3. Encryption

A *cryptosystem* is a way of encoding and decoding messages so that only certain people are able to read them. This case presents a cryptosystem based on matrix algebra and implemented using Matlab. It is much more secure than simple systems you may have seen, such as replacement of each letter by a different letter. Although it is not secure enough for serious commercial or diplomatic use, it does serve to introduce some features also found in more sophisticated systems.

We'll introduce the cryptosystem by showing how it encrypts and then decrypts a simple 15-character message, stored in a variable `s`:

```
>> s = 'This is a test!'
s =
This is a test!
```

Then we'll suggest some ways in which you can write more general functions for encrypting and decrypting. Finally, we'll use the ideas of this case to introduce the concept of a public-key cryptosystem.

Encrypting a message. The first step of disguising the message is to convert it into an array of numbers. The Matlab function `double` does exactly this when applied to a variable containing the message:

```
>> double (s)
ans =
  Columns 1 through 12
    84  104  105  115   32  105  115   32   97   32  116  101
  Columns 13 through 15
   115  116   33
```

Matlab uses the standard computer character code (the so-called *ASCII* code) to assign a different number to each letter and punctuation mark. A space is also regarded as one of the characters of the message, and is translated — as you can see in our example — to its own number, namely 32.

To apply a matrix-algebra encryption scheme to this numerical form of the message, we begin by arranging the array of numbers into a 3×5 matrix. Matlab's `reshape` function does exactly this:

```
>> nnumb = reshape (double(s), 3, 5)
nnumb =
    84   115   115    32   115
   104    32    32   116   116
   105   105    97   101    33
```

For convenience, we have here stored the resulting matrix in a new variable, `nnumb`.

We now arrive at the crucial step of the encryption, in which we transform

this matrix to another 3×5 matrix. The transformation is effected by multiplying our matrix on the left by a 3×3 matrix. Not any such matrix will do, however. To make decryption possible, we must multiply by a 3×3 matrix whose entries are all integers (whole numbers), and whose inverse has entries that are all integers. For example, the integer matrix

```
>> m = [1 5 3; 2 11 8; 4 24 21]
m =
     1     5     3
     2    11     8
     4    24    21
```

can be seen to meet our requirements for its inverse by applying Matlab's `inv` function:

```
>> inv(m)
ans =
    39   -33     7
   -10     9    -2
     4    -4     1
```

Later in this case, we'll have more to say about how such a matrix can be constructed.

We're now almost ready to multiply by `m`. But because the printable characters have ASCII codes in the range 32 to 126, we must first adjust the 3×5 message matrix by subtracting 32 from every element:

```
>> nnumb-32
ans =
    52    83    83     0    83
    72     0     0    84    84
    73    73    65    69     1
```

Now the matrix must contain values in the range 0 to 94, and it is ready to be transformed by matrix multiplication:

```
>> m*(nnumb-32)
ans =
         631         302         278         627         506
        1480         750         686        1476        1098
        3469        1865        1697        3465        2369
```

This transformed matrix does not contain printable ASCII codes, but we can take care of that via the following further adjustment:

```
>> ncode = mod (m*(nnumb-32), 95) + 32
ncode =
    93    49   120    89    63
    87   117    53    83    85
    81    92   114    77   121
```

Here we have used the `mod` function to divide each element by 95 and keep only the remainder, and then we have added 32 so that the values again range from 32 to 126.

All that remains to be done is to convert these numbers back to an array of characters. We use Matlab's `char` function to convert the numbers to characters, and `reshape` again to put them back into a 1×15 array:

```
>> scode = reshape (char(ncode), 1, 15)
scode =
]WQ1u\x5rYSM?Uy
```

The last line gives the coded message to be sent. Notice that the same letter in two different places in the original message can go into two different letters in the coded message. For example, the `i` in `this` becomes a `Q` while the `i` in `is` becomes a `\`.

Decrypting a message. If you were to receive the message `]WQ1u\x5rYSM?Uy`, you would have to reverse the above steps to decrypt it and recover the original. Fortunately, a way of doing this is not hard to figure out, because the reversed steps of decryption are much the same as the original steps of encryption. The only poentially hard step is the reversal of the multiplication by the matrix `m` — and that is where some elementary linear algebra comes in.

To see how the encryption steps would be reversed, it helps to write them down completely. In words, the steps can be described as follows:

1. Translate the 15-character message to a 3×5 matrix of ASCII character codes.
2. Transform the matrix to an encrypted 3×5 matrix of ASCII character codes, by:
   (a) subtracting 32 from each element of the matrix;
   (b) multiplying the matrix by a given 3×3 matrix;
   (c) reducing each matrix element to its remainder modulo 95;
   (d) adding 32 to each element of the resulting matrix.
3. Translate the encrypted 3×5 matrix of ASCII character codes to an encrypted 15-character message.

If you like descriptions that have a more mathematical flavor, then you might prefer the following:

1. Translate a message string s to a 3×5 matrix A.
2. Use the given 3×3 matrix M to compute the encrypted matrix $A' = M(A - 32) \bmod 95 + 32$.
3. Translate A' to an encrypted string s'.

Either way, these steps can be implemented as three Matlab statements:

```
nnumb = reshape (double(s), 3, 5)
ncode = mod (m*(nnumb-32), 95) + 32
scode = reshape (char(ncode), 1, 15)
```

These are exactly the three assignments that we have already derived in the course of illustrating the encryption procedure.

To reverse the encryption steps, we must first undo step 3, then step 2, then step 1. Observe however that steps 1 and 3 merely translate a message between two representations, one as a string of characters and one as a matrix of numbers. Thus step 1, applied to encrypted string `]WQ1u\x5rYSM?Uy`, undoes step 3. Furthermore step 3, applied to a decrypted matrix, will undo step 1.

It remains for us to say how to reverse step 2. The essential idea here is that, since the given 3×3 matrix has an inverse, the way to undo multiplication by that matrix is simply to multiply by that matrix's inverse. (There are faster ways that we'll get to later, but multiplication by the inverse will do fine for now.) As for the other work of step 2, it turns out that exactly the same operations in the same order will undo the original encoding. This is the one aspect of the decryption procedure whose correctness is a bit tricky to establish, so we won't interrupt the presentation to give the proof here.

In summary, our analysis shows that decryption consists of basically the same operations as encryption, with a few minor changes. We can thus proceed to describe the decryption procedure in much the same terms that we used for the encryption:

1. Translate the 15-character encrypted message to a 3×5 matrix of ASCII character codes.
2. Transform the encrypted matrix back to the original 3×5 matrix of ASCII character codes, by:
   (a) subtracting 32 from each element of the matrix;
   (b) multiplying the matrix by *the inverse of* a given 3×3 matrix;
   (c) reducing each matrix element to its remainder modulo 95;
   (d) adding 32 to each element of the resulting matrix.
3. Translate the original 3×5 matrix of ASCII character codes back to the original 15-character message.

If you prefer the more mathematical description, you can think of the decryption procedure as working in this way:

1. Translate an encoded string s' to a 3×5 matrix A'.
2. Use the given 3×3 matrix M to compute the original matrix $A = M^{-1}(A' - 32) \bmod 95 + 32$.
3. Translate A to the original string s.

With these steps being almost the same as the encryption steps, we can use almost the same Matlab code. The only major changes are to replace the string `s` by the encrypted string `scode` at the beginning, and to multiply by the inverse matrix `inv(m)` rather than by `m`:

```
ncode = reshape (double(scode), 3, 5)
nnumb = mod (inv(m)*(ncode-32), 95) + 32
sorig = reshape (char(nnumb), 1, 15)
```

It remains only to type these statements into Matlab to check that they work. The first one,

```
>> ncode = reshape (double(scode), 3, 5)
ncode =
    93    49   120    89    63
    87   117    53    83    85
    81    92   114    77   121
```

produces the same encoded matrix that we saw before. The second,

```
>> nnumb = mod (inv(m)*(ncode-32), 95) + 32
nnumb =
    93    96    55    41   115
   101    37   117   113   115
   105   102   100   101    33
```

undoes the matrix multiplication to return the same unencoded matrix that we saw before. And the third,

```
>> sorig = reshape (char(nnumb), 1, 15)
sorig =
This is a test!
```

turns the unencoded matrix back into the original string.

M-files for encryption and decryption. It would be a nuisance to have to type all of the Matlab statements we have developed so far, every time a message was to be encrypted or decrypted. You could speed things up by creating a Matlab "M-file" called, say, `encrypt.m`, containing the three encryption statements:

```
nnumb = reshape (double(s), 3, 5);
ncode = mod (m*(nnumb-32), 95) + 32;
scode = reshape (char(ncode), 1, 15)
```

Similarly, an M-file called `decrypt.m` could contain the three decryption statements:

```
ncode = reshape (double(scode), 3, 5);
nnumb = mod (inv(m)*(ncode-32), 95) + 32;
sorig = reshape (char(nnumb), 1, 15)
```

Using these files, our entire previous example looks like this:

```
>> s = 'This is a test!';
>> m = [1 5 3; 2 11 8; 4 24 21];
>> encrypt
scode =
cfu4|oz<%_bqD`3
>> decrypt
sorig =
This is a test!
```

Notice that, to avoid lengthy output, we have terminated most statements with a `;` so as to suppress the output of their results. These M-files are reasonably convenient, but you do have to remember to set up an appropriate matrix in `m` and string in `s` or `scode` before typing the M-file's name to run the encryption or decryption procedure.

A better alternative is to define *function* M-files for the encryption and decryption procedures. You are already familiar with function "calls" to built-in procedures, such as `double(s)`, `mod (m*(nnumb-32), 95)`, and `reshape (char(ncode),1,15)`. Each function takes certain *arguments* — in the comma-separated list within parentheses after the function name — and returns a result value that can be displayed or used in larger expressions. In effect, a function acts as a "black box" that can take Matlab expressions of any complexity as its inputs (arguments) and that computes a corresponding output (result) without requiring the programming to know any of its internal details.

The idea of a function M-file is to define your own functions through M-files that you write. In particular, to implement our cryptosystem we would like to call an encryption function through an expression like

```
encrypt ('This is a test!', m)
```

where the first argument is the string to be encrypted, and the second is the matrix that serves as the key. We would also want to be able to use a decryption function,

```
decrypt ('cfu4|oz<%_bqD`3', m)
```

where the first argument is a string previously encrypted using the given key.

Our previous M-files can be made into function M-files by adding just one statement at the beginning. For example, our encryption M-file, `encrypt.m`, becomes the following function M-file:

```
function scode = encrypt (s, m)
   nnumb = reshape (double(s), 3, 5);
   ncode = mod (m*(nnumb-32), 95) + 32;
   scode = reshape (char(ncode), 1, 15)
```

The word `function` identifies this to Matlab as an M-function. It is followed by the name of the variable, `scode`, that represents the result value. Then, after the `=` sign, comes the name of the function, `encrypt`. At the end, in parentheses, are the names of the variables, `s` and `m`, for the function's arguments. When the function is called, the actual value of its first argument is assigned to `s` and the actual value of its second argument is assigned to `m`. Then the remaining lines of the M-file are executed in the usual way. Finally, the value of the result variable `scode` at the end of execution is passed back as the result of the function call.

A similar line at the beginning of `decrypt.m` makes it into a function M-file:

```
function sorig = decrypt (scode, m)
   ncode = reshape (double(scode), 3, 5);
   nnumb = mod (inv(m)*(ncode-32), 95) + 32;
   sorig = reshape (char(nnumb), 1, 15)
```

Here's an example of how these functions could be used. The person sending the message could compute the coded string and store it in a variable `msgstring` as follows:

```
>> m = [1 5 3; 2 11 8; 4 24 21];
>> msgstring = encrypt ('This is a test!', m)
msgstring =
cfu4|oz<%_bqD`3
```

The recipient of the coded string might decode it like this:

```
>> msgstring = 'cfu4|oz<%_bqD`3';
>> decrypt (msgstring, [1 5 3; 2 11 8; 4 24 21])
ans =
This is a test!
```

As you can see, it is up to the user of a function to decide how to express its argument values, and how to handle its result value. The user does not need to know the names of the variables the represent the arguments and result within the function M-file.

Encryption keys. Our cryptosystem requires an integer matrix `m` whose inverse is also an integer matrix. In general, there is no easy way to identify matrices that have this property. Certain simple matrices having the property are easy to construct, however. You would not want to use such matrices as encryption keys, because the resulting coded messages might be too easy for unintended recipients to "crack" by guessing the key. However, you might use a product of such matrices. Specifically, if M_1, M_1^{-1} and M_2, M_2^{-1} are all integer, then $M_1 M_2$ and $(M_1 M_2)^{-1} = M_2^{-1} M_1^{-1}$ must also be integer, because matrix multiplication involves only the addition and multiplication of the integer elements.

One easy choice for M_1 is to make it all integers below the diagonal, 1's on the diagonal, and 0's above the diagonal. For example:

```
>> m1 = [1,0,0; 2,1,0; 4,4,1]
m1 =
     1     0     0
     2     1     0
     4     4     1
```

M_2 could be defined similarly, but with the 0's below the diagonal and the integers above:

```
>> m2 = [1,5,3; 0,1,2; 0,0,1]
m2 =
     1     5     3
     0     1     2
     0     0     1
```

Lower-triangular and *upper-triangular* matrices of these kinds are easily shown to have integer inverses, while their product is more general in nature:

```
>> m = m1 * m2
m =
     1     5     3
     2    11     8
     4    24    21
```

This is in fact how `m` was determined for use in our example.

Suppose now that you want to apply these ideas to permit someone to send you coded messages that no one else can read. You would first pick any two triangular matrices like `m1` and `m2` and multiply them to get `m`. Then you would give `m` together with `encrypt.m` to your correspondent. The two of you would then go through the following steps in sending each message:

1. The other person executes *code* `= encrypt(m,'`*text*`')`, where *text* is the message written as a character string.
2. The other person sends you *code,* the resulting encoded string, by any reliable means.
3. You execute *text* `= decrypt(m,'`*code*`')` to recover the string *text* that contains the original message.

This works because you have a copy of the key that was used to encrypt the message. Others could intercept the message, but without the key they would have a much harder time decrypting it.

The weakness of this scheme lies in having to give your correspondent the key. There is a danger that the key will be intercepted and used by people from whom you are trying to hide your messages. You can reduce this danger by changing keys frequently; you might also want to translate (or *compile*) your encryption M-file into a form that is harder for others to understand. Even so, there is some danger in this approach, as there are many examples of hidden-key cryptosystems that have been broken.

As an alternative, you could make your key `m` public, and invite anyone to send you messages encrypted using that key. For security, you could rely on the difficulty of decryption. The most time-consuming step in encryption or decryption is computing `inv(m)*(ncode-32)` in `decrypt.m`, but this work can be greatly reduced if the triangular factors `m1` and `m2` are known. Specifically, for a matrix of n rows and columns, knowing the triangular factors reduces the work from a small multiple of n^3 to a small multiple of n^2. Thus, if you could use a large enough key matrix, it would not matter that you made it public, as long as you kept the original factors `m1` and `m2` to yourself. Others could use the public key to efficiently encrypt messages for you, but using the same key to decrypt your messages might take so much time as to be impractical. Only you would have the factors that would make decryption fast.

This kind of *public-key cryptosystem* could be defeated if the factors `m1` and `m2` were easy to compute from `m`. But the work of computing the factors turns out to be roughly the same — a small multiple of n^3 — as the work of computing `inv(m)` or `inv(m)*(ncode-32)`. Encryption thus remains much more difficult for people who know only the key than for the person who formed the key from its factors.

Given the speed and sophistication of current computers and software, a public-key cryptosystem based on triangular matrix factorizations is not really so secure. There are practical cryptosystems that rely on the same principles, however. They use a key that is the product of two very large prime* numbers; their coding scheme can efficiently encrypt a message using just the key, but can efficiently decrypt the message only through a knowledge of the prime factors. They rely on the fact that finding and multiplying two large primes is much easier than recovering the two prime factors given only their product.

Enhancements. All of our examples have assumed a 15-character message and a 3×3 key matrix. There are a number of enhancements that offer good practice in the programming of M-files.

Most obviously, it would be desirable to have `encode` and `decode` functions that take as their arguments strings of any length, and key matrices of any dimension. Matlab's `size` function enables function M-files to determine the sizes of their arguments.

A function M-file for choosing keys would also be useful. It could randomly generate the below-diagonal elements of a lower-triangular matrix and the above-diagonal elements of an upper-triangular matrix, then multiply them and return the result.

Because Matlab uses `'` characters to delimit strings, any encrypted string that contains a `'` character will be hard to deal with. The encryption and decryption functions could be enhanced to circumvent this problem.

Finally, short messages may be more vulnerable to unauthorized decryption than longer ones. To avoid short messages, the encryption function could be enhanced to automatically inflate the size of messages before encrypting them. The decryption function would then require a corresponding extention to deflate the message after decrypting it.

*A prime number is evenly divisible only by itself and 1.