1 Matrices

Matrices are covered in Section 2.4.2 of our DLN textbook, but here's the short version. A *matrix* is a rectangular grid of numbers. A matrix with p rows and q columns is said to be a $p \times q$ matrix. For example, here's a 2×3 matrix:

$$\begin{bmatrix} 5 & -1.2 & 0 \\ 7 & 7 & 2.5 \end{bmatrix}.$$

If M is a $p \times q$ matrix, then for any i and j (satisyfing $1 \leq i \leq p$ and $1 \leq j \leq q$), M_{ij} denotes the number in the ith row (counting from the top) and jth column (counting from the left) of M. There are three basic operations on matrices:

- If M is a $p \times q$ matrix and c is a number, then there is a *scalar product* matrix cM, which is also $p \times q$, defined by multiplying each entry of M by c. In other words,

$$(cM)_{ij} = cM_{ij}.$$

- If M and N are both $p \times q$ matrices, then there is a *sum* matrix $M + N$, which is also $p \times q$, defined by adding the corresponding entries of M and N. That is,

$$(M + N)_{ij} = M_{ij} + N_{ij}.$$

  Notice that the dimensions p and q of the two matrices must match, for their sum to be defined.

- If M is a $p \times q$ matrix and N is a $q \times r$ matrix, then there is a *product* matrix MN, which is $p \times r$, defined by

$$(MN)_{ij} = \sum_{k=1}^{q} M_{ik} N_{kj}.$$

  Here's another way to think of it. The (i, j)th entry of MN is what you get by multiplying the ith row of M by the jth column of N, entry by entry, and then summing up those products. Notice that the dimensions of the two matrices must match in a particular way, for their product to be defined.

There are two special matrices, denoted O and I, that play roles similar to those of 0 and 1 in the real numbers. For any p and q, O is the $p \times q$ *zero matrix*, consisting entirely of zeros. (There is an O for each combination of p and q. When we see O, we figure out p and q from context.) For any n, I is the $n \times n$ *identity matrix*, defined by $I_{ij} = 1$ if $i = j$ and $I_{ij} = 0$ if $i \neq j$. (There is an I for each n.) For example, the 3×3 identity matrix is

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Matrices enjoy many algebraic properties like those of the real numbers. Here are a few that we'll need later on.

Question A: Let N be any $q \times r$ matrix. Then $ON = O = NO$. Why? And what exactly does O mean in each part of this equation?

Problem B: Let L be a $p \times q$ matrix and M and N be $q \times r$ matrices. Prove the distributive law: $L(M + N) = LM + LN$.
Problem C: Let L be a $p \times q$ matrix, M a $q \times r$ matrix, and N an $r \times s$ matrix. Prove the associative law: $L(MN) = (LM)N$.

2 Matrices in Python

We'll be working in Python, representing matrices as lists of lists, where each sublist is a row (and all of the sublists have the same length). For example, here's a 2×3 matrix:

```
myMatrix = [[5, -1.2, 0], [7, 7, 2.5]]
```

Here's the same matrix, but defined using tuples instead of lists. Whereas lists can be modified — they are *mutable* — tuples are *immutable*. Using immutable data structures, where appropriate, can help us avoid certain programming errors.

```
myMatrix = ((5, -1.2, 0), (7, 7, 2.5))
```

Now let's establish some basic matrix operations. All of our basic functions will observe this convention: They do not alter their input. This prevents a lot of errors, in which a user programmer inadvertently alters a list. It also lets our functions work with tuples. The `printMatrix` function pretty-prints a matrix, with a little space between the entries.

```
def printMatrix(m):
    for i in range(len(m)):
        for j in range(len(m[i])):
            print m[i][j], " ",
        print
```

Here are two implementations of the matrix scalar product. The second one uses a Python feature called *list comprehension*, which builds a list from another list using very little code.

```
def scalarProductMatrix(c, m):
    res = []
    for i in range(len(m)):
        row = []
        for j in range(len(m[i])):
            row.append(c * m[i][j])
        res.append(row)
    return res

def scalarProductMatrix(c, m):
    return [[c * mij for mij in row] for row in m]
```

Question D: What's wrong with this third implementation?

```
def scalarProductMatrix(c, m):
    for i in range(len(m)):
        for j in range(len(m[i])):
            m[i][j] *= c
    return m
```

Similarly, here are two implementations of the matrix sum.

```
def sumMatrix(m, n):
    res = []
    for i in range(len(m)):
        row = []
        for j in range(len(m[i])):
            row.append(m[i][j] + n[i][j])
        res.append(row)
    return res
```

```
def sumMatrix(m, n):
    return [[m[i][j] + n[i][j] for j in range(len(m[0]))] for i in range(len(m))]
```

Problem E: Write a `productMatrix` function, to compute the product of two matrices. Use any style you want, as long as it's correct. (My solution uses the terse style and the Python `sum` function, which computes the sum of a list of numbers.)

Problem F: Matrix multiplication is not commutative; order matters. To prove this, find two 2×2 matrices M and N such that $MN \neq NM$.

Question G: What do you get when you multiply the $n \times n$ identity matrix I by an $n \times p$ matrix N?

3 Setting up the (7, 4) Hamming code

Now we'll study the classic $(7, 4)$ *Hamming code*, which is used to correct errors in the transmission and storage of data. This material is discussed, in a slightly different way, in Section 4.7 of our DLN textbook.

Henceforth we're going to work with matrices of 0s and 1s, and we're going to work modulo 2. That is, whenever we compute a new matrix, we will replace each even number with 0 and each odd number with 1. The algebraic properties (distributivity, associativity, identity, etc.) that you explored earlier will still hold. In Python, we'll just follow up each basic matrix operation (`productMatrix`, etc.) with a call to this function:

Problem H: Write a function `mod2Matrix`, that takes a matrix as input, and returns the matrix reduced modulo 2. As always, the function should not alter the input matrix.

Define the *encoder*, *decoder*, and *checker* matrices E, D, and C like this:

$$E = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \; D = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \; C = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

Before we go any further, inspect these matrices. They are not random gibberish; they contain patterns. Parts of E and D look like matrices that you've seen already. Which ones? And look at the columns of C, from right to left. Can you see the pattern in them?

Question I: Which products of E, D, and C are well-defined? (You should find three.) What are they? What patterns do you observe in them?

4 Transmission with zero errors

Suppose that you want to send your friend the message 1101. You place this bit string into a 4×1 matrix called M:

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

Then you encode the message, by computing EM. (Is this a well-defined matrix product? What are its dimensions?) You send that bit string EM to your friend, over the Internet or a competing communications network (word of mouth? trained cephalopods?).

When she receives your transmission EM, she multiplies it by the checker matrix C. So she now has the matrix $C(EM)$, which equals $(CE)M$ by associativity. What are the dimensions of this matrix? What does it look like?

Then your friend multiplies your transmission EM by the decoder matrix D. So she now has the matrix $D(EM) = (DE)M$. What are its dimensions? What does it look like?

Try this entire process with various initial messages, other than 1101, until you can solve this problem:

Problem J: Summarize this section: "If no errors occur in transmission, then $C(EM)$ will always be..., because.... $D(EM)$ will always be..., because..."

5 Transmission with up to one error

Suppose again that you want to send your friend the message 1101. So you put it into a matrix M, compute the matrix EM, and send EM to her.

Unfortunately, communications networks such as the Internet are noisy, due to equipment faults, electrical disturbances, etc. Sometime between when you send EM and when your friend receives it, the

first bit gets flipped. That is, your friend receives the message $EM + N$, where N is the noise matrix

$$N = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Remember that we're working with matrices modulo 2. When we add EM to N, we reduce our answer modulo 2. If the first bit of EM is 1, then the first bit of $EM + N$ is 0, and *vice-versa*. So adding N to EM really is flipping the first bit.

Your friend receives $EM+N$. She multiplies it by the checker matrix C. So now she has $C(EM+N) = (CE)M + CN$, by distributivity and associativity. What does this look like?

By inspecting $C(EM + N)$, your friend infers (magically?) that she must flip the first bit back. She does this by adding N to $EM + N$. The result is EM. Why? Now that your friend has EM, she multiplies it by the decoder matrix D, to obtain $(DE)M$ — which is what, again?

Try this entire process, flipping the second bit instead of the first, flipping the third bit instead of the first, and so forth. This procedure can be used to detect and correct any single-bit error in transmission. You just have to figure out how the checking result $C(EM + N)$ tells us which bit was flipped.

Problem K: Describe the algorithm for detecting and correcting errors, under the assumption that no more than one error has occurred.

6 Transmission with up to two errors

Suppose again that you want to send your friend the message 1101. So you put it into a matrix M, compute the matrix EM, and send EM to her. But this time, two errors occur in transmission. She receives $EM + N + P$, where

$$N = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \; P = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

She multiplies it by the checker matrix C, to obtain $C(EM + N + P) = (CE)M + CN + CP$.

Question L: If your friend follows through with her correction algorithm from the previous section, then what happens?

There's nothing special about the first two bits in the bit string. Play around with other combinations of two errors, until you are sure that you understand what's happening.

7 Two separate uses for the $(7, 4)$ Hamming code

Problem M: Explain: If one or two errors occur, then the checking process cannot produce the 3×1 zero matrix. If zero or three errors occur, then the checking process may produce the zero matrix.

Once you have completed this problem, you should understand this summary: The Hamming code can be used in two distinct ways. On noisy communications lines, it can be used to detect (but not correct) up to two errors. On communications lines that are known to be only slightly noisy, so that only one error is likely to occur in a seven-bit transmission, the Hamming code can be used to detect and correct one error. The Hamming code cannot be used for both purposes at the same time. That is, it cannot detect up to two errors and correct one of those errors. Also, it cannot be used to detect three or more errors.

8 Other Hamming codes

The Hamming code that we've discussed so far is the $(7, 4)$ Hamming code, meaning that it encodes any 4-bit message into a 7-bit codeword. The number 7 here is significant, in that it is the largest (unsigned) integer that can be expressed using 3 bits. That is, the 3-bit check result can encode any integer from 0 to 7. A check result of 0 indicates "no error"; any other check result indicates an error in a certain bit, from bit 1 to bit 7.

Now we are prepared to understand the next Hamming code. This code uses 4 check bits instead of 3. These check bits can encode any integer from 0 to 15. Hence they can check 15-bit code words. In these 15 bits, 4 bits must be devoted to the check bits, so there are 11 bits for storing the actual message. In short, this Hamming code encodes any 11-bit message into a 15-bit codeword. It is called the $(15, 11)$ Hamming code.

Problem N: What are the dimensions of the E, D, and C matrices in the $(15, 11)$ Hamming code? Write the D and C matrices explicitly (but not the E matrix). Which products of E, D, and C are defined? What are the dimensions of those products? What relationships must those products satisfy?

More generally, for any $n \geq 2$ there is a Hamming code, that uses codewords of length $2^n - 1$, containing n check bits, to encode messages of length $2^n - 1 - n$.