not for generating random characters. One way we could use the Blum-Blum-Shub algorithm is to generate a string of bits, break the string into blocks of 5 bits, convert each block of 5 into a decimal number, remove any that are larger than 25, convert each 0–25 decimal number to A–Z. Using the Blum-Blum-Shub algorithm we generated a list of 1000 bits.

```
01010110010101111001001011111100011011111101110011
10101001011101000000111110100111001101111010111100
10011010101110110000110110010010110011010110010101
01101110000111010001100011001000110000010011111101
11110001111000000001010001110001100000010011000011 1
00001101100101001111001110000100100110100001001100
00011111010101111011101101011110010100101110111011
10001001101000010100100000110110110111100001111110
01000001010100101010001101001111010000111110000001
01101011110000110011000110011110101011001101001101
11110110100100000111100010000010101101100101111110
10101101100001000101110000100110010001101100000100
01001001100100110111100001000101100100101100111110
11010100001001000111010000000100011001011010001110 0
10000110010000111110101011011000111101110011010000
01111001111011010101010001110111110011010101000010
01001100110100011101010010100110000000111110101111
01011000101110010100011110001110011110010000011000
00001110000111001010000000110100001111001010100000
10100010011100000110011101010111011001001001010010
```

Breaking this stream into blocks of 5 bits produces,

```
01010 11001 01011 11001 00101 11111 00011 01111 11011 10011 10101
00101 11010 00000 11111 01001 11001 10111 10101 11100 10011 01010
11101 10000 11011 00100 10110 01101 01100 10101 01101 11000 01110
10001 10001 10010 00110 00001 00111 11101 11110 00111 10000 00010
10001 11000 11000 00010 01100 00111 00001 10110 01010 01111 00111
00001 00100 11010 00010 01100 00011 11101 01011 11011 10110 10111
10010 10010 11101 11011 10001 00110 10000 10100 10000 01101 10110
11110 00011 11110 01000 00101 01001 01010 00110 10011 11010 00011
11100 00001 01101 01111 00001 10011 00011 00111 10101 01100 11010
01101 11110 11010 01000 00111 10001 00000 10101 10110 01011 11110
10101 10110 00010 00101 11000 01001 10010 00110 11000 00100 01001
00110 01001 10111 10000 10001 01100 10010 11001 11110 11010 10000
10010 00111 01000 00010 00110 01011 01000 11100 10000 11001 00001
11110 10101 10110 00111 10111 00110 10000 01111 00111 10110 10101
01000 11101 11110 01101 01010 00010 01001 10011 01000 11101 01001
01001 10000 00011 11101 01111 01011 00010 11100 10100 01111 00011
10011 11001 00000 11000 00001 11000 01110 01010 00000 01101 00001
11100 10101 00000 10100 01001 11000 00110 01110 10101 11011 00100
10010 10010
```

Converting those binary numbers to decimal gives,

```
10 25 11 25 5 31 3 15 27 19 21 5 26 0 31 9 25 23
21 28 19 10 29 16 27 4 22 13 12 21 13 24 14 17 17
18 6 1 7 29 30 7 16 2 17 24 24 2 12 7 1 22 10 15
7 1 4 26 2 12 3 29 11 27 22 23 18 18 29 27 17 6
16 20 16 13 22 30 3 30 8 5 9 10 6 19 26 3 28 1 13
15 1 19 3 7 21 12 26 13 30 26 8 7 17 0 21 22 11
30 21 22 2 5 24 9 18 6 24 4 9 6 9 23 16 17 12 18
25 30 26 16 18 7 8 2 6 11 8 28 16 25 1 30 21 22 7
23 6 16 15 7 22 21 8 29 30 13 10 2 9 19 8 29 9 9
16 3 29 15 11 2 28 20 15 3 19 25 0 24 1 24 14 10
0 13 1 28 21 0 20 9 24 6 14 21 27 4 18 18
```

Removing the numbers larger than 25 gives,

```
10 25 11 25 5 3 15 19 21 5 0 9 25 23 21 19 10 16
4 22 13 12 21 13 24 14 17 17 18 6 1 7 7 16 2 17
24 24 2 12 7 1 22 10 15 7 1 4 2 12 3 11 22 23 18
18 17 6 16 20 16 13 22 3 8 5 9 10 6 19 3 1 13 15
1 19 3 7 21 12 13 8 7 17 0 21 22 11 21 22 2 5 24
9 18 6 24 4 9 6 9 23 16 17 12 18 25 16 18 7 8 2 6
11 8 16 25 1 21 22 7 23 6 16 15 7 22 21 8 13 10 2
9 19 8 9 9 16 3 15 11 2 20 15 3 19 25 0 24 1 24
14 10 0 13 1 21 0 20 9 24 6 14 21 4 18 18
```

Conversion of 0–25 to A–Z produces,

```
KZLZFDPTVFAJZXVTKQEWNMVNYORRSGBHHQCRYYCMHBWKPHBECM
DLWXSSRGQUQNWDIFJKGTDBNPBTDHVMNIHRAVWLVWCFYJSGYEJG
JXQRMSZQSHICGLIQZBVWHXGQPHWVINKCJTIJJQDPLCUPDTZAYB
YOKANBVAUJYGOVESS
```

Taking the first 34 characters from this character stream produces,

```
KZLZFDPTVFAJZXVTKQEWNMVNYORRSGBHHQ
```

Finally, using this as a Vigenère cipher key with the plaintext produces the ciphertext,

```
XNMNIBTQKJCCRQCXCFEJVECVLELZKOUPVD
```

Since the key was produced at random and is the same length as the plaintext, so there is no repetition of the key, there is no way to recover the message without the key. Δ

We will now encrypt the same message but use the LFSR cipher as a base technique.

Example 45: As before we will encrypt the message "Nobody expects the Spanish Inquisition"[7] but this time as a one time pad form of the LFSR cipher. Since the LFSR uses a bit stream we need to convert our message to a bit stream. There are numerous ways to do this, one method would be to change the characters of the message into their ASCII numeric equivalents and then convert the ASCII numbers into binary. This will allow us to keep the mixed cases and the spaces. On the down side it will also force every 8^{th} bit to be 0 (since the ASCII values will not exceed 127) and it will produce a repetition of `00100000` in the plaintext (the value of the space). We could remove the extra zeros and remove the spaces to combat these problems but for this example we will not go to the extra trouble. Converting to ASCII and then to binary gives,

```
01001110011011110110001001101111011001000111100100
100000011001010111100001110000001100101011000110111
```

There are 304 bits in this coded message, so we will use the Blum-Blum-Shub algorithm to generate a list of 304 bits.

```
11111010000001101101001110000100110101111011110111
00111101011010110000001010000100111011010011111001
11011011110010000010010000100001110001001101000101
10010001111110111101000011011001001000101001010111
11100011001010101111011001000010010010010000011010
10000110101111110111010010110101011100110111100111
0001
```

Now all we need to do is XOR these two streams to get our ciphertext,

```
10110100011010011011000111101011101100111100010011
10111100111111101110001101000101011110001011001110
10011100111110100010001101100111010000101000001101
10000101001001111100100010000010101110001100100100
10001011000010101011111100101100001110000111001111
00100011011100101101000101100100110101101100010001
1111
```

Δ

2.12.2 Cryptography Explorer

The Cryptography Explorer program does not have a special tool for a One Time Pad, instead you can use the tools it does have to construct one time pads. We will go over the tools necessary to construct one time pads using either the Vigenère cipher or the LFSR cipher. We will find in the next section that for smaller messages, a one time pad can be constructed from a Hill cipher, but we will not discuss that method here.

The tools you need to construct a one time pad using the Vigenère cipher are,

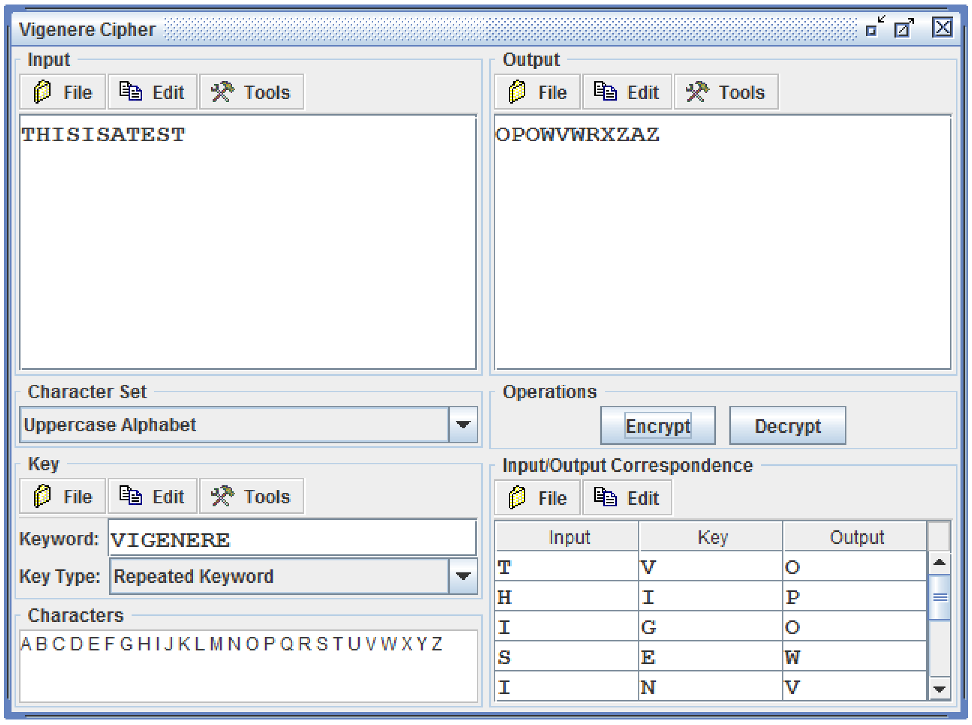
1. The Vigenère Cipher tool.
2. The Random Number Generator tool.
3. The Text Converter tool.

The tools you need to construct a one time pad using the LFSR cipher are,

1. The LFSR Cipher tool.
2. The Random Number Generator tool.
3. The Text Converter tool.

Vigenère Cipher Tool

A more general description of this tool can be found in the appendix on the Cryptography Explorer program. Here we will simply discuss the features you need to complete a one time pad implementation.



Input: THISISATEST

Output: 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 2.68: Vigenère 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. In most cases you will be using the Uppercase Alphabet.
2. Input a Keyword. For the one time pad, we will be generating the keyword using random numbers and it will be the same length as the plaintext.
3. Select the Key Type. The key type determines how the key is extended to fit the size of the message. Since for the one time pad the key is the same length as the plaintext it will not need to be extended and hence any key type that is used will produce the same result.
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 —

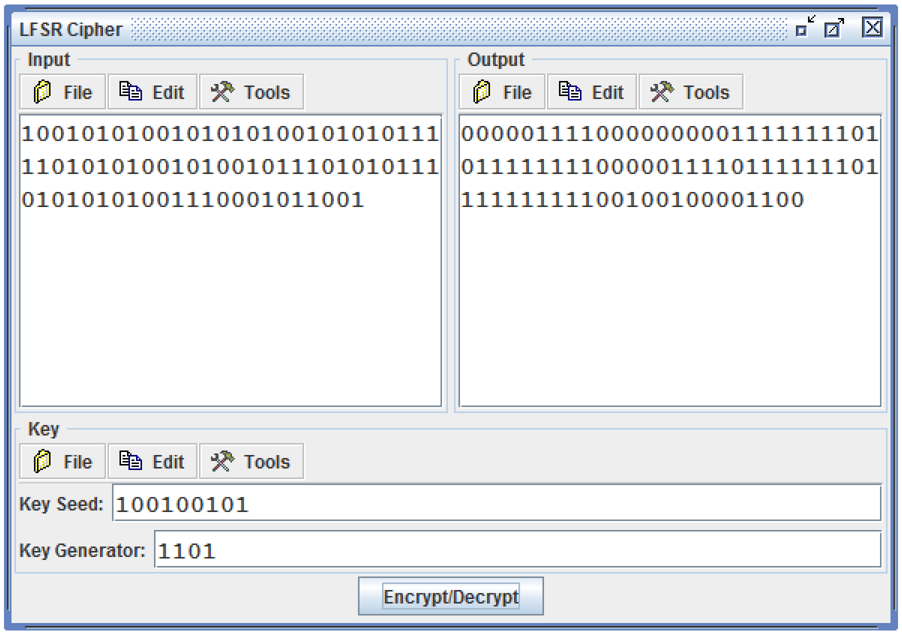
1. Input the ciphertext message into the Input box. Make sure that the characters are from the same character set as the one selected. In most cases you will be using the Uppercase Alphabet.
2. Input a Keyword. For the one time pad, we will be generating the keyword using random numbers and it will be the same length as the ciphertext.
3. Select the Key Type. The key type determines how the key is extended to fit the size of the message. Since for the one time pad the key is the same length as the plaintext it will not need to be extended and hence any key type that is used will produce the same result.
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.

Linear Feedback Shift Register Tool

A more general description of this tool can be found in the appendix on the Cryptography Explorer program. Here we will simply discuss the features you need to complete a one time pad implementation.

To Encrypt or Decrypt —

1. Input the binary plaintext (or ciphertext) message into the Input box. The input must be a binary (0 and 1) string.
2. Input a Key Seed. For a one time pad, we will be generating the seed using a random number generator.



Input:

```
1001010100101010100101010111
1101010100101001011101010111
01010101001110001011001
```

Output:

```
0000011110000000001111111101
0111111110000011110111111101
11111111100100100001100
```

Key Seed: 100100101

Key Generator: 1101

Figure 2.69: Linear Feedback Shift Register Cipher Tool

3. Input a Key Generator. The key generator is how the key is extended to match the length of the plaintext or the ciphertext. Since our key (the seed) will have the same length as the message the key generator will not be used. The program does require that the Key Generator box contain a bit stream, so simply put a 0 or 1 in the box, again it does not make a difference to the output.
4. Click the Encrypt/Decrypt button.

Random Number Generator Tool

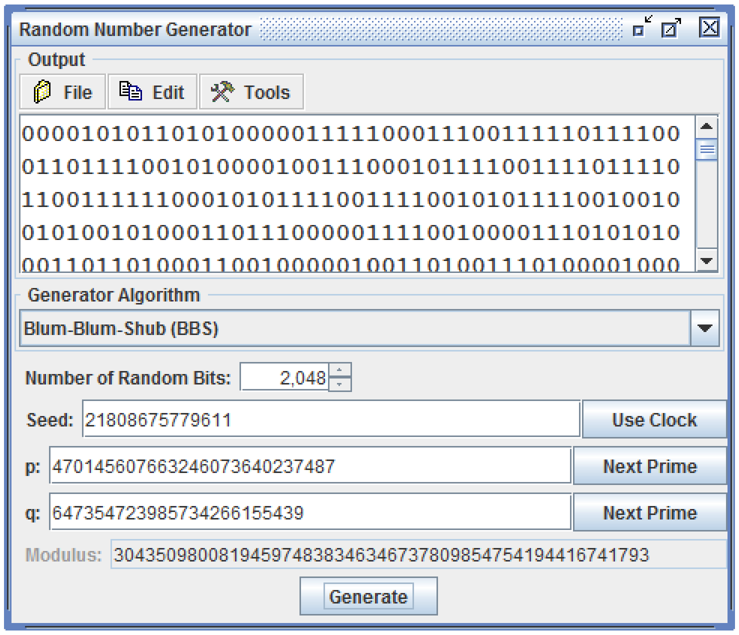
A more general description of this tool can be found in the appendix on the Cryptography Explorer program. Here we will simply discuss the features you need to complete a one time pad implementation.

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.

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. For a one time pad the Blum-Blum-Shub algorithm is preferred, the Linear Congruential algorithm (which is what Java's Random class uses as well) is not secure enough.

2. Input the number of random numbers (or random bits) you wish to generate.



```
0000101011010100000111110001110011110111100
0110111100101000010011100010111100111011110
1100111111000101011110011110010101110010010
01010010100011011100000111100100001110101010
00110110100011001000001001101001110100001000
```

Generator Algorithm: Blum-Blum-Shub (BBS)

Number of Random Bits: 2,048

Seed: 21808675779611

p: 470145607663246073640237487

q: 647354723985734266155439

Modulus: 304350980081945974838346346737809854754194416741793

Figure 2.70: Random Number Generator

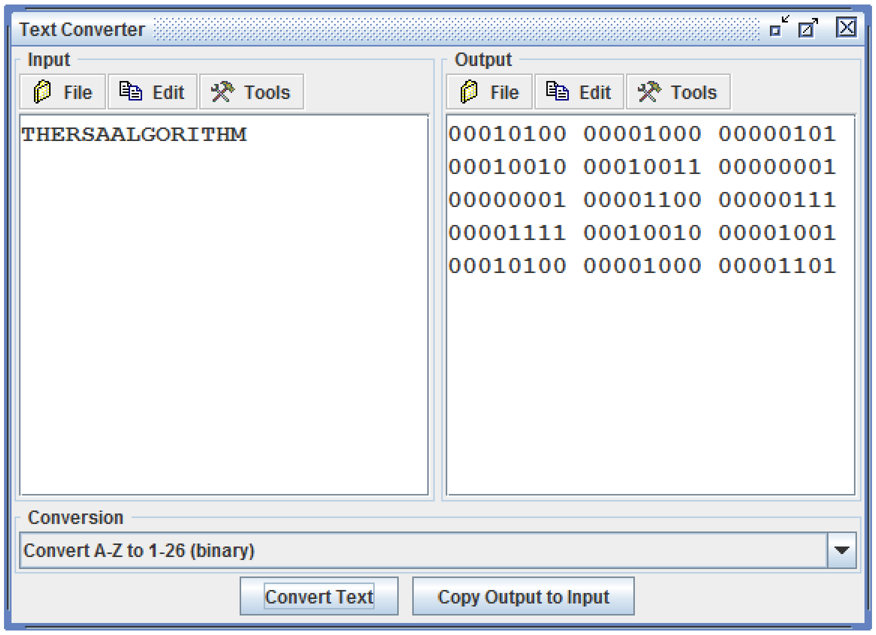
3. Input the parameters for the method, this will depend on the method chosen. For the Blum-Blum-Shub algorithm do the following,
   (a) Click on the Use Clock button to create the Seed.
   (b) Type in a fairly large number for p and then click on the Next Prime button.
   (c) Type in a fairly large number for q and then click on the Next Prime button. Make sure that p and q are not the same, or even close to each other.
4. Click the Generate button to generate the numbers or bits.

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.

Text Converter Tool

A more general description of this tool can be found in the appendix on the Cryptography Explorer program. Here we will simply discuss the features you need to complete a one time pad implementation.

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
```

Convert A-Z to 1-26 (binary)

Figure 2.71: 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 that will be used more often in the creation of a one time pad with either the Vigenère Cipher or LFSR Cipher as your base encryption and decryption technique.

Convert to UPPERCASE: Converts the input box contents to uppercase.

Remove White Space: Removes all whitespace from the input box contents. Spaces, returns, tabs, etc. are removed.

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 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 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 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 (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.

Break Character Stream into Blocks of 1: Breaks a character stream onto blocks of 1.

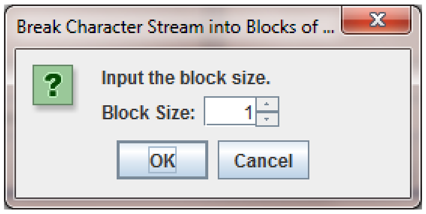
Break Character Stream into Blocks of 2: Breaks a character stream onto blocks of 2.

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 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.


Break Character Stream into Blocks of ...

Input the block size.

Block Size: 1

Figure 2.72: Character Stream Block Dialog

2.12.3 Examples Using Cryptography Explorer

We will go through the previous two examples but use the Cryptography Explorer program to do the conversions and to generate the random bits.

Example 46: Let's encrypt the message "Nobody expects the Spanish Inquisition"[7] using a one time pad form of the Vigenère Cipher. Open up the Vigenère Cipher tool, select Ciphers > Vigenere from the main menu. Copy and paste (or type) Nobody expects the Spanish Inquisition into the input box. From the tools menu of the input box, select Convert to Uppercase, and then select emove White Space. This should leave

```
NOBODYEXPECTSTHESPANISHINQUISITION
```

in the input box.

Open the Random Number Generator by selecting Tools > Calculators > Random Number Generator from the main menu. Under the Generator Algorithm drop down box, select Blum-Blum-Shub (BBS), input 1000 for the number of bits, click the Use Clock button to set the seed value, type in a large number for p and click the Next Prime button beside the p input box, type in a large number for q and click the Next Prime button beside the q input box, finally, click the Generate button. At this point you will have 1000 bits in the output box. Note that your stream and our stream will be different.

```
01010110010101111001001011111100011011111101110011
10101001011101000000111110100111001101111010111100
10011010101110110000110110010010110011010110010101
01101110000111010001100011001000110000010011111101
11110001111000000001010001110001100000010011000011 1
00001101100101001111001110000100100110100001001100
00011111010101111011101101011110010100101110111011
10001001101000010100100000110110110111100001111110
01000001010100101010001101001111010000111110000001
01101011110000110011000110011110101011001101001101
11110110100100000111100010000010101101100101111110
10101101100001000101110000100110010001101100000100
01001001100100110111100001000101100100101100111110
11010100001001000111010000000100011001011010001110 0
10000110010000111110101011011000111101110011010000
01111001111011010101010001110111111001101010100001 0
01001100110100011101010010100110000000111110101111
01011000101110010100011110001110011110010000011000
00001110000111001010000000110100001111001010100000
10100010011100000110011101010111011001001001010010
```

Now we need to do some conversions. Open the Text Converter tool by selecting Tools > Text Tools > Text Converter from the main menu. Copy and paste the random bit stream from the output box of the random number generator to the input box of the text converter.

In the Conversion selector at the bottom of the window, select Break Character Stream into Blocks of 5, then click the Convert Text button. This will break the stream into blocks of 5, as below.

```
01010 11001 01011 11001 00101 11111 00011 01111 11011 10011
10101 00101 11010 00000 11111 01001 11001 10111 10101 11100
10011 01010 11101 10000 11011 00100 10110 01101 01100 10101
01101 11000 01110 10001 10001 10010 00110 00001 00111 11101
11110 00111 10000 00010 10001 11000 11000 00010 01100 00111
00001 10110 01010 01111 00111 00001 00100 11010 00010 01100
00011 11101 01011 11011 10110 10111 10010 10010 11101 11011
10001 00110 10000 10100 10000 01101 10110 11110 00011 11110
01000 00101 01001 01010 00110 10011 11010 00011 11100 00001
01101 01111 00001 10011 00011 00111 10101 01100 11010 01101
11110 11010 01000 00111 10001 00000 10101 10110 01011 11110
10101 10110 00010 00101 11000 01001 10010 00110 11000 00100
01001 00110 01001 10111 10000 10001 01100 10010 11001 11110
11010 10000 10010 00111 01000 00010 00110 01011 01000 11100
10000 11001 00001 11110 10101 10110 00111 10111 00110 10000
01111 00111 10110 10101 01000 11101 11110 01101 01010 00010
01001 10011 01000 11101 01001 01001 10000 00011 11101 01111
01011 00010 11100 10100 01111 00011 10011 11001 00000 11000
00001 11000 01110 01010 00000 01101 00001 11100 10101 00000
10100 01001 11000 00110 01110 10101 11011 00100 10010 10010
```

Click the Copy Output to Input button to move the text to the input box. Select the Convert Binary to Decimal conversion and click the Convert Text, this will produce the list of numbers below.

```
10 25 11 25 5 31 3 15 27 19 21 5 26 0 31 9 25 23
21 28 19 10 29 16 27 4 22 13 12 21 13 24 14 17 17
18 6 1 7 29 30 7 16 2 17 24 24 2 12 7 1 22 10 15
7 1 4 26 2 12 3 29 11 27 22 23 18 18 29 27 17 6
16 20 16 13 22 30 3 30 8 5 9 10 6 19 26 3 28 1 13
15 1 19 3 7 21 12 26 13 30 26 8 7 17 0 21 22 11
30 21 22 2 5 24 9 18 6 24 4 9 6 9 23 16 17 12 18
25 30 26 16 18 7 8 2 6 11 8 28 16 25 1 30 21 22 7
23 6 16 15 7 22 21 8 29 30 13 10 2 9 19 8 29 9 9
16 3 29 15 11 2 28 20 15 3 19 25 0 24 1 24 14 10
0 13 1 28 21 0 20 9 24 6 14 21 27 4 18 18
```

Click the Copy Output to Input button to move the text to the input box. Now we do not have a special conversion for this so we will have to do it by hand. Remove any number from the above list that exceeds 25. When you are done, you should get,

```
10 25 11 25 5 3 15 19 21 5 0 9 25 23 21 19 10 16
4 22 13 12 21 13 24 14 17 17 18 6 1 7 7 16 2 17
24 24 2 12 7 1 22 10 15 7 1 4 2 12 3 11 22 23 18
18 17 6 16 20 16 13 22 3 8 5 9 10 6 19 3 1 13 15
1 19 3 7 21 12 13 8 7 17 0 21 22 11 21 22 2 5 24
9 18 6 24 4 9 6 9 23 16 17 12 18 25 16 18 7 8 2 6
11 8 16 25 1 21 22 7 23 6 16 15 7 22 21 8 13 10 2
9 19 8 9 9 16 3 15 11 2 20 15 3 19 25 0 24 1 24
14 10 0 13 1 21 0 20 9 24 6 14 21 4 18 18
```

Select the Convert of 0–25 to A–Z and click the Convert Text button, this will produce,

```
KZLZFDPTVFAJZXVTKQEWNMVNYORRSGBHHQCRYYCMHBWKPHBECM
DLWXSSRGQUQNWDIFJKGTDBNPBTDHVMNIHRAVWLVWCFYJSGYEJG
JXQRMSZQSHICGLIQZBVWHXGQPHWVINKCJTIJJQDPLCUPDTZAYB
YOKANBVAUJYGOVESS
```

Extract the first 34 characters, or really any consecutive string of 34 characters. You can do this the old fashioned way of counting as you are selecting or you can do the following, which will work better for longer strings. Click the Copy Output to Input button to move the text to the input box. Select the Break Character Stream into Blocks of n... conversion and click the Convert Text, when a dialog box appears input 34 into the Block Size and click OK, this will break the string at every 34 character. Extract the first substring to get.

```
KZLZFDPTVFAJZXVTKQEWNMVNYORRSGBHHQ
```

Finally, go back to the Vigenère Cipher tool and copy and paste this string into the Keyword box and click Encrypt to produce the ciphertext,

```
XNMNIBTQKJCCRQCXCFEJVECVLELZKOUPVD
```

Δ

Example 47: As before we will encrypt the message "Nobody expects the Spanish Inquisition"[7] but this time as a one time pad form of the LFSR cipher. To convert "Nobody expects the Spanish Inquisition" into binary, open up the Text Converter by selecting Tools > Text Tools > Text Converter from the main menu. Copy or type Nobody expects the Spanish Inquisition into the input box, select the Convert Text to ASCII (binary) conversion and click Convert Text. Now click Copy Output to Input, select the Remove White Space conversion and click Convert Text again. At this point you will have the following bit stream.

```
01001110011011110110001001101111011001000111100100
10000001100101011110000111000001100101011000110111
01000111001100100000011101000110100001100101001000
00010100110111000001100001011011100110100101110011
01101000001000000100100101101110011100010111010101
10100101110011011010010111010001101001011011110110
1110
```

There are 304 bits in this coded message, so we will use the Blum-Blum-Shub algorithm to generate a list of 304 bits.

Open the Random Number Generator by selecting Tools > Calculators > Random Number Generator from the main menu. Under the Generator Algorithm drop down box, select Blum-Blum-Shub (BBS), input 304 for the number of bits, click the Use Clock button to set the seed value, type in a large number for p and click the Next Prime button beside the p input box, type in a large number for q and click the Next Prime button beside the q input box, finally, click the Generate button. At this point you will have 304 bits in the output box. Note that your stream and our stream will be different. If you want to generate the same bit stream use 23462350276969 for the seed, and 710346501376503176504316150445 9 and 135431590175093475031780458714 07 for p and q.

```
11111010000001101101001110000100110101111011110111
00111101011010110000001010000100111011010011111001
11011011110010000010010000100001110001001101000101
10010001111110111101000011011001001000101001010111
11100011001010101111011001000010010010010000011010
10000110101111110111010010110101011100110111100111
0001
```

Now all we need to do is XOR these two streams. Open up the LFSR tool, select Ciphers > LFSR from the main menu. Put the "Nobody expects the Spanish Inquisition" bit stream into the input box, put the above random key into the Key Seed box and put a 0 in the Key Generator box. Click the Encrypt/Decrypt button to get the following ciphertext.

```
10110100011010011011000111101011101100111100010011
10111100111111101110001101000101011110001011001110
10011100111110100010001101100111010000101000001101
10000101001001111100100010000010101110001100100100
10001011000010101011111100101100001110000111001111
00100011011100101101000101100100110101101100010001
1111
```

Δ

2.13 The Hill Cipher

2.13.1 Introduction

You have probably noticed that there are two sections on the Hill Cipher. The two sections contain the same material, the only difference is the approach we take. When Lester Hill devised this cipher it was 1929, at the time, mathematicians more commonly viewed applying a linear transformation to a vector as taking a row vector $\mathbf{v}$ and transformation matrix M and applying the transformation by, $\mathbf{v}M = \mathbf{w}$. Now, we tend to view a linear transformation more as a function from one vector space to another. That is, $T : V \to W$, and hence we have adopted a more functional notation. So if $\mathbf{v}$ is a vector in the vector space V and T is a linear transformation from V to W, then we would write $T(\mathbf{v}) = \mathbf{w}$ or sometimes just $T\mathbf{v} = \mathbf{w}$ for applying the transformation T to the vector $\mathbf{v}$. This functional notation has also transferred over to matrices. So if V and W are finite dimensional vector spaces, M is the matrix for a linear transformation T over some bases for V and W, and $\mathbf{v}$ is a vector in the vector space V then $T(\mathbf{v}) = M\mathbf{v} = \mathbf{w}$. Note that in this notation, $\mathbf{v}$ and $\mathbf{w}$ are doing double duty by representing vectors in V and W respectively and representing their coordinate vectors over bases for V and W respectively, also they are column vectors instead of row vectors.

The only difference between this section and the next is approach we take. Here, we use the more modern notation of $M\mathbf{v}$ whereas in the next section we use the classical notation that Lester Hill did in 1929 of $\mathbf{v}M$. If you recall from your linear algebra class, if you have a matrix M and row vectors $\mathbf{v}$ and $\mathbf{w}$ with $\mathbf{v}M = \mathbf{w}$ and you transpose both sides you get, $(\mathbf{v}M)^T = M^T\mathbf{v}^T = \mathbf{w}^T$. Of course, $\mathbf{v}^T$ and $\mathbf{w}^T$ are column vectors, so the only difference between this section and the next is that all of the transformation matrices are transposes of each other. So why bother having two separate sections? We did this just for the convenience of the reader and the instructor. If you are using this set of notes as a supplement for a linear algebra class then you will probably want to use this section, whereas, if you are using this set of notes for a cryptography course then you will probably want to use the next section. The majority of cryptography textbooks that discuss the Hill Cipher use the classical notation of $\mathbf{v}M = \mathbf{w}$.

2.13.2 History

The Hill Cipher is a block cipher that was developed by Lester Hill in 1929, setting it firmly in the classical age of cryptography. 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*[22]. Although it seems that this method was not used much in practice, it marked a transitional period in cryptography, where cryptography shifted 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 Germans in World War II, and other cipher machines of the same period, cryptanalysis of these ciphertexts required advanced mathematics and an enormous amount of computation, far beyond that of a single person or group of people.

2.13.3 Encryption with the Hill Cipher

Although we can apply the Hill Cipher to any language and any alphabet we will consider the uppercase English alphabet for now and then generalize it to other alphabets later in the section. As with most of our other methods, we will be doing calculations modulo 26. The key to the Hill cipher is a square matrix of size $n \times n$, where n can be as large as desired. When working with matrices modulo 26, we simply do all calculations modulo 26. So if we were to add or multiply two matrices, we would do it in the same manner as we would in a linear algebra class but at the end we would mod each matrix entry by 26. Also, if we take the determinant of a matrix, we do the calculation as we would in a linear algebra class and then mod out the determinant value by 26. In addition, if we wanted to invert a square matrix modulo 26, we can use the standard reduction algorithm but do all calculations modulo 26. This would include division, so if we wanted to multiply a row by $\frac{1}{3}$ we would multiply the row by $3^{-1} \pmod{26}$. So it goes without saying that we can only divide a row by a number that is invertible modulo 26, for example, dividing by 3 would be fine but we would not be able to divide by 8, since $\gcd(26, 8) \neq 1$. The appendix on modular arithmetic discusses matrix operations and gives several examples of matrix arithmetic on matrices, including inverting a square matrix.

The Hill cipher is really just a block cipher where the plaintext is broken up into blocks of size n, converted to an n-dimensional vector (column vector in this section), multiplied (on the left) by an $n \times n$ matrix (which is the encryption key) to produce another n-dimensional vector, and then converted back into the alphabet as the ciphertext. Decryption is done in a similar manner except that the decryption key is the inverse of the encryption matrix, modulo 26.

The Hill Cipher Encryption Algorithm

1. Find an $n \times n$ matrix E that is invertible modulo 26. This is the encryption key.

2. Take the message that is to be sent (the plaintext), remove all of the spaces and punctuation symbols, and convert the letters into all uppercase.

3. Convert each character to a number between 0 and 25. The usual way to do this is $A = 0$, $B = 1$, $C = 2$, $\ldots$, $Z = 25$.

   As a historical note, Lester Hill did not use this coding of letters to numbers, he simply mixed up the order. Mixing up the order does not make the method more secure, it simply combines the Hill cipher with a simple substitution cipher.

4. Divide this string of numbers up into blocks of size n. Note that if E is an $n \times n$ matrix then the block size is n. Another note, if the message does not break evenly

Table 2.37: English Alphabet Numeric Coding 0–25

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M
Code	0	1	2	3	4	5	6	7	8	9	10	11	12
Letter	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Code	13	14	15	16	17	18	19	20	21	22	23	24	25

into blocks of size n we pad the ending of the message with characters, this can be done at random.

5. Write each block as a column vector of size n. At this point the message is a sequence of n-dimensional vectors, $v_1, v_2, \ldots, v_t$.

6. Take each of the vectors and multiply them by the encryption matrix E, so

$$\begin{aligned} Ev_1 &= w_1 \\ Ev_2 &= w_2 \\ Ev_3 &= w_3 \\ &\vdots \\ Ev_t &= w_t \end{aligned}$$

7. Take the vectors $w_1, w_2, \ldots, w_t$, write the entries of the vectors in order, convert the numbers back to characters and you have your ciphertext.

One note about this algorithm is that we can do step 6 with a single matrix multiplication. If we let the message matrix M be the matrix produced by having the vectors $v_1, v_2, \ldots, v_t$ as columns, that is, $M = [v_1\ v_2\ \ldots\ v_t]$ then $EM = [w_1\ w_2\ \ldots\ w_t] = C$ would be our ciphertext matrix.

Example 48: Say Alice wants to send Bob the message "Cryptography is cool!"

1. Alice chooses the block size $n = 3$ and chooses the encryption matrix E to be,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

   Since $\det(E) \pmod{26} = 11$, and 11 is invertible modulo 26, the matrix E is also invertible modulo 26.

2. The message that is to be sent is "Cryptography is cool!", removing the spaces and punctuation symbols, and convert the letters into all uppercase gives

   CRYPTOGRAPHYISCOOL

3. Conversion to numbers using $A = 0$, $B = 1$, $C = 2$, ..., $Z = 25$, gives

   2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

4. Dividing this string of numbers up into blocks of size 3.

   2 17 24   15 19 14   6 17 0   15 7 24   8 18 2   14 14 11

   so no padding is needed here.

5. Converting these blocks into a message matrix M gives,

$$M = \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix}$$

6. Multiply by the encryption matrix E,

$$EM = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix} \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix} = C$$

7. Convert C into the ciphertext.

   25 18 7 11 5 6 11 10 19 21 3 20 22 0 16 1 2 6

   ZSHLFGLKTVDUWAQBCG

   So Alice will send "ZSHLFGLKTVDUWAQBCG" to Bob.

Δ

Since this is a symmetric cipher, Alice and Bob would have to share this key with each other. They obviously could not simply call or text each other with this information since Eve could easily intercept that call or text and would know the key. So either Alice and Bob would have to meet in person, in a secure location, and exchange the key or they would need some other trusted person to deliver the key from Alice to Bob. This difficulty in exchanging the key securely gave rise to the creation of public-key systems which are commonly used today, for more information on public-key systems please see the references [55] and [62].

2.13.4 Decryption with the Hill Cipher

Now that Bob has the encrypted message and the encryption key he can decrypt the message that Alice had sent to him. The decryption algorithm is essentially the same as the encryption algorithm, except that we use E^{-1} in place of E. Since $EM = C$, and E is invertible we can calculate $M = E^{-1}C$. We will call $D = E^{-1}$ the decryption matrix, so $DC = M$. Remember that this inverse is the inverse modulo 26.

The Hill Cipher Decryption Algorithm

1. Find $D = E^{-1} \pmod{26}$. This is the decryption key.
2. Take the ciphertext and convert it to the matrix C.
3. Calculate $DC = M$.
4. Convert the matrix M to the plaintext message. You may need to insert the appropriate spaces and punctuation symbols since these were removed.

Example 49: Bob has the encrypted message ZSHLFGLKTVDUWAQBCG.

1. He calculates

$$\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}^{-1} \pmod{26} = \begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$$

2. He also converts the ciphertext to the matrix C.

ZSHLFGLKTVDUWAQBCG

25 18 7 11 5 6 11 10 19 21 3 20 22 0 16 1 2 6

and since he knows that the block size is 3 he constructs C as

$$C = \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$$

3. Calculate $DC = M$.

$$DC = \begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix} \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix} = M$$

4. Convert the matrix M to the plaintext message.

2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

CRYPTOGRAPHYISCOOL

So Bob adds in a couple spaces to get CRYPTOGRAPHY IS COOL!

Δ

2.13.5 Breaking the Hill Cipher

Now it is Eve's turn, how can she find the key to a Hill cipher? Looking at the encryption algorithm we know that $EM = C$. If we have a portion of the plaintext and its corresponding ciphertext (a Known Plaintext attack) then we have a little of M and C. If we are lucky, the portion of M that we have might form an invertible $n \times n$ matrix (modulo 26). Then $EM = C$ could be rewritten as $E = CM^{-1}$, giving her the encryption matrix. From there, she simply inverts E modulo 26 to get D and then she can decrypt the entire message.

There is really one more piece of information that Eve needs, if she just has plaintext and ciphertext characters she does not know the block size n and hence she does not know the size of E nor the size of the matrices M and C she needs to use to find E. This is clearly a problem. But there is a way to guess the possible block sizes, if the message is not too long. Since Eve can get the entire ciphertext, she knows the number of letters in the message (possibly padded) and that this number of characters must be a multiple of the block size. So if she has intercepted ZSHLFGLKTVDUWAQBCG she knows that the message has 18 characters in it, so the block sizes could possibly be, 2, 3, 6, 9 or 18. If the block size was 6, 9, or 18 she would not have enough characters to create M and C so it would not be possible for her to find E and hence D. So the only possibilities she would have that would allow her to find the key would be block sizes of 2 or 3. If these both fail to produce a key then she knows that she will not be able to break the code and not know the original message. As an example we will assume that Eve knows that CRYPTOGRAPHYISCOOL encrypts as ZSHLFGLKTVDUWAQBCG and we will follow the process outlined above to find that the block size is 3 and the matrix E.

Example 50: Eve has intercepted the encrypted message ZSHLFGLKTVDUWAQBCG and from other espionage knows that this message was CRYPTOGRAPHYISCOOL. She also knows that other messages sent that day between Alice and Bob are using the same key and she wishes to decrypt them as well, but she has no other information about these other messages.

Since the message size 18 she knows that that block size must be 2, 3, 6, 9 or 18, and since she has only 18 characters to work with her only hope is that the block size is either 2 or 3. If both of these fail it is back to other espionage.

Let's start with a block size of 2. Since,

CRYPTOGRAPHYISCOOL — 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

encrypts as

ZSHLFGLKTVDUWAQBCG — 25 18 7 11 5 6 11 10 19 21 3 20 22 0 16 1 2 6

we know that

$$\begin{bmatrix} 2 \\ 17 \end{bmatrix} \longrightarrow \begin{bmatrix} 25 \\ 18 \end{bmatrix} \qquad \begin{bmatrix} 24 \\ 15 \end{bmatrix} \longrightarrow \begin{bmatrix} 7 \\ 11 \end{bmatrix} \qquad \begin{bmatrix} 19 \\ 14 \end{bmatrix} \longrightarrow \begin{bmatrix} 5 \\ 6 \end{bmatrix} \cdots$$

So we want to construct a 2×2 matrix out of the plaintext blocks that is invertible modulo 26. If we take the first two blocks 2 17 and 24 15 and build a matrix from them

$$M = \begin{bmatrix} 2 & 24 \\ 17 & 15 \end{bmatrix}$$

then $\det(M) = -378$ which is not relatively prime to 26, so M is not invertible. Actually, we should have seen this coming with what we know about determinants, notice that the first row has all even numbers in it and 2 is a factor of 26.

If we move on to the next block, 19 14 and keep the first block our M matrix becomes,

$$M = \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}$$

then $\det(M) = -295$ which is 17 when we take it modulo 26. Since 17 has an inverse modulo 26 we are in business. Our equation becomes,

$$E \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix} = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix}$$

So

$$E = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}^{-1} = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \begin{bmatrix} 10 & 5 \\ 25 & 20 \end{bmatrix} = \begin{bmatrix} 11 & 17 \\ 18 & 2 \end{bmatrix}$$

To see if this works we test it on our plaintext message CRYPTOGRAPHYISCOOL.

$$\begin{aligned} C = EM &= \begin{bmatrix} 11 & 17 \\ 18 & 2 \end{bmatrix} \begin{bmatrix} 2 & 24 & 19 & 6 & 0 & 7 & 8 & 2 & 14 \\ 17 & 15 & 14 & 17 & 15 & 24 & 18 & 14 & 11 \end{bmatrix} \\ &= \begin{bmatrix} 25 & 25 & 5 & 17 & 21 & 17 & 4 & 0 & 3 \\ 18 & 20 & 6 & 12 & 4 & 18 & 24 & 12 & 14 \end{bmatrix} \end{aligned}$$

As we can see, columns 1 and 3 encrypt correctly, they should since those were the ones we used, but the rest of the encryption is incorrect. Conclusion, the block size is not 2. So we move on to block size 3. If it is then there is a 3×3 matrix E with

$$E \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$$

As before, we want to select three columns from our message matrix that will produce an invertible 3×3 matrix. Looking at the last row, all entries except for 11 are even, so the last column must be used. Since the last column first row is even we must have at least one odd number in the first row of the columns we use. So we could not select the remaining two columns from columns 1, 3, and 5. Furthermore, column 5 is all even so selecting it would be pointless. So in our selection we must have column 6 and at least one of columns 2 and 4. We will try columns 1, 2, and 6. This gives,

$$M = \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}$$

Since $\det(M) = -791 \equiv 15 \pmod{26}$, we know that M is invertible. So

$$EM = E \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix}$$

and then,

$$E = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}^{-1} = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \begin{bmatrix} 13 & 9 & 24 \\ 3 & 12 & 14 \\ 8 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Now, we know this is correct but Eve would still need to check her possible solution, doing so she would get,

$$\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix} \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$$

which checks with the ciphertext and she has found the encryption matrix, E.

Δ

If Eve did not have a crib or could not do a known plaintext or known ciphertext attack she might still be able break the cipher with a ciphertext only attack by looking at n-gram frequencies if there was a sufficient amount of ciphertext to go on. This would, of course, be much more difficult to do as the size of the encryption matrix increases.

2.13.6 The Hill Cipher from a Linear Algebra Point of View

We are going to abstract the Hill cipher a little bit and relate it to the material that is commonly covered in a linear algebra class. Most of this will be done in the exercises but we will give a quick introduction here along with some notation.

From the description of the algorithm in this section, it is clear that the Hill cipher can be viewed as a linear transformation. In fact, it is an invertible matrix transformation. Let's put this observation into a more mathematical context. First of all, we are no longer working over the real numbers. We are working with the integers 0–25 with addition and multiplication being done modulo 26.

Notation: Let $\mathbb{A}$ denote the set $\{0, 1, 2, \ldots, 25\}$ where addition and multiplication are done modulo 26.

We use $\mathbb{A}$ as our notation here to designate our "alphabet". If you have taken a course in abstract algebra you would call this structure a ring and denote it as $\mathbb{Z}_{26}$ or $\mathbb{Z}/26\mathbb{Z}$. We will not need this notation nor will we take a digression into ring theory.

The key to a Hill cipher is an $n \times n$ matrix which has an inverse (modulo 26). So it sends n-dimensional vectors with entries from $\mathbb{A}$ to n-dimensional vectors with entries from $\mathbb{A}$.

Notation: Let $\mathbb{A}^n$ as the set of n-dimensional vectors with entries from $\mathbb{A}$

With this notation, the Hill cipher can be viewed simply as an invertible linear transformation $E : \mathbb{A}^n \to \mathbb{A}^n$. Its decryption is also an invertible linear transformation $D : \mathbb{A}^n \to \mathbb{A}^n$. From the Invertible Matrix Theorem in linear algebra, there are many properties of an invertible matrix and the transformation it induces. Some of these properties are crucial when it comes to cryptography. For example, say that E was not invertible but we used it as an encryption matrix for a Hill cipher. One immediate problem comes to mind in that if E is not invertible, finding the decryption matrix D is not possible, hence Bob has a bit of a problem when he receives the ciphertext from Alice. We also know that when we consider E as a linear transformation, this transformation is not one-to-one nor is it onto.

If the transformation is not one-to-one then it is many-to-one which means that there are different vectors in the domain that are mapped to the same vector in the range. Since the domain is associated with the plaintext and the range with the ciphertext this translates to having the two different plaintext words (or segments of words) encrypt to the same ciphertext block. So if we could devise a way to decrypt the message this block of ciphertext would decrypt as two or more possible plaintext blocks. In this situation, it may be possible to figure out which one of the possible decryptions is the correct one but there may be cases where this is not possible either.

Example 51: Say that Alice chooses the following matrix for her Hill cipher,

$$E = \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix}$$

As we can see, the first row is a multiple of 2 and hence this matrix will not be invertible modulo 26. If we take the determinant modulo 26 we get,

$$\begin{vmatrix} 2 & 4 \\ 3 & 5 \end{vmatrix} \pmod{26} = 24$$

and since the GCD of 24 and 26 is not 1 we have verified that E does not have an inverse. Now say that Alice wants to send the message HELP to Bob. So Alice encodes the message as usual, HELP becomes 7 4 11 15, which produces the message matrix

$$M = \begin{bmatrix} 7 & 11 \\ 4 & 15 \end{bmatrix}$$

Then, modulo 26,

$$EM = \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} 7 & 11 \\ 4 & 15 \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 15 & 4 \end{bmatrix}$$

Which produces the ciphertext EPEE, which Alice sends to Bob.

Now Bob receives the ciphertext EPEE from Alice and when he goes to decrypt the message he finds out that the encryption matrix E is not invertible modulo 26. So what does he do? Since Bob is not the type to give up, he starts to reason this through. He knows that the first two letters of the message must encrypt to EP and he knows the encryption matrix E, this gives him the equation,

$$\begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 15 \end{bmatrix}$$

So he can find all of the letters xy that encrypt to EP by solving this system. He knows that E is not invertible, hence the encoding transformation is many-to-one and therefore he will get several possible solutions. Unlike when working over the real or complex numbers he is working modulo 26 and so there will not be an infinite number of solutions, so he will have only a finite number of possibilities to worry about. He then solves the system, remember he is working modulo 26.

$$\begin{aligned}
\begin{bmatrix} 2 & 4 & 4 \\ 3 & 5 & 15 \end{bmatrix} & \xrightarrow{R_1 \leftrightarrow R_2} \begin{bmatrix} 3 & 5 & 15 \\ 2 & 4 & 4 \end{bmatrix} \\
& \xrightarrow{9R_1} \begin{bmatrix} 1 & 19 & 5 \\ 2 & 4 & 4 \end{bmatrix} \\
& \xrightarrow{24R_1+R_2} \begin{bmatrix} 1 & 19 & 5 \\ 0 & 18 & 20 \end{bmatrix} \\
& \xrightarrow{25R_2+R_1} \begin{bmatrix} 1 & 1 & 11 \\ 0 & 18 & 20 \end{bmatrix}
\end{aligned}$$

From this he knows that $18y = 20$ and that $x + y = 11$, that is $x = 11 - y$. If we evaluate $18y$ for $0 \leq y \leq 25$ we see that the only two values of y that give us $18y = 20$ are $y = 4$ and $y = 17$. When $y = 4$ we have $x = 11 - 4 = 7$ and when $y = 17$ we have $x = 11 - 17 = 20$. So the first two letters are either 7 4 or 20 17, that is, either HE or UT. The number of English words that begin with UT is fairly small so Bob probably figures that the letters are HE but we will keep all of our options open at this point, who knows, Alice may be telling Bob she plans to take a trip to UTAH. Now he attacks the last two letters in the same way.

$$\begin{aligned}
\begin{bmatrix} 2 & 4 & 4 \\ 3 & 5 & 4 \end{bmatrix} & \xrightarrow{R_1 \leftrightarrow R_2} \begin{bmatrix} 3 & 5 & 4 \\ 2 & 4 & 4 \end{bmatrix} \\
& \xrightarrow{9R_1} \begin{bmatrix} 1 & 19 & 10 \\ 2 & 4 & 4 \end{bmatrix} \\
& \xrightarrow{24R_1+R_2} \begin{bmatrix} 1 & 19 & 10 \\ 0 & 18 & 10 \end{bmatrix} \\
& \xrightarrow{25R_2+R_1} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 18 & 10 \end{bmatrix}
\end{aligned}$$

From this he knows that $18y = 10$ and that $x + y = 0$, that is $x = -y$. If we evaluate $18y$ for $0 \leq y \leq 25$ we see that the only two values of y that give us $18y = 10$ are $y = 2$ and $y = 15$. When $y = 2$ we have $x = -2 = 24$ and when $y = 15$ we have $x = -15 = 11$. So the last two letters are either 24 2 or 11 15, that is, either YC or LP. Again Bob probably figures that the letters are LP since the number of English words ending in YC is surprisingly small but again we will consider all possibilities. So there are four words that will encrypt to EPEE under this transformation, they are, HEYC, HELP, UTYC, and UTLP. Under the assumption that Alice sent an English word we would guess that the plaintext for the message was HELP.

A few things to note here,

1. If this message had been longer the HEYC option could have been valid if the original message was "Hey c…".

2. In the reductions we did above we had to stop where we did since 18 does not have an inverse modulo 26.

3. If we take a closer look at our two-letter decryptions, HE and UT, that is, 7 4 and 20 17. Note that,
$$\begin{bmatrix} 20 \\ 17 \end{bmatrix} - \begin{bmatrix} 7 \\ 4 \end{bmatrix} = \begin{bmatrix} 13 \\ 13 \end{bmatrix}$$
and
$$\begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} 13 \\ 13 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
We could have also come to this conclusion by solving the homogeneous system,
$$\begin{aligned}
\begin{bmatrix} 2 & 4 & 0 \\ 3 & 5 & 0 \end{bmatrix} &\xrightarrow{R_1 \leftrightarrow R_2} \begin{bmatrix} 3 & 5 & 0 \\ 2 & 4 & 0 \end{bmatrix} \\
&\xrightarrow{9R_1} \begin{bmatrix} 1 & 19 & 0 \\ 2 & 4 & 0 \end{bmatrix} \\
&\xrightarrow{24R_1+R_2} \begin{bmatrix} 1 & 19 & 0 \\ 0 & 18 & 0 \end{bmatrix} \\
&\xrightarrow{25R_2+R_1} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 18 & 0 \end{bmatrix}
\end{aligned}$$
From this we know that $18y = 0$ and that $x + y = 0$, that is $x = -y$. If we evaluate $18y$ for $0 \le y \le 25$ we see that the only two values of y that give us $18y = 0$ are $y = 0$ and $y = 13$. When $y = 0$ we have $x = 0$ and when $y = 13$ we have $x = -13 = 13$. So the vectors that get sent to the zero vector are
$$\begin{bmatrix} 13 \\ 13 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
In other words, thinking of E as the transformation,
$$\ker(E) = \left\{ \begin{bmatrix} 13 \\ 13 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}$$
From linear algebra, we know that the entire set of solutions to a consistent non-homogeneous system of equations can be written as a single solution to the non-homogeneous system plus any element of the kernel (or null space) of the coefficient matrix. So here that translates to,
$$\begin{bmatrix} 7 \\ 4 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 7 \\ 4 \end{bmatrix} + \begin{bmatrix} 13 \\ 13 \end{bmatrix} = \begin{bmatrix} 20 \\ 17 \end{bmatrix}$$
and
$$\begin{bmatrix} 24 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 24 \\ 2 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 24 \\ 2 \end{bmatrix} + \begin{bmatrix} 13 \\ 13 \end{bmatrix} = \begin{bmatrix} 11 \\ 15 \end{bmatrix}$$
Although, this observation may not have saved us too many calculations here, but in some cases, especially with larger block sizes, this method could save a lot of work.

Δ

The above discussion and example shows us why we should use an invertible matrix as our encryption matrix for the Hill cipher algorithm. Furthermore, it puts both the theory and calculations we have been doing into the language of linear algebra.

Let's take a quick look at Eve's job, the cryptanalysis process. Recall that we did a known plaintext attack on the system to find the encryption matrix, and hence the decryption matrix. In this discussion, we will be referring to the "Cryptography is Cool" example from before, we reproduce portions of it in the example below.

Example 52: Recall that Eve knows the plaintext and the corresponding ciphertext of the message,

CRYPTOGRAPHYISCOOL — 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

encrypts as

ZSHLFGLKTVDUWAQBCG — 25 18 7 11 5 6 11 10 19 21 3 20 22 0 16 1 2 6

When she started with block size of 2, she came up with the vector correspondence,

$$\begin{bmatrix} 2 \\ 17 \end{bmatrix} \longrightarrow \begin{bmatrix} 25 \\ 18 \end{bmatrix} \quad \begin{bmatrix} 24 \\ 15 \end{bmatrix} \longrightarrow \begin{bmatrix} 7 \\ 11 \end{bmatrix} \quad \begin{bmatrix} 19 \\ 14 \end{bmatrix} \longrightarrow \begin{bmatrix} 5 \\ 6 \end{bmatrix} \cdots$$

Then proceeded to build a 2×2 matrix out of the plaintext blocks (vectors) that was invertible modulo 26. Obtaining, after some work,

$$M = \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}$$

which was invertible. Then knowing that,

$$E \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix} = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix}$$

She calculated,

$$E = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}^{-1} = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \begin{bmatrix} 10 & 5 \\ 25 & 20 \end{bmatrix} = \begin{bmatrix} 11 & 17 \\ 18 & 2 \end{bmatrix}$$

Which turned out not to work for the message, telling her that the block size of 2 was incorrect. When she moved on to block size 3, she constructed,

$$M = \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}$$

which was invertible. So

$$EM = E \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix}$$

and then,

$$E = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}^{-1} = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \begin{bmatrix} 13 & 9 & 24 \\ 3 & 12 & 14 \\ 8 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

And this matrix worked for the entire message and hence we found the key, the encryption matrix, E.

Constructing an invertible matrix M from the message to calculate the encryption matrix makes perfect sense arithmetically. We know that

$$EM = C$$

so if M is invertible, we have,

$$E = E(MM^{-1}) = (EM)M^{-1} = CM^{-1}$$

and we are done. Let's also think about this in terms of vectors and transformations. We are trying to determine the transformation E by what the transformation does. That is, we know some vectors in the domain (the plaintext) and what those vectors get sent to in the range (the ciphertext). The columns of M are our domain vectors and the columns of C are where those vectors are mapped. If M is invertible, the Invertible Matrix Theorem tells us that the columns of M are linearly independent. Since there are n column vectors in M and the dimension of our domain space is n we also know that the columns of M form a basis to $\mathbb{A}^n$. We also know that any linear transformation is completely determined by what it does to a basis for the space. Hence we know that if we can find a set of linearly independent vectors from the plaintext message, and hence construct M, we will have enough information to completely determine E.

This also tells us that if we cannot find a set of linearly independent vectors from the plaintext then we do not have a basis for $\mathbb{A}^n$ and subsequently we will not be able to determine E. We may, in some cases, be able to determine a set of possibilities for E, as we did for the message HELP in the previous example. If we were to try to determine E from a non-basis set of vectors we would need to set up a system of equations and solve the system. Since the set is not a basis we would be guaranteed to have at least one free variable in the solution and hence a set of possible solutions for E, each of which would need to be tested on the plaintext to see which one, or ones, produced the corresponding ciphertext. One positive note here is that since we are working modulo 26 we will have only a finite number of possible matrices E to test, just like we had only four possibilities for the decryption of the message in the previous example.

Δ

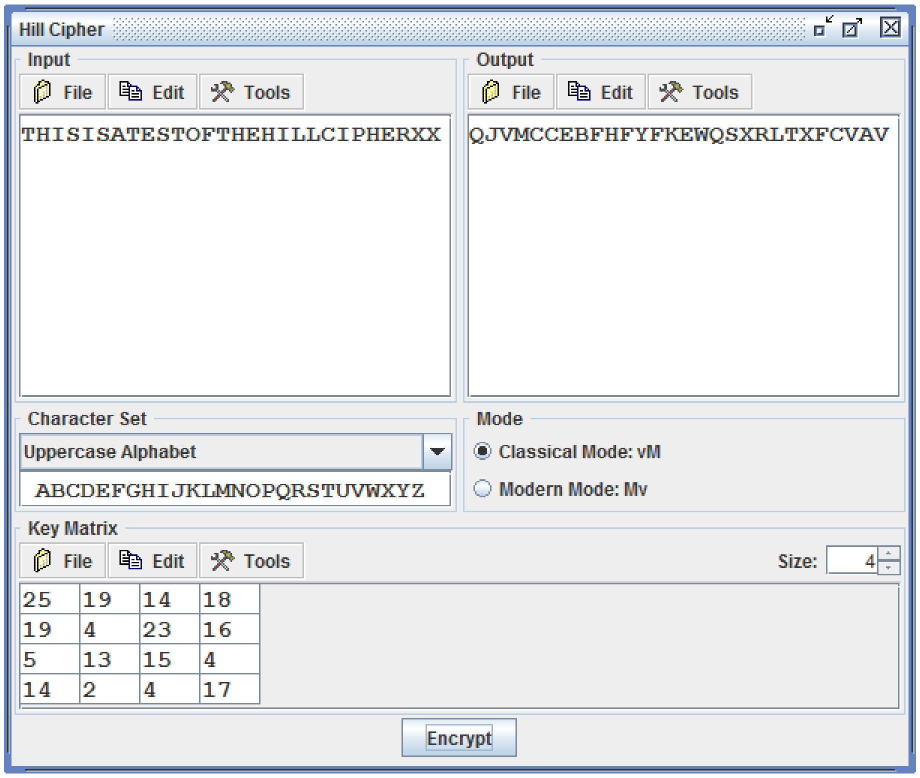
2.13.7 Cryptography Explorer

When working with the Cryptography Explorer program with a Hill cipher there are three tools you will need, the Hill Cipher tool, the Modular Matrix Calculator and the Integer

Calculator. In fact, unless you are going to find the inverses of the encryption matrices using the reduction tools or you are working with a non-invertible encryption matrix you will probably not need the Integer Calculator.

We will first go over the tools in general and then do an example at the end using them. For each of these tools you can find a more detailed description of the options in the appendix on the Cryptography Explorer program.

Hill Cipher Tool



Input: THISISATESTOFTHEHILLCIPHERXX

Output: 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 2.73: 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. Select the mode that you wish to use, either classical or modern. Classical computes $\mathbf{v}E = \mathbf{w}$ and modern computes $E\mathbf{v} = \mathbf{w}$.
4. 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. Select the mode that you wish to use, either classical or modern. Classical computes $\mathbf{v}E = \mathbf{w}$ and modern computes $E\mathbf{v} = \mathbf{w}$.
4. 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.

- To change the size of the matrix, there is a size selection to the right of the menu.
- 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 $n \times n$).
  2. Translate each of these blocks into a row (or column) vector $\mathbf{v}$. 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 (ABCDEFGHIJKLMNOPQRSTUVWXYZ) 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 $\mathbf{w} = \mathbf{v}M$, if you are using classical mode. If you are in modern mode it will apply the matrix on the left, that is $\mathbf{w} = M\mathbf{v}$.
  4. Translate $\mathbf{w}$ back to letters for the output.

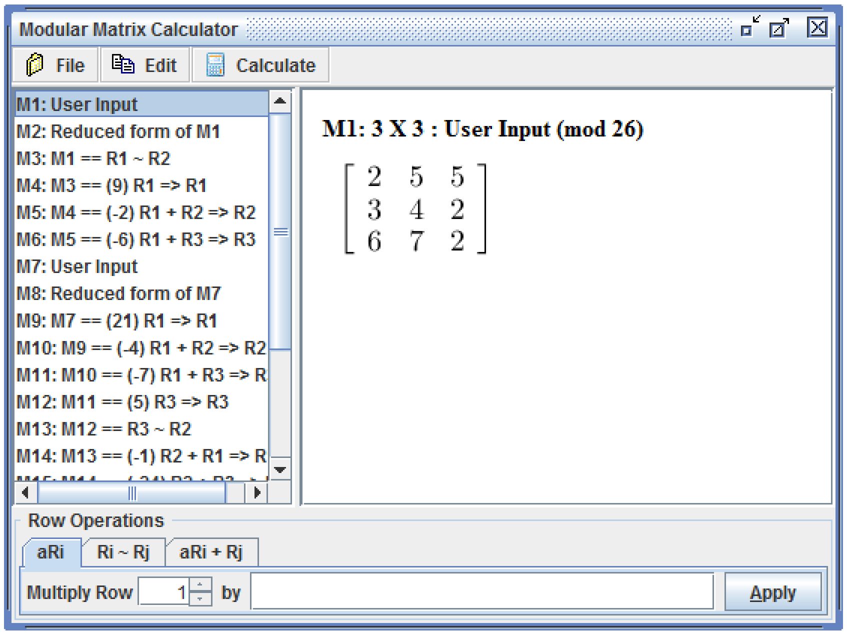
Notes

- 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.

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)

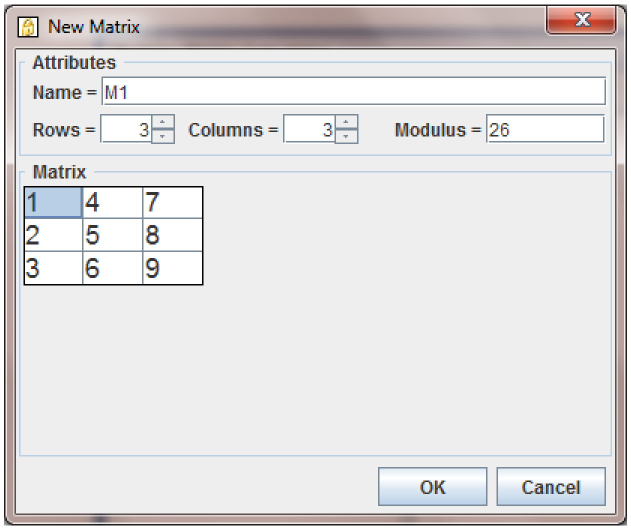
$$\begin{bmatrix} 2 & 5 & 5 \\ 3 & 4 & 2 \\ 6 & 7 & 2 \end{bmatrix}$$

Multiply Row 1 by

Figure 2.74: 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 2.75: 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 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

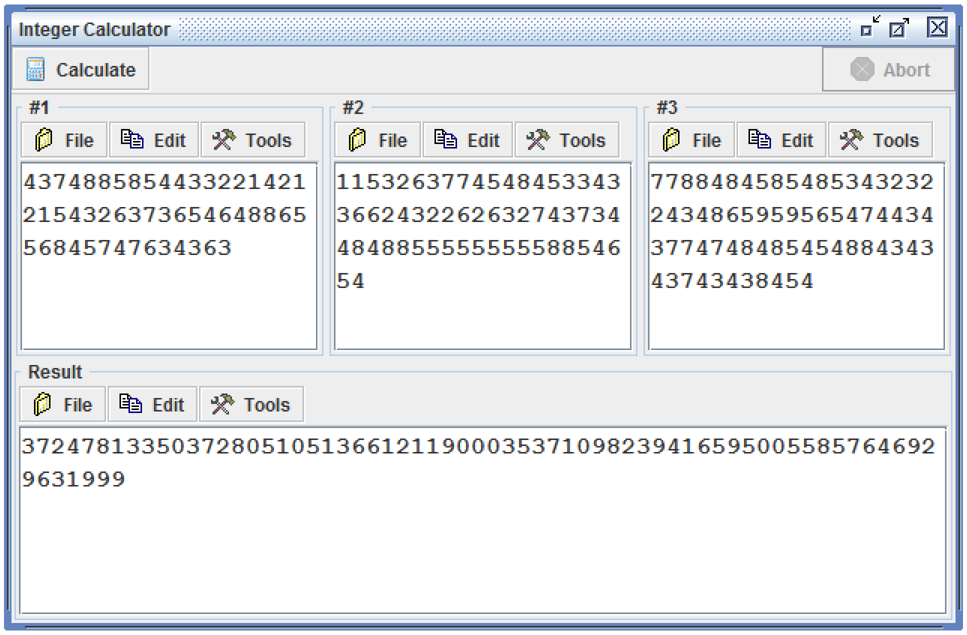
- When an operation is done on a matrix the original matrix is not altered, instead a new matrix is loaded into the workspace.
- As with any operation in linear algebra, if the matrix sizes are not compatible for the selected operation you will get an error.

Integer Calculator

The Integer Calculator is a simple infinite precision integer arithmetic tool. We will not go into all of the features of the Integer Calculator here since for Hill cipher exercises you will probably only be using it to find the inverse of a number modulo n, which we will calculate a couple ways. For a more detailed description of the features of this tool please see the section on the Integer Calculator in the appendix for the Cryptography Explorer program.

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.



```
#1
4374885854433221421215432637365464886556845747634363
```

Figure 2.76: Integer Calculator

Finding Inverses Modulo n

There are two main ways to find the inverse of a number modulo n, if the inverse exists. As an example, we will use the tool to calculate the inverse of 3 modulo 26, specifically $3^{-1} \pmod{26}$.

- In the Integer Calculator put 3 into input #1, -1 into input #2 and 26 into input #3. Then select Calculate > Modular Arithmetic > #1 ˆ #2 Mod(#3). At this point, the Result box should display 9, which is the inverse of 3 modulo 26.
- In the Integer Calculator put the following command into input #1, `powermod(3,-1,26)` and then select Calculate > Evaluate > Evaluate #1. At this point, the Result box should display 9, which is the inverse of 3 modulo 26.

Examples

Let's look at a couple examples of using the Cryptography Explorer tools.

Example 53: Say Alice wants to send Bob the message "Cryptography is cool!" using the hill cipher with encryption matrix,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

1. Open the Hill Cipher tool by selecting Ciphers > Hill from the main menu.

2. Copy or type Cryptography is cool! into the input box.
3. Use the tools for the input box to convert the message to uppercase, remove the white space, and remove the punctuation.
4. Select Modern Mode in the mode options.
5. In the Key Matrix, make sure that the size is set to 3, then input the above matrix E into the matrix grid. In the Tools menu for the Key Matrix there is an option for checking if the matrix is invertible, specifically Check for Inverse Matrix. Although the program will encrypt using non-invertible matrices, use this option to check that the matrix is in fact invertible, it should be.
6. Click the Encrypt key and the ciphertext ZSHLFGLKTVDUWAQBCG will appear in the output box.

Δ

Example 54: Now say that Bob receives the message ZSHLFGLKTVDUWAQBCG from Alice and knows that the encryption matrix was,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Bob knows that he needs the inverse of the matrix E, modulo 26. There are several ways to do this with the Cryptography Explorer program, we will use the easiest one here and then examine some alternatives in the next example.

1. Open up the Modular Matrix Calculator by selecting, Tools > Calculators > Modular Matrix Calculator from the main menu.
2. Select Edit > New Matrix from the tool's menu.
3. In the dialog box that appears, make sure that the rows and columns are set to 3 each, put in 26 for the modulus, and then input the matrix E into the grid. Then click OK. At this point the matrix will be loaded into the workspace of the matrix calculator.
4. Select Calculate > Invert from the menu and the inverse matrix (modulo 26) will be loaded into the workspace, you should have,

$$\begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$$

   Now we are ready to decrypt the message.
5. Open the Hill Cipher tool by selecting Ciphers > Hill from the main menu.

6. Copy or type the ciphertext ZSHLFGLKTVDUWAQBCG into the input box.

7. Select Modern Mode in the mode options.

8. In the Key Matrix, make sure that the size is set to 3, then input the above inverse matrix into the matrix grid. Note that you can copy and past the matrix from the matrix calculator by selecting Edit > Copy Matrix from the menu of the calculator, select the upper left cell of the grid in the Hill Cipher tool, and press Ctrl+V.

9. Click the Encrypt key and the plaintext CRYPTOGRAPHYISCOOL will appear in the output box.

Δ

Example 55: In the above example we used the Modular Matrix Calculator to find the inverse of the encryption matrix E. The easiest way to do this is by the Invert command from the menu system. So why would we want to use a different, and probably more difficult method? Depending on what course this section is being integrated into, the instructor may wish that you practice other calculation skills. For example, if this is being integrated into a course that does not assume that you know any linear algebra or matrix theory then most likely the instructor will have you calculate the inverse as we did above. On the other hand, if this is being integrated into a linear algebra class, or a class that assumes a knowledge of linear algebra, then the instructor may wish use this as an example of how the topics and techniques in linear algebra work when we are doing our calculations modulo 26 instead of over the real numbers.

Recall from linear algebra that if a square matrix, M, is invertible then it can be reduced to the identity matrix. Furthermore, the row operations that are done in reducing M to the identity will also convert the identity matrix to the inverse of M. So if we construct the $n \times 2n$ matrix that has M in the left half and the $n \times n$ identity in the right half, notationally, $[M|I]$, and reduce this matrix to reduced row echelon form we will get the matrix $[I|M^{-1}]$. So our decryption matrix will be the right side $n \times n$ matrix after the reduction.

The two methods we outline below use this idea, the first does the reduction to reduced row echelon form using a single command from the menu and the second uses the reduction interface that is built into the calculator to do each row reduction operation.

- Finding the Inverse using the Reduce Command:

  The exercise is to find the inverse (modulo 26) of the following matrix.

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

  1. Open the Modular Matrix Calculator and input the following 3×6 matrix with modulus 26,

$$\begin{bmatrix} 2 & 3 & 15 & 1 & 0 & 0 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 1 & 13 & 4 & 0 & 0 & 1 \end{bmatrix}$$

2. Select Calculate > Reduce from the tool's menu and you will get the following output,
$$\begin{bmatrix} 1 & 0 & 0 & 10 & 19 & 16 \\ 0 & 1 & 0 & 4 & 23 & 7 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$
This tells you that the inverse matrix is,
$$\begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$$

- Finding the Inverse using Matrix Reduction:

  The exercise is to find the inverse (modulo 26) of the following matrix. This time we will do the matrix reduction step by step using the Modular Matrix Calculator to help with the arithmetic.
$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

1. Open the Modular Matrix Calculator and input the following 3×6 matrix with modulus 26,
$$\begin{bmatrix} 2 & 3 & 15 & 1 & 0 & 0 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 1 & 13 & 4 & 0 & 0 & 1 \end{bmatrix}$$

2. Since we are going through the reduction process step by step we will want to use the row operations panel. Select Calculate > Show/Hide Row Operations Panel. This will toggle the Row Operations Panel at the bottom. The Row Operations Panel has three tabs, each for one of the three elementary row operations. The first tab multiplies a row by a constant value, the second tab interchanges two rows and the third multiplies one row by a constant and adds it to another row. All calculations are done modulo the modulus of the matrix that is being operated on.

   The row selectors are numeric selection boxes that are restricted to the rows in the matrix and the other input boxes are general text input boxes but they are expecting an integer input.

3. Since 2 is not invertible modulo 26 we need to either interchange rows 1 and 2 or rows 1 and 3, since the first entry in row 3 is a 1 we will interchange rows 1 and 3. Select the second tab, `Ri ~ Rj`, set the numeric selectors to 1 and 3 respectively, and click the Apply button. Now the displayed matrix is,
$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 2 & 3 & 15 & 1 & 0 & 0 \end{bmatrix}$$

4. Now we will "zero out" the first column under the 1. Select the third tab `aRi + Rj`, select the rows and inputs so that the display reads,

   ```
   Multiply Row 1 by -5 and add to row 2
   ```

   then click Apply. Now select the rows and inputs so that the display reads,

   ```
   Multiply Row 1 by -2 and add to row 3
   ```

   then click Apply. Now the displayed matrix is,

   $$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 0 & 21 & 18 & 0 & 1 & 21 \\ 0 & 3 & 7 & 1 & 0 & 24 \end{bmatrix}$$

   Note that the program automatically converted the -5 and -2 to 21 and 24 respectively, since our calculations are done modulo 26. We could have used 21 and 24 in place of -5 and -2.

5. At this point we want a 1 in the second row and second column. Since 21 is invertible modulo 26, $\gcd(21, 26) = 1$, we can use the integer calculator to calculate $21^{-1} \pmod{26} \equiv 5$. Just use either method described above to do this calculation. Now select the first tab `aRi`, set the row to 2 and input 5 into the multiplier box, and then click Apply. The displayed matrix is now,

   $$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 3 & 7 & 1 & 0 & 24 \end{bmatrix}$$

6. Now we will "zero out" the second column, other than the 1. Select the third tab `aRi + Rj`, select the rows and inputs so that the display reads,

   ```
   Multiply Row 2 by -13 and add to row 1
   ```

   then click Apply. Now select the rows and inputs so that the display reads,

   ```
   Multiply Row 2 by -3 and add to row 3
   ```

   then click Apply. Now the displayed matrix is,

   $$\begin{bmatrix} 1 & 0 & 4 & 0 & 13 & 14 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 0 & 23 & 1 & 11 & 21 \end{bmatrix}$$

   As with the previous use of this option, we could have used 13 and 23 in place of -13 and -3 respectively.

7. At this point we want a 1 in the third row and third column. Since 23 is invertible modulo 26, $\gcd(23, 26) = 1$, we can use the integer calculator to calculate $23^{-1} \pmod{26} \equiv 17$. Just use either method described above to do this calculation.

Now select the first tab `aRi`, set the row to 3 and input 17 into the multiplier box, and then click Apply. The displayed matrix is now,

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 13 & 14 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$

8. Now we will "zero out" the third column, other than the 1. Select the third tab `aRi + Rj`, select the rows and inputs so that the display reads,

   ```
   Multiply Row 3 by -4 and add to row 1
   ```

   then click Apply. Now select the rows and inputs so that the display reads,

   ```
   Multiply Row 3 by -12 and add to row 2
   ```

   then click Apply. Now the displayed matrix is,

$$\begin{bmatrix} 1 & 0 & 0 & 10 & 19 & 16 \\ 0 & 1 & 0 & 4 & 23 & 7 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$

   As before, we could have used 22 and 14 in place of -4 and -12 respectively. We have finished the reduction process to the form $[I|M]$ which tells us that the inverse matrix (modulo 26) is

$$\begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$$

   The entire workspace for this process is as follows,

```
M1: 3 X 6 : User Input (mod 26)
```

$$\begin{bmatrix} 2 & 3 & 15 & 1 & 0 & 0 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 1 & 13 & 4 & 0 & 0 & 1 \end{bmatrix}$$

```
M2: 3 X 6 : M1 == R1 ~ R3  (mod 26)
```

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 2 & 3 & 15 & 1 & 0 & 0 \end{bmatrix}$$

```
M3: 3 X 6 : M2 == (-5) R1 + R2 => R2  (mod 26)
```

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 0 & 21 & 18 & 0 & 1 & 21 \\ 2 & 3 & 15 & 1 & 0 & 0 \end{bmatrix}$$

```
M4: 3 X 6 : M3 == (-2) R1 + R3 => R3  (mod 26)
```

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 0 & 21 & 18 & 0 & 1 & 21 \\ 0 & 3 & 7 & 1 & 0 & 24 \end{bmatrix}$$

```
M5: 3 X 6 : M4 == (5) R2 => R2  (mod 26)
```

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 3 & 7 & 1 & 0 & 24 \end{bmatrix}$$

```
M6: 3 X 6 : M5 == (-13) R2 + R1 => R1  (mod 26)
```

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 13 & 14 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 3 & 7 & 1 & 0 & 24 \end{bmatrix}$$

```
M7: 3 X 6 : M6 == (-3) R2 + R3 => R3  (mod 26)
```

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 13 & 14 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 0 & 23 & 1 & 11 & 21 \end{bmatrix}$$

```
M8: 3 X 6 : M7 == (17) R3 => R3  (mod 26)
```

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 13 & 14 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$

```
M9: 3 X 6 : M8 == (-4) R3 + R1 => R1  (mod 26)
```

$$\begin{bmatrix} 1 & 0 & 0 & 10 & 19 & 16 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$

```
M10: 3 X 6 : M9 == (-12) R3 + R2 => R2  (mod 26)
```

$$\begin{bmatrix} 1 & 0 & 0 & 10 & 19 & 16 \\ 0 & 1 & 0 & 4 & 23 & 7 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$

Δ

The Cryptography Explorer program can also help with the breaking of a Hill Cipher. We will redo the example from the section on breaking the Hill cipher.

Example 56: Eve has intercepted the encrypted message ZSHLFGLKTVDUWAQBCG and from other espionage knows that this message was CRYPTOGRAPHYISCOOL. She also knows that other messages sent that day between Alice and Bob are using the same key and she wishes to decrypt them as well, but she has no other information about these other messages.

Since the message size 18 she knows that that block size must be 2, 3, 6, 9 or 18, and since she has only 18 characters to work with her only hope is that the block size is either 2 or 3. If both of these fail it is back to other espionage.

Let's start with a block size of 2. Since,

CRYPTOGRAPHYISCOOL — 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

encrypts as

ZSHLFGLKTVDUWAQBCG — 25 18 7 11 5 6 11 10 19 21 3 20 22 0 16 1 2 6

we know that

$$\begin{bmatrix} 2 \\ 17 \end{bmatrix} \longrightarrow \begin{bmatrix} 25 \\ 18 \end{bmatrix} \qquad \begin{bmatrix} 24 \\ 15 \end{bmatrix} \longrightarrow \begin{bmatrix} 7 \\ 11 \end{bmatrix} \qquad \begin{bmatrix} 19 \\ 14 \end{bmatrix} \longrightarrow \begin{bmatrix} 5 \\ 6 \end{bmatrix} \cdots$$

So we want to construct a 2×2 matrix out of the plaintext blocks that is invertible modulo 26. If we take the first two blocks 2 17 and 24 15 and build a matrix from them

$$M = \begin{bmatrix} 2 & 24 \\ 17 & 15 \end{bmatrix}$$

then $\det(M) = -378$ which is not relatively prime to 26, so M is not invertible. Actually, we should have seen this coming with what we know about determinants, notice that the first row has all even numbers in it and 2 is a factor of 26.

If we move on to the next block, 19 14 and keep the first block our M matrix becomes,

$$M = \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}$$

then $\det(M) = -295$ which is 17 when we take it modulo 26. Since 17 has an inverse modulo 26 we are in business. Our equation becomes,

$$E \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix} = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix}$$

So

$$E = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}^{-1} = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \begin{bmatrix} 10 & 5 \\ 25 & 20 \end{bmatrix} = \begin{bmatrix} 11 & 17 \\ 18 & 2 \end{bmatrix}$$

To see if this works we test it on our plaintext message CRYPTOGRAPHYISCOOL.

$$\begin{aligned} C = EM &= \begin{bmatrix} 11 & 17 \\ 18 & 2 \end{bmatrix} \begin{bmatrix} 2 & 24 & 19 & 6 & 0 & 7 & 8 & 2 & 14 \\ 17 & 15 & 14 & 17 & 15 & 24 & 18 & 14 & 11 \end{bmatrix} \\ &= \begin{bmatrix} 25 & 25 & 5 & 17 & 21 & 17 & 4 & 0 & 3 \\ 18 & 20 & 6 & 12 & 4 & 18 & 24 & 12 & 14 \end{bmatrix} \end{aligned}$$

As we can see, columns 1 and 3 encrypt correctly, they should since those were the ones we used, but the rest of the encryption is incorrect. Conclusion, the block size is not 2.

To do the above calculations using the Modular Matrix Calculator do the following,

1. Input the following three matrices, each with modulus 26,

$$\begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix} \qquad \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 2 & 24 & 19 & 6 & 0 & 7 & 8 & 2 & 14 \\ 17 & 15 & 14 & 17 & 15 & 24 & 18 & 14 & 11 \end{bmatrix}$$

   We will assume that they are input in this order so that they correspond to matrices M1, M2 and M3 respectively.

2. To calculate the possible encryption matrix E, highlight matrix M1, select Calculate > Invert, now M4 is the inverse of M1 (modulo 26). Select Calculate > Multiply from the menu, at this point a dialog box will appear, select matrix M2 for A and matrix M4 for B using the drop-down selection boxes and click OK. The matrix M5 should be E.

3. To check the validity of the encryption matrix, select Calculate > Multiply from the menu. Select matrix M5 for A and matrix M3 for B using the drop-down selection boxes and click OK. The matrix M6 should be $C = EM$ from above. Your conclusion would be the same as above, the block size is not 2.

So we move on to block size 3. If it is then there is a 3×3 matrix E with

$$E \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$$

As before, we want to select three columns from our message matrix that will produce an invertible 3×3 matrix. Looking at the last row, all entries except for 11 are even, so the last column must be used. Since the last column first row is even we must have at least one odd number in the first row of the columns we use. So we could not select the remaining two columns from columns 1, 3, and 5. Furthermore, column 5 is all even so selecting it would be pointless. So in our selection we must have column 6 and at least one of columns 2 and 4. We will try columns 1, 2, and 6. This gives,

$$M = \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}$$

Since $\det(M) = -791 \equiv 15 \pmod{26}$, we know that M is invertible. So

$$EM = E \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix}$$

and then,

$$E = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}^{-1} = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \begin{bmatrix} 13 & 9 & 24 \\ 3 & 12 & 14 \\ 8 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Now, we know this is correct but Eve would still need to check her possible solution, doing so she would get,

$$\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix} \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$$

which checks with the ciphertext and she has found the encryption matrix, E.

To do the above calculations using the Modular Matrix Calculator do the following,

1. Input the following three matrices, each with modulus 26,

$$\begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix} \quad \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix}$$

   We will assume that they are input in this order so that they correspond to matrices M1, M2 and M3 respectively. If you have matrices in the workspace, select File > New before inputting these three matrices.

2. To calculate the possible encryption matrix E, highlight matrix M1, select Calculate > Invert, now M4 is the inverse of M1 (modulo 26). Select Calculate > Multiply from the menu, at this point a dialog box will appear, select matrix M2 for A and matrix M4 for B using the drop-down selection boxes and click OK. The matrix M5 should be E.

3. To check the validity of the encryption matrix, select Calculate > Multiply from the menu. Select matrix M5 for A and matrix M3 for B using the drop-down selection boxes and click OK. The matrix M6 should be $C = EM$ from above. The ciphertext matrix checks out, so the block size was three and the encryption matrix is,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Δ

2.13.8 Mathematica

When using a general computer algebra system you will follow the same procedure as you would with the Cryptography Explorer program except that the operations will be done with the computer algebra system commands instead of input dialog boxes and menu selections.

There is an entire section on vectors and matrices in the appendix on Mathematica, we suggest that you read over that section before continuing.

For the Hill cipher the main things you need to be concerned with are, defining a matrix, multiplying two matrices, and inverting a square matrix. We will quickly recap these below.

Defining a Matrix and a Vector

A vector is simply a list and a matrix is a list of lists, where 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 discuss matrix operations in detail in the Mathematica appendix, here we just need to know about multiplication. The multiplication operator for matrices in Mathematica is the period. So to multiply matrices `m` and `n` the syntax is `m . n`.

Another interesting feature with Mathematica is that it will treat a vector as both a row vector and a column vector, it simply uses the form that is needed. 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}
```

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}}
```

Which assignment you use will depend on what you intend to do with the matrices. In most cases you will probably want the immediate assignment = instead of the delayed assignment of `:=`.

Another thing you should be aware of is that when you assign a matrix to a variable or assign one matrix to another do not use the `MatrixForm` command or pipe in the assignment statement. The `MatrixForm` command alters the form of the data so that doing matrix operations becomes impossible. So, do the assignment in one command and the `MatrixForm` in another.

Joining Matrices

Depending on how you choose to do some of the matrix operations you may want to join two matrices together or join a matrix and a vector, that is, augment a matrix with a vector. The Mathematica command for joining matrices is `Join`.

```
In[1]:= m = {{1, 2, 1}, {3, 2 - 1, 2}, {4, -2, 1}}

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

    In[2]:= m // MatrixForm

Out[2]//MatrixForm=
          ( 1   2  1 )
          ( 3   1  2 )
          ( 4  -2  1 )
```

Mathematica as an `IdentityMatrix` command that will produce an identity matrix of any given size.

```
In[3]:= id = IdentityMatrix [3]

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

    In[4]:= id // MatrixForm

Out[4]//MatrixForm=
          ( 1  0  0 )
          ( 0  1  0 )
          ( 0  0  1 )
```

The `Join` command takes either two or three arguments. To join two matrices vertically, that is, one over the other simply use `Join[m, n]` where `m` and `n` are matrices.

```
    In[5]:= Join [m , id] // MatrixForm

Out[5]//MatrixForm=
          ( 1   2  1 )
          ( 3   1  2 )
          ( 4  -2  1 )
          ( 1   0  0 )
          ( 0   1  0 )
          ( 0   0  1 )
```

To join two matrices horizontally, that is, one beside the other simply use `Join[m, n, 2]` where `m` and `n` are matrices.

```
In[6]:= d = Join[m, id, 2]

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

In[7]:= d // MatrixForm

Out[7]//MatrixForm=
```

$$\begin{pmatrix} 1 & 2 & 1 & 1 & 0 & 0 \\ 3 & 1 & 2 & 0 & 1 & 0 \\ 4 & -2 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Matrix Arithmetic

For the Hill Cipher, the main matrix operations you need to concentrate on are matrix multiplication and matrix inverses. For the Hill cipher you are really interested in these operations done modulo an integer n, we will discuss modular operations later in this section. For now we will simply look at the commands without a modulus. As we pointed out above, the period is the multiplication operator for matrices. To find an inverse to a square matrix, if it exists, is the `Inverse` command. We can also find the determinant of a matrix with the `Det` command. You can find a more detailed description of other arithmetic operations on matrices in the Mathematica appendix.

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[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}}
```

Finding the inverse of a matrix can be done 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]= $\left\{\left\{\frac{10}{9}, -\frac{2}{9}, \frac{1}{9}\right\}, \left\{-\frac{5}{3}, \frac{1}{3}, \frac{1}{3}\right\}, \left\{-\frac{1}{9}, \frac{2}{9}, -\frac{1}{9}\right\}\right\}$

Matrix Reduction

Depending how you go about the calculations, you may need to reduce a matrix. 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 reduced 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}}
```

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. Inverses require a different method which we will discuss below.

```
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}}
```

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 )
```

Examples

Let's look at a couple example of using Mathematica.

Example 57: Say Alice wants to send Bob the message "Cryptography is cool!" using the hill cipher with encryption matrix,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

If we convert "Cryptography is cool!" to matrix form we get,

$$M = \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix}$$

Since Mathematica reserves the character `E` for the natural base e we need to use another character.

```
In[1]:= H = {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

Out[1]= {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

In[2]:= H // MatrixForm

Out[2]//MatrixForm=
         ( 2   3  15 )
         ( 5   8  12 )
         ( 1  13   4 )
```

Put the message matrix into Mathematica, assigned to the variable M,

```
In[3]:= M = {{2, 15, 6, 15, 8, 14}, {17, 19, 17, 7, 18, 14},
          {24, 14, 0, 24, 2, 11}}

Out[3]= {{2, 15, 6, 15, 8, 14},
          {17, 19, 17, 7, 18, 14}, {24, 14, 0, 24, 2, 11}}
```

```
In[4]:= M // MatrixForm

Out[4]//MatrixForm=
        ( 2  15   6  15   8  14 )
        (17  19  17   7  18  14 )
        (24  14   0  24   2  11 )
```

Multiply the two matrices, modulo 26,

```
In[5]:= CT = Mod[H.M, 26]

Out[5]= {{25, 11, 11, 21, 22, 1}, {18, 5, 10, 3, 0, 2}, {7, 6, 19, 20, 16, 6}}


In[6]:= CT // MatrixForm

Out[6]//MatrixForm=
        (25  11  11  21  22  1 )
        (18   5  10   3   0  2 )
        ( 7   6  19  20  16  6 )
```

Then reading the columns we have the numeric list, 25 18 7 11 5 6 11 10 19 21 3 20 22 0 16 1 2 6 which converts to the ciphertext ZSHLFGLKTVDUWAQBCG. Δ

Example 58: Now say that Bob receives the message ZSHLFGLKTVDUWAQBCG from Alice and knows that the encryption matrix was,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Bob knows that he needs the inverse of the matrix E, modulo 26. There are several ways to do this with Mathematica, we will use the easiest one here and then examine some alternatives in the next example. First put in the encryption matrix and the ciphertext matrix.

```
In[1]:= H = {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

Out[1]= {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

In[2]:= H // MatrixForm

Out[2]//MatrixForm=
        (2   3  15 )
        (5   8  12 )
        (1  13   4 )


In[3]:= CT = {{25, 11, 11, 21, 22, 1}, {18, 5, 10, 3, 0, 2},
           {7, 6, 19, 20, 16, 6}}

Out[3]= {{25, 11, 11, 21, 22, 1}, {18, 5, 10, 3, 0, 2}, {7, 6, 19, 20, 16, 6}}
```

```
In[4]:= CT // MatrixForm

Out[4]//MatrixForm=
  25  11  11  21  22  1
  18   5  10   3   0  2
   7   6  19  20  16  6
```

Now find the inverse of the encryption matrix modulo 26, this is the decryption matrix.

```
In[5]:= HI = Inverse[H, Modulus → 26]

Out[5]= {{10, 19, 16}, {4, 23, 7}, {17, 5, 19}}

In[6]:= HI // MatrixForm

Out[6]//MatrixForm=
  10  19  16
   4  23   7
  17   5  19
```

Multiply the decryption matrix by the ciphertext matrix to retrieve the message.

```
In[7]:= M = Mod[HI.CT, 26]

Out[7]= {{2, 15, 6, 15, 8, 14},
          {17, 19, 17, 7, 18, 14}, {24, 14, 0, 24, 2, 11}}

In[8]:= M // MatrixForm

Out[8]//MatrixForm=
   2  15   6  15   8  14
  17  19  17   7  18  14
  24  14   0  24   2  11
```

Reading down the columns we get the numeric list, 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11, which converts to the plaintext message CRYPTOGRAPHYISCOOL. Δ

Example 59: The easiest way to find an inverse matrix in Mathematica is with the `Inverse` command. So why would we want to use a different, and probably more difficult method? Depending on what course this section is being integrated into, the instructor may wish that you practice other calculation skills. For example, if this is being integrated into a course that does not assume that you know any linear algebra or matrix theory then most likely the instructor will have you calculate the inverse as we did above. On the other hand, if this is being integrated into a linear algebra class, or a class that assumes a knowledge of linear algebra, then the instructor may wish use this as an example of how the topics and techniques in linear algebra work when we are doing our calculations modulo 26 instead of over the real numbers.

Recall from linear algebra that if a square matrix, M, is invertible then it can be reduced to the identity matrix. Furthermore, the row operations that are done in reducing M to the identity will also convert the identity matrix to the inverse of M. So if we construct the $n \times 2n$ matrix that has M in the left half and the $n \times n$ identity in the right half, notationally, $[M|I]$, and reduce this matrix to reduced row echelon form we will get the matrix $[I|M^{-1}]$. So our decryption matrix will be the right side $n \times n$ matrix after the reduction.

To apply this process in Mathematica, we will do the following. The exercise is to find the inverse (modulo 26) of the following matrix using a reduction approach.

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

First, we input the encryption matrix,

```
In[1]:= H = {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

Out[1]= {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

    In[2]:= H // MatrixForm

Out[2]//MatrixForm=
        ( 2   3  15 )
        ( 5   8  12 )
        ( 1  13   4 )
```

Create a 3×3 identity matrix,

```
In[3]:= id = IdentityMatrix[3]

Out[3]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

Join the encryption matrix to the identity matrix,

```
In[4]:= A = Join[H, id, 2]

Out[4]= {{2, 3, 15, 1, 0, 0}, {5, 8, 12, 0, 1, 0}, {1, 13, 4, 0, 0, 1}}

    In[5]:= A // MatrixForm

Out[5]//MatrixForm=
        ( 2   3  15  1  0  0 )
        ( 5   8  12  0  1  0 )
        ( 1  13   4  0  0  1 )
```

Now we would like to reduce this matrix using modulo 26 operations. Unfortunately Mathematica does not allow a composite number to be used as a modulus in the `RowReduce` command. So we will work around this.

```
In[6]:= RowReduce[A, Modulus → 26]

        RowReduce::nmod : {{2, 3, 15, 1, 0, 0}, {0, 1, 1, 21, 2, 0}, {0, 0, 22, 10, 6, 2}} is not valid modulo 26. >>

Out[6]= RowReduce[{{2, 3, 15, 1, 0, 0},
           {5, 8, 12, 0, 1, 0}, {1, 13, 4, 0, 0, 1}}, Modulus → 26]
```

First we do a standard row reduction,

```
In[7]:= RA = RowReduce[A]
```

Out[7]= $\left\{\left\{1, 0, 0, -\frac{124}{583}, \frac{183}{583}, -\frac{84}{583}\right\}, \left\{0, 1, 0, -\frac{8}{583}, -\frac{7}{583}, \frac{51}{583}\right\}, \left\{0, 0, 1, \frac{57}{583}, -\frac{23}{583}, \frac{1}{583}\right\}\right\}$

```
In[8]:= RA // MatrixForm
```

Out[8]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 & -\frac{124}{583} & \frac{183}{583} & -\frac{84}{583} \\ 0 & 1 & 0 & -\frac{8}{583} & -\frac{7}{583} & \frac{51}{583} \\ 0 & 0 & 1 & \frac{57}{583} & -\frac{23}{583} & \frac{1}{583} \end{pmatrix}$$

Extract the last three columns,

```
In[9]:= HIr = RA[[All, 4 ;; 6]]
```

Out[9]= $\left\{\left\{-\frac{124}{583}, \frac{183}{583}, -\frac{84}{583}\right\}, \left\{-\frac{8}{583}, -\frac{7}{583}, \frac{51}{583}\right\}, \left\{\frac{57}{583}, -\frac{23}{583}, \frac{1}{583}\right\}\right\}$

```
In[10]:= HIr // MatrixForm
```

Out[10]//MatrixForm=

$$\begin{pmatrix} -\frac{124}{583} & \frac{183}{583} & -\frac{84}{583} \\ -\frac{8}{583} & -\frac{7}{583} & \frac{51}{583} \\ \frac{57}{583} & -\frac{23}{583} & \frac{1}{583} \end{pmatrix}$$

Since $\gcd(583, 26) = 1$ we know that each of the entries in the above matrix have a value modulo 26. All we need to do is clear out the denominator by multiplying by 583, then multiply by the inverse of 583 modulo 26, which must exist, and then reduce the entries of the matrix modulo 26. First find the inverse of 583 modulo 26 using the `PowerMod` function.

```
In[11]:= PowerMod[583, -1, 26]

Out[11]= 19
```

Multiply the matrix by this inverse, 19, and by 583, and finally reduce the matrix modulo 26.

```
In[12]:= HI = Mod[HIr * 19 * 583, 26]

Out[12]= {{10, 19, 16}, {4, 23, 7}, {17, 5, 19}}

In[13]:= HI // MatrixForm

Out[13]//MatrixForm=
```

$$\begin{pmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{pmatrix}$$

Δ

Mathematica can also help with the breaking of a Hill Cipher. We will redo the example from the section on breaking the Hill cipher.

Example 60: Eve has intercepted the encrypted message ZSHLFGLKTVDUWAQBCG and from other espionage knows that this message was CRYPTOGRAPHYISCOOL. She also knows that other messages sent that day between Alice and Bob are using the same key and she wishes to decrypt them as well, but she has no other information about these other messages.

Since the message size 18 she knows that that block size must be 2, 3, 6, 9 or 18, and since she has only 18 characters to work with her only hope is that the block size is either 2 or 3. If both of these fail it is back to other espionage.

Let's start with a block size of 2. Since,

CRYPTOGRAPHYISCOOL — 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

encrypts as

ZSHLFGLKTVDUWAQBCG — 25 18 7 11 5 6 11 10 19 21 3 20 22 0 16 1 2 6

we know that

$$\begin{bmatrix} 2 \\ 17 \end{bmatrix} \longrightarrow \begin{bmatrix} 25 \\ 18 \end{bmatrix} \qquad \begin{bmatrix} 24 \\ 15 \end{bmatrix} \longrightarrow \begin{bmatrix} 7 \\ 11 \end{bmatrix} \qquad \begin{bmatrix} 19 \\ 14 \end{bmatrix} \longrightarrow \begin{bmatrix} 5 \\ 6 \end{bmatrix} \cdots$$

So we want to construct a 2×2 matrix out of the plaintext blocks that is invertible modulo 26. If we take the first two blocks 2 17 and 24 15 and build a matrix from them

$$M = \begin{bmatrix} 2 & 24 \\ 17 & 15 \end{bmatrix}$$

then $\det(M) = -378$ which is not relatively prime to 26, so M is not invertible. Actually, we should have seen this coming with what we know about determinants, notice that the first row has all even numbers in it and 2 is a factor of 26.

If we move on to the next block, 19 14 and keep the first block our M matrix becomes,

$$M = \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}$$

then $\det(M) = -295$ which is 17 when we take it modulo 26. Since 17 has an inverse modulo 26 we are in business. Our equation becomes,

$$E \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix} = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix}$$

So

$$E = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}^{-1} = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \begin{bmatrix} 10 & 5 \\ 25 & 20 \end{bmatrix} = \begin{bmatrix} 11 & 17 \\ 18 & 2 \end{bmatrix}$$

To see if this works we test it on our plaintext message CRYPTOGRAPHYISCOOL.

$$\begin{aligned} C = EM &= \begin{bmatrix} 11 & 17 \\ 18 & 2 \end{bmatrix} \begin{bmatrix} 2 & 24 & 19 & 6 & 0 & 7 & 8 & 2 & 14 \\ 17 & 15 & 14 & 17 & 15 & 24 & 18 & 14 & 11 \end{bmatrix} \\ &= \begin{bmatrix} 25 & 25 & 5 & 17 & 21 & 17 & 4 & 0 & 3 \\ 18 & 20 & 6 & 12 & 4 & 18 & 24 & 12 & 14 \end{bmatrix} \end{aligned}$$

As we can see, columns 1 and 3 encrypt correctly, they should since those were the ones we used, but the rest of the encryption is incorrect. Conclusion, the block size is not 2.

To do the above calculations using Mathematica do the following. Input the following three matrices,

$$\begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix} \qquad \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 2 & 24 & 19 & 6 & 0 & 7 & 8 & 2 & 14 \\ 17 & 15 & 14 & 17 & 15 & 24 & 18 & 14 & 11 \end{bmatrix}$$

```
In[1]:= A = {{2, 19}, {17, 14}}

Out[1]= {{2, 19}, {17, 14}}

In[2]:= B = {{25, 5}, {18, 6}}

Out[2]= {{25, 5}, {18, 6}}

In[3]:= CT = {{2, 24, 19, 6, 0, 7, 8, 2, 14},
           {17, 15, 14, 17, 15, 24, 18, 14, 11}}

Out[3]= {{2, 24, 19, 6, 0, 7, 8, 2, 14}, {17, 15, 14, 17, 15, 24, 18, 14, 11}}
```

Find the inverse of matrix A modulo 26.

```
In[4]:= AI = Inverse[A, Modulus → 26]

Out[4]= {{10, 5}, {25, 20}}
```

Multiply this matrix on the right with matrix B, and take the result modulo 26, to get the possible encryption matrix.

```
In[5]:= H = Mod[B.AI, 26]

Out[5]= {{11, 17}, {18, 2}}
```

Finally, multiply this resulting matrix on the right with matrix B, and take the result modulo 26, to get the possible encryption matrix.

```
In[6]:= M = Mod[H.CT, 26]

Out[6]= {{25, 25, 5, 17, 21, 17, 4, 0, 3}, {18, 20, 6, 12, 4, 18, 24, 12, 14}}
```

The above calculation was to check the validity of the encryption matrix. If we convert the result to ciphertext letters we get, ZSZUFGRMVERSEYAMDO which is not our ciphertext of ZSHLFGLKTVDUWAQBCG. Your conclusion would be the same as above, the block size is not 2.

So we move on to block size 3. If it is, then there is a 3×3 matrix E with

$$E \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$$

As before, we want to select three columns from our message matrix that will produce an invertible 3×3 matrix. Looking at the last row, all entries except for 11 are even, so the last column must be used. Since the last column first row is even we must have at least one odd number in the first row of the columns we use. So we could not select the remaining two columns from columns 1, 3, and 5. Furthermore, column 5 is all even so selecting it would be pointless. So in our selection we must have column 6 and at least one of columns 2 and 4. We will try columns 1, 2, and 6. This gives,

$$M = \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}$$

Since $\det(M) = -791 \equiv 15 \pmod{26}$, we know that M is invertible. So

$$EM = E \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix}$$

and then,

$$E = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}^{-1} = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \begin{bmatrix} 13 & 9 & 24 \\ 3 & 12 & 14 \\ 8 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Now, we know this is correct but Eve would still need to check her possible solution, doing so she would get,

$$\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix} \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$$

which checks with the ciphertext and she has found the encryption matrix, E.

To do the above calculations using Mathematica do the following.

Input the following three matrices,

$$\begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix} \qquad \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix}$$

```
In[1]:= A = {{2, 15, 14}, {17, 19, 14}, {24, 14, 11}}

Out[1]= {{2, 15, 14}, {17, 19, 14}, {24, 14, 11}}

In[2]:= B = {{25, 11, 1}, {18, 5, 2}, {7, 6, 6}}

Out[2]= {{25, 11, 1}, {18, 5, 2}, {7, 6, 6}}

In[3]:= CT = {{2, 15, 6, 15, 8, 14}, {17, 19, 17, 7, 18, 14},
          {24, 14, 0, 24, 2, 11}}

Out[3]= {{2, 15, 6, 15, 8, 14},
          {17, 19, 17, 7, 18, 14}, {24, 14, 0, 24, 2, 11}}
```

Find the inverse of matrix A modulo 26.

```
In[4]:= AI = Inverse[A, Modulus → 26]

Out[4]= {{13, 9, 24}, {3, 12, 14}, {8, 10, 15}}
```

Multiply this matrix on the right with matrix B, and take the result modulo 26, to get the possible encryption matrix.

```
In[5]:= H = Mod[B.AI, 26]

Out[5]= {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}
```

Finally, multiply this resulting matrix on the right with matrix B, and take the result modulo 26, to get the possible encryption matrix.

```
In[6]:= M = Mod[H.CT, 26]

Out[6]= {{25, 11, 11, 21, 22, 1}, {18, 5, 10, 3, 0, 2}, {7, 6, 19, 20, 16, 6}}
```

The above calculation was to check the validity of the encryption matrix. If we convert the result to ciphertext letters we get, ZSHLFGLKTVDUWAQBCG which is our ciphertext of ZSHLFGLKTVDUWAQBCG. The ciphertext matrix checks out, so the block size was three and the encryption matrix is,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Δ

2.13.9 Maxima

When using a general computer algebra system you will follow the same procedure as you would with the Cryptography Explorer program except that the operations will be done with the computer algebra system commands instead of input dialog boxes and menu selections.

There is an entire section on vectors and matrices in the appendix on Maxima, we suggest that you read over that section before continuing.

For the Hill cipher the main things you need to be concerned with are, defining a matrix, multiplying two matrices, and inverting a square matrix. We will quickly recap these below.

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")`.

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.

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 `^` but rather `^^`. 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) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

```
(%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) A+M;
```

(%o5) $\begin{bmatrix}2 & 2 & 4\\ 6 & 6 & 11\\ 10 & 10 & 9\end{bmatrix}$

```
(%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) $\begin{bmatrix} 1 & 0 & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{5} & \frac{5}{6} \\ \frac{3}{7} & \frac{1}{4} & 0 \end{bmatrix}$

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) $\begin{bmatrix}1 & 0 & -1 & 0 & -1\\0 & 1 & 2 & 0 & 1\\0 & 0 & 0 & 1 & 1\end{bmatrix}$

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,

$$A^{-1} = \frac{1}{\det(A)}\operatorname{adj}(A)$$

where $\operatorname{adj}(A)$ 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) $$\begin{bmatrix}1&0&0&0&7\\0&1&0&0&12\\0&0&1&0&6\\0&0&0&1&1\\0&0&0&0&0\end{bmatrix}$$

Examples

Let's look at a couple examples using Maxima.

Example 61: Say Alice wants to send Bob the message "Cryptography is cool!" using the Hill cipher with encryption matrix,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

If we convert "Cryptography is cool!" to matrix form we get,

$$M = \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix}$$

Then we would simply multiply EM modulo 26 and convert the result back to ciphertext characters. In Maxima, all we need to do is define the two matrices, E and M and then invoke the command `mod(E.M, 26)` and we are done. Specifically,

```
(%i1) E:matrix([2,3,15],[5,8,12],[1,13,4]);
```

(%o1) $\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$

```
(%i2) M:matrix([2,15,6,15,8,14],[17,19,17,7,18,14],[24,14,0,24,2,11]);
```

(%o2) $\begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix}$

```
(%i3) CT:mod(E.M, 26);
```

(%o3) $\begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$

Reading the columns we have the numeric list, 25 18 7 11 5 6 11 10 19 21 3 20 22 0 16 1 2 6 which converts to the ciphertext ZSHLFGLKTVDUWAQBCG. Δ

Example 62: Now say that Bob receives the message ZSHLFGLKTVDUWAQBCG from Alice and knows that the encryption matrix was,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Bob knows that he needs the inverse of the matrix E, modulo 26. There are several ways to do this with Maxima, we will use the easiest one here and then examine some alternatives in the next example. First put in the encryption matrix and the ciphertext matrix. Then load in the `mat_mod_inverse` script that we developed for finding the inverse of a square matrix modulo n, and use it to find the inverse of E. Finally, multiply this inverse, the decryption matrix, by the ciphertext matrix to get the plaintext matrix.

```
(%i1) E:matrix([2,3,15],[5,8,12],[1,13,4]);
```

(%o1) $\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$

```
(%i2) CT:matrix([25,11,11,21,22,1],[18,5,10,3,0,2],[7,6,19,20,16,6]);
```

(%o2) $\begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$

```
(%i3) 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)
      )$

(%i4) EI:mat_mod_inverse(E, 26);
```

(%o4) $\begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$

```
(%i5) mod(EI.CT, 26);
```

(%o5) $\begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix}$

Reading down the columns we get the numeric list, 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11, which converts to the plaintext message CRYPTOGRAPHYISCOOL. Δ

Example 63: The easiest way to find an inverse matrix in Maxima is with the `invert` command and to find a modular inverse with the script we developed so it can be done is a single `mat_mod_inverse` command. So why would we want to use a different, and probably more difficult method? Depending on what course this section is being integrated into, the instructor may wish that you practice other calculation skills. For example, if this is being integrated into a course that does not assume that you know any linear algebra or matrix theory then most likely the instructor will have you calculate the inverse as we did above. On the other hand, if this is being integrated into a linear algebra class, or a class that assumes a knowledge of linear algebra, then the instructor may wish use this as an example of how the topics and techniques in linear algebra work when we are doing our calculations modulo 26 instead of over the real numbers.

Recall from linear algebra that if a square matrix, M, is invertible then it can be reduced to the identity matrix. Furthermore, the row operations that are done in reducing M to the identity will also convert the identity matrix to the inverse of M. So if we construct the $n \times 2n$ matrix that has M in the left half and the $n \times n$ identity in the right half, notationally, $[M|I]$, and reduce this matrix to reduced row echelon form we will get the matrix $[I|M^{-1}]$. So our decryption matrix will be the right side $n \times n$ matrix after the reduction.

To apply this process in Maxima, we will do the following. The exercise is to find the inverse (modulo 26) of the following matrix using a reduction approach.

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

1. First load in the `mod_rref` script to be able to do modular matrix reduction.
2. Load in the matrix.
3. Use the `addcol` and the `ident` commands to join the matrix with the 3×3 identity matrix.
4. Apply the `mod_rref` command to the joined matrix.
5. Finally, use the `submatrix` command to extract the last three columns.

```
(%i1) 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)$

(%i2) A:matrix([2,3,15],[5,8,12],[1,13,4]);
```

(%o2) $\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$

```
(%i3) B:addcol(A,ident(3));
```

(%o3) $\begin{bmatrix} 2 & 3 & 15 & 1 & 0 & 0 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 1 & 13 & 4 & 0 & 0 & 1 \end{bmatrix}$

```
(%i4) RB:mod_rref(B,26);
```

(%o4) $\begin{bmatrix} 1 & 0 & 0 & 10 & 19 & 16 \\ 0 & 1 & 0 & 4 & 23 & 7 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$

```
(%i5) AI:submatrix(RB,1,2,3);
```

(%o5) $\begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$

Δ

Maxima can also help with the breaking of a Hill Cipher. We will redo the example from the section on breaking the Hill cipher.

Example 64: Eve has intercepted the encrypted message ZSHLFGLKTVDUWAQBCG and from other espionage knows that this message was CRYPTOGRAPHYISCOOL. She also knows that other messages sent that day between Alice and Bob are using the same key and she wishes to decrypt them as well, but she has no other information about these other messages.

Since the message size 18 she knows that that block size must be 2, 3, 6, 9 or 18, and since she has only 18 characters to work with her only hope is that the block size is either 2 or 3. If both of these fail it is back to other espionage.

Let's start with a block size of 2. Since,

CRYPTOGRAPHYISCOOL — 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

encrypts as

ZSHLFGLKTVDUWAQBCG — 25 18 7 11 5 6 11 10 19 21 3 20 22 0 16 1 2 6

we know that

$$\begin{bmatrix} 2 \\ 17 \end{bmatrix} \longrightarrow \begin{bmatrix} 25 \\ 18 \end{bmatrix} \qquad \begin{bmatrix} 24 \\ 15 \end{bmatrix} \longrightarrow \begin{bmatrix} 7 \\ 11 \end{bmatrix} \qquad \begin{bmatrix} 19 \\ 14 \end{bmatrix} \longrightarrow \begin{bmatrix} 5 \\ 6 \end{bmatrix} \cdots$$

So we want to construct a 2×2 matrix out of the plaintext blocks that is invertible modulo 26. If we take the first two blocks 2 17 and 24 15 and build a matrix from them

$$M = \begin{bmatrix} 2 & 24 \\ 17 & 15 \end{bmatrix}$$

then $\det(M) = -378$ which is not relatively prime to 26, so M is not invertible. Actually, we should have seen this coming with what we know about determinants, notice that the first row has all even numbers in it and 2 is a factor of 26.

If we move on to the next block, 19 14 and keep the first block our M matrix becomes,

$$M = \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}$$

then $\det(M) = -295$ which is 17 when we take it modulo 26. Since 17 has an inverse modulo 26 we are in business. Our equation becomes,

$$E \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix} = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix}$$

So

$$E = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}^{-1} = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \begin{bmatrix} 10 & 5 \\ 25 & 20 \end{bmatrix} = \begin{bmatrix} 11 & 17 \\ 18 & 2 \end{bmatrix}$$

To see if this works we test it on our plaintext message CRYPTOGRAPHYISCOOL.

$$\begin{aligned} C = EM &= \begin{bmatrix} 11 & 17 \\ 18 & 2 \end{bmatrix} \begin{bmatrix} 2 & 24 & 19 & 6 & 0 & 7 & 8 & 2 & 14 \\ 17 & 15 & 14 & 17 & 15 & 24 & 18 & 14 & 11 \end{bmatrix} \\ &= \begin{bmatrix} 25 & 25 & 5 & 17 & 21 & 17 & 4 & 0 & 3 \\ 18 & 20 & 6 & 12 & 4 & 18 & 24 & 12 & 14 \end{bmatrix} \end{aligned}$$

As we can see, columns 1 and 3 encrypt correctly, they should since those were the ones we used, but the rest of the encryption is incorrect. Conclusion, the block size is not 2.

To do the above calculations using Maxima do the following. Input the `mat_mod_inverse` script and the following three matrices,

$$A = \begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix} \qquad B = \begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix} \quad \text{and} \quad PT = \begin{bmatrix} 2 & 24 & 19 & 6 & 0 & 7 & 8 & 2 & 14 \\ 17 & 15 & 14 & 17 & 15 & 24 & 18 & 14 & 11 \end{bmatrix}$$

Find the inverse of A modulo 26, multiply it on the left by B, this should be the encryption matrix E, then apply it to the plaintext matrix to verify if it produced the ciphertext or not.

```
(%i1) 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)
      )$

(%i2) A:matrix([2,19],[17,14]);
```

(%o2) $\begin{bmatrix} 2 & 19 \\ 17 & 14 \end{bmatrix}$

```
(%i3) B:matrix([25,5],[18,6]);
```

(%o3) $\begin{bmatrix} 25 & 5 \\ 18 & 6 \end{bmatrix}$

```
(%i4) PT:matrix([2,24,19,6,0,7,8,2,14],
      [17,15,14,17,15,24,18,14,11]);
```

(%o4) $\begin{bmatrix} 2 & 24 & 19 & 6 & 0 & 7 & 8 & 2 & 14 \\ 17 & 15 & 14 & 17 & 15 & 24 & 18 & 14 & 11 \end{bmatrix}$

```
(%i5) AI:mat_mod_inverse(A,26);
```

(%o5) $\begin{bmatrix} 10 & 5 \\ 25 & 20 \end{bmatrix}$

```
(%i6) E:mod(B.AI,26);
```

(%o6) $\begin{bmatrix} 11 & 17 \\ 18 & 2 \end{bmatrix}$

```
(%i7) M:mod(E.PT,26);
```

(%o7) $\begin{bmatrix} 25 & 25 & 5 & 17 & 21 & 17 & 4 & 0 & 3 \\ 18 & 20 & 6 & 12 & 4 & 18 & 24 & 12 & 14 \end{bmatrix}$

If we convert the result to ciphertext letters we get, ZSZUFGRMVERSEYAMDO which is not our ciphertext of ZSHLFGLKTVDUWAQBCG. Your conclusion would be the same as above, the block size is not 2.

So we move on to block size 3. If it is then there is a 3×3 matrix E with

$$E \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$$

As before, we want to select three columns from our message matrix that will produce an invertible 3×3 matrix. Looking at the last row, all entries except for 11 are even, so the last column must be used. Since the last column first row is even we must have at least one odd number in the first row of the columns we use. So we could not select the remaining two columns from columns 1, 3, and 5. Furthermore, column 5 is all even so selecting it would be pointless. So in our selection we must have column 6 and at least one of columns 2 and 4. We will try columns 1, 2, and 6. This gives,

$$M = \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}$$

Since $\det(M) = -791 \equiv 15 \pmod{26}$, we know that M is invertible. So

$$EM = E \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix}$$

and then,

$$E = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}^{-1} = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \begin{bmatrix} 13 & 9 & 24 \\ 3 & 12 & 14 \\ 8 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Now, we know this is correct but Eve would still need to check her possible solution, doing so she would get,

$$\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix} \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix} = \begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$$

which checks with the ciphertext and she has found the encryption matrix, E.

To do the above calculations using Maxima do the following. Input the following three matrices along with the `mat_mod_inverse` script.

$$A = \begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix} \qquad B = \begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix} \quad \text{and} \quad PT = \begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix}$$

Then follow the same procedure as we did above. Find the inverse of A modulo 26, multiply it on the left by B, this should be the encryption matrix E, then apply it to the plaintext matrix to verify if it produced the ciphertext or not.

```
(%i1) 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)
      )$

(%i2) A:matrix([2,15,14],[17,19,14],[24,14,11]);
```

(%o2) $\begin{bmatrix} 2 & 15 & 14 \\ 17 & 19 & 14 \\ 24 & 14 & 11 \end{bmatrix}$

```
(%i3) B:matrix([25,11,1],[18,5,2],[7,6,6]);
```

(%o3) $\begin{bmatrix} 25 & 11 & 1 \\ 18 & 5 & 2 \\ 7 & 6 & 6 \end{bmatrix}$

```
(%i4) PT:matrix([2,15,6,15,8,14],
      [17,19,17,7,18,14],[24,14,0,24,2,11]);
```

(%o4) $\begin{bmatrix} 2 & 15 & 6 & 15 & 8 & 14 \\ 17 & 19 & 17 & 7 & 18 & 14 \\ 24 & 14 & 0 & 24 & 2 & 11 \end{bmatrix}$

```
(%i5) AI:mat_mod_inverse(A,26);
```

(%o5) $\begin{bmatrix} 13 & 9 & 24 \\ 3 & 12 & 14 \\ 8 & 10 & 15 \end{bmatrix}$

```
(%i6) E:mod(B.AI,26);
```

(%o6) $\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$

```
(%i7) M:mod(E.PT,26);
```

(%o7) $\begin{bmatrix} 25 & 11 & 11 & 21 & 22 & 1 \\ 18 & 5 & 10 & 3 & 0 & 2 \\ 7 & 6 & 19 & 20 & 16 & 6 \end{bmatrix}$

If we convert the result to ciphertext letters we get, ZSHLFGLKTVDUWAQBCG which is our ciphertext of ZSHLFGLKTVDUWAQBCG. The ciphertext matrix checks out, so the block size was three and the encryption matrix is,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Δ

2.13.10 Exercises

In some of these exercises you may want, or be required, to do some matrix reduction. Remember that in this context all operations are done modulo 26. So, as we did above, if you divide a row by a number make sure that the number is invertible modulo 26 and multiply the row by its inverse. If you are using Cryptography Explorer to help with the calculations you should use the the Row Operations interface at the bottom of the modular matrix calculator window.

All of the exercises here assume that we are using the modern method of applying the Hill cipher, so the plaintext blocks are converted to column vectors $\mathbf{v}$ and left multiplied by the encryption matrix M to get $M\mathbf{v} = \mathbf{w}$, and $\mathbf{w}$ will be the column vector representing the ciphertext block. Also, if you are using the Hill Cipher tool from the Cryptography Explorer program, make sure that you select the Modern Mode option.

1. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 2 & 7 \\ 13 & 9 \end{bmatrix}$$

   (a) What is the determinant of E?
   (b) Is E invertible modulo 26? Why or why not?
   (c) If E is invertible modulo 26, find its inverse modulo 26.
   (d) Use E to encrypt the message HELP.
   (e) What message or messages encrypt to XXXX?

2. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 2 & 7 \\ 15 & 8 \end{bmatrix}$$

   (a) What is the determinant of E?
   (b) Is E invertible modulo 26? Why or why not?
   (c) If E is invertible modulo 26, find its inverse modulo 26.
   (d) Use E to encrypt the message HELP.
   (e) What message or messages encrypt to XXXX?

3. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 2 & 6 \\ 15 & 8 \end{bmatrix}$$

   (a) What is the determinant of E?
   (b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message HELP.

(e) What message or messages encrypt to XXXX?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does HELP? If so, find all plaintext messages that encrypt to the same ciphertext as does HELP. If not, explain why.

4. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 8 & 7 & 17 \\ 19 & 18 & 4 \\ 20 & 1 & 4 \end{bmatrix}$$

(a) What is the determinant of E?

(b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message ATTACK.

(e) What message or messages encrypt to ATTACK?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does ATTACK? If so, find all plaintext messages that encrypt to the same ciphertext as does ATTACK. If not, explain why.

(g) Are there any plaintext messages that are sent to the zero vector? If so, find all of them and if not explain why?

(h) Compare your results from the last two parts of this exercise, what is the relation between these two answers?

5. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 8 & 7 & 18 \\ 19 & 18 & 4 \\ 20 & 1 & 4 \end{bmatrix}$$

(a) What is the determinant of E?

(b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message CHARGE.

(e) What message or messages encrypt to CHARGE?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does CHARGE? If so, find all plaintext messages that encrypt to the same ciphertext as does CHARGE. If not, explain why.

(g) Are there any plaintext messages that are sent to the zero vector? If so, find all of them and if not explain why?

(h) Compare your results from the last two parts of this exercise, what is the relation between these two answers?

6. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 4 & 8 & 5 & 22 \\ 8 & 16 & 21 & 17 \\ 21 & 25 & 5 & 17 \\ 17 & 4 & 9 & 10 \end{bmatrix}$$

(a) What is the determinant of E?

(b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message RUN AWAY. Note that since RUN AWAY, when the space is removed, has only seven letters we will need to pad the end with a random letter, in many cases an X is used. we will use this convention and encrypt RUNAWAYX.

(e) What message or messages encrypt to RUNAWAYX?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does RUNAWAYX? If so, find all plaintext messages that encrypt to the same ciphertext as does RUNAWAYX. If not, explain why.

(g) Are there any plaintext messages that are sent to the zero vector? If so, find all of them and if not explain why?

(h) Compare your results from the last two parts of this exercise, what is the relation between these two answers?

7. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 24 & 17 & 20 & 17 \\ 25 & 21 & 7 & 9 \\ 23 & 22 & 2 & 5 \\ 3 & 17 & 18 & 23 \end{bmatrix}$$

(a) What is the determinant of E?

(b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message RUN AWAY. Note that since RUN AWAY, when the space is removed, has only seven letters we will need to pad the end with a random letter, in many cases an X is used. we will use this convention and encrypt RUNAWAYX.

(e) What message or messages encrypt to RUNAWAYX?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does RUNAWAYX? If so, find all plaintext messages that encrypt to the same ciphertext as does RUNAWAYX. If not, explain why.

(g) Are there any plaintext messages that are sent to the zero vector? If so, find all of them and if not explain why?

(h) Compare your results from the last two parts of this exercise, what is the relation between these two answers?

8. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 20 & 15 & 7 & 17 \\ 23 & 5 & 9 & 19 \\ 3 & 5 & 25 & 22 \\ 9 & 25 & 15 & 1 \end{bmatrix}$$

(a) What is the determinant of E?

(b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message RUN AWAY. Note that since RUN AWAY, when the space is removed, has only seven letters we will need to pad the end with a random letter, in many cases an X is used. we will use this convention and encrypt RUNAWAYX.

(e) What message or messages encrypt to RUNAWAYX?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does RUNAWAYX? If so, find all plaintext messages that encrypt to the same ciphertext as does RUNAWAYX. If not, explain why.

(g) Are there any plaintext messages that are sent to the zero vector? If so, find all of them and if not explain why?

(h) Compare your results from the last two parts of this exercise, what is the relation between these two answers?

9. Which of the matrices in the above examples could be used for a Hill cipher and which are not suitable? In each case explain why?

10. For each of the matrices in the previous exercises that are invertible, find their inverses using,

(a) The reduction technique.

(b) The adjoint technique.

11. For each of the matrices in the previous exercises that are not invertible, reduce them as far as possible. Do any reduce to the identity matrix?

12. For each of the matrix transformations in the previous exercises, find the kernel of the transformation.

13. For each of the matrices in the previous exercises, find a maximum set of columns that are linearly independent.

14. For each of the matrices in the previous exercises, which ones have columns that form a basis to $\mathbb{A}^n$? Why?

15. In the description of the Hill cipher we require that the encryption matrix E is invertible modulo 26. What properties of an invertible matrix are important in the encryption and decryption process and why?

16. We have intercepted a segment of the plaintext of a message

    SENDMORETROOPS

    and we have the corresponding ciphertext

    SYTKAWQVHOECOB

    We know that the segment was at the beginning of the message and that there was more ciphertext that followed but we do not know the corresponding plaintext. We, of course, wish to decrypt the entire message, so we need the decryption matrix. Find the encryption and decryption matrices for this cipher, if possible. If there is not enough information to do this, state why.

17. We have intercepted a segment of the plaintext of a message

    WEAREINNEEDOFASSISTANCER

    and we have the corresponding ciphertext

    KWULUEKTYVJEZCQAGSPWRWYJ

    We know that the segment was at the beginning of the message and that there was more ciphertext that followed but we do not know the corresponding plaintext. We, of course, wish to decrypt the entire message, so we need the decryption matrix. Find the encryption and decryption matrices for this cipher, if possible. If there is not enough information to do this, state why.

18. We have intercepted the plaintext of a message

    INEEDABIGCUPOFCOFFEE

    and we have the corresponding ciphertext

YSPMYLQNPBTDGWLLXILVV

Find the encryption and decryption matrices for this cipher, if possible. If there is not enough information to do this, state why.

We suspect that the same matrix was used to send messages for that entire day. Later we intercepted the following ciphertext, find the message, if possible.

JRLACWNLZQEYBSHOWQSFMXFFIKFAXSCOT

19. Say that Alice uses the following matrix for her encryption matrix,

$$E = \begin{bmatrix} 2 & 7 \\ 13 & 9 \end{bmatrix}$$

and encrypts the message,

SOLONGANDTHANKSFORALLTHEFISH

into

EWQJQPNNJCONSZTTRXZVZCQXOHHL

When she sends the message to Bob there is one error in the transmission and Bob receives the ciphertext,

EWQJQPNNJCONSZTARXZVZCQXOHHL

When Bob decrypts the message how many errors are in his decryption?

20. Say that Alice uses the following matrix for her encryption matrix,

$$E = \begin{bmatrix} 23 & 7 & 22 & 3 \\ 25 & 16 & 3 & 4 \\ 6 & 22 & 6 & 7 \\ 7 & 20 & 8 & 6 \end{bmatrix}$$

and encrypts the message,

SOLONGANDTHANKSFORALLTHEFISH

into

QJIGQFPDSKKPANZXGQPKGSIRQXZR

When she sends the message to Bob there are two errors in the transmission and Bob receives the ciphertext, QMIGQFPDSKKPYNZXGQPKGSIRQXZR

When Bob decrypts the message how many errors are in his decryption?

21. From the two exercises above, what is the relationship between transmission errors and errors in the decryption of the message? Does it make any difference where the errors are, for example, if the errors are close together or far apart?

    In World War I the German Military used the ADFGX code. In this code, the first step was to rewrite each letter in the message as a pair of letters from the set $\{A, D, F, G, X\}$, for example, a might be coded as DF, b as AA, and so on. From there the encryption method used only the characters ADFGX. The reason for this choice was that in Morse code, the method of transmission, the letters A, D, F, G, and X were easy to distinguish. So if there was a transmission error, usually human error in this case, the receiver could easily tell what the letter should have been. This was one of the first uses of error correcting codes in cryptography. Today, the use of error correction codes is commonplace in cryptography.

22. Say we use the following matrix to encrypt the message,

    ATTACK

$$E = \begin{bmatrix} 8 & 19 & 9 & 22 & 7 & 10 \\ 6 & 18 & 25 & 25 & 10 & 0 \\ 0 & 25 & 24 & 0 & 24 & 0 \\ 0 & 22 & 0 & 6 & 16 & 3 \\ 19 & 6 & 10 & 0 & 0 & 7 \\ 19 & 23 & 10 & 10 & 21 & 23 \end{bmatrix}$$

    (a) What does the message encrypt to?
    (b) Find the decryption matrix.
    (c) We will now put in a single transmission error. Change the third letter of the ciphertext to a C, and decrypt the new (flawed) message.
    (d) How many errors are in the flawed decryption? Is it possible to recover the original message with this decryption?
    (e) If we do not know the number or position of the errors in the transmission of a single word, as is usually the case, what are all of the possible plantext messages with one word that could be received as the ciphertext ABCDEF?

23. In some of the exercises above we examined what happens if there is an error in the transmission of the ciphertext. This exercise looks at what happens if we have an error in the encryption matrix. Say Alice and Bob share the key, the encryption matrix, but Bob's handwriting is not all that great and he cannot read the entry in the first row

and third column but he knows that the rest of the encryption matrix is correct. Bob currently has,

$$E = \begin{bmatrix} 20 & 22 & x \\ 1 & 5 & 5 \\ 7 & 4 & 20 \end{bmatrix}$$

(a) Just knowing this, what are the possibilities for the value of x? How did you come to this conclusion?

(b) Later that day Bob receives the following ciphertext from Alice

IXHSZCMQOERVTCZ

From this, can Bob determine the value of x? If so what is it and what does the message say?

24. In our discussion above we stated that we can think of the encryption matrix, E, as a linear transformation $E : \mathbb{A}^n \to \mathbb{A}^n$. Formally, we define $E(\mathbf{v}) = E\mathbf{v}$ for all $\mathbf{v} \in \mathbb{A}^n$. With this definition, show that E is a linear transformation.

25. In the definition of the Hill cipher, the matrix E must be invertible modulo 26. As we saw, that meant that the determinant of E had to be invertible in $\mathbb{A}$, which is where the entries of E came from. As we noted above as well, this was in line with the Invertible Matrix Theorem, one part of which states that an $n \times n$ matrix with real coefficients is invertible if and only if the determinant of that matrix is not 0. That is, if the determinant is invertible in the real number system. Since the only real number that is not invertible is 0 we need only exclude it and can simplify the statement. In this exercise we are going to examine some of the other parts of the Invertible Matrix Theorem in relation to $\mathbb{A}$ and its implications to the Hill cipher. Also recall that we can think of the encryption matrix, E, as a linear transformation $E : \mathbb{A}^n \to \mathbb{A}^n$. We will use these two points of view interchangeably.

(a) Say that E is invertible. What does this imply about the properties of one-to-one and onto of the transformation?

(b) Say that E is not invertible. What does this imply about the properties of one-to-one and onto of the transformation?

(c) If the transformation was not one-to-one, what would this imply about encryption and decryption with the Hill cipher? Be specific as to the difficulties that would arise.

(d) If the transformation was not onto, what would this imply about encryption and decryption with the Hill cipher? Be specific as to the difficulties that would arise.

(e) If E is invertible, what is the kernel of E? How does this relate to your answers above?

(f) If E is not invertible, what is the kernel of E? How does this relate to your answers above?

(g) If E is not invertible, is it possible that two English messages could be sent to the same ciphertext? How does this relate to your answers above? Find two examples of non-invertible encryption matrices E and for each, two different English plaintexts that are encrypted to the same ciphertext.

(h) If E is not invertible, and you know one decryption of a given ciphertext as well as the kernel of E, how can you determine, without knowing E, what all of the possible decryptions are? Justify your answer.

(i) If E is invertible, what can be said about the columns of E in relation to the set $\mathbb{A}^n$? For each observation, discuss the relevance of it to the Hill cipher.

26. In the work we did above, we used a Known Plaintext attack on the cipher to find its key, that is, the encryption matrix. One question would be if other types of attacks could gather enough information for us to determine the encryption, and hence decryption matrices?

    (a) Chosen Plaintext: Say that Eve temporally gains access to the encryption machine. Think of it as a black box that she can input messages and get the encryption back.

        i. Suppose further that she knows the block size. What plaintexts should she choose to find the encryption matrix? How would she use the output of the machine to construct the encryption matrix? If she knows that she has only a few tries before she is detected and locked out of the system, what is the fewest number of plaintexts she needs to determine the encryption matrix?

        ii. Suppose she does not know the block size and the machine automatically pads the input with X's as well as breaks the message into blocks. What plaintexts should she choose to find the encryption matrix? How would she use the output of the machine to construct the encryption matrix? If she knows that she has only a few tries before she is detected and locked out of the system, what is the fewest number of plaintexts she needs to determine the encryption matrix?

    (b) Chosen Ciphertext: Say that Eve temporally gains access to the decryption machine. Think of it also as a black box that she can input ciphertext and get the plaintext back.

        i. Suppose further that she knows the block size. What ciphertexts should she choose to find the encryption matrix? How would she use the output of the machine to construct the encryption matrix? If she knows that she has only a few tries before she is detected and locked out of the system, what is the fewest number of ciphertexts she needs to determine the encryption matrix?

        ii. Suppose she does not know the block size and the machine automatically pads the input with X's as well as breaks the message into blocks. What ciphertexts should she choose to find the encryption matrix? How would she use the output of the machine to construct the encryption matrix? If she knows that she has only a few tries before she is detected and locked out of the system, what is the fewest number of ciphertexts she needs to determine the encryption matrix?

(c) Ciphertext Only: Here Eve has only the ciphertext of several messages.

i. How could she determine the block size? Or at least a small set of possible block sizes.

ii. Once she has determined the set of possible block sizes, how would she proceed to find the encryption matrix?

iii. In many cases a simple substitution cipher can be broken using only a single ciphertext, if it is sufficiently long. Could Eve break the Hill cipher with only a single, fairly long, ciphertext? If so, what restrictions would need to be assumed?

27. Look back at the definition of a vector space. If we let our set of scalars be $\mathbb{A}$ instead of $\mathbb{R}$ and if we let $V = \mathbb{A}^n$ instead of $\mathbb{R}^n$, which of the properties of a vector space still hold for V and which do not? For each, if the property holds prove it and if the property does not hold then find a counter example to show that the property fails.

28. The Hill cipher is defined to work with square encryption matrices E. Since the decryption matrix $D = E^{-1}$, it is clear why this restriction is imposed. One question would be if we could relax this condition and let E be an $n \times m$ matrix where $n \neq m$? Our immediate answer to this would be no since if E was not square, it would not have an inverse D for decryption.

On the other hand, if we think about E and D we see that there is a little overkill on this restriction. Specifically, when Alice encrypts her message M, she divides the message into blocks of n letters that is then converted into a column of n numbers. Each block, or column, can, and is, looked at as a vector $\mathbf{v} \in \mathbb{A}^n$. Then when she encrypts her message, she applies E to each of the vectors, getting an encrypted vector $\mathbf{w} = E\mathbf{v}$. Since D is the inverse of E we know that $D\mathbf{w} = \mathbf{v}$ and hence we can decrypt the message. So what we require is that $DE\mathbf{v} = D(E\mathbf{v}) = D\mathbf{w} = \mathbf{v}$ for all vectors $\mathbf{v} \in \mathbb{A}^n$. But if D is the inverse of E we also have that $ED\mathbf{v} = \mathbf{v}$ for all vectors $\mathbf{v} \in \mathbb{A}^n$, which is not required by the Hill cipher algorithm. That is, we only need to get back to $\mathbf{v}$ with the product DE and not ED.

Before we get to far into this lets step back and ask the same question but with matrices over the real number system. That is, can we produce two non-square matrices A and B such that $AB = I$ but $BA \neq I$? From the products it is clear that if A is $n \times m$ then B must be $m \times n$ and the products AB will be $n \times n$ and BA will be $m \times m$.

(a) Consider the matrix

$$E = \begin{bmatrix} -6 & -4 & 7 \\ 1 & 5 & 5 \end{bmatrix}$$

over the real numbers, does there exist a matrix D with $DE = I$? To prove or disprove this construct a generic 3×2 matrix D,

$$D = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}$$

and compute

$$DE = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix} \begin{bmatrix} -6 & -4 & 7 \\ 1 & 5 & 5 \end{bmatrix}$$

From this, can we solve for each of the components of D? If so, then we have $DE = I$ as desired. Also, if it is possible one would ask if D is unique?

i. Do the above product and solve the resulting system, over the real numbers, does a solution exist?
ii. If a solution exists write the matrix D.
iii. Is D unique?
iv. If not, write a general expression for all matrices D with $DE = I$.

(b) Consider the matrix

$$E = \begin{bmatrix} -6 & 1 \\ 1 & 5 \\ 7 & 4 \end{bmatrix}$$

over the real numbers, does there exist a matrix D with $DE = I$? To prove or disprove this construct a generic 2×3 matrix D,

$$D = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

and compute

$$DE = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} -6 & 1 \\ 1 & 5 \\ 7 & 4 \end{bmatrix}$$

From this, can we solve for each of the components of D? If so, then we have $DE = I$ as desired. Also, if it is possible one would ask if D is unique?

i. Do the above product and solve the resulting system, over the real numbers, does a solution exist?
ii. If a solution exists write the matrix D.
iii. Is D unique?
iv. If not, write a general expression for all matrices D with $DE = I$.

(c) From the above calculations what would you conjecture as the answer to the question,

> Given an $n \times m$ matrix A does there exist an $m \times n$ matrix B with $AB = I$?

What restrictions must be placed on n and m? With these restrictions, is it always possible? Why or why not?

(d) Think of the matrices A and B as transformations between $\mathbb{R}^n$ and $\mathbb{R}^m$. Revise your statement about the restrictions on the sizes of n and m to the language of transformations. That is, discuss the properties of one-to-one and onto of the transformation induced by A and B and the necessity of these restrictions needed for AB to be both one-to-one and onto, as is the identity matrix.

(e) If we can find two non-square matrices A and B such that the matrix AB produces a one-to-one and onto transformation can we find another matrix C with $AC = I$ or can we find another matrix D with $DB = I$? If so, how would we create either C or D? If not find two matrices A and B with the matrix AB producing a one-to-one and onto transformation but there exists no matrices C and D with either $AC = I$ or $DB = I$.

(f) Discuss this situation with matrices over $\mathbb{A}$ in place of matrices over $\mathbb{R}$. Specifically,

i. Is it possible to relax the Hill requirement of using a square matrix?
ii. If so, what restrictions must be placed on n and m?
iii. If so, with these restrictions, can this always be done? Why or why not?
iv. If so, give examples of non-square matrices E and D over $\mathbb{A}$ with $ED = I$ modulo 26.

29. In the Hill cipher we do all of calculations modulo 26 since, of course, there are 26 letters in the English alphabet. What if we had a different alphabet? Say we had an alphabet with 27 letters, or 29 letters. What if our language symbolism was based on syllables instead of letters, then we would probably have around 80 to 100 characters? What if our language were based on pictures, then we could have thousands of characters. Updating the Hill cipher would be easy in this case, all we do is change the modulus.

(a) Say our alphabet had only 5 letters. What would the determinant of a Hill cipher encryption matrix need to be?

(b) Say our alphabet had only 12 letters. What would the determinant of a Hill cipher encryption matrix need to be?

(c) Say our alphabet had 29 letters. What would the determinant of a Hill cipher encryption matrix need to be?

(d) What would the advantage be to the Hill cipher if we had a prime number of letters in our alphabet?

(e) Say we had an alphabet with 5 letters in it, $\{A, B, C, D, E\}$. So in this case $\mathbb{A} = \{0, 1, 2, 3, 4\}$, with addition and multiplication done modulo 5. Consider the following matrix with entries in $\mathbb{A}$.

$$E = \begin{bmatrix} 4 & 1 & 2 \\ 1 & 0 & 0 \\ 2 & 4 & 4 \end{bmatrix}$$

i. What is the determinant of E?
ii. Could E be used as an encryption matrix for a Hill cipher?
iii. If so, find the decryption matrix D using both the reduction technique and the adjoint technique.
iv. If so, encrypt the message ABBEDA.
v. If so, decrypt the message AAABBB.

(f) Say we had an alphabet with 5 letters in it, $\{A, B, C, D, E\}$. So in this case $\mathbb{A} = \{0, 1, 2, 3, 4\}$, with addition and multiplication done modulo 5. Consider the following matrix with entries in $\mathbb{A}$.

$$E = \begin{bmatrix} 1 & 3 & 4 \\ 1 & 4 & 2 \\ 1 & 1 & 3 \end{bmatrix}$$

i. What is the determinant of E?
ii. Could E be used as an encryption matrix for a Hill cipher?
iii. If so, find the decryption matrix D using both the reduction technique and the adjoint technique.
iv. If so, encrypt the message ABBEDA.
v. If so, decrypt the message CCDDEE.

30. Pair up with another student in your class. Both of you create your own plaintext message (in English) and an invertible matrix modulo 26. Make the message fairly long, at least 80 alphabetic characters. Keep the matrix you use for the cipher fairly small, say 3×3 to 5×5. Encrypt your message using your matrix. Give the other person the ciphertext of the message and a portion of the plaintext. The portion of plaintext you give must be at least 40 consecutive characters. The plaintext can start anywhere in the message but you need to tell the person where the plaintext starts. Break the code your partner gives you and decrypt their message.

2.14 The Hill Cipher Classical Approach

2.14.1 Introduction

You have probably noticed that there are two sections on the Hill Cipher. The two sections contain the same material, the only difference is the approach we take. When Lester Hill devised this cipher it was 1929, at the time, mathematicians more commonly viewed applying a linear transformation to a vector as taking a row vector $\mathbf{v}$ and transformation matrix M and applying the transformation by, $\mathbf{v}M = \mathbf{w}$. Now, we tend to view a linear transformation more as a function from one vector space to another. That is, $T : V \to W$, and hence we have adopted a more functional notation. So if $\mathbf{v}$ is a vector in the vector space V and T is a linear transformation from V to W, then we would write $T(\mathbf{v}) = \mathbf{w}$ or sometimes just $T\mathbf{v} = \mathbf{w}$ for applying the transformation T to the vector $\mathbf{v}$. This functional notation has also transferred over to matrices. So if V and W are finite dimensional vector spaces, M is the matrix for a linear transformation T over some bases for V and W, and $\mathbf{v}$ is a vector in the vector space V then $T(\mathbf{v}) = M\mathbf{v} = \mathbf{w}$. Note that in this notation, $\mathbf{v}$ and $\mathbf{w}$ are doing double duty by representing vectors in V and W respectively and representing their coordinate vectors over bases for V and W respectively, also they are column vectors instead of row vectors.

The only difference between this section and the last is approach we take. Here, we use the more classical notationthat Lester Hill did in 1929 of $\mathbf{v}M$, and in the previous section we used the more modern notation of $M\mathbf{v}$. If you recall from your linear algebra class, if you have a matrix M and row vectors $\mathbf{v}$ and $\mathbf{w}$ with $\mathbf{v}M = \mathbf{w}$ and you transpose both sides you get, $(\mathbf{v}M)^T = M^T\mathbf{v}^T = \mathbf{w}^T$. Of course, $\mathbf{v}^T$ and $\mathbf{w}^T$ are column vectors, so the only difference between this section and the last is that all of the transformation matrices are transposes of each other. So why bother having two separate sections? We did this just for the convenience of the reader and the instructor. If you are using this set of notes as a supplement for a linear algebra class then you will probably want to use the last section, whereas, if you are using this set of notes for a cryptography course then you will probably want to use this section. The majority of cryptography textbooks that discuss the Hill Cipher use the classical notation of $\mathbf{v}M = \mathbf{w}$.

2.14.2 History

The Hill Cipher is a block cipher that was developed by Lester Hill in 1929, setting it firmly in the classical age of cryptography. 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*[22]. Although it seems that this method was not used much in practice, it marked a transitional period in cryptography, where cryptography shifted 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 Germans in World War II, and other cipher machines of the same period, cryptanalysis of these ciphertexts required advanced mathematics and an enormous amount of computation, far beyond that of a single person or group of people.

2.14.3 Encryption with the Hill Cipher

Although we can apply the Hill Cipher to any language and any alphabet we will consider the uppercase English alphabet for now and then generalize it to other alphabets later in the section. As with most of our other methods, we will be doing calculations modulo 26. The key to the Hill cipher is a square matrix of size $n \times n$, where n can be as large as desired. When working with matrices modulo 26, we simply do all calculations modulo 26. So if we were to add or multiply two matrices, we would do it in the same manner as we would in a linear algebra class but at the end we would mod each matrix entry by 26. Also, if we take the determinant of a matrix, we do the calculation as we would in a linear algebra class and then mod out the determinant value by 26. In addition, if we wanted to invert a square matrix modulo 26, we can use the standard reduction algorithm but do all calculations modulo 26. This would include division, so if we wanted to multiply a row by $\frac{1}{3}$ we would multiply the row by $3^{-1} \pmod{26}$. So it goes without saying that we can only divide a row by a number that is invertible modulo 26, for example, dividing by 3 would be fine but we would not be able to divide by 8, since $\gcd(26, 8) \neq 1$. The appendix on modular arithmetic discusses matrix operations and gives several examples of matrix arithmetic on matrices, including inverting a square matrix.

The Hill cipher is really just a block cipher where the plaintext is broken up into blocks of size n, converted to an n-dimensional vector (row vector in this section), multiplied (on the right) by an $n \times n$ matrix (which is the encryption key) to produce another n-dimensional vector, and then converted back into the alphabet as the ciphertext. Decryption is done in a similar manner except that the decryption key is the inverse of the encryption matrix, modulo 26.

The Hill Cipher Encryption Algorithm

1. Find an $n \times n$ matrix E that is invertible modulo 26. This is the encryption key.

2. Take the message that is to be sent (the plaintext), remove all of the spaces and punctuation symbols, and convert the letters into all uppercase.

3. Convert each character to a number between 0 and 25. The usual way to do this is $A = 0$, $B = 1$, $C = 2$, …, $Z = 25$.

   As a historical note, Lester Hill did not use this coding of letters to numbers, he simply mixed up the order. Mixing up the order does not make the method more secure, it simply combines the Hill cipher with a simple substitution cipher.

4. Divide this string of numbers up into blocks of size n. Note that if E is an $n \times n$ matrix then the block size is n. Another note, if the message does not break evenly

Table 2.38: English Alphabet Numeric Coding 0–25

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M
Code	0	1	2	3	4	5	6	7	8	9	10	11	12
Letter	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Code	13	14	15	16	17	18	19	20	21	22	23	24	25

into blocks of size n we pad the ending of the message with characters, this can be done at random.

5. Write each block as a row vector of size n. At this point the message is a sequence of n-dimensional vectors, $v_1, v_2, \ldots, v_t$.

6. Take each of the vectors and multiply them by the encryption matrix E, so

$$\begin{aligned} v_1E &= w_1 \\ v_2E &= w_2 \\ v_3E &= w_3 \\ &\vdots \\ v_tE &= w_t \end{aligned}$$

7. Take the vectors $w_1, w_2, \ldots, w_t$, write the entries of the vectors in order, convert the numbers back to characters and you have your ciphertext.

One note about this algorithm is that we can do step 6 with a single matrix multiplication. If we let the message matrix M be the matrix produced by having the vectors $v_1, v_2, \ldots, v_t$ as rows, that is,

$$M = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_t \end{bmatrix}$$

then

$$ME = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_t \end{bmatrix} = C$$

would be our ciphertext matrix.

Example 65: Say Alice wants to send Bob the message "Cryptography is cool!"

1. Alice chooses the block size $n = 3$ and chooses the encryption matrix E to be,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Since $\det(E) \pmod{26} = 11$, and 11 is invertible modulo 26, the matrix E is also invertible modulo 26.

2. The message that is to be sent is "Cryptography is cool!", removing the spaces and punctuation symbols, and convert the letters into all uppercase gives

   CRYPTOGRAPHYISCOOL

3. Conversion to numbers using $A = 0$, $B = 1$, $C = 2$, $\ldots$, $Z = 25$, gives

   2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

4. Dividing this string of numbers up into blocks of size 3.

   2 17 24   15 19 14   6 17 0   15 7 24   8 18 2   14 14 11

   so no padding is needed here.

5. Converting these blocks into a message matrix M gives,

$$M = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix}$$

6. Multiply by the encryption matrix E,

$$ME = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix} \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix} = C$$

7. Convert C into the ciphertext.

   9 12 18 9 15 15 19 24 8 11 23 15 4 12 6 5 11 6

   JMSJPPTYILXPEMGFLG

   So Alice will send "JMSJPPTYILXPEMGFLG" to Bob.

Δ

Since this is a symmetric cipher, Alice and Bob would have to share this key with each other. They obviously could not simply call or text each other with this information since Eve could easily intercept that call or text and would know the key. So either Alice and Bob would have to meet in person, in a secure location, and exchange the key or they would need some other trusted person to deliver the key from Alice to Bob. This difficulty in exchanging the key securely gave rise to the creation of public-key systems which are commonly used today, for more information on public-key systems please see the references [55] and [62].

2.14.4 Decryption with the Hill Cipher

Now that Bob has the encrypted message and the encryption key he can decrypt the message that Alice had sent to him. The decryption algorithm is essentially the same as the encryption algorithm, except that we use E^{-1} in place of E. Since $ME = C$, and E is invertible we can calculate $M = CE^{-1}$. We will call $D = E^{-1}$ the decryption matrix, so $CD = M$. Remember that this inverse is the inverse modulo 26.

The Hill Cipher Decryption Algorithm

1. Find $D = E^{-1} \pmod{26}$. This is the decryption key.
2. Take the ciphertext and convert it to the matrix C.
3. Calculate $CD = M$.
4. Convert the matrix M to the plaintext message. You may need to insert the appropriate spaces and punctuation symbols since these were removed.

Example 66: Bob has the encrypted message JMSJPPTYILXPEMGFLG.

1. He calculates
$$\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}^{-1} \pmod{26} = \begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$$
2. He also converts the ciphertext to the matrix C.

JMSJPPTYILXPEMGFLG

9 12 18 9 15 15 19 24 8 11 23 15 4 12 6 5 11 6

and since he knows that the block size is 3 he constructs C as
$$C = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$$

3. Calculate $CD = M$.

$$CD = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix} \begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix} = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix} = M$$

4. Convert the matrix M to the plaintext message.

2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

CRYPTOGRAPHYISCOOL

So Bob adds in a couple spaces to get CRYPTOGRAPHY IS COOL!

Δ

2.14.5 Breaking the Hill Cipher

Now it is Eve's turn, how can she find the key to a Hill cipher? Looking at the encryption algorithm we know that $ME = C$. If we have a portion of the plaintext and its corresponding ciphertext (a Known Plaintext attack) then we have a little of M and C. If we are lucky, the portion of M that we have might form an invertible $n \times n$ matrix (modulo 26). Then $ME = C$ could be rewritten as $E = M^{-1}C$, giving her the encryption matrix. From there, she simply inverts E modulo 26 to get D and then she can decrypt the entire message.

There is really one more piece of information that Eve needs, if she just has plaintext and ciphertext characters she does not know the block size n and hence she does not know the size of E nor the size of the matrices M and C she needs to use to find E. This is clearly a problem. But there is a way to guess the possible block sizes, if the message is not too long. Since Eve can get the entire ciphertext, she knows the number of letters in the message (possibly padded) and that this number of characters must be a multiple of the block size. So if she has intercepted JMSJPPTYILXPEMGFLG she knows that the message has 18 characters in it, so the block sizes could possibly be, 2, 3, 6, 9 or 18. If the block size was 6, 9, or 18 she would not have enough characters to create M and C so it would not be possible for her to find E and hence D. So the only possibilities she would have that would allow her to find the key would be block sizes of 2 or 3. If these both fail to produce a key then she knows that she will not be able to break the code and not know the original message. As an example we will assume that Eve knows that CRYPTOGRAPHYISCOOL encrypts as JMSJPPTYILXPEMGFLG and we will follow the process outlined above to find that the block size is 3 and the matrix E.

Example 67: Eve has intercepted the encrypted message JMSJPPTYILXPEMGFLG and from other espionage knows that this message was CRYPTOGRAPHYISCOOL. She also knows that other messages sent that day between Alice and Bob are using the same key and she wishes to decrypt them as well, but she has no other information about these other messages.

Since the message size 18 she knows that that block size must be 2, 3, 6, 9 or 18, and since she has only 18 characters to work with her only hope is that the block size is either 2 or 3. If both of these fail it is back to other espionage.

Let's start with a block size of 2. Since,

$$\text{CRYPTOGRAPHYISCOOL} \text{ — } 2\ 17\ 24\ 15\ 19\ 14\ 6\ 17\ 0\ 15\ 7\ 24\ 8\ 18\ 2\ 14\ 14\ 11$$

encrypts as

$$\text{JMSJPPTYILXPEMGFLG} \text{ — } 9\ 12\ 18\ 9\ 15\ 15\ 19\ 24\ 8\ 11\ 23\ 15\ 4\ 12\ 6\ 5\ 11\ 6$$

we know that

$$[2\ 17] \longrightarrow [9\ 12] \qquad [24\ 15] \longrightarrow [18\ 9] \qquad [19\ 14] \longrightarrow [15\ 15] \cdots$$

So we want to construct a 2×2 matrix out of the plaintext blocks that is invertible modulo 26. If we take the first two blocks 2 17 and 24 15 and build a matrix from them

$$M = \begin{bmatrix} 2 & 17 \\ 24 & 15 \end{bmatrix}$$

then $\det(M) = -378$ which is not relatively prime to 26, so M is not invertible. Actually, we should have seen this coming with what we know about determinants, notice that the first column has all even numbers in it and 2 is a factor of 26.

If we move on to the next block, 19 14 and keep the first block our M matrix becomes,

$$M = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}$$

then $\det(M) = -295$ which is 17 when we take it modulo 26. Since 17 has an inverse modulo 26 we are in business. Our equation becomes,

$$\begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix} E = \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix}$$

So

$$E = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}^{-1} \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} = \begin{bmatrix} 10 & 25 \\ 5 & 20 \end{bmatrix} \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} = \begin{bmatrix} 23 & 1 \\ 7 & 22 \end{bmatrix}$$

To see if this works we test it on our plaintext message CRYPTOGRAPHYISCOOL.

$$C = ME = \begin{bmatrix} 2 & 17 \\ 24 & 15 \\ 19 & 14 \\ 6 & 17 \\ 0 & 15 \\ 7 & 24 \\ 8 & 18 \\ 2 & 14 \\ 14 & 11 \end{bmatrix} \begin{bmatrix} 23 & 1 \\ 7 & 22 \end{bmatrix} = \begin{bmatrix} 9 & 12 \\ 7 & 16 \\ 15 & 15 \\ 23 & 16 \\ 1 & 18 \\ 17 & 15 \\ 24 & 14 \\ 14 & 24 \\ 9 & 22 \end{bmatrix}$$

As we can see, rows 1 and 3 encrypt correctly, they should since those were the ones we used, but the rest of the encryption is incorrect. Conclusion, the block size is not 2. So we move on to block size 3. If it is then there is a 3×3 matrix E with

$$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix} E = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$$

As before, we want to select three rows from our message matrix that will produce an invertible 3×3 matrix. Looking at the last column, all entries except for 11 are even, so the last row must be used. Since the last row first column is even we must have at least one odd number in the first column of the rows we use. So we could not select the remaining two rows from rows 1, 3, and 5. Furthermore, row 5 is all even so selecting it would be pointless. So in our selection we must have row 6 and at least one of rows 2 and 4. We will try rows 1, 2, and 6. This gives,

$$M = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}$$

Since $\det(M) = -791 \equiv 15 \pmod{26}$, we know that M is invertible. So

$$ME = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix} E = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix}$$

and then,

$$E = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}^{-1} \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} = \begin{bmatrix} 13 & 3 & 8 \\ 9 & 12 & 10 \\ 24 & 14 & 15 \end{bmatrix} \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Now, we know this is correct but Eve would still need to check her possible solution, doing so she would get,

$$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix} \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$$

which checks with the ciphertext and she has found the encryption matrix, E.

Δ

If Eve did not have a crib or could not do a known plaintext or known ciphertext attack she might still be able break the cipher with a ciphertext only attack by looking at n-gram frequencies if there was a sufficient amount of ciphertext to go on. This would, of course, be much more difficult to do as the size of the encryption matrix increases.

2.14.6 The Hill Cipher from a Linear Algebra Point of View

We are going to abstract the Hill cipher a little bit and relate it to the material that is commonly covered in a linear algebra class. Most of this will be done in the exercises but we will give a quick introduction here along with some notation.

From the description of the algorithm in this section, it is clear that the Hill cipher can be viewed as a linear transformation. In fact, it is an invertible matrix transformation. Let's put this observation into a more mathematical context. First of all, we are no longer working over the real numbers. We are working with the integers 0–25 with addition and multiplication being done modulo 26.

Notation: Let $\mathbb{A}$ denote the set $\{0, 1, 2, \ldots, 25\}$ where addition and multiplication are done modulo 26.

We use $\mathbb{A}$ as our notation here to designate our "alphabet". If you have taken a course in abstract algebra you would call this structure a ring and denote it as $\mathbb{Z}_{26}$ or $\mathbb{Z}/26\mathbb{Z}$. We will not need this notation nor will we take a digression into ring theory.

The key to a Hill cipher is an $n \times n$ matrix which has an inverse (modulo 26). So it sends n-dimensional vectors with entries from $\mathbb{A}$ to n-dimensional vectors with entries from $\mathbb{A}$.

Notation: Let $\mathbb{A}^n$ as the set of n-dimensional vectors with entries from $\mathbb{A}$

With this notation, the Hill cipher can be viewed simply as an invertible linear transformation $E : \mathbb{A}^n \to \mathbb{A}^n$. Its decryption is also an invertible linear transformation $D : \mathbb{A}^n \to \mathbb{A}^n$. From the Invertible Matrix Theorem in linear algebra, there are many properties of an invertible matrix and the transformation it induces. Some of these properties are crucial when it comes to cryptography. For example, say that E was not invertible but we used it as an encryption matrix for a Hill cipher. One immediate problem comes to mind in that if E is not invertible, finding the decryption matrix D is not possible, hence Bob has a bit of a problem when he receives the ciphertext from Alice. We also know that when we consider E as a linear transformation, this transformation is not one-to-one nor is it onto.

If the transformation is not one-to-one then it is many-to-one which means that there are different vectors in the domain that are mapped to the same vector in the range. Since the domain is associated with the plaintext and the range with the ciphertext this translates to having the two different plaintext words (or segments of words) encrypt to the same ciphertext block. So if we could devise a way to decrypt the message this block of ciphertext would decrypt as two or more possible plaintext blocks. In this situation, it may be possible to figure out which one of the possible decryptions is the correct one but there may be cases where this is not possible either.

Example 68: Say that Alice chooses the following matrix for her Hill cipher,

$$E = \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix}$$

As we can see, the first row is a multiple of 2 and hence this matrix will not be invertible modulo 26. If we take the determinant modulo 26 we get,

$$\begin{vmatrix} 2 & 4 \\ 3 & 5 \end{vmatrix} \pmod{26} = 24$$

and since the GCD of 24 and 26 is not 1 we have verified that E does not have an inverse. Now say that Alice wants to send the message HELP to Bob. So Alice encodes the message as usual, HELP becomes 7 4 11 15, which produces the message matrix

$$M = \begin{bmatrix} 7 & 4 \\ 11 & 15 \end{bmatrix}$$

Then, modulo 26,

$$ME = \begin{bmatrix} 7 & 4 \\ 11 & 15 \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} = \begin{bmatrix} 0 & 22 \\ 15 & 15 \end{bmatrix}$$

Which produces the ciphertext AWPP, which Alice sends to Bob.

Now Bob receives the ciphertext AWPP from Alice and when he goes to decrypt the message he finds out that the encryption matrix E is not invertible modulo 26. So what does he do? Since Bob is not the type to give up, he starts to reason this through. He knows that the first two letters of the message must encrypt to AW and he knows the encryption matrix E, this gives him the equation,

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} = \begin{bmatrix} 0 & 22 \end{bmatrix}$$

So he can find all of the letters xy that encrypt to AW by solving this system. He knows that E is not invertible, hence the encoding transformation is many-to-one and therefore he will get several possible solutions. Unlike when working over the real or complex numbers he is working modulo 26 and so there will not be an infinite number of solutions, so he will have only a finite number of possibilities to worry about. He then solves the system, remember he is working modulo 26.

If we multiply out the above system we get

$$\begin{aligned} 2x + 3y &= 0 \\ 4x + 5y &= 22 \end{aligned}$$

giving us the following matrix to reduce.

$$\begin{bmatrix} 2 & 3 & 0 \\ 4 & 5 & 22 \end{bmatrix}$$

$$\begin{aligned}\begin{bmatrix} 2 & 3 & 0 \\ 4 & 5 & 22 \end{bmatrix} &\xrightarrow{24R_1+R_2} \begin{bmatrix} 2 & 3 & 0 \\ 0 & 25 & 22 \end{bmatrix} \\ &\xrightarrow{25R_2} \begin{bmatrix} 2 & 3 & 0 \\ 0 & 1 & 4 \end{bmatrix} \\ &\xrightarrow{23R_2+R_1} \begin{bmatrix} 2 & 0 & 14 \\ 0 & 1 & 4 \end{bmatrix}\end{aligned}$$

At this point it is very tempting to divide the first row by 2 and get the solution $(7, 4)$ which would be a correct decipherment but incorrect mathematics. We cannot divide by 2 since 2 does not have an inverse modulo 26. What we really need to do is solve the equation $2x = 14$ modulo 26, where there are two solutions, 7 and 20. So the first two letters are either 7 4 or 20 4, that is, either HE or UE. The number of English words that begin with UE is surprisingly small, in fact, in my Webster's Unabridged Dictionary there is only one such word, Uele, which is a river in central Africa that flows 700 miles from Zaire to the Ubangi river. So Bob probably figures that the letters are HE but we will keep all of our options open at this point, who knows, Alice may be telling Bob she plans to take a trip to the Uele river. Now he attacks the last two letters in the same way.

$$\begin{aligned}\begin{bmatrix} 2 & 3 & 15 \\ 4 & 5 & 15 \end{bmatrix} &\xrightarrow{24R_1+R_2} \begin{bmatrix} 2 & 3 & 15 \\ 0 & 25 & 11 \end{bmatrix} \\ &\xrightarrow{25R_2} \begin{bmatrix} 2 & 3 & 15 \\ 0 & 1 & 15 \end{bmatrix} \\ &\xrightarrow{23R_2+R_1} \begin{bmatrix} 2 & 0 & 22 \\ 0 & 1 & 15 \end{bmatrix}\end{aligned}$$

Again we stop at this point, solving $2x = 22$ modulo 26 gives us two solutions, 11 and 24. So our last two letters are either LP or YP. So the possibilities are HELP, HEYP, UELP, or UEYP. Under the assumption that Alice sent an English word we would guess that the plaintext for the message was HELP.
A few things to note here,

1. If this message had been longer the HEYP option could have been valid if the original message was "Hey P...", such as in "Hey Paul...".

2. In the reductions we did above we had to stop where we did since 2 does not have an inverse modulo 26.

3. If we take a closer look at our two-letter decryptions, HE and UE, that is, 7 4 and 20 4. Note that,
$$\begin{bmatrix} 20 & 4 \end{bmatrix} - \begin{bmatrix} 7 & 4 \end{bmatrix} = \begin{bmatrix} 13 & 0 \end{bmatrix}$$

and

$$\begin{bmatrix} 13 & 0 \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

We could have also come to this conclusion by solving the homogeneous system,

$$\begin{bmatrix} 2 & 3 & 0 \\ 4 & 5 & 0 \end{bmatrix} \xrightarrow{24R_1+R_2} \begin{bmatrix} 2 & 3 & 0 \\ 0 & 25 & 0 \end{bmatrix}$$
$$\xrightarrow{25R_2} \begin{bmatrix} 2 & 3 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$
$$\xrightarrow{23R_2+R_1} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

From this we know that $2x = 0$ and so $x = 0$ or $x = 13$. So the vectors that get sent to the zero vector are

$$\begin{bmatrix} 13 & 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & 0 \end{bmatrix}$$

In other words, thinking of E as the transformation,

$$\ker(E) = \left\{ \begin{bmatrix} 0 & 0 \end{bmatrix}, \begin{bmatrix} 13 & 0 \end{bmatrix} \right\}$$

From linear algebra, we know that the entire set of solutions to a consistent non-homogeneous system of equations can be written as a single solution to the non-homogeneous system plus any element of the kernel (or null space) of the coefficient matrix. So here that translates to,

$$\begin{bmatrix} 7 & 4 \end{bmatrix} + \begin{bmatrix} 0 & 0 \end{bmatrix} = \begin{bmatrix} 7 & 4 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 7 & 4 \end{bmatrix} + \begin{bmatrix} 13 & 0 \end{bmatrix} = \begin{bmatrix} 20 & 4 \end{bmatrix}$$

and

$$\begin{bmatrix} 11 & 15 \end{bmatrix} + \begin{bmatrix} 0 & 0 \end{bmatrix} = \begin{bmatrix} 11 & 15 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 11 & 15 \end{bmatrix} + \begin{bmatrix} 13 & 0 \end{bmatrix} = \begin{bmatrix} 24 & 15 \end{bmatrix}$$

Although, this observation may not have saved us too many calculations here, but in some cases, especially with larger block sizes, this method could save a lot of work.

Δ

The above discussion and example shows us why we should use an invertible matrix as our encryption matrix for the Hill cipher algorithm. Furthermore, it puts both the theory and calculations we have been doing into the language of linear algebra.

Let's take a quick look at Eve's job, the cryptanalysis process. Recall that we did a known plaintext attack on the system to find the encryption matrix, and hence the decryption matrix. In this discussion, we will be referring to the "Cryptography is Cool" example from before, we reproduce portions of it in the example below.

Example 69: Recall that Eve knows the plaintext and the corresponding ciphertext of the message,

CRYPTOGRAPHYISCOOL — 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11 encrypts as

JMSJPPTYILXPEMGFLG — 9 12 18 9 15 15 19 24 8 11 23 15 4 12 6 5 11 6

When she started with block size of 2, she came up with the vector correspondence,

$$[2\ 17] \longrightarrow [9\ 12] \qquad [24\ 15] \longrightarrow [18\ 9] \qquad [19\ 14] \longrightarrow [15\ 15] \cdots$$

Then proceeded to build a 2×2 matrix out of the plaintext blocks (vectors) that was invertible modulo 26. Obtaining, after some work,

$$M = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}$$

which was invertible. Then knowing that,

$$\begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix} E = \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix}$$

She calculated,

$$E = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}^{-1} \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} = \begin{bmatrix} 10 & 25 \\ 5 & 20 \end{bmatrix} \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} = \begin{bmatrix} 23 & 1 \\ 7 & 22 \end{bmatrix}$$

Which turned out not to work for the message, telling her that the block size of 2 was incorrect. When she moved on to block size 3, she constructed,

$$M = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}$$

which was invertible. So

$$ME = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix} E = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix}$$

and then,

$$E = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}^{-1} \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} = \begin{bmatrix} 13 & 3 & 8 \\ 9 & 12 & 10 \\ 24 & 14 & 15 \end{bmatrix} \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

And this matrix worked for the entire message and hence we found the key, the encryption matrix, E.

Constructing an invertible matrix M from the message to calculate the encryption matrix makes perfect sense arithmetically. We know that

$$ME = C$$

so if M is invertible, we have,

$$E = (M^{-1}M)E = M^{-1}(ME) = M^{-1}C$$

and we are done. Let's also think about this in terms of vectors and transformations. We are trying to determine the transformation E by what the transformation does. That is, we know some vectors in the domain (the plaintext) and what those vectors get sent to in the range (the ciphertext). The rows of M are our domain vectors and the rows of C are where those vectors are mapped. If M is invertible, the Invertible Matrix Theorem tells us that the rows of M are linearly independent. Since there are n row vectors in M and the dimension of our domain space is n we also know that the rows of M form a basis to $\mathbb{A}^n$. We also know that any linear transformation is completely determined by what it does to a basis for the space. Hence we know that if we can find a set of linearly independent vectors from the plaintext message, and hence construct M, we will have enough information to completely determine E.

This also tells us that if we cannot find a set of linearly independent vectors from the plaintext then we do not have a basis for $\mathbb{A}^n$ and subsequently we will not be able to determine E. We may, in some cases, be able to determine a set of possibilities for E, as we did for the message HELP in the previous example. If we were to try to determine E from a non-basis set of vectors we would need to set up a system of equations and solve the system. Since the set is not a basis we would be guaranteed to have at least one free variable in the solution and hence a set of possible solutions for E, each of which would need to be tested on the plaintext to see which one, or ones, produced the corresponding ciphertext. One positive note here is that since we are working modulo 26 we will have only a finite number of possible matrices E to test, just like we had only four possibilities for the decryption of the message in the previous example. Δ

2.14.7 Cryptography Explorer

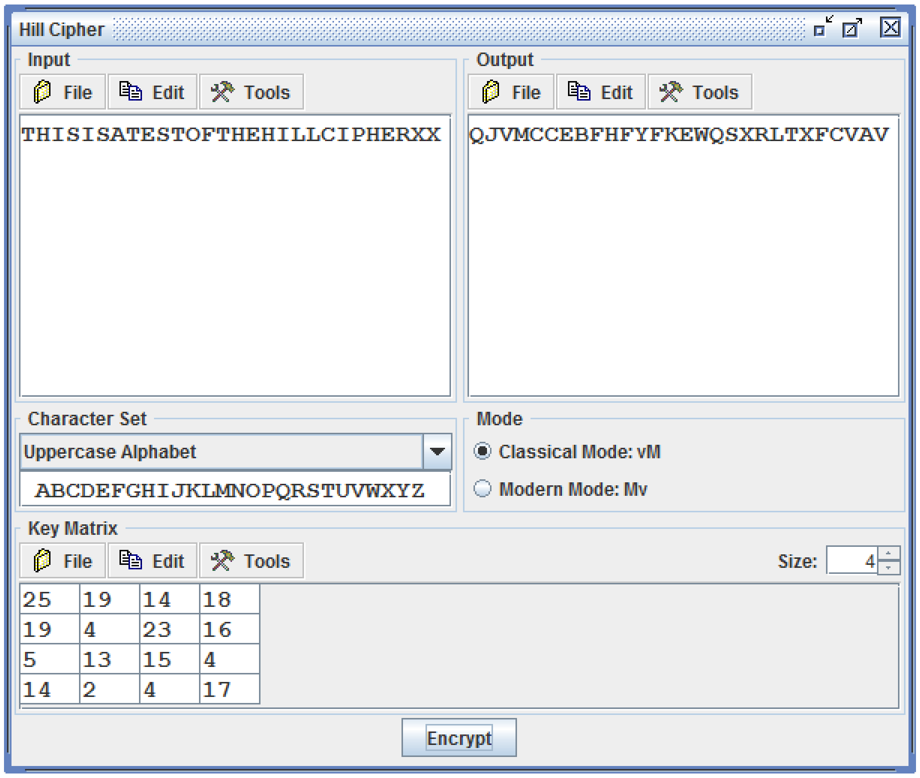
When working with the Cryptography Explorer program with a Hill cipher there are three tools you will need, the Hill Cipher tool, the Modular Matrix Calculator and the Integer Calculator. In fact, unless you are going to find the inverses of the encryption matrices using the reduction tools or you are working with a non-invertible encryption matrix you will probably not need the Integer Calculator.

We will first go over the tools in general and then do an example at the end using them. For each of these tools you can find a more detailed description of the options in the appendix on the Cryptography Explorer program.

Hill Cipher Tool

How to Use the Tool

To Encrypt —



Input: THISISATESTOFTHEHILLCIPHERXX

Output: QJVMCCEBFHFYFKEWQSXRLTXFCVAV

Character Set: Uppercase Alphabet

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Mode: [✓ Classical Mode: vM] Modern Mode: Mv

Key Matrix — Size: 4

25	19	14	18
19	4	23	16
5	13	15	4
14	2	4	17

Figure 2.77: Hill Cipher Tool

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. Select the mode that you wish to use, either classical or modern. Classical computes $\mathbf{v}E = \mathbf{w}$ and modern computes $E\mathbf{v} = \mathbf{w}$.
4. 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. Select the mode that you wish to use, either classical or modern. Classical computes $\mathbf{v}E = \mathbf{w}$ and modern computes $E\mathbf{v} = \mathbf{w}$.
4. 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.

- To change the size of the matrix, there is a size selection to the right of the menu.

- 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 $n \times n$).

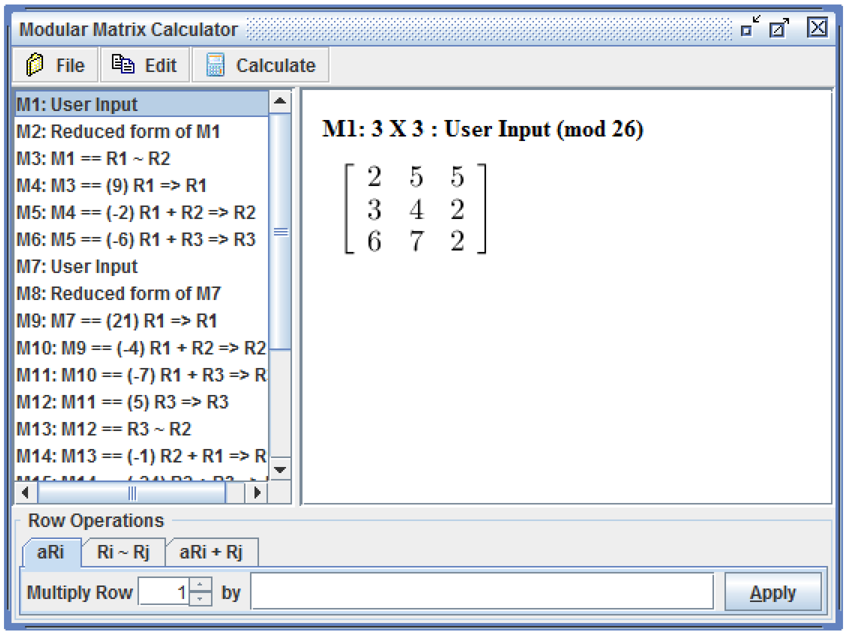
2. Translate each of these blocks into a row (or column) vector **v**. The correspondence of letters to numbers is done by the position of the letter in the character set. So if the character set is the uppercase alphabet (ABCDEFGHIJKLMNOPQRSTUVWXYZ) then A = 0, B = 1, C = 2, and so on. If, in the other 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 $\mathbf{w} = \mathbf{v}M$, if you are using classical mode. If you are in modern mode it will apply the matrix on the left, that is $\mathbf{w} = M\mathbf{v}$.
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.

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.



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)

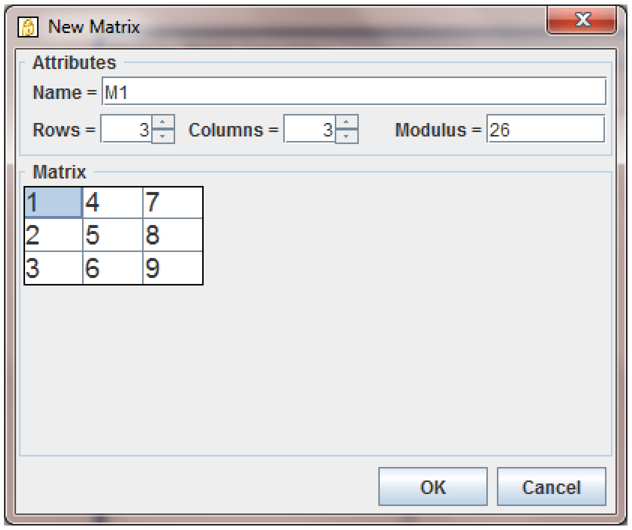
$$\begin{bmatrix} 2 & 5 & 5 \\ 3 & 4 & 2 \\ 6 & 7 & 2 \end{bmatrix}$$

Figure 2.78: Modular Matrix Calculator

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.

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 2.79: 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 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.

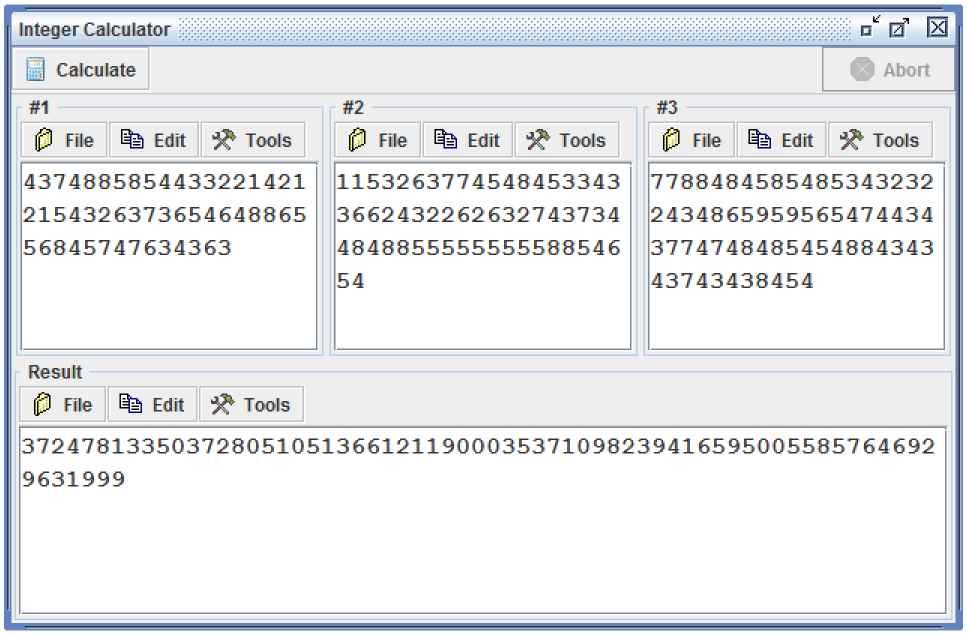
Transpose: This will transpose the current matrix.

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.

Integer Calculator

The Integer Calculator is a simple infinite precision integer arithmetic tool. We will not go into all of the features of the Integer Calculator here since for Hill cipher exercises you will probably only be using it to find the inverse of a number modulo n, which we will calculate a couple ways. For a more detailed description of the features of this tool please see the section on the Integer Calculator in the appendix for the Cryptography Explorer program.



```
4374885854433221421
2154326373654648865
56845747634363
```

```
1153263774548453343
3662432262632743734
4848855555555588546
54
```

```
7788484585485343232
2434865959565474434
3774748485454884343
43743438454
```

```
3724781335037280510513661211900035371098239416595005585764692
9631999
```

Figure 2.80: 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.

Finding Inverses Modulo n

There are two main ways to find the inverse of a number modulo n, if the inverse exists. As an example, we will use the tool to calculate the inverse of 3 modulo 26, specifically $3^{-1} \pmod{26}$.

- In the Integer Calculator put 3 into input #1, −1 into input #2 and 26 into input #3. Then select Calculate > Modular Arithmetic > #1 ^ #2 Mod(#3). At this point, the Result box should display 9, which is the inverse of 3 modulo 26.
- In the Integer Calculator put the following command into input #1, `powermod(3,-1,26)` and then select Calculate > Evaluate > Evaluate #1. At this point, the Result box should display 9, which is the inverse of 3 modulo 26.

Examples

Let's look at a couple examples of using the Cryptography Explorer tools.

Example 70: Say Alice wants to send Bob the message "Cryptography is cool!" using the hill cipher with encryption matrix,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

1. Open the Hill Cipher tool by selecting Ciphers > Hill from the main menu.
2. Copy or type Cryptography is cool! into the input box.
3. Use the tools for the input box to convert the message to uppercase, remove the white space, and remove the punctuation.
4. Select Classical Mode in the mode options.
5. In the Key Matrix, make sure that the size is set to 3, then input the above matrix E into the matrix grid. In the Tools menu for the Key Matrix there is an option for checking if the matrix is invertible, specifically Check for Inverse Matrix. Although the program will encrypt using non-invertible matrices, use this option to check that the matrix is in fact invertible, it should be.
6. Click the Encrypt key and the ciphertext JMSJPPTYILXPEMGFLG will appear in the output box.

Δ

Example 71: Now say that Bob receives the message JMSJPPTYILXPEMGFLG from Alice and knows that the encryption matrix was,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Bob knows that he needs the inverse of the matrix E, modulo 26. There are several ways to do this with the Cryptography Explorer program, we will use the easiest one here and then examine some alternatives in the next example.

1. Open up the Modular Matrix Calculator by selecting, Tools > Calculators > Modular Matrix Calculator from the main menu.

2. Select Edit > New Matrix from the tool's menu.

3. In the dialog box that appears, make sure that the rows and columns are set to 3 each, put in 26 for the modulus, and then input the matrix E into the grid. Then click OK. At this point the matrix will be loaded into the workspace of the matrix calculator.

4. Select Calculate > Invert from the menu and the inverse matrix (modulo 26) will be loaded into the workspace, you should have,

$$\begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$$

   Now we are ready to decrypt the message.

5. Open the Hill Cipher tool by selecting Ciphers > Hill from the main menu.

6. Copy or type the ciphertext JMSJPPTYILXPEMGFLG into the input box.

7. Select Classical Mode in the mode options.

8. In the Key Matrix, make sure that the size is set to 3, then input the above inverse matrix into the matrix grid. Note that you can copy and past the matrix from the matrix calculator by selecting Edit > Copy Matrix from the menu of the calculator, select the upper left cell of the grid in the Hill Cipher tool, and press Ctrl+V.

9. Click the Encrypt key and the plaintext CRYPTOGRAPHYISCOOL will appear in the output box.

Δ

Example 72: In the above example we used the Modular Matrix Calculator to find the inverse of the encryption matrix E. The easiest way to do this is by the Invert command from the menu system. So why would we want to use a different, and probably more difficult method? Depending on what course this section is being integrated into, the instructor may wish that you practice other calculation skills. For example, if this is being integrated into a course that does not assume that you know any linear algebra or matrix theory then most likely the instructor will have you calculate the inverse as we did above. On the other hand, if this is being integrated into a linear algebra class, or a class that assumes a knowledge of linear algebra, then the instructor may wish use this as an example of how the topics and techniques in linear algebra work when we are doing our calculations modulo 26 instead of over the real numbers.

Recall from linear algebra that if a square matrix, M, is invertible then it can be reduced to the identity matrix. Furthermore, the row operations that are done in reducing M to the identity will also convert the identity matrix to the inverse of M. So if we construct the $n \times 2n$ matrix that has M in the left half and the $n \times n$ identity in the right half, notationally, $[M|I]$, and reduce this matrix to reduced row echelon form we will get the matrix $[I|M^{-1}]$. So our decryption matrix will be the right side $n \times n$ matrix after the reduction.

The two methods we outline below use this idea, the first does the reduction to reduced row echelon form using a single command from the menu and the second uses the reduction interface that is built into the calculator to do each row reduction operation.

- Finding the Inverse using the Reduce Command:

  The exercise is to find the inverse (modulo 26) of the following matrix.

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

  1. Open the Modular Matrix Calculator and input the following 3×6 matrix with modulus 26,
$$\begin{bmatrix} 2 & 3 & 15 & 1 & 0 & 0 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 1 & 13 & 4 & 0 & 0 & 1 \end{bmatrix}$$
  2. Select Calculate > Reduce from the tool's menu and you will get the following output,
$$\begin{bmatrix} 1 & 0 & 0 & 10 & 19 & 16 \\ 0 & 1 & 0 & 4 & 23 & 7 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$
     This tells you that the inverse matrix is,
$$\begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$$

- Finding the Inverse using Matrix Reduction:

The exercise is to find the inverse (modulo 26) of the following matrix. This time we will do the matrix reduction step by step using the Modular Matrix Calculator to help with the arithmetic.

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

1. Open the Modular Matrix Calculator and input the following 3×6 matrix with modulus 26,

$$\begin{bmatrix} 2 & 3 & 15 & 1 & 0 & 0 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 1 & 13 & 4 & 0 & 0 & 1 \end{bmatrix}$$

2. Since we are going through the reduction process step by step we will want to use the row operations panel. Select Calculate > Show/Hide Row Operations Panel. This will toggle the Row Operations Panel at the bottom. The Row Operations Panel has three tabs, each for one of the three elementary row operations. The first tab multiplies a row by a constant value, the second tab interchanges two rows and the third multiplies one row by a constant and adds it to another row. All calculations are done modulo the modulus of the matrix that is being operated on.

   The row selectors are numeric selection boxes that are restricted to the rows in the matrix and the other input boxes are general text input boxes but they are expecting an integer input.

3. Since 2 is not invertible modulo 26 we need to either interchange rows 1 and 2 or rows 1 and 3, since the first entry in row 3 is a 1 we will interchange rows 1 and 3. Select the second tab, `Ri ~ Rj`, set the numeric selectors to 1 and 3 respectively, and click the Apply button. Now the displayed matrix is,

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 2 & 3 & 15 & 1 & 0 & 0 \end{bmatrix}$$

4. Now we will "zero out" the first column under the 1. Select the third tab `aRi + Rj`, select the rows and inputs so that the display reads,

   `Multiply Row 1 by -5 and add to row 2`

   then click Apply. Now select the rows and inputs so that the display reads,

   `Multiply Row 1 by -2 and add to row 3`

   then click Apply. Now the displayed matrix is,

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 0 & 21 & 18 & 0 & 1 & 21 \\ 0 & 3 & 7 & 1 & 0 & 24 \end{bmatrix}$$

Note that the program automatically converted the -5 and -2 to 21 and 24 respectively, since our calculations are done modulo 26. We could have used 21 and 24 in place of -5 and -2.

5. At this point we want a 1 in the second row and second column. Since 21 is invertible modulo 26, $\gcd(21, 26) = 1$, we can use the integer calculator to calculate $21^{-1} \pmod{26} \equiv 5$. Just use either method described above to do this calculation. Now select the first tab `aRi`, set the row to 2 and input 5 into the multiplier box, and then click Apply. The displayed matrix is now,

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 3 & 7 & 1 & 0 & 24 \end{bmatrix}$$

6. Now we will "zero out" the second column, other than the 1. Select the third tab `aRi + Rj`, select the rows and inputs so that the display reads,

```
Multiply Row 2 by -13 and add to row 1
```

then click Apply. Now select the rows and inputs so that the display reads,

```
Multiply Row 2 by -3 and add to row 3
```

then click Apply. Now the displayed matrix is,

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 13 & 14 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 0 & 23 & 1 & 11 & 21 \end{bmatrix}$$

As with the previous use of this option, we could have used 13 and 23 in place of -13 and -3 respectively.

7. At this point we want a 1 in the third row and third column. Since 23 is invertible modulo 26, $\gcd(23, 26) = 1$, we can use the integer calculator to calculate $23^{-1} \pmod{26} \equiv 17$. Just use either method described above to do this calculation. Now select the first tab `aRi`, set the row to 3 and input 17 into the multiplier box, and then click Apply. The displayed matrix is now,

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 13 & 14 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$

8. Now we will "zero out" the third column, other than the 1. Select the third tab `aRi + Rj`, select the rows and inputs so that the display reads,

```
Multiply Row 3 by -4 and add to row 1
```

then click Apply. Now select the rows and inputs so that the display reads,

```
Multiply Row 3 by -12 and add to row 2
```

then click Apply. Now the displayed matrix is,

$$\begin{bmatrix} 1 & 0 & 0 & 10 & 19 & 16 \\ 0 & 1 & 0 & 4 & 23 & 7 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$

As before, we could have used 22 and 14 in place of -4 and -12 respectively. We have finished the reduction process to the form $[I|M]$ which tells us that the inverse matrix (modulo 26) is

$$\begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$$

The entire workspace for this process is as follows,

```
M1: 3 X 6 : User Input (mod 26)
```

$$\begin{bmatrix} 2 & 3 & 15 & 1 & 0 & 0 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 1 & 13 & 4 & 0 & 0 & 1 \end{bmatrix}$$

```
M2: 3 X 6 : M1 == R1 ~ R3  (mod 26)
```

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 2 & 3 & 15 & 1 & 0 & 0 \end{bmatrix}$$

```
M3: 3 X 6 : M2 == (-5) R1 + R2 => R2  (mod 26)
```

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 0 & 21 & 18 & 0 & 1 & 21 \\ 2 & 3 & 15 & 1 & 0 & 0 \end{bmatrix}$$

```
M4: 3 X 6 : M3 == (-2) R1 + R3 => R3  (mod 26)
```

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 0 & 21 & 18 & 0 & 1 & 21 \\ 0 & 3 & 7 & 1 & 0 & 24 \end{bmatrix}$$

```
M5: 3 X 6 : M4 == (5) R2 => R2  (mod 26)
```

$$\begin{bmatrix} 1 & 13 & 4 & 0 & 0 & 1 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 3 & 7 & 1 & 0 & 24 \end{bmatrix}$$

```
M6: 3 X 6 : M5 == (-13) R2 + R1 => R1  (mod 26)
```

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 13 & 14 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 3 & 7 & 1 & 0 & 24 \end{bmatrix}$$

```
M7: 3 X 6 : M6 == (-3) R2 + R3 => R3  (mod 26)
```

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 13 & 14 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 0 & 23 & 1 & 11 & 21 \end{bmatrix}$$

```
M8: 3 X 6 : M7 == (17) R3 => R3  (mod 26)
```

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 13 & 14 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$

```
M9: 3 X 6 : M8 == (-4) R3 + R1 => R1  (mod 26)
```

$$\begin{bmatrix} 1 & 0 & 0 & 10 & 19 & 16 \\ 0 & 1 & 12 & 0 & 5 & 1 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$

```
M10: 3 X 6 : M9 == (-12) R3 + R2 => R2  (mod 26)
```

$$\begin{bmatrix} 1 & 0 & 0 & 10 & 19 & 16 \\ 0 & 1 & 0 & 4 & 23 & 7 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$$

Δ

The Cryptography Explorer program can also help with the breaking of a Hill Cipher. We will redo the example from the section on breaking the Hill cipher.

Example 73: Eve has intercepted the encrypted message JMSJPPTYILXPEMGFLG and from other espionage knows that this message was CRYPTOGRAPHYISCOOL. She also knows that other messages sent that day between Alice and Bob are using the same key and she wishes to decrypt them as well, but she has no other information about these other messages.

Since the message size 18 she knows that that block size must be 2, 3, 6, 9 or 18, and since she has only 18 characters to work with her only hope is that the block size is either 2 or 3. If both of these fail it is back to other espionage.

Let's start with a block size of 2. Since,

CRYPTOGRAPHYISCOOL — 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

encrypts as

JMSJPPTYILXPEMGFLG — 9 12 18 9 15 15 19 24 8 11 23 15 4 12 6 5 11 6 we know that

$$[2\ 17] \longrightarrow [9\ 12] \qquad [24\ 15] \longrightarrow [18\ 9] \qquad [19\ 14] \longrightarrow [15\ 15] \cdots$$

So we want to construct a 2×2 matrix out of the plaintext blocks that is invertible modulo 26. If we take the first two blocks 2 17 and 24 15 and build a matrix from them

$$M = \begin{bmatrix} 2 & 17 \\ 24 & 15 \end{bmatrix}$$

then $\det(M) = -378$ which is not relatively prime to 26, so M is not invertible. Actually, we should have seen this coming with what we know about determinants, notice that the first column has all even numbers in it and 2 is a factor of 26.

If we move on to the next block, 19 14 and keep the first block our M matrix becomes,

$$M = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}$$

then $\det(M) = -295$ which is 17 when we take it modulo 26. Since 17 has an inverse modulo 26 we are in business. Our equation becomes,

$$\begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix} E = \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix}$$

So

$$E = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}^{-1} \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} = \begin{bmatrix} 10 & 25 \\ 5 & 20 \end{bmatrix} \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} = \begin{bmatrix} 23 & 1 \\ 7 & 22 \end{bmatrix}$$

To see if this works we test it on our plaintext message CRYPTOGRAPHYISCOOL.

$$C = ME = \begin{bmatrix} 2 & 17 \\ 24 & 15 \\ 19 & 14 \\ 6 & 17 \\ 0 & 15 \\ 7 & 24 \\ 8 & 18 \\ 2 & 14 \\ 14 & 11 \end{bmatrix} \begin{bmatrix} 23 & 1 \\ 7 & 22 \end{bmatrix} = \begin{bmatrix} 9 & 12 \\ 7 & 16 \\ 15 & 15 \\ 23 & 16 \\ 1 & 18 \\ 17 & 15 \\ 24 & 14 \\ 14 & 24 \\ 9 & 22 \end{bmatrix}$$

As we can see, rows 1 and 3 encrypt correctly, they should since those were the ones we used, but the rest of the encryption is incorrect. Conclusion, the block size is not 2. To do these calculations in the Cryptography Explorer program, open up the Modular Matrix Calculator and do the following.

1. Input the following three matrices, each with modulus 26,

$$\begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix} \qquad \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 2 & 17 \\ 24 & 15 \\ 19 & 14 \\ 6 & 17 \\ 0 & 15 \\ 7 & 24 \\ 8 & 18 \\ 2 & 14 \\ 14 & 11 \end{bmatrix}$$

   We will assume that they are input in this order so that they correspond to matrices M1, M2 and M3 respectively.

2. To calculate the possible encryption matrix E, highlight matrix M1, select Calculate > Invert, now M4 is the inverse of M1 (modulo 26). Select Calculate > Multiply from the menu, at this point a dialog box will appear, select matrix M4 for A and matrix M2 for B using the drop-down selection boxes and click OK. The matrix M5 should be E.

3. To check the validity of the encryption matrix, select Calculate > Multiply from the menu. Select matrix M3 for A and matrix M5 for B using the drop-down selection boxes and click OK. The matrix M6 should be $C = ME$ from above. Your conclusion would be the same as above, the block size is not 2.

So we move on to block size 3. If it is then there is a 3×3 matrix E with

$$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix} E = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$$

As before, we want to select three rows from our message matrix that will produce an invertible 3×3 matrix. Looking at the last column, all entries except for 11 are even, so the last row must be used. Since the last row first column is even we must have at least one odd number in the first column of the rows we use. So we could not select the remaining two rows from rows 1, 3, and 5. Furthermore, row 5 is all even so selecting it would be pointless. So in our selection we must have row 6 and at least one of rows 2 and 4. We will try rows 1, 2, and 6. This gives,

$$M = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}$$

Since $\det(M) = -791 \equiv 15 \pmod{26}$, we know that M is invertible. So

$$ME = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix} E = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix}$$

and then,

$$E = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}^{-1} \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} = \begin{bmatrix} 13 & 3 & 8 \\ 9 & 12 & 10 \\ 24 & 14 & 15 \end{bmatrix} \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Now, we know this is correct but Eve would still need to check her possible solution, doing so she would get,

$$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix} \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$$

which checks with the ciphertext and she has found the encryption matrix, E.

To do the above calculations using the Modular Matrix Calculator do the following,

1. Input the following three matrices, each with modulus 26,

$$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix} \qquad \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} \qquad \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix}$$

   We will assume that they are input in this order so that they correspond to matrices M1, M2 and M3 respectively. If you have matrices in the workspace, select File > New before inputting these three matrices.

2. To calculate the possible encryption matrix E, highlight matrix M1, select Calculate > Invert, now M4 is the inverse of M1 (modulo 26). Select Calculate > Multiply from the menu, at this point a dialog box will appear, select matrix M4 for A and matrix M2 for B using the drop-down selection boxes and click OK. The matrix M5 should be E.

3. To check the validity of the encryption matrix, select Calculate > Multiply from the menu. Select matrix M3 for A and matrix M5 for B using the drop-down selection

boxes and click OK. The matrix M6 should be $C = ME$ from above. The ciphertext matrix checks out, so the block size was three and the encryption matrix is,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Δ

2.14.8 Mathematica

When using a general computer algebra system you will follow the same procedure as you would with the Cryptography Explorer program except that the operations will be done with the computer algebra system commands instead of input dialog boxes and menu selections.

There is an entire section on vectors and matrices in the appendix on Mathematica, we suggest that you read over that section before continuing.

For the Hill cipher the main things you need to be concerned with are, defining a matrix, multiplying two matrices, and inverting a square matrix. We will quickly recap these below.

Defining a Matrix and a Vector

A vector is simply a list and a matrix is a list of lists, where 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 discuss matrix operations in detail in the Mathematica appendix, here we just need to know about multiplication. The multiplication operator for matrices in Mathematica is the period. So to multiply matrices `m` and `n` the syntax is `m . n`.

Another interesting feature with Mathematica is that it will treat a vector as both a row vector and a column vector, it simply uses the form that is needed. 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}
```

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}}
```

Which assignment you use will depend on what you intend to do with the matrices. In most cases you will probably want the immediate assignment = instead of the delayed assignment of `:=`.

Another thing you should be aware of is that when you assign a matrix to a variable or assign one matrix to another do not use the `MatrixForm` command or pipe in the assignment statement. The `MatrixForm` command alters the form of the data so that doing matrix operations becomes impossible. So, do the assignment in one command and the `MatrixForm` in another.

Joining Matrices

Depending on how you choose to do some of the matrix operations you may want to join two matrices together or join a matrix and a vector, that is, augment a matrix with a vector. The Mathematica command for joining matrices is `Join`.

```
In[1]:= m = {{1, 2, 1}, {3, 2 - 1, 2}, {4, -2, 1}}

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

In[2]:= m // MatrixForm

Out[2]//MatrixForm=
```

$$\begin{pmatrix} 1 & 2 & 1 \\ 3 & 1 & 2 \\ 4 & -2 & 1 \end{pmatrix}$$

Mathematica as an `IdentityMatrix` command that will produce an identity matrix of any given size.

```
In[3]:= id = IdentityMatrix [3]

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

In[4]:= id // MatrixForm

Out[4]//MatrixForm=
        ⎛ 1  0  0 ⎞
        ⎜ 0  1  0 ⎟
        ⎝ 0  0  1 ⎠
```

The `Join` command takes either two or three arguments. To join two matrices vertically, that is, one over the other simply use `Join[m, n]` where `m` and `n` are matrices.

```
In[5]:= Join[m, id] // MatrixForm

Out[5]//MatrixForm=
        ⎛ 1   2  1 ⎞
        ⎜ 3   1  2 ⎟
        ⎜ 4  -2  1 ⎟
        ⎜ 1   0  0 ⎟
        ⎜ 0   1  0 ⎟
        ⎝ 0   0  1 ⎠
```

To join two matrices horizontally, that is, one beside the other simply use `Join[m, n, 2]` where `m` and `n` are matrices.

```
In[6]:= d = Join[m, id, 2]

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

In[7]:= d // MatrixForm

Out[7]//MatrixForm=
        ⎛ 1   2  1  1  0  0 ⎞
        ⎜ 3   1  2  0  1  0 ⎟
        ⎝ 4  -2  1  0  0  1 ⎠
```

Matrix Arithmetic

For the Hill Cipher, the main matrix operations you need to concentrate on are matrix multiplication and matrix inverses. For the Hill cipher you are really interested in these operations done modulo an integer n, we will discuss modular operations later in this section. For now we will simply look at the commands without a modulus. As we pointed out above, the period is the multiplication operator for matrices. To find an inverse to a square matrix, if it exists, is the `Inverse` command. We can also find the determinant of a matrix with the `Det` command. You can find a more detailed description of other arithmetic operations on matrices in the Mathematica appendix.

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[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}}
```

Finding the inverse of a matrix can be done 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]= $\left\{\left\{\frac{10}{9}, -\frac{2}{9}, \frac{1}{9}\right\}, \left\{-\frac{5}{3}, \frac{1}{3}, \frac{1}{3}\right\}, \left\{-\frac{1}{9}, \frac{2}{9}, -\frac{1}{9}\right\}\right\}$

Matrix Reduction

Depending how you go about the calculations, you may need to reduce a matrix. 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 reduced 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}}
```

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. Inverses require a different method which we will discuss below.

```
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}}
```

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=
```

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

Examples

Let's look at a couple example of using Mathematica.

Example 74: Say Alice wants to send Bob the message "Cryptography is cool!" using the hill cipher with encryption matrix,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

If we convert "Cryptography is cool!" to matrix form we get,

$$M = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix}$$

Since Mathematica reserves the character `E` for the natural base e we need to use another character.

```
In[1]:= H = {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

Out[1]= {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

    In[2]:= H // MatrixForm

Out[2]//MatrixForm=
```

$$\begin{pmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{pmatrix}$$

Put the message matrix into Mathematica, assigned to the variable M,

```
In[3]:= M = {{2, 17, 24}, {15, 19, 14}, {6, 17, 0}, {15, 7, 24},
          {8, 18, 2}, {14, 14, 11}}

Out[3]= {{2, 17, 24}, {15, 19, 14}, {6, 17, 0},
          {15, 7, 24}, {8, 18, 2}, {14, 14, 11}}

    In[4]:= M // MatrixForm

Out[4]//MatrixForm=
```

$$\begin{pmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{pmatrix}$$

Multiply the two matrices, modulo 26,

```
In[5]:= CT = Mod[M.H, 26]

Out[5]= {{9, 12, 18}, {9, 15, 15}, {19, 24, 8},
          {11, 23, 15}, {4, 12, 6}, {5, 11, 6}}

    In[6]:= CT // MatrixForm

Out[6]//MatrixForm=
```

$$\begin{pmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{pmatrix}$$

Then reading the columns we have the numeric list, 9 12 18 9 15 15 19 24 8 11 23 15 4 12 6 5 11 6 which converts to the ciphertext JMSJPPTYILXPEMGFLG. Δ

Example 75: Now say that Bob receives the message JMSJPPTYILXPEMGFLG from Alice and knows that the encryption matrix was,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Bob knows that he needs the inverse of the matrix E, modulo 26. There are several ways to do this with Mathematica, we will use the easiest one here and then examine some alternatives in the next example. First put in the encryption matrix and the ciphertext matrix.

```
In[1]:= H = {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

Out[1]= {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

In[2]:= H // MatrixForm

Out[2]//MatrixForm=
        2   3  15
        5   8  12
        1  13   4

In[3]:= CT = {{9, 12, 18}, {9, 15, 15}, {19, 24, 8}, {11, 23, 15},
          {4, 12, 6}, {5, 11, 6}}

Out[3]= {{9, 12, 18}, {9, 15, 15}, {19, 24, 8},
         {11, 23, 15}, {4, 12, 6}, {5, 11, 6}}

In[4]:= CT // MatrixForm

Out[4]//MatrixForm=
         9  12  18
         9  15  15
        19  24   8
        11  23  15
         4  12   6
         5  11   6
```

Now find the inverse of the encryption matrix modulo 26, this is the decryption matrix.

```
In[5]:= HI = Inverse[H, Modulus → 26]

Out[5]= {{10, 19, 16}, {4, 23, 7}, {17, 5, 19}}
```

```
In[6]:= HI // MatrixForm

Out[6]//MatrixForm=
        ⎛10  19  16⎞
        ⎜ 4  23   7⎟
        ⎝17   5  19⎠
```

Multiply the decryption matrix by the ciphertext matrix to retrieve the message.

```
In[7]:= M = Mod[CT.HI, 26]

Out[7]= {{2, 17, 24}, {15, 19, 14}, {6, 17, 0},
         {15, 7, 24}, {8, 18, 2}, {14, 14, 11}}

In[8]:= M // MatrixForm

Out[8]//MatrixForm=
        ⎛ 2  17  24⎞
        ⎜15  19  14⎟
        ⎜ 6  17   0⎟
        ⎜15   7  24⎟
        ⎜ 8  18   2⎟
        ⎝14  14  11⎠
```

Reading down the columns we get the numeric list, 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11, which converts to the plaintext message CRYPTOGRAPHYISCOOL. Δ

Example 76: The easiest way to find an inverse matrix in Mathematica is with the `Inverse` command. So why would we want to use a different, and probably more difficult method? Depending on what course this section is being integrated into, the instructor may wish that you practice other calculation skills. For example, if this is being integrated into a course that does not assume that you know any linear algebra or matrix theory then most likely the instructor will have you calculate the inverse as we did above. On the other hand, if this is being integrated into a linear algebra class, or a class that assumes a knowledge of linear algebra, then the instructor may wish use this as an example of how the topics and techniques in linear algebra work when we are doing our calculations modulo 26 instead of over the real numbers.

Recall from linear algebra that if a square matrix, M, is invertible then it can be reduced to the identity matrix. Furthermore, the row operations that are done in reducing M to the identity will also convert the identity matrix to the inverse of M. So if we construct the $n \times 2n$ matrix that has M in the left half and the $n \times n$ identity in the right half, notationally, $[M|I]$, and reduce this matrix to reduced row echelon form we will get the matrix $[I|M^{-1}]$. So our decryption matrix will be the right side $n \times n$ matrix after the reduction.

To apply this process in Mathematica, we will do the following. The exercise is to find the inverse (modulo 26) of the following matrix using a reduction approach.

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

First, we input the encryption matrix,

```
In[1]:= H = {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

Out[1]= {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}

In[2]:= H // MatrixForm

Out[2]//MatrixForm=
  ( 2   3  15 )
  ( 5   8  12 )
  ( 1  13   4 )
```

Create a 3×3 identity matrix,

```
In[3]:= id = IdentityMatrix [3]

Out[3]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

Join the encryption matrix to the identity matrix,

```
In[4]:= A = Join [H, id, 2]

Out[4]= {{2, 3, 15, 1, 0, 0}, {5, 8, 12, 0, 1, 0}, {1, 13, 4, 0, 0, 1}}

In[5]:= A // MatrixForm

Out[5]//MatrixForm=
  ( 2   3  15  1  0  0 )
  ( 5   8  12  0  1  0 )
  ( 1  13   4  0  0  1 )
```

Now we would like to reduce this matrix using modulo 26 operations. Unfortunately Mathematica does not allow a composite number to be used as a modulus in the `RowReduce` command. So we will work around this.

```
In[6]:= RowReduce [A , Modulus → 26]

    RowReduce::nmod : {{2, 3, 15, 1, 0, 0}, {0, 1, 1, 21, 2, 0}, {0, 0, 22, 10, 6, 2}} is not valid modulo 26. >>

Out[6]= RowReduce [{{2, 3, 15, 1, 0, 0},
          {5, 8, 12, 0, 1, 0}, {1, 13, 4, 0, 0, 1}}, Modulus → 26]
```

First we do a standard row reduction,

```
In[7]:= RA = RowReduce[A]
```

Out[7]= $\left\{\left\{1, 0, 0, -\frac{124}{583}, \frac{183}{583}, -\frac{84}{583}\right\}, \left\{0, 1, 0, -\frac{8}{583}, -\frac{7}{583}, \frac{51}{583}\right\}, \left\{0, 0, 1, \frac{57}{583}, -\frac{23}{583}, \frac{1}{583}\right\}\right\}$

```
In[8]:= RA // MatrixForm
```

Out[8]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 & -\frac{124}{583} & \frac{183}{583} & -\frac{84}{583} \\ 0 & 1 & 0 & -\frac{8}{583} & -\frac{7}{583} & \frac{51}{583} \\ 0 & 0 & 1 & \frac{57}{583} & -\frac{23}{583} & \frac{1}{583} \end{pmatrix}$$

Extract the last three columns,

```
In[9]:= HIr = RA[[All, 4 ;; 6]]
```

Out[9]= $\left\{\left\{-\frac{124}{583}, \frac{183}{583}, -\frac{84}{583}\right\}, \left\{-\frac{8}{583}, -\frac{7}{583}, \frac{51}{583}\right\}, \left\{\frac{57}{583}, -\frac{23}{583}, \frac{1}{583}\right\}\right\}$

```
In[10]:= HIr // MatrixForm
```

Out[10]//MatrixForm=

$$\begin{pmatrix} -\frac{124}{583} & \frac{183}{583} & -\frac{84}{583} \\ -\frac{8}{583} & -\frac{7}{583} & \frac{51}{583} \\ \frac{57}{583} & -\frac{23}{583} & \frac{1}{583} \end{pmatrix}$$

Since $\gcd(583, 26) = 1$ we know that each of the entries in the above matrix have a value modulo 26. All we need to do is clear out the denominator by multiplying by 583, then multiply by the inverse of 583 modulo 26, which must exist, and then reduce the entries of the matrix modulo 26. First find the inverse of 583 modulo 26 using the `PowerMod` function.

```
In[11]:= PowerMod[583, -1, 26]

Out[11]= 19
```

Multiply the matrix by this inverse, 19, and by 583, and finally reduce the matrix modulo 26.

```
In[12]:= HI = Mod[HIr * 19 * 583, 26]

Out[12]= {{10, 19, 16}, {4, 23, 7}, {17, 5, 19}}
```

```
In[13]:= HI // MatrixForm

Out[13]//MatrixForm=
          ( 10  19  16 )
          (  4  23   7 )
          ( 17   5  19 )
```

Δ

Mathematica can also help with the breaking of a Hill Cipher. We will redo the example from the section on breaking the Hill cipher.

Example 77: Eve has intercepted the encrypted message JMSJPPTYILXPEMGFLG and from other espionage knows that this message was CRYPTOGRAPHYISCOOL. She also knows that other messages sent that day between Alice and Bob are using the same key and she wishes to decrypt them as well, but she has no other information about these other messages.

Since the message size 18 she knows that that block size must be 2, 3, 6, 9 or 18, and since she has only 18 characters to work with her only hope is that the block size is either 2 or 3. If both of these fail it is back to other espionage.

Let's start with a block size of 2. Since,

$$\text{CRYPTOGRAPHYISCOOL} \text{ — } 2\ 17\ 24\ 15\ 19\ 14\ 6\ 17\ 0\ 15\ 7\ 24\ 8\ 18\ 2\ 14\ 14\ 11$$

encrypts as

$$\text{JMSJPPTYILXPEMGFLG} \text{ — } 9\ 12\ 18\ 9\ 15\ 15\ 19\ 24\ 8\ 11\ 23\ 15\ 4\ 12\ 6\ 5\ 11\ 6$$

we know that

$$[2\ 17] \longrightarrow [9\ 12] \qquad [24\ 15] \longrightarrow [18\ 9] \qquad [19\ 14] \longrightarrow [15\ 15] \cdots$$

So we want to construct a 2×2 matrix out of the plaintext blocks that is invertible modulo 26. If we take the first two blocks 2 17 and 24 15 and build a matrix from them

$$M = \begin{bmatrix} 2 & 17 \\ 24 & 15 \end{bmatrix}$$

then $\det(M) = -378$ which is not relatively prime to 26, so M is not invertible. Actually, we should have seen this coming with what we know about determinants, notice that the first column has all even numbers in it and 2 is a factor of 26.

If we move on to the next block, 19 14 and keep the first block our M matrix becomes,

$$M = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}$$

then $\det(M) = -295$ which is 17 when we take it modulo 26. Since 17 has an inverse modulo 26 we are in business. Our equation becomes,

$$\begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix} E = \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix}$$

So

$$E = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}^{-1} \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} = \begin{bmatrix} 10 & 25 \\ 5 & 20 \end{bmatrix} \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} = \begin{bmatrix} 23 & 1 \\ 7 & 22 \end{bmatrix}$$

To see if this works we test it on our plaintext message CRYPTOGRAPHYISCOOL.

$$C = ME = \begin{bmatrix} 2 & 17 \\ 24 & 15 \\ 19 & 14 \\ 6 & 17 \\ 0 & 15 \\ 7 & 24 \\ 8 & 18 \\ 2 & 14 \\ 14 & 11 \end{bmatrix} \begin{bmatrix} 23 & 1 \\ 7 & 22 \end{bmatrix} = \begin{bmatrix} 9 & 12 \\ 7 & 16 \\ 15 & 15 \\ 23 & 16 \\ 1 & 18 \\ 17 & 15 \\ 24 & 14 \\ 14 & 24 \\ 9 & 22 \end{bmatrix}$$

As we can see, rows 1 and 3 encrypt correctly, they should since those were the ones we used, but the rest of the encryption is incorrect. Conclusion, the block size is not 2. To do these calculations in Mathematica, open up a new notebook and do the following. First input the following three matrices,

$$A = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix} \qquad B = \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} \qquad PT = \begin{bmatrix} 2 & 17 \\ 24 & 15 \\ 19 & 14 \\ 6 & 17 \\ 0 & 15 \\ 7 & 24 \\ 8 & 18 \\ 2 & 14 \\ 14 & 11 \end{bmatrix}$$

```
In[1]:= A = {{2, 17}, {19, 14}}

Out[1]= {{2, 17}, {19, 14}}

In[2]:= B = {{9, 12}, {15, 15}}

Out[2]= {{9, 12}, {15, 15}}

In[3]:= PT = {{2, 17}, {24, 15}, {19, 14}, {6, 17}, {0, 15}, {7, 24},
          {8, 18}, {2, 14}, {14, 11}}

Out[3]= {{2, 17}, {24, 15}, {19, 14}, {6, 17},
          {0, 15}, {7, 24}, {8, 18}, {2, 14}, {14, 11}}
```

Find the inverse of matrix A modulo 26.

```
In[4]:= AI = Inverse[A, Modulus → 26]

Out[4]= {{10, 25}, {5, 20}}
```

Multiply this matrix on the right with matrix B, and take the result modulo 26, to get the possible encryption matrix.

```
In[5]:= H = Mod[AI.B, 26]

Out[5]= {{23, 1}, {7, 22}}
```

Finally, multiply this resulting matrix on the left with the plaintext matrix, and take the result modulo 26, to get the possible ciphertext matrix.

```
In[6]:= M = Mod[PT.H, 26]

Out[6]= {{9, 12}, {7, 16}, {15, 15}, {23, 16},
          {1, 18}, {17, 15}, {24, 14}, {14, 24}, {9, 22}}
```

The above calculation was to check the validity of the encryption matrix. If we convert the result to ciphertext letters we get, JMHQPPXQBSRPYOOYJW which is not our ciphertext of JMSJPPTYILXPEMGFLG. Your conclusion would be the same as above, the block size is not 2.

So we move on to block size 3. If it is then there is a 3×3 matrix E with

$$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix} E = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$$

As before, we want to select three rows from our message matrix that will produce an invertible 3×3 matrix. Looking at the last column, all entries except for 11 are even, so the last row must be used. Since the last row first column is even we must have at least one odd number in the first column of the rows we use. So we could not select the remaining two rows from rows 1, 3, and 5. Furthermore, row 5 is all even so selecting it would be pointless. So in our selection we must have row 6 and at least one of rows 2 and 4. We will try rows 1, 2, and 6. This gives,

$$M = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}$$

Since $\det(M) = -791 \equiv 15 \pmod{26}$, we know that M is invertible. So

$$ME = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix} E = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix}$$

and then,

$$E = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}^{-1} \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} = \begin{bmatrix} 13 & 3 & 8 \\ 9 & 12 & 10 \\ 24 & 14 & 15 \end{bmatrix} \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Now, we know this is correct but Eve would still need to check her possible solution, doing so she would get,

$$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix} \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$$

which checks with the ciphertext and she has found the encryption matrix, E. To do the above calculations using Mathematica, open up a new notebook and do the following. First, input the following three matrices.

$$A = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix} \qquad B = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} \qquad PT = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix}$$

```
In[1]:= A = {{2, 17, 24}, {15, 19, 14}, {14, 14, 11}}

Out[1]= {{2, 17, 24}, {15, 19, 14}, {14, 14, 11}}

In[2]:= B = {{9, 12, 18}, {9, 15, 15}, {5, 11, 6}}

Out[2]= {{9, 12, 18}, {9, 15, 15}, {5, 11, 6}}

In[3]:= PT = {{2, 17, 24}, {15, 19, 14}, {6, 17, 0}, {15, 7, 24},
          {8, 18, 2}, {14, 14, 11}}

Out[3]= {{2, 17, 24}, {15, 19, 14}, {6, 17, 0},
         {15, 7, 24}, {8, 18, 2}, {14, 14, 11}}
```

Find the inverse of matrix A modulo 26.

```
In[4]:= AI = Inverse[A, Modulus → 26]

Out[4]= {{13, 3, 8}, {9, 12, 10}, {24, 14, 15}}
```

Multiply this matrix on the right with matrix B, and take the result modulo 26, to get the possible encryption matrix.

```
In[5]:= H = Mod[AI.B, 26]

Out[5]= {{2, 3, 15}, {5, 8, 12}, {1, 13, 4}}
```

Finally, multiply this resulting matrix on the left with the plaintext matrix, and take the result modulo 26, to get the possible ciphertext matrix.

```
In[6]:= M = Mod[PT.H, 26]

Out[6]= {{9, 12, 18}, {9, 15, 15}, {19, 24, 8},
         {11, 23, 15}, {4, 12, 6}, {5, 11, 6}}
```

The above calculation was to check the validity of the encryption matrix. If we convert the result to ciphertext letters we get, JMSJPPTYILXPEMGFLG which is our ciphertext of JMSJPPTYILXPEMGFLG. The ciphertext matrix checks out, so the block size was three and the encryption matrix is,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Δ

2.14.9 Maxima

When using a general computer algebra system you will follow the same procedure as you would with the Cryptography Explorer program except that the operations will be done with the computer algebra system commands instead of input dialog boxes and menu selections.

There is an entire section on vectors and matrices in the appendix on Maxima, we suggest that you read over that section before continuing.

For the Hill cipher the main things you need to be concerned with are, defining a matrix, multiplying two matrices, and inverting a square matrix. We will quickly recap these below.

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")`.

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, $\ldots$, 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.

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 ^ but rather `^^`. 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) $\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$

```
(%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) A+M;
```

(%o5) $\begin{bmatrix}2 & 2 & 4\\ 6 & 6 & 11\\ 10 & 10 & 9\end{bmatrix}$

```
(%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) $\begin{bmatrix} 1 & 0 & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{5} & \frac{5}{6} \\ \frac{3}{7} & \frac{1}{4} & 0 \end{bmatrix}$

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]);
```

$$(\%o2)\quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 6 & 7 & 8 \\ 7 & 8 & 9 & 11 & 12 \end{bmatrix}$$

```
(%i3) rref(A);
```

$$(\%o3)\quad \begin{bmatrix} 1 & 0 & -1 & 0 & -1 \\ 0 & 1 & 2 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

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]);
```

$$(\%o1)\quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
(%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) mod(A+M, 7);
```

$$(\%o3)\quad \begin{bmatrix} 2 & 2 & 4 \\ 6 & 6 & 4 \\ 3 & 3 & 2 \end{bmatrix}$$

```
(%i4) mod(A.M, 7);
```

$$(\%o4)\quad \begin{bmatrix} 1 & 3 & 5 \\ 6 & 0 & 1 \\ 4 & 2 & 0 \end{bmatrix}$$

```
(%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) $\begin{bmatrix}1&0&0&0&7\\0&1&0&0&12\\0&0&1&0&6\\0&0&0&1&1\\0&0&0&0&0\end{bmatrix}$

Examples

Let's look at a couple examples using Maxima.

Example 78: Say Alice wants to send Bob the message "Cryptography is cool!" using the Hill cipher with encryption matrix,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

If we convert "Cryptography is cool!" to matrix form we get,

$$M = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix}$$

Then we would simply multiply ME modulo 26 and convert the result back to ciphertext characters. In Maxima, all we need to do is define the two matrices, E and M and then invoke the command `mod(M.E, 26)` and we are done. Specifically,

```
(%i1) E:matrix([2,3,15],[5,8,12],[1,13,4]);
```

(%o1) $\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$

```
(%i2) M:matrix([2,17,24],[15,19,14],[6,17,0],[15,7,24],[8,18,2],[14,14,11])
```

(%o2) $\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix}$

```
(%i3) CT:mod(M.E,26);
```

(%o3) $\begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$

Reading the rows we have the numeric list, 9 12 18 9 15 15 19 24 8 11 23 15 4 12 6 5 11 6 which converts to the ciphertext JMSJPPTYILXPEMGFLG. Δ

Example 79: Now say that Bob receives the message JMSJPPTYILXPEMGFLG from Alice and knows that the encryption matrix was,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Bob knows that he needs the inverse of the matrix E, modulo 26. There are several ways to do this with Maxima, we will use the easiest one here and then examine some alternatives in the next example. First put in the encryption matrix and the ciphertext matrix. Then load in the `mat_mod_inverse` script that we developed for finding the inverse of a square matrix modulo n, and use it to find the inverse of E. Finally, multiply this inverse, the decryption matrix, by the ciphertext matrix to get the plaintext matrix.

```
(%i1) E:matrix([2,3,15],[5,8,12],[1,13,4]);
```

(%o1) $\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$

```
(%i2) CT:matrix([9,12,18],[9,15,15],[19,24,8],[11,23,15],[4,12,6],[5,11,6])
```

(%o2) $\begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$

```
(%i3) 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)
      )$

(%i4) EI:mat_mod_inverse(E,26);
```

(%o4) $\begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$

```
(%i5) mod(CT.EI, 26);
```

(%o5) $$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix}$$

Reading the rows we get the numeric list, 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11, which converts to the plaintext message CRYPTOGRAPHYISCOOL. Δ

Example 80: The easiest way to find an inverse matrix in Maxima is with the `invert` command and to find a modular inverse with the script we developed so it can be done is a single `mat_mod_inverse` command. So why would we want to use a different, and probably more difficult method? Depending on what course this section is being integrated into, the instructor may wish that you practice other calculation skills. For example, if this is being integrated into a course that does not assume that you know any linear algebra or matrix theory then most likely the instructor will have you calculate the inverse as we did above. On the other hand, if this is being integrated into a linear algebra class, or a class that assumes a knowledge of linear algebra, then the instructor may wish use this as an example of how the topics and techniques in linear algebra work when we are doing our calculations modulo 26 instead of over the real numbers.

Recall from linear algebra that if a square matrix, M, is invertible then it can be reduced to the identity matrix. Furthermore, the row operations that are done in reducing M to the identity will also convert the identity matrix to the inverse of M. So if we construct the $n \times 2n$ matrix that has M in the left half and the $n \times n$ identity in the right half, notationally, $[M|I]$, and reduce this matrix to reduced row echelon form we will get the matrix $[I|M^{-1}]$. So our decryption matrix will be the right side $n \times n$ matrix after the reduction.

To apply this process in Maxima, we will do the following. The exercise is to find the inverse (modulo 26) of the following matrix using a reduction approach.

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

1. First load in the `mod_rref` script to be able to do modular matrix reduction.

2. Load in the matrix.

3. Use the `addcol` and the `ident` commands to join the matrix with the 3×3 identity matrix.

4. Apply the `mod_rref` command to the joined matrix.

5. Finally, use the `submatrix` command to extract the last three columns.

```
(%i1) 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)$

(%i2) A:matrix([2,3,15],[5,8,12],[1,13,4]);
```

(%o2) $$\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

```
(%i3) B:addcol(A,ident(3));
```

(%o3) $\begin{bmatrix} 2 & 3 & 15 & 1 & 0 & 0 \\ 5 & 8 & 12 & 0 & 1 & 0 \\ 1 & 13 & 4 & 0 & 0 & 1 \end{bmatrix}$

```
(%i4) RB:mod_rref(B,26);
```

(%o4) $\begin{bmatrix} 1 & 0 & 0 & 10 & 19 & 16 \\ 0 & 1 & 0 & 4 & 23 & 7 \\ 0 & 0 & 1 & 17 & 5 & 19 \end{bmatrix}$

```
(%i5) AI:submatrix(RB,1,2,3);
```

(%o5) $\begin{bmatrix} 10 & 19 & 16 \\ 4 & 23 & 7 \\ 17 & 5 & 19 \end{bmatrix}$

Δ

Maxima can also help with the breaking of a Hill Cipher. We will redo the example from the section on breaking the Hill cipher.

Example 81: Eve has intercepted the encrypted message JMSJPPTYILXPEMGFLG and from other espionage knows that this message was CRYPTOGRAPHYISCOOL. She also knows that other messages sent that day between Alice and Bob are using the same key and she wishes to decrypt them as well, but she has no other information about these other messages.

Since the message size 18 she knows that that block size must be 2, 3, 6, 9 or 18, and since she has only 18 characters to work with her only hope is that the block size is either 2 or 3. If both of these fail it is back to other espionage.

Let's start with a block size of 2. Since,

CRYPTOGRAPHYISCOOL — 2 17 24 15 19 14 6 17 0 15 7 24 8 18 2 14 14 11

encrypts as

JMSJPPTYILXPEMGFLG — 9 12 18 9 15 15 19 24 8 11 23 15 4 12 6 5 11 6

we know that

$$[2\ 17] \longrightarrow [9\ 12] \qquad [24\ 15] \longrightarrow [18\ 9] \qquad [19\ 14] \longrightarrow [15\ 15] \cdots$$

So we want to construct a 2×2 matrix out of the plaintext blocks that is invertible modulo 26. If we take the first two blocks 2 17 and 24 15 and build a matrix from them

$$M = \begin{bmatrix} 2 & 17 \\ 24 & 15 \end{bmatrix}$$

then $\det(M) = -378$ which is not relatively prime to 26, so M is not invertible. Actually, we should have seen this coming with what we know about determinants, notice that the first column has all even numbers in it and 2 is a factor of 26.

If we move on to the next block, 19 14 and keep the first block our M matrix becomes,

$$M = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}$$

then $\det(M) = -295$ which is 17 when we take it modulo 26. Since 17 has an inverse modulo 26 we are in business. Our equation becomes,

$$\begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix} E = \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix}$$

So

$$E = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}^{-1} \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} = \begin{bmatrix} 10 & 25 \\ 5 & 20 \end{bmatrix} \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} = \begin{bmatrix} 23 & 1 \\ 7 & 22 \end{bmatrix}$$

To see if this works we test it on our plaintext message CRYPTOGRAPHYISCOOL.

$$C = ME = \begin{bmatrix} 2 & 17 \\ 24 & 15 \\ 19 & 14 \\ 6 & 17 \\ 0 & 15 \\ 7 & 24 \\ 8 & 18 \\ 2 & 14 \\ 14 & 11 \end{bmatrix} \begin{bmatrix} 23 & 1 \\ 7 & 22 \end{bmatrix} = \begin{bmatrix} 9 & 12 \\ 7 & 16 \\ 15 & 15 \\ 23 & 16 \\ 1 & 18 \\ 17 & 15 \\ 24 & 14 \\ 14 & 24 \\ 9 & 22 \end{bmatrix}$$

As we can see, rows 1 and 3 encrypt correctly, they should since those were the ones we used, but the rest of the encryption is incorrect. Conclusion, the block size is not 2. To do these calculations in Maxima, do the following. Input the `mat_mod_inverse` script and the following three matrices,

$$A = \begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix} \qquad B = \begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix} \qquad PT = \begin{bmatrix} 2 & 17 \\ 24 & 15 \\ 19 & 14 \\ 6 & 17 \\ 0 & 15 \\ 7 & 24 \\ 8 & 18 \\ 2 & 14 \\ 14 & 11 \end{bmatrix}$$

Find the inverse of A modulo 26, multiply it on the right by B, this should be the encryption matrix E, then apply it to the plaintext matrix to verify if it produced the ciphertext or not.

```
(%i1) A:matrix([2,17],[19,14]);
```

(%o1) $\begin{bmatrix} 2 & 17 \\ 19 & 14 \end{bmatrix}$

```
(%i2) B:matrix([9,12],[15,15]);
```

(%o2) $\begin{bmatrix} 9 & 12 \\ 15 & 15 \end{bmatrix}$

```
(%i3) PT:matrix([2,17],[24,15],[19,14],[6,17],[0,15],[7,24],[8,18],[2,14],[
```

(%o3) $\begin{bmatrix} 2 & 17 \\ 24 & 15 \\ 19 & 14 \\ 6 & 17 \\ 0 & 15 \\ 7 & 24 \\ 8 & 18 \\ 2 & 14 \\ 14 & 11 \end{bmatrix}$

```
(%i4) 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)
      )$
```

```
(%i5) AI:mat_mod_inverse(A,26);
```

(%o5) $\begin{bmatrix} 10 & 25 \\ 5 & 20 \end{bmatrix}$

```
(%i6) E:mod(AI.B, 26);
```

(%o6) $\begin{bmatrix} 23 & 1 \\ 7 & 22 \end{bmatrix}$

```
(%i7) CT:mod(PT.E,26);
```

(%o7) $$\begin{bmatrix} 9 & 12 \\ 7 & 16 \\ 15 & 15 \\ 23 & 16 \\ 1 & 18 \\ 17 & 15 \\ 24 & 14 \\ 14 & 24 \\ 9 & 22 \end{bmatrix}$$

If we convert the result to ciphertext letters we get, JMHQPPXQBSRPYOOYJW which is not our ciphertext of JMSJPPTYILXPEMGFLG. Your conclusion would be the same as above, the block size is not 2.

So we move on to block size 3. If it is then there is a 3×3 matrix E with

$$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix} E = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$$

As before, we want to select three rows from our message matrix that will produce an invertible 3×3 matrix. Looking at the last column, all entries except for 11 are even, so the last row must be used. Since the last row first column is even we must have at least one odd number in the first column of the rows we use. So we could not select the remaining two rows from rows 1, 3, and 5. Furthermore, row 5 is all even so selecting it would be pointless. So in our selection we must have row 6 and at least one of rows 2 and 4. We will try rows 1, 2, and 6. This gives,

$$M = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}$$

Since $\det(M) = -791 \equiv 15 \pmod{26}$, we know that M is invertible. So

$$ME = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix} E = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix}$$

and then,

$$E = \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}^{-1} \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} = \begin{bmatrix} 13 & 3 & 8 \\ 9 & 12 & 10 \\ 24 & 14 & 15 \end{bmatrix} \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Now, we know this is correct but Eve would still need to check her possible solution, doing so she would get,

$$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix} \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$$

which checks with the ciphertext and she has found the encryption matrix, E. To do the above calculations using Maxima do the following. Input the following three matrices along with the `mat_mod_inverse` script.

$$\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix} \qquad \begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix} \qquad \begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix}$$

Then follow the same procedure as we did above. Find the inverse of A modulo 26, multiply it on the roght by B, this should be the encryption matrix E, then apply it to the plaintext matrix to verify if it produced the ciphertext or not.

```
(%i1) A:matrix([2,17,24],[15,19,14],[14,14,11]);
```

(%o1) $\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 14 & 14 & 11 \end{bmatrix}$

```
(%i2) B:matrix([9,12,18],[9,15,15],[5,11,6]);
```

(%o2) $\begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 5 & 11 & 6 \end{bmatrix}$

```
(%i3) PT:matrix([2,17,24],[15,19,14],[6,17,0],[15,7,24],[8,18,2],[14,14,11]
```

(%o3) $\begin{bmatrix} 2 & 17 & 24 \\ 15 & 19 & 14 \\ 6 & 17 & 0 \\ 15 & 7 & 24 \\ 8 & 18 & 2 \\ 14 & 14 & 11 \end{bmatrix}$

```
(%i4) 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)
      )$

(%i5) AI:mat_mod_inverse(A,26);
```

(%o5) $\begin{bmatrix} 13 & 3 & 8 \\ 9 & 12 & 10 \\ 24 & 14 & 15 \end{bmatrix}$

```
(%i6) E:mod(AI.B, 26);
```

(%o6) $\begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$

```
(%i7) CT:mod(PT.E,26);
```

(%o7) $\begin{bmatrix} 9 & 12 & 18 \\ 9 & 15 & 15 \\ 19 & 24 & 8 \\ 11 & 23 & 15 \\ 4 & 12 & 6 \\ 5 & 11 & 6 \end{bmatrix}$

If we convert the result to ciphertext letters we get, JMSJPPTYILXPEMGFLG which is our ciphertext of JMSJPPTYILXPEMGFLG. The ciphertext matrix checks out, so the block size was three and the encryption matrix is,

$$E = \begin{bmatrix} 2 & 3 & 15 \\ 5 & 8 & 12 \\ 1 & 13 & 4 \end{bmatrix}$$

Δ

2.14.10 Exercises

In some of these exercises you may want, or be required, to do some matrix reduction. Remember that in this context all operations are done modulo 26. So, as we did above, if you divide a row by a number make sure that the number is invertible modulo 26 and multiply the row by its inverse. If you are using Cryptography Explorer to help with the calculations you should use the the Row Operations interface at the bottom of the modular matrix calculator window.

All of the exercises here assume that we are using the classical method of applying the Hill cipher, so the plaintext blocks are converted to row vectors $\mathbf{v}$ and right multiplied by the encryption matrix M to get $\mathbf{v}M = \mathbf{w}$, and $\mathbf{w}$ will be the row vector representing the ciphertext block. Also, if you are using the Hill Cipher tool from the Cryptography Explorer program, make sure that you select the Classical Mode option.

1. Consider the following matrix with entries from $\mathbb{A}$,
$$E = \begin{bmatrix} 2 & 7 \\ 13 & 9 \end{bmatrix}$$
   (a) What is the determinant of E?
   (b) Is E invertible modulo 26? Why or why not?
   (c) If E is invertible modulo 26, find its inverse modulo 26.
   (d) Use E to encrypt the message HELP.
   (e) What message or messages encrypt to XXXX?

2. Consider the following matrix with entries from $\mathbb{A}$,
$$E = \begin{bmatrix} 2 & 7 \\ 15 & 8 \end{bmatrix}$$
   (a) What is the determinant of E?
   (b) Is E invertible modulo 26? Why or why not?
   (c) If E is invertible modulo 26, find its inverse modulo 26.
   (d) Use E to encrypt the message HELP.
   (e) What message or messages encrypt to XXXX?

3. Consider the following matrix with entries from $\mathbb{A}$,
$$E = \begin{bmatrix} 2 & 6 \\ 15 & 8 \end{bmatrix}$$
   (a) What is the determinant of E?
   (b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message HELP.

(e) What message or messages encrypt to XXXX?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does HELP? If so, find all plaintext messages that encrypt to the same ciphertext as does HELP. If not, explain why.

4. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 8 & 7 & 17 \\ 19 & 18 & 4 \\ 20 & 1 & 4 \end{bmatrix}$$

(a) What is the determinant of E?

(b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message ATTACK.

(e) What message or messages encrypt to ATTACK?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does ATTACK? If so, find all plaintext messages that encrypt to the same ciphertext as does ATTACK. If not, explain why.

(g) Are there any plaintext messages that are sent to the zero vector? If so, find all of them and if not explain why?

(h) Compare your results from the last two parts of this exercise, what is the relation between these two answers?

5. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 8 & 7 & 18 \\ 19 & 18 & 4 \\ 20 & 1 & 4 \end{bmatrix}$$

(a) What is the determinant of E?

(b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message CHARGE.

(e) What message or messages encrypt to CHARGE?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does CHARGE? If so, find all plaintext messages that encrypt to the same ciphertext as does CHARGE. If not, explain why.

(g) Are there any plaintext messages that are sent to the zero vector? If so, find all of them and if not explain why?

(h) Compare your results from the last two parts of this exercise, what is the relation between these two answers?

6. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 4 & 8 & 5 & 22 \\ 8 & 16 & 21 & 17 \\ 21 & 25 & 5 & 17 \\ 17 & 4 & 9 & 10 \end{bmatrix}$$

(a) What is the determinant of E?

(b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message RUN AWAY. Note that since RUN AWAY, when the space is removed, has only seven letters we will need to pad the end with a random letter, in many cases an X is used. we will use this convention and encrypt RUNAWAYX.

(e) What message or messages encrypt to RUNAWAYX?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does RUNAWAYX? If so, find all plaintext messages that encrypt to the same ciphertext as does RUNAWAYX. If not, explain why.

(g) Are there any plaintext messages that are sent to the zero vector? If so, find all of them and if not explain why?

(h) Compare your results from the last two parts of this exercise, what is the relation between these two answers?

7. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 24 & 17 & 20 & 17 \\ 25 & 21 & 7 & 9 \\ 23 & 22 & 2 & 5 \\ 3 & 17 & 18 & 23 \end{bmatrix}$$

(a) What is the determinant of E?

(b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message RUN AWAY. Note that since RUN AWAY, when the space is removed, has only seven letters we will need to pad the end with a random letter, in many cases an X is used. we will use this convention and encrypt RUNAWAYX.

(e) What message or messages encrypt to RUNAWAYX?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does RUNAWAYX? If so, find all plaintext messages that encrypt to the same ciphertext as does RUNAWAYX. If not, explain why.

(g) Are there any plaintext messages that are sent to the zero vector? If so, find all of them and if not explain why?

(h) Compare your results from the last two parts of this exercise, what is the relation between these two answers?

8. Consider the following matrix with entries from $\mathbb{A}$,

$$E = \begin{bmatrix} 20 & 15 & 7 & 17 \\ 23 & 5 & 9 & 19 \\ 3 & 5 & 25 & 22 \\ 9 & 25 & 15 & 1 \end{bmatrix}$$

(a) What is the determinant of E?

(b) Is E invertible modulo 26? Why or why not?

(c) If E is invertible modulo 26, find its inverse modulo 26.

(d) Use E to encrypt the message RUN AWAY. Note that since RUN AWAY, when the space is removed, has only seven letters we will need to pad the end with a random letter, in many cases an X is used. we will use this convention and encrypt RUNAWAYX.

(e) What message or messages encrypt to RUNAWAYX?

(f) Are there any other plaintext messages that encrypt to the same ciphertext as does RUNAWAYX? If so, find all plaintext messages that encrypt to the same ciphertext as does RUNAWAYX. If not, explain why.

(g) Are there any plaintext messages that are sent to the zero vector? If so, find all of them and if not explain why?

(h) Compare your results from the last two parts of this exercise, what is the relation between these two answers?

9. Which of the matrices in the above examples could be used for a Hill cipher and which are not suitable? In each case explain why?

10. For each of the matrices in the previous exercises that are invertible, find their inverses using,

(a) The reduction technique.

(b) The adjoint technique.

11. For each of the matrices in the previous exercises that are not invertible, reduce them as far as possible. Do any reduce to the identity matrix?

12. For each of the matrix transformations in the previous exercises, find the kernel of the transformation.
13. For each of the matrices in the previous exercises, find a maximum set of rows that are linearly independent.
14. For each of the matrices in the previous exercises, which ones have rows that form a basis to $\mathbb{A}^n$? Why?
15. In the description of the Hill cipher we require that the encryption matrix E is invertible modulo 26. What properties of an invertible matrix are important in the encryption and decryption process and why?
16. We have intercepted a segment of the plaintext of a message

    SENDMORETROOPS

    and we have the corresponding ciphertext

    SYTKAWQVHOECOB

    We know that the segment was at the beginning of the message and that there was more ciphertext that followed but we do not know the corresponding plaintext. We, of course, wish to decrypt the entire message, so we need the decryption matrix. Find the encryption and decryption matrices for this cipher, if possible. If there is not enough information to do this, state why.
17. We have intercepted a segment of the plaintext of a message

    WEAREINNEEDOFASSISTANCER

    and we have the corresponding ciphertext

    KWULUEKTYVJEZCQAGSPWRWYJ

    We know that the segment was at the beginning of the message and that there was more ciphertext that followed but we do not know the corresponding plaintext. We, of course, wish to decrypt the entire message, so we need the decryption matrix. Find the encryption and decryption matrices for this cipher, if possible. If there is not enough information to do this, state why.
18. We have intercepted the plaintext of a message

    INEEDABIGCUPOFCOFFEE

    and we have the corresponding ciphertext

LAYZOKQUPQNSXKUNVMONE

Find the encryption and decryption matrices for this cipher, if possible. If there is not enough information to do this, state why.

We suspect that the same matrix was used to send messages for that entire day. Later we intercepted the following ciphertext, find the message, if possible.

KPMJQWWRHZUQBKEUWCSUPVNHUCVHDUANIKNKIBN

19. Say that Alice uses the following matrix for her encryption matrix,

$$E = \begin{bmatrix} 2 & 7 \\ 13 & 9 \end{bmatrix}$$

and encrypts the message,

SOLONGANDTHANKSFORALLTHEFISH

into

KSWVAPNNTKOXAZXPPRNVJOOHKDXH

When she sends the message to Bob there is one error in the transmission and Bob receives the ciphertext,

KSWVAPINTKOXAZXPPRNVJOOHKDXH

When Bob decrypts the message how many errors are in his decryption?

20. Say that Alice uses the following matrix for her encryption matrix,

$$E = \begin{bmatrix} 23 & 7 & 22 & 3 \\ 25 & 16 & 3 & 4 \\ 6 & 22 & 6 & 7 \\ 7 & 20 & 8 & 6 \end{bmatrix}$$

and encrypts the message,

SOLONGANDTHANKSFORALLTHEFISH

into

SOSLUFSLOLJEQTWBSSFUSRJAEXMH

When she sends the message to Bob there are two errors in the transmission and Bob receives the ciphertext, SOTLUFSLOLJEQTWBSSJUSRJAEXMH

When Bob decrypts the message how many errors are in his decryption?

21. From the two exercises above, what is the relationship between transmission errors and errors in the decryption of the message? Does it make any difference where the errors are, for example, if the errors are close together or far apart?

    In World War I the German Military used the ADFGX code. In this code, the first step was to rewrite each letter in the message as a pair of letters from the set $\{A, D, F, G, X\}$, for example, a might be coded as DF, b as AA, and so on. From there the encryption method used only the characters ADFGX. The reason for this choice was that in Morse code, the method of transmission, the letters A, D, F, G, and X were easy to distinguish. So if there was a transmission error, usually human error in this case, the receiver could easily tell what the letter should have been. This was one of the first uses of error correcting codes in cryptography. Today, the use of error correction codes is commonplace in cryptography.

22. Say we use the following matrix to encrypt the message,

    ATTACK

$$E = \begin{bmatrix} 8 & 19 & 9 & 22 & 7 & 10 \\ 6 & 18 & 25 & 25 & 10 & 0 \\ 0 & 25 & 24 & 0 & 24 & 0 \\ 0 & 22 & 0 & 6 & 16 & 3 \\ 19 & 6 & 10 & 0 & 0 & 7 \\ 19 & 23 & 10 & 10 & 21 & 23 \end{bmatrix}$$

    (a) What does the message encrypt to?
    (b) Find the decryption matrix.
    (c) We will now put in a single transmission error. Change the third letter of the ciphertext to a C, and decrypt the new (flawed) message.
    (d) How many errors are in the flawed decryption? Is it possible to recover the original message with this decryption?
    (e) If we do not know the number or position of the errors in the transmission of a single word, as is usually the case, what are all of the possible plantext messages with one word that could be received as the ciphertext ABCDEF?

23. In some of the exercises above we examined what happens if there is an error in the transmission of the ciphertext. This exercise looks at what happens if we have an error in the encryption matrix. Say Alice and Bob share the key, the encryption matrix, but Bob's handwriting is not all that great and he cannot read the entry in the first row

and third column but he knows that the rest of the encryption matrix is correct. Bob currently has,

$$E = \begin{bmatrix} 20 & 22 & x \\ 1 & 5 & 5 \\ 7 & 4 & 20 \end{bmatrix}$$

(a) Just knowing this, what are the possibilities for the value of x? How did you come to this conclusion?

(b) Later that day Bob receives the following ciphertext from Alice

PSTBOOPFRMJXNTDNCD

From this, can Bob determine the value of x? If so what is it and what does the message say?

24. In our discussion above we stated that we can think of the encryption matrix, E, as a linear transformation $E : \mathbb{A}^n \to \mathbb{A}^n$. Formally, we define $E(\mathbf{v}) = \mathbf{v}E$ for all $\mathbf{v} \in \mathbb{A}^n$. With this definition, show that E is a linear transformation.

25. In the definition of the Hill cipher, the matrix E must be invertible modulo 26. As we saw, that meant that the determinant of E had to be invertible in $\mathbb{A}$, which is where the entries of E came from. As we noted above as well, this was in line with the Invertible Matrix Theorem, one part of which states that an $n \times n$ matrix with real coefficients is invertible if and only if the determinant of that matrix is not 0. That is, if the determinant is invertible in the real number system. Since the only real number that is not invertible is 0 we need only exclude it and can simplify the statement. In this exercise we are going to examine some of the other parts of the Invertible Matrix Theorem in relation to $\mathbb{A}$ and its implications to the Hill cipher. Also recall that we can think of the encryption matrix, E, as a linear transformation $E : \mathbb{A}^n \to \mathbb{A}^n$. We will use these two points of view interchangeably.

(a) Say that E is invertible. What does this imply about the properties of one-to-one and onto of the transformation?

(b) Say that E is not invertible. What does this imply about the properties of one-to-one and onto of the transformation?

(c) If the transformation was not one-to-one, what would this imply about encryption and decryption with the Hill cipher? Be specific as to the difficulties that would arise.

(d) If the transformation was not onto, what would this imply about encryption and decryption with the Hill cipher? Be specific as to the difficulties that would arise.

(e) If E is invertible, what is the kernel of E? How does this relate to your answers above?

(f) If E is not invertible, what is the kernel of E? How does this relate to your answers above?

(g) If E is not invertible, is it possible that two English messages could be sent to the same ciphertext? How does this relate to your answers above? Find two examples of non-invertible encryption matrices E and for each, two different English plaintexts that are encrypted to the same ciphertext.

(h) If E is not invertible, and you know one decryption of a given ciphertext as well as the kernel of E, how can you determine, without knowing E, what all of the possible decryptions are? Justify your answer.

(i) If E is invertible, what can be said about the columns of E in relation to the set $\mathbb{A}^n$? For each observation, discuss the relevance of it to the Hill cipher.

26. In the work we did above, we used a Known Plaintext attack on the cipher to find its key, that is, the encryption matrix. One question would be if other types of attacks could gather enough information for us to determine the encryption, and hence decryption matrices?

    (a) Chosen Plaintext: Say that Eve temporally gains access to the encryption machine. Think of it as a black box that she can input messages and get the encryption back.

        i. Suppose further that she knows the block size. What plaintexts should she choose to find the encryption matrix? How would she use the output of the machine to construct the encryption matrix? If she knows that she has only a few tries before she is detected and locked out of the system, what is the fewest number of plaintexts she needs to determine the encryption matrix?

        ii. Suppose she does not know the block size and the machine automatically pads the input with X's as well as breaks the message into blocks. What plaintexts should she choose to find the encryption matrix? How would she use the output of the machine to construct the encryption matrix? If she knows that she has only a few tries before she is detected and locked out of the system, what is the fewest number of plaintexts she needs to determine the encryption matrix?

    (b) Chosen Ciphertext: Say that Eve temporally gains access to the decryption machine. Think of it also as a black box that she can input ciphertext and get the plaintext back.

        i. Suppose further that she knows the block size. What ciphertexts should she choose to find the encryption matrix? How would she use the output of the machine to construct the encryption matrix? If she knows that she has only a few tries before she is detected and locked out of the system, what is the fewest number of ciphertexts she needs to determine the encryption matrix?

        ii. Suppose she does not know the block size and the machine automatically pads the input with X's as well as breaks the message into blocks. What ciphertexts should she choose to find the encryption matrix? How would she use the output of the machine to construct the encryption matrix? If she knows that she has only a few tries before she is detected and locked out of the system, what is the fewest number of ciphertexts she needs to determine the encryption matrix?

(c) Ciphertext Only: Here Eve has only the ciphertext of several messages.

i. How could she determine the block size? Or at least a small set of possible block sizes.

ii. Once she has determined the set of possible block sizes, how would she proceed to find the encryption matrix?

iii. In many cases a simple substitution cipher can be broken using only a single ciphertext, if it is sufficiently long. Could Eve break the Hill cipher with only a single, fairly long, ciphertext? If so, what restrictions would need to be assumed?

27. Look back at the definition of a vector space. If we let our set of scalars be $\mathbb{A}$ instead of $\mathbb{R}$ and if we let $V = \mathbb{A}^n$ instead of $\mathbb{R}^n$, which of the properties of a vector space still hold for V and which do not? For each, if the property holds prove it and if the property does not hold then find a counter example to show that the property fails.

28. The Hill cipher is defined to work with square encryption matrices E. Since the decryption matrix $D = E^{-1}$, it is clear why this restriction is imposed. One question would be if we could relax this condition and let E be an $n \times m$ matrix where $n \neq m$? Our immediate answer to this would be no since if E was not square, it would not have an inverse D for decryption.

On the other hand, if we think about E and D we see that there is a little overkill on this restriction. Specifically, when Alice encrypts her message M, she divides the message into blocks of n letters that is then converted into a row of n numbers. Each block, or row, can, and is, looked at as a vector $\mathbf{v} \in \mathbb{A}^n$. Then when she encrypts her message, she applies E to each of the vectors, getting an encrypted vector $\mathbf{w} = \mathbf{v}E$. Since D is the inverse of E we know that $\mathbf{w}D = \mathbf{v}$ and hence we can decrypt the message. So what we require is that $\mathbf{v}ED = (\mathbf{v}E)D = \mathbf{w}D = \mathbf{v}$ for all vectors $\mathbf{v} \in \mathbb{A}^n$. But if D is the inverse of E we also have that $\mathbf{v}DE = \mathbf{v}$ for all vectors $\mathbf{v} \in \mathbb{A}^n$, which is not required by the Hill cipher algorithm. That is, we only need to get back to $\mathbf{v}$ with the product ED and not DE.

Before we get to far into this lets step back and ask the same question but with matrices over the real number system. That is, can we produce two non-square matrices A and B such that $AB = I$ but $BA \neq I$? From the products it is clear that if A is $n \times m$ then B must be $m \times n$ and the products AB will be $n \times n$ and BA will be $m \times m$.

(a) Consider the matrix

$$E = \begin{bmatrix} -6 & -4 & 7 \\ 1 & 5 & 5 \end{bmatrix}$$

over the real numbers, does there exist a matrix D with $DE = I$? To prove or disprove this construct a generic 3×2 matrix D,

$$D = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}$$

and compute

$$DE = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix} \begin{bmatrix} -6 & -4 & 7 \\ 1 & 5 & 5 \end{bmatrix}$$

From this, can we solve for each of the components of D? If so, then we have $DE = I$ as desired. Also, if it is possible one would ask if D is unique?

i. Do the above product and solve the resulting system, over the real numbers, does a solution exist?
ii. If a solution exists write the matrix D.
iii. Is D unique?
iv. If not, write a general expression for all matrices D with $DE = I$.

(b) Consider the matrix

$$E = \begin{bmatrix} -6 & 1 \\ 1 & 5 \\ 7 & 4 \end{bmatrix}$$

over the real numbers, does there exist a matrix D with $DE = I$? To prove or disprove this construct a generic 2×3 matrix D,

$$D = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

and compute

$$DE = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} -6 & 1 \\ 1 & 5 \\ 7 & 4 \end{bmatrix}$$

From this, can we solve for each of the components of D? If so, then we have $DE = I$ as desired. Also, if it is possible one would ask if D is unique?

i. Do the above product and solve the resulting system, over the real numbers, does a solution exist?
ii. If a solution exists write the matrix D.
iii. Is D unique?
iv. If not, write a general expression for all matrices D with $DE = I$.

(c) From the above calculations what would you conjecture as the answer to the question,

> Given an $n \times m$ matrix A does there exist an $m \times n$ matrix B with $AB = I$?

What restrictions must be placed on n and m? With these restrictions, is it always possible? Why or why not?

(d) Think of the matrices A and B as transformations between $\mathbb{R}^n$ and $\mathbb{R}^m$. Revise your statement about the restrictions on the sizes of n and m to the language of transformations. That is, discuss the properties of one-to-one and onto of the transformation induced by A and B and the necessity of these restrictions needed for AB to be both one-to-one and onto, as is the identity matrix.

(e) If we can find two non-square matrices A and B such that the matrix AB produces a one-to-one and onto transformation can we find another matrix C with $AC = I$ or can we find another matrix D with $DB = I$? If so, how would we create either C or D? If not find two matrices A and B with the matrix AB producing a one-to-one and onto transformation but there exists no matrices C and D with either $AC = I$ or $DB = I$.

(f) Discuss this situation with matrices over $\mathbb{A}$ in place of matrices over $\mathbb{R}$. Specifically,

i. Is it possible to relax the Hill requirement of using a square matrix?
ii. If so, what restrictions must be placed on n and m?
iii. If so, with these restrictions, can this always be done? Why or why not?
iv. If so, give examples of non-square matrices E and D over $\mathbb{A}$ with $ED = I$ modulo 26.

29. In the Hill cipher we do all of calculations modulo 26 since, of course, there are 26 letters in the English alphabet. What if we had a different alphabet? Say we had an alphabet with 27 letters, or 29 letters. What if our language symbolism was based on syllables instead of letters, then we would probably have around 80 to 100 characters? What if our language were based on pictures, then we could have thousands of characters. Updating the Hill cipher would be easy in this case, all we do is change the modulus.

(a) Say our alphabet had only 5 letters. What would the determinant of a Hill cipher encryption matrix need to be?

(b) Say our alphabet had only 12 letters. What would the determinant of a Hill cipher encryption matrix need to be?

(c) Say our alphabet had 29 letters. What would the determinant of a Hill cipher encryption matrix need to be?

(d) What would the advantage be to the Hill cipher if we had a prime number of letters in our alphabet?

(e) Say we had an alphabet with 5 letters in it, $\{A, B, C, D, E\}$. So in this case $\mathbb{A} = \{0, 1, 2, 3, 4\}$, with addition and multiplication done modulo 5. Consider the following matrix with entries in $\mathbb{A}$.

$$E = \begin{bmatrix} 4 & 1 & 2 \\ 1 & 0 & 0 \\ 2 & 4 & 4 \end{bmatrix}$$

i. What is the determinant of E?
ii. Could E be used as an encryption matrix for a Hill cipher?
iii. If so, find the decryption matrix D using both the reduction technique and the adjoint technique.
iv. If so, encrypt the message ABBEDA.
v. If so, decrypt the message AAABBB.

(f) Say we had an alphabet with 5 letters in it, $\{A, B, C, D, E\}$. So in this case $\mathbb{A} = \{0, 1, 2, 3, 4\}$, with addition and multiplication done modulo 5. Consider the following matrix with entries in $\mathbb{A}$.

$$E = \begin{bmatrix} 1 & 3 & 4 \\ 1 & 4 & 2 \\ 1 & 1 & 3 \end{bmatrix}$$

i. What is the determinant of E?
ii. Could E be used as an encryption matrix for a Hill cipher?
iii. If so, find the decryption matrix D using both the reduction technique and the adjoint technique.
iv. If so, encrypt the message ABBEDA.
v. If so, decrypt the message CCDDEE.

30. Pair up with another student in your class. Both of you create your own plaintext message (in English) and an invertible matrix modulo 26. Make the message fairly long, at least 80 alphabetic characters. Keep the matrix you use for the cipher fairly small, say 3×3 to 5×5. Encrypt your message using your matrix. Give the other person the ciphertext of the message and a portion of the plaintext. The portion of plaintext you give must be at least 40 consecutive characters. The plaintext can start anywhere in the message but you need to tell the person where the plaintext starts. Break the code your partner gives you and decrypt their message.

2.15 Cipher Machines

2.15.1 Introduction

In the latter years of the 19th century through the first quarter of the 20th century cryptography was making a transition from a linguistic discipline to a mathematical discipline. Ciphers got more technical and involved much more work to complete the algorithm. This, of course, made the encryption and decryption of messages more difficult and time-consuming, especially for field ciphers where you are encrypting or decrypting a message in the middle of gun fire. As usual, when things get too difficult to do by hand we enlist the power of machines.

Just prior to World War II, cipher machines were invented for both military and industrial purposes. These machines carried out intricate encryptions and decryptions quickly, that could not be done by hand in any reasonable amount of time. The three most famous are the Enigma, Purple, and the M-209. The history of these and other cipher machines would take several books worth of information to do them justice, which better people than I have already written. Hence we will not delve into much of the history and instead site other, more detailed, works.

2.15.2 Enigma

The Enigma machine is by far the most famous of the cipher machines from World War II.

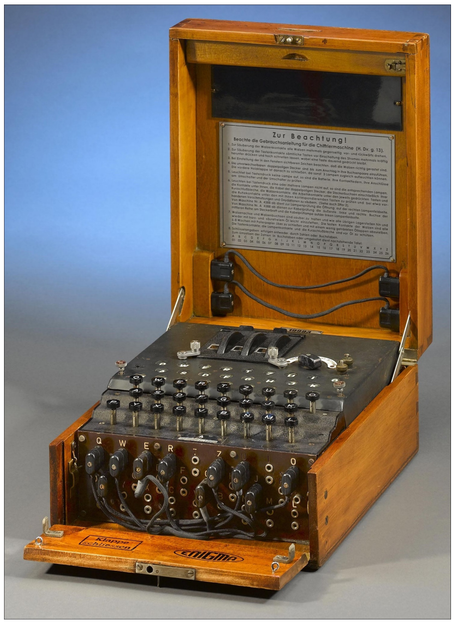


Figure 2.81: The Enigma

The Enigma cipher machine was invented by Arthur Scherbius, a German engineer, in 1917. He then applied for a patent in 1918. The Enigma machine was a rotary cipher machine, where an electronic signal would pass through several rotors to scramble the message and at each letter in the message the rotors would advance. Arthur Scherbius was not the only inventor to conceive of this type of cipher machine, Edward Hebern from the USA, Arvid Damm from Sweden, and Hugo Koch from The Netherlands all invented rotor type cipher machines at about the same time.

Scherbius originally tried to sell his Enigma machine to the German military. But since its invention came after the end of World War I, there was not as much of a military need for it. Hence the military was not interested in spending the money for a new cryptographic method. Scherbius then decided to start up his own company to manufacture the Enigma for commercial sale. He started producing the machines in 1923, but they did not sell all that well. Industrial cryptography was just starting at this time, so many banks and companies were not yet using cryptographic techniques for communications. Also, the first Enigma machine models (Models A and B) were very heavy, weighing in at 110 pounds. These machines also did not incorporate a reflector so there were different encryption and decryption machines.

In 1925, Scherbius developed the Model C, which was much smaller and lighter. The Model C also incorporated a reflector, which turned the machine into both an encryption machine and a decryption machine. It also replaced a typewriter output with a lamp panel. Then in 1927 the Model D was developed which was much more of a commercial success. Furthermore, with the size and weight reduction, the German military started seeing the possible war-time uses of the machine.

The German Navy adopted the Enigma in 1926, followed by the Army in 1928. They both added the plug board and additional rotors, and they continued to update the machine throughout World War II. The Army Enigma was reduced to 26 pounds, making it possible to take the machine into the field.

How it Worked

The Enigma schematic is pictured below.

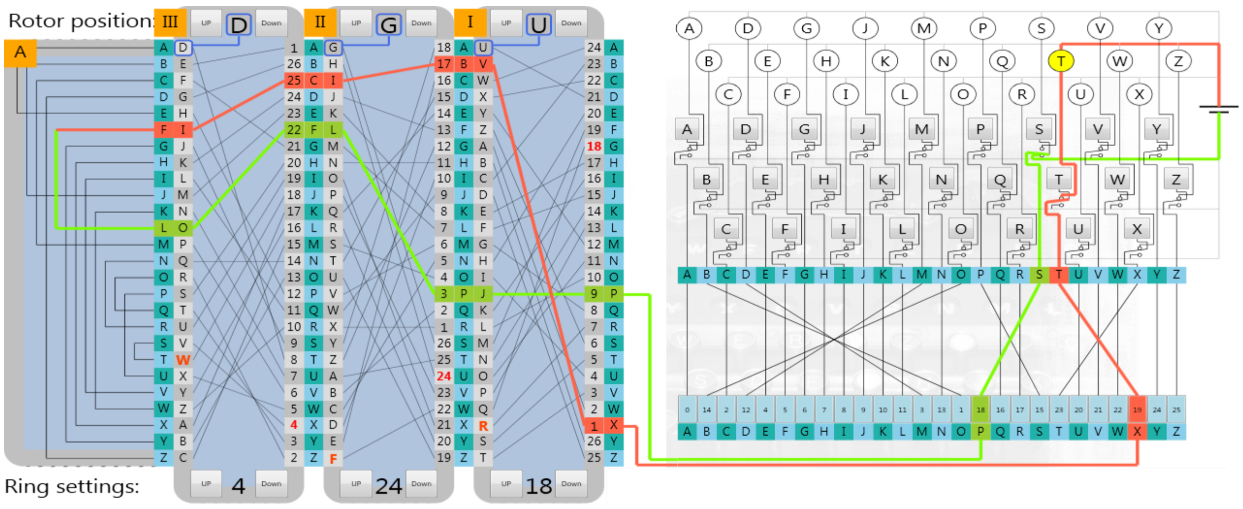


Figure 2.82: Enigma Layout: Image taken from CrypTool 2.0 Enigma Presentation View[41]

Its principle is fairly simple, there is a battery in the upper right of the diagram that produces the electricity for the device. The green line shows the flow of electricity from the battery to the reflector and the orange line shows the flow of electricity on the return trip from the reflector back to the the lamp panel.

1. When the user types a key on the keyboard the first rotor moves up one, the rotor labeled I at the top of the diagram. The key that was typed in the diagram example was S.

2. Electricity then flows from the battery through the typed key and into the plug board, displayed in the lower right. The plug board in the example interchanges S and P so the letter P is coming out of the plug board.

3. The signal then hits the first rotor (labeled I), so the letter comes in at position P, from the position of the rotor it hits pin 9, the internal wiring of the rotor sends it to J.

4. The signal then goes into the second rotor (labeled II). By the setting of rotor I, J is at position P and by the setting of rotor II hits pin 3 and is then sent to L.

5. The signal then goes into the third rotor (labeled III). By the setting of rotor II, L is at position F and it comes into pin 22 of rotor III. The rotor III wiring sends it to O.

6. The signal then hits the reflector. By the setting of rotor III, O is at position L, the reflector changes L to F. The signal now goes back through the rotors.

7. On the return trip, F hits pin I of rotor III and is changed to pin 25 at position C. Rotor II takes the input of C, it hits pin I and is transferred to pin 17 at position B. Rotor I takes the input of B, it hits pin V and is transferred to pin 1 at position X.

8. From X the signal hits the plug board where X and T are interchanged, so the signal comes out at T which is transferred directly to the lamp panel and the T light lights up.

9. The entire process is then repeated when the next letter is pressed.

Each rotor had a notch, in fact rotors VI, VII, and VIII had two notches. When the right hand rotor hit the notch it would advance the rotor to its left, in an odometer-style fashion. The machine also had the feature of double-stepping. This happened when rotor II hits a notch so that rotor III is advanced the action of rotor III advancing would also advance rotor II.

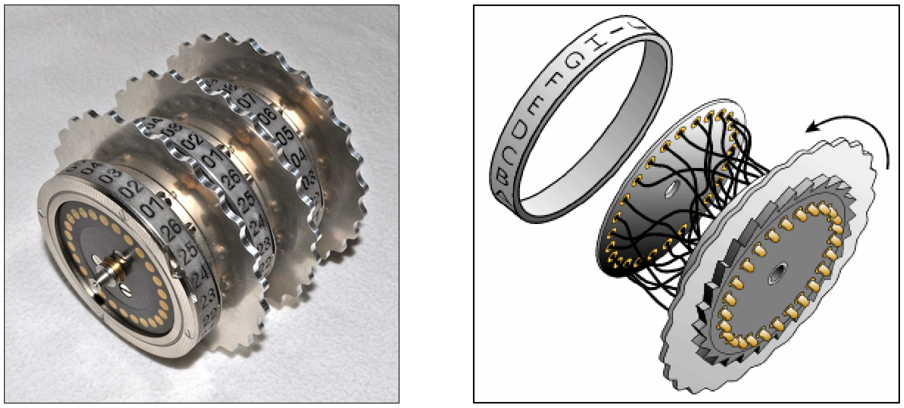


Figure 2.83: The Enigma Rotor Assembly

As you can see from the description of how the machine worked, the scrambling of the message was quite intricate. A quick calculation shows that there were $26 \cdot 25 \cdot 26 = 16,900$ different rotor positions during operation (the 25 is due to the double stepping). So as long as the message was less than 16,900 characters in length, there would be no duplication of

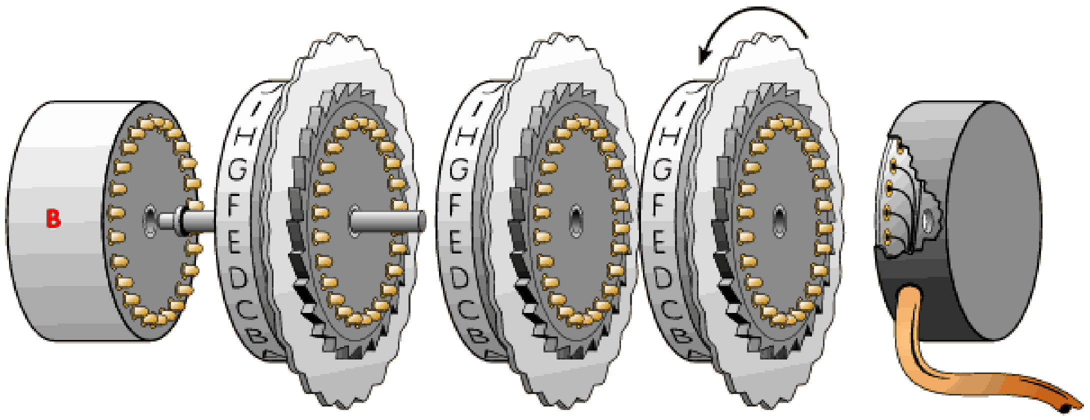


Figure 2.84: Exploded View of Enigma Rotor Assembly

rotor positions. This was a mandate from the German high command that all messages be fewer than 16,900 characters, although this was not always followed.

As is usually the case in cryptography, the breaking of the cipher was more a result of human error on the part of the operator or on the procedure than it is on the machine, or the algorithm. In the case of the Enigma, there was one flaw in the machine in that it would not encrypt any character to itself, so A was never encrypted as A. The rest was the fault was the operators and the procedure of encryption.

Before any message was encrypted or decrypted, the operator needed to set up the machine. Looking at the construction of the machine it is clear that the operator needed to plug in the wires in the plug board, select the correct rotors, put the rotors in the correct orientation and set each rotor to the correct initial starting position. These settings changed each day and they were printed each month into codebooks and distributed to each Enigma operator. If you think about it, this distribution system alone must have been very cumbersome. New codebooks would be continually delivered to operators in the field and on the front lines during the fighting.

The initial settings of the rotors were in the codebooks, of course, and these settings were called day codes. So for example, the codebook might say that the day's code is GEC, then the operator would set the rotors to an initial setting of GEC (that is, the first rotor to C, second to E and third to G). The Germans knew that even with a machine of this complexity using the same code each day for all transmissions was a bad practice, so what they did was use a message code. The message code was three letters, just like the day code, but it was chosen (at random) by the Enigma machine operator who was encoding the message and was supposed to be different for each message. Now the encoder of the message needed to communicate these settings to the recipient of the message or they would be unable to decode the message. So the sender of the message would set the machine to the day code and then encrypt the message code twice. For example, say the day code was GEC and the message code was chosen to be TYU. The encoder would set the machine to GEC and then encode TYUTYU. Let's say the output from the machine was AFMOPR. The sender of the message would reset the machine to a setting of TYU and encrypt the message and place the result after the six letters AFMOPR, and then send the message.

As we all know, any repetition embedded in a cryptographic system gives a handle for the breaking of the code. This simple repetition of three characters was no different. We will not get into the details of the cryptanalysis of the Enigma, but basically if one takes a large set of the first six letters in the transmission they can deduce groups of cyclic pattern of letters in the alphabet. This was, in a sense, a fingerprint of the rotor settings. Using huge catalogs, painstakingly constructed, of rotor settings and their "fingerprints" the cryptanalysist, with the help of a machine known as a Bombe, could deduce the day code in a matter of a few hours.

The Bombe got its name from the ticking it made while the gears and leavers clicked through possible day codes.

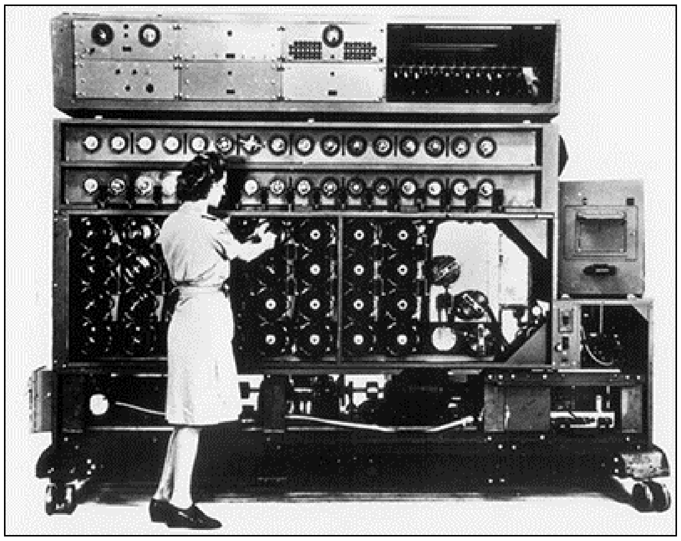


Figure 2.85: The Bombe

In addition to this purely ciphertext only attack, many other methods were employed, for example, capturing codebooks. Even when codebooks were captured the commanders did not share the information with the cryptanalysists, since they wanted the cryptanalysists to become very proficient in breaking the codes, for the times when there were no captured codebooks. Other methods included cribs, for example, one German outpost transmitted a weather report each morning and the first word in the transmission was always WETTER, the German word for weather. Still other methods involved more espionage than mathematics. When a message is sent it was done through Mores Code, and each operator had their own distinct rhythm when typing out the dots and dashes, similar to the way you know who is walking down the hallway by the footstep pattern. This was referred to as the operator's hand. Once the operator was identified, the cryptanalysists knew the message codes thet the operator tended to use. Many of the operators, being male, tended to use the initials of their girlfriends back home.

The Enigma was not the only cipher machine used by the Germans in World War II. In fact, the Enigma was used for "Army level and below" ciphers, essentially only for field operations. The Enigma machine itself was not a small piece of equipment but it could be carried by a soldier into combat. For high-level communications the Germans used two other cipher machines, these were the T type 52-B/C/D/E, built by Siemens & Halske, and the SZ-40/SZ-42, built by Standard Elektrik Lorenz. The British referred to these machines by the generic name of FISH and specifically as STURGEON and TUNNY, respectively.[38] These machines were much larger than the Enigma and were designed to connect to radioprinter communication devices, hence neither could be used in the field.

If you are interested in further reading on the Enigma machine and its history please consider starting with Simon Singh's book *The Code Book: The Science of Secrecy form Ancient Egypt to Quantum Cryptography.* This gives a nice concise history of the machine and a primarily non-mathematical account of the breaking of the cipher that still gives the mathematician a satisfactory amount of detail. For a bit of a longer read, we suggest David Kahn's 1967 book *The Codebreakers: The Story of Secret Writing* or his updated 1996 version *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet.*

2.15.3 The Red and Purple

In 1922, the American Black Chamber broke the codes used for Japanese diplomatic communications. Unfortunately, this was not kept secret and Japan was under pressure to develop another encryption method.

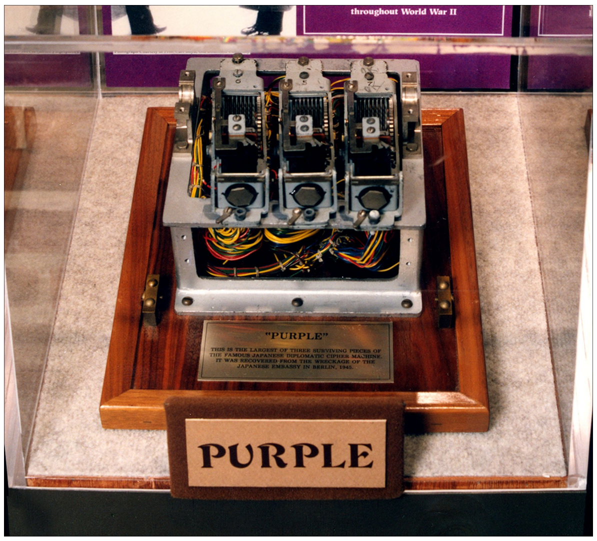


Figure 2.86: Part of the Purple Machine

In 1931, the Japanese completed the development of the 91-shiki injiki (Type 91 print machine). Note that the year 1931 was year 2591 in the Japanese Imperial calendar, hence the prefix of 91.

The Japanese Foreign Ministry in 1935 adopted the device and renamed it Angōki A-kata (Type A Cipher Machine). When the U.S. Army's Signal Intelligence Service (SIS) first encountered intercept from the Type A machine in 1935, the personnel nicknamed the system RED. Color nicknames were used extensively in the U.S. military at that time for plans and programs, so it seemed only natural to the SIS staff to adopt the first color of the spectrum for the first machine cryptosystem they worked on.[65] The SIS produced its first translation from a RED machine decrypt in February 1937.

Although the Japanese did not realize that the RED had been broken they were completing their next generation cipher machines, the 97-shiki injiki or Angōki B-kata (Type B Cipher Machine). It seems that the motivation for the new machine was not the concern with the RED's security but the fact that the RED machine tended to break down if it was not carefully maintained on a daily basis. When the SIS first intercepted Type-B communications in 1939 they nicknamed the new machine PURPLE.

The Japanese made two major errors in the construction and use of the RED and PURPLE machines, aside from thinking that their ciphers were unbreakable. First, the machines split the encryption between vowels and consonants, and encrypted them separately. This 20/6 split made the cryptanalysis much easier. Second the RED and PURPLE machines were used simultaneously for much of the war. The PURPLE was used for communication with major diplomatic posts while the RED was used for less important communications. Many messages were sent using both devices and since we had already broken the RED machine we could use these cribs to help break the PURPLE.

The solution of PURPLE was a team effort, under the overall direction of William Friedman, with Frank Rowlett leading the day-to-day efforts. Genevieve Grotjan, Albert Small, and Samuel Snyder, junior cryptanalyts, also made important contributions in solving the system.[65]

2.15.4 M-209

In 1938, Swedish cryptographer Boris Hagelin designed the C-38 cipher machine in response to a request for a smaller portable cipher machine, over the current C-36 machine. In 1940, the U.S. Army adopted it and renamed it the M-209, the U.S. Navy designated it as the CSP-1500. It was first used by the U.S. military during the Africa Campaign in 1942 and its use continued throughout World War II and through the Korean War. It was not the most secure of ciphers which is why it was used for tactical and low level communications. As early as 1943, the Germans had cracked the M-209 but since it still took about 4 hours to decode an M-209 transmission the machine was still useful for tactical communications, where the result of the communication usually happened in far less then four hours.

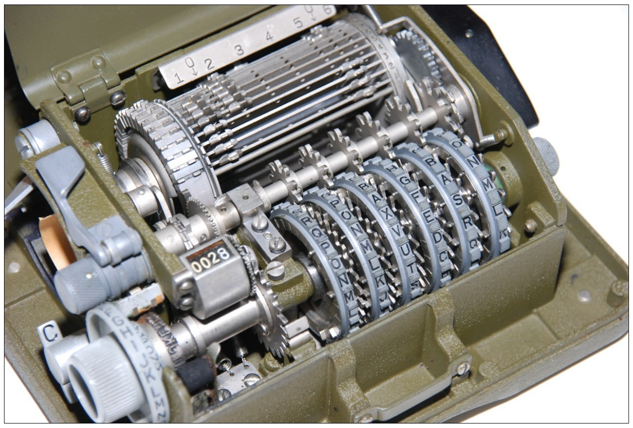


Figure 2.87: The M-209 Machine

The M-209 was much smaller than the Enigma machine, weighing in at only 6 pounds, and was completely mechanical, so no battery was needed for its operation. On the inside of the machine, as pictured above, the main mechanism consisted of 6 key wheels and a cylindrical drum consisting of 27 horizontal bars. There was no keyboard or light panel as on the Enigma. The input letters were set using the first key wheel and the output was printed on a tin piece of paper, like a ticker-tape.

Each key wheel contains a different number of letters. From left to right, the wheels have:

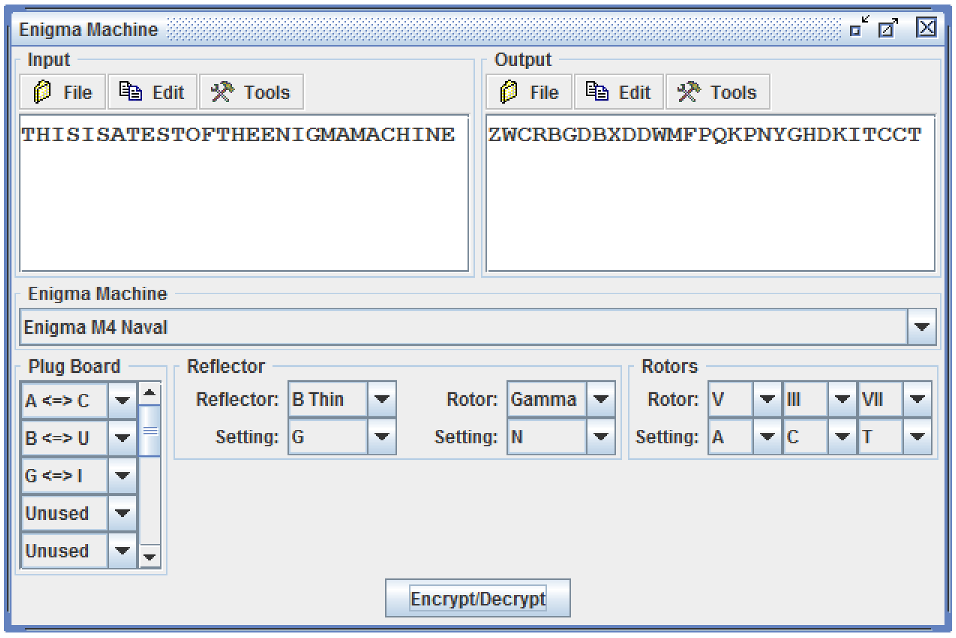
- 26 letters, from A to Z
- 25 letters, from A to Z, minus W
- 23 letters, from A to X, minus W
- 21 letters, from A to U
- 19 letters, from A to S
- 17 letters, from A to Q

The reason the key wheels were constructed in this manner was so that the wheels had a coprime number of settings. This ensured that the machine would not repeat key sequences due to the alignment of the wheels. This allowed the machine to have a period of $26 \cdot 25 \cdot 23 \cdot 21 \cdot 19 \cdot 17 = 101,405,850$ characters before repeating, sufficient for any message being sent. The key wheels could not be removed and reordered as with the Enigma machine, but later (post-war) models developed by Hagelin did have removable wheels. The later models never saw service in the field due to the emergence of secure, digital encryption techniques.

The cylindrical drum consisted of 27 horizontal bars. Each bar had two movable lugs that could be aligned with any of the six key wheels (called an effective position), or may be placed in one of two neutral positions (called an ineffective position). The positioning of these bar lugs alters the way the wheels are advanced and hence provides a second type of key setting. This setting of lugs was rarely altered and the key was primarily the initial settings of the key wheels. The initial key wheel settings were changed once a day, as was common with most cipher machine usage.

2.15.5 Cryptography Explorer

The Cryptography Explorer program has an Enigma Machine simulator that simulates four versions of the Enigma Machine, the Enigma I, the M3 Army, the M3 Naval and the M4 Naval. The simulator allows for all of the machine settings but does not show the path of electronic flow through the rotors or reflector.



Input: THISISATESTOFTHEENIGMAMACHINE

Output: ZWCRBGDBXDDWMFPQKPNYGHDKITCCT

Enigma Machine: Enigma M4 Naval

Plug Board: A <=> C, B <=> U, G <=> I, Unused, Unused

Reflector: B Thin, Setting: G

Rotor: Gamma, Setting: N

Rotors: Rotor: V, III, VII; Setting: A, C, T

Figure 2.88: 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.

Chapter 3

Modern Cryptography

3.1 Introduction

Modern cryptography refers to encryption methods that are currently in use. The division between classical and modern is, by most people's definition, the invention of the computer, although this is not universally accepted. The classical methods we looked at in the last chapter were strong enough to withstand attacks by humans but are easily broken with high-speed computers, hence they are not in use today.

Another major milestone in cryptography at the beginning of the modern era was the solution to the key distribution problem. In the 1960's and early 1970's computers and digital encryption methods were extensively used in both the military and in business. As a society, we were moving into a global communication paradigm. Being separated by an ocean was becoming less of an obstacle, businesses were becoming increasingly international, as was our military. Sensitive communications over very long distances were becoming more routine and in that creating a very difficult and expensive problem, key distribution. Prior to 1977, all encryption techniques were symmetric.

In symmetric-key cryptography, Alice and Bob would share an encryption and decryption key that only the two of them knew. When Alice wanted to send a message (plaintext) to Bob she would use the key to encrypt the message into ciphertext, she would then send the ciphertext to Bob where he would use the decryption key to decrypt the message back into plaintext and read what Alice had to say. In all transmissions we assume that a third person, Eve, could and does intercept the message. Now Eve does not have the key, she only has the ciphertext. It is her job to break the code either by finding the key and decrypting the message or by finding the meaning of the message without finding the entire key.

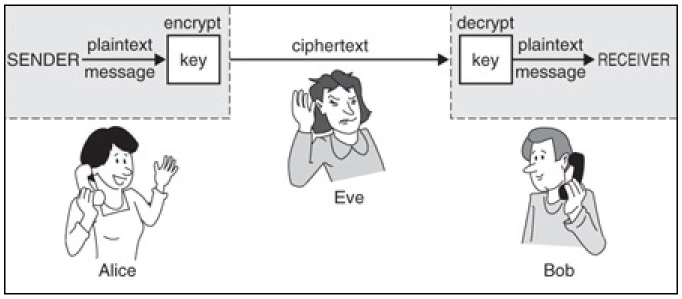


Figure 3.1: Symmetric-Key Cryptography

The Key Distribution Problem was simply, how do Alice and Bob share the key to their symmetric algorithm securely? Since we assume that all transmissions are done over a non-secure channel, there was no way to securely send a key. This meant that the key needed to be transferred between Alice and Bob in person or by a trusted third party. If you watch an old movie where there is a government agent with a briefcase handcuffed to his wrist transferring sensitive documents or encryption keys to another agent or embassy, this stuff really happened. In fact, it happened quite often. Keys needed to be changed frequently to maintain security. Think about the expense of continually sending agents all over the world with cryptographic information, not to mention the danger these agents were in during this transaction.

In 1976, Whitfield Diffie and Martin Hellman described a method in which a key could be transferred securely over a non-secure channel but they did not have a practical method for its implementation. Although the following is not the method that Diffie and Hellman devised, it is a non-mathematical way of envisioning how this type of transfer is possible.

The following method is sometimes called the three-pass protocol. Imagine that Alice wants to send a message to Bob. Alice has a lockable box and both Alice and Bob have padlocks each with its own key (physical key in this case). Note that Alice and Bob do not need to share the padlock keys, Alice has padlock A with key A and Bob has padlock B with key B. Alice takes the message and places it in the box and puts her padlock (padlock A) on it. She then sends the locked box to Bob. Even if Eve intercepts the box she does not have key A hence she cannot open the box and get the message. Once Bob receives the locked box, he puts his padlock on the box and sends the box back to Alice. When Alice receives the box she removes her lock and sends the box back to Bob who removes his padlock, opens the box and reads the message.

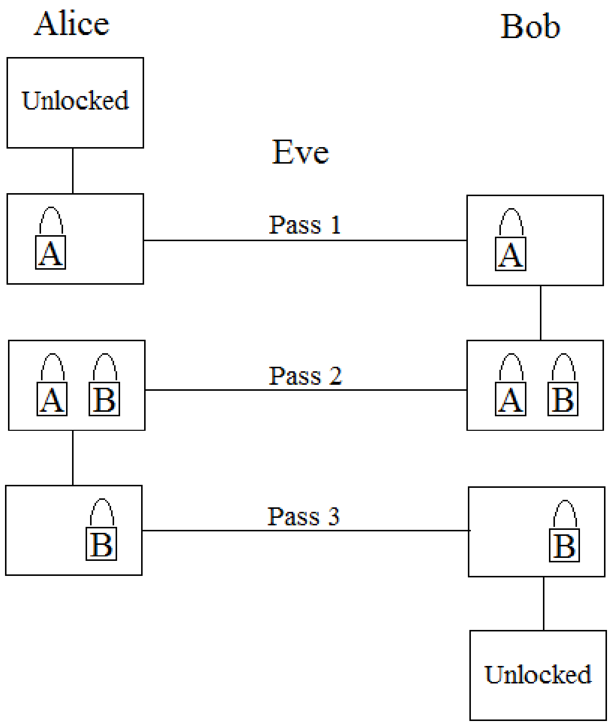


Figure 3.2: Three-Pass Protocol

We clearly need a better method for any practical implementation but this does show the possibility of a scheme. Note that in all transmissions Eve can only get a hold of a locked box that she does not have the key for. In fact, during pass 2 there are two locks on the box. The problem with this method, even if we could mathematically implement it in such a way that the transmissions are done over the airwaves or the wire, is that there are three transmissions to send one message. A far better scheme would be to have a single transmission that is still secure when sent over a non-secure channel.

One way that this could be done, keeping the same padlock scenario, is if Bob had a lot of padlocks all keyed to the same key that he gave out freely to everyone. Then if Alice wanted to send Bob a message she would put the message in the box, as before, pick up one of Bob's freely available padlocks, lock the box with it and then send it to Bob. When Bob receives the box, he would have the key to open the box. Now Eve would have access to Bob's padlock but not the key that would open the box. So one thing that is vitally important here is that given the padlock, Eve would not be able to make a key that fit the lock.

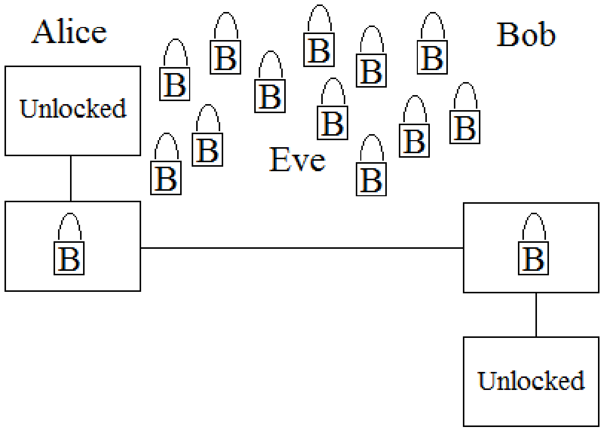


Figure 3.3: One-Pass Protocol

A few things to note here, in both of these methods Alice and Bob do not need to communicate secretly to exchange a key or the message. Also, if we think about the one pass protocol above and convert it to modern cryptographical language, the padlock represents an encryption method and the key to the lock represents a decryption method. So we would want to devise a method where we could let everyone know the encryption key and we would keep the decryption key to ourselves. Furthermore, since encryption and decryption are inverse operations of each other we need to make the encryption process such that even if someone knows the encryption process and the encryption key they would be unable to find the decryption key.

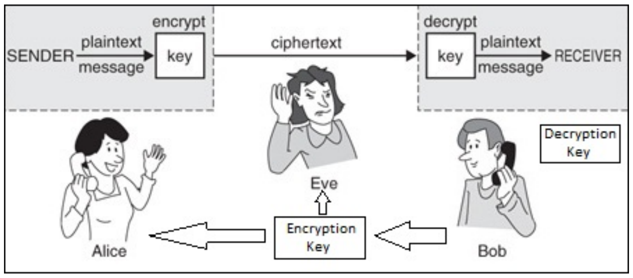


Figure 3.4: Public-Key Cryptography

At this point you are probably thinking that we have just entered the realm of impossibility. If someone, say Eve, knows *everything* about the encryption process and the decryption process is simply the mathematical inverse of the encryption process then the decryption process should be computable. And, of course, it is, there is no denying it. So what we need to do is devise a method where finding the decryption process if difficult, and we mean *really* difficult.

Here is an example of one such method. Consider the following two numbers, 18357401411 and 8540270425901, both of which are prime. Although you may not be able to do this in your head, I certainly cannot, you could multiply these two numbers together in a fairly short amount of time and get the the result 156777172366756588346311. Now, what if we gave you the number 156777172366756588346311 and asked you to factor it, without knowing the above two numbers of course. That task might take you a little more time and effort. Now you may say, I will just give this thing to Mathematica and be done with it. True, Mathematica will factor this number in a few milliseconds, as will Maxima, but what if you give it the number,

```
10423636324252086521969096104334469950556895479433
32434324571064380779117019991316395313756031593714
14337017212270038085902048393951330989513609761561
0352854919547281673201526702322016635291
```

Even with the power of a computer algebra system like Mathematica, this is a difficult and time-consuming task. On the other hand, this number was created from multiplying the two prime numbers,

```
18357401411183570160417601604175385696931543965491
3654913549135496374639817691691554915709
```

and

```
56781654934569716954139645346376489316917693769174
69137693164971693475691374659137659817566432138999
```

which took the computer less than a millisecond to complete. Now you may say that you simply need a faster computer or a supercomputer but you are probably getting the idea that an operation as simple as multiplying two integers together is one of those methods that is easy to do but doing the reverse of that method (that is, factoring) is very difficult. In mathematical terms we call this a one-way function. More specifically, a One-Way Function is a function f, such that, given a value x in the domain of f it is easy to calculate $f(x) = y$ but given a value y in the range of f it is difficult to calculate a value x in the domain with $f(x) = y$. Simply put, calculating the function is easy but finding the inverse of the function is difficult.

When we use the terms easy and difficult we are actually referring to a specific mathematical concept. If a problem is difficult to compute we really mean that it is *Computationally Infeasible*. A process is computationally infeasible if, using the current technology, it would take too long to complete the calculation. So in terms of a one-way function, computing the function is computationally feasible and to compute its inverse is computationally infeasible.

Going back to the above example of multiplication, the number we asked you to factor above was 190 digits in length. In the early 1990's, the record for factorization was about 160 digits and this took several months of computer time on the supercomputers of the time. In 2012, $2^{1061} - 1$, a 320 digit number was factored in a little over a year of computer time. Even now, factoring a 600 digit number, which is the product of two 300(ish) digit prime numbers, would take the world's fastest machines billions of years to complete, which is definitely in the realm of computational infeasibility.

A few notes about one-way functions. With one-way functions, a linear increase in computational power is not sufficient for computing its inverse. Even if we create a computer that is 10 times faster than the one we have, one tenth of a billion years is 100 million years, still infeasible. Computing inverses of one-way functions are done with algorithms that are, at best, exponential in nature. So the fastest known algorithm for factoring an integer into its prime factorization is an exponential time algorithm. What this means is that if computers get to the point of being able to compute some inverses we only need to increase the size of the key a small amount to put the problem back into the computationally infeasible realm. For example, 1567771723667565588346311 can be factored quickly with a personal computer but a 190 digit number cannot. A supercomputer could factor that 190 digit number relatively quickly but if we increase it to 500 or 600 digits then we are back to a billion year calculation. So computational infeasibility deals more with the fastest algorithm we have to complete the inverse of the one-way function. So if we ever find an algorithm that factors an integer in polynomial time then multiplication is no longer a one-way function, and any cryptographic technique that relies on factoring is no longer secure.

In the next section we will look at the RSA algorithm which utilizes the one-way function of multiplication. There are many one-way functions in mathematics, several of which have been used for the creation of public-key systems.

Not all of the modern cryptographic methods are public-key, in fact most are symmetric private-key methods. So why would we bother using private-key methods when we have this "Holy Grail" of cryptography in the public-key methods? The short answer is speed. Most of the public-key methods that employ the use of one-way functions are relatively slow. Using their encryption and decryption algorithms is fast for small amounts of data but when you consider moving the large amounts of data around that tends to be transmitted today, petabytes, these methods are too slow to keep up with the transmission rates. So what is common to do is use a public-key method to send the key of a faster private-key method, one that can keep up with today's transmission rates. The key to a private symmetric method is usually relatively small and although these symmetric methods are not as secure as the public-key methods, they are sufficient, especially since their key was kept secret.

3.2 RSA

3.2.1 History

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. In 1976, when Whitfield Diffie and Martin Hellman came up with the idea of public-key cryptography, they did not have a method for implementing the concept. Rivest, Shamir and Adlemen took the concept of Diffie and Hellman and devised a method that uses the one-way function of integer multiplication to devise a digital implementation.

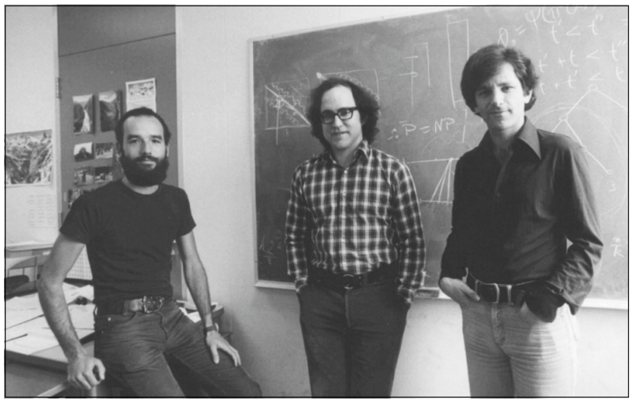


Figure 3.5: Shamir, Rivest and Adleman[1]

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. The Government Communications Headquarters is a British intelligence agency responsible for providing signals intelligence and information assurance to the UK government and armed forces. GCHQ was originally established after the First World War as the Government Code and Cypher School (GC&CS or GCCS). During the Second World War it was located at Bletchley Park, which is where the German Enigma machine was cracked. The GCHQ is the British equivalent of the NSA (National Security Agency) in the United States. Hence anything that was discovered at GCHQ had to remain classified and Clifford Cocks did not get the recognition, or the money, from the discovery of the method. In fact, it was not until 1997 that GCHQ declassified his work.

3.2.2 The RSA Algorithm

First, Bob creates the encryption and decryption keys as follows.

1. Bob chooses secret primes p and q with $p \neq q$ and computes $n = pq$. In practice, the primes p and q are usually several hundred digits in length.

2. Bob chooses e with $\gcd(e, (p-1)(q-1)) = 1$, that is e and $(p-1)(q-1)$ have no common factors.

3. Bob computes d with $de \equiv 1 \pmod{(p-1)(q-1)}$.

[1]Photo taken in 1978, http://www.acm.org/fcrc/PlenaryTalks/rivest.pdf

4. Bob makes n and e public and keeps p, q and d secret. So the public encryption key is (n, e) and the private decryption key is (p, q, d).

Now Alice is ready to send Bob a message and does the following.

1. Alice takes the message and converts it into a number m. If this number m is larger than n then we must take our message and block it into segments so that when we convert it into numbers, each of those numbers is less than n. So for a long message, Alice will have a sequence of numbers $m_1, m_2, m_3, \ldots, m_k$ each of which represents a portion of the message and each of which is less than n. In this case, Alice would do the following for each of the numbers m_i and send the set of encryptions to Bob.
2. She then takes the n and e that Bob published and creates the cyphertext by computing
$$c \equiv m^e \pmod{n}$$
3. Alice then sends c to Bob.

When Bob receives c he decrypts the message by the following computation.

1. Bob takes the cyphertext c.
2. He computes
$$m \equiv c^d \pmod{n}$$
3. Bob converts the number m back into the message.

To get a feel for how the method works we will look at a small example.

Example 82:

1. Bob uses the primes $p = 7$ and $q = 11$. Then he computes $n = 7 \cdot 11 = 77$.
2. Next he chooses a number e so that e and the number $(p-1)(q-1) = 6 \cdot 10 = 60$ have no common factors. He chooses $e = 13$.
3. Now Bob finds the number d such that $de \equiv 1 \pmod{(p-1)(q-1)}$. A little calculation and we get $d = 37$.
4. Bob publishes $n = 77$ and $e = 13$.

Now Alice wants to send him the very short message "H".

1. "H" is the eighth letter in the alphabet so we convert the message to $m = 8$.
2. Now Alice encrypts the message as
$$c \equiv 8^{13} \pmod{77} \equiv 549755813888 \pmod{77} \equiv 50 \pmod{77}$$

3. Alice sends 50 to Bob.

Once Bob receives the encrypted message of 50 he does the calculation.

1. Bob takes the cyphertext $c = 50$.
2. He computes
$$\begin{aligned} m &\equiv 50^{37} \pmod{77} \\ &\equiv 727595761418342590332031250000000000000000000000000000000000000 \pmod{77} \\ &\equiv 8 \pmod{77} \end{aligned}$$
3. Bob converts the number $m = 8$ back into the message “H”.

Δ

At this point there are two questions to answer, why does the method work and why is the method hard to break? Why the method works is a simple result from number theory known as Euler's generalization of Fermat's little theorem.

Theorem 2: *(Fermat's Little Theorem): Given a prime number p and an integer a then*

$$a^p \equiv a \pmod{p}$$

A quick corollary of this is,

Theorem 3: *Given a prime number p and an integer a that is not divisible by p then*

$$a^{p-1} \equiv 1 \pmod{p}$$

Euler then generalized this to the case where the modulus was not prime. Recall that the Euler Phi function, also known as the Euler Totient function, $\varphi(n)$ is the number of positive integers that are less than n and relatively prime to n. Euler's Generalization to Fermat's Little Theorem is as follows,

Theorem 4: *(Euler's Generalization to Fermat's Little Theorem): Given a number n and an integer a that is relatively prime to n then if $\varphi(n)$ represents the Euler Totient function of n we have,*

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

Before we finish the discussion on why the RSA algorithm works lets look at some properties of the Euler Totient function. We will simply state them here, proofs can be found in nearly any number theory textbook.

Theorem 5: *The Euler Totient function has the following properties,*

1. *If p is prime then $\varphi(p) = p - 1$.*
2. *If p is prime then $\varphi(p^k) = p^k - p^{k-1} = p^{k-1}(p-1)$.*
3. *If n and m are relatively prime then $\varphi(nm) = \varphi(n)\varphi(m)$.*
4. *If p and p are prime numbers then $\varphi(pq) = (p-1)(q-1)$.*
5. *For any positive integer n,*
$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$
where the product is over the distinct prime numbers p dividing n.

Given these properties, it is fairly easy to see why the RSA algorithm works. When Bob decrypts the message he calculates

$$c^d \equiv (m^e)^d \equiv m^{ed} \pmod{n}$$

Since $n = pq$ with p and q two distinct primes then $\varphi(n) = \varphi(pq) = (p-1)(q-1)$. Also, by construction of d we have that $de \equiv 1 \pmod{(p-1)(q-1)}$, in other words, $de \equiv 1 \pmod{\varphi(n)}$. Hence $de - 1 = k\varphi(n)$ and so $de = 1 + k\varphi(n)$ for some positive integer k. So

$$c^d \equiv (m^e)^d \equiv m^{ed} \equiv m^{1+k\varphi(n)} \equiv m^1 \cdot m^{k\varphi(n)} \equiv m \cdot (m^{\varphi(n)})^k \equiv m \cdot (1)^k \equiv m \pmod{n}$$

One might want to take issue with this calculation. Note that at the end we used Euler's Generalization to Fermat's Little Theorem to write $m^{\varphi(n)} \equiv 1 \pmod{n}$, but we can only do this if we know that m and n are relatively prime. Since $n = pq$ for (probably large) primes p and q, this would mean that for the RSA algorithm to fail we would have either $m = kp$ or $m = kq$ for some integer k, which is highly unlikely.

3.2.3 Examples

Example 83: Let's say Alice wants to send Bob the message "Meet you at the coffee shop at 4PM." Now with classical ciphers we would remove the spaces and convert the letters to uppercase. This was because our encryption and decryption techniques usually only worked with this as the alphabet. Now we want to create a single number, or sequence of numbers, out of our text, so any coding technique where we can change the characters to text and back will work. So in our message above we can incorporate the spaces, numbers, and the punctuation if we would like. There are, of course, many different ways to do this, the one we would like to use here is the following. Each computer keyboard letter is associated with a number, called its ASCII number. ASCII is the abbreviation for American Standard Code for Information Interchange, it was created in 1960 and was used for several decades. It has since been replaced by Unicode, a character set that incorporates not only character sets for European languages but also, Russian, Chinese, Japanese, Korean, ... The first 128 characters of the Unicode table are the ASCII table. So each keyboard character has a number between 0 and 127, for example, ‘a’ is number 97, ‘b’ is 98 and so on, ‘A’ is 65, ‘B’ is 66 and so on, the space is 32 and the ‘.’ is 46 and so on.

So using ASCII numbers there are still a couple ways to convert “Meet you at the coffee shop at 4PM.” to a number. One would be to simply change all of the letters to their ASCII equivalents in decimal form. Specifically,

```
77 101 101 116 32 121 111 117 32 97 116 32 116 104 101
32 99 111 102 102 101 101 32 115 104 111 112 32 97 116
32 52 80 77 46
```

and since we do not want to send each letter individually, why send 35 numbers when we can send one, we would out it altogether into one number,

```
771011011163212111111732971163211610410132991111102
10210110132115104111112329711632528077 46
```

If we do that we need to be a little careful since some of the ASCII values in this string will be two digits and some will be three. Now when we decrypt the message and get this number back we can figure out where the divisions are since no printable character has ASCII value less then 32 nor greater than 127. So 7710110111632. . . would have to divide up as 77 101 101 116 32. . . but this is rather cumbersome to do, it would be better if we had a method that did not require our intervention. One alteration we could make is to give each ascii number three digits by placing a 0 in front of any two-digit ASCII number. This would produce the following,

```
077 101 101 116 032 121 111 117 032 097 116 032 116 104
101 032 099 111 102 102 101 101 032 115 104 111 112 032
097 116 032 052 080 077 046
```

and when we smash this together we would get,

```
077101101116032121111117032097116032116104101032O
99111102102101101032115104111112032097116032052O8
0077046
```

Now when we encrypt and then decrypt this number we would get back the same thing except that the leading 0 would be removed. This would be easy to detect since there would be 104 digits instead of 105, that is, we would need to add a 0 in front to make the length divisible by 3. Once we did that we would simply break the stream into blocks of three and convert the ASCII numbers back to characters.

Another way we could do this process is to convert the characters to ASCII numbers but represent those numbers in binary form,

```
01001101 01100101 01100101 01110100 00100000 01111001 01101111
01110101 00100000 01100001 01110100 00100000 01110100 01101000
01100101 00100000 01100011 01101111 01100110 01100110 01100101
01100101 00100000 01110011 01101000 01101111 01110000 00100000
01100001 01110100 00100000 00110100 01010000 01001101 00101110
```

Then remove the spaces between the bytes and convert this new big binary number to decimal form, which would produce.

```
5873240455257707465258138591208392610622963873859
65250698382489584542803227306904878
```

Note that this produces a slightly smaller number than our other methods above, which is usually a good thing since we would like to pack as much into a single transmission as possible. Now when we encrypt this and then decrypt this we will get the same number, but when we convert it back to binary we will be missing the leading 0 from the first character, that is, our binary form will be 1001101... instead of 01001101.... But as before that will be easy to figure out since the number of bits will not be a multiple of 8. As before, we will simply pad the beginning of the binary stream with 0's until the length is divisible by 8, then break the stream into blocks of 8 and convert back to characters.

With all of these methods we are jumping the gun a little bit. First we need to know what our modulus is since all of our numbers must be smaller than the modulus before we encrypt them. Here we will use a modulus that is larger than our last number and in the next example we will use a modulus that is smaller and we will then block the message.

Bob chooses the primes

$$\begin{aligned} p &= 306401701460264076403564036543165043653146831 \\ q &= 87950465070854720645760260278567309497548397483193 27 \end{aligned}$$

Then he computes $n = pq$,

$$\begin{aligned} n = \;& 26948172141931411504841629415657502220657476399160627713339780922 \\ & 169577062730112864588272577834 1 \end{aligned}$$

and $\varphi(n) = (p-1)(q-1)$,

$$\begin{aligned} \varphi(n) = \;& 26948172141931411504841629415657502220657475519655973940775559861 \\ & 9262102414166524574538612144332 \end{aligned}$$

and chooses e with $\gcd(e, \varphi(n)) = 1$, $e = 2^{16} + 1 = 65537$ will fit the bill. Finally, he uses the extended Euclidean Algorithm and computes d,

$$\begin{aligned} d = \;& 16731740431013489563811446092322190867155550854789221899447614421 \\ & 4968555309746403580611725693349 \end{aligned}$$

Now Alice is ready to send him her message "Meet you at the coffee shop at 4PM." She chooses to use the method of converting the message to a number by converting each character, including the spaces, to their ASCII binary form, putting the bits all together into one number and then converting the binary number to decimal, finally getting the message,

$$\begin{aligned} m &= 5873240455257707465258138591208392610622963873859652506983824895 8\\ &\quad 4542803227306904878 \end{aligned}$$

Now Alice encrypts the message as $c = m^e \pmod{n}$ and gets,

$$\begin{aligned} c &= 196638776756328951414291125578545401053190989387975381545505139845\\ &\quad 909057501712173565944662145528 \end{aligned}$$

and sends it to Bob.

Once Bob receives the encrypted message he calculates $c^d \pmod{n}$ and gets the message m back. Then he converts t to binary, adds a 0 to the beginning, breaks the stream into blocks of 8, converts each of these to decimal, and finally uses the ASCII table to convert these numbers back into characters and reads Alice's message. Δ

As we pointed out above, the first thing that is done is that Bob sets up the public and private keys for the system, and publishes the public key (e, n). Then Alice encrypts and sends her message. If Alice's message is too long, that is, when she converts the message to a number the converted message turns out to be larger than Bob's modulus. In this case Alice must block the message into smaller segments so that Bob can decrypt the entire message. If Alice does not block the message then when she encrypts the message it will be reduced modulo Bob's smaller modulus and hence be destroyed. One way to block the message would be to take the single number message as before and simply chop it up into smaller numbers that would be concatenated together when decoded. For example, we could take the message m above and chop it up as,

```
587324045525770746525 813859120839261062229 638738596525069838248
958454280322730690 4878
```

Now as long as Bob's modulus was at least 21 digits in length this blocking would work fine. Alice would do 5 encryptions, send the five numbers, then Bob would decrypt each of the five to get the above blocks, concatenate the numbers and convert back. The only thing we would have to be careful about is that we did not break the number at a place that would create a leading 0 on any of the blocks. If this happened then the result would be disastrous.

Although this is a very easy way to do the blocking, usually what is done is that we block the original message text before we convert each to a number. For example, say that the modulus that Bob publishes is 55127205455478826773353043117409637, if we convert this to binary we get a 116 bit number, we could also simply calculate

$$\log_2(55127205455478826773353043117409637) \approx 115.4$$

divide that by 8 and we get a little over 14, so this quick calculation tells us that if we block the message into blocks of 14 characters we will be able to encrypt each of the message blocks.

Example 84: Say Bob chooses the primes

$$\begin{aligned} p &= 6136736013465097 \\ q &= 89831476104756458 21 \end{aligned}$$

Then he computes $n = pq$,

$$n = 551272054554788267733530431174096 37$$

Which is really not a good choice since this number can be factored easily and not with a lot of computational power. The totient $\varphi(n) = (p-1)(q-1)$ is

$$\varphi(n) = 551272054554788177840686966282987 20$$

and the same $e = 2^{16} + 1 = 65537$ still works. Finally, Bob computes d as,

$$d = 256924388274126830705798902881541 13$$

Now Alice is ready to send him her message "Meet you at the coffee shop at 4PM." She chooses to use the method of converting the message to a number by converting each character, including the spaces, to their ASCII binary form, putting the bits all together into one number and then converting the binary number to decimal. From the logarithm base two calculation above she knows that she needs to block the message into blocks of at most 14 characters each, doing so, she gets the following three blocks.

```
Meet␣you␣at␣th
e␣coffee␣shop␣
at␣4PM.
```

Converting each of these to their numeric form gives the following. These are all less than the modulus n in the public key.

```
1569778982269862166735672676742248
2051089444859583717437180642029600
27430754406386990
```

Encrypting each of these gives us the following numbers.

```
6777965976230027299274056666395346
21500752441706753723374141487249440
41649095931053440239001398596914392
```

Now Alice would send these three numbers to Bob. Bob would then raise each of these to the power of d modulo n and retrieve the three numbers,

```
1569778982269862166735672676742248
2051089444859583717437180642029600
27430754406386990
```

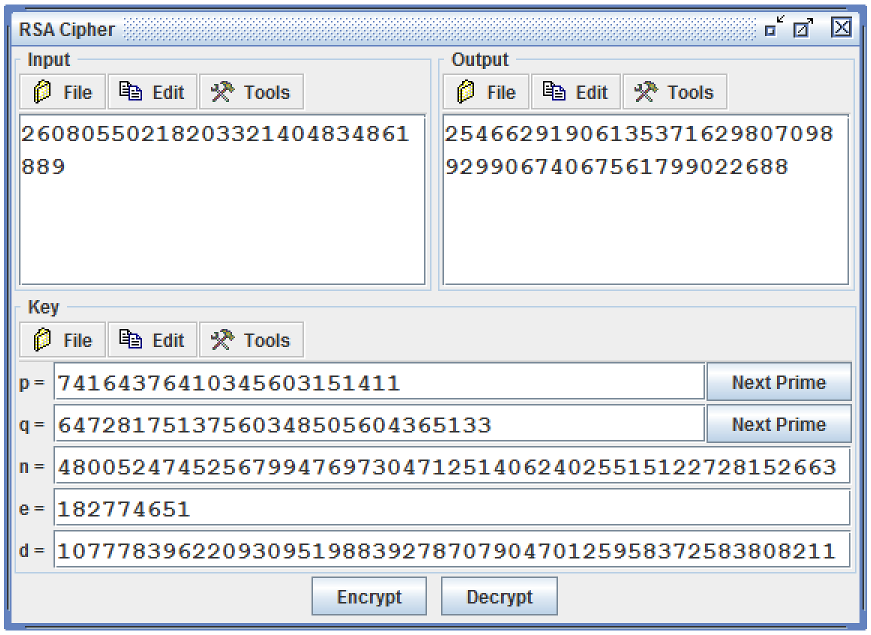
Then after converting each of these back to characters and concatenating the messages, Bob knows that he has a date at 4PM. Δ

Cryptography Explorer

There are two main ways to do RSA calculations using the Cryptography Explorer program. The first is to use the RSA cipher tool and the other is to use the Integer Calculator. A complete discussion of these tools can be found in the appendix on the Cryptography Explorer program, here we will give an abbreviated discussion and use the tools for some examples. Both tools have a standard input box for program input, that has the facilities to do all of the text conversions the program allows, but you may find it helpful to also open up one or more text conversion tool windows.

RSA Cipher Tool

The RSA cipher tool expects an input of either a single number or set of numbers that are separated by spaces. The tool will encrypt (or decrypt) each of the numbers separately using the given parameters.



```
26080550218203321404834861
889
```

```
25466291906135371629807098
92990674067561799022688
```

```
p = 74164376410345603151411
q = 64728175137560348505604365133
n = 4800524745256799476973047125140624025515122728152663
e = 182774651
d = 1077783962209309519883927870790470125958372583808211
```

Figure 3.6: 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.

Note that if you are using this tool to set up the keys for an RSA cipher the p and q lines have Next Prime buttons to their right. So you can simply type in a long number and click the button to produce a prime, actually a probable prime.

We will redo the two examples above using the RSA cipher tool.

Example 85: Recall that Alice wants to send Bob the message "Meet you at the coffee shop at 4PM." Bob will use the same numbers as he did before, so his public key is,

$$\begin{aligned} n &= 2694817214193141150484162941565750222065747639916062771333978092\\ &\quad 21695770627301128645882725778341 \\ e &= 65537 \end{aligned}$$

So Alice will put the value of n above into the n text box of the tool and the value of e into the e text box of the tool. Now for the message, Alice puts *Meet you at the coffee shop at 4PM.* into the input box of the tool. Now this needs to be converted to a number and we will use the same process as before, which will take three separate conversions, but all easy to do. From the Tools menu of the input box, select Convert Text.... At this point a small dialog box will appear with a drop-down selector in it. Using the drop-down box select the Convert Text to ASCII (binary) option and click OK. This will replace Alice's test with bytes of 1's and 0's. Now we will remove the spaces, from the Tools menu of the input box, select Remove White Space. At this point the spaces between the bytes should be removed. Now for the final step of converting this large binary number back to decimal. From the

Tools menu of the input box, select Convert Text.... Now select the Convert Binary to Decimal option from the drop-down box and click OK. Now the message number

$$\begin{aligned} m &= 5873240455257707465258138591208392610622963873859652506983824895 8\\ & 4542803227306904878 \end{aligned}$$

should be in the input box. At this point Alice simply clicks the Encrypt button, which just calculates $m^e \pmod{n}$, and the ciphertext

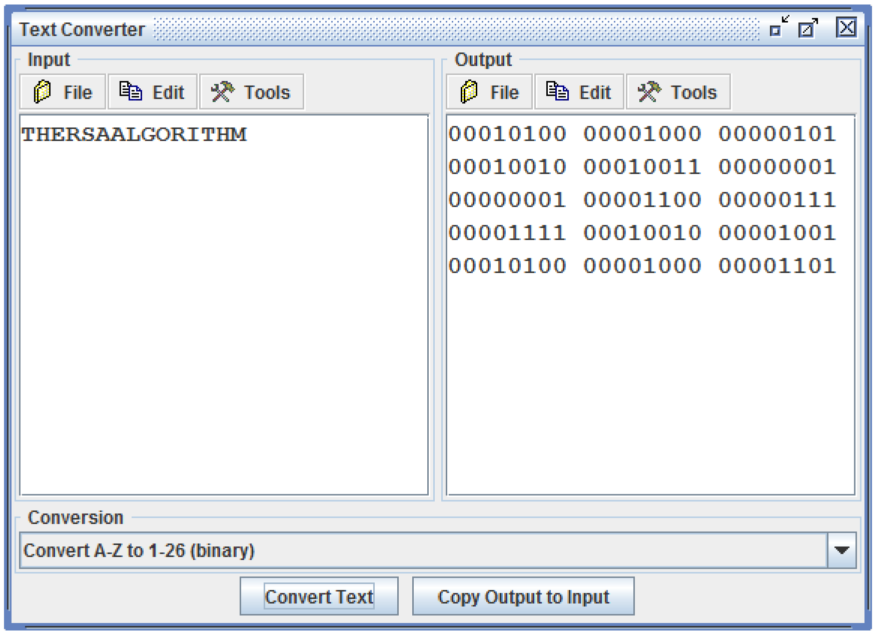
$$\begin{aligned} c &= 1966387767563289514142911255785454010531909893879753815455051398 45\\ & 9090575017121735659446621455 28 \end{aligned}$$

shows up in the output window and of course she then sends it to Bob. Δ

Although we did all of the conversions we needed through the input box, the Text Converter tool is usually quicker and since the output boxes do not incorporate the conversion options, when decrypting an RSA cipher the Text Converter tool is handy for converting the number back to text. As usual, we will only discuss the needed options here for the RSA, a more complete description can be found in the appendix.

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 3.7: 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.

The main options you would use for the encryption process, as we did above, are

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.

Remove White Space: Removes all whitespace from the input box contents. Spaces, returns, tabs, etc. are removed.

Convert Binary to Decimal: Converts a binary number to a decimal.

In the process of taking the decryption number and converting it back to text, you will use the options,

Convert Decimal to Binary: Converts a decimal number to binary.

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.

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.

Example 86: Completing the example above, Bob receives from Alice the ciphertext,

$$\begin{aligned} c &= 196638776756328951414291125578545401053190989387975381545505139845 \\ &\quad 909057501712173565944662145528 \end{aligned}$$

Bob takes his computed private key (d, n), where,

$$\begin{aligned} n &= 269481721419314115048416294156575022206574763991606277133397809 22 \\ &\quad 169577062730112864588272577834 1 \\ d &= 167317404310134895638114460923221908671555508547892218994476144 21 \\ &\quad 496855530974640358061172569334 9 \end{aligned}$$

He opens up an RSA Cipher tool, puts n and d in the respective dialog boxes and the ciphertext number in the input box and hits the Decrypt key, which just calculates $c^d \pmod{n}$. The result is the message number

$$\begin{aligned} m &= 587324045525770746525813859120839261062296387385965250698382489 58 \\ &\quad 4542803227306904878 \end{aligned}$$

Now to convert the number back to a textual message, Bob opens a Text Converter tool and copies and pastes the message number into the input box. At this point he does the following,

1. Select Convert Decimal to Binary in the conversion drop-down and click the Convert Text button.
2. Click the Copy Output to Input button.
3. On the Input Box menu select Tools > Statistics. You will get that there are 279 characters, one less than a multiple of 8. So this means that we are missing a leading 0. Type in a 0 at the beginning of the input string.
4. Select Break Binary Stream into Blocks of 8 in the conversion drop-down and click the Convert Text button.
5. Click the Copy Output to Input button.
6. Select Convert ASCII (binary) to Text in the conversion drop-down and click the Convert Text button. At this point the output box should contain the message *Meet you at the coffee shop at 4PM.*

Δ

Once the message is converted to a number, or set of numbers, the RSA calculations are simply a modular exponent. So the calculations could have been done using the integer calculator. Furthermore, the integer calculator has all of the functions necessary to do the setup for the RSA as well.

Integer Calculator

The Integer Calculator is a simple infinite precision integer arithmetic tool. The appendix discusses all of the options the calculator has but here we will simply discuss the functions needed to encrypt and decrypt an RSA message as well as set up the keys for an RSA method.

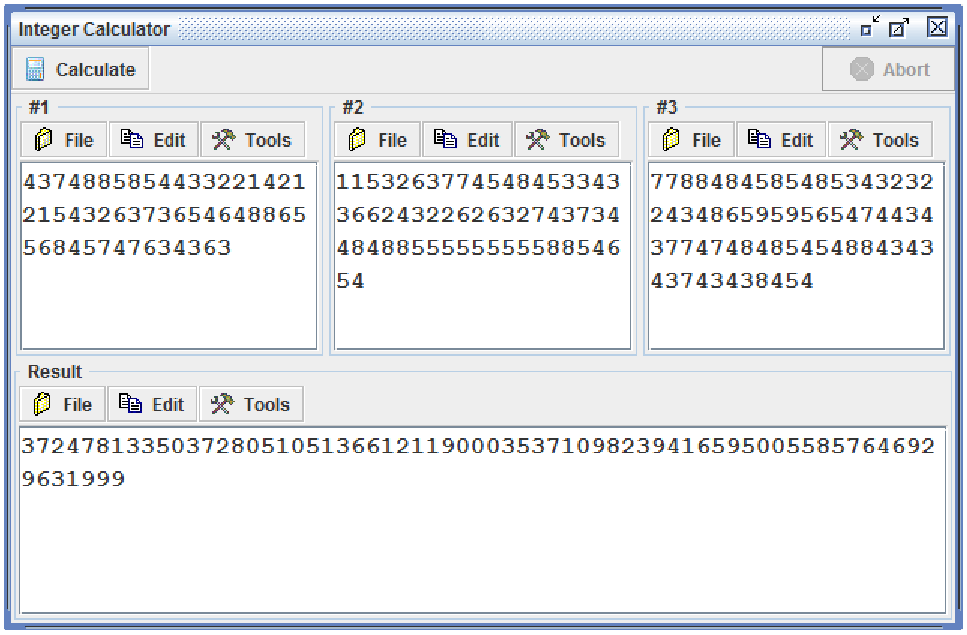
How to Use the Tool

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.

The Calculate menu contains all the functions that the calculator can do, we will look at a small subset of those that are useful here.

Arithmetic —

#1 * #2 Multiplies the numbers from input box #1 and #2.



```
#1
4374885854433221421
2154326373654648865
56845747634363
```

```
#2
1153263774548453343
3662432262632743734
4848855555555588546
54
```

```
#3
7788484585485343232
2434865959565474434
3774748485454884343
43743438454
```

```
Result
37247813350372805105136612119000353710982394165950055857646929
9631999
```

Figure 3.8: Integer Calculator

Modular Arithmetic —

#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.

GCD —

GCD These commands find the GCD of the numbers listed in the menu option.

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.

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.

Factor — Returns the factorization of the number in the selected input box. Factorization of a number can be a lengthy operation.

Options

- Several options in the calculate menu require lengthy derivations, such as totients, primitive roots and factoring. 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.

We will redo the above example using the integer calculator.

Example 87: First, we will go through the setup. Bob had chosen the following two prime numbers.

$$\begin{aligned} p &= 30640170146026407640356403654316504365314683 \\ q &= 8795046507085472064576026027856730949754839748319327 \end{aligned}$$

The integer calculator can check that these numbers are probable primes by using the Calculate > Primes > Is #1 Prime to check if the number in the first input box is prime. If the result is a probable prime, the probability that the number is composite is less than 2^{-100}. There are similar options for checking the second and third input boxes as well.

The integer calculator can also find the next probable prime (and previous probable prime) from the input number. Hence if we put the number

$$30640170146026407640356403654316504365314600$$

into input box number 1 and then select Calculate > Primes > Next Prime from #1, and the result will be the number p above. There are similar options for the second and third input boxes as well.

To calculate n we simply need to multiply p and q, so put p in input #1, q into input #2, and then select Calculate > Arithmetic > #1 * #2 from the menu. The output box will now contain the product n. You may want to open up a Notepad window (Tools > Notepad from the main program menu) to store intermediate calculations and numbers that you will need later in the process. Copy the value of n to the notepad. You can also copy the values of p and q if you would like but we will only need them for the next step and after that they are not needed. The values of op and q should still be in input boxes number one and two. Subtract one from each of then and then multiply these two numbers, this is the Euler totient of n. Copy this number to the notepad for safe keeping. These numbers should be,

$$\begin{aligned} n &= 269481721419314115048416294156575022206574763991606277133397809221695770627301128645882725778341 \\ \varphi(n) &= 269481721419314115048416294156575022206574755196559739407755598619262102414166524574538612144332 \end{aligned}$$

At this point we need to choose e so that $\gcd(e, \varphi(n)) = 1$. We already know that $e = 2^{16} + 1 = 65537$ will work but lets go through all the motions. Put the value for e in input #1 and the totient in input #2. Now select Calculate > GCD > GCD(#1, #2) and a 1 should be the result in the result window. This confirms that e and $\varphi(n)$ are relatively prime and hence $e = 65537$ can be used as an encryption exponent. Copy e to the notepad as well, we will be needing it later.

To complete the setup and finish all of the keys we simply need to calculate d. Put the value of e into input #1, a -1 into input #2 and $\varphi(n)$ into input #3. Now select Calculate > Modular Arithmetic > #1 ^ #2 Mod(#3). The value of d will now be in the result window. Again, copy this to the notepad for later use. The value of d should be,

$$d = 167317404310134895638114460923221908671555508547892218994476144214968555309746403580611725693349$$

Now Alice is ready to send him her message "Meet you at the coffee shop at 4PM." As before, she chooses to use the method of converting the message to a number by converting each character, including the spaces, to their ASCII binary form, putting the bits all together into one number and then converting the binary number to decimal, finally getting the message,

$$m = 587324045525770746525813859120839261062296387385965250698382489584542803227306904878$$

To encrypt the message Alice will put the message into input #1, the value of e into input #2 and the value of n into input #3. She would then select Calculate > Modular Arithmetic > #1 ^ #2 Mod(#3). The value of c, the ciphertext, will now be in the result window. The ciphertext should be,

$$c = 196638776756328951414291125578545401053190989387975381545505139845909057501712173565944662145528$$

and she sends this to Bob.

Once Bob receives the encrypted message he calculates $c^d \pmod{n}$ by putting the ciphertext into input #1, the value of d into input #2 and the value of n into input \$3, then he selects Calculate > Modular Arithmetic > #1 ^ #2 Mod(#3). The value of m, the plaintext, will now be in the result window. Then, just as in a previous example, he converts m to binary, adds a 0 to the beginning, breaks the stream into blocks of 8, converts each of these to decimal, and finally uses the ASCII table to convert these numbers back into characters and reads Alice's message. Δ

Mathematica

If you are using Mathematica then the commands that you will probably need are `PrimeQ`, `NextPrime`, `GCD` and `PowerMod`. You can use Mathematica to convert a string to ASCII and binary to decimal and vice versa, but frankly using the text conversion tools of the Cryptography Explorer program is much easier. Please look at the examples in the Cryptography Explorer section above for instructions on these conversions.

Example 88: First, we will go through the setup. Bob had chosen the following two prime numbers.

$$\begin{aligned} p &= 30640170146026407640356403654316504365314683 \\ q &= 8795046507085472064576026027856730949754839748319327 \end{aligned}$$

In Mathematica, you can check if a number is prime (or probably prime) with the PrimeQ command,

```
In[1]:= PrimeQ[30 640 170 146 026 407 640 356 403 654 316 504 365 314 683 ]

Out[1]= True
```

When the command returns True, as it did above then the number is a probable prime. If the command returns False, then the number is definitely composite. Mathematica also has a command for finding the next probable prime,

```
In[2]:= NextPrime[30 640 170 146 026 407 640 356 403 654 316 504 365 314 600 ]

Out[2]= 30 640 170 146 026 407 640 356 403 654 316 504 365 314 683
```

First we will check that both p and q are prime and then set them to the variables p and q respectively.

```
In[1]:= PrimeQ[30 640 170 146 026 407 640 356 403 654 316 504 365 314 683 ]

Out[1]= True

In[2]:= PrimeQ[
         8 795 046 507 085 472 064 576 026 027 856 730 949 754 839 748 319 \
          327]

Out[2]= True

In[3]:= p = 30 640 170 146 026 407 640 356 403 654 316 504 365 314 683

Out[3]= 30 640 170 146 026 407 640 356 403 654 316 504 365 314 683
```

```
In[4]:= q =
          8 795 046 507 085 472 064 576 026 027 856 730 949 754 839 748 319 ⋱
           327

Out[4]= 8 795 046 507 085 472 064 576 026 027 856 730 949 754 839 748 319 ⋱
          327
```

Next we calculate n and the Euler totient $\varphi(n)$ and set them to the variables n and *phin* respectively.

```
In[5]:= n = p * q

Out[5]= 269 481 721 419 314 115 048 416 294 156 575 022 206 574 763 991 ⋱
          606 277 133 397 809 221 695 770 627 301 128 645 882 725 778 341

In[6]:= phin = (p - 1) (q - 1)

Out[6]= 269 481 721 419 314 115 048 416 294 156 575 022 206 574 755 196 ⋱
          559 739 407 755 598 619 262 102 414 166 524 574 538 612 144 332
```

At this point we need to choose e so that $\gcd(e, \varphi(n)) = 1$. We already know that $e = 2^{16} + 1 = 65537$ will work but lets go through all the motions.

```
In[7]:= e = 2^16 + 1

Out[7]= 65 537
```

Verify that e and $\varphi(n)$ are relatively prime using the GCD function.

```
In[8]:= GCD[e, phin]

Out[8]= 1
```

Calculate the decryption exponent, d, using the PowerMod function.

```
In[9]:= d = PowerMod[e, -1, phin]

Out[9]= 167 317 404 310 134 895 638 114 460 923 221 908 671 555 508 547 ⋱
          892 218 994 476 144 214 968 555 309 746 403 580 611 725 693 349
```

At this point the setup is finished, Bob would publish the public key of (e, n). Now Alice is ready to send him her message "Meet you at the coffee shop at 4PM." As before, she chooses to use the method of converting the message to a number by converting each character, including the spaces, to their ASCII binary form, putting the bits all together into one number and then converting the binary number to decimal, finally getting the message,

$$\begin{aligned} m &= 5873240455257707465258138591208392610622963873859652506983824895 8\\ &\quad 4542803227306904878 \end{aligned}$$

As we discussed in the previous examples, one can use the Cryptography Explorer conversion tools to do this. Once this has been converted, copy it into your Mathematica Notebook and assign it to the variable m.

```
In[10]:= m =
          587 324 045 525 770 746 525 813 859 120 839 261 062 296 387 385 ⋱
            965 250 698 382 489 584 542 803 227 306 904 878


Out[10]= 587 324 045 525 770 746 525 813 859 120 839 261 062 296 387 385 ⋱
           965 250 698 382 489 584 542 803 227 306 904 878
```

Using the PowerMod function, Alice calculates the ciphertext message. She then sends it to Bob.

```
In[11]:= c = PowerMod[m, e, n]

Out[11]= 196 638 776 756 328 951 414 291 125 578 545 401 053 190 989 387 ⋱
           975 381 545 505 139 845 909 057 501 712 173 565 944 662 145 528
```

Once this has been received by Bob he uses the PowerMod function with the decryption exponent d to retrieve the plaintext number.

```
In[12]:= PowerMod[c, d, n]

Out[12]= 587 324 045 525 770 746 525 813 859 120 839 261 062 296 387 385 ⋱
           965 250 698 382 489 584 542 803 227 306 904 878
```

Then, just as in a previous example, he converts m to binary, adds a 0 to the beginning, breaks the stream into blocks of 8, converts each of these to decimal, and finally uses the ASCII table to convert these numbers back into characters and reads Alice's message.

Note that when doing a power with a modulus you definitely want to use the PowerMod function. Theoretically the command `Mod[m^e,n]` will work as well but the difference between this and `PowerMod[m,e,n]`, is that the first command will calculate m^e first and then mod the result by n. The problem here is fairly obvious, the calculation m^e as well as c^d creates extremely large numbers, far too large to store in your computer memory, or in fact any computer memory or even possibly too large to store even if we used every electron available to us in the universe. The `PowerMod[m,e,n]` command combines taking the modulus with the power calculation so the numbers do not become too large. In fact, it uses a very efficient algorithm to accomplish this task. Δ

Maxima

If you are using Maxima then the commands that you will probably need are `primep`, `next_prime`, `gcd`, and `power_mod`. You can use Maxima to convert a string to ASCII and binary to decimal and vice versa, but frankly using the text conversion tools of the Cryptography Explorer program is much easier. Please look at the examples in the Cryptography Explorer section above for instructions on these conversions.

Example 89: First, we will go through the setup. Bob had chosen the following two prime numbers.

$$\begin{aligned} p &= 30640170146026407640356403654316504365314683 \\ q &= 8795046507085472064576026027856730949754839748319327 \end{aligned}$$

In Maxima, you can check if a number is prime (or probably prime) with the `primep` command. Maxima also has a command, `next_prime`, for finding the next probable prime.

```
(%i1) primep(30640170146026407640356403654316504365314683);
```

(%o1) true

```
(%i2) next_prime(30640170146026407640356403654316504365314600);
```

(%o2) 30640170146026407640356403654316504365314683

When the command returns true, as it did above then the number is a probable prime. If the command returns false, then the number is definitely composite.

As with the examples in the previous sections we will go through the entire process of creating the keys and then encrypting and decrypting the message. On the first two inputs we assign the values of p and q, and on the third and fourth lines we set the value of n and $\varphi(n)$. Input 5 sets the value of e and input 6 tests that e and $\phi(n)$ are relatively prime. Input 7 uses the `power_mod` function to invert e modulo $\varphi(n)$ to calculate the decryption exponent.

At this point the setup is finished, Bob would publish the public key of (e, n). Now Alice is ready to send him her message "Meet you at the coffee shop at 4PM." As before, she chooses to use the method of converting the message to a number by converting each character, including the spaces, to their ASCII binary form, putting the bits all together into one number and then converting the binary number to decimal, finally getting the message,

$$\begin{aligned} m = {} & 587324045525770746525813859120839261062296387385965250698382489 58 \\ & 4542803227306904878 \end{aligned}$$

As we discussed in the previous examples, one can use the Cryptography Explorer conversion tools to do this. Once this has been converted, copy it into Maxima and assign it to the variable m.

The final two inputs simply calculate the ciphertext given the message, what Alice would do, and send the result to Bob. Then when Bob receives the ciphertext he would also use the `power_mod` function to decrypt the message.

```
(%i1) p:30640170146026407640356403654316504365314683;
```

(%o1) 30640170146026407640356403654316504365314683

```
(%i2) q:8795046507085472064576026027856730949754839748319327;
```

(%o2) 8795046507085472064576026027856730949754839748319327

```
(%i3) n:p*q;

(%o3) 269481721419314115048416294156575022206574763991606277133397809221695
      770627301128645882725778341

(%i4) phin:(p-1)*(q-1);

(%o4) 269481721419314115048416294156575022206574755196559739407755598619262
      102414166524574538612144332

(%i5) e: 2^16+1;

(%o5) 65537

(%i6) gcd(e,phin);

(%o6) 1

(%i7) d:power_mod(e,-1,phin);

(%o7) 167317404310134895638114460923221908671555508547892218994476144214968
      555309746403580611725693349

(%i8) m:5873240455257707465258138591208392610622963873859652506983824895845
      42803227306904878;

(%o8) 587324045525770746525813859120839261062296387385965250698382489584542
      803227306904878

(%i9) c:power_mod(m,e,n);

(%o9) 196638776756328951414291125578545401053190989387975381545505139845909
      057501712173565944662145528

(%i10)power_mod(c,d,n);

(%o10) 587324045525770746525813859120839261062296387385965250698382489584542
      803227306904878
```

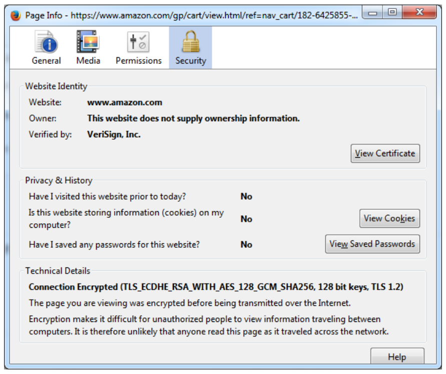
Then, just as in a previous example, he converts m to binary, adds a 0 to the beginning, breaks the stream into blocks of 8, converts each of these to decimal, and finally uses the ASCII table to convert these numbers back into characters and reads Alice's message.

Note that when doing a power with a modulus you definitely want to use the `power_mod` function. Theoretically the command `mod(m^e,n)` will work as well but the difference between this and `power_mod(m,e,n)`, is that the first command will calculate m^e first and then mod the result by n. The problem here is fairly obvious, the calculation m^e as well as c^d creates extremely large numbers, far too large to store in your computer memory, or in fact any computer memory or even possibly too large to store even if we used every electron available to us in the universe. The `power_mod(m,e,n)` command combines taking the modulus with the power calculation so the numbers do not become too large. In fact, it uses a very efficient algorithm to accomplish this task. △

3.2.4 Applications

Public-key systems, like the RSA algorithm, are commonly used in transmission of secure data over e-mail and the Internet. When you log into a site or make a purchase on Amazon (or any other on-line shopping website) your passwords and credit card numbers are encrypted with either the RSA algorithm or an equivalent method that utilizes another one-way function.

Most on-line shopping websites use the RSA algorithm when sending your credit card number from your computer to their computer. When you go to Amazon.com and select the shopping cart you will see some type of lock symbol in the address bar. If you click on this and select to see more information you will see something like the image to the right. Note that we are using FireFox as our browser, other browsers may look differently.


Page Info - https://www.amazon.com/gp/cart/view.html/ref=nav_cart/182-6425855-...

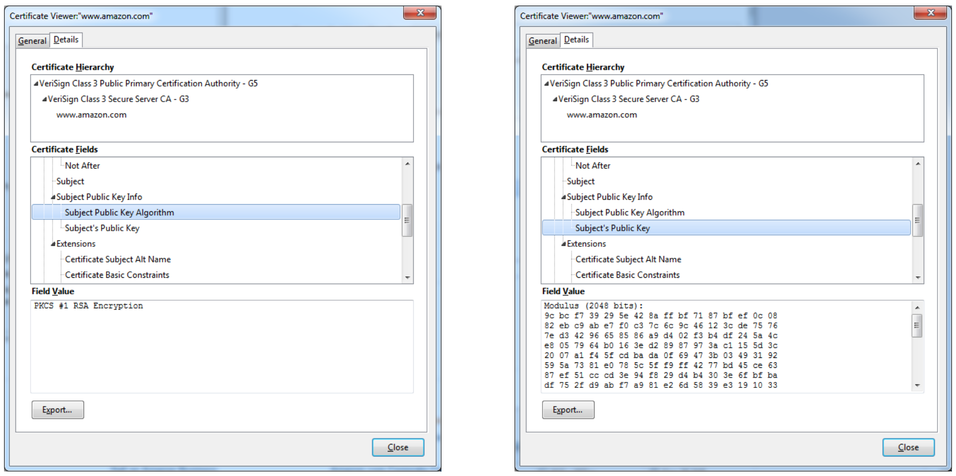
Website Identity
Website: www.amazon.com
Owner: This website does not supply ownership information.
Verified by: VeriSign, Inc.

Privacy & History
Have I visited this website prior to today? No
Is this website storing information (cookies) on my computer? No
Have I saved any passwords for this website? No

Technical Details
Connection Encrypted (TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256, 128 bit keys, TLS 1.2)
The page you are viewing was encrypted before being transmitted over the Internet.
Encryption makes it difficult for unauthorized people to view information traveling between computers. It is therefore unlikely that anyone read this page as it traveled across the network.

Figure 3.9: Encryption Information for Amazon.com

Note that VeriSign Inc. is the provider of the certificate (basically the key) that is being used to make this public-key transmission. So Amazon is not providing the encryption key for the transaction, a third party, VeriSign, is doing that. Then if you click on View Certificate you will see another window with the certificate information. Click on the Details tab and then scroll down to Subject: Public Key Algorithm. When you click on that you should see the following screen, which is telling you that the algorithm is the RSA algorithm. Then if you click on Subject: Public Key you will see the actual key that is being used.


Certificate Viewer:"www.amazon.com"

Certificate Hierarchy
VeriSign Class 3 Public Primary Certification Authority - G5
VeriSign Class 3 Secure Server CA - G3
www.amazon.com

Certificate Fields
Not After
Subject
Subject Public Key Info
Subject Public Key Algorithm
Subject's Public Key
Extensions
Certificate Subject Alt Name
Certificate Basic Constraints

Field Value
```
PKCS #1 RSA Encryption
```

Certificate Viewer:"www.amazon.com"

Certificate Hierarchy
VeriSign Class 3 Public Primary Certification Authority - G5
VeriSign Class 3 Secure Server CA - G3
www.amazon.com

Certificate Fields
Not After
Subject
Subject Public Key Info
Subject Public Key Algorithm
Subject's Public Key
Extensions
Certificate Subject Alt Name
Certificate Basic Constraints

Field Value
```
Modulus (2048 bits):
9c bc f7 39 29 5e 42 8a ff bf 71 87 bf ef 0c 08
82 eb c9 ab e7 f0 c3 7c 6c 9c 46 12 3c de 75 76
7e d3 42 96 65 85 86 a9 d4 02 f3 b4 df 24 5a 4c
e8 05 79 64 b0 16 3e d2 89 87 97 3a c1 15 5d 3c
20 07 a1 f4 5f cd ba da 0f 69 47 3b 03 49 31 92
59 5a 73 81 e0 78 5c 5f f9 ff 42 77 bd 45 ce 63
87 ef 51 cc cd 3e 94 f8 29 d4 b4 30 3e 6f bf ba
df 75 2f d9 ab f7 a9 81 e2 6d 58 39 e3 19 10 33
```

Figure 3.10: Encryption Method for Amazon.com

In this key window the first thing you see is the modulus, which is 2048 bits long (about 616 decimal digits) and then it is given in hexadecimal form,

9c bc f7 39 29 5e 42 8a ff bf 71 87 bf ef 0c 08 82 eb c9 ab e7 f0 c3 7c 6c 9c 46 12 3c de 75 76 7e d3 42 96 65 85 86 a9 d4 02 f3 b4 df 24 5a 4c e8 05 79 64 b0 16 3e d2 89 87 97 3a c1 15 5d 3c 20 07 a1 f4 5f cd ba da 0f 69 47 3b 03 49 31 92 59 5a 73 81 e0 78 5c 5f f9 ff 42 77 bd 45 ce 63 87 ef 51 cc cd 3e 94 f8 29 d4 b4 30 3e 6f bf ba df 75 2f d9 ab f7 a9 81 e2 6d 58 39 e3 19 10 33 c8 6b de 2e 6e 7f f4 00 23 d2 90 0c 8d 8a 99 92 d8 34 c7 a3 a7 e9 20 d4 f0 a0 7a b9 57 e7 f5 75 57 28 d9 66 bd 77 c7 f0 8f 75 5f 4c 71 e2 12 b2 7b f7 fd 64 f0 29 8f f9 d7 ef 9e 67 a4 ae 93 ce 69 10 25 35 90 f6 f0 33 57 6a e4 23 d4 4e 78 a6 b3 ad dd a7 b3 6a 8d 1f 1a 8d 18 66 f1 4b 5f 74 b4 ec 2c 31 e4 06 97 09 8f 16 d1 1c d6 87 03 7f 8e f8 5b 40 a9 8e 2c 58 47 39 27 80 36 fd 57 a7

If we convert this to decimal, we have the following. It is 617 digits in length.

```
197863580422064560287487501835877269018304783346545
5513536875946275415818656786861474994382915111617
71988608174982548580903771303280678274477153852457
95928332123849168642735246864035597337120789510892
18989587242565178831684772100273472959754176086650
67892863995610487822523815942572351910290146133197
71432554217767875411397958877226506230741204119633
36208126262901765931991959999539757973048527862326
96090656880216083512379827304102545386703650541634
95889011124511603592286963012760436410634672028130
54942815795573427757818580350071923606182388284029
51876081620007203225398207981358639034049230931885
15325019823101863
```

Below this is an exponent for the encryption, 65537. So the public key consists of $e = 65537$ abd n being the number above.

Methods like this are commonly used for small amounts of data. When transmitting large amounts of data, say terabytes or petabytes of information, the use of RSA would be too slow for nearly any computer to keep up with the transmission rate. So what is commonly done is the RSA algorithm is used to transfer a key to a faster symmetric encryption algorithm and the faster symmetric-key algorithm is used to encrypt and decrypt the large data set.

3.3 More to Come

Obviously there are more modern encryption methods than just the RSA algorithm. As my time permits, I will be adding to this chapter with other modern methods of encryption and cryptanalysis. In the meantime I suggest a very good book that is written for third year mathematics and computer science undergraduate students.

Introduction to Cryptography with Coding Theory by Wade Trappe and Lawrence C. Washington, Prentice Hall, Upper Saddle River, NJ 07458, 2nd edition, 2006. ISBN 0-13-198199-4.

It goes through just enough number theory that is needed to understand the methods. It covers both symmetric and public key systems, discusses their strengths and weaknesses, the cryptanalysis techniques involved, and gives numerous examples of the uses of the systems. If this textbook is not to your liking or not written at your level there are numerous other textbooks on the market and lots of materials that can be found on the Internet.

Below are appendices on using Mathematica, Maxima, and a piece of software I wrote, Cryptography Explorer, to explore both classical and modern cryptography. Mathematics, like Chemistry or Physics, is an experimental science and the learning is in the doing. Find a book you like, a software package you like, and start exploring.

Appendix A

Introduction to Mathematica

A.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,[77]

`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.

A.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.

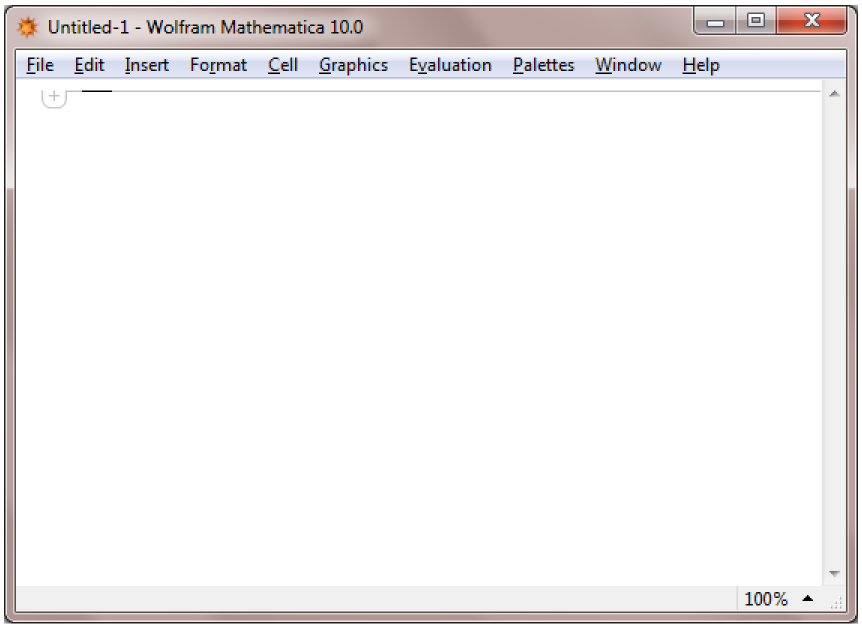
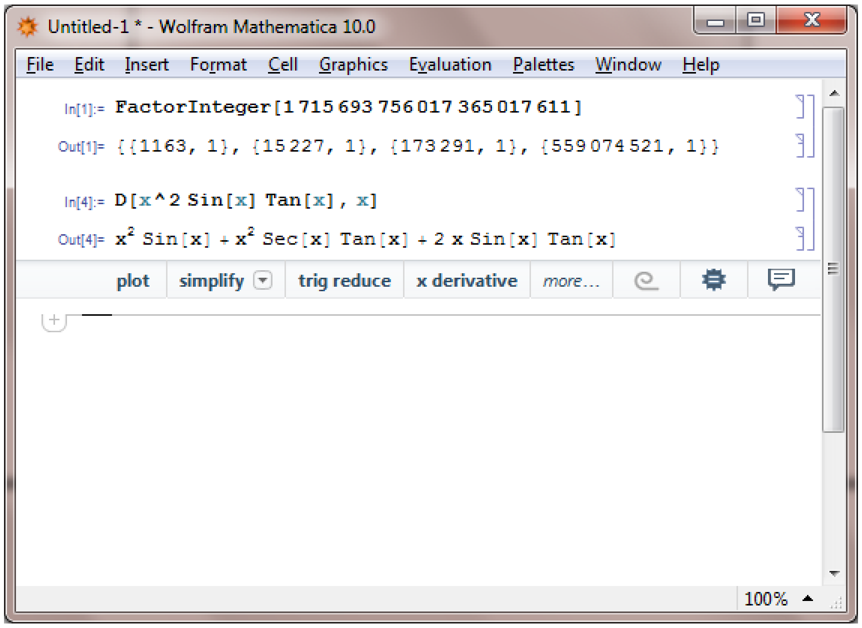

Untitled-1 - Wolfram Mathematica 10.0
File Edit Insert Format Cell Graphics Evaluation Palettes Window Help
100%

Figure A.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[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]
```

Figure A.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]= $3\,x^2\,\mathrm{Sec}\left[x^3\right]^2$

```
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 A.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-

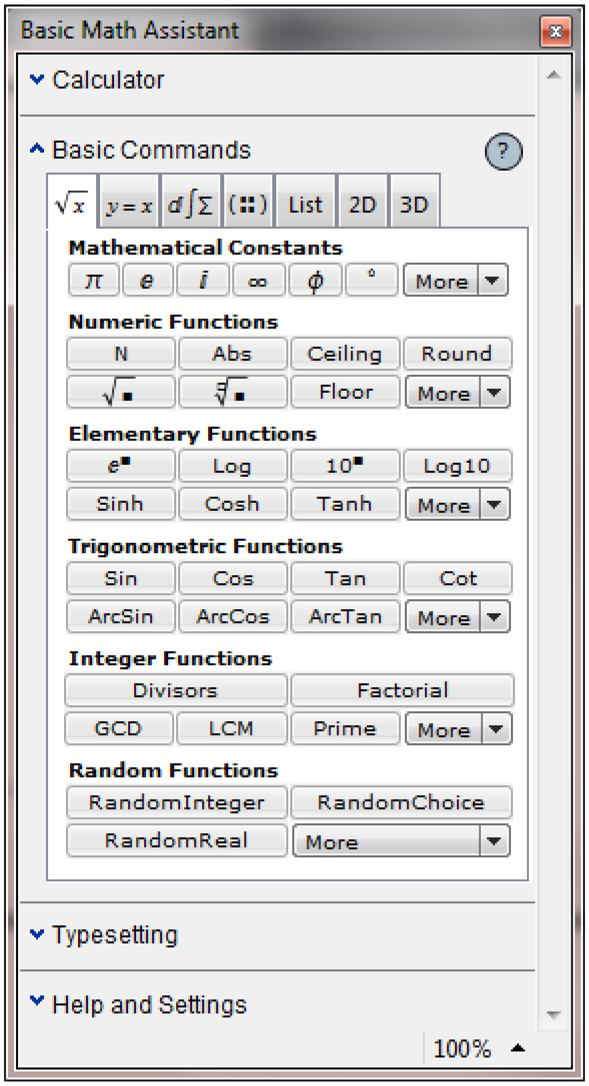
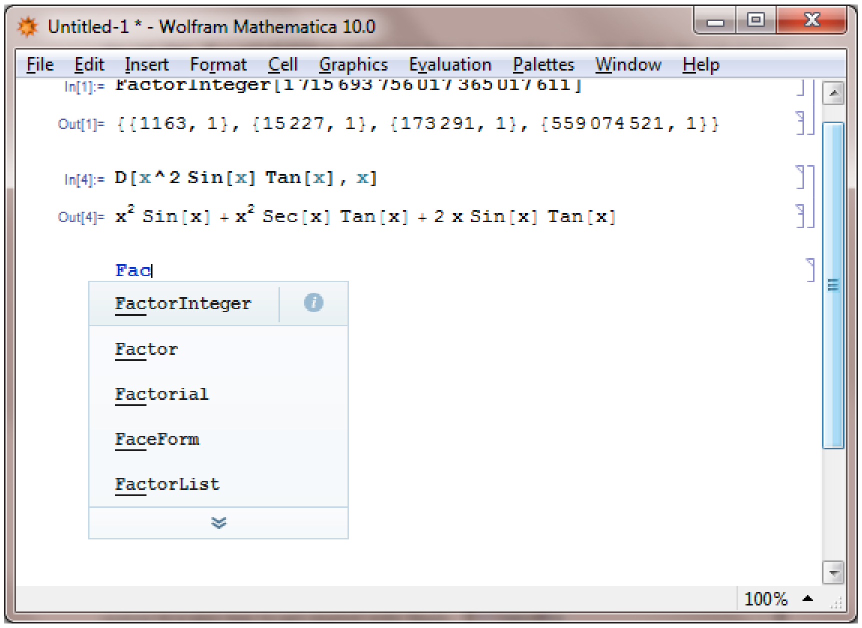


Figure A.3: Mathematica Palette

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]= x² Sin[x] + x² Sec[x] Tan[x] + 2 x Sin[x] Tan[x]

        Fac
        FactorInteger
        Factor
        Factorial
        FaceForm
        FactorList
```

Figure A.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.

A.3 Basic Calculations

A.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]= 1.41421356237309504880168872420969807856967187537694807317667973799⋱
          73799073247846210703885038753432764157273501384623091229702 49⋱
          24836055850737212644121497099935831413222665927505592755799950⋱
          50115278206057147010955997160597027453459686201472851741864 0⋱
          88919860955232923048430871432145083976260362799525140798968 72⋱
          53396546331808829640620615258352395054745750287759961729835 57⋱
          52203375318570113543746034084988471603868999706990048150305 44⋱
          02779031645424782306849293691862158057846311159666871301301 56⋱
          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.

A.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
```

A.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}}}
```

A.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]= $\{1 - t^2, 4 - t^2, 9 - t^2, 16 - t^2\}$

```
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]= $\sqrt{6}$

A.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.

A.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
```

A.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, $a^{-1} \pmod{n}$. 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
```

A.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
```

A.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
```

A.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
```

A.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
```

A.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
```

A.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.

A.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]
```

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}
```

A.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]
```

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]
```

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]= {}
```

A.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.

A.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)
```

A.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.

A.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.

A.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
```

A.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.

A.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.

A.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]= $\left\{\left\{1, 0, \frac{1}{3}\right\}, \left\{\frac{1}{2}, \frac{1}{5}, \frac{5}{6}\right\}, \left\{\frac{3}{7}, \frac{1}{4}, 0\right\}\right\}$

A.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}}
```

A.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 )
```

A.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.

A.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
]
```

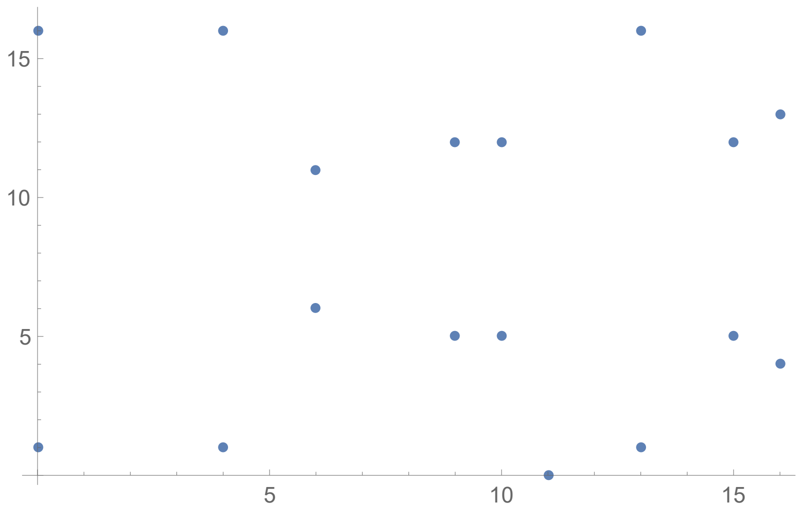
If one takes a quick look at the code it is clear, even if you never programmed in Mathematica, that these are brute force algorithms and hence not the most efficient. So one should be careful with using large moduli.

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[•]=
```

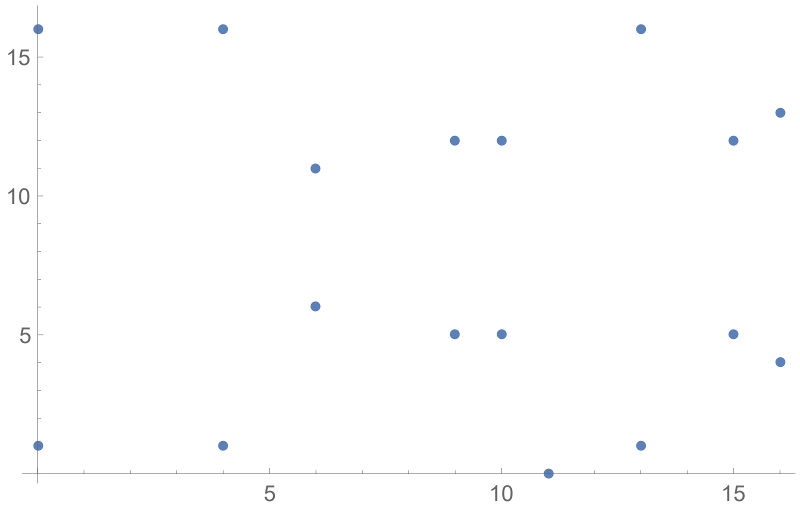


We have also included a function that will do the same thing in one step.

```
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]
]
```

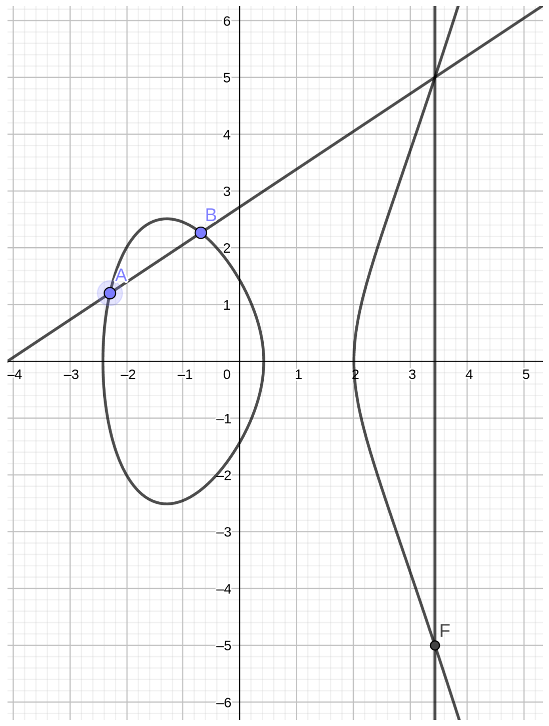
For example, say we wanted the point $(7657, 74389)$ to be on the curve with linear term 3284 and modulus 3263561.

```
In[•]:= ECGenerateCurveConstant[3284, 3 263 561, {7657, 74 389}]

Out[•]= 1 388 410

In[•]:= ECPointOnCurve[3284, 1 388 410, 3 263 561, {7657, 74 389}]

Out[•]= True
```

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

A.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.

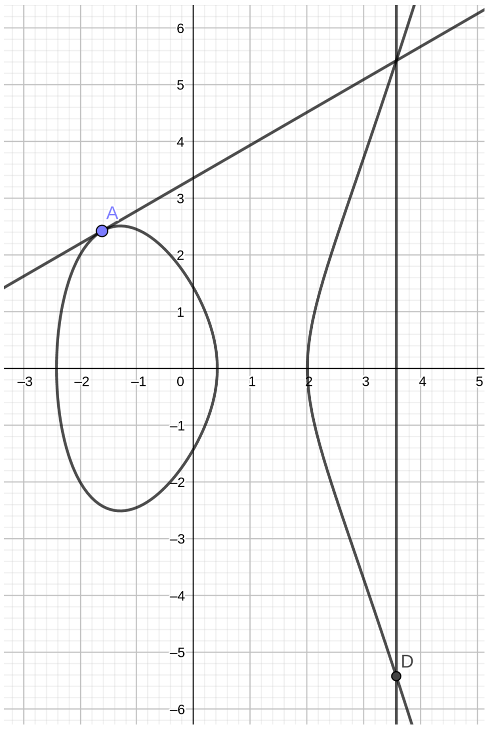
Let E be given by $y^2 = x^3 + bx + c$ and let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, then $P_1 + P_2 = P_3 = (x_3, y_3)$ where

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

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
```

A.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.

A.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
]
```

Appendix B

Introduction to Maxima

B.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/).[33] 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.

B.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.

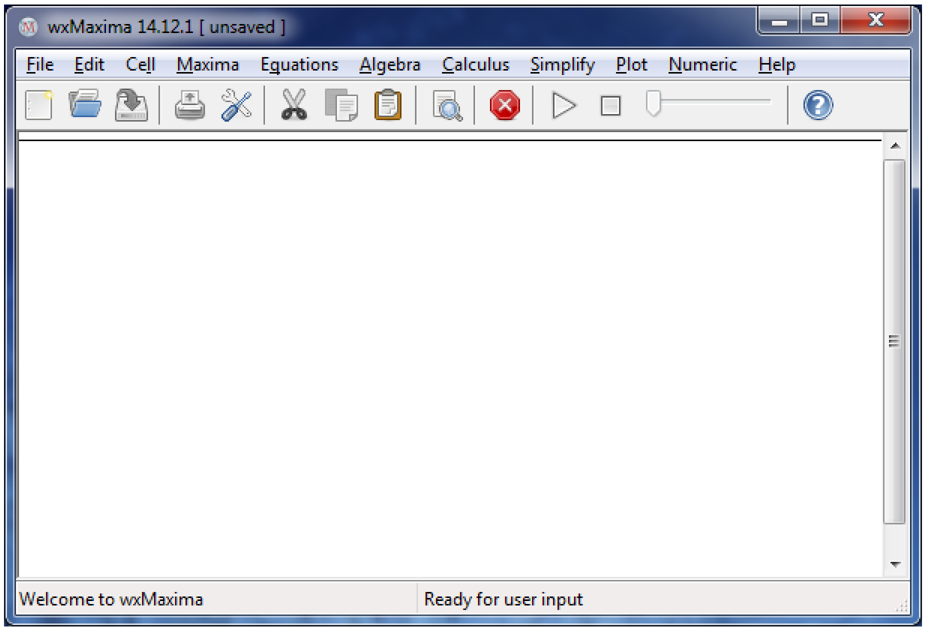
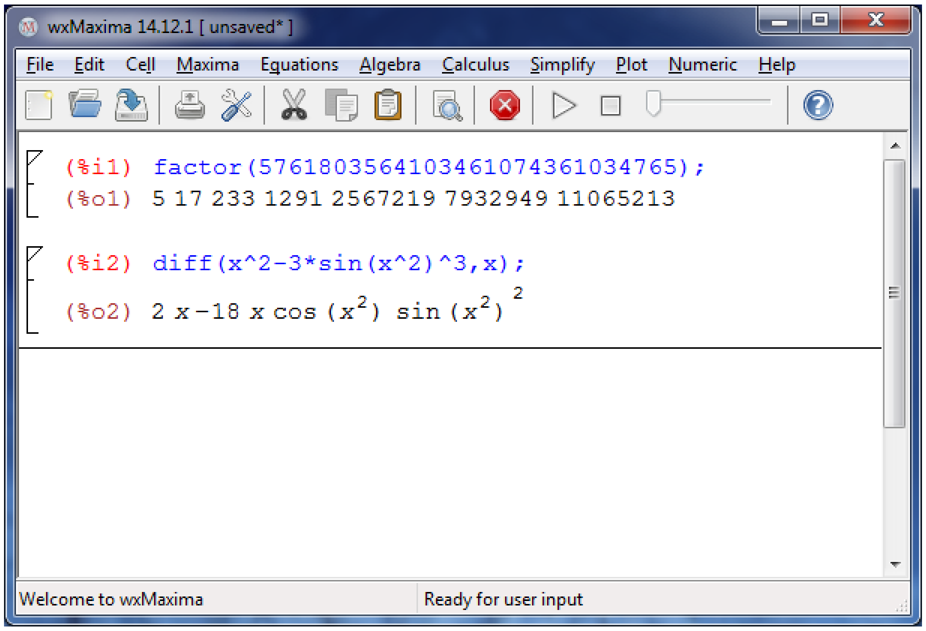

wxMaxima 14.12.1 [unsaved]
File Edit Cell Maxima Equations Algebra Calculus Simplify Plot Numeric Help
Welcome to wxMaxima
Ready for user input

Figure B.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);
                          2
(%o2) 2 x-18 x cos (x^2) sin (x^2)
```

Figure B.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 B.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.

B.3 Basic Calculations

B.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) 3.2188758248682

```
(%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
99358314132226659275055927557999505011527820605715b0
(%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.

B.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 `*` 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.

B.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) $17 \cdot 1376431130111329 \cdot 27578442919l2727$

```
(%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) $17 \cdot 1376431130111329 \cdot 27578442919l2727$

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.

B.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) $h(x) := f(f(f(f(x))))$

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

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

(%o2) $g(x) := \sin(x)$

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

```
(%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

B.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.

B.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

B.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
```

B.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
```

B.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
```

B.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
```

B.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) $[7, 1, 2, 3, 5]$

B.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

B.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

B.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(643782540243018347051875041875015);
(%o11) 643782540243018347051875041874789
(%i12) primep(643782540243018347051875041874789);
(%o12) true
```

B.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.

B.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) $$2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{11 + \frac{1}{5}}}}}}}}$$

```
(%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) from_cf (L) := ratsimp (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) from_cf_n (L, n) := ratsimp (cfdisrep (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) $\frac{1129}{419}$

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

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) $\left[x = \frac{5}{3}, x = -1\right]$

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

B.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
```

B.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.

B.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`.

B.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(565845373623236457659686706707675584 84);
```

(%o3) 187657687360646426668866371606645452 80

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.

B.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))$
```

$\mathrm{mod}\,(3,7) = 3$
$\mathrm{mod}\,(9,7) = 2$
$\mathrm{mod}\,(27,7) = 6$
$\mathrm{mod}\,(81,7) = 4$
$\mathrm{mod}\,(243,7) = 5$
$\mathrm{mod}\,(729,7) = 1$

```
(%i4) zn_primroot(9);
```

(%o4) 2

```
(%i5) for a:1 thru 8 do display(mod(2^a, 9))$
```

$\text{mod}\,(2, 9) = 2$
$\text{mod}\,(4, 9) = 4$
$\text{mod}\,(8, 9) = 8$
$\text{mod}\,(16, 9) = 7$
$\text{mod}\,(32, 9) = 5$
$\text{mod}\,(64, 9) = 1$
$\text{mod}\,(128, 9) = 2$
$\text{mod}\,(256, 9) = 4$

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.

B.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
```

B.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

B.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")`.

B.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, $\dots$, 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.

B.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 ^ but rather `^^`. 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) $$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
(%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) A+M;
```

(%o5) $$\begin{bmatrix} 2 & 2 & 4 \\ 6 & 6 & 11 \\ 10 & 10 & 9 \end{bmatrix}$$

```
(%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) $$\begin{bmatrix} 1 & 0 & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{5} & \frac{5}{6} \\ \frac{3}{7} & \frac{1}{4} & 0 \end{bmatrix}$$

B.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) $\begin{bmatrix} 1 & 0 & -1 & 0 & -1 \\ 0 & 1 & 2 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$

B.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,

$$A^{-1} = \frac{1}{\det(A)}\operatorname{adj}(A)$$

where $\operatorname{adj}(A)$ 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) $\begin{bmatrix}1&0&0&0&7\\0&1&0&0&12\\0&0&1&0&6\\0&0&0&1&1\\0&0&0&0&0\end{bmatrix}$

B.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.

B.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);

$$[[0,1],[0,4],[2,1],[2,4],[3,1],[3,4],[4,2],[4,3]] \qquad (\%\ o2)$$

(% i3) ec_allpoints(1,1,5);

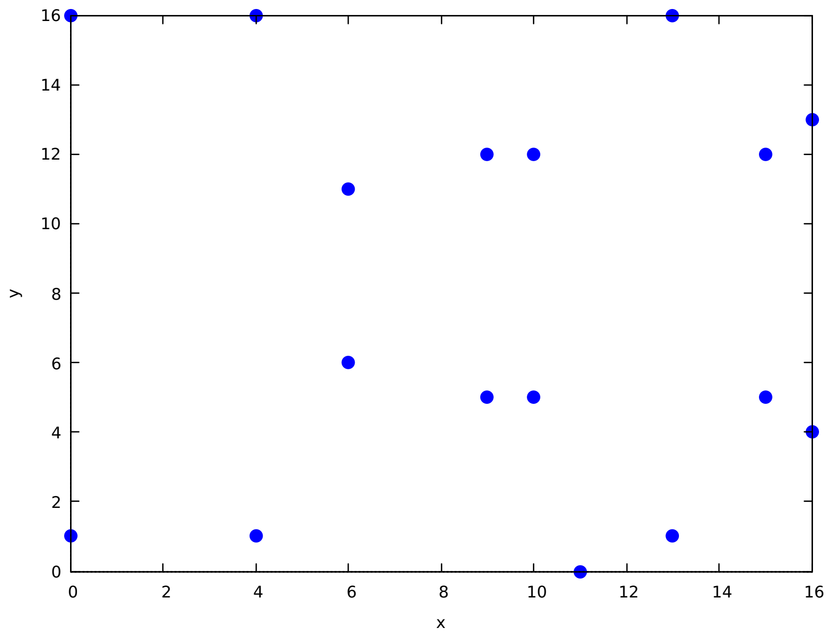
$$[[0,1],[0,4],[2,1],[2,4],[3,1],[3,4],[4,2],[4,3],\infty] \qquad (\%\ o3)$$

(% i4) ec_order(1,1,5);

$$9 \qquad (\%\ o4)$$

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]);

$$\text{true} \quad (\%\ o9)$$

(% i10) ec_pointOnCurve(1,1,5,[1,1]);

$$\text{false} \quad (\%\ o10)$$

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);

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

(% i4) ec_pointWithX(1,1,5,1);

$$none \quad (\%\ o4)$$

(% i5) ec_pointWithY(1,1,5,1);

$$[0, 1] \quad (\%\ o5)$$

(% i6) ec_pointWithY(1,1,5,0);

$$none \quad (\%\ o6)$$

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]);

$$1388410 \quad (\% \, o7)$$

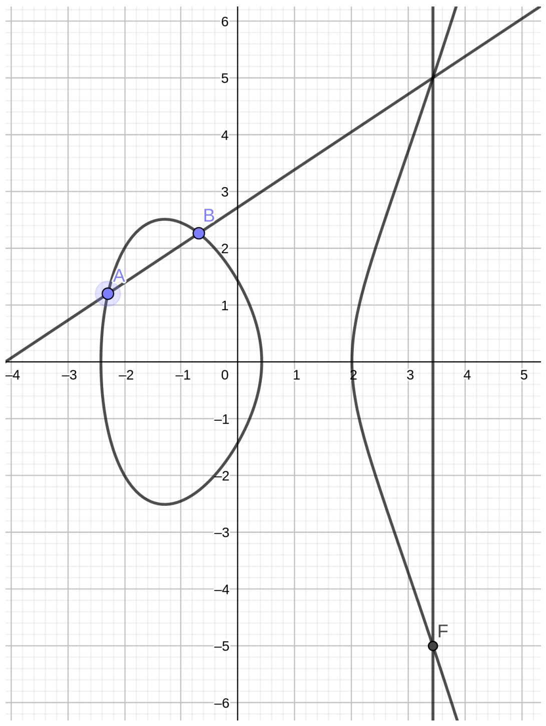
(% i8) ec_pointOnCurve(3284,1388410,3263561,[7657,74389]);

$$\text{true} \quad (\% \, o8)$$

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

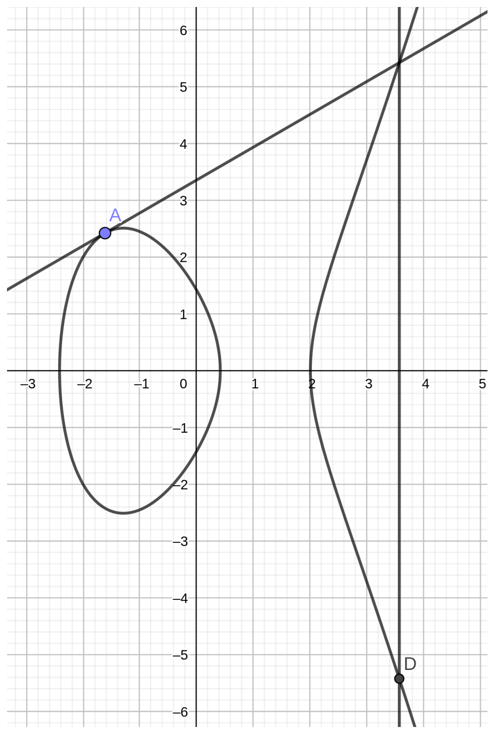
B.7.2 Arithmetic on an Elliptic Curve

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]);

$$[9,6] \qquad (\% \text{o3})$$

(% i4) ec_pointScalarMult(2,3,17,738956431,[13,4]);

$$[5,11] \qquad (\% \text{o4})$$

(% i5) ec_pointFactorialScalarMult(2,3,17,5,[13,4]);

$$[3,6] \qquad (\% \text{o5})$$

(% i6) ec_pointFactorialScalarMult(2,3,17,20,[13,4]);

$$\infty \qquad (\% \text{o6})$$

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]);

$$22 \qquad (\% \text{o7})$$

B.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.

B.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])
)$
```

Appendix C

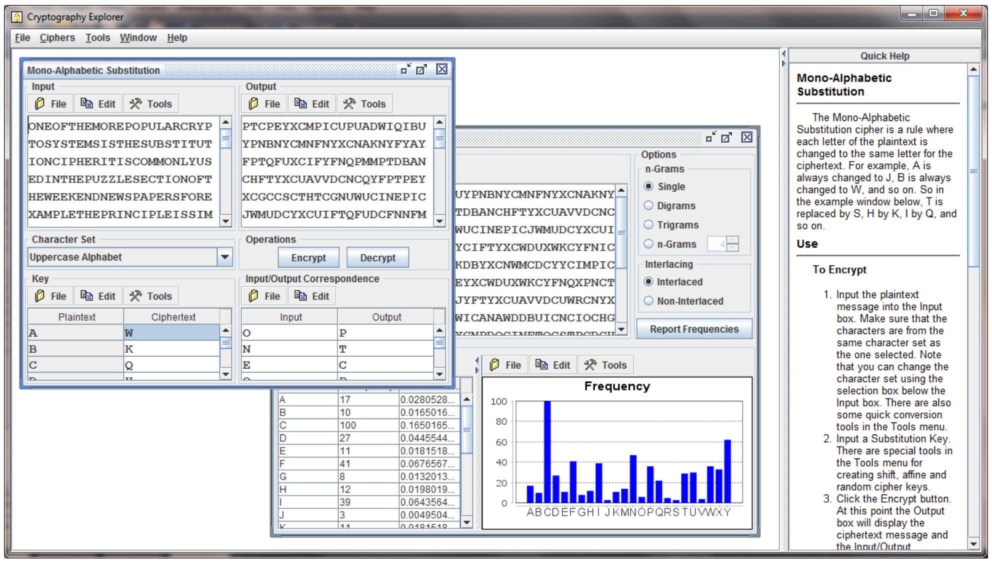
Introduction to Cryptography Explorer

C.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.

C.2 Introduction

The main window to the Cryptography Explorer program is pictured below.



Input:

```
ONEOFTHEMOREPOPULARCRYP
TOSYSTEMSISTHESUBSTITUT
IONCIPHERITISCOMMONLYUS
EDINTHEPUZZLESECTIONOFT
HEWEEKENDNEWSPAPERSFORE
XAMPLETHEPRINCIPLEISSIM
```

Output:

```
PTCPEYXCMPICUPUADWIQIBU
YPNBNYCMNFNYXCNAKNYFYAY
FPTQFUXCIFYFNQPMMPTDBAN
CHFTYXCUAVVDCNCQYFPTPEY
XCGCCSCTHTCGNUWUCINEPIC
JWMUDCYXCUIFTQFUDCFNNFM
```

Character Set: Uppercase Alphabet

Key:

Plaintext	Ciphertext
A	W
B	K
C	Q

Input/Output Correspondence:

Input	Output
O	P
N	T
E	C

```
UYPNBNYCMNFNYXCNAKNY
TDBANCHFTYXCUAVVDCNC
WUCINEPICJWMUDCYXCUI
YCIFTYXCWDUXWKCYFNIC
KDBYXCNWMCDCYYCIMPIC
EYXCWDUXWKCYFNQXPNCT
JYFTYXCUAVVDCUWRCNYX
WICANAWDDBUICNCIOCHG
```

Options: n-Grams: [✓ Single] Digrams Trigrams n-Grams; Interlacing: [✓ Interlaced] Non-Interlaced

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 C.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.

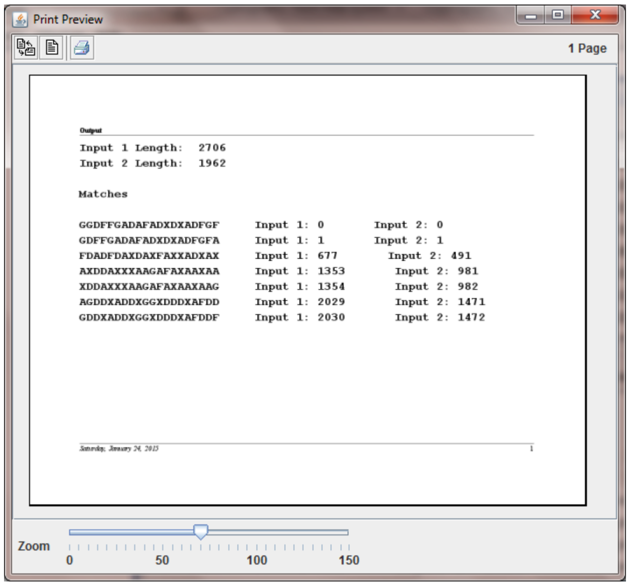
Copy All: Copies the current contents of the input box to the clipboard.

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
```

Figure C.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.

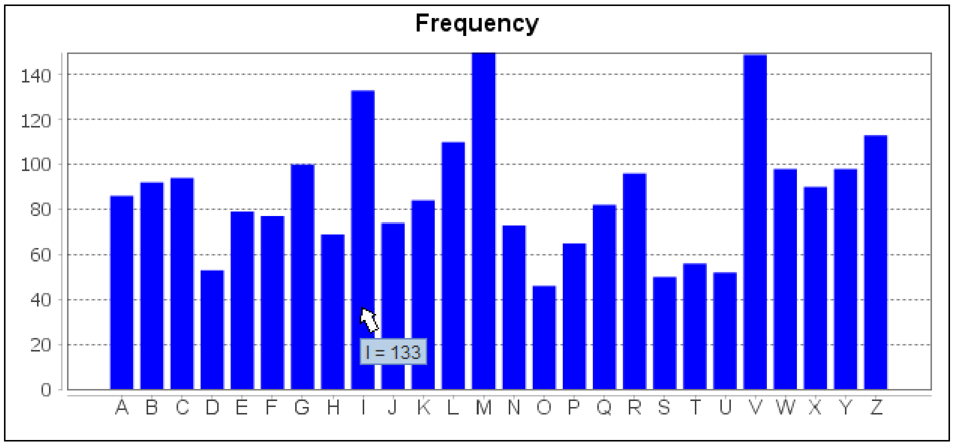


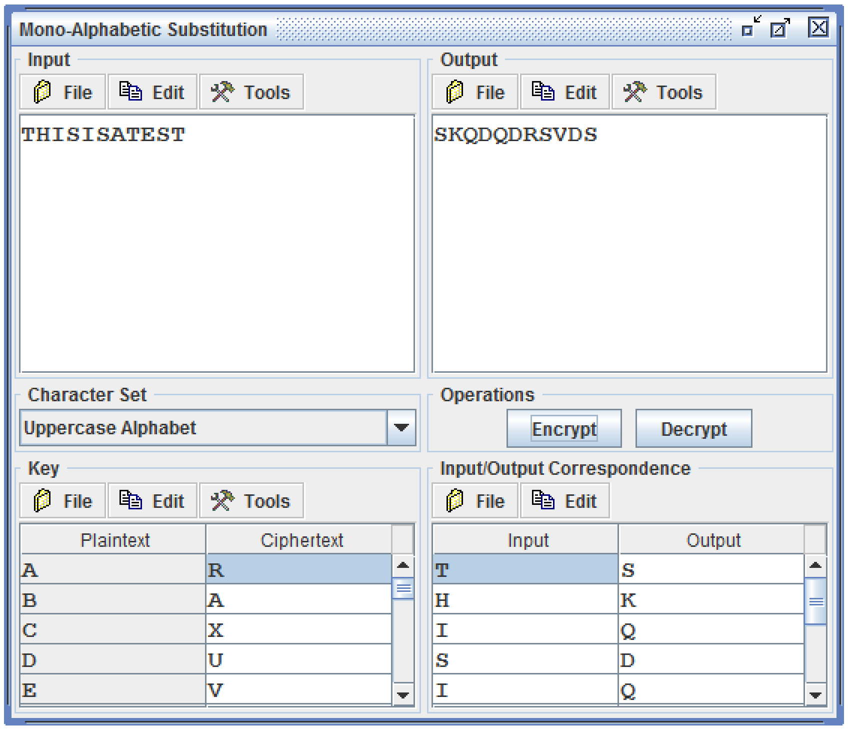
Figure C.3: Bar Charts

C.3 Ciphers

C.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 C.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.

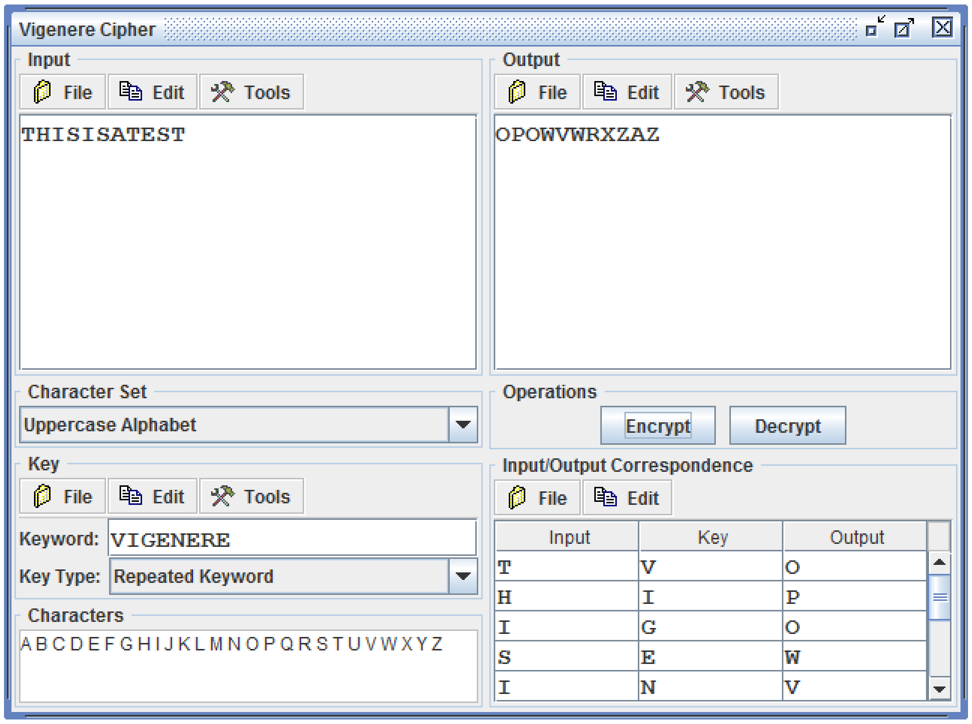
C.3.2 Vigenère

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



Input: THISISATEST

Output: 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 C.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.

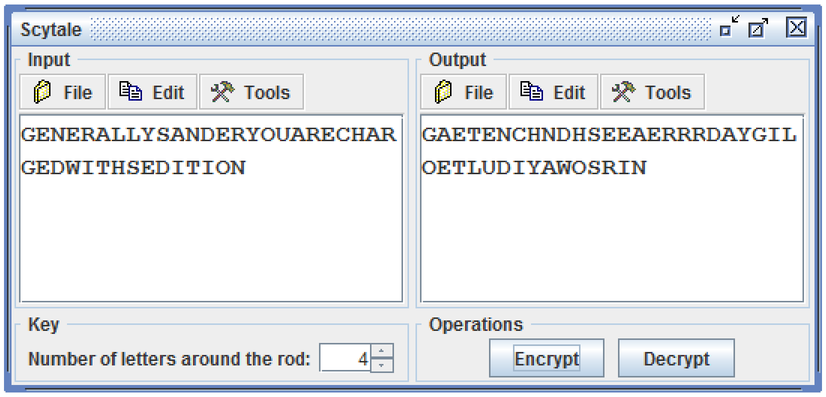
  Repeated Keyword: The Repeated Keyword option is the classical Vigenère cipher that simply repeats the keyword enough times to cover the message.

  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.

C.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 C.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 —

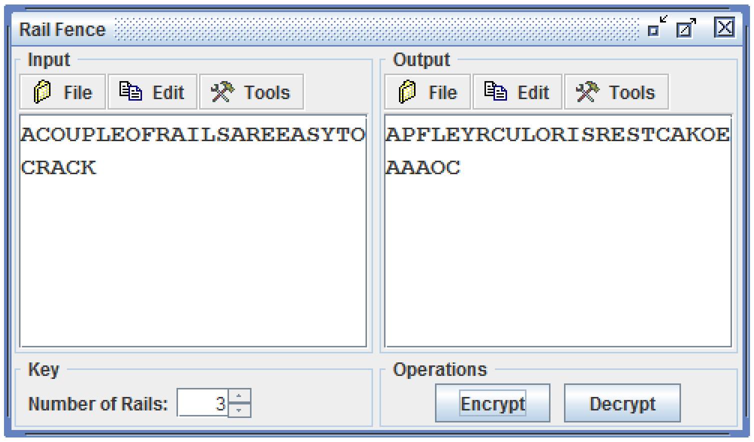
1. Input the ciphertext message into the Input box.
2. Input the number of letters that are written around the rod before coming back to the starting position.
3. Click the Decrypt button. At this point the Output box will display the plaintext message.

C.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.



Number of Rails: 3

```
ACOUPLEOFRAILSAREEASYTO
CRACK
```

```
APFLEYRCULORISRESTCAKOE
AAAOC
```

Figure C.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.

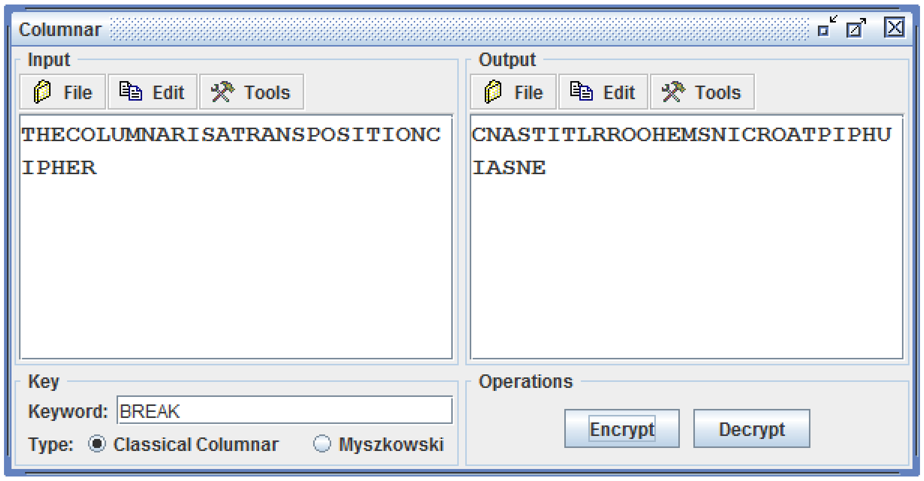
C.3.5 Columnar

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

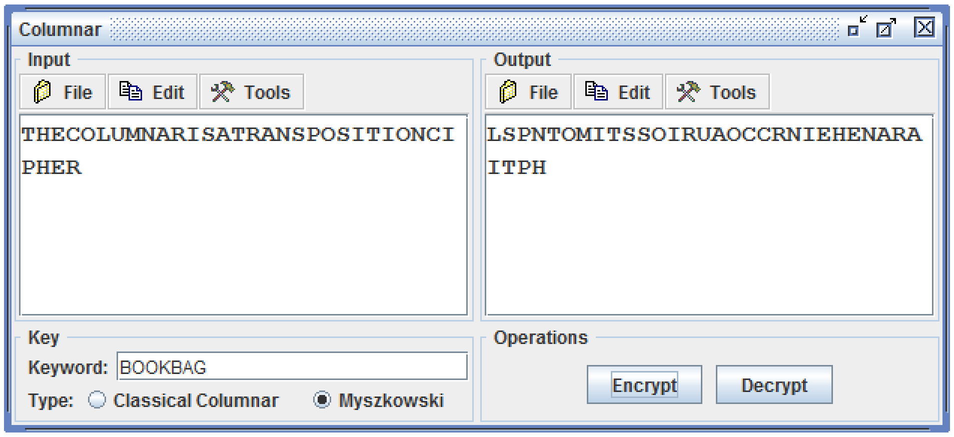
Figure C.8: Columnar Cipher Tool: Classical Mode

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 C.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.

C.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.

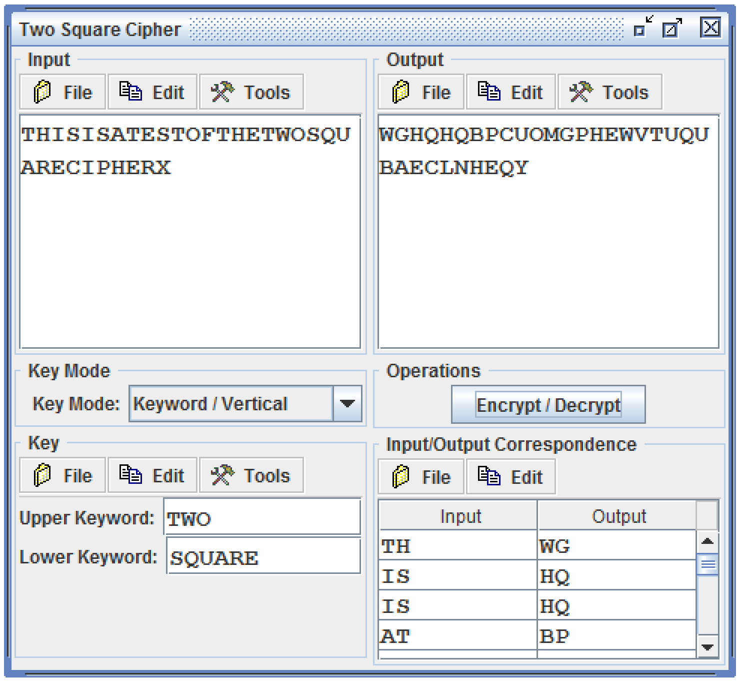
So in vertical mode, TH becomes WG, IS becomes HQ, AD becomes AD, and so on. In horizontal mode, TH becomes HQ, IS becomes WG, TR becomes TR, and so on. The decryption process is exactly like the encryption process.

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

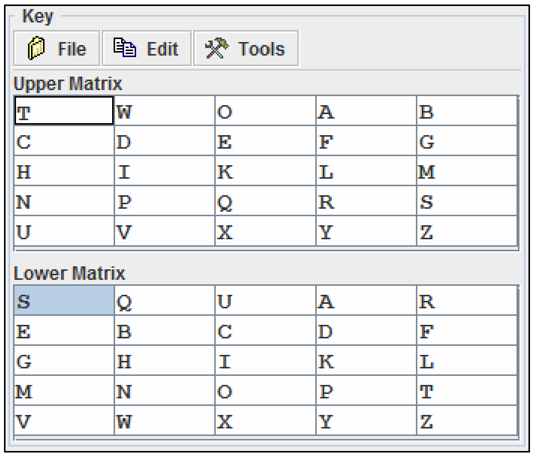
Key Mode: Keyword / Vertical

Upper Keyword: TWO

Lower Keyword: SQUARE

Input	Output
TH	WG
IS	HQ
IS	HQ
AT	BP

Figure C.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 C.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.

C.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

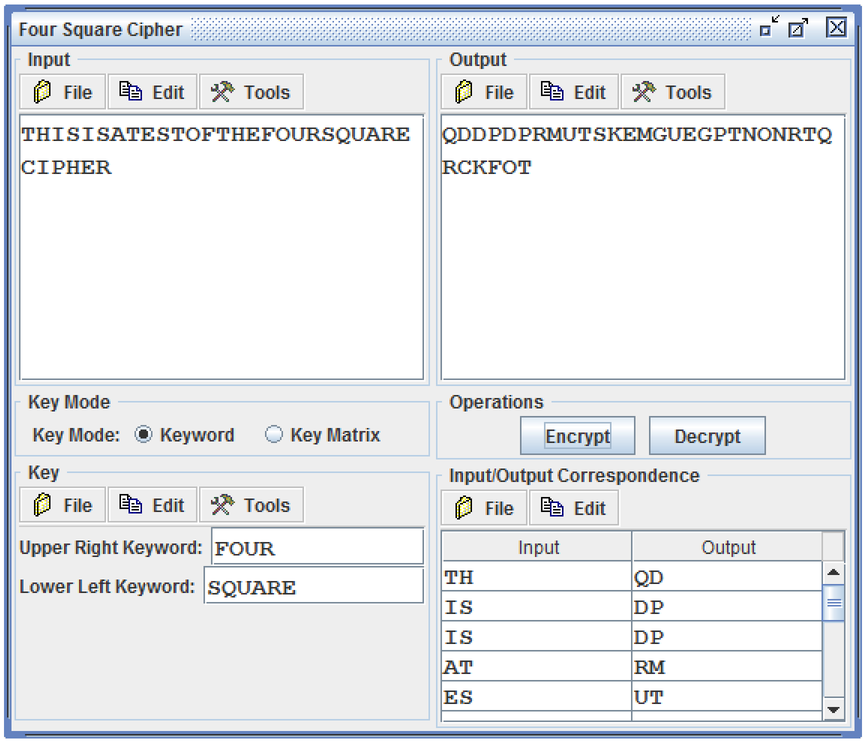
and

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

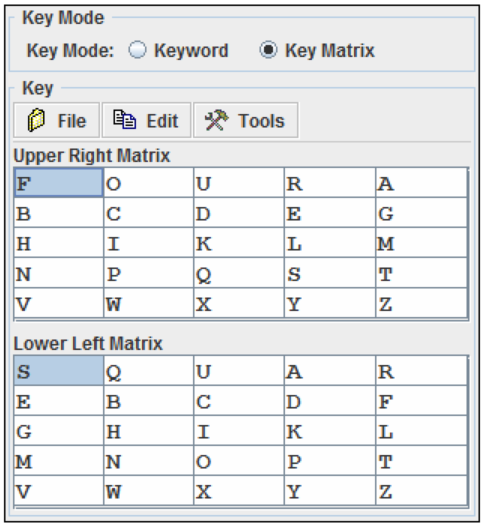
Lower Left Keyword: SQUARE

Input	Output
TH	QD
IS	DP
IS	DP
AT	RM
ES	UT

Figure C.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 C.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.

C.3.8 Playfair

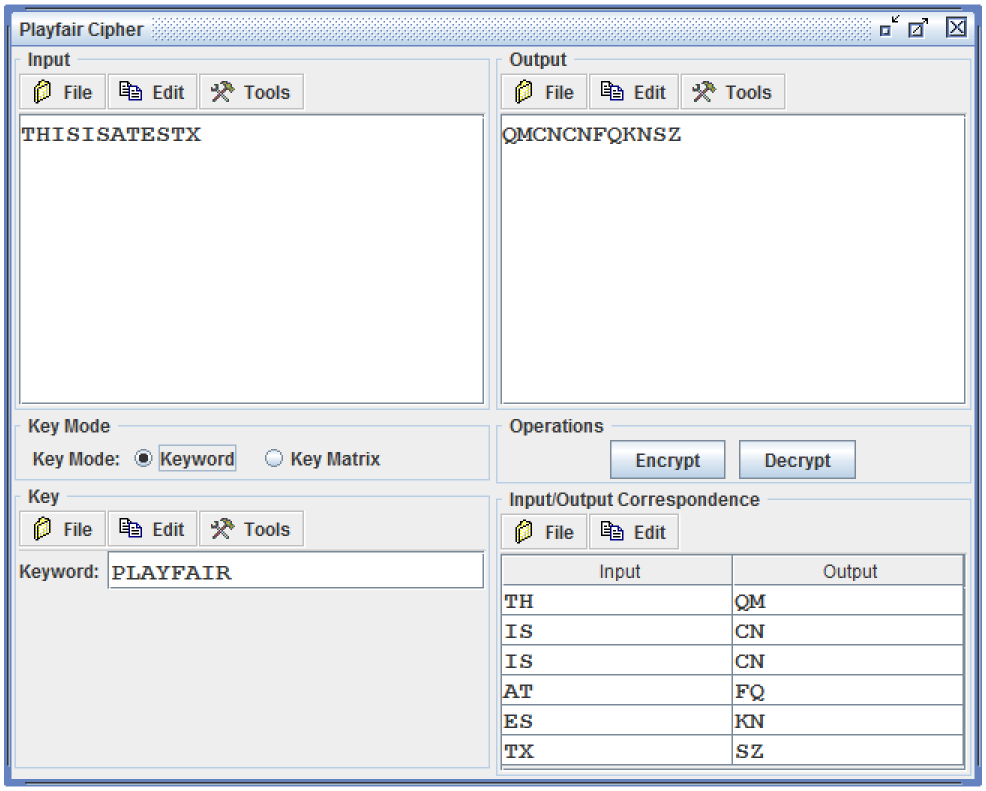
The Playfair cipher is a symmetric encryption technique and was the first digram substitution cipher. It was invented in 1854 by Charles Wheatstone, but was given the name of Lord Playfair (Lyon Playfair) who promoted the use of the cipher. The Playfair cipher was used as a field cipher by British forces 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.



Input: THISISATESTX

Output: QMCNCNFQKNSZ

Key Mode: [✓ Keyword] Key Matrix

Keyword: PLAYFAIR

Input	Output
TH	QM
IS	CN
IS	CN
AT	FQ
ES	KN
TX	SZ

Figure C.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.

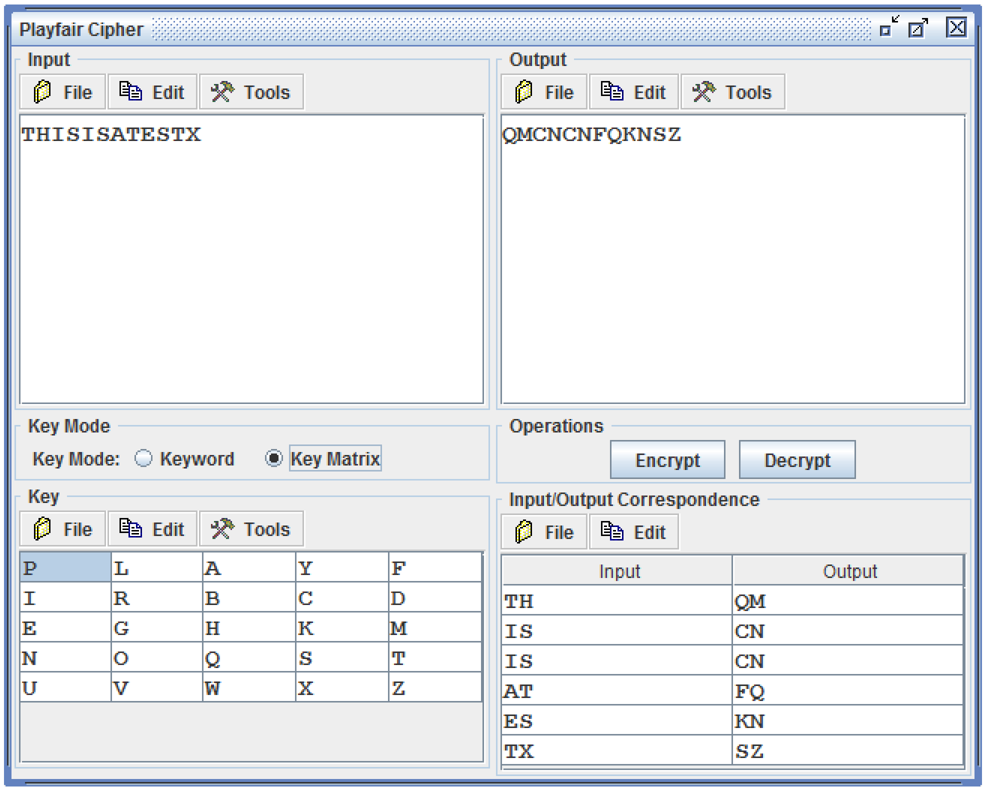
To Decrypt —

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: 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 C.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.

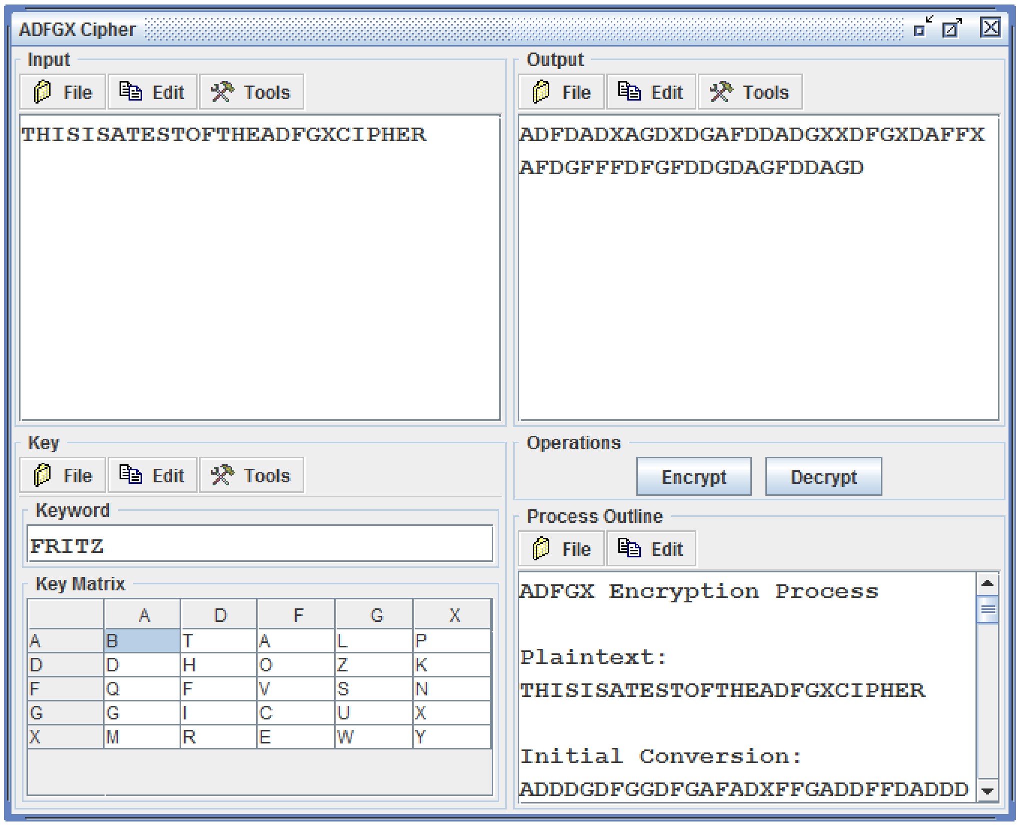
Notes

- 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.

C.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 C.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.

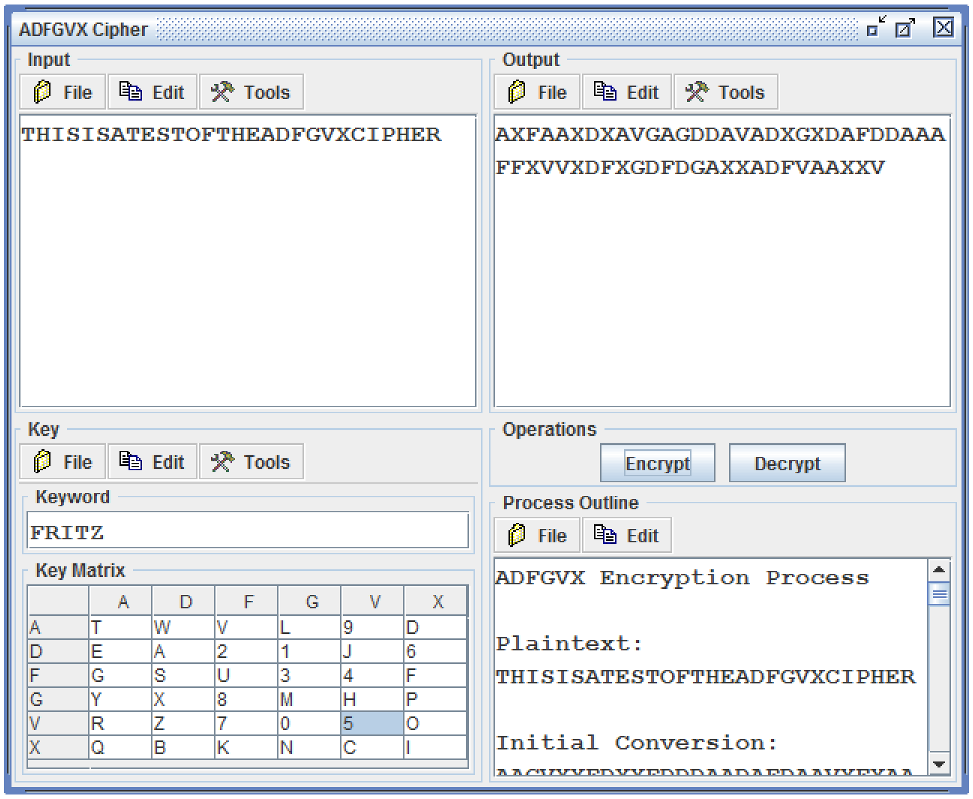
Notes

- We equate I and J in order to get 26 letters down to 25, so the plaintext cannot have any J's in it.

C.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
```

```
FRITZ
```

	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 C.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 —

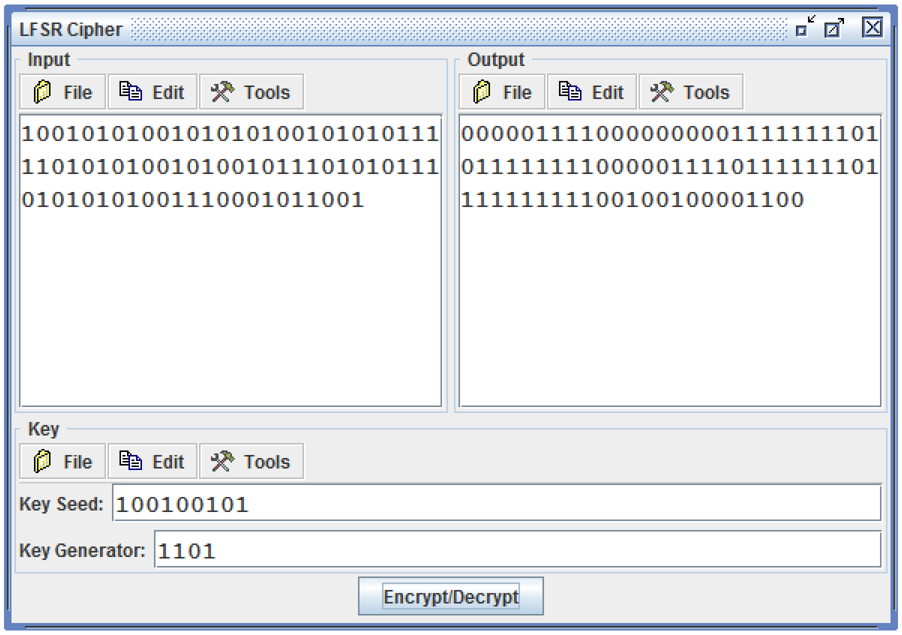
  Copy: Copies the selected portion of the entire Process Outline to the clipboard.

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.

C.3.11 Linear Feedback Shift Register (LFSR)

The LFSR Cipher is a binary stream cipher.



Input

```
1001010100101010100101010111
1101010100101001011101010111
01010101001110001011001
```

Output

```
0000011110000000001111111101
0111111110000011110111111101
11111111100100100001100
```

Key Seed: 100100101

Key Generator: 1101

Figure C.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.

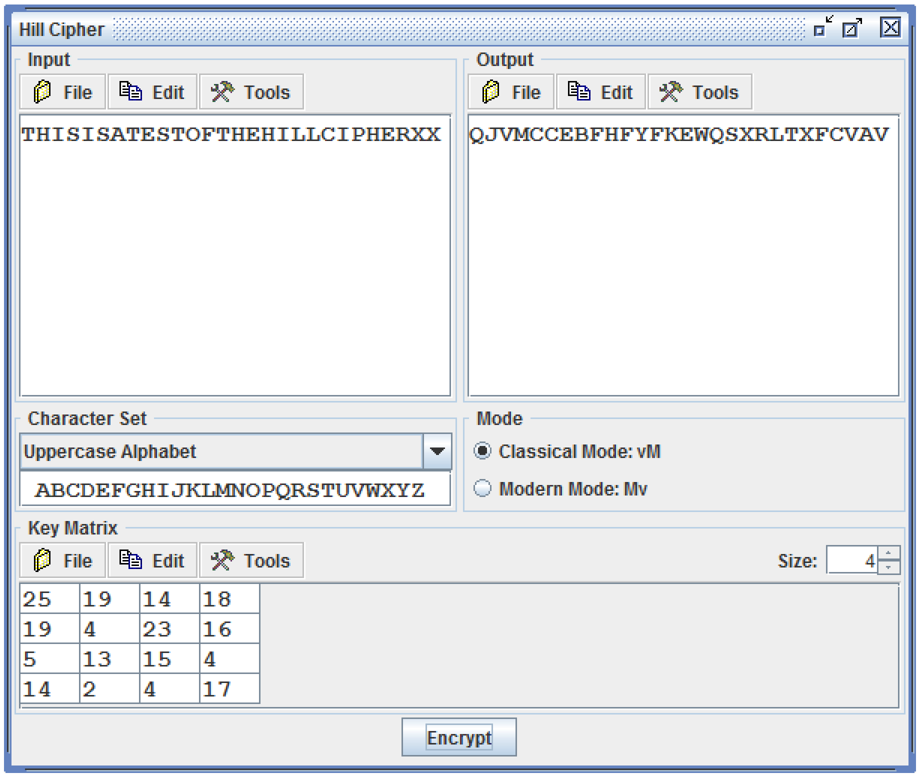
Notes

- 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...

C.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.



Input: THISISATESTOFTHEHILLCIPHERXX

Output: 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 C.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 **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

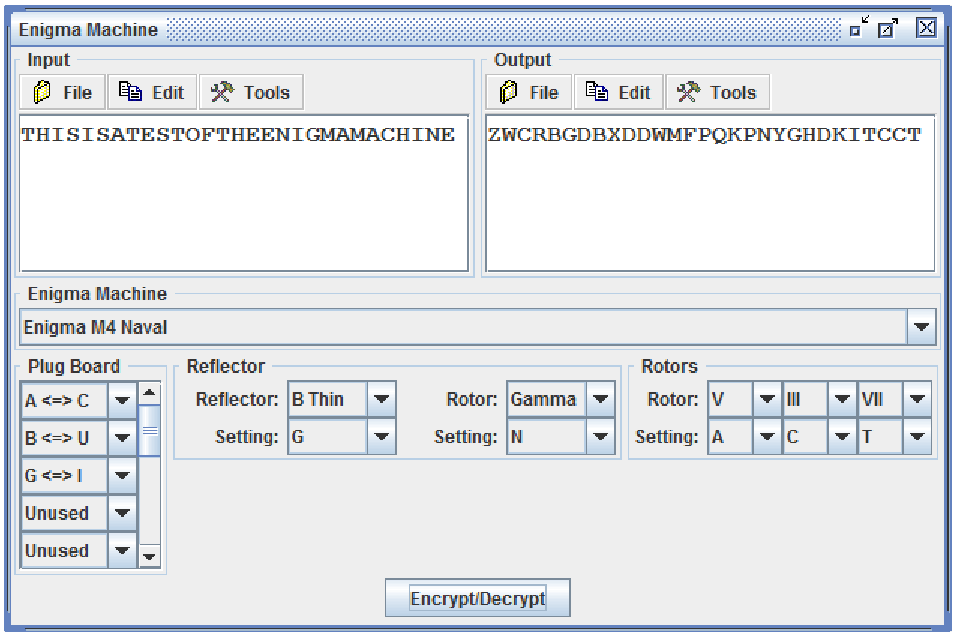
  1. Block the input into blocks of size n (where the key matrix is n X n).
  2. Translate each of these blocks into a row vector v. The correspondence of letters to numbers is done by the position of the letter in the character set. So if the character set is the uppercase alphabet (A B C D E F G H I 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.

C.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 C.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

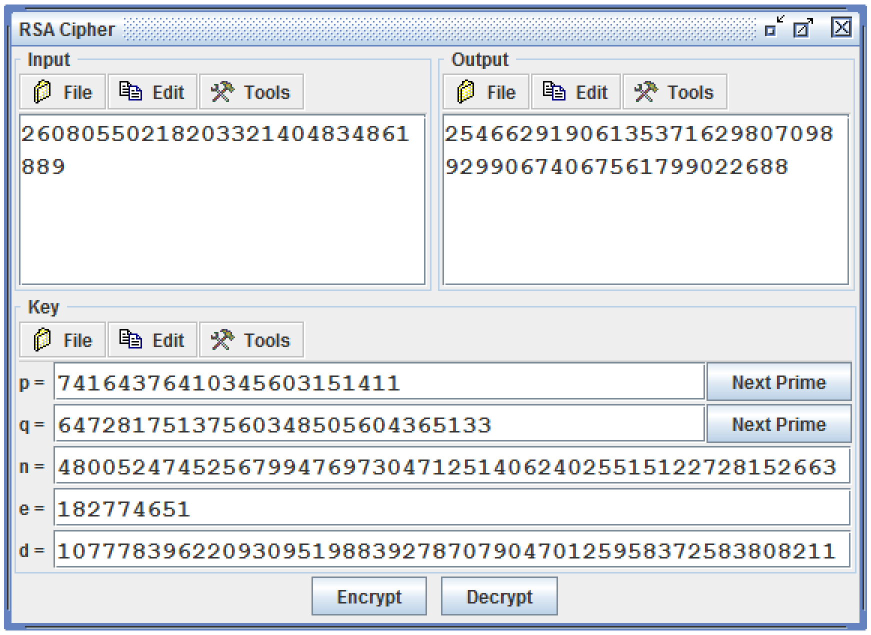
- The rotor and reflector wirings are as follows,
  - Rotor I: Substitution: EKMFLGDQVZNTOWYHXUSPAIBRCJ with Notch at Q.
  - Rotor II: Substitution: AJDKSIRUXBLHWTMCQGZNPYFVOE with Notch at E.
  - Rotor III: Substitution: BDFHJLCPRTXVZNYEIWGAKMUSQO with Notch at V.
  - Rotor IV: Substitution: ESOVPZJAYQUIRHXLNFTGKDCMWB 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.

C.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 C.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.

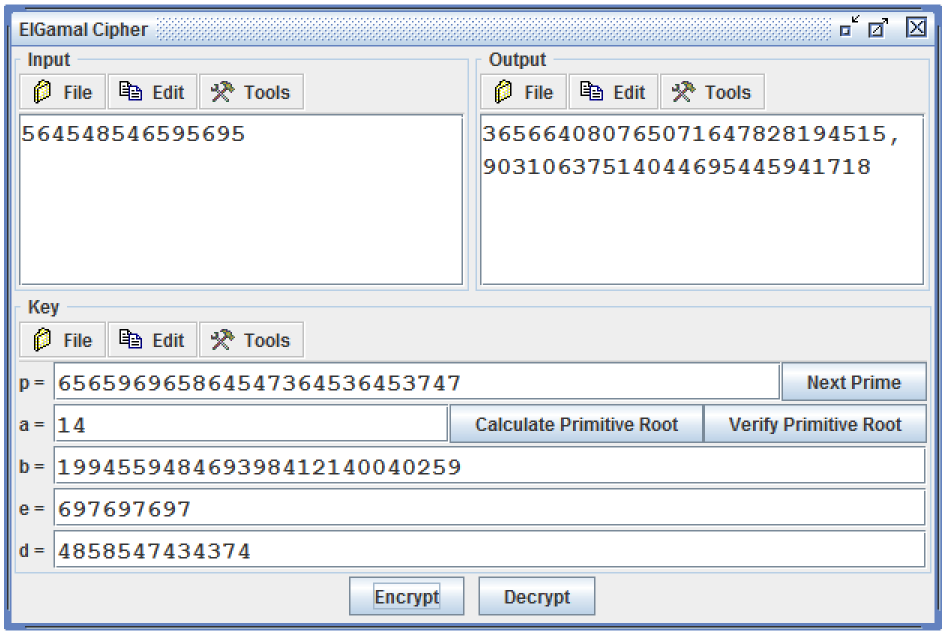
- The Encrypt and Decrypt buttons will apply the RSA algorithm to the input using the current key. Encrypt will use e as the exponent and Decrypt will use d as the exponent.

C.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 $tr^{-e} = m \pmod{p}$.



```
564548546595695
```

```
365664080765071647828194515,
90310637514044695445941718
```

```
p = 656596965864547364536453747
a = 14
b = 199455948469398412140040259
e = 697697697
d = 4858547434374
```

Figure C.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.

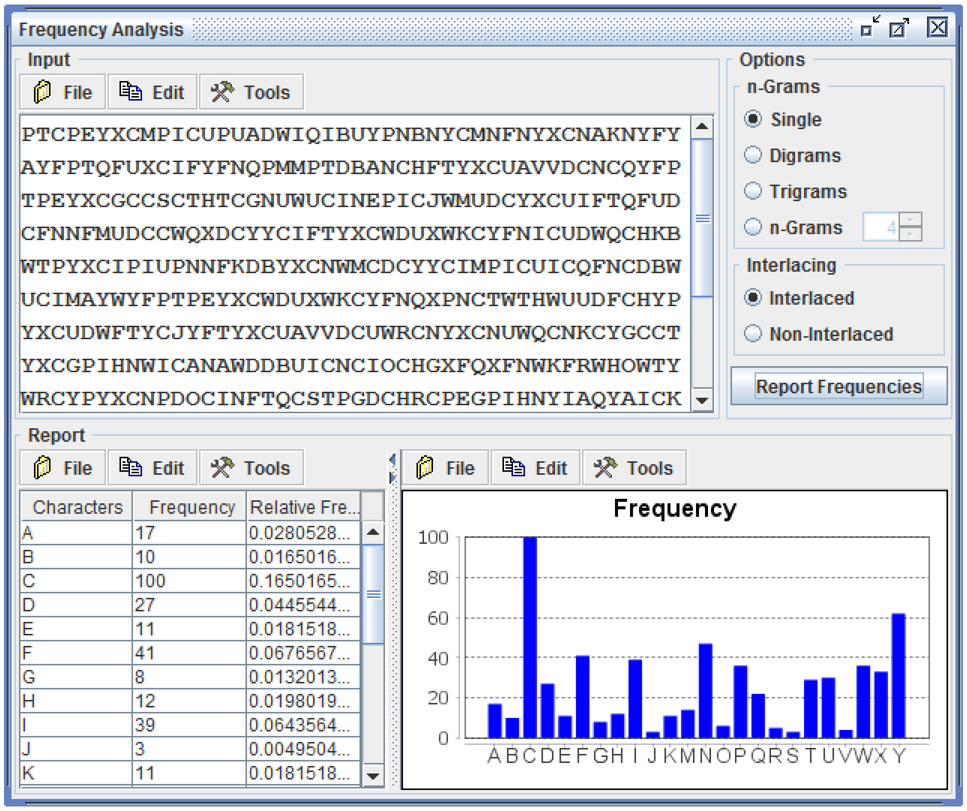
Notes

- 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.

C.4 Text and Stream Analysis

C.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 C.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.

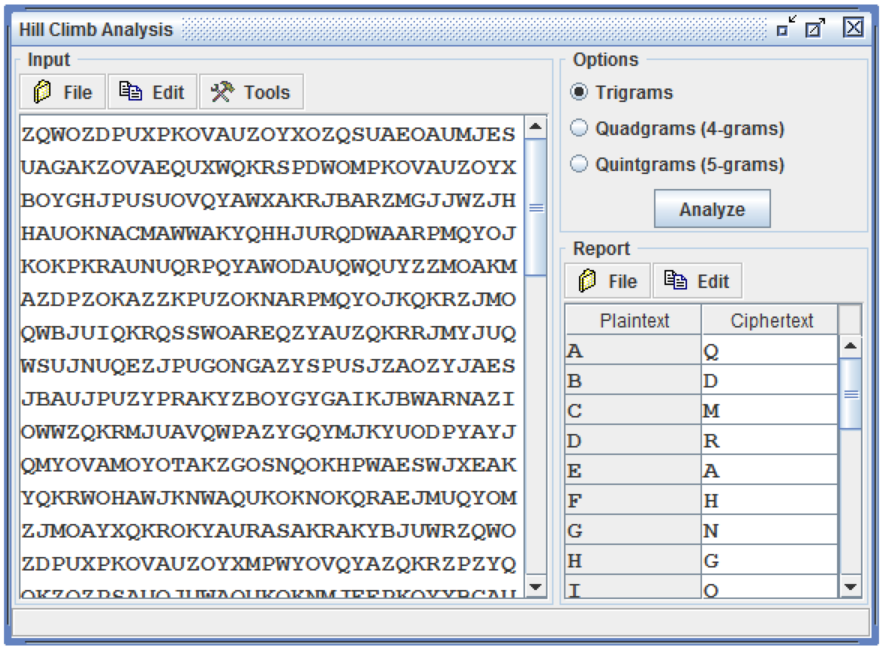
Plot Range of Data: Plots a selected range of the data. When the user selects this option a small dialog box will appear asking for the number of data items to plot. When the user selects the number of items the graph is changed to have only that number of bars showing. Also there will appear a slider 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.

C.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.



Options: [✓ Trigrams] Quadgrams (4-grams) Quintgrams (5-grams)

```
ZQWOZDPUXPKOVAUZOYXOZQSUAEOAUMJES
UAGAKZOVAEQUXWQKRSPDWOMPKOVAUZOYX
BOYGHJPUSUOVQYAWXAKRJBARZMGJJWZJH
HAUOKNACMAWWAKYQHHJURQDWAARPMQYOJ
KOKPKRAUNUQRPQYAWODAUQWQUYZZMOAKM
AZDPZOKAZZKPUZOKNARPMQYOJKQKRZJMO
QWBJUIQKRQSSWOAREQZYAUZQKRRJMYJUQ
WSUJNUQEZJPUGONGAZYSPUSJZAOZYJAES
JBAUJPUZYPRAKYZBOYGYGAIKJBWARNAZI
OWWZQKRMJUAVQWPAZYGQYMJKYUODPYAYJ
QMYOVAMOYOTAKZGOSNQOKHPWAESWJXEAK
YQKRWOHAWJKNWAQUKOKNOKQRAEJMUQYOM
ZJMOAYXQKROKYAURASAKRAKYBJUWRZQWO
ZDPUXPKOVAUZOYXMPWYOVQYAZQKRZPZYQ
```

Plaintext	Ciphertext
A	Q
B	D
C	M
D	R
E	A
F	H
G	N
H	G
I	O

Figure C.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.

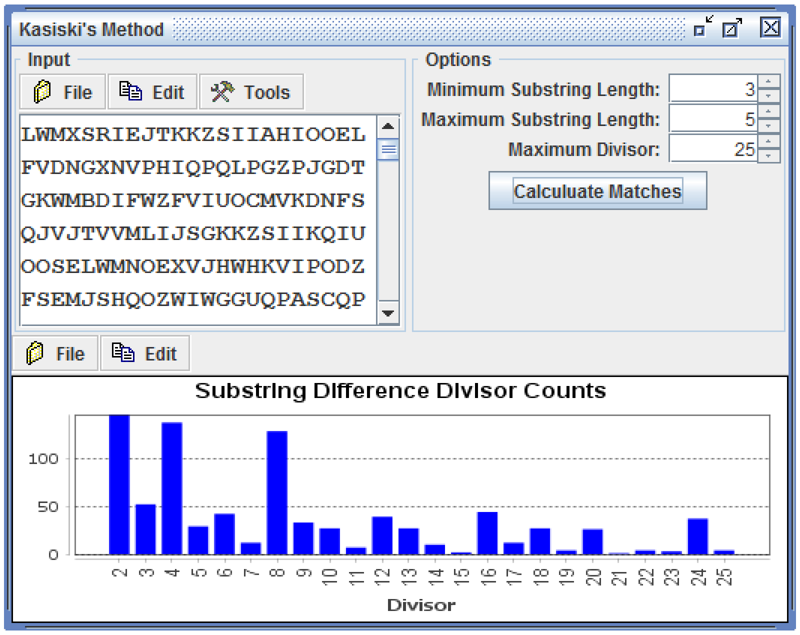
- Depending on your ciphertext, using a different data set may produce better results. In some cases, using trigrams may get closer to the substitution key than using quadgrams or quintgrams.

- 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.

C.4.3 Kasiski's Method

Kasiski's Method, which is also known as Kasiski's Test or Kasiski examination was developed by Friedrich Kasiski[28] in 1863 but it seems to have been independently discovered by Charles Babbage[14] 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 C.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 —

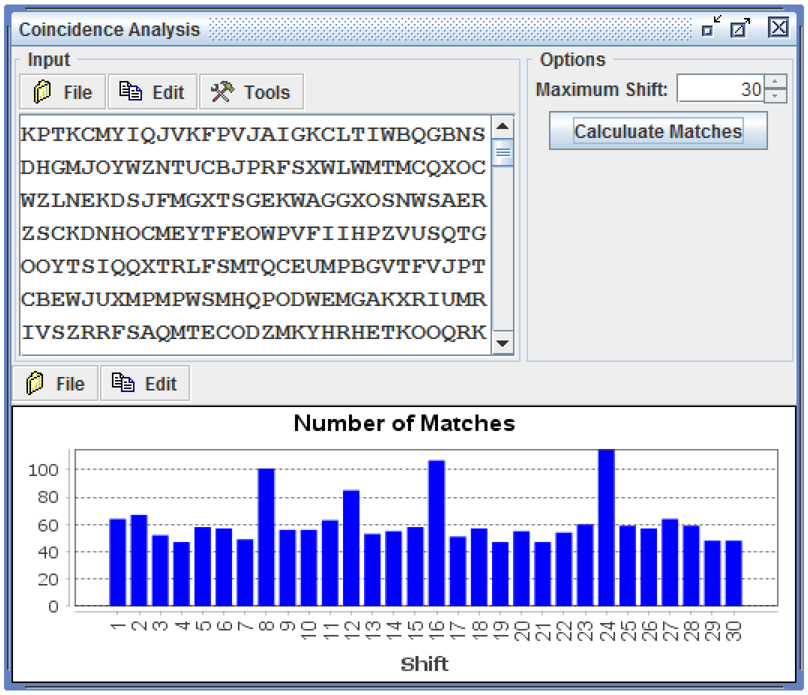
  Save As: Saves the current chart to a png file.
  Print: Prints the current chart to the selected printer.
  Print Preview: Prints the current chart to the print preview display.

  Edit —

  Copy: Copies the chart to the clipboard.

C.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.



```
KPTKCMYIQJVKFPVJAIGKCLTIWBQGBNS
DHGMJOYWZNTUCBJPRFSXWLWMTMCQXOC
WZLNEKDSJFMGXTSGEKWAGGXOSNWSAER
ZSCKDNHOCMEYTFEOWPVFIIHPZVUSQTG
OOYTSIQQXTRLFSMTQCEUMPBGVTFVJPT
CBEWJUXMPMPWSMHQPODWEMGAKXRIUMR
IVSZRRFSAQMTECODZMKYHRHETKOOQRK
```

Maximum Shift: 30

Figure C.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 —

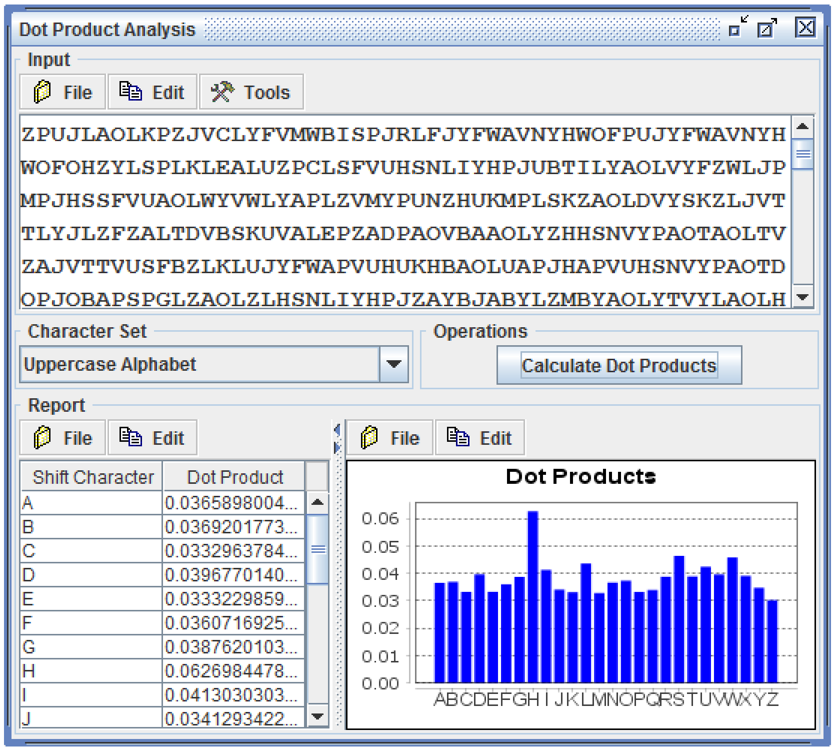
  Save As: Saves the current chart to a png file.
  Print: Prints the current chart to the selected printer.
  Print Preview: Prints the current chart to the print preview display.

Edit —

Copy: Copies the chart to the clipboard.

C.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 C.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 —

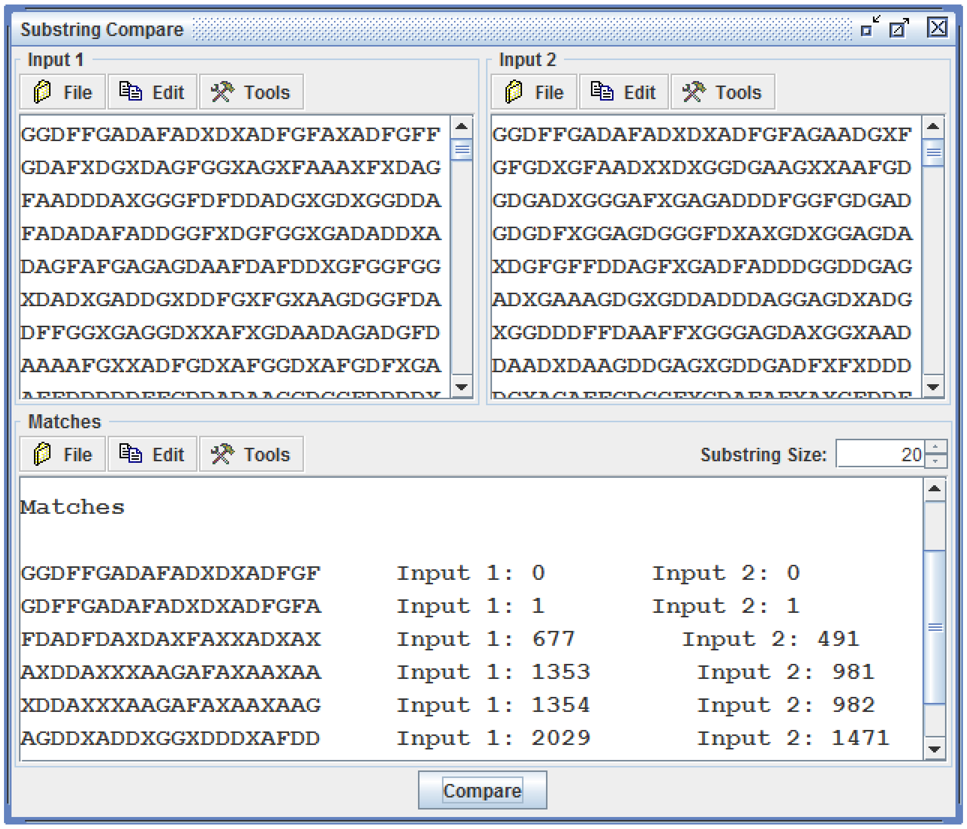
  Copy: Copies the chart to the clipboard.

C.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.



```
GGDFFGADAFADXDXADFGFAXADFGFF
GDAFXDGXDAGFGGXAGXFAAAXFXDAG
FAADDDAXGGGFDFDDADGXGDXGGDDA
FADADAFADDGGFXDGFGGXGADADDXA
DAGFAFGAGAGDAAFDAFDDXGFGGFGG
XDADXGADDGXDDFGXFGXAAGDGGFDA
DFFGGXGAGGDXXAFXGDAADAGADGFD
AAAAFGXXADFGDXAFGGDXAFGDFXGA
```

```
GGDFFGADAFADXDXADFGFAGAADGXF
GFGDXGFAADXXDXGGDGAAGXXAAFGD
GDGADXGGGAFXGAGADDDFGGFGDGAD
GDGDFXGGAGDGGGFDXAXGDXGGAGDA
XDGFGFFDDAGFXGADFADDDGGDDGAG
ADXGAAAGDGXGDDADDDAGGAGDXADG
XGGDDDFFDAAFFXGGGAGDAXGGXAAD
DAADXDAAGDDGAGXGDDGADFXFXDDD
```

```
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 C.28: Substring Comparison Tool

C.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.

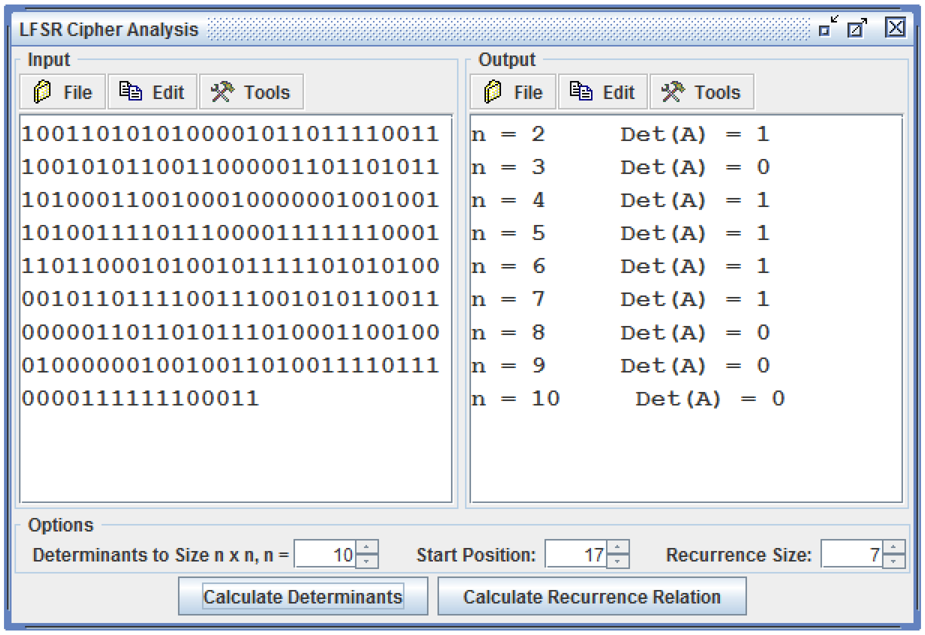
How to Use the Tool

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.

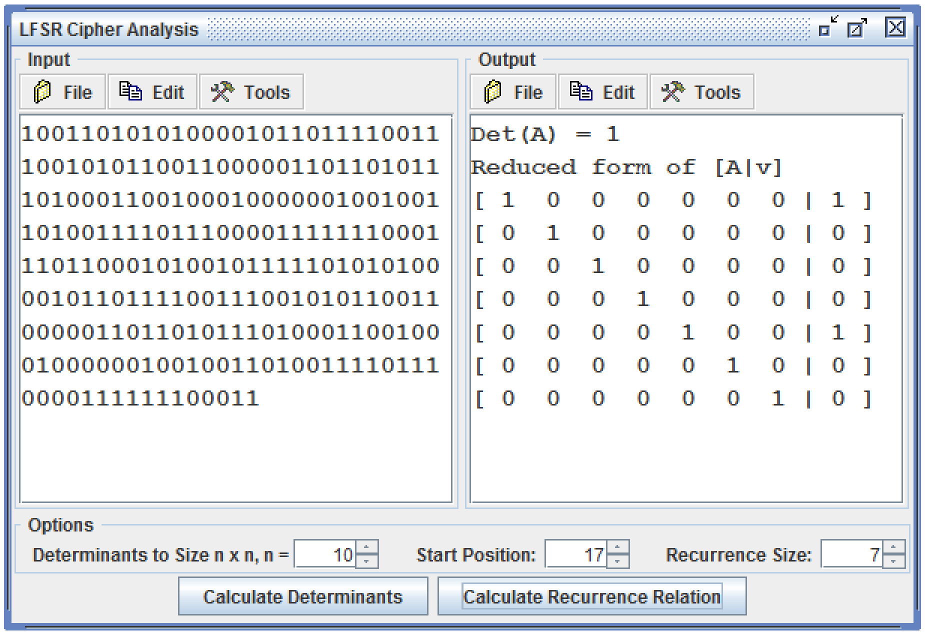


```
1001101010100001011011110011
1001010110011000001101101011
1010001100100010000001001001
1010011110111000011111110001
1101100010100101111101010100
0010110111100111001010110011
0000011011010111010001100100
0100000010010011010011110111
0000111111100011
```

```
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 C.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 C.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.

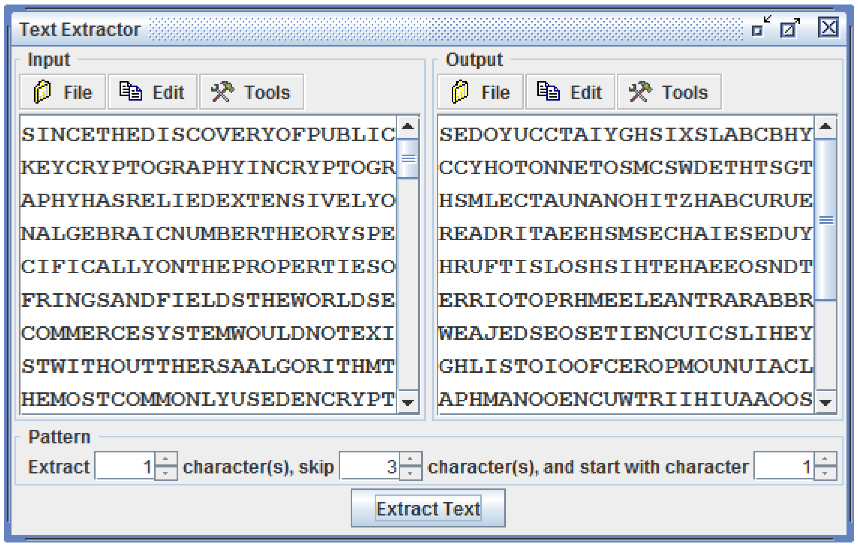
Notes

- 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.

C.5 Text Tools

C.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
GHLISTOIOOFCEROPMOUNUIACL
APHMANOOENCUWTRIIHIUAAOOS
```

Extract 1 character(s), skip 3 character(s), and start with character 1

Figure C.31: Text Extractor Tool

How to Use the Tool

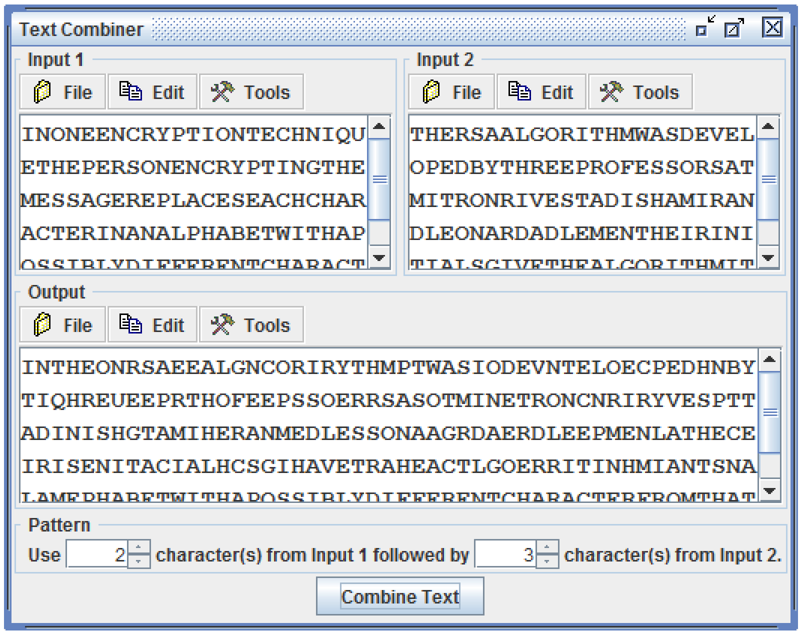
1. Input the text you wish to extract from into the Input box.
2. Select the extraction pattern.
3. Click on the Extract Text button. At this point the number of matches between shifts of 1 and the maximum will be displayed in the report bar chart.

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.

C.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 C.32: Text Combiner Tool

How to Use the Tool

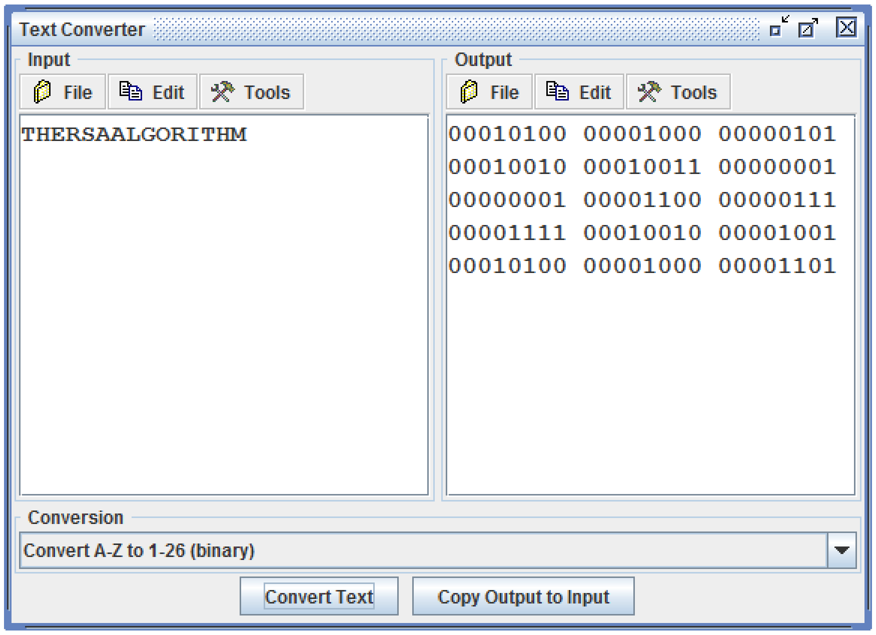
1. Input the two texts you wish to combine into the two Input boxes.
2. Select the combining pattern.
3. Click on the Combine Text button.

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.

C.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 C.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.

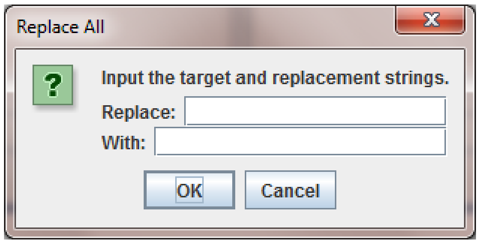
Remove Double Characters: This will place an X between any double characters and if the double character is XX it will place a Z between them. So OO would be changed to OXO and XX to XZX.

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

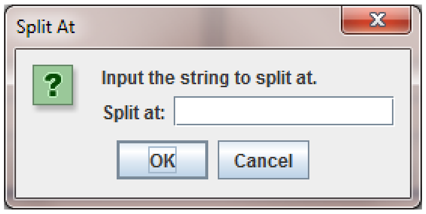
Input the target and replacement strings.

Replace:

With:

Figure C.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 C.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.

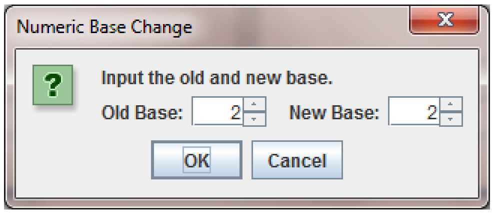
Convert Binary to Octal: Converts a binary number to octal.

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 C.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.

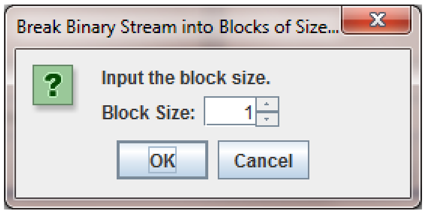
Break Binary Stream into Blocks of 128: Breaks a binary stream of 0's and 1's onto blocks of 128. If the input is not 0's and 1's the program will generate an error.

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.


Break Binary Stream into Blocks of Size...

Input the block size.

Block Size: 1

Figure C.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.

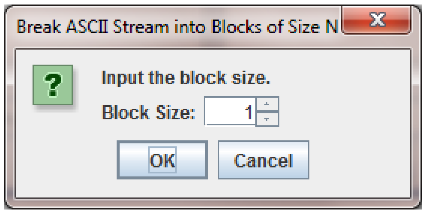
Break ASCII Stream into Blocks of 30: Breaks a number stream onto blocks of 30. If the input is not a stream of numbers with no spaces the program will generate an error.

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.


Break ASCII Stream into Blocks of Size N

Input the block size.

Block Size: 1

Figure C.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.

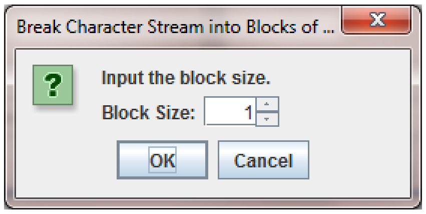
Break Character Stream into Blocks of 50: Breaks a character stream onto blocks of 50.

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.


Break Character Stream into Blocks of ...

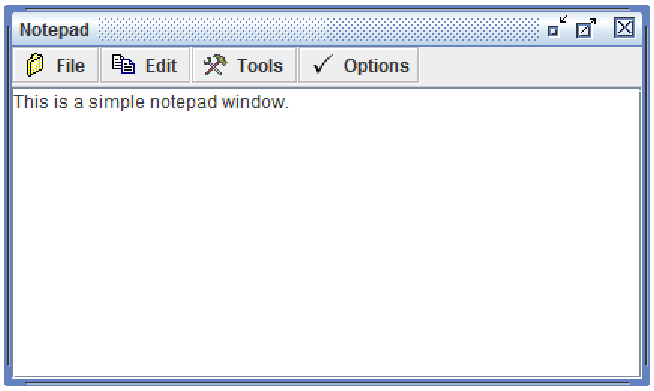
Input the block size.

Block Size: 1

Figure C.39: Character Stream Block Dialog

C.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.

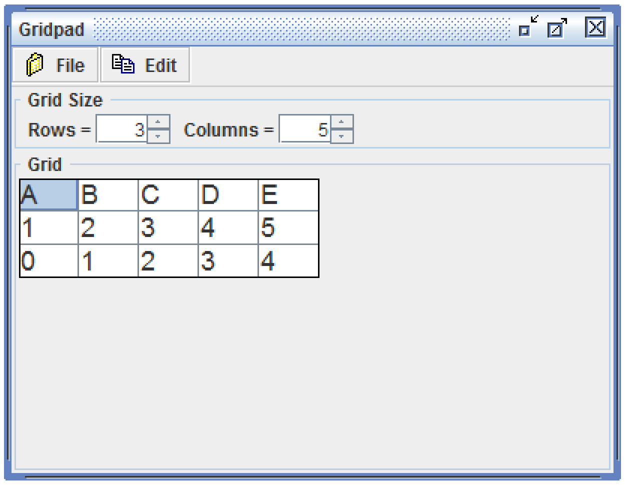


This is a simple notepad window.

Figure C.40: Notepad

C.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.



Grid Size

Rows = 3 Columns = 5

Grid

A	B	C	D	E
1	2	3	4	5
0	1	2	3	4

Figure C.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.

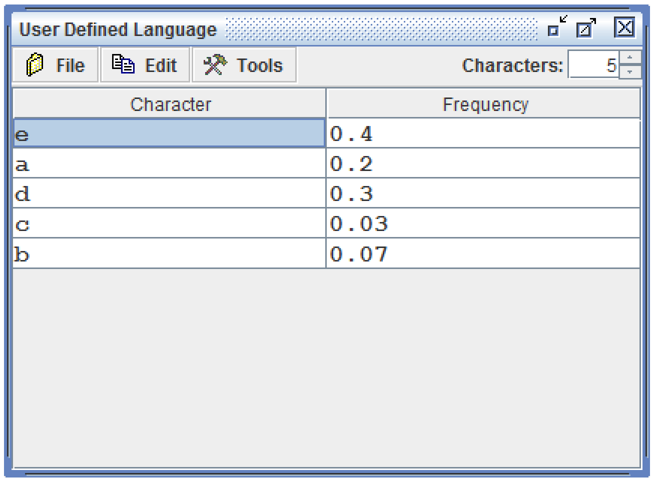
Transpose Grid: This transposes the grid, it makes rows into columns and columns into rows.

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).

C.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.



Character	Frequency
e	0.4
a	0.2
d	0.3
c	0.03
b	0.07

Figure C.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.

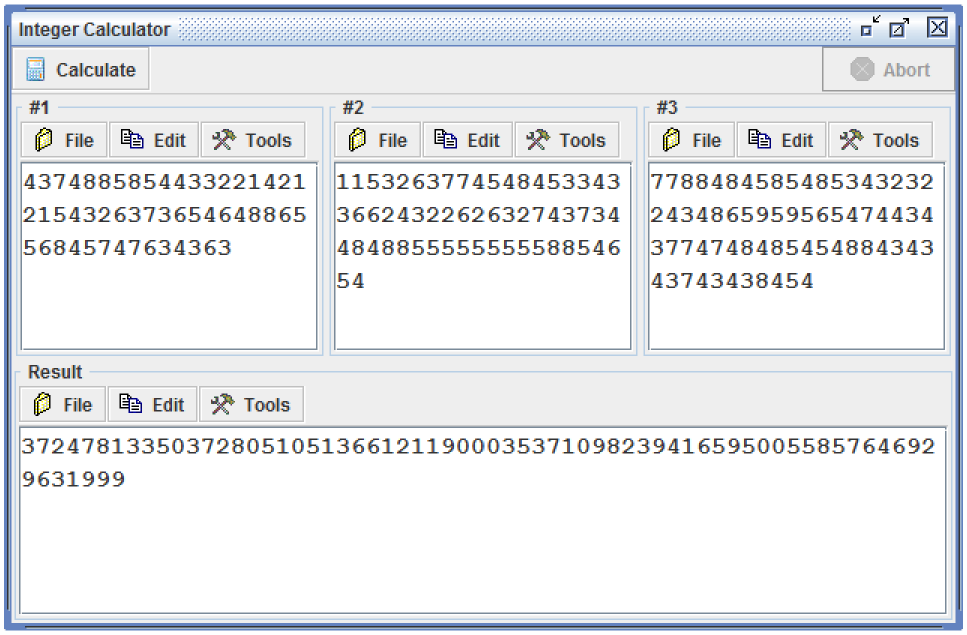
Notes

- 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.

C.6 Calculators

C.6.1 Integer Calculator

The Integer Calculator is a simple infinite precision integer arithmetic tool.



```
#1
4374885854433221421
2154326373654648865
56845747634363
```

```
#1
4374885854433221421
2154326373654648865
56845747634363

#2
1153263774548453343
3662432262632743734
4848855555555588546
54

#3
7788484585485343232
2434865959565474434
3774748485454884343
43743438454

Result
3724781335037280510513661211900035371098239416595005585764692
9631999
```

Figure C.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 $a^e \pmod{n}$.

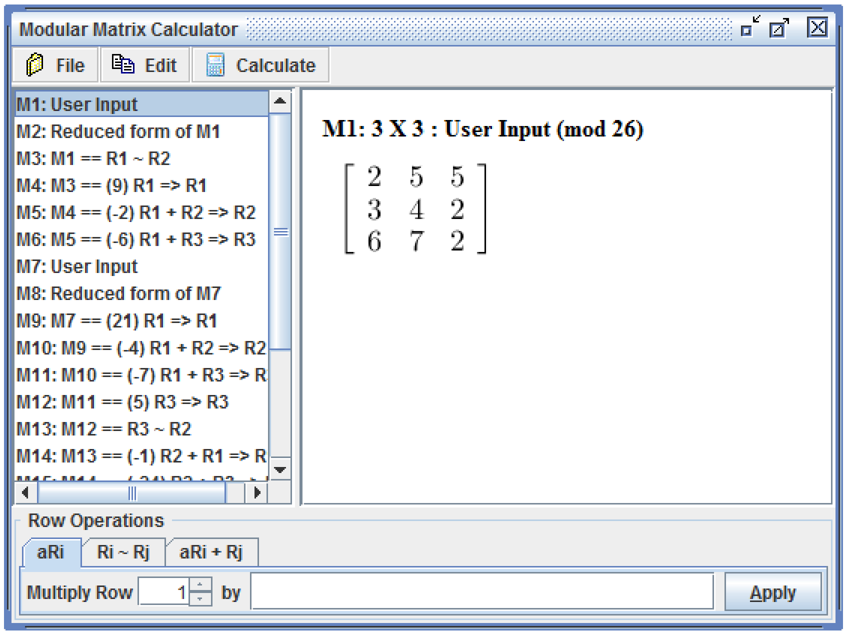
chrem(a1, n1, a2, n2, ... ar, nr) returns the value x that satisfies the congruences $x = a_1 \pmod{n}_1$, $x = a_2 \pmod{n}_2$, $\ldots$, $x = a_r \pmod{n}_r$. The list of residues and moduli may be as long as the user chooses.

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.

C.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)

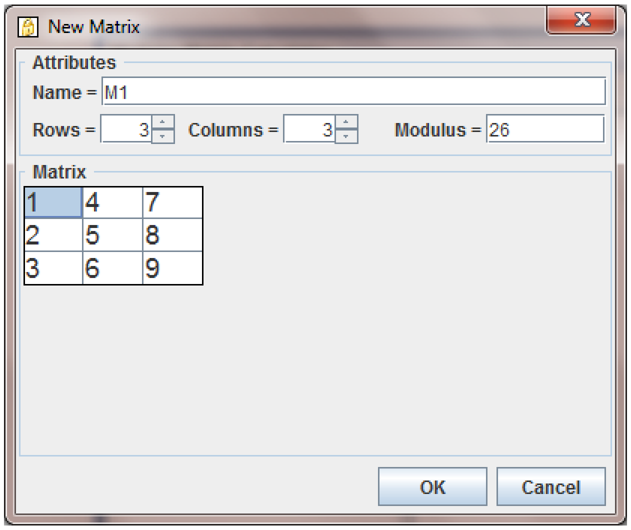
$$\begin{bmatrix} 2 & 5 & 5 \\ 3 & 4 & 2 \\ 6 & 7 & 2 \end{bmatrix}$$

Multiply Row 1 by

Figure C.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 C.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.

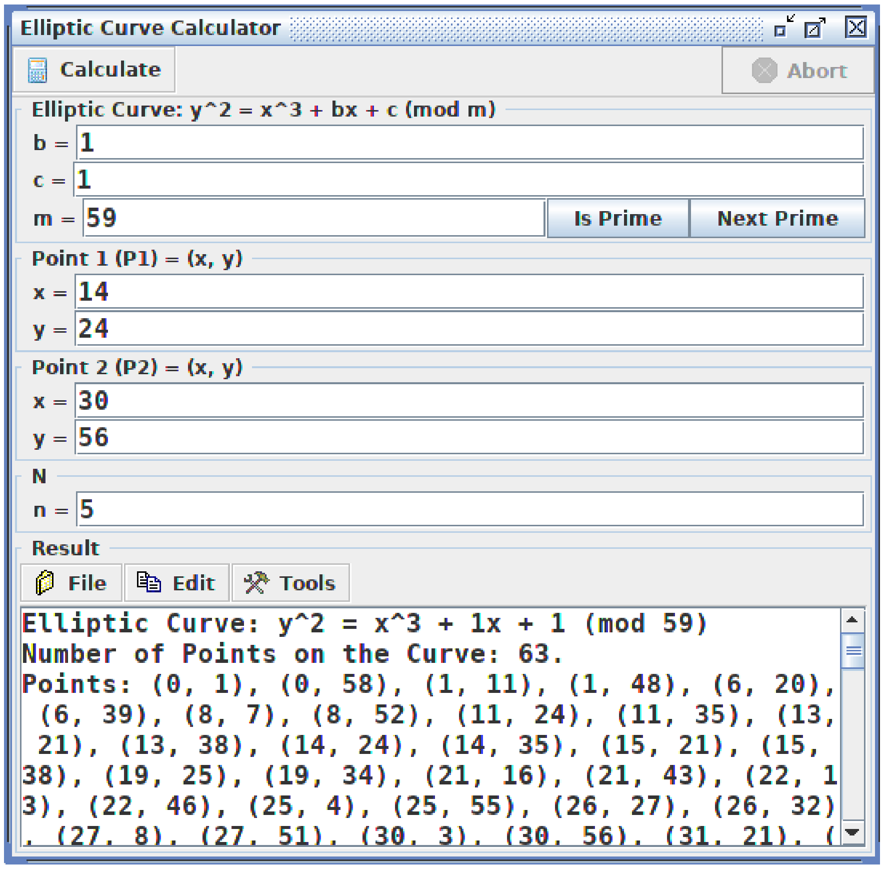
Notes

- 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.

C.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

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



Elliptic Curve: y^2 = x^3 + bx + c (mod m)
b = 1
c = 1
m = 59
Point 1 (P1) = (x, y)
x = 14
y = 24
Point 2 (P2) = (x, y)
x = 30
y = 56
N
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 C.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.

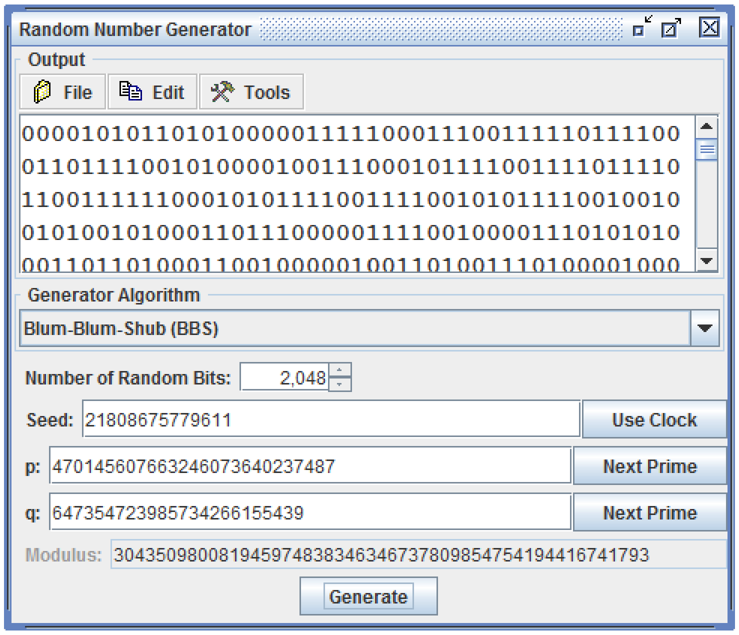
Generate all points on the elliptic curve: Returns a list of all the points on the elliptic curve. A list of Mathematica style points and Maxima style points are returned as well for loading into these computer algebra systems.

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.

C.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
11001111110001010111100111100101011110010010
01010010100011011100000111100100001110101010
00110110100011001000001001101001110100001000
```

Generator Algorithm: Blum-Blum-Shub (BBS)

Number of Random Bits: 2,048

Seed: 21808675779611

p: 470145607663246073640237487

q: 647354723985734266155439

Modulus: 304350980081945974838346346737809854754194416741793

Figure C.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.

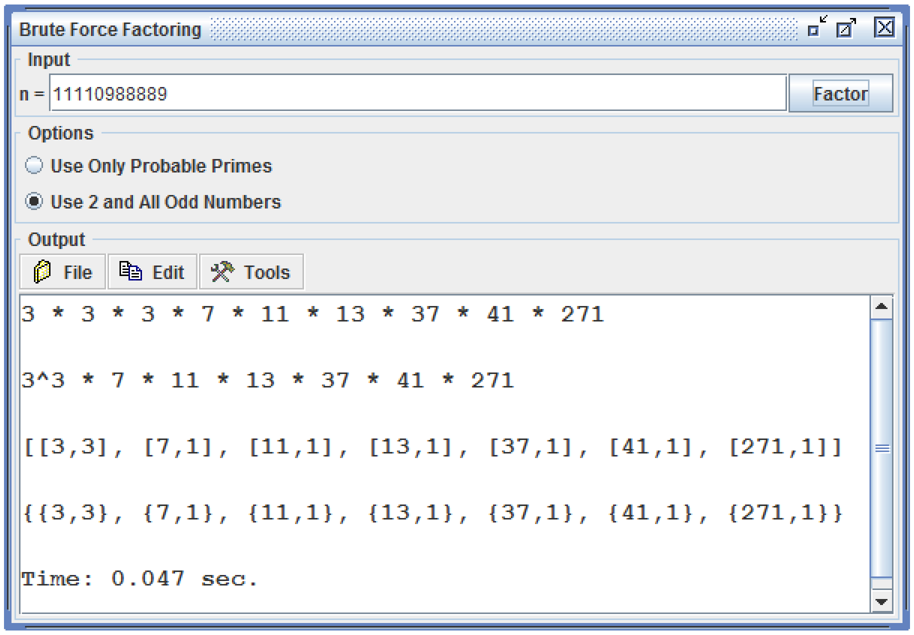
Java's Random Class: The built-in random number generator in Java (the language this program was written in) also uses a linear congruential algorithm but keeps a constant adder and multiplier, so you need only supply the seed and modulus. As with the linear congruential algorithm you can click on the 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.

C.7 Factoring Tools

C.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 C.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,

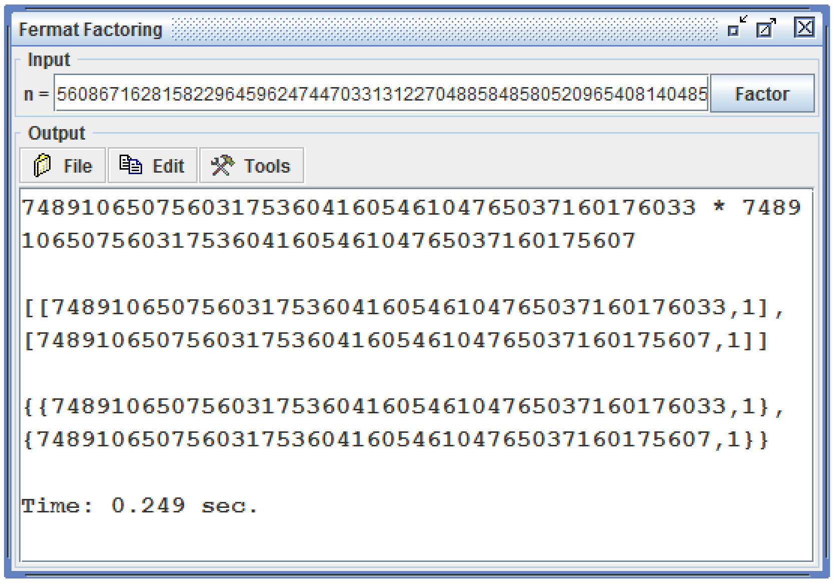
```
3 * 3 * 3 * 7 * 11 * 13 * 37 * 41 * 271
3^3 * 7 * 11 * 13 * 37 * 41 * 271
[[3,3], [7,1], [11,1], [13,1], [37,1], [41,1], [271,1]]
{{3,3}, {7,1}, {11,1}, {13,1}, {37,1}, {41,1}, {271,1}}
Time: 0.047 sec.
```

- 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.

C.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 C.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.

C.7.3 Pollard $p-1$ Factoring

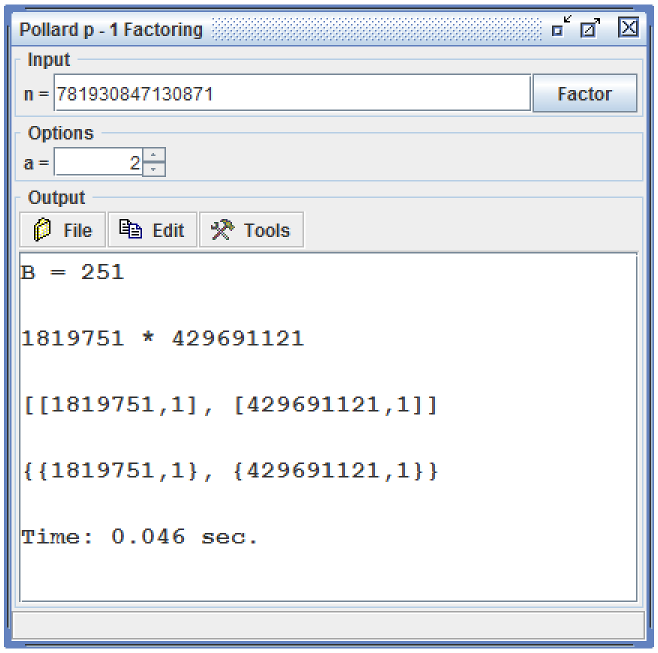
The Pollard $p-1$ Factoring algorithm simply computes $b = a^{B!} \pmod{n}$ and then finds $d = \gcd(b-1, n)$. If $1 < d < n$ then we have a non-trivial factor of n. This tool makes a slight alteration of the algorithm by iterating until a factorization if found or the user aborts the calculation. At 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.



n = 781930847130871

a = 2

```
B = 251

1819751 * 429691121

[[1819751,1], [429691121,1]]

{{1819751,1}, {429691121,1}}

Time: 0.046 sec.
```

Figure C.50: Pollard $p - 1$ Factoring Tool

Notes

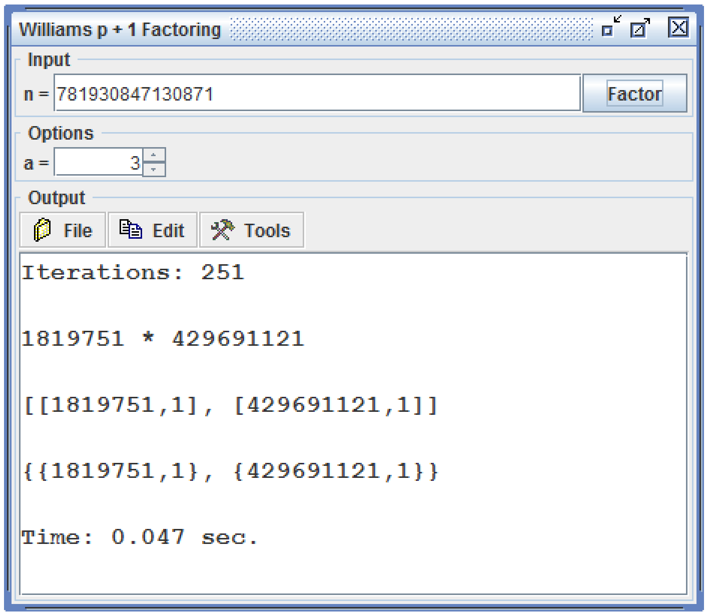
- The method does not find a complete prime factorization of n, even if it finds a factor. So the factors in the output could still be composite.
- The range for the base, a, is 2 to 1,000,000.
- The output of the factorization is in 3 forms. The first is as a product, the second is a list format equivalent 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.

C.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 C.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 $V_{251!}$.

C.7.5 Pollard Rho Factoring

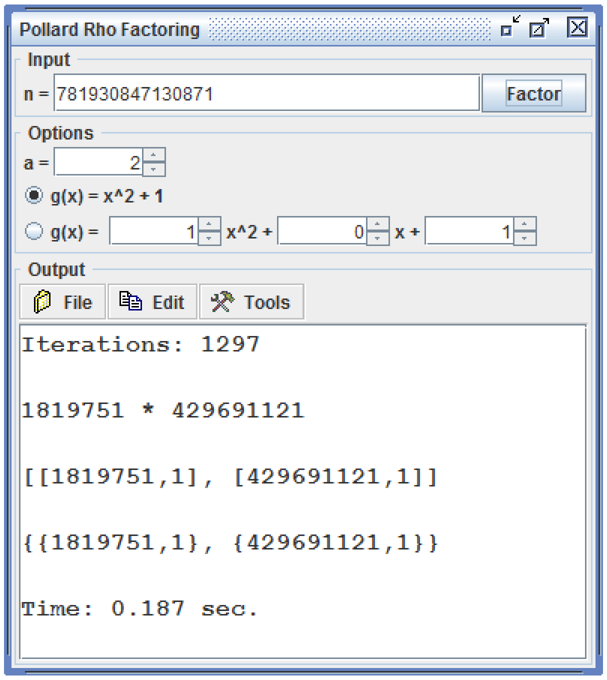
The Pollard Rho Factoring algorithm simply computes two sequences, $\{x_i\}$ and $\{y_i\}$ defined as $x_0 = y_0 = a$, $x_i = g(x_{i-1}))$ and $y_i = g(g(y_{i-1}))$, where $g(x) = x^2 + 1 \pmod{n}$ or $g(x) = a \cdot x^2 + b \cdot x + c \pmod{n}$, depending on your selection of options. For each i, we 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 C.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.

C.7.6 Brent's Method Factoring

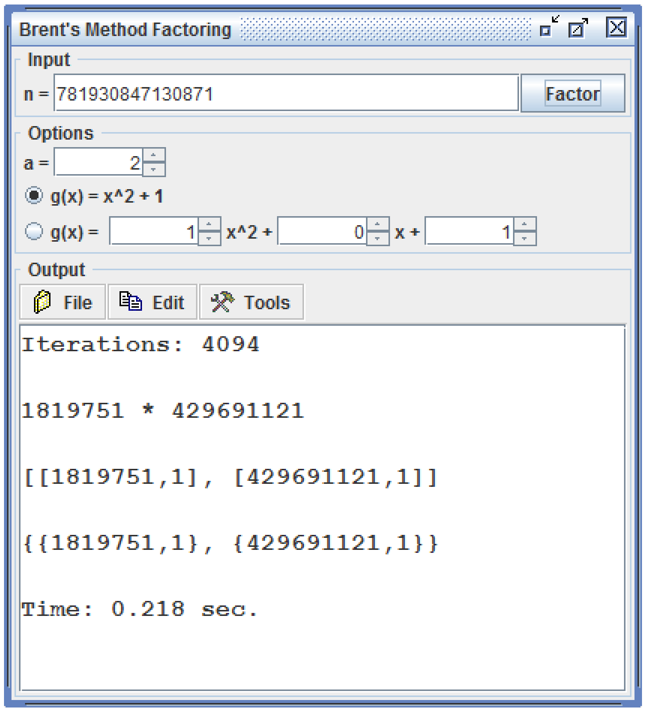
The Brent's Method Factoring algorithm simply computes the sequence, $\{x_i\}$ defined as $x_0 = a$, $x_i = g(x_{i-1})$, where $g(x) = x^2 + 1 \pmod{n}$ or $g(x) = a \cdot x^2 + b \cdot x + c \pmod{n}$, depending on your selection of options. For each i, we compute $d = \gcd(|x_i - x_j|, n)$, where j 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 C.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.

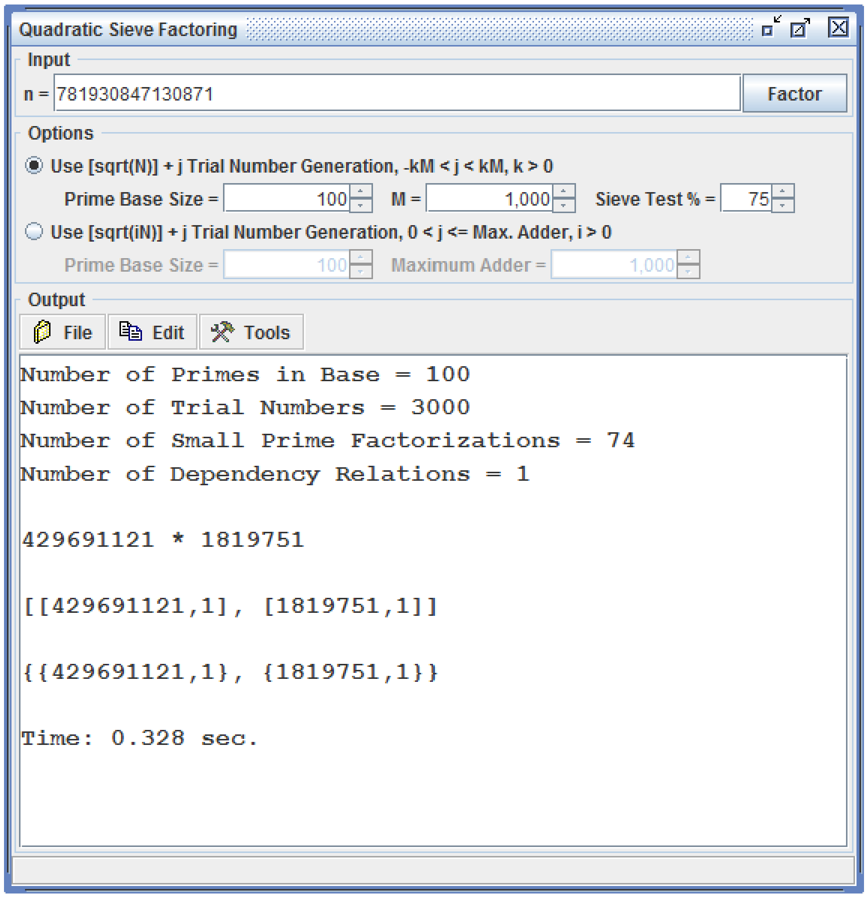
C.7.7 Quadratic Sieve Factoring

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 C.54: Quadratic Sieve Factoring Tool: Classical Sieve

How to Use the Tool

1. Input the number to be factored into the Input box.

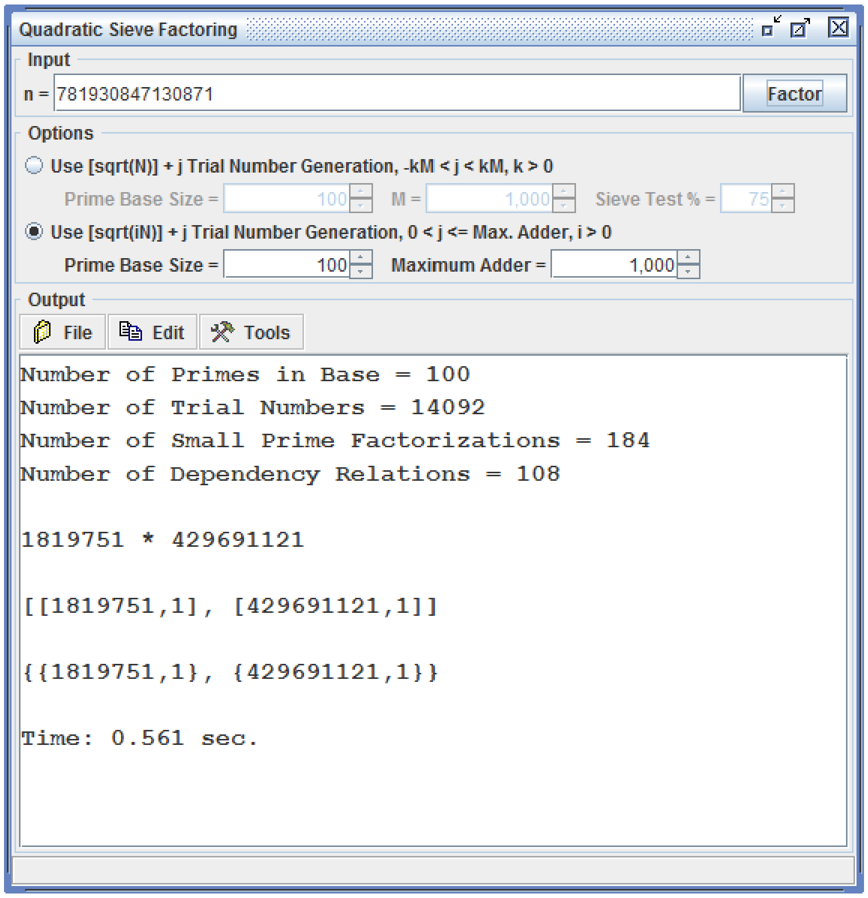
2. Select the mode of operation for the sieve. The `[sqrt(N)]+j` is the classical method and the `[sqrt(iN)]+j` is the method that skips the sieving step.

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

( ) 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 C.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.

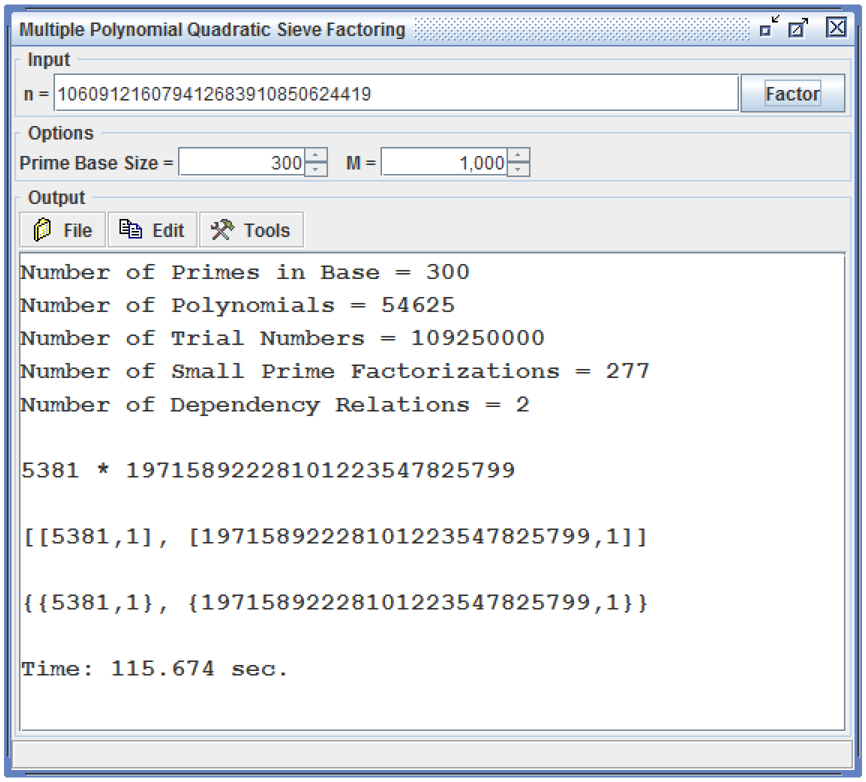
C.7.8 Multiple Polynomial Quadratic Sieve Factoring

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 C.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.

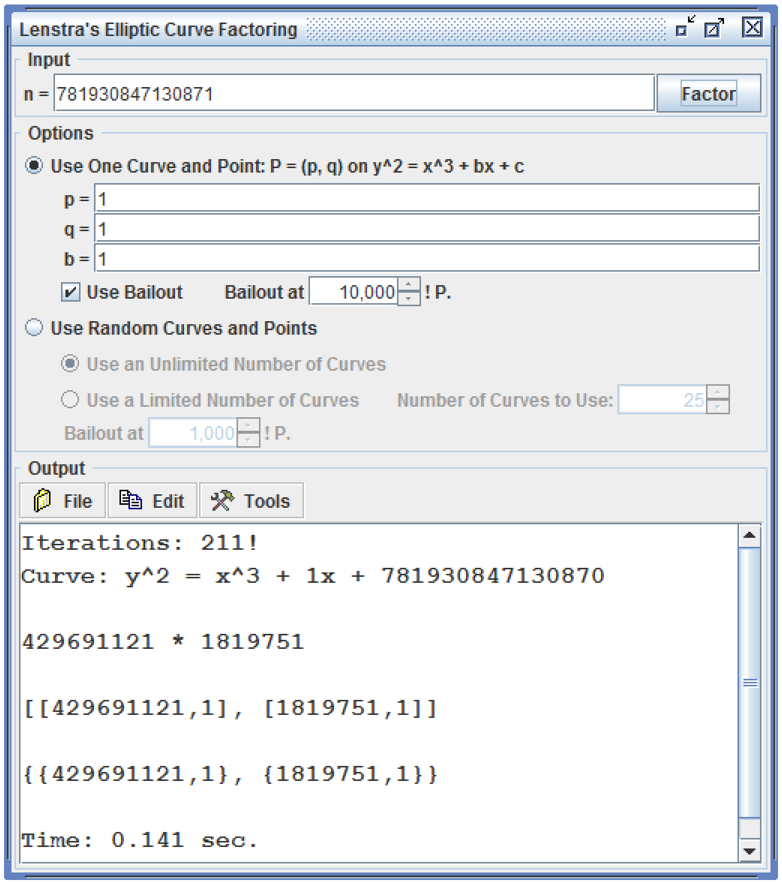
C.7.9 Lenstra's Elliptic Curve Factoring

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 C.57: Lenstra's Elliptic Curve Factoring Tool

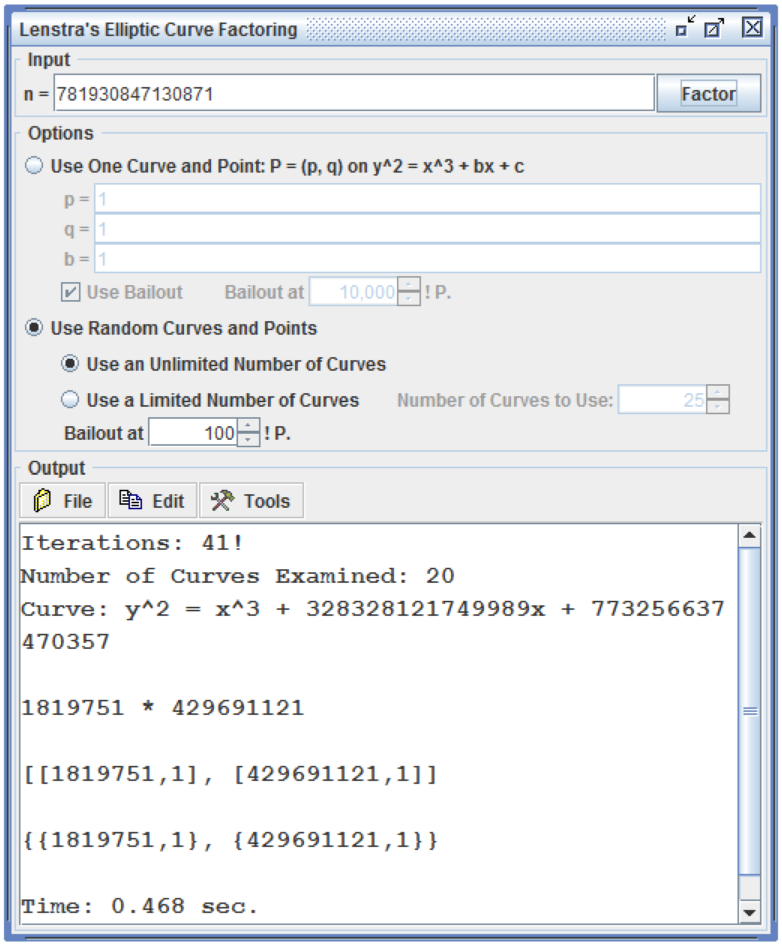
Options

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 C.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.

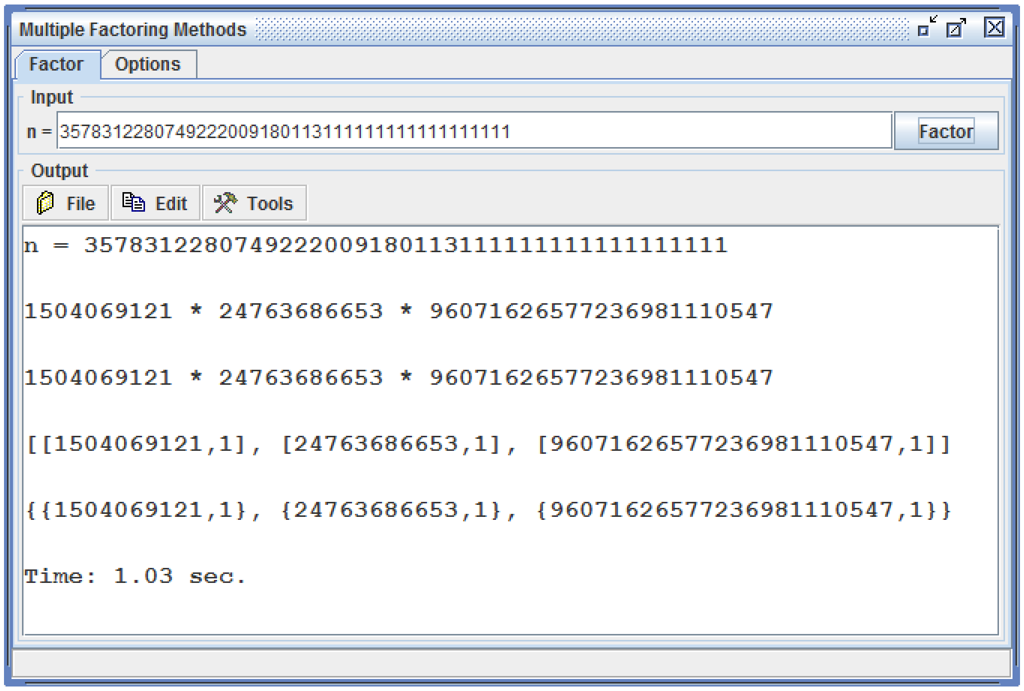
- The number of curves has a range of 1 to 1,000,000.
- The output of the factorization is in 3 forms. The first is as a product, the second is a list format equivalent to the Maxima output of the ifactors command, and the last is a list format equivalent to the Mathematica output of the FactorInteger 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, $2P$, $3(2P)$, $4(3!P)$, $5(4!P)$, $\ldots$, $m((m-1)!P)$.

C.7.10 Multiple Factoring Methods

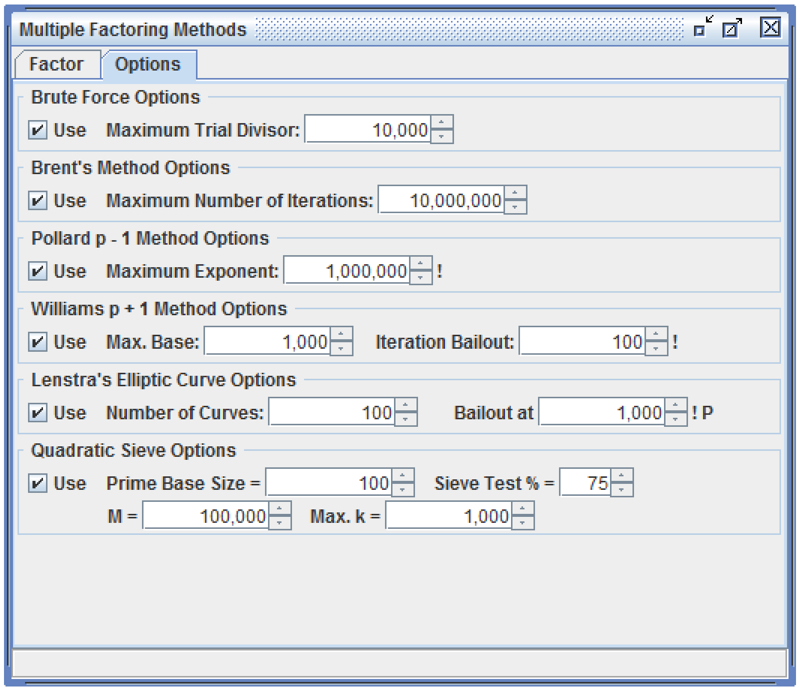
The Multiple Factoring Methods tool will factor an integer into its prime factor decomposition using a combination of various methods. The methods used can be found on the Options tab, they are applied in order of their listing. This tool allows the user to experiment with different factoring methods 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 = 357831228074922200918011311111111111111111

```
n = 357831228074922200918011311111111111111111

1504069121 * 24763686653 * 96071626577236981110547

1504069121 * 24763686653 * 96071626577236981110547

[[1504069121,1], [24763686653,1], [96071626577236981110547,1]]

{{1504069121,1}, {24763686653,1}, {96071626577236981110547,1}}

Time: 1.03 sec.
```

Figure C.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 C.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.

C.8 Discrete Logarithm Tools

C.8.1 Brute Force Discrete Logarithm

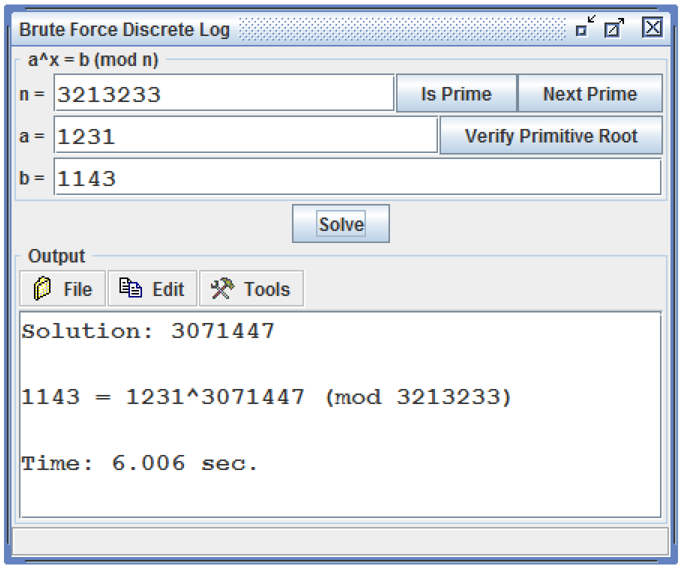
The Brute Force Discrete Logarithm tool will solve the discrete log problem of $b = a^x \pmod{n}$ given integers a, b, and n. The method simply tries all possible exponents x until a solution is found or it is determined that a solution does not exist.

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.



```
Solution: 3071447

1143 = 1231^3071447 (mod 3213233)

Time: 6.006 sec.
```

Figure C.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

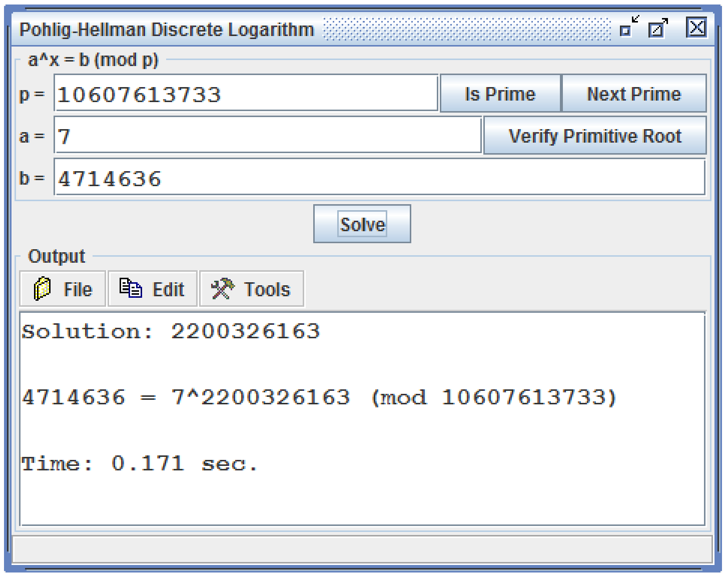
- Solving discrete log problems can be a lengthy process, the operation is done in a separate thread so that the user can use other facilities of the program while the operation is being completed.
- When the solving operation is invoked, the Solve button will be changed to an Abort button. To end the process simply click on the Abort button.

C.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 C.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
```

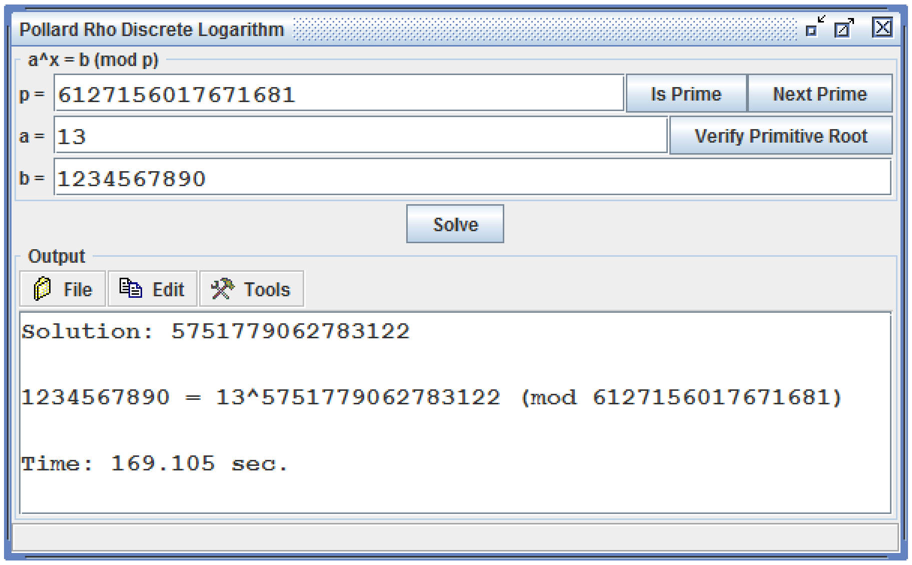
This means that there are 5 distinct primes in the factorization of $p-1$, and the program is currently working on the third prime, which is 121259. The Round indicates that in the factorization of $p-1$ there was a factor of 121259^3, so the program must run through 3 total rounds and it is currently 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.

C.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.



```
Solution: 5751779062783122

1234567890 = 13^5751779062783122 (mod 6127156017671681)

Time: 169.105 sec.
```

Figure C.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.

C.8.4 Pohlig-Hellman with Pollard Rho Discrete Logarithm

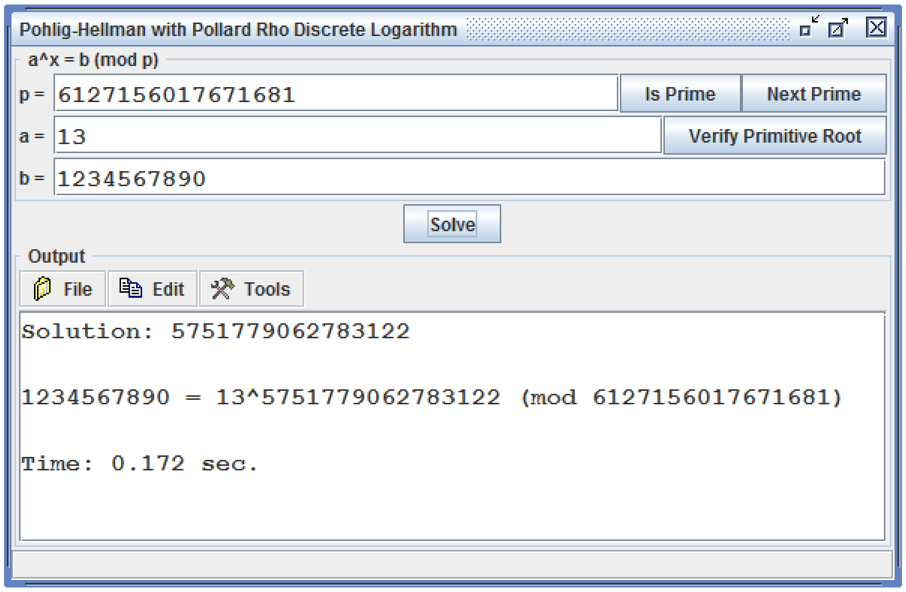
The Pohlig-Hellman with Pollard Rho Discrete Logarithm tool uses the Pohlig-Hellman algorithm to solve the discrete log problem of $b = a^x \pmod{p}$ given integers a, b, and p. In the classical Pohlig-Hellman algorithm a brute force algorithm is used to solve smaller discrete logarithm problems. In 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 C.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.

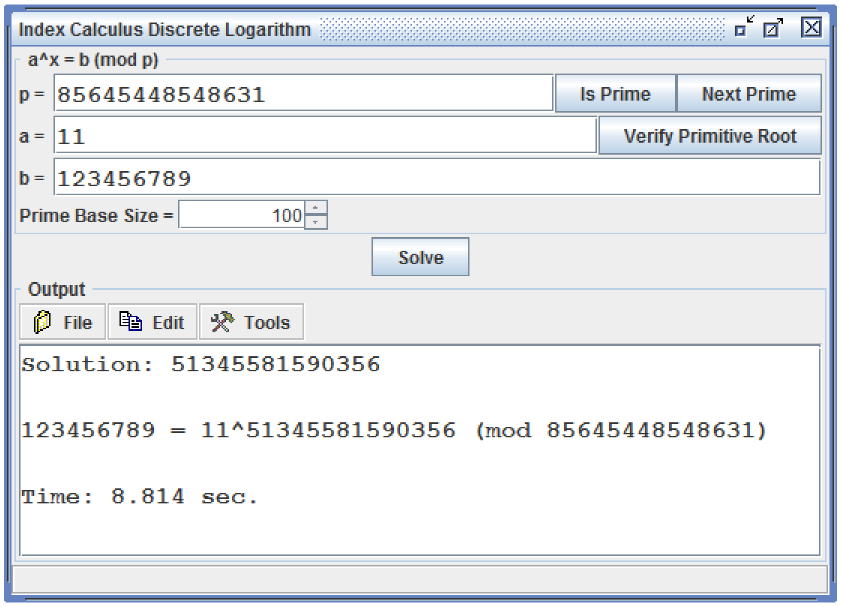
Notes

- 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.

C.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 C.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

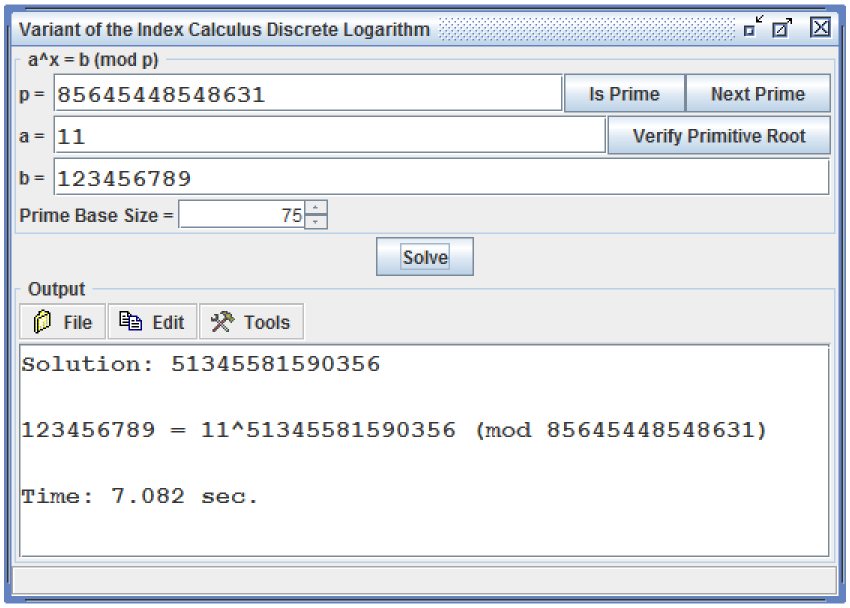
- Solving discrete log problems can be a lengthy process, the operation is done in a separate thread so that the user can use other facilities of the program while the operation is being completed.
- When the solving operation is invoked, the Solve button will be changed to an Abort button. To end the process simply click on the Abort button.

C.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 C.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.

Appendix D

Steganography

As we pointed out in the introduction, Steganography is the art of concealing a message, that is, its existence. In modern times this has become the practice of concealing a file, message, image, or video within another (seemingly harmless) file, message, image, or video.

Historically, cryptography and steganography were used hand in hand. It was much better for the health of the messenger if the enemy did not know that he was carrying a message. In this appendix we will give just a few examples of how steganography was used throughout history and discuss one modern technique, image steganography,

D.1 Some Examples from Antiquity

D.1.1 Wax Tablets

From as early 1400 BC through the middle ages, wax tablets were used as a portable and reusable writing surface. The wax tablet was usually a piece of wood that was covered with a layer of wax, the scribe would use a wood or metal stylus to mark the wax on the tablet that had a spatula shaped instrument on the other end to scrape the tablet wax like we would use an eraser. Since making marks in the wax was much harder and slower than writing with ink on parchment, early forms of shorthand were created to speed up the writing process. The tablets were easily reusable by heating up the tablet and melting the wax.

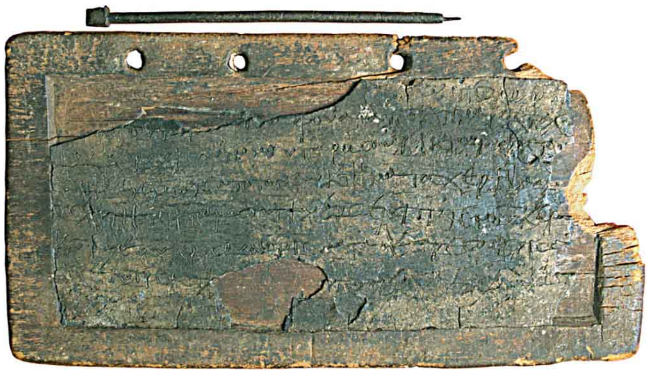


Figure D.1: A wooden wax tablet with bronze stylus and eraser, originating from Egypt circa 600 AD.

By around the middle of the 8th century BC the Greeks started using a folding pair of wax tablets. This gave them a nice way to conceal a message. The message was written on the inside of the two pieces of wood and then sealed together at the edges by the wax.

D.1.2 Scytale

As we saw in the chapter on classical cryptography, the scytale was an ancient tool for creating a transposition cipher. The way it worked is that the person who was encoding the message would wrap a strip of parchment or leather around a staff of a certain diameter and then write the message along the length of the staff, as pictured on the right. When the parchment or leather was unwound from the staff the letters would be seen lengthwise on the strip but in a nonsensical order. When the message was delivered the recipient would wind the message around a staff of the same diameter and then read the message.

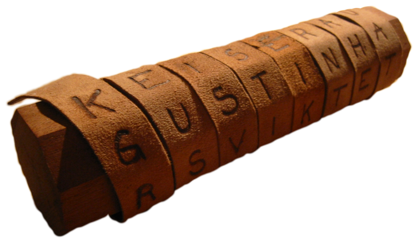


Figure D.2: Skytale

Where the steganography comes in was how the scytale was transferred from sender to receiver. If the messenger had carried a long strip of parchment with letters on it, it was a bit of a giveaway that it was a scytale. Furthermore the scytale was not very difficult to break if you knew how it worked. One way would be to simply try different staffs of different diameters until the message could be read, but the staff was not even necessary. The interceptor of the message just needed to line up the characters with different skips until they discovered the message. When the message was written on a strip of leather, the messenger would ware the leather strip as a belt with the lettering on the inside.

The scytale dates back to the 5^{th} century BC. One account of its use was in 404 BC, five messengers were sent from Persia to Sparta with a message to Lysander that Pharnabazus of Persia was planning to attack Sparta. Only one of the five messengers made it to Lysander of Sparta, bloody and battered, the messenger handed Lysander his belt, Lysander wrapped the belt around his scytale to read the warning. Lysander prepared for the attack and was successful in driving back Pharnabazus's army.[55]

D.1.3 Messages Written on the Messenger's Body

My favorite example of this comes from antiquity, although this technique has continued to be used in modern times.

Herodotus (484–425 BC) was a Greek historian who is now considered to be the *The Father of History.* He was the first historian known to collect his materials systematically and critically, and then to arrange them into a historiographic narrative [12]. Histiaeus (?–493 BC) was the son of Lysagoras, and is commonly known as the tyrant of Miletus.

In one of Herodotus's writings, he tells the story of a message tattooed on the shaved head of a slave of Histiaeus in 499 BC, hidden by the hair that afterwards grew over it, and exposed by shaving the head again. The message allegedly carried a warning to Greece about Persian invasion plans.

D.1.4 Messages on the Printed Page

Typeface Changes

In 1605, Sir. Francis Bacon created a method for hiding a message inside a "fake message" by changing the typeface between characters of the fake message to indicate a code for the concealed message. He was well ahead of his time in this idea, similar techniques are used today in image steganography which is discussed in the next section. Furthermore, his method is essentially to apply a binary coding scheme to the English alphabet, 450 years before the digital computer.

Figure D.3: Lead Type

Bacon (1561–1626), was an English jurist, philosopher, statesman, scientist, orator, and author. He held many high-ranking political positions during his lifetime but is probably most remembered, by the scientific community at least, as the father of empiricism.[73] He established and popularized specific inductive and deductive reasoning methodologies for scientific inquiry, not called the scientific method.

His steganographic technique is known as Bacon's cipher (or the Baconian cipher). It worked as follows, first create a secret message that you want to send, let's say, "Dear Alice". First convert the message into a binary code using the following scheme,

Letter	Code	Letter	Code	Letter	Code	Letter	Code
A	AAAAA	G	AABBA	N	ABBAA	T	BAABA
B	AAAAB	H	AABBB	O	ABBAB	U/V	BAABB
C	AAABA	I/J	ABAAA	P	ABBBA	W	BABAA
D	AAABB	K	ABAAB	Q	ABBBB	X	BABAB
E	AABAA	L	ABABA	R	BAAAA	Y	BABBA
F	AABAB	M	ABABB	S	BAAAB	Z	BABBB

So "Dear Alice" becomes,

AAABB AABAA AAAAA BAAAA AAAAA ABABA ABAAA AAABA AABAA

and removing the spaces we get,

AAABBAABAAAAAAABAAAAAAAAAABABAABAAAAAAABAAABAA

Now we create a fake message that has as many characters as the A and B string above, if we get an extra letter or two, that is fine as well since they can easily be ignored. We will take one of Francis Bacon's quotes.

Beauty itself is but the sensible image of the infinite.

There are 45 characters in the A and B string and there are 46 non-space characters in the quote, so this will work well. Remove the spaces in the quote and place it directly under the A and B string, as below.

```
AAABBAABAAAAAAAABAAAAAAAAAAABABAABAAAAAAABAAAABAA
BEAUTYITSELFISBUTTHESENSIBLEIMAGEOFTHEINFINITE
```

Now we choose two different typefaces, one that will be used for all the A positions and one for all of the B positions. We will use a plain typeface for the A's and italics for the B's. Now every letter below an A will be typed in a plain font and every letter below a B will be in italics. So our message will be,

Bea*ut*y i*t*self is b*u*t the sensib*l*e *i*ma*g*e of the *i*nfi*n*ite.

The decryption process is simply done in reverse. We would take the message, remove spaces and create our string of A's and B's by the typeface. This gives the following string, when we block it into groups of 5 we have an extra character, that is ignored.

```
AAABBAABAAAAAAAABAAAAAAAAAAABABAABAAAAAAABAAAABAAA
AAABB AABAA AAAAA BAAAA AAAAA ABABA ABAAA AAABA AABAA A
```

Finally, we decode the string to, Dear Alice.

If we were to see some text like "Bea*ut*y i*t*self is b*u*t the sensib*l*e *i*ma*g*e of the *i*nfi*n*ite." we would become very suspicious that something was going on. It is not common, by today's standards, to do something this strange to the fonts in a written passage of text. But in 1600, the printing press was still a relatively young invention (c. 1440) and it was common in longer printing jobs of the time to run out of some letters of the lead type that was used in the press. So typesetters frequently had to substitute different typefaces or even different sizes of a font to complete a print job. Hence to a 16^{th} or 17^{th} century English reader, the multiple font typefaces would not be as out of the ordinary as it would be today.

Newspaper Code

During the Victorian era, newspapers could be sent through the mail without any postal charge, the poorer classes took advantage of this by sending their letters to each other encoded into the newspaper, this then became what is known as the *newspaper code.* The way this worked is the sender would poke small pinholes just above the letters of the newspaper to spell out their message. When the letters below the dots were transferred and written together the message would be revealed.

Newspaper codes were used again during World War II and into the Cold War, although during this time the pinholes were replaced by either secret ink markings or invisible ink, which made the codes much harder to detect. Newspaper codes in the twentieth century were not very effective, the problem was speed. Newspapers were sent as third-class mail, which often took too long for the message to arrive to the receiver. In addition, the man-hours required by American censors quickly made checking every newspaper clipping impractical, and eventually all newspapers were banned from entering the country.[29]

Type Spacing and Offsetting

Type spacing or type offsetting is a way of subtly distorting the text in a message to hide additional data. Type spacing was created as a way to discourage illegal copying of textual material. While this makes its intended purpose as a form of watermark, type spacing can also be used to send a message in secret. To encode a secret message using type spacing all one would have to do is adjust specific letters ever so slightly from their normal position. The letters that are out of position indicate the secret message. [29]

For example, say we wanted to send the message HELP. If we convert the letters to numbers between 1 and 26 the message is coded as 8 5 12 16. We could then put one space between between the words to indicate a count and then two spaces between the words to indicate that we are moving onto the next character, we ignore the changes in lines. Doing so with the above paragraph gives the following.

```
Type␣spacing␣or␣type␣offsetting␣is␣a␣way␣of␣␣subtly␣distorting␣the
text␣in␣a␣message␣␣to␣hide␣additional␣data.␣Type␣spacing␣was
created␣as␣a␣way␣to␣discourage␣illegal␣␣copying␣of␣textual
material.␣While␣this␣makes␣its␣intended␣purpose␣as␣a␣form␣of
watermark,␣type␣spacing␣can␣also␣␣be␣used␣to␣send␣a␣message␣in
secret.
```

D.2 The 20^{th} Century

D.2.1 Invisible Ink

We have probably all done this as kids, write a message in lemon juice on a piece of paper and let it dry. The receiver then simply heats the paper to reveal the message. Many other liquids will do the same thing. Even though this tends to be a child's game, invisible inks were commonly used through the middle ages and are still used in the world of espionage today.

1. In 1999, the U.S. Central Intelligence Agency successfully requested that World War I invisible ink remained exempt from mandatory declassification, based on the claim that invisible ink was still relevant to national security.[49] The World War I documents remained classified until 2011.[34][74]

2. Former MI-6 agent Richard Tomlinson alleges that Pentel Rolling Writer rollerball pens were extensively used by MI-6 agents to produce secret writing while on missions.[60]

3. In 2002, a gang was indicted for spreading a riot between federal penitentiaries using coded telephone messages, and messages in invisible ink.[48]

4. In 2008, a British Muslim, Rangzieb Ahmed, was alleged to have a contact book with Al-Qaeda telephone numbers, written in invisible ink.[47]

D.2.2 The Microdot

The Microdot is a photographic image that is substantially reduced in size, like the old library microfilm. The reduced images could be photographs or photographic images of text or important documents. Although a microdot could be fashioned into any shape, they were usually circular so that they could be pasted to a telegram or letter over a period, or the dot in a lowercase i or j. Microdots were hard to detect and often went unnoticed by inspectors. When the message was received, the recipient could use a microscope to read the contents of the dot.

Microdots were used in Germany between World War I and World War II and later by many countries to pass messages through insecure postal channels.

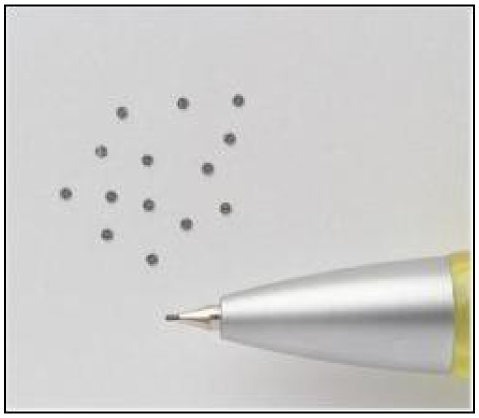


Figure D.4: Microdot

D.3 Image Steganography

In Image Steganography, one embeds an image, text or some other type of file into an image. In the example below, the picture of a nuclear submarine was embedded into the image of the grumpy cat who is on a diet. The embedding is done by slightly altering the colors of the cat image and is done in a way that does not produce any type of ghost images of the submarine in the cat picture. In the same way, we can hide textual information in the cat image or a word processor or PDF file, even an executable program.

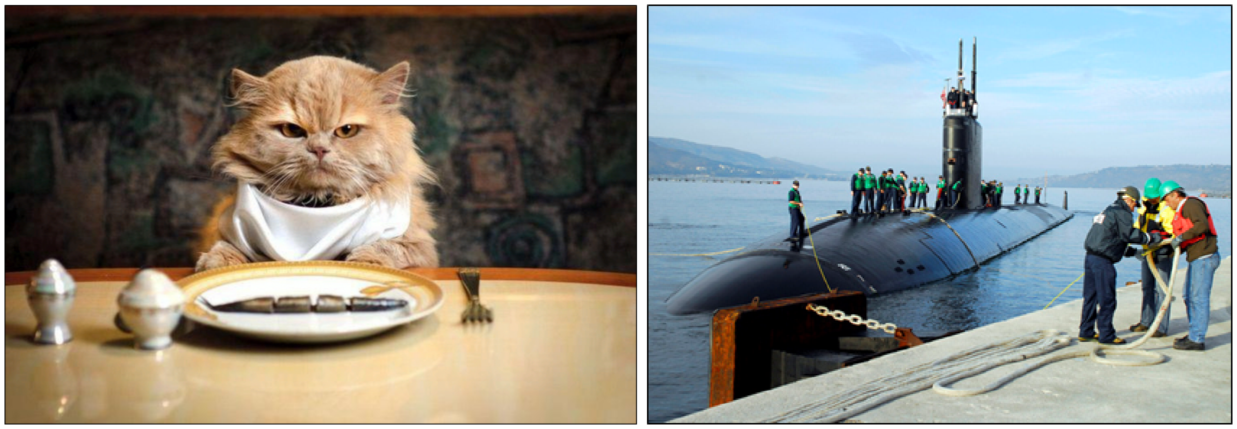


Figure D.5: Image Steganography

There are many ways that you can embed a file into an image, the one we will be discussing here is one of several *Least Significant Bit* algorithms. Throughout this discussion we will call the image that will store the hidden information simply the image (that is, the cat picture) and the information to be stored will be called the file (here it is the picture of the submarine, but can be any type of file).

Computer files are simply a string of bits, (1's and 0's). So the information file we wish to store can be easily converted into a string of 0's and 1's. Hence, all we need to do is come up with a way to store a string of 0's and 1's into an image. Before we get too deep into this, lets first discuss how images are stored in a computer. There are actually many different ways this can be done but several file formats, such as bitmap files BMP and portable network graphics PNG files, use a raster of colored dots called pixels, which stands for picture elements. As you can see from the zoom in of the image on the right, the pixels are simply colored dots arranged in a series of rows and columns. In the original image, these pixels are very small. So if we change their color, but only slightly, the observer of the image will not be able to tell that the image was altered. Furthermore, these pixels are so close together the human eye will blend the colors of adjacent pixels together, making it even harder to notice any alterations in the image.

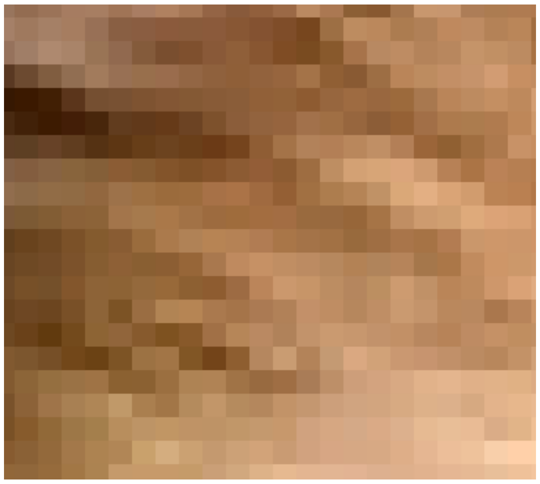


Figure D.6: Zoom in of a portion of the cat image.

So now the question is, how do we create the colors of a single pixel so that when we put them into this raster we get a very grumpy cat? Again, there are many ways to do this but the most common is to designate the amount of red, green, and blue color that will make the image. If you have ever looked at additive and subtractive colors this will be familiar to you. Subtractive colors are commonly used in color film photography when printing and enlarging photographs from color film negatives. These work by taking a bright white light and then by placing subtractive color filters in front of the light you remove certain visible wavelengths of light. If you use all of the filters you will eventually remove all of the wavelengths and be left with black. So in a nutshell, subtractive colors take all light and subtract off color. The most commonly used subtractive primary colors are cyan, magenta, and yellow.

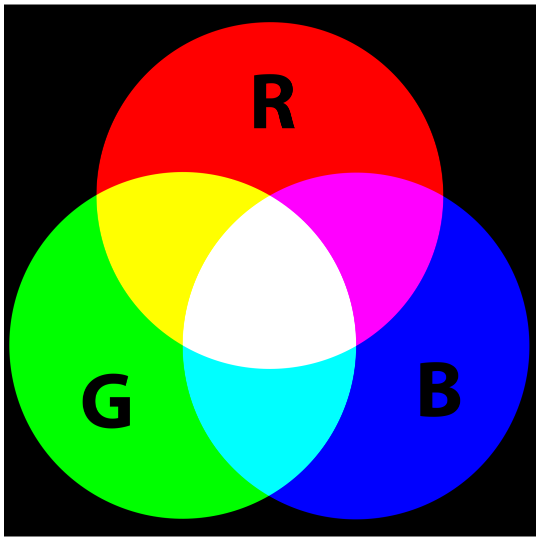


Figure D.7: Primary Additive Colors

A computer screen works the opposite way. We start with black and we add wavelengths of primary colors to create the color we want. If we use all of the primary colors at full intensity we have white light. The most commonly used additive primary colors are red, green, and blue. So the color of each pixel in the image is a combination of red, green, and blue, which we call RGB. Mathematically, we can think of this RGB designation as a vector or three-tuple, (r, g, b).

The next question is, what are the values of r, g, and b? Physically, these are intensities of light, so they would be continuous values between 0 (no light) and 1 (full intensity). There are an infinite number of values between 0 and 1 so, of course, a computer cannot store all the possibilities, so a different method was devised. As you have probably guessed, there are several ways to do this. The most common is to let r, g, and b be integers between 0 and 255. This came from the fact that a byte (8 bits) can store all integer values from 0 to 255, 0 being 00000000 and 255 being 11111111. So each pixel will take 3 bytes (or 24 bits) of information. In all, there are 16,777,216 different colors that any pixel on the screen can be. In the image to the right, we have put in (r, g, b) values for two of the pixels. If we take the pixel with RGB value $(248, 207, 170)$ and rewrite it in binary we have $(11111000, 11001111, 10101010)$. The least significant bit is the one that, if changed, alters the number the least amount, in our case this is the furthest bit to the right. If we change this bit from a 0 to 1 or a 1 to 0 it will change the number by at most 1. So 11111001 is 249, 11001110 is 206, and 10101011 is 171. If we place both of the colors $(248, 207, 170)$ and $(249, 206, 171)$ side by side we get the following figure. As you can see from the figure, there is very little difference between these two colors, and when these boxes are reduced to the size of a single pixel the difference is indistinguishable with the human eye. What this is telling us is that we can change that least significant bit on every pixel in the image, at will, and the result on the image will not be evident to the observer.

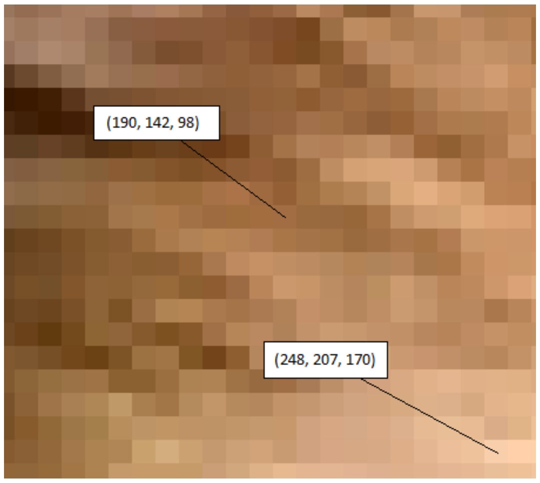


Figure D.8: (r, g, b) Color Values

Figure D.9: Colors $(248, 207, 170)$ and $(249, 206, 171)$

This is exactly how we are going to store that string of 0's and 1's from the file. We simply replace the last bit of each of the RGB numbers for each pixel with the next 0 or 1 in the binary stream of the file. Specifically, after we convert the file to our binary stream we break the stream of 0's and 1's into blocks of three, the first bit becomes the least significant bit of red for the first pixel, the second bit becomes the least significant bit of green for the first pixel, and the third bit becomes the least significant bit of blue for the first pixel. Then we take the next three bits and the second pixel and do the same thing, the next three bits with the third pixel, and so on until all of the bits from the file are stored.

For example, say that when we convert our file to binary we get the stream

```
1001010011100111010011010101001010 10...
```

Break the stream into blocks of three bits to get,

```
100 101 001 110 011 101 001 101 010 100 101 010...
```

Now do the conversion,

Table D.2: Color Conversion Example (Encoding)

Pixel	RGB	RGB Binary	File Bits	New RGB Binary	New RGB
1	(123, 61, 52)	(01111011, 00111101, 00110100)	100	(01111011, 00111100, 00110100)	(123, 60, 52)
2	(57, 127, 142)	(00111001, 01111111, 10001110)	101	(00111001, 01111110, 10001111)	(57, 126, 143)
3	(222, 91, 139)	(11011110, 01011011, 10001011)	001	(11011110, 01011010, 10001011)	(222, 90, 139)
4	(84, 161, 252)	(01010100, 10100001, 11111100)	110	(01010101, 10100001, 11111100)	(85, 161, 252)
5	(173, 201, 146)	(10101101, 11001001, 10010010)	011	(10101100, 11001001, 10010011)	(172, 201, 147)
⋮	⋮	⋮	⋮	⋮	⋮

Now you would recolor each pixel with the New RGB values and save the altered image. The receiver of the image would then take the RGB values of the pixels, convert them to binary and extract the least significant bit. Then take the binary stream and create a file from the binary stream. From our example, the receiver would do the following calculation,

Table D.3: Color Conversion Example (Decoding)

Pixel	RGB	RGB Binary	File Bits
1	(123, 60, 52)	(01111011, 00111100, 00110100)	100
2	(57, 126, 143)	(00111001, 01111110, 10001111)	101
3	(222, 90, 139)	(11011110, 01011010, 10001011)	001
4	(85, 161, 252)	(01010101, 10100001, 11111100)	110
5	(172, 201, 147)	(10101100, 11001001, 10010011)	011
⋮	⋮	⋮	⋮

Giving the same binary stream that we started with and hence giving us the same file. We should note couple things to note about this process.

1. There is really no reason to convert the RGB values to binary, this was just done here to illustrate the storage of the bits. All we need to do is see if the RGB value is even or odd and alter it if it does not match. For example, in pixel 1, the green value is odd

Table D.4: Color Conversion Chart

RGB Value	Bit	Do
Even	0	Nothing
Even	1	Add 1
Odd	0	Subtract 1
Odd	1	Nothing

and the bit was 0 so all we needed to do was subtract 1 from 61 to get the new value. Also, in pixel 4 the red value was even and the bit was one so all we needed to do was add one to 84 to get 85. In general,

In the decoding process all we needed to do is determine if the RGB value was even or odd. If it was even we add a 0 to the binary stream and if it was odd we add 1 to the binary stream.

2. In the encoding chart you will notice that not all RGB values were changed. Probabilistically speaking we would only suspect that half of the values would change. This means that, in practice, the colors are closer to each other then our original worst case scenario, that we assumed in the above discussion.

3. In practice, there is other information we need to store as well. Such as, the size of the file in bits, so we know where to stop extracting bits. We would usually also store the file type, doc, PDF, exe, bmp, png, and so on.

4. Since the image is storing only three bits of the file per pixel the image must be fairly large. For a quick example, say we are storing a 1 MB file (which is a fairly small), this is 1,048,576 bytes, and hence 8,388,608 bits. Since each pixel can store only 3 bits we must have at least 2,796,203 pixels in the image. This is an image that is about 1920×1500 pixels, most wide-screen computer monitors made today have a resolution of 1920×1200 pixels.

5. There are many freeware programs out there that will automatically do the encoding and decoding on an image and usually offer other coding schemes. One other scheme that can store twice as much information is to use the least two significant bits instead of just one. This will alter the colors a bit more but still not be noticeable to the human eye.

6. Some image file formats are not good for steganographic purposes. For example, JPEG images use lossy compression schemes to save file size, in the process they will alter groups of colors and thus change the RGB values of some pixels. Consequently, this will change the least significant bits of the RGB values and the stored file. Image file formats must be lossless types, such as bitmaps and portable network graphics files.

7. Using this method you can also "sign" an image. That is, put some retrievable information in the image that can be used to identify that an image copyright is yours.

Appendix E

A Quick Reminder on Modular Arithmetic

As we will see shortly, the Hill Cipher is a linear algebra technique but it relies on modular arithmetic. So we will give a quick reminder on modular calculations. Throughout the discussion we will let n be the modulus, so n will be an integer and $n \geq 2$.

E.1 Definition

Definition 26: *Let m and k be two integers, then we say that $m \equiv k \pmod{n}$, if $n|(m-k)$, that is, n divides the quantity $m-k$ evenly.*

Example 90: A few examples of $m \equiv k \pmod{n}$,

1. $12 \equiv 2 \pmod{5}$ since $\frac{12-2}{5} = 2$.
2. $156 \equiv 3 \pmod{17}$ since $\frac{156-3}{17} = 9$.
3. $-24 \equiv 10 \pmod{17}$ since $\frac{-24-10}{17} = -2$.

Δ

The next definition is really more of a convention than it is a definition.

Definition 27: *Let k be an integer, when we write $k \pmod{n}$, we mean the number m that is in the range $0 \leq m < n$ such that $m \equiv k \pmod{n}$.*

Example 91: A few examples of $k \pmod{n}$,

1. $12 \pmod{5} \equiv 2$
2. $156 \pmod{17} \equiv 3$
3. $-24 \pmod{17} \equiv 10$

Δ

E.2 Arithmetic

Addition, subtraction, and multiplication modulo a number is fairly straight forward. Simply do the addition, subtraction, or multiplication as usual then take the result mod the number.

Example 92: A few examples of modular arithmetic,

1. $34 + 12 \pmod 5 \equiv 46 \pmod 5 \equiv 1$
2. $15 - 23 \pmod{17} \equiv -8 \pmod{17} \equiv 9$
3. $24 \cdot 13 \pmod{17} \equiv 312 \pmod{17} \equiv 6$

Δ

When it comes to division we need to be a bit more careful. We also need to consider what division is. When we write the fraction $\frac{2}{3}$ we really mean 2 times the "inverse" of 3, that is, $2 \cdot \frac{1}{3}$. And what is the "inverse" of 3? Well when we are doing arithmetic using real numbers we represent the "inverse" of 3 as $\frac{1}{3}$ so that, with our rules of algebra, when we multiply 3 by its inverse we get 1, that is, $3 \cdot \frac{1}{3} = 1$. Why do we want the result to be 1? Answer, since 1 is the multiplicative identity in the real number system, that is $1 \cdot a = a$ for all real numbers a. The general concept of an inverse, specifically a multiplicative inverse, is the same. To invert a number a in a particular number system we want another b in that same system such that when we multiply a and b together we get the multiplicative identity in that system.

Definition 28: *Let k be an integer, then $k^{-1} \pmod n$ is the number m that is in the range $0 \leq m < n$ such that $m \cdot k \pmod n \equiv 1$, if m exists. If no such number m exists then we say that k does not have an inverse modulo n or that k is not invertible mod n.*

Example 93: A few examples of $k^{-1} \pmod n$,

1. $2^{-1} \pmod 5 \equiv 3$ since $2 \cdot 3 \pmod 5 \equiv 6 \pmod 5 \equiv 1$
2. $12^{-1} \pmod{17} \equiv 10$ since $12 \cdot 10 \pmod{17} \equiv 120 \pmod{17} \equiv 1$
3. $11^{-1} \pmod{26} \equiv 19$ since $11 \cdot 19 \pmod{26} \equiv 209 \pmod{26} \equiv 1$
4. $4^{-1} \pmod{26}$ does not exist since there is no number b between 0 and 25 with $4 \cdot b \pmod{26} \equiv 1$

Δ

One obvious question at this point would be, when does a number have an inverse and when does it not have an inverse mod n? For the real number system this is an easy question since we know that 0 is the only number you cannot invert, but the question is a little more difficult when working modulo n. The next theorem will give us the answer, we will not prove this result, you can find a proof in any discrete mathematics text of elementary number theory text.

Theorem 6: *Let k be an integer, then $k^{-1} \pmod{n}$ exists if and only if the greatest common divisor of k and n is 1, that is,* $\gcd(k,n) = 1$. *Another way to say this is that k and n are relatively prime.*

From our last example,

Example 94:

1. $2^{-1} \pmod{5}$ exists since $\gcd(2,5) = 1$
2. $12^{-1} \pmod{17}$ exists since $\gcd(12,17) = 1$
3. $11^{-1} \pmod{26}$ exists since $\gcd(11,26) = 1$
4. $4^{-1} \pmod{26}$ does not exist since $\gcd(4,26) = 2 \neq 1$

Δ

Now that we know what is necessary and sufficient for a number to have an inverse modulo n the next question is if an inverse exists, how do you find it? There is an algorithm for doing this, it is the Extended Euclidean Algorithm. The process is not difficult but it is a bit tedious and does not add anything to our discussion. One can find detailed discussions of the algorithm and why it works in any elementary number theory textbook. In the section on technology we will go over the commands to find modular inverses using a number of different computer algebra systems.

E.3 Matrices

Our next goal is to generalize the above modular calculations to matrices. Taking a matrix that has only integer entries mod n is simple, we just take each entry mod n. For example,

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \pmod{5} \equiv \begin{bmatrix} 1 & 4 & 2 \\ 2 & 0 & 3 \\ 3 & 1 & 4 \end{bmatrix}$$

Addition, subtraction, and multiplication mod n is just as easy. First do the addition, subtraction, or multiplication as you would normally and then take the result mod n, that is, take each entry mod n. For example,

Example 95:

1.

$$\begin{aligned} \begin{bmatrix} 2 & 1 & -1 \\ 3 & 5 & 2 \end{bmatrix} + \begin{bmatrix} 2 & -1 & 5 \\ -4 & -7 & 9 \end{bmatrix} \pmod{5} &\equiv \begin{bmatrix} 4 & 0 & 4 \\ -1 & -2 & 11 \end{bmatrix} \pmod{5} \\ &\equiv \begin{bmatrix} 4 & 0 & 4 \\ 4 & 3 & 1 \end{bmatrix} \pmod{5} \end{aligned}$$

2.

$$\begin{aligned}\begin{bmatrix} 2 & 1 & -1 \\ 3 & 5 & 2 \end{bmatrix} - \begin{bmatrix} 2 & -1 & 5 \\ -4 & -7 & 9 \end{bmatrix} \pmod 5 &\equiv \begin{bmatrix} 0 & 2 & -6 \\ 7 & 12 & -7 \end{bmatrix} \pmod 5 \\ &\equiv \begin{bmatrix} 0 & 2 & 4 \\ 2 & 2 & 3 \end{bmatrix} \pmod 5\end{aligned}$$

3.

$$\begin{aligned}\begin{bmatrix} 2 & 1 & -1 \\ 3 & 5 & 2 \end{bmatrix} \begin{bmatrix} 2 & -4 \\ -1 & -7 \\ 5 & 9 \end{bmatrix} \pmod 5 &\equiv \begin{bmatrix} -2 & -24 \\ 11 & -29 \end{bmatrix} \pmod 5 \\ &\equiv \begin{bmatrix} 3 & 1 \\ 1 & 1 \end{bmatrix} \pmod 5\end{aligned}$$

Δ

As you probably expected, matrix inverses are a little more difficult. When we compute a matrix inverse by hand we usually use the reduction algorithm where we place the matrix beside the identity, reduce, and then extract what the identity had become. For example, say we wanted to find the inverse of

$$\begin{bmatrix} 2 & 1 & -1 \\ 3 & 5 & 2 \\ -4 & -7 & 9 \end{bmatrix}$$

First we set up the matrix,

$$\begin{bmatrix} 2 & 1 & -1 & 1 & 0 & 0 \\ 3 & 5 & 2 & 0 & 1 & 0 \\ -4 & -7 & 9 & 0 & 0 & 1 \end{bmatrix}$$

Reduce this matrix,

$$\begin{bmatrix} 1 & 0 & 0 & \frac{59}{84} & -\frac{1}{42} & \frac{1}{12} \\ 0 & 1 & 0 & -\frac{5}{12} & \frac{1}{6} & -\frac{1}{12} \\ 0 & 0 & 1 & -\frac{1}{84} & \frac{5}{42} & \frac{1}{12} \end{bmatrix}$$

Since the left thee columns are the 3×3 identity we know that the original matrix is invertible and its inverse are the last three columns. We then extract the last three columns to obtain the inverse,

$$\begin{bmatrix} \frac{59}{84} & -\frac{1}{42} & \frac{1}{12} \\ -\frac{5}{12} & \frac{1}{6} & -\frac{1}{12} \\ -\frac{1}{84} & \frac{5}{42} & \frac{1}{12} \end{bmatrix}$$

Although we did not show all of the steps in the reduction, it is clear by the result that at some stage we needed to divide rows by a number. When we are dealing with modular arithmetic this division must be thought of, and executed as, the multiplication of inverses mod n. Say we wanted to take the inverse of this matrix mod 5. First we reduce the matrix mod 5 to obtain,

$$\begin{bmatrix} 2 & 1 & 4 \\ 3 & 0 & 2 \\ 1 & 3 & 4 \end{bmatrix}$$

Append the identity,

$$\begin{bmatrix} 2 & 1 & 4 & 1 & 0 & 0 \\ 3 & 0 & 2 & 0 & 1 & 0 \\ 1 & 3 & 4 & 0 & 0 & 1 \end{bmatrix}$$

Now we will reduce the matrix to reduced echelon form. During this derivation all arithmetic will be done mod 5, we will drop the $\pmod 5$ notation here.

$$\begin{bmatrix} 2 & 1 & 4 & 1 & 0 & 0 \\ 3 & 0 & 2 & 0 & 1 & 0 \\ 1 & 3 & 4 & 0 & 0 & 1 \end{bmatrix} \xrightarrow{3R_1} \begin{bmatrix} 1 & 3 & 2 & 3 & 0 & 0 \\ 3 & 0 & 2 & 0 & 1 & 0 \\ 1 & 3 & 4 & 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{-3R_1+R_2} \begin{bmatrix} 1 & 3 & 2 & 3 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 3 & 4 & 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{-R_1+R_3} \begin{bmatrix} 1 & 3 & 2 & 3 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 2 & 2 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{3R_3} \begin{bmatrix} 1 & 3 & 2 & 3 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 3 \end{bmatrix}$$

$$\xrightarrow{-R_3+R_2} \begin{bmatrix} 1 & 3 & 2 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 & 0 & 3 \end{bmatrix}$$

$$\xrightarrow{-2R_3+R_1} \begin{bmatrix} 1 & 3 & 0 & 1 & 0 & 4 \\ 0 & 1 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 & 0 & 3 \end{bmatrix}$$

$$\xrightarrow{-3R_2+R_1} \begin{bmatrix} 1 & 0 & 0 & 1 & 2 & 3 \\ 0 & 1 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 & 0 & 3 \end{bmatrix}$$

So the inverse of this matrix mod 5 is,

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 1 & 0 & 3 \end{bmatrix}$$

Notice in the reduction above there were two steps where we needed to divide by 2, the first row operation and the fourth row operation. In both cases we multiplied by 3, which is the inverse of 2 modulo 5. We can check our answer by multiplying it by the original and reducing the result mod 5.

$$\begin{bmatrix} 2 & 1 & 4 \\ 3 & 0 & 2 \\ 1 & 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 1 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 20 \\ 5 & 6 & 15 \\ 5 & 5 & 21 \end{bmatrix} \equiv \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \pmod 5$$

Since we got the identity matrix, we know that our process worked.

Another way to calculate an inverse to a matrix is by using the adjugate (or classical adjoint) of the matrix and dividing it by the determinant. Recall that

$$A^{-1} = \frac{1}{\det(A)} \operatorname{adj}(A)$$

Where 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. Let's do our above example using the adjugate method. We will also use the notation A_{ij} to denote the i, j minor of the matrix A and C_{ij} as the i, j cofactor of A.

$$A = \begin{bmatrix} 2 & 1 & 4 \\ 3 & 0 & 2 \\ 1 & 3 & 4 \end{bmatrix}$$

$$C_{11} = \begin{vmatrix} 0 & 2 \\ 3 & 4 \end{vmatrix} = 4 \qquad C_{21} = -\begin{vmatrix} 1 & 4 \\ 3 & 4 \end{vmatrix} = 3 \qquad C_{31} = \begin{vmatrix} 1 & 4 \\ 0 & 2 \end{vmatrix} = 2$$

$$C_{12} = -\begin{vmatrix} 3 & 2 \\ 1 & 4 \end{vmatrix} = 0 \qquad C_{22} = \begin{vmatrix} 2 & 4 \\ 1 & 4 \end{vmatrix} = 4 \qquad C_{32} = -\begin{vmatrix} 2 & 4 \\ 3 & 2 \end{vmatrix} = 3$$

$$C_{13} = \begin{vmatrix} 3 & 0 \\ 1 & 3 \end{vmatrix} = 4 \qquad C_{23} = -\begin{vmatrix} 2 & 1 \\ 1 & 3 \end{vmatrix} = 0 \qquad C_{33} = \begin{vmatrix} 2 & 1 \\ 3 & 0 \end{vmatrix} = 2$$

So the cofactor matrix is

$$\begin{bmatrix} 4 & 0 & 4 \\ 3 & 4 & 0 \\ 2 & 3 & 2 \end{bmatrix}$$

and the adjugate matrix is,

$$\begin{bmatrix} 4 & 3 & 2 \\ 0 & 4 & 3 \\ 4 & 0 & 2 \end{bmatrix}$$

Since $\det(A) \pmod 5 \equiv 4$ and $4^{-1} \pmod 5 \equiv 4$ we have

$$A^{-1} = 4 \cdot \begin{bmatrix} 4 & 3 & 2 \\ 0 & 4 & 3 \\ 4 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 1 & 0 & 3 \end{bmatrix}$$

The $A^{-1} = \frac{1}{\det(A)} \operatorname{adj}(A)$ formula for the inverse is what led us to the theorem that a square matrix has an inverse if and only if the determinant of the matrix was not 0. Since 0 was the only number that could not be inverted in the real number system. What does this mean when doing modular arithmetic? As we saw in the beginning of this section, numbers other than 0 may not be able to be inverted modulo n. One example we have seen was $4^{-1} \pmod{26}$ does not exist since $\gcd(4, 26) = 2 \neq 1$, so 4 is not invertible modulo 26. So if the determinant of A modulo 26 turns out to be 4 then we will not be able to invert the matrix. On the other hand, if the determinant of A modulo 26 turns out be 1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, or 25, then the matrix will be invertible and its inverse can be computed by the adjucate formula. This gives us a slightly modified version of our determinant criteria for invertibility.

Theorem 7: *Let A be an $n \times n$ integer matrix, then $A^{-1} \pmod{n}$ exists if and only if $\det(A)$ is invertible modulo n. That is, if and only if $(\det(A))^{-1} \pmod{n}$ exists.*

Bibliography

[1] Douglas Adams. *The More Than Complete Hitchhiker's Guide.* Longmeadow Press, 1986.

[2] Ibrahim A. Al-Kadi. The origins of cryptology: The Arab contributions. *Cryptologia*, 16(2):97–126, 1992.

[3] Elad Barkam, Eli Biham, and Nathan Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. *Journal of Cryptology*, 21(3), 2008.

[4] Wayne G. Barker. *Cryptanalysis of the Hagelin Cryptograph.* Aegean Park Press, 1977.

[5] H. Beker and F. Piper. *Cipher Systems: The Protection of Communications.* Wiley-Interscience, 1982.

[6] Brian Bowers. *Sir Charles Wheatstone FRS: 1802–1875.* Institution of Electrical Engineers, 2^{nd} edition, 2001.

[7] John Cleese, Eric Idle, Graham Chapman, Terry Jones, Michael Palin, and Terry Gilliam. Monty Python's Flying Circus. BBC, 1969–1974.

[8] Michael J. Crowe. *A History of Vector Analysis: The Evolution of the Idea of a Vectorial System.* Dover Books on Mathematics, 2011.

[9] Félix Marie Delastelle. *Traité Élémentaire de Cryptographie.* Paris: Gauthier-Villars, 1902.

[10] Arthur Conan Doyle. *The Original Illustrated Sherlock Holmes.* Castle Books, 1891–1905.

[11] William Dunham. *Journey through Genius: The Great Theorems of Mathematics.* Penguin Books, 1991.

[12] American editors at the Oxford University Press. *New Oxford American Dictionary.* Oxford University Press, Oxford, England, 3^{rd} edition, 2010.

[13] Ralph Erskine. Enigma's Security: What the Germans Really Knew. *Action this Day*, 2001.

[14] O. I. Franksen. *Mr. Babbage's Secret: the Tale of a Cipher—and APL*. Prentice Hall, 1985.

[15] Wes Freeman, Geoff Sullivan, and Frode Weierud. Purple Revealed: Simulation and Computer-Aided Cryptanalysis of Angooki Taipu B. *Cryptologia*, 27, 2003.

[16] William F. Friedman. Preliminary Historical Report on the Solution of the 'B' Machine. *Army Security Agency, SHR-159, CCH Files*, 1945.

[17] Helen F. Gaines. *Cryptanalysis: A Study of Ciphers and Their Solution*. Dover Publications, 1989.

[18] Greg Goebel. Codes & Codebreakers In World War I. http://www.vectorsite.net/ttcode_04.html#m3, 2014. [Online; accessed June 11, 2015].

[19] George Gore. *The Art of Scientific Discovery: Or, The General Conditions and Methods of Research in Physics and Chemistry*. Longmans, Green, and Co., 1878.

[20] David A. Hatch. Enigma and PURPLE: How the Allies Broke German and Japanese Codes During the War. *Center for Cryptologic History, National Security Agency*.

[21] P. Hetzel. Time dissemination via the LF transmitter DCF77 using a pseudo-random phase-shift keying of the carrier. Second European Frequency and Time Forum, 1988.

[22] Lester S. Hill. Cryptography in an Algebraic Alphabet. *Amer. Math. Monthly*, 36(6):306–312, 1929.

[23] Raymond Hill. *A First Course in Coding Theory*. Clarendon Press, Oxford, New York, 1st edition, 1986.

[24] Paul Y. Hoskisson. Jeremiah's Game. *Insights*, 30(1), 2013.

[25] Philip H. Jacobson. The Japanese Cipher Machines. *Turner Publishing Company*, 1996.

[26] David Kahn. *The Codebreakers: The Story of Secret Writing*. MacMillan Company, New York, 1967.

[27] David Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996.

[28] F. W. Kasiski. *Die Geheimschriften und die Dechiffrir-Kunst*. E. S. Mittler und Sohn, Berlin, 1863.

[29] Gregory Kipper. *Investigator's Guide to Steganography*. CRC Press, 1st edition, 2003.

[30] Melville Klein. Securing Record Communications: The TSEC/KW-26. https://www.nsa.gov/about/_files/cryptologic_heritage/publications/misc/tsec_kw26.pdf. [Online; accessed June 25, 2015].

[31] David Lay. *Linear Algebra and its Applications.* Pearson, Boston, 3^{rd} edition, 2006.

[32] Yi Lu, Willi Meier, and Serge Vaudenay. The Conditional Correlation Attack: A Practical Attack on Bluetooth Encryption. *Crypto*, 2005.

[33] Maxima. Maxima, a Computer Algebra System. http://maxima.sourceforge.net/, 2014. [Online; accessed January 2, 2014].

[34] Bill Miller. The Very Visible Battle Over Invisible Ink. *Los Angeles Times*, 2001.

[35] Frank Miller. *Telegraphic code to insure privacy and secrecy in the transmission of telegrams.* C.M. Cornwell, 1882.

[36] Richard A. Mollin. *An Introduction to Cryptography.* Chapman & Hall/CRC, Taylor & Francis Group, 6000 Broken Sound Parkway NW, Suite 300, 2^{nd} edition, 2007.

[37] David P. Mowry. Regierunges-Oberinspektor Fritz Menzer: Cryptographic Inventor Extraordinaire. *Cryptologic Quarterly*, 2(3–4), 1983–1984.

[38] David P. Mowry. German Cipher Machines of World War II. *Center for Cryptologic History, National Security Agency*, 2014.

[39] John Munro. *Heroes of the Telegraph.* The Religious Tract Society, 1891.

[40] David E. Newton. *Encyclopedia of Cryptography.* ABC-CLIO, 1997.

[41] University of Siegen et al. CrypTool 2.0. http://www.cryptool.org, 2014. [Online; accessed May 27, 2015].

[42] Signal Intelligence Service of the Army High Command. The C-38/M-209 Cipher Machine. *European Axis Signal Intelligence*, 4.

[43] Alberto Perez. Secrets Abroad: A History of the Japanese Purple Machine. http://www.wondersandmarvels.com/, 2013. [Online; accessed July 29, 2015].

[44] A. Poorghanad, A. Sadr, and A. Kashanipour. Generating High Quality Pseudo Random Number Using Evolutionary Methods. *IEEE Congress on Computational Intelligence and Security*, 9(3), 2008.

[45] Francis A. Raven. Some Notes on Early Japanese Naval/Diplomatic Cipher Machines. *CCH Series File, IV.W.III.23.*

[46] J. Reeds, D. Ritchie, and R. Morris. The Hagelin Cipher Machine (M-209): Cryptanalysis from Ciphertext M-209 Alone. *unpublished technical memorandum, Bell Laboratories*, 1978.

[47] Daily Mail Reporter. British Muslim 'had Al Qaeda contacts book with terrorists' numbers written in invisible ink'. *Daily Mail (London)*, 2008.

[48] Daily News Reporter. ARYAN PRISON GANG LINKS WITH MAFIA Drugs, money & the Gambinos. *Daily News (New York)*, 2002.

[49] Times Reporter. The formula for invisible ink will remain classified. *St. Petersburg Times*, 1999.

[50] Ron Roth. *Introduction to Coding Theory*. Cambridge University Press, Boston, 1^{st} edition, 2006.

[51] C. Edward Sandifer. *The Early Mathematics of Leonhard Euler*. The Mathematical Association of America, 2007.

[52] William Shakespeare. *The Plays of William Shakespeare*. The Tech, MIT, 1993.

[53] William Shakespeare. *The Complete Works of William Shakespeare*. Project Gutenberg, 1994.

[54] Claude Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28(4), 1949.

[55] Simon Singh. *The Code Book: The Science of Secrecy form Ancient Egypt to Quantum Cryptography*. Anchor Books, New York, 1^{st} edition, 1999.

[56] Simon Singh. The Black Chamber. http://www.simonsingh.net/The_Black_Chamber/, 2015. [Online; accessed May 12, 2015].

[57] Abraham Sinkov. *Elementary Cryptanalysis: A Mathematical Approach*. Mathematical Association of America Textbooks, 1998.

[58] Don Spickler. Linear ME. http://facultyfp.salisbury.edu/despickler/personal/LinearME.asp, 2014. [Online; accessed January 2, 2014].

[59] Don Spickler. Cryptography Explorer. http://facultyfp.salisbury.edu/despickler/personal/CryptographyExplorer.asp, 2015. [Online; accessed April 17, 2015].

[60] Richard Tomlinson. *The Big Breach: From Top Secret to Maximum Security*. Mainstream Publishing, 2001.

[61] C. Suetonius Tranquillus. *The Lives of the Twelve Caesars*. Corner House, Williamstown, Mass., 1978.

[62] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, Upper Saddle River, NJ 07458, 2^{nd} edition, 2006.

[63] United States Army. *Field Manual 34-40-2*. United States Army http://www.umich.edu/~umich/fm-34-40-2/, 1996. [Online; accessed June 7, 2015].

[64] Unknown. Linear Feedback Shift Registers in Virtex Devices.

[65] Unknown. Pearl Harbor Review - Red and Purple. *Center for Cryptologic History, National Security Agency*.

[66] Unknown. Red and Purple: A Story Retold (U). *Cryptologic Quarterly*.

[67] Unknown. Section 9.5 of the SATA Specification, revision 2.6.

[68] Unknown. Stasi Sprach Morse Machine. The Numbers Stations Research and Information Center.

[69] Unknown. The Achievements of the Signal Security Agency in World War II, German analysis of converter M-209 — POW Interrogations .

[70] Unknown. Traditional Pseudo-random Sequences. RFC 4086 section 6.1.3.

[71] Unknown. The Role of Radio Intelligence in the American-Japanese Naval War (August, 1941 - June, 1942). *SRH-012, CCH files*, 1942.

[72] Unknown. M-209 Manual, 1944.

[73] Unknown. Empiricism: The influence of Francis Bacon, John Locke, and David Hume. https://web.archive.org/web/20130708012140/http://www.psychology.sbc.edu/Empiricism.htm, 2013. [Online; accessed June 10, 2015].

[74] Laurie Ure. Spy agency reveals invisible ink formula. *CNN*, 2011.

[75] J. H. van Lint. *Introduction to Coding Theory*. Springer, Berlin Heidelberg, 3rd edition, 1998.

[76] Jennifer Wilcox. Solving the ENIGMA: History of the Cryptanalytic Bombe. *Center for Cryptologic History, National Security Agency*.

[77] Wolfram. Mathematica. http://www.wolfram.com/, 2014. [Online; accessed January 2, 2014].

[78] Fred B. Wrixon. *Codes Ciphers & Other Cryptic & Clandestine Communication*. Tess Press, New York, 1998.

Index

5-UCO, 183
91-shiki injiki, 385
97-shiki injiki, 385

ADFGVX, 136, 139
ADFGVX Cipher, 557
ADFGX, 136, 139
ADFGX Cipher, 554
Adlemen, Leonard, 395, 567
Affine Cipher, 532
Affine Substitution Cipher, 19
Affine Transformation, 19
Al-Kindi, 18
Alice, Bob & Eve, 2
Amazon, 416
American Black Chamber, 385
American Standard Code for Information Interchange, 398
Angōki A-kata, 385
Angōki B-kata, 385
ASCII, 398
Associated Polynomial, 171
Atbash, 15
Auguste Kerckhoffs, 3
Authentication, 11

Babbage, Charles, 54, 575
Bacon's Cipher, 642
Bacon, Francis, 642
Baconian Cipher, 642
Bellaso, Giovan Battista, 533
Bifid Cipher, 127
Blaise de Vigenère, 47
Bletchley Park, 395
Block Cipher, 9
Blum-Blum-Shub Algorithm, 183, 190, 606
Bombe, 384
Brent's Method Factoring, 614
Brute Force Discrete Logarithm, 630
Brute Force Factoring, 606
Brute-Force Attack, 3

C-38, 386
Caesar, 15
Caesar Cipher, 15
Caesar Shift Cipher, 15, 18, 532
Certificate, 416
Check Digit, 6
Chiffre Indéchiffrable, 47
Chinese Remainder Theorem, 596
Chosen Ciphertext Attack, 11
    Adaptive Chosen Ciphertext Attack, 11
Chosen Plaintext Attack, 11
    Adaptive Chosen Plaintext Attack, 11
Cipher, 4
Cipher Machines, 380
Ciphertext, 2
Ciphertext Only Attack, 10
Classical Columnar, 93, 140, 538
Classical Cryptography, 1, 13
Claude Shannon, 4
Cocks, Clifford, 395, 567
Code, 4
Coding Theory, 5
Coincidence Analysis, 55, 78, 577
Columnar Cipher, 93, 538
Computationally Infeasible, 9, 393
Confidentiality, 11

Crib, 17
Cryptanalysis, 1
Cryptography, 1
Cryptography Explorer, 526
  Affine Cipher, 532
  Arithmetic, 407, 595
  Bar Charts, 529
  Chinese Remainder Theorem, 596
  Classical Columnar, 538
  Discrete Logarithm, 597
  Double Factorial, 596
  Euler Phi, 409, 597
  Euler Totient, 409, 597
  Evaluate, 598
  Expression Syntax, 598
  Factor, 409, 597
  Factorial, 596
  GCD, 408, 596
  Index Calculus, 598
  Input and Output Boxes, 527
  Integer Calculator Functions, 595
  Introduction, 526
  Is Prime, 408, 597
  Is Primitive Root, 597
  Jacobi Symbol, 597
  Kama-Sutra Cipher, 532
  Modular Arithmetic, 408, 596
  Myszkowski Method Columnar, 539
  Next Prime, 408, 597
  Nth Prime, 597
  Pi(n), 597
  Pohlig-Hellman/Pollard Rho, 597
  Previous Prime, 408, 597
  Prime Number Functions, 408, 597
  Primitive Root, 597
  Print Preview, 528
  Program Layout, 526
  Quick Help Bar, 527
  Random Substitution Cipher, 532
  Semiprime, 408, 597
  Shift Cipher, 532
  Square Root, 596
  Text Conversion Options, 191, 584
  Tools
    ADFGVX, 152, 557
    ADFGX, 152, 554
    Brent's Method Factoring, 614
    Brute Force Discrete Logarithm, 630
    Brute Force Factoring, 606
    Coincidence Analysis, 78, 577
    Columnar, 102, 538
    Dot Product Analysis, 80, 578
    ElGamal, 569
    Elliptic Curve Calculator, 602
    Enigma, 387, 565
    Factoring, 626
    Fermat Factoring, 608
    Four Square, 133, 545
    Frequency Analysis, 41, 78, 571
    Gridpad, 591
    Hill, 210, 300, 561
    Hill Climbing Analysis, 44, 156, 573
    Index Calculus Discrete Logarithm, 636
    Integer Calculator, 215, 306, 407, 595
    Kasiski's Method, 77, 575
    Lenstra's Elliptic Curve Factoring, 623
    LFSR Cipher Analysis, 174, 580
    Linear Feedback Shift Register (LFSR), 173, 188, 560
    Modular Matrix Calculator, 212, 303, 599
    Monoalphabetic Substitution, 40, 530
    Multiple Polynomial Quadratic Sieve Factoring, 620
    Notepad, 591
    Playfair, 114, 550
    Pohlig-Hellman Discrete Logarithm, 631
    Pohlig-Hellman with Pollard Rho Discrete Logarithm, 634
    Pollard $p - 1$ Factoring, 609
    Pollard Rho Discrete Logarithm, 633
    Pollard Rho Factoring, 612
    Quadratic Sieve Factoring, 616

Rail Fence, 92, 537
Random Number Generator, 189, 605
RSA, 403, 567
Scytale, 89, 536
Substring Compare, 104, 155, 579
Text Combiner, 583
Text Converter, 190, 405, 584
Text Extractor, 81, 582
Two Square, 123, 541
User Defined Language Creator, 593
Variant of the Index Calculus Discrete Logarithm, 637
Vigenère, 75, 187, 533
Williams $p + 1$ Factoring, 611
Vigenère Ciphertext Autokey, 76, 534
Vigenère Plaintext Autokey, 76, 533
Vigenère Repeated Keyword, 76, 533
Cryptology, 1
CSP-1500, 386

Damm, Arvid, 380
Delastelle, Félix Marie, 118, 127
Diffe, Whitfield, 567
Diffie, Whitfield, 391, 395
Digram, 13
Dot Product Analysis, 66, 80, 578
Double Factorial, 24, 596
Double Transposition Cipher, 93

ElGamal Cipher, 569
ElGamal, Taher, 569
English Digram Frequencies, 27
English Letter Relative Frequencies, 27
English Trigram Frequencies, 27
English Word Frequencies, 28
Enigma, 380
Enigma Cipher, 387, 565
Enigma Machine Rotor and Reflector Wirings, 388, 566
Euler Phi, 409, 597
Euler Phi Function, 21
Euler Totient, 21, 409, 597
Euler Totient Function, 397
Euler's Generalization to Fermat's Little Theorem, 397

Factorial, 596
Factoring, 626
Feedback Shift Register, 172
Fermat Factoring, 608
Fermat's Little Theorem, 397
FISH, 384
Fitness Measure, 38, 94
Four Square Cipher, 127, 545
Fractionation Cipher, 127
Freemasons, 15
Frequency Analysis, 17, 27, 41, 78, 571
FSR, 172

GC&CS, 395
GCCS, 395
GCHQ, 395
GEE, 183
Giovanni Porta, 47
Government Code and Cypher School, 395
Government Communications Headquarters (GCHQ), 567

Hagelin, Boris, 386
Hebern, Edward, 380
Hellman, Martin, 391, 395, 567
Herodotus, 641
Hill Cipher, 197, 287, 561
Hill Climbing, 36, 94, 113, 123, 132
Hill Climbing Analysis, 156, 573
Hill, Lester, 197, 287, 561
Histiaeus, 641
Holmes, Oliver Wendell, 106

Index Calculus Discrete Logarithm Algorithm, 636
Integrity, 11
Irrevocability, 11
ISBN-10, 6

Jacobi Symbol, 597
Johannes Trithemius, 47

Kama-Sutra Cipher, 15, 23, 532

Kasiski's Method, 54, 77, 575
Kasiski, Friedrich, 47, 54, 575
Kerckhoffs' Principle, 4
Key, 3
Key Distribution Problem, 182, 391
Key Space, 3
Keyword Substitution, 26
Known Plaintext Attack, 10
Koch, Hugo, 380

Length of a Recurrence Relation, 160
Lenstra's Elliptic Curve Factoring, 623
Lenstra's Elliptic Curve Factoring Algorithm, 623
Leon Battista Alberti, 47
LFSR, 159, 162, 560
LFSR Cipher Analysis, 164, 174, 580
Linear Congruential Algorithm, 605
Linear Feedback Shift Register, 159, 162, 560
Lysander of Sparta, 88

M-209, 386
Main Objectives of Cryptography, 11
Mathematica, iii, 419
  Algebra, 427
  Arithmetic on an Elliptic Curve, 459
  Assignment, 427
  Brackets and Parentheses, 422
  Chinese Remainder Theorem, 434
  Command Color Coding, 424
  Commands
    ., 229, 321, 449
    //, 426
    :=, 427, 430
    ;;, 439, 447
    =, 427
    =., 427
    [[]], 446
    AbsoluteTiming, 429
    All, 447
    ChineseRemainder, 434
    Clear, 427
    ContinuedFraction, 437
    Convergents, 439
    Det, 230, 321, 450
    EulerPhi, 442
    ExtendedGCD, 433
    FactorInteger, 441
    FromContinuedFraction, 438
    FromContinuedFractionN, 439
    FullSimplify, 428
    GCD, 411, 433
    Inverse, 230, 231, 321, 323, 450, 452
    JacobiSymbol, 436
    Join, 448
    LCM, 434
    MatrixForm, 226, 317, 445
    MatrixPower, 450
    Mod, 231, 322, 413, 432, 452
    Modulus, 231, 323, 453
    MultiplicativeOrder, 443, 444
    N, 426
    NextPrime, 411, 435
    PowerMod, 411, 433
    PrimeQ, 411, 435
    PrimitiveRoot, 443
    PrimitiveRootList, 443
    PrimitiveRootQ, 444
    RowReduce, 230, 232, 322, 324, 451, 453
    Simplify, 428
    Solve, 440
    Table, 448
    Timing, 429
  Continued Fractions, 437
  CryptDSEC.nb, 462
  CryptDSEC.nb Code, 463
  Defining Functions, 430
  Delayed Assignment, 427
  Discrete Logarithms, 444
  Element Orders, 443
  Elliptic Curves, 454
  Euler Totient Function, 442
  Execution Timing, 429
  Extended Greatest Common Divisor, 433

Factoring, 441
Factoring Polynomials, 177, 442
Factoring Polynomials Modulo a Prime, 177, 442
Greatest Common Divisor, 433
Immediate Assignment, 427
Jacobi and Legendre Symbols, 436
Juxtaposition, 428
kernel, 420
Least Common Multiple, 434
Matrix Arithmetic, 229, 320, 449
Matrix Reduction, 230, 322, 451
Modular Inverses, 433
Modular Matrix Operations, 231, 322, 452
Modular Powers, 432
Modulus, 432
Number Theory, 432
Numeric Calculations, 425
Palettes, 423
Points on an Elliptic Curve, 455
Primes, 435
Primitive Roots, 443
Resetting Assignments, 427
Solving Equations, 440
User Interface, 420
Vectors and Matrices, 445
www.wolfram.com, 419

Mauborgne, Joseph, 181

Maxima, iii, 466
Aborting a Calculation, 471
Adjugate, 254, 347, 507
Arithmetic on an Elliptic Curve, 519
ASCII Mode, 473
Brackets and Parentheses, 470
Chinese Remainder Theorem, 482
Classical Adjoint, 254, 347, 507
Commands
%, 469
addcol, 244, 336, 496
addrow, 244, 336, 496
bfloat, 472
cf, 486
cfdisrep, 486
cflength, 486
chinese, 482
col, 244, 336, 496
columnvector, 243, 335, 495
copymatrix, 249, 341, 501
covect, 243, 335, 495
determinant, 249, 341, 502
echelon, 251, 344, 504
ezgcd, 481
factor, 491
float, 472
fpprec, 473
from_cf, 487
from_cf_n, 487
gcd, 413, 480
gcdex, 481
gf_sqrt, 490
ifactors, 491
igcdex, 481
inv_mod, 480
invert, 249, 341, 502
jacobi, 484
lcm, 482
mat_mod_inverse, 256, 348, 509
matrix, 242, 334, 495
minor, 245, 337, 498
mod, 415, 479
mod_echelon, 259, 351, 511
mod_rref, 259, 351, 511
Modulus, 440
modulus, 489
msqrt, 490
next_prime, 413, 483
numer, 472
power_mod, 413, 480
prev_prime, 483
primep, 413, 482
primep_number_of_tests, 482
ratsimp, 474, 486
row, 243, 336, 496
rref, 253, 345, 506
set_display, 473

ssqrt, 490
submatrix, 245, 337, 498
totient, 492
transpose, 243, 335, 495
triangularize, 251, 344, 504
zn_log, 493
zn_order, 494
zn_primroot, 492
Continued Fractions, 485
CryptDS.mac, 523
CryptDS.mac Code, 523
Defining a Matrix, 495
Defining Functions, 475
Discrete Logarithms, 493
eigen Package, 242, 334, 495
Elliptic Curves, 516
engine, 468
Euler Totient Function, 492
Execution Timing, 475
Extended Greatest Common Divisor, 481
Factoring, 491
Factoring Polynomials, 177, 491
Factoring Polynomials Modulo a Prime, 177, 491
gf Package, 490
Greatest Common Divisor, 480
In Lines, 468
Jacobi and Legendre Symbols, 484
Juxtaposition, 474
Least Common Multiple, 482
Lucas Pseudo-Primality Test, 482
Matrix Arithmetic, 249, 341, 502
maxima.sourceforge.net, 466
Miller-Rabin's Pseudo-Primality Test, 482
Modular Computations, 479
Modular Inverses, 480
Modular Matrix Operations, 254, 346, 506
Modular Matrix Reduction, 257, 349, 509
Modular Powers, 479
Modular Roots, 490
None Mode, 473
Number Theory, 479
Numeric Calculations, 471
Order of an Element, 494
Out Lines, 468
Panes, 471
Pohlig-Hellman, 493
Points on an Elliptic Curve, 516
Pollard's Rho Method, 493
Primality Testing, 482
Prime Number Functions, 482
Primitive Roots, 492
Simplification, 474
Solving Equations, 488
User Interface, 468
Variable Assignment, 473
Vectors and Matrices, 242, 334, 495
wxMaxima, 468
XML Mode, 473
Mersenne Primes, 171
Message Unit, 13
Miller Frank, 179
Modern Cryptography, 1
Monoalphabetic Substitution, 15
Monoalphabetic Substitution Cipher, 530
Morse Code, 4
Multiple Polynomial Quadratic Sieve Factoring, 620
Myszkowski Method, 93
Myszkowski Method Columnar, 539
Myszkowski, Emile Victor Théodore, 93, 539

Nebel, Fritz, 136, 554, 557
Nomenclator, 18
Nomenclature, 18
Noreen, 183

One Time Pad, 179
One-Way Function, 393

Painvin, Georges, 140
Pan Galactic Gargle Blaster, 129, 137

Parity Bit, 6
Perfect Secrecy, 179
Pharnabazus of Persia, 88
Pigpen Cipher, 15
Plaintext, 2
Playfair Cipher, 106, 109, 550
Playfair, Lyon, 106, 550
Pohlig-Hellman Discrete Logarithm Algorithm, 631
Pollard $p-1$ Factoring, 609
Pollard Rho Discrete Logarithm Algorithm, 633
Pollard Rho Factoring, 612
Polyalphabetic Cipher, 47
Polybius Square, 108, 118, 136
Primitive Root, 597
Pseudorandom Numbers, 182
Public-Key Cryptography, 2, 8
PURPLE, 385

Quadratic Sieve Factoring, 616

Rail Fence Cipher, 91, 537
Random Substitution, 24
Random Substitution Cipher, 532
RED, 385
Rivest, Ron, 395, 567
Rockex, 183
RSA, 395
RSA Algorithm, 395, 567
RSA Cipher, 567

Scherbius, Arthur, 380
Scytale, 88
Scytale Cipher, 536
Shamir, Adi, 395, 567
Shannon, Claude, 182
Sherlock Holmes, 16
Siemens & Halske, 384
Signal Intelligence Service, 385
SIGSALY, 182
SIGTOT, 183
Silverman, Robert, 616
Standard Elektrik Lorenz, 384
Stasi Sprach Machine, 183
Statistical Randomness, 182
Steganography, 1, 640
    Additive Colors, 647
    Image Steganography, 645
    Invisible Ink, 644
    Least Significant Bit, 645
    Newspaper Code, 643
    Pentel Rolling Writer, 644
    Richard Tomlinson, 644
    Scytale, 641
    Subtractive Colors, 646
    The Microdot, 645
    Type Spacing and Offsetting, 644
    Typeface Changes, 642
    Wax Tablets, 640
Stereoscope, 106
Stream Cipher, 9
STURGEON, 384
Substitution Cipher, 14
Substring Compare, 155, 579
Symmetric-Key Cryptography, 2, 8, 390

The Adventures of the Dancing Men, 16
The Hotline, 182
Three-Pass Protocol, 391
Traictè des Chiffres, 47
Transparency, 122
Transposition Cipher, 14
Trifid Cipher, 127
TUNNY, 384
Two Square Cipher, 118, 541

Unicode, 398

Variant of the Index Calculus Discrete Logarithm Algorithm, 637
VeriSign Inc., 416
Vernam Cipher, 181
Vernam Cipher Machine, 181
Vernam, Gilbert S., 181
Vernam-Mauborgne Cipher, 181
View-Master, 106
Vigenère Cipher, 47, 533

Vigenère Ciphertext Autokey, 50, 76, 534
Vigenère Plaintext Autokey, 50, 76, 533
Vigenère Repeated Keyword, 48, 76, 533
Vigenère, Blaise de, 533
Wheatstone Bridge, 106
Wheatstone, Charles, 106
Williams $p + 1$ Factoring, 611