1 Matrices

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

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

We locate elements of a matrix by the row and column in which they fall. The top row of a $p \times q$ matrix is row 0; the bottom row is row $p-1$. Similarly, the left column of a $p \times q$ matrix is column 0; the right column is column $q-1$. (The convention among mathematicians, and in your textbook, is to number starting from 1 instead. Many programming languages, including Python, number starting from 0, so that's the convention we adopt here. Starting from 0 makes the mathematical notation uglier, but your code cleaner.)

If M is a $p \times q$ matrix, then for any i and j (satisyfing $0 \leq i \leq p-1$ and $0 \leq j \leq q-q$), M_{ij} denotes the number in row i (starting from zero, remember) and column j of M. There are three basic operations on matrices:

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

  $$(cM)_{ij} = c \cdot M_{ij}.$$

- **Addition:** If M and N are both $p \times q$ matrices, then the *sum* $M + N$ is a $p \times q$ matrix, 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 in order for their sum to be defined.

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

  $$(MN)_{ij} = \sum_{k=0}^{q-1} M_{ik} N_{kj}.$$

  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.

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

- Let L be a $p \times q$ matrix and M and N be $q \times r$ matrices. Then $L(M + N) = LM + LN$.

  Question A: Why? Sketch a proof to convince yourself of this observation. Here's a start: Let $T = M + N$, and $V = LT = L(M + N)$. What is T_{ij}? What is V_{ij}?

  This observation tells us that *matrix left-multiplication is distributive over matrix addition.*

- Let L be a $p \times q$ matrix, M a $q \times r$ matrix, and N an $r \times s$ matrix. Then $L(MN) = (LM)N$.

  Question B: Why? Sketch a proof to convince yourself of this observation. Here's a start: Let $T = MN$, and $V = LM$, $X = LT = L(MN)$, and $Y = VN = (LM)N$. What is T_{ij}? What is V_{ij}? What are X_{ij} and Y_{ij}? (See below for a couple lemmas that may be helpful.)

  This observation tells us that *matrix multiplication is associative.*

Here are a couple properties of summations that can be useful in proving things about matrix multiplication:

- **Lemma 1:**

$$k\sum_i x_i = \sum_i kx_i.$$

That is, if you have a summation over some index (here i; the range of the summation doesn't matter) and a scalar multiplicative factor that is constant with respect to the change of that index (as k is above), it can go inside or outside the summation. There's nothing mysterious about this; the above statement is just a more compact and formal way of writing the following:

$$k(x_0 + x_1 + x_2 + \ldots) = kx_0 + kx_1 + kx_2 + \ldots$$

This is obviously true, due to the distributive property of scalar multiplication over scalar addition.

- **Lemma 2:**

$$\begin{aligned}\sum_i x_i\Big(\sum_j y_j\Big) &= \sum_i \Big(\sum_j x_i y_j\Big) \\ &= \sum_j \Big(\sum_i x_i y_j\Big) = \sum_j y_j\Big(\sum_i x_i\Big)\end{aligned}$$

That is, if you have nested summations of products of things that are entirely independent of each other's summation index, then you can switch the order of the summation at will. The equality in the first line is justified by Lemma 1 above: x_i is a constant with respect to the changing inner index j. Similarly on the second line, y_j is a constant with respect to the changing inner index i. The equality between the lines is just the observation that if nothing is between the summation symbols, then you can change the order of the summations.

Notice I said "left-multiplication" up in the observation about distributivity? What was that about? Well, there's one property of arithmetic over the real numbers that matrices *don't* have: *matrix multiplication is not commutative* — MN and NM may be two entirely different matrices. We'll investigate this more later, but for right now, we'll consider an easy case:

- Let L be a $p \times q$ matrix, and M a $q \times p$ matrix, where $p \neq q$. Then $LM \neq ML$.

  Question C: Why not?

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

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

- Let N be any $q \times r$ matrix. Then $ON = O = NO$.

  Question D: Why? Prove it from the definitions. And what is the specific matrix that O represents in each part of this equation?

  The way that O behaves should remind you of a problem from one of the early problem sets. O is the *zero of the matrix multiplication operation*, or the *multiplicative zero for matrices.*

- Let N be any $q \times r$ matrix. Then $IN = N = NI$.

  Question E: Why? Prove it from the definitions. And what is the specific matrix that I represents in each part of this equation?

  The way that I behaves should also be familiar. I is the *identity of the matrix multiplication operation*, or the *multiplicative identity for matrices*.

2 Matrices in Python, with Numpy

Today we'll use Python to explore matrix computation, and how it fits with Hamming codes.

1. Open up the text editor of your choice, enter the line `print "Hello world!"` in a new file, and save it as `lab.py` in some convenient working location (perhaps the Desktop?).

2. Open two terminal windows, and navigate to your working directory in both. Put the terminal windows on the left side of your screen, one in the upper left corner, and one in the lower left.

3. **Do not skip this step!**

   - Run the command `pwd` in both terminals. Confirm that you get the same output from each.
   - Run the command `ls` in both terminals. Confirm that you see `lab.py` in the output from each.
   - Run the command `python lab.py` in both terminals. Confirm that you see `Hello world!` as the output both times.

2.1 Working with the Interactive Interpreter

1. In the upper terminal, run the command `python` with no arguments. You should get something like:

```
Enthought Canopy Python 2.7.3 | 64-bit | (default, Dec  2 2013, 16:19:29)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

   If you've never seen this before, welcome to the Python interactive interpreter! You can enter short snippets of Python code here, and the interpreter will run them. It's like entering a program line-by-line. The "`>>> `" is your *prompt*, which tells you that the interpreter is ready for you to enter some code. Try duplicating this session:

```
>>> x = 5
>>> y = 6
>>> z = x + y
>>> print z
11
```

   The truly special feature about the interactive interpreter is this: if you enter an *expression*, rather than a *statement*, the interpreter evaluates your expression and then shows you the value. Behold!

```
>>> 3 + 7
10
>>> 3.0 / 7.0
0.42857142857142855
>>> x + 4
9
>>> x + 4 + y
15
>>> y
6
```

This lets you investigate the values of variables while you're in the middle of thinking through your code. It also lets you test out code quickly without needing to write it in a file, save it, and run it from the command line.

2. Okay, now we're ready to work with matrices, using the `numpy` (*num*erical *Py*thon) module. Duplicate the following in your interactive interpreter (type it in by hand; don't just copy and paste!):

```
>>> from numpy import *
>>> M = array([[5, -1.2, 0], [7, 7, 2.5]])
>>> M
array([[ 5. , -1.2,  0. ],
       [ 7. ,  7. ,  2.5]])
```

Here we create a matrix M that duplicates the example from the first page. Notice how this happens: we pass a list of lists as the argument to the constructor for the `array` class, which is defined in `numpy` module. (An array is basically the same as a matrix.) Each inner list is a row of the matrix we want to make. Note that the inner lists have to be the same length; if they differ, then you'll get an error.

When the interpreter shows you the value of an `array` object, it prints it out one row at a time, (kinda) similar to how you would write it on paper.

3. Let's look more closely at that matrix, and what we can do with it.

   - We can find out the number of rows and columns, and get the value at a particular index.

     ```
     >>> M.shape
     (2, 3)
     >>> M[0,0]
     5.0
     >>> M[1,2]
     2.5
     ```

   - We can also use the indexing notation to get a whole row or a whole column. Rather than entering a specific index, we can instead put in a `:`, which means "all possible indices".

     ```
     >>> M[0,:]
     array([ 5. , -1.2,  0. ])
     >>> M[:,1]
     array([-1.2,  7. ])
     ```

```
>>> M[:,:]
array([[ 5. , -1.2,  0. ],
       [ 7. ,  7. ,  2.5]])
```

Notice that when we give a specific row index, but `:` as the column index, we get every entry with that row index, and any column index. In other words, we get the entire row! Ditto for entering `:` as the row index, and specifying a column index. If we put in `:` for both indices, then we get back the whole matrix. (This last thing is rarely necessary; if we wanted the whole matrix, we could just write `M`. But it's important to note that `:` doesn't mean anything more than "all possible indices".)

- We can take the dot product of two vectors, using the `dot()` method. Remember, the dot product of two vectors is the sum of the products of their corresponding entries. So the lengths of the vectors must match in order to take their dot product.

```
>>> M[1,:]
array([ 7. ,  7. ,  2.5])
>>> M[0,:].dot(M[1,:])
26.600000000000001
```

Don't worry about the `00000000000001` bit.

Question F: Do the math by hand: why does the dot product of `M[0,:]` and `M[1,:]` equal 26.6?

2.2 Working with Your Editor

Let's switch over to writing code in the editor.

DO NOT SKIP THIS STEP: Before you do anything else, you must configure your editor so that it indents using only spaces, and by exactly four spaces. **This is absolutely critical.** You may recall that whitespace is meaningful in Python; that's why I'm having you do this. All the example code that I give you in this document has four spaces as the indent, so you need to be configured to match it, or things may be way more difficult than necessary. If you're running TextWrangler, here's how to do it:

1. Go to the **TextWrangler** menu and select **Preferences**.
2. Select "Editor Defaults" in the left panel.
3. Make sure that the "Auto-expand tabs" option is checked.
4. You also need to specify the tab width as four spaces. Depending on your version, there may be a box at the bottom where you can enter this, or you might need to click the **"Set..."** button next to the "Default font" box, which opens another window where you can set the tab width. Either way, enter 4 in the tab-width box.
5. Close out of the preferences window.
6. With your document open, go to the **Edit** menu and select **Text Options...**.
7. Make sure that the "Auto-expand tabs" option is checked.
8. Type `1234567890` on a blank line of your document. Hit Return.
9. At the beginning of the next line, hit Tab, then type `abcd`. The `a` should line up with the `5` above.
10. If this doesn't work for you, ask someone for help before moving on.

Now erase the stuff currently in your editor window and replace it with this:

```
from numpy import *
```

Anything else that you write should go somewhere below this line; the first thing your program needs to do is load up the `numpy` module.

1. Now that we know how to take the dot product of two vectors (which, again, is the sum of the products of their corresponding elements), let's take another look at the definition of the matrix product.

   If M is a $p \times q$ matrix, and N is a $q \times r$ matrix, then the product MN is a $p \times r$ matrix with these entries:

   $$(MN)_{ij} = \sum_{k=0}^{q-1} M_{ik} N_{kj}.$$

   What's that summation on the right-hand side? It's just the dot product of row i in M with column j in N.

   Question G: In your editor, write a function called `mult()` that takes two matrices as arguments, and computes and returns a new matrix equal to their product. Let me get you started:

```
def mult(M, N):
    p = M.shape[0]
    if N.shape[0] != M.shape[1]:
        raise ValueError("Matrix dimensions must match!")
    r = N.shape[1]
    L = zeros([p,r])
    for i in range(p):
        for j in range(r):
            # ... Fill in just one line here ...
    return L
```

   Read through each line here and make sure you understand what's going on. Investigate by running snippets of code in the interactive interpreter. In particular, what exactly does `L = zeros([p,q])` mean? Why did I do that?

   After you've filled in your multiplication function, test it. Come up with a couple pairs of matrices that can be multiplied together, and work out by hand what you expect the answer to be. Then compare to what your `mult()` function produces. There are a couple ways to do this: non-interactively (the way you're probably used to from Intro) and interactively:

   - **Non-interactively**: In your program, after the definition of `mult()`, define variables `A` and `B` for one pair of matrices (using the `array` constructor). Then print out `A`, `B`, and `mult(A, B)` using three calls of the `print` command, just as you ordinarily would.

     Now switch to the lower terminal window (the one that doesn't have the interactive interpreter running in it). Run the command `python lab.py`, and check to see that the results make sense. Then go change `A` and `B` to be your next example pair, and run again, and so on.

   - **Interactively**: Switch over to your interactive interpreter (the top terminal window). Define two matrices `A` and `B` as in the instructions above. Then enter the command `import lab`. This will grab the code from `lab.py`; namely, the definition of your `mult()` function. Then just run `lab.mult(A,B)` and see what output it gives you. Redefine `A` and `B` as your next pair of test matrices, and run `lab.mult(A,B)` again, and so on. (Notice you need type `lab.mult`, not just `mult`; when you import a module, its contents are only accessible through the module's name.)

     When you're running interactively, the code defined in your `.py` file only gets loaded when you call `import`. If you change the file, you need to reload it, like so: `reload(lab)`.

2. Now let's investigate the tricky case of non-commutativity of matrix multiplication.

   - Let L and M both be $p \times p$ matrices. Then it is not necessarily the case that $LM = ML$.

     Question H: Why not? Use your multiplication function to investigate. Provide a counterexample for $p = 2$.

   With this example, we can conclude that *matrix multiplication is not commutative*. In other words, order matters in matrix multiplication, and *left-multiplying* M by L (that is, computing LM) is not the same as *right-multiplying* M by L (that is, computing ML).

3. `numpy` includes a special command to create an identity matrix: `eye`. `eye(3)` makes a 3×3 identity matrix. Use this and your multiplication function to answer the following question:

   Question I: 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 4.5 of our 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 (`mult`, etc.) with a "`% 2`". `numpy` knows how to apply the mod operation to every element of the 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?

Side note: It can be a hassle to manually type in big matrices like these. Here's how I created E in Python, using the `concatenate()`, `ones()`, and `eye()` functions from `numpy`:

```
E = concatenate( [concatenate( [ones([3,1]), 1-eye(3)], 1), eye(4)], 0)
```

Inspect that code, and see if you can understand what's going on in it. Experiment with the `concatenate` and `ones` functions. Here's a simpler example to dissect—the inner call to `concatenate` above:

```
concatenate( [ones([3,1]), 1-eye(3)], 1)
```

Question J: What are the shapes of the two matrices I created in the middle of that command? What does `concatenate()` do, in general? What does the second argument `concatenate()` mean? (Googling for the `numpy` documentation of `concatenate()` might help you answer this question, along with experimenting with building your own matrices with it, using the interactive interpreter.)

Now use these functions (and maybe `zeros()`) to build D. There isn't a good shortcut for defining C (not using those functions, anyway), so you should just type that one in.

Question K: Which products of E, D, and C are well-defined? (You should find three.) What are they? What patterns do you observe in them? (Remember that we're doing everything mod 2, so after multiplying, you've got to have a `% 2`!)

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:

Question L: 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.

Question M: Describe (in words) the algorithm for detecting and correcting errors, under the assumption that no more than one error has occurred. Part of this process is determining whether any error occurred, remember! Implement this algorithm in your Python program as a function called `correctOneError()`.

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 N: 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

Question O: 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.