Luke Sellmayer
May 4th, 2023
APPM 3310
Section 003

Analysis of Speed, Security, and Usability of the Golden Matrix Cryptographic Method

Abstract

A family of matrices related to the Fibonacci Q-matrices, known as golden matrices, can act as a cryptographical method to encrypt and decrypt messages. The method is very fast, running in constant time regardless of the message or key used to encrypt and decrypt. The method is also very simple to use, as encryption is done by multiplying a message matrix with a golden matrix, with decryption handled by using the inverse of the golden matrix. Despite these benefits, the method is limited by the range of possible key values that can be used to form the golden matrix used to encrypt messages, and is not very secure due to its weaknesses against chosen-plaintext and brute force attacks.

Introduction

Cryptography, derived from the Greek words 'kryptos' meaning 'hidden' and 'graphein' meaning 'to write' (Merriam-Webster), applies mathematics and logic to design encryption methods. The primary aim of cryptography is to ensure confidentiality, integrity, and authenticity of data that is being transmitted or stored. Cryptography has become more important than ever in the modern age, with methods such as the RSA algorithm used to encrypt web traffic, the Elliptic Curve Digital Signature Algorithm used by Bitcoin to ensure the ownership of funds, and the Advanced Encryption Standard used by the United States Military for classified information (Rawat, 2021). As such, it is incredibly important that cryptographical techniques be fast, easy to use, and most importantly, secure.

Cryptography starts with plaintext data, which is the original data that the sender wishes to keep secret from third parties. The sender will use some type of cryptographical technique to encrypt the plaintext data into ciphertext, completely hiding the original plaintext data. The sender uses a key in order to carry out this encryption. The sender can then transmit the encrypted data to a designated receiver, who can decrypt the enciphered data back into the plaintext data if they know what the method of encryption was and the key used to do it. Knowledge of the encryption technique and the key used is essential to being able to decrypt data.

One can categorize encryption techniques into two categories: asymmetric-key and symmetric-key (Rawat, 2021). Asymmetric-key encryption uses a pair of keys that depend on each other: a public key and a private key. If a sender wishes to encrypt data to send to a receiver, they encrypt the data using the receiver's public key. The receiver is then able to decrypt the data using their corresponding private key. On the other hand, symmetric-key encryption uses a single key to both encrypt and decrypt data.

The topic of this paper, the golden matrix encryption method, is a symmetric-key encryption method (Stakhov, 2007) created by A.P. Stakhov, a professor at the Ukrainian Academy of Engineering Sciences. After deriving the formula for golden matrices and working through how the cryptographic method works, benefits and drawbacks of the method will be examined, such as ease of use, speed, key value ranges, and security. Ultimately, the goal is to answer whether this cryptographic method can be used in the real world.

Mathematical Formulation

Definition

Golden matrices are square matrices that are a generalization of the Fibonacci Q-matrix for a continuous domain. Recalling the Fibonacci sequence, which is given by the relation $F_1 = F_2 = 1, F_{n+1} = F_n + F_{n-1}$ where $n = 1, 2, \ldots$, Fibonacci Q-matrices have the following property (Hoggatt, 1979):

$$Q^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

We derive the golden matrix by extending the domain of the Fibonacci Q-matrix from a discrete domain to a continuous one (Stakhov, 2007). We do so by defining the symmetrical hyperbolic Fibonacci functions (Stakhov & Rozin, 2005):

$$sFs(x) = \frac{\varphi^x - \varphi^{-x}}{\sqrt{5}}$$

$$cFs(x) = \frac{\varphi^x + \varphi^{-x}}{\sqrt{5}}$$

$sFs(x)$ is the symmetrical hyperbolic Fibonacci sine, $cFs(x)$ is the symmetrical hyperbolic Fibonacci cosine, and $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. These functions are related to the Fibonacci numbers, and thus the entries of our golden matrices, as follows:

$$F_n = \begin{cases} sFs(n), & n = 2k \\ cFs(n), & n = 2k+1 \end{cases}$$

By rewriting the Fibonacci Q-matrix in one of the following forms:

$$Q^{2k} = \begin{bmatrix} F_{2k+1} & F_{2k} \\ F_{2k} & F_{2k-1} \end{bmatrix}$$

$$Q^{2k+1} = \begin{bmatrix} F_{2k+2} & F_{2k+1} \\ F_{2k+1} & F_{2k} \end{bmatrix}$$

We can then rewrite it using the symmetrical hyperbolic Fibonacci functions, using symmetrical hyperbolic Fibonacci sine for even-valued entries and symmetrical hyperbolic Fibonacci cosine for odd-valued entries.

$$Q^{2k} = \begin{bmatrix} cFs(2k+1) & sFs(2k) \\ sFs(2k) & cFs(2k-1) \end{bmatrix}$$

$$Q^{2k+1} = \begin{bmatrix} sFs(2k+2) & cFs(2k+1) \\ cFs(2k+1) & sFs(2k) \end{bmatrix}$$

Finally, we switch from the discrete variable k to the continuous variable x, giving us our golden matrix equations:

$$Q^{2x} = \begin{bmatrix} cFs(2x+1) & sFs(2x) \\ sFs(2x) & cFs(2x-1) \end{bmatrix}$$

$$Q^{2x+1} = \begin{bmatrix} sFs(2x+2) & cFs(2x+1) \\ cFs(2x+1) & sFs(2x) \end{bmatrix}$$

For clarity's sake, I'll refer to Q^{2x} as the "even golden matrix" and Q^{2x+1} as the "odd golden matrix." The inverses of the golden matrices (Hoggatt, 1979) are given by:

$$(Q^{2x})^{-1} = Q^{-2x} = \begin{bmatrix} cFs(2x-1) & -sFs(2x) \\ -sFs(2x) & cFs(2x+1) \end{bmatrix}$$

$$(Q^{2x+1})^{-1} = Q^{-(2x+1)} = \begin{bmatrix} -sFs(2x) & cFs(2x+1) \\ cFs(2x+1) & -sFs(2x+2) \end{bmatrix}$$

It is difficult to derive the above formulas[1], but one can verify that they are true:

$$Q^{2x}Q^{-2x} = \begin{bmatrix} cFs(2x+1) & sFs(2x) \\ sFs(2x) & cFs(2x-1) \end{bmatrix} \begin{bmatrix} cFs(2x-1) & -sFs(2x) \\ -sFs(2x) & cFs(2x+1) \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$$c_{11} = cFs(2x+1)cFs(2x-1) - \left(sFs(2x)\right)^2$$

$$c_{12} = -cFs(2x+1)sFs(2x) + cFs(2x+1)sFs(2x)$$

$$c_{21} = sFs(2x)sFs(2x-1) - sFs(2x-1)sFs(2x)$$

$$c_{22} = (sFs(2x))^2 + cFs(2x+1)cFs(2x-1)$$

We see that $c_{12} = c_{21} = 0$. We can find the other two entries by using the following identities that connect the symmetric hyperbolic trigonometric functions (Stakhov & Rozin, 2005):

$$\left(sFs(x)\right)^2 - cFs(x+1)cFs(x-1) = -1$$

$$(cFs(x))^2 - sFs(x+1)sFs(x-1) = 1$$

Which results in $c_{11} = c_{22} = 1$. Thus,

$$Q^{2x}Q^{-2x} = \begin{bmatrix} cFs(2x+1) & sFs(2x) \\ sFs(2x) & cFs(2x-1) \end{bmatrix} \begin{bmatrix} cFs(2x-1) & -sFs(2x) \\ -sFs(2x) & cFs(2x+1) \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Q^{-2x} is therefore the inverse of the even golden matrix Q^{2x}. A similar approach can be used to show that $Q^{-(2x+1)}$ is the inverse of the odd golden matrix Q^{2x+1} (Stakhov, 2007).

Encryption Method

We start with the plaintext data that we wish to encrypt. For this method, the plaintext data can be any sequence of real numbers $m_1, m_2, m_3, m_4 \ldots$. Using this message, we construct a square 2 x 2 message matrix M.

$$M = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix}$$

For a 2 x 2 matrix, there are 4! = 24 possible permutations (Stakhov, 2007) that we could arrange the plaintext in, so we'll designate this permutation as P_1 with P_i being the ith possible permutation. Once we have created our message matrix out of our plaintext, we then choose a golden matrix to encipher the message through matrix multiplication. The key that we are using for encryption, $K = \{P_i, x\}$, has two parts, the first being the permutation of the plaintext data, and the second being the x-key, which

[1] Let the record show that I tried to.

determines which golden matrix we will use for encryption (Stakhov, 2007). Once a permutation and x-key have been chosen, we then multiply our message matrix by the associated golden matrix. Choosing our x-key to be $x = x_1$, the encryption process is:

$$M \times Q^{2x_1} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \begin{bmatrix} cFs(2x_1+1) & sFs(2x_1) \\ sFs(2x_1) & cFs(2x_1-1) \end{bmatrix} = \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix} = E(x_1)$$

$$M \times Q^{2x_1+1} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \begin{bmatrix} sFs(2x_1+2) & fFs(2x_1+1) \\ cFs(2x_1+1) & sFs(2x_1) \end{bmatrix} = \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{bmatrix} = F(x_1)$$

Note that we have two choices in which golden matrix we choose to use, the even golden matrix or the odd golden matrix, and that using the same x-key for both matrices results in completely different cipher matrices, $E(x_1)$ and $F(x_1)$. The cipher matrices can be decrypted by multiplying them by the corresponding inverses of the golden matrices used previously (Stakhov, 2007):

$$E(x_1) \times Q^{-2x_1} = \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix} \begin{bmatrix} cFs(2x_1-1) & -sFs(2x_1) \\ -sFs(2x_1) & cFs(2x_1+1) \end{bmatrix} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} = M$$

$$F(x_1) \times Q^{-(2x_1+1)} = \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{bmatrix} \begin{bmatrix} -sFs(2x_1) & cFs(2x_1+1) \\ cFs(2x_1+1) & -sFs(2x_1+2) \end{bmatrix} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} = M$$

Example

Let's use the above process to encrypt and decrypt some plaintext. For simplicity's sake, our original message will be "LUKE" since it only has four characters; longer plaintext can be encrypted by splitting the plaintext into groups of four numbers and performing the encryption individually on each block of numbers. This method requires that the plaintext be a sequence of integers, so we'll start by transforming the alphabetical message into numbers by letting A = 1, B = 2, …, Z = 26.

$$LUKE = 12\ 21\ 11\ 5$$

We then place these numbers into our message matrix, M, using any of the 24 potential permutations: this example uses P_1.

$$M = \begin{bmatrix} 12 & 21 \\ 11 & 5 \end{bmatrix}$$

Next, we'll encrypt the matrices with a golden matrix. We'll need to select an x-key to use to complete our key, so I'll use x = 2.3. Our key is now $\{P_1, 2.3\}$. We can now choose which golden matrix to use: in this example, the even golden matrix is used.

$$MQ^{2\cdot 2.3} = \begin{bmatrix} 12 & 21 \\ 11 & 5 \end{bmatrix} \begin{bmatrix} cFs(2(2.3)+1) & sFs(2(2.3)) \\ sFs(2(2.3)) & cFs(2(2.3)-1) \end{bmatrix} \approx \begin{bmatrix} 164.6904 & 103.2691 \\ 93.3622 & 57.5044 \end{bmatrix} = E(2.3)$$

So E(2.3) is our encrypted cipher matrix. In order to decrypt it, we simply multiply it by the inverse of the golden matrix used to encrypt it with the same x-key value.

$$E(2.3)Q^{-(2\cdot 2.3)} = \begin{bmatrix} 164.6904 & 103.2691 \\ 93.3622 & 57.5044 \end{bmatrix} \begin{bmatrix} cFs(2(2.3)-1) & -sFs(2(2.3)) \\ -sFs(2(2.3)) & cFs(2(2.3)+1) \end{bmatrix} = \begin{bmatrix} 12 & 21 \\ 11 & 5 \end{bmatrix} = M$$

Numerical Results

Key Value Ranges

One question that arises when using this encryption technique is for what values of the x-key will the resulting decrypted matrix actually match the original plaintext matrix. To determine this, I used

MATLAB to create functions that encrypt[A] and decrypt[B] using the golden matrix method. Starting with a message matrix of M = [5, 6; 7, 8], I wrote a MATLAB script to perform a series of encryption and decryption trials on this matrix, using a different x-key value for every trial. The x-keys used were between -20 and 20 with an interval of 0.001. Once the message matrix had been encrypted and then decrypted back to its (supposed) original form, I then calculated the relative error, $relative\ error = \frac{|observed - expected|}{|expected|}$ for each entry in the original plaintext matrix and the resulting decrypted matrix. Here, the observed result is the value of the entry in the decrypted matrix and the expected result is the value of the entry in the original plaintext matrix. I then added the relative errors of all four entries in the matrix for each key value and plotted the key values against the relative errors that they produced [C].

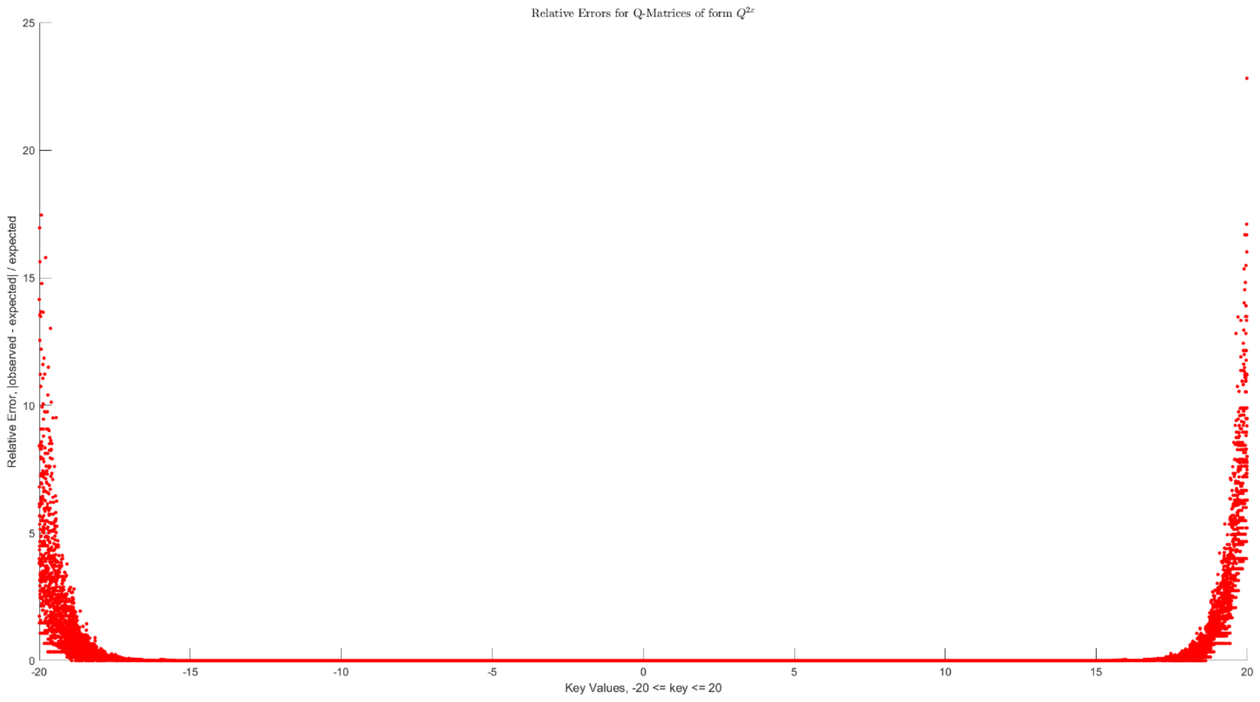


Figure 1: Relative Errors of Key Values for Even Golden Matrices

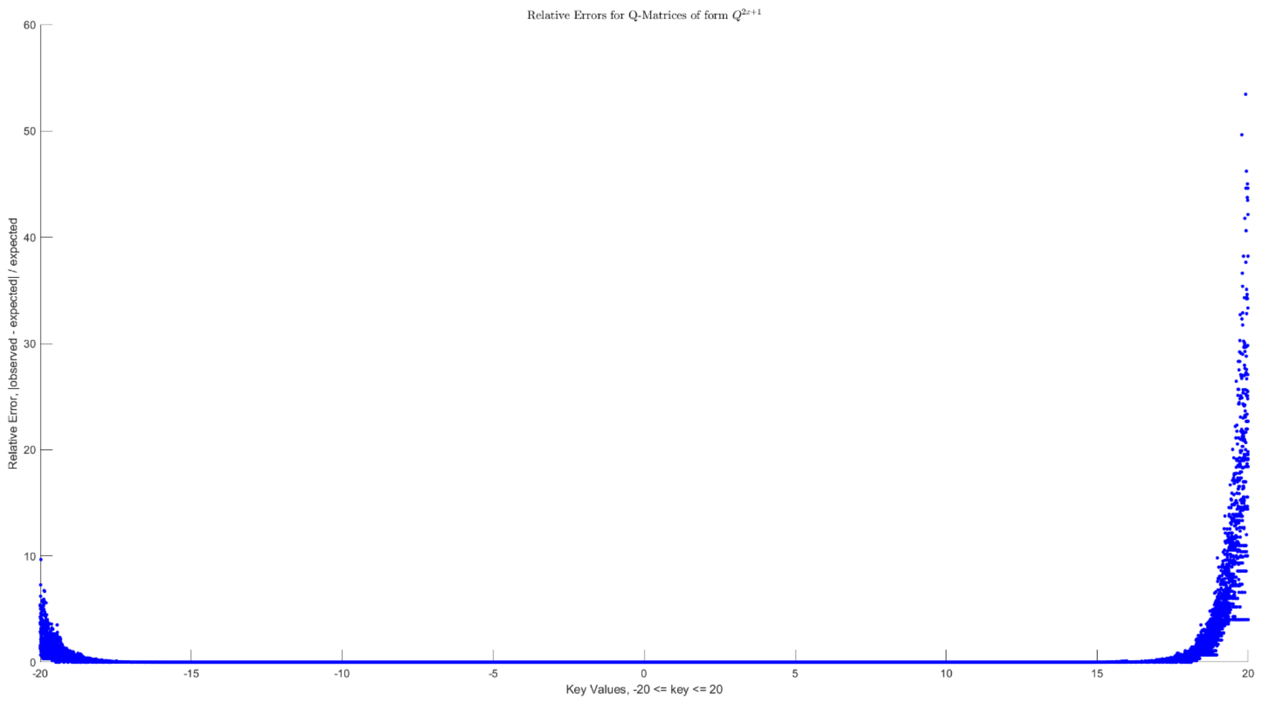


Figure 2: Relative Errors of Key Values for Odd Golden Matrices

The relative error for key values between – 15 and 15 is zero, indicating no change at all between the original plaintext matrix and deciphered matrix; this is what we want from any encryption method, as the original data should be the same once it has been encrypted and then decrypted. However, outside of this range, relative error increases dramatically, meaning that the decrypted matrix no longer matches the original plaintext matrix. As such, this method only works when the domain of key values is restricted to [-15, 15]; while there are still infinitely many key values within this range, this decreases the security of the method dramatically. Interestingly, relative errors for x-key values less than -15 are substantially smaller for odd golden matrices compared to even golden matrices, while relative errors for x-key values greater than 15 are substantially smaller for even golden matrices compared to odd golden matrices.

Time Complexity

Another question that arises is how quickly plaintext can be encoded and then decoded with this method; if the algorithm is fast enough, then it can be used to encrypt and decrypt in real time, an incredibly important feature in our digital interconnected world. To measure this, we first start by constructing a runtime algorithm for the encryption and decryption functions.

As a quick reminder of asymptotic analysis on runtime algorithms, if $T, g, f: \mathbb{Z}_{\geq k} \to \mathbb{R}$, we say that $T(n) \in O(g(n))$ if $\exists c, \exists h \in \mathbb{Z}^+$ such that $|T(n)| \leq c \cdot |g(n)|\ \forall n \geq h$. Similarly, we say that $T(n) \in \Omega(f(n))$ if $\exists c, \exists h \in \mathbb{Z}^+$ such that $|T(n)| \geq c \cdot |g(n)|\ \forall n \geq h$. In other words, if $T(n) \in O(g(n))$, then $g(n)$ is an upper bound for $T(n)$ for all $n \geq h$, and if $T(n) \in \Omega(f(n))$, then $f(n)$ is a lower bound for $T(n)$ for all $n \geq h$. Additionally, if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$, then we say that $T(n) \in \Theta(f(n))$, i.e. $T(n)$ grows at the same rate as $f(n)$ (Levet 2021).

With this in mind, we can construct a runtime function for our encryption and decryption algorithms and see if we can bound it with some other function. In order to make runtime calculations, we'll make two assumptions:

1) Single arithmetic, Boolean, and assignment operations, single numeric and Boolean comparisons, single input/output statements, return statements, and array index operations all take one time-step (Levet 2021).
2) Matrix multiplication in MATLAB, which is based on BLAS, or Basic Linear Algebra Subprograms, has a time complexity of $O(n^3)$ when multiplying two n x n matrices (Taylor 2013).

Using this, we can find an exact expression for the runtimes of the encryption[A] and decryption[B] algorithms as[D]:

$$GE(n) = GD(n) = 89 + 2 \cdot O(n^3)$$

Where $GE(n)$ if the runtime algorithm for encryption and $GD(n)$ is the runtime algorithm for decryption. We note that:

$$|89 + 2 \cdot O(n^3)| \geq 1 \cdot |n^3| \text{ for } n > 1$$

$$|89 + 2 \cdot O(n^3)| \leq 3 \cdot |n^3| \text{ for } n > 100$$

We have $GE(n) = GD(n) \in O(n^3), GE(n) = GD(n) \in \Omega(n^3)$, which implies that $GE(n) = GD(n) \in \Theta(n^3)$.

Both encryption and decryption are limited by the matrix multiplication that occurs within them. Luckily, this method only multiplies 2 x 2 matrices, which simplifies their runtime bound function from $O(n^3)$ to $O(8)$, which is constant runtime. Thus, the encryption and decryption programs should have a constant runtime regardless of the original message matrix and x-key value. To verify this, I wrote a script using MATLAB's *timeit* function to measure the time it takes to run the encryption and decryption functions with 100 random plaintext message matrices and random x-keys, resulting in the following figures[E].

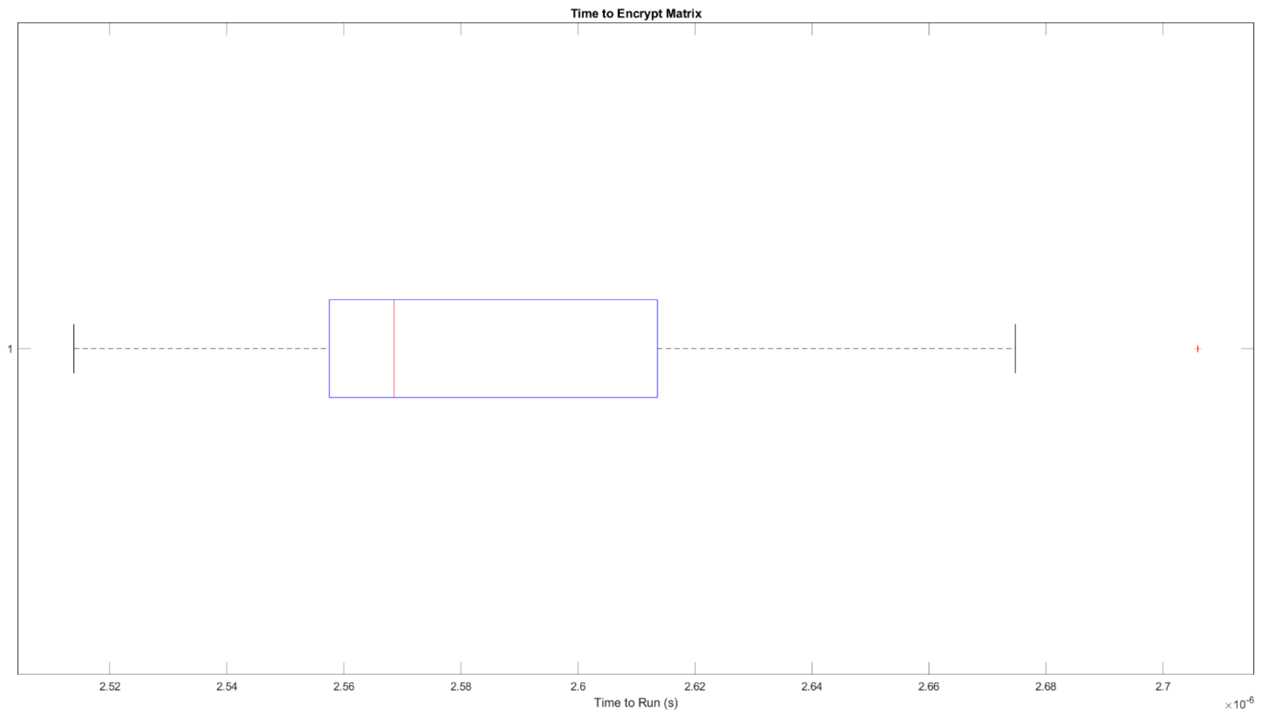


Figure 3: Box Plot of Time for Encryption Algorithm to Run

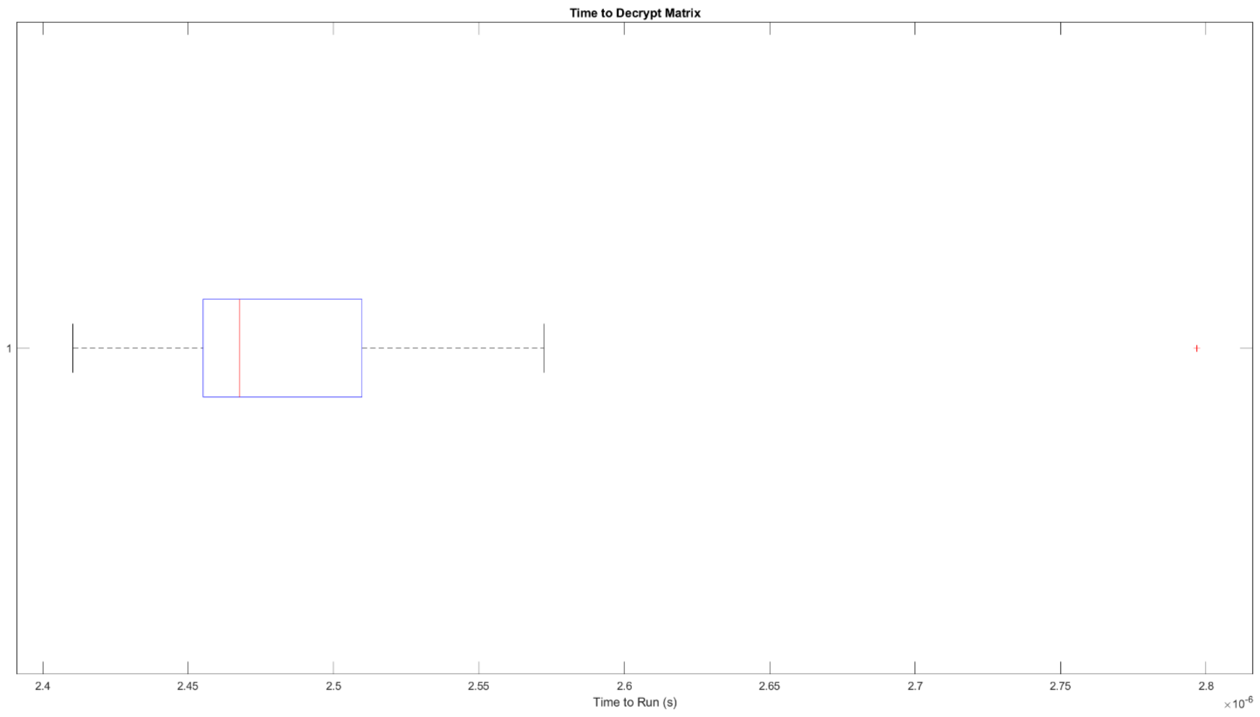


Figure 4: Box Plot of Time for Decryption Algorithm to Run

For both encryption and decryption, all runtimes were less than 2.8 microseconds, confirming that the functions have a constant runtime. It appears that decrypting is marginally faster than encrypting, likely due to the matrix memory writes required to construct the message matrix in the encryption algorithm.

Security

Despite the speed and simplicity of this cryptographic method, the method is ultimately insecure, being unable to defend against chosen-plaintext attacks (del Rey & Sanchez, 2008) and brute force attacks. Both of these attack methods require an adversary to have access to a plaintext matrix and its corresponding cipher matrix, upon which the adversary can then deduce the x-key value used in the golden matrix used to encrypt the plaintext. Once the x-key is found, the adversary can then decrypt any cipher matrix encrypted with that x-key back into plaintext.

Chosen-Plaintext

This attack involves carefully selecting a plaintext to encrypt which can then be used to find the x-key by looking at the cipher matrix. Consider the following plaintext matrix M:

$$M = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

After the plaintext is encrypted with an unknown x-key, the attacker receivers two cipher matrices, E(x) and F(x):

$$MQ^{2x} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} cFs(2x+1) & sFs(2x) \\ sFs(2x) & cFs(2x-1) \end{bmatrix} = \begin{bmatrix} cFs(2x+1) & sFs(2x) \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix} = E(x)$$

$$MQ^{2x+1} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} sFs(2x+2) & cFs(2x+1) \\ cFs(2x+1) & sFs(2x) \end{bmatrix} = \begin{bmatrix} sFs(2x+2) & cFs(2x+1) \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{bmatrix} = F(x)$$

Noting that $sFs(2x) = e_{12}$:

$$sFs(2x) = \frac{\varphi^{2x} - \varphi^{-2x}}{\sqrt{5}} = e_{12}$$

$$\varphi^{2x} - \varphi^{-2x} = e_{12}\sqrt{5}$$

$$\varphi^{4x} - 1 = e_{12}\sqrt{5}\varphi^{2x}$$

And letting $u = \varphi^{2x}$:

$$u^2 - e_{12}\sqrt{5}u - 1 = 0$$

$$u = \frac{e_{12}\sqrt{5} \pm \sqrt{5e_{12}^2 + 4}}{2}$$

$$x = \log_\varphi\left(\sqrt{\frac{e_{12}\sqrt{5} + \sqrt{5e_{12}^2 + 4}}{2}}\right)$$[2]

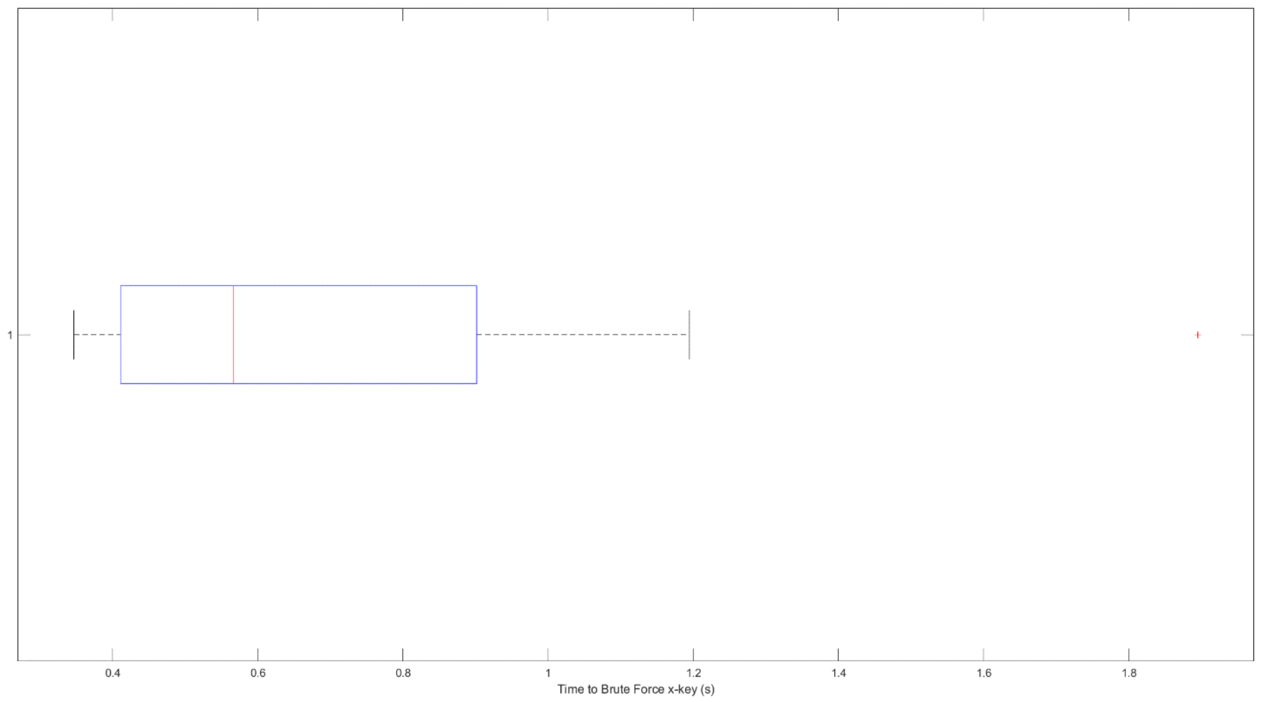
As such, the secret key needed to decrypt enciphered matrices can be found, assuming that the cipher of the plaintext matrix M is known (del Rey & Sanchez, 2008).

Brute Force

This attack involves repeatedly guessing the value of the x-key used to encrypt the plaintext matrix into the encrypted matrix. The ciphered matrix is repeatedly decrypted using different x-keys until the decrypted matrix matches the plaintext matrix associated with the cipher matrix. The name "brute force" comes from the fact that up to hundreds of thousands of guesses can be made per second.

To determine the average time needed to brute force the value of the x-key, I wrote a MATLAB function that, given a plaintext and x-key, will create a cipher matrix using that x-key and then proceed to guess x-key values to decrypt the cipher matrix until the decrypted matrix matches the original plaintext matrix[F]. I then conducted a series of 20 trials, using a different plaintext matrix and x-key for each trial, and used MATLAB's *timeit* function to record how long it took for the true x-key to be guessed for each trial[G]. The resulting timings are below.

[2] We choose the principal square root for $\pm\sqrt{5e_{12}^2 + 4}$ to ensure x is real.



After removing an outlier of 52.62 seconds, I was left with 19 trials with a median of 0.57 seconds required to find the x-key of a golden matrix given a plaintext matrix and its resulting ciphered matrix. The quick speed of x-key cracking was partially due to the fact that I did not need to find the precise x-key value; guessed x-key values within a certain range of the original x-key would be accepted if all the entries of the matrix decrypted using the guessed x-key were within a certain tolerance of all the entries in the original plaintext matrix. The reason this works is because if chosen x-keys x_1, x_2 are approximately equal to each other, $x_1 \approx x_2$, their resulting golden matrices will also be approximately equal, meaning that the resulting cipher matrices encrypted from the same plaintext matrix will also be approximately equal. Consider $x_1 = 5.6788$ and $x_2 = 5.6789$. The resulting even golden matrices are:

$$Q^{2\cdot 5.6788} = \begin{bmatrix} 171.0423 & 105.7073 \\ 105.7073 & 65.3349 \end{bmatrix}, Q^{2\cdot 5.6789} = \begin{bmatrix} 171.0588 & 105.7175 \\ 105.7175 & 65.3412 \end{bmatrix}$$

Encrypting the same plaintext matrix with both golden matrices leads to cipher matrices that are approximately equal.

$$MQ^{2\cdot 5.6788} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 171.0423 & 105.7073 \\ 105.7073 & 65.3349 \end{bmatrix} = \begin{bmatrix} 382.4571 & 236.3773 \\ 935.9565 & 578.4620 \end{bmatrix}$$

$$MQ^{2\cdot 5.6789} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 171.0588 & 105.7175 \\ 105.7175 & 65.3412 \end{bmatrix} = \begin{bmatrix} 382.4939 & 236.4001 \\ 936.0466 & 578.5177 \end{bmatrix}$$

In other words, x-key values close to each other will lead to very similar cipher matrices. This could be exploited even further beyond what my function does; x-key value guesses that lead to decrypted matrices that are reasonably close to the original plaintext matrix can be used to limit the range for future x-key value guesses, resulting in less time required to find the true x-key value. The fact that you can guess an x-key close enough, but not precisely equal to the true x-key, while still being able to decrypt the cipher matrix is another security flaw in this method.

Conclusion

At first glance, the cryptographical method of golden matrices seems quite appealing. The fact that the method is symmetrical in terms of encryption and decryption gives it an upper hand over asymmetrical methods, which have much more complicated encryption and decryption algorithms, making them slower and difficult to use in real time. On top of that, the complexity of asymmetrical algorithms can more easily lead to errors such as roundoff errors, and can be difficult to implement computationally. Overall, the benefits of the golden cryptography method are fast, real-time encryption and decryption as well as ease-of-use and ease to implement computationally.

However, this method is not without serious security flaws. As mentioned previously, the range of usable x-key values is quite narrow, and different but approximately equal x-keys will lead to similar cipher matrices. As such, this makes it very vulnerable to chosen-plaintext attacks (del Rey & Sanchez, 2008) as well as brute force attacks, and so should not be used in the real world to encrypt sensitive information.

Despite this, the paper outlining the method offers some suggestions to improve security (Stakhov, 2007). When transmitting a long message, the message will have to be broken up into blocks of 4 numbers in order to be placed within the 2 x 2 plaintext matrices. By using a different permutation and x-key for the encryption of each matrix, finding the x-key for a particular cipher matrix using a brute force or chosen-plaintext attack will not necessarily lead to the discovery of the other x-keys for the other cipher matrices. The paper specifically proposes using a random number generator to create the x-keys and permutations for each cipher matrix and plaintext matrix respectively (Stakhov, 2007). In order to safely transmit all of the permutations and x-keys to the receiver so that they can decrypt the cipher matrices, the paper proposes using an asymmetrical encryption method to do so (Stakhov, 2007).

If I were to continue exploring this topic, I would first investigate why the relative error between decrypted and plaintext matrices spikes so suddenly for x-key values outside of the interval [-15, 15]. I suspect that it may be occurring due to roundoff error, but perhaps there is something intrinsic about the symmetrical hyperbolic trigonometric functions that causes this. Relatedly, I am also curious as to why the amount of relative error varies between the even and odd golden matrices for certain x-key value ranges.

Additionally, I would like to explore whether this method could be generalized to work on n x n matrices by finding a definition of the golden matrix that works for any n x n size; using larger matrices could make it more difficult to brute force the value of the x-key. Furthermore, I originally planned to explore more methods of matrix cryptography in this paper, but was unable to do so due to time and paper space constraints. I would like to investigate some of these methods in the future to see how their security compares to the golden matrix method.

References

del Rey, A. M., & Sanchez, G. R. (2008). On the Security of "Golden" Cryptography. International Journal of Network Security, 7(3), 448–450. https://doi.org/10.6633/IJNS.200811.7(3).16

The Editors of Encyclopaedia Britannica. (2023, March 14). Cryptography. Encyclopædia Britannica. Retrieved May 1, 2023, from https://www.britannica.com/topic/cryptography

Hoggatt, V. E. (1979). Fibonacci and Lucas Numbers. The Fibonacci Association.

Levet, M. (2021, October 19). CSCI 3104 Algorithms- Lecture Notes. Boulder; Michael Levet.

Merriam-Webster. (n.d.). Cryptography. In Merriam-Webster.com dictionary. Retrieved May 1, 2023, from https://www.merriam-webster.com/dictionary/cryptography

Rawat, S. (2021, October 2). Characteristics, Types and Applications of Cryptography. Analytics Steps. Retrieved May 1, 2023, from https://www.analyticssteps.com/blogs/characteristics-types-and-applications-cryptography

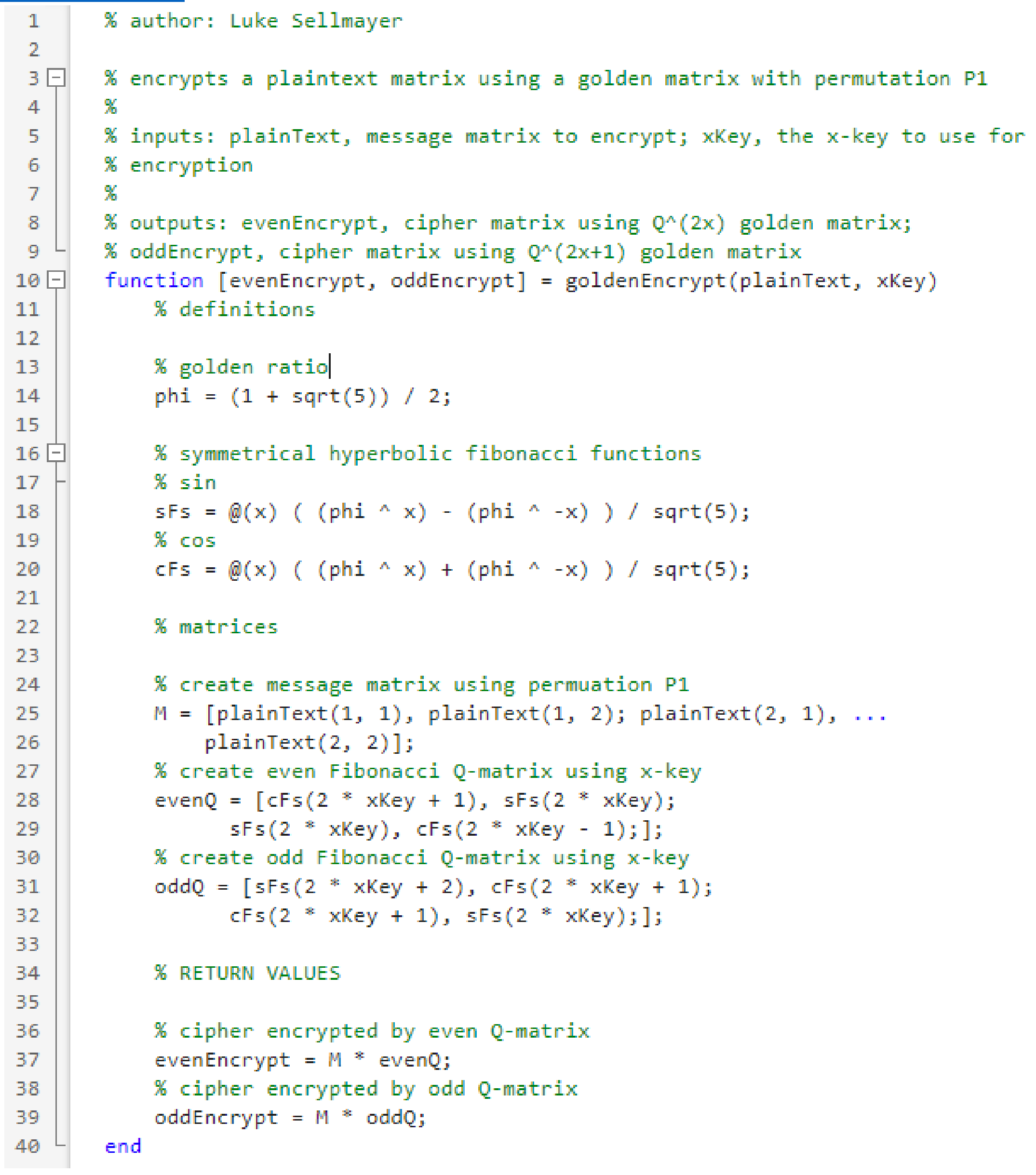
Stakhov, A. P. (2007). The “golden” matrices and a new kind of cryptography. Chaos, Solitons & Fractals, 32(3), 1138–1146. https://doi.org/10.1016/j.chaos.2006.03.069

Stakhov, A., & Rozin, B. (2005). On a new class of hyperbolic functions. Chaos, Solitons & Fractals, 23(2), 379–389. https://doi.org/10.1016/j.chaos.2004.04.022

Taylor, Chris. (2013, July 18). *Matrix multiplication time complexity in MATLAB*. Stack Overflow. https://stackoverflow.com/a/17717767

Appendix

A: goldenEncrypt.m

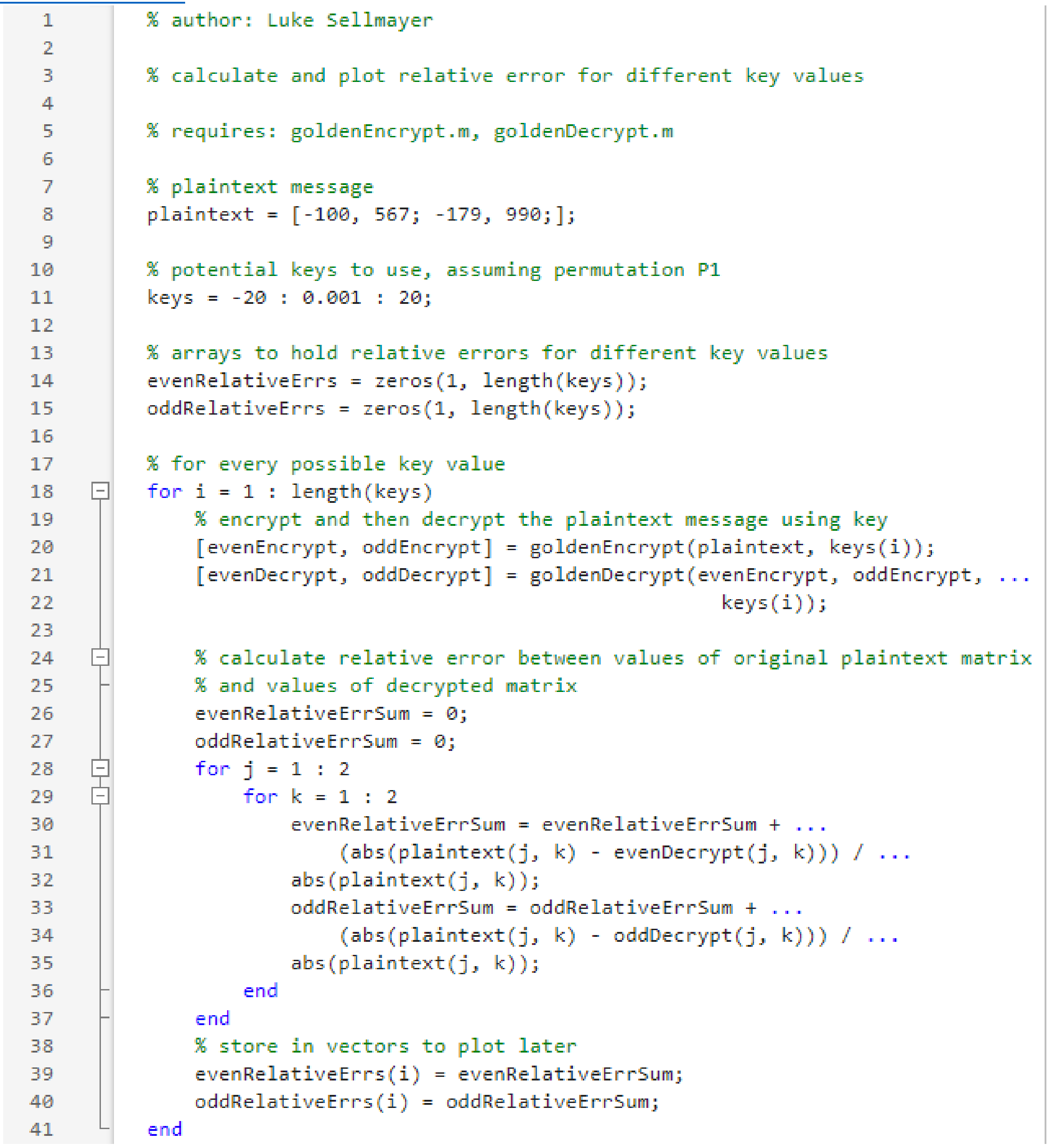


```matlab
% author: Luke Sellmayer

% encrypts a plaintext matrix using a golden matrix with permutation P1
%
% inputs: plainText, message matrix to encrypt; xKey, the x-key to use for
% encryption
%
% outputs: evenEncrypt, cipher matrix using Q^(2x) golden matrix;
% oddEncrypt, cipher matrix using Q^(2x+1) golden matrix
function [evenEncrypt, oddEncrypt] = goldenEncrypt(plainText, xKey)
    % definitions

    % golden ratio
    phi = (1 + sqrt(5)) / 2;

    % symmetrical hyperbolic fibonacci functions
    % sin
    sFs = @(x) ( (phi ^ x) - (phi ^ -x) ) / sqrt(5);
    % cos
    cFs = @(x) ( (phi ^ x) + (phi ^ -x) ) / sqrt(5);

    % matrices

    % create message matrix using permuation P1
    M = [plainText(1, 1), plainText(1, 2); plainText(2, 1), ...
        plainText(2, 2)];
    % create even Fibonacci Q-matrix using x-key
    evenQ = [cFs(2 * xKey + 1), sFs(2 * xKey);
          sFs(2 * xKey), cFs(2 * xKey - 1);];
    % create odd Fibonacci Q-matrix using x-key
    oddQ = [sFs(2 * xKey + 2), cFs(2 * xKey + 1);
          cFs(2 * xKey + 1), sFs(2 * xKey);];

    % RETURN VALUES

    % cipher encrypted by even Q-matrix
    evenEncrypt = M * evenQ;
    % cipher encrypted by odd Q-matrix
    oddEncrypt = M * oddQ;
end
```

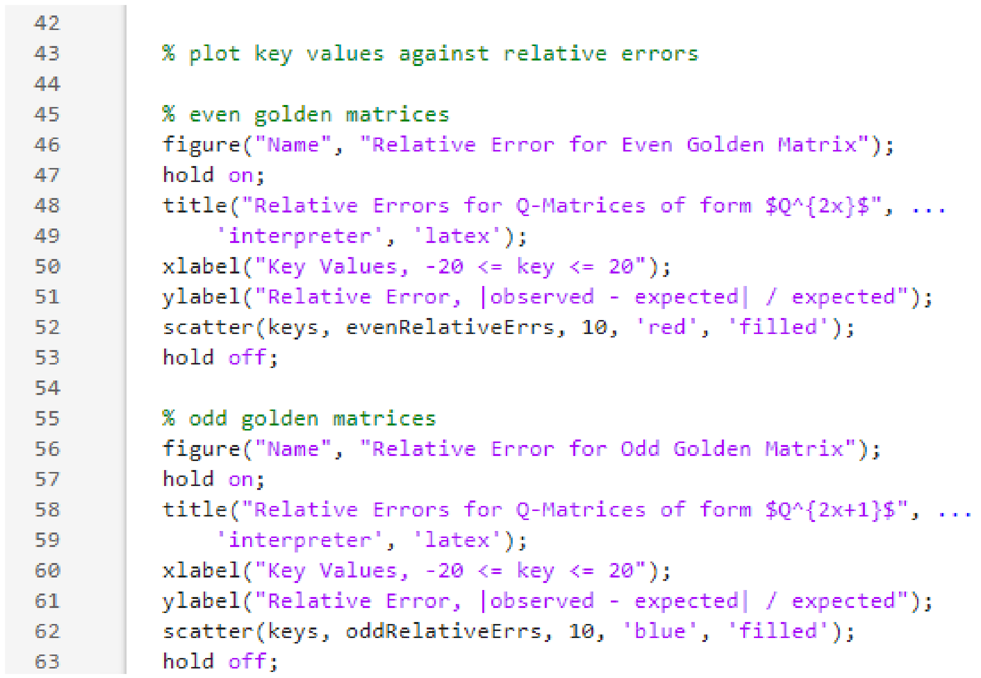
B: goldenDecrypt.m

```matlab
% author: Luke Sellmayer

% decrypts a cipher matrix created from goldenEncrypt back into plaintext
%
% inputs: evenEncrypt, cipher matrix created by Q^(2x) golden matrix;
% oddEncrypt, cipher matrix created by Q^(2x+1) golden matrix; xKey, x-key
% to use to decrypt cipher matrices
%
% outputs: evenDecrypt, plaintext matrix decrypted by Q^(2x) golden matrix;
% oddDecrypt, plaintext matrix decrypted by Q^(2x+1) golden matrix
function [evenDecrypt, oddDecrypt] = goldenDecrypt(evenEncrypt, ...
    oddEncrypt, xKey)
    % definitions

    % golden ratio
    phi = (1 + sqrt(5)) / 2;

    % symmetrical hyperbolic fibonacci functions
    % sin
    sFs = @(x) ( (phi ^ x) - (phi ^ -x) ) / sqrt(5);
    % cos
    cFs = @(x) ( (phi ^ x) + (phi ^ -x) ) / sqrt(5);

    % create inverse even Q-matrix
    inverseEvenQ = [cFs(2 * xKey - 1), -1 * sFs(2 * xKey);
           -1 * sFs(2 * xKey), cFs(2 * xKey + 1);];
    % create inverse odd Q-matrix
    inverseOddQ = [-1 * sFs(2 * xKey), cFs(2 * xKey + 1);
           cFs(2 * xKey + 1), -1 * sFs(2 * xKey + 2);];

    % RETURN VALUES

    % decrypt using even Q-matrix
    evenDecrypt = evenEncrypt * inverseEvenQ;
    % decrypt using odd Q-matrix
    oddDecrypt = oddEncrypt * inverseOddQ;
end
```

C: RelativeError.m



```matlab
% author: Luke Sellmayer

% calculate and plot relative error for different key values

% requires: goldenEncrypt.m, goldenDecrypt.m

% plaintext message
plaintext = [-100, 567; -179, 990;];

% potential keys to use, assuming permutation P1
keys = -20 : 0.001 : 20;

% arrays to hold relative errors for different key values
evenRelativeErrs = zeros(1, length(keys));
oddRelativeErrs = zeros(1, length(keys));

% for every possible key value
for i = 1 : length(keys)
    % encrypt and then decrypt the plaintext message using key
    [evenEncrypt, oddEncrypt] = goldenEncrypt(plaintext, keys(i));
    [evenDecrypt, oddDecrypt] = goldenDecrypt(evenEncrypt, oddEncrypt, ...
                                              keys(i));

    % calculate relative error between values of original plaintext matrix
    % and values of decrypted matrix
    evenRelativeErrSum = 0;
    oddRelativeErrSum = 0;
    for j = 1 : 2
        for k = 1 : 2
            evenRelativeErrSum = evenRelativeErrSum + ...
                (abs(plaintext(j, k) - evenDecrypt(j, k))) / ...
            abs(plaintext(j, k));
            oddRelativeErrSum = oddRelativeErrSum + ...
                (abs(plaintext(j, k) - oddDecrypt(j, k))) / ...
            abs(plaintext(j, k));
        end
    end
    % store in vectors to plot later
    evenRelativeErrs(i) = evenRelativeErrSum;
    oddRelativeErrs(i) = oddRelativeErrSum;
end
```



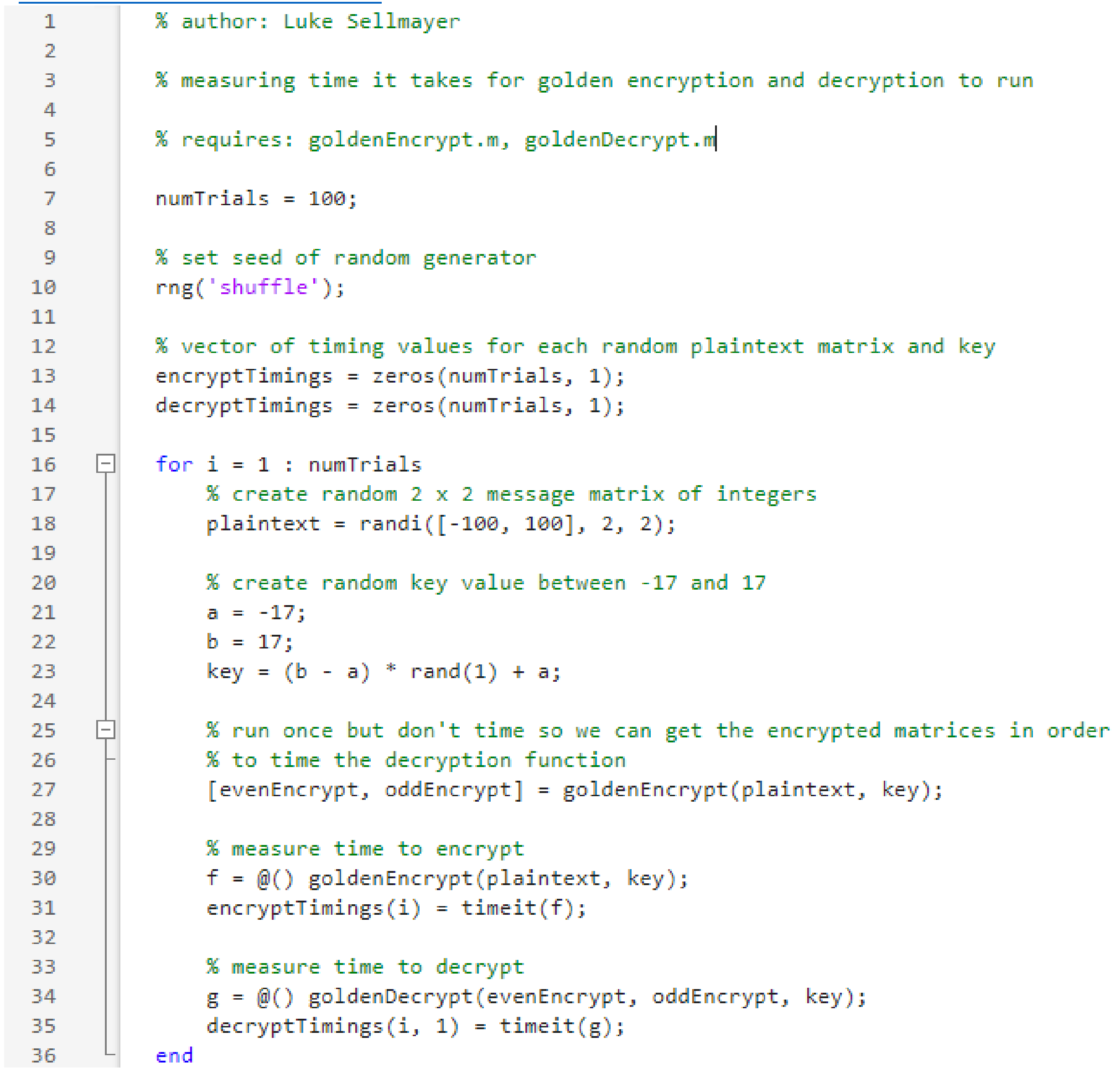
```matlab

    % plot key values against relative errors

    % even golden matrices
    figure("Name", "Relative Error for Even Golden Matrix");
    hold on;
    title("Relative Errors for Q-Matrices of form $Q^{2x}$", ...
        'interpreter', 'latex');
    xlabel("Key Values, -20 <= key <= 20");
    ylabel("Relative Error, |observed - expected| / expected");
    scatter(keys, evenRelativeErrs, 10, 'red', 'filled');
    hold off;

    % odd golden matrices
    figure("Name", "Relative Error for Odd Golden Matrix");
    hold on;
    title("Relative Errors for Q-Matrices of form $Q^{2x+1}$", ...
        'interpreter', 'latex');
    xlabel("Key Values, -20 <= key <= 20");
    ylabel("Relative Error, |observed - expected| / expected");
    scatter(keys, oddRelativeErrs, 10, 'blue', 'filled');
    hold off;
```

D: Runtime Analysis of goldenEncrypt.m and goldenDecrypt.m

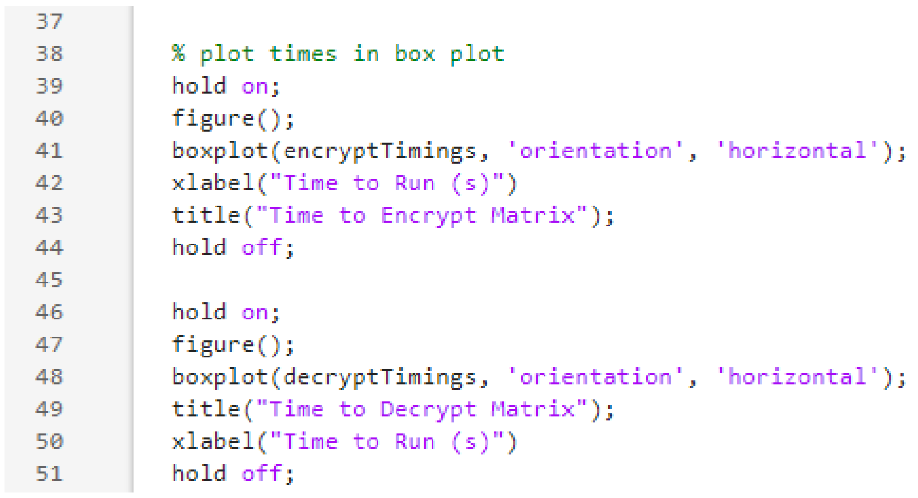
```matlab
% golden encryption function
function [evenEncrypt, oddEncrypt] = goldenEncrypt(message, key)
    % definitions

    % golden ratio
    phi = (1 + sqrt(5)) / 2; → 4

    % symmetrical hyperbolic fibonacci functions
    % sin
    sFs = @(x) ( (phi ^ x) - (phi ^ -x) ) / sqrt(5); → 6
    % cos
    cFs = @(x) ( (phi ^ x) + (phi ^ -x) ) / sqrt(5); → 6

    % matrices

    %4create message matrix using permuation P1
    M = [message(1, 1), message(1, 2); message(2, 1), message(2, 2)]; → 8
    % create even Fibonacci Q-matrix using key
    evenQ = [cFs(2 * key + 1), sFs(2 * key);
          sFs(2 * key), cFs(2 * key - 1);]; → 31
    % create odd Fibonacci Q-matrix using key
    oddQ = [sFs(2 * key + 2), cFs(2 * key + 1);
          cFs(2 * key + 1), sFs(2 * key);]; → 32

    % RETURN VALUES

    % cipher encrypted O(n^3) by even Q-matrix
    evenEncrypt = M * evenQ; → 1 + O(2^3) = 1+O(8)
    % cipher encrypted O(n^3) by odd Q-matrix
    oddEncrypt = M * oddQ; → 1+ O(8)
end
```

```
% golden decryption function
function [evenDecrypt, oddDecrypt] = goldenDecrypt(eCipher, oCipher, key)
    % definitions

    % golden ratio
    phi = (1 + sqrt(5)) / 2; → 4

    % symmetrical hyperbolic fibonacci functions
    % sin
    sFs = @(x) ( (phi ^ x) - (phi ^ -x) ) / sqrt(5); → 6
    % cos
    cFs = @(x) ( (phi ^ x) + (phi ^ -x) ) / sqrt(5); → 6

    % create inverse even Q-matrix
    inverseEvenQ = [cFs(2 * key - 1), -1 * sFs(2 * key);
           -1 * sFs(2 * key), cFs(2 * key + 1);]; → 35
    % create inverse odd Q-matrix
    inverseOddQ = [-1 * sFs(2 * key), cFs(2 * key + 1);
           cFs(2 * key + 1), -1 * sFs(2 * key + 2);]; → 36

    % RETURN VALUES

    % decrypt using even Q-matrix        O(n^3)
    evenDecrypt = eCipher * inverseEvenQ; → 1+O(n^3)
    % decrypt using odd Q-matrix         O(n^3)
    oddDecrypt = oCipher * inverseOddQ; → 1+O(n^3)
end
```

E: RandomizedRuntimeTrials.m

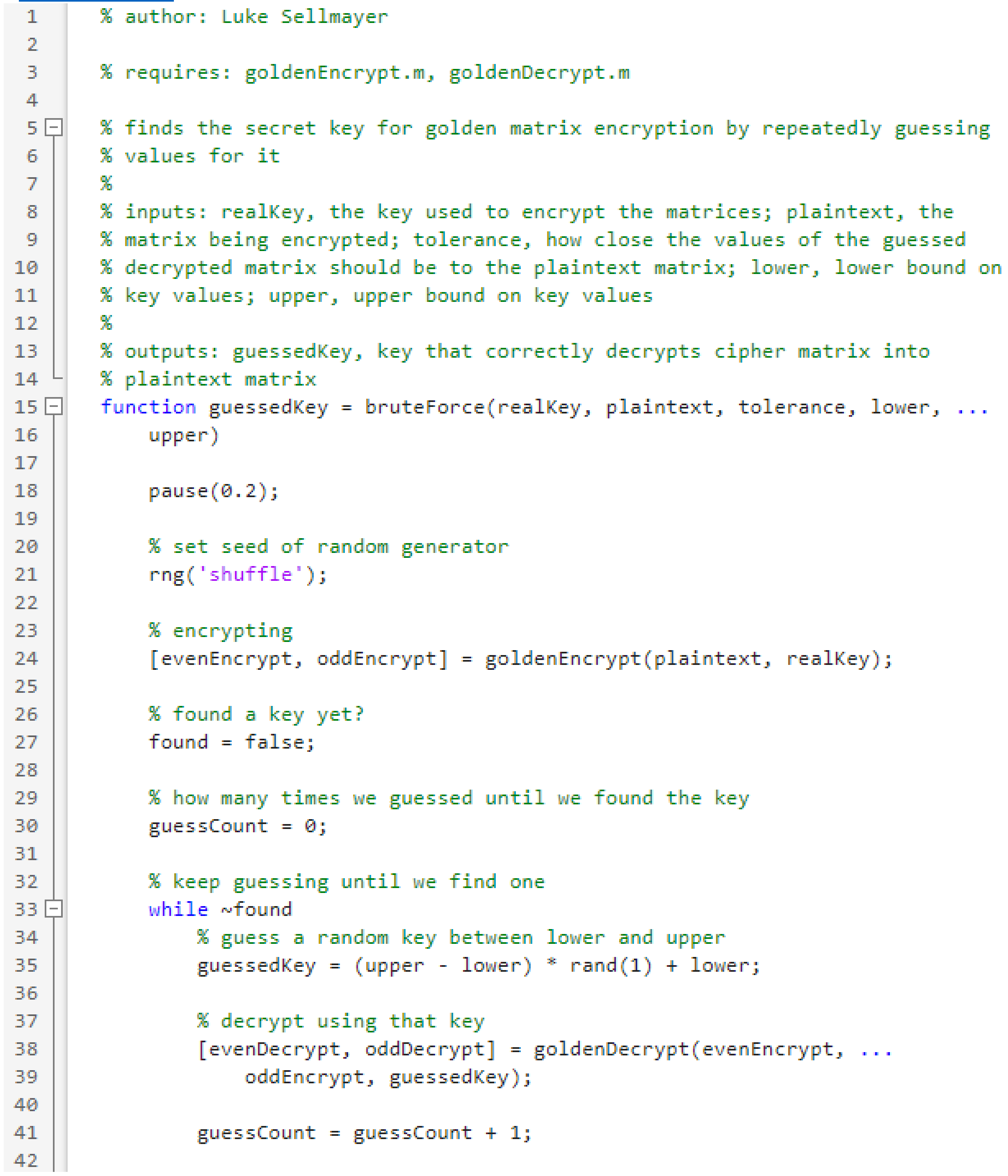


```matlab
% author: Luke Sellmayer

% measuring time it takes for golden encryption and decryption to run

% requires: goldenEncrypt.m, goldenDecrypt.m

numTrials = 100;

% set seed of random generator
rng('shuffle');

% vector of timing values for each random plaintext matrix and key
encryptTimings = zeros(numTrials, 1);
decryptTimings = zeros(numTrials, 1);

for i = 1 : numTrials
    % create random 2 x 2 message matrix of integers
    plaintext = randi([-100, 100], 2, 2);

    % create random key value between -17 and 17
    a = -17;
    b = 17;
    key = (b - a) * rand(1) + a;

    % run once but don't time so we can get the encrypted matrices in order
    % to time the decryption function
    [evenEncrypt, oddEncrypt] = goldenEncrypt(plaintext, key);

    % measure time to encrypt
    f = @() goldenEncrypt(plaintext, key);
    encryptTimings(i) = timeit(f);

    % measure time to decrypt
    g = @() goldenDecrypt(evenEncrypt, oddEncrypt, key);
    decryptTimings(i, 1) = timeit(g);
end
```

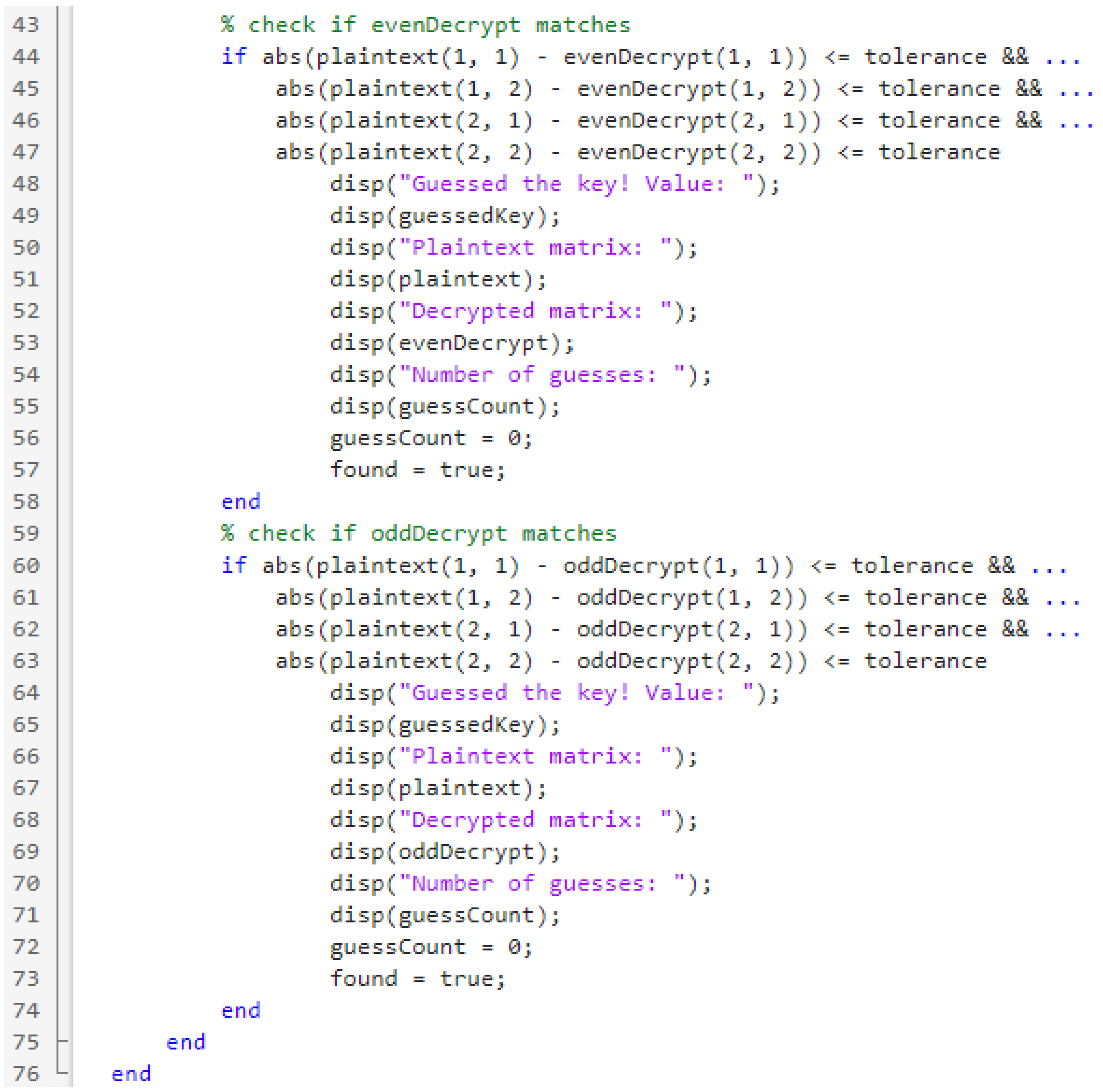


```matlab

% plot times in box plot
hold on;
figure();
boxplot(encryptTimings, 'orientation', 'horizontal');
xlabel("Time to Run (s)")
title("Time to Encrypt Matrix");
hold off;

hold on;
figure();
boxplot(decryptTimings, 'orientation', 'horizontal');
title("Time to Decrypt Matrix");
xlabel("Time to Run (s)")
hold off;
```

F: bruteForce.m

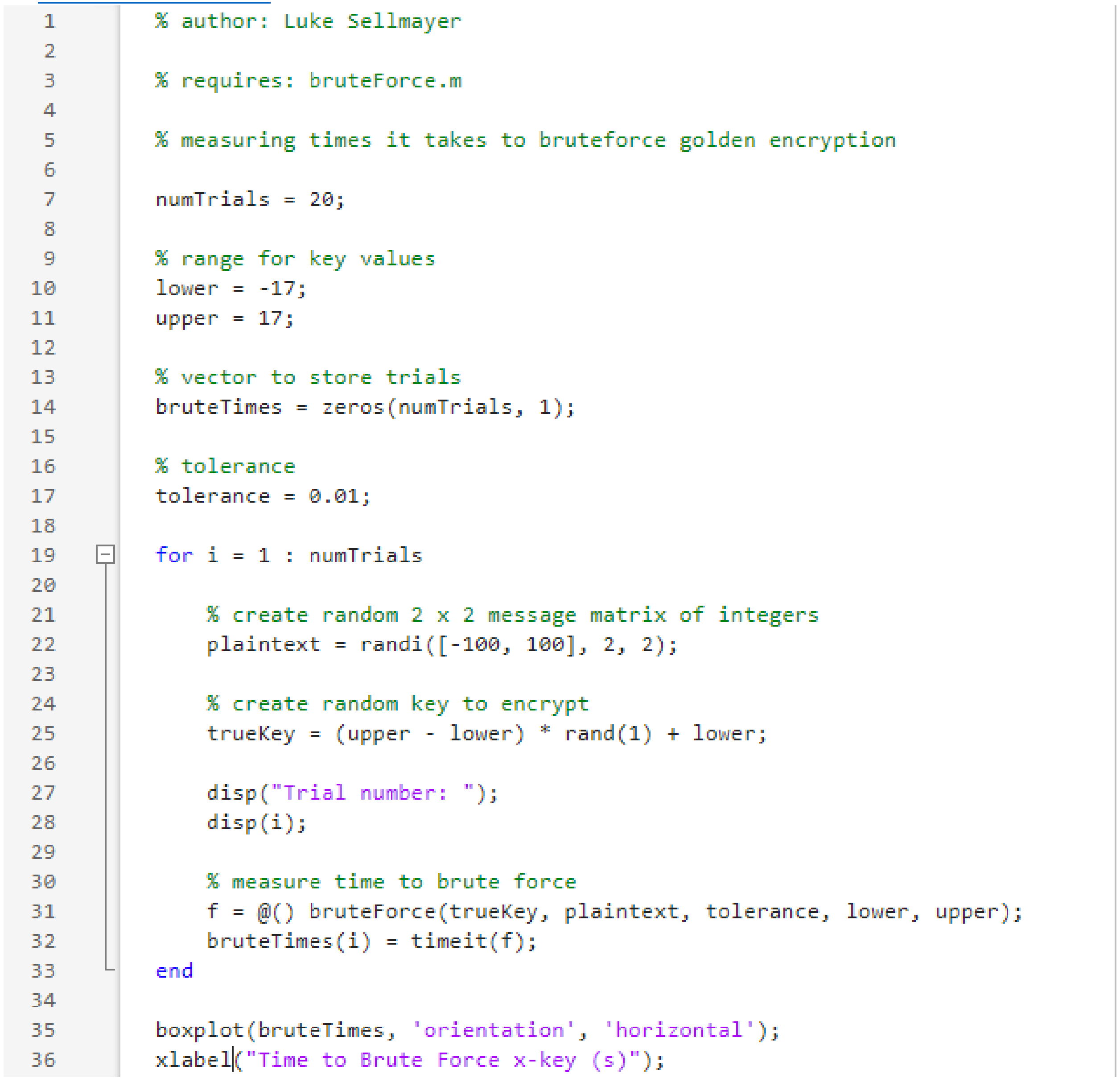


```
% author: Luke Sellmayer

% requires: goldenEncrypt.m, goldenDecrypt.m

% finds the secret key for golden matrix encryption by repeatedly guessing
% values for it
%
% inputs: realKey, the key used to encrypt the matrices; plaintext, the
% matrix being encrypted; tolerance, how close the values of the guessed
% decrypted matrix should be to the plaintext matrix; lower, lower bound on
% key values; upper, upper bound on key values
%
% outputs: guessedKey, key that correctly decrypts cipher matrix into
% plaintext matrix
function guessedKey = bruteForce(realKey, plaintext, tolerance, lower, ...
    upper)

    pause(0.2);

    % set seed of random generator
    rng('shuffle');

    % encrypting
    [evenEncrypt, oddEncrypt] = goldenEncrypt(plaintext, realKey);

    % found a key yet?
    found = false;

    % how many times we guessed until we found the key
    guessCount = 0;

    % keep guessing until we find one
    while ~found
        % guess a random key between lower and upper
        guessedKey = (upper - lower) * rand(1) + lower;

        % decrypt using that key
        [evenDecrypt, oddDecrypt] = goldenDecrypt(evenEncrypt, ...
            oddEncrypt, guessedKey);

        guessCount = guessCount + 1;

```



```
        % check if evenDecrypt matches
        if abs(plaintext(1, 1) - evenDecrypt(1, 1)) <= tolerance && ...
            abs(plaintext(1, 2) - evenDecrypt(1, 2)) <= tolerance && ...
            abs(plaintext(2, 1) - evenDecrypt(2, 1)) <= tolerance && ...
            abs(plaintext(2, 2) - evenDecrypt(2, 2)) <= tolerance
                disp("Guessed the key! Value: ");
                disp(guessedKey);
                disp("Plaintext matrix: ");
                disp(plaintext);
                disp("Decrypted matrix: ");
                disp(evenDecrypt);
                disp("Number of guesses: ");
                disp(guessCount);
                guessCount = 0;
                found = true;
        end
        % check if oddDecrypt matches
        if abs(plaintext(1, 1) - oddDecrypt(1, 1)) <= tolerance && ...
            abs(plaintext(1, 2) - oddDecrypt(1, 2)) <= tolerance && ...
            abs(plaintext(2, 1) - oddDecrypt(2, 1)) <= tolerance && ...
            abs(plaintext(2, 2) - oddDecrypt(2, 2)) <= tolerance
                disp("Guessed the key! Value: ");
                disp(guessedKey);
                disp("Plaintext matrix: ");
                disp(plaintext);
                disp("Decrypted matrix: ");
                disp(oddDecrypt);
                disp("Number of guesses: ");
                disp(guessCount);
                guessCount = 0;
                found = true;
        end
    end
end
```

G: bruteForceTrials.m



```matlab
% author: Luke Sellmayer

% requires: bruteForce.m

% measuring times it takes to bruteforce golden encryption

numTrials = 20;

% range for key values
lower = -17;
upper = 17;

% vector to store trials
bruteTimes = zeros(numTrials, 1);

% tolerance
tolerance = 0.01;

for i = 1 : numTrials

    % create random 2 x 2 message matrix of integers
    plaintext = randi([-100, 100], 2, 2);

    % create random key to encrypt
    trueKey = (upper - lower) * rand(1) + lower;

    disp("Trial number: ");
    disp(i);

    % measure time to brute force
    f = @() bruteForce(trueKey, plaintext, tolerance, lower, upper);
    bruteTimes(i) = timeit(f);
end

boxplot(bruteTimes, 'orientation', 'horizontal');
xlabel("Time to Brute Force x-key (s)");
```