Table of Contents

Overview

Converting decimal to binary

It's important to first understand that decimals work in a base 10 (0-9) and binary works in a base 2 (0-1). This means that the weight of each number increases by a factor of 10 in the decimal system, and a factor of 2 in the binary system. This means we can express decimals as powers of 2 to find the binary representation.

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
1	2	4	8	16	32	64	128

We can express the equivalency of a decimal and a binary as, $1001_2 = 9_{10}$, where the subscript 2 means base 2 and the subscript 10 means base 10.

One way to find the exact binary representation is to use the method of repeatedly dividing by 2. Taking 9_{10} as an example;

Result after dividing by 2	Remainder
4	1
2	0
1	0
0	1

The reason for the last result equaling zero is because we did the equivalent of taking the modulus. Once you get to a point in which the result after dividing by 2 is less than or equal to 1, you take the modulus, which is the remainder of the previous result. It is important to note, at least in this case, that the modulus **must be an integer**. If you end up with a fraction, *always* round down to the nearest integer. In the case of $\frac{1}{2}$, $\frac{1}{2} = 0.5 = 0$, *for* $0.5 < 1$, and the remainder is therefore 1.

Another example for 294_{10},

Result after dividing by 2	Remainder
147	0
73	1
36	1
18	0
9	0
4	1
2	0
1	0
0	1

Therefore, $294_{10} = 100100110$

How do you decide which number goes first and which number goes last? You order them from least significant bit (LSB) to most significant bit (MSB). The LSB is the first value you get for your remainder, in the case of 294, it's 0. The MSB is the last value you get for your remainder, in the case of 294, it's 1.

The Code

Overview

The code is separated into 6 pieces;

1. Calling any needed libraries
2. The actual function that converts our decimals into binary
3. Where the user inputs the decimals to be converted
4. Where the binary is formatted into a list
5. Where the list is converted to an array
6. Where the array is used to form the matrix

First cell

```
In [1]: import numpy as np
        from numpy import linalg as LA
        import itertools
        from itertools import islice
```

The first cell is where I call *numpy,* which is a library that helps create arrays and matrices, I also import *linalg* which is a function that will help us create the eigenvalues and eigenvectors later on. I also import *itertools* which is a library that handles iterators, I also import *islice* which returns a specific element from the iterable.

Second cell

```
In [2]: def DecToBin(n):

            if n<=1:
                return str(n)
            else:
                return DecToBin(n//2) + DecToBin(n%2)
```

The second piece of the code is where I define the process for turning our decimal into binary. There are two parts to this part, the first being;

```
if n<=1:
    return str(n)
```

This means that if the user enters anything less than or equal to 1, it'll just return the value as a string.

The second part is;

```
else:
    return DecToBin(n//2) + DecToBin(n%2)
```

If the user inputs a number greater then 1, it will go through two different processes. The first being that it divides by 2 until it doesn't get an integer back, that's what the double forward slashes mean. Next, it will take the modulus, %, to find out the final value. Finally, it will return the binary number.

Third cell

```
In [3]: NewDec = int()
        NewDecList = []
        repeat = 1

        while repeat < 5:
            while NewDec <= 20:
                if repeat == 5:
                    break
                NewDec = int(input("Enter an integer: "))
                NewDec2 = list(DecToBin(NewDec))

                if NewDec > 15:
                    print("This will not be added to NewDecList, try again.\n")
                    repeat -= 1
                elif NewDec <= 15 and NewDec >= 0:
                    NewDecList.append(NewDec)
                    print("This will be added to NewDecList.\n")
                elif NewDec < 0:
                    print("This will not be added to NewDecList, try again.\n")
                    repeat -= 1

                repeat += 1
```

The third cell is composed of 2 parts with a few subparts. The first one being where we define out variables, *NewDec, NewDecList,* and *repeat*.

This first part is about defining some variables.

NewDec = int( )

This means that the new decimal, as entered by the user, will be an integer. Since the brackets are empty, we can put most any value in.

NewDecList = [ ]

This is the list where the approved numbers that the user has entered will be stored.

repeat = 1

This dictates how many times the program will ask the user to enter an integer.

The second part is how we use DecToBin. I'm going to be going line by line for the most part.

while repeat < 5:

This means that the block of code will repeat until repeat is no longer less than 5.

while NewDec <= 20**:**

This is a secondary loop that continues so long as the user inputs numbers that are below 20.

```
    if repeat == 5:
        break
```

This means that as soon as repeat hits 5, the code will stop running this portion and move on to stuff that is outside of the while loops. I've done this so that the matrix will always be a square.

```
    NewDec = int(input("Enter an integer: "))
    NewDec2 = list(DecToBin(NewDec))
```

The first line is asking the user to give a number and the second line is then running that number through the definition to get a binary number and then adding it to the list that *NewDec2* has.

```
    if NewDec > 15:
        print("This will not be added to NewDecList, try again.\n")
        repeat -= 1
    elif NewDec <= 15 and NewDec >= 0:
        NewDecList.append(NewDec)
        print("This will be added to NewDecList.\n")
    elif NewDec < 0:
        print("This will not be added to NewDecList, try again.\n")
        repeat -= 1
```

The first statement will appear if you enter a number greater than 15 and it will subtract 1 from the running total for *repeat* and redo the loop. If the number is greater than 20, the loop will end. The second line will add the new number to NewDecList, meaning it will append NewDec to NewDecList. The last statement is for when the user inputs a number that is less than zero, it will do the same process of subtracting 1 from the running total of *repeat*.

```
    repeat += 1
```

At the end of the loop, before it goes back through it again, it will increase the value of repeat by one.

Fourth cell

```
In [4]: Results = []

        Input = NewDecList

        for i in Input:
            Input = ('{num:04b}'.format(num=i))
            Results.append(Input)

        length = [1,1,1,1]
        Results2 = iter(Results)
        Output = [list(islice(Results2, elem)) for elem in length]
        ResultList = [[int(i) for i in sub] for i in Output for sub in i]
```

The fourth cell is mainly responsible for creating the lists that are composed of each element of the binary number and has 2 parts.

The first part is responsible for creating the list of binary numbers.

```
Results = [ ]
```

This is the list that will be composed of all the binary numbers.

```
Input = NewDecList
```

I'm just making up a new variable to use for the next part that has the same properties as NewDecList.

```
for i in Input:
    Input = ('{num:04b}'.format(num=i))
    Results.append(Input)
```

The *for* loop is going to run over every entry in *Input* and modify it. What's happening on the second line is derived from a python function called 'format', which allows us to do substitution and value formatting. Anything inside the curly brackets is defined as a placeholder, it's the things being formated. The stuff on the other side of the colon serves to identify what type of formatting you want to be used. The 'b' means binary format, and the '04' means I want the number to be 4 characters long, this was the influence for using 15 as the max integer allowed to be used.

The second part is used for converting the list from string values to integer values and is also important for putting it in a form that can then be used to create a matrix. This is also where we use *itertools* and *islice*.

```
length=[1,1,1,1]
```

This is defining how long I want my sublists to be.

```
Results2=iter(Results)
```

This is defining *Results2* to be a version of *Results* that can be iterated one element at a time.

```
Output=[list(islice(Results2, elem)) for elem in length]
```

There are a lot of things going on here so I'll start with the *islice* function. *islice* returns selected elements from an iterable object, in this case, I'm telling it to return all elements located inside *Results2*. I'm then creating a list of these elements that *islice* produces. Finally, I split each element up according to length, which, in this case, is every new string and creates a new sublist for each one. This gives us the *Output* as a list containing sublists that contain the four-character binary number.

```
ResultList=[[int(i) for i in sub] for i in Output for sub in i]
```

The stuff in the first set of square brackets is used to convert the *Output* list from strings to integers. The other stuff is then used to separate each character of the integer and put them in a comma-separated list.

```
Results2:  <list_iterator object at 0x11365a550>
Output:  [['0000'], ['0001'], ['0010'], ['0011']]
ResultList:  [[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0], [0, 0, 1, 1]]
```

Fifth cell

```
In [5]: ResultArray = np.array(ResultList)
```

The fifth cell takes our *ResultList* and makes it a matrix using *numpy*.

ResultArray = np.array(ResultList)

This is me telling it I want to create an array-like object from the *ResultList* list we made in the previous part. It should be noted that, even though it looks identical to what the matrix will be, it's an array type of object. That being said, you can perform operations on using *numpy*, such as finding the eigenvalues and eigenvectors, you don't necessarily need to go on and create a matrix.

```
ResultArray:
 [[0 0 0 0]
 [0 0 0 1]
 [0 0 1 0]
 [0 0 1 1]]
```

Sixth cell

```
In [6]: ResultMatrix = np.matrix(ResultArray)
```

The sixth cell is where we convert the array to a matrix.

ResultMatrix = np.matrix(ResultArray)

I'm telling it to use *ResultArray* to form the matrix.

```
ResultMatrix:
 [[0 0 0 0]
 [0 0 0 1]
 [0 0 1 0]
 [0 0 1 1]]
```

Extras

Overview

There are a few different things you can do with the information the program has created. The main ones I've done are:

- Finding the eigenvalues and eigenvectors using the arrays
- Testing the matrices to see if they are:
    - Hermitian
    - Unitary
    - Normal

Eigenvalues and eigenvectors

Numpy has functions that will find eigenvalues and eigenvectors in 8 significant figures. We specifically use *linalg* which is short for linear algebra.

```
In [7]: w, v = LA.eig(ResultArray)
```

Here I'm defining variable *w* to be a vector that contains the eigenvalues and *v* to be the array that contains the eigenvectors.

```
The eigenvalues are:
 [0. 0. 1. 1.]
The eigenvectors are:
 [[ 1.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00  7.07106781e-01 -7.07106781e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.57009246e-16]
 [ 0.00000000e+00  0.00000000e+00  7.07106781e-01 -7.07106781e-01]]
```

I used [0,1,2,3] since they are integers I've been using for all the examples so far.

It can also do this with complex numbers. Using (0,6,3,8) we get;

```
ResultArray:
 [[1 1 1 0]
 [0 0 1 0]
 [1 0 0 1]
 [1 1 0 0]]
```

```
ResultMatrix:
 [[1 1 1 0]
 [0 0 1 0]
 [1 0 0 1]
 [1 1 0 0]]
```

```
The eigenvalues are:
 [ 2.13039543e+00+0.j         -5.65197717e-01+1.04342744j
 -5.65197717e-01-1.04342744j -1.59566277e-16+0.j        ]
The eigenvectors are:
 [[-6.85125116e-01+0.j          -2.90412688e-01+0.09087583j
  -2.90412688e-01-0.09087583j -5.77350269e-01+0.j        ]
 [-2.47400790e-01+0.j          -2.41187593e-01-0.44526321j
  -2.41187593e-01+0.44526321j  5.77350269e-01+0.j        ]
 [-5.27061513e-01+0.j           6.00918531e-01+0.j
   6.00918531e-01-0.j          -2.87989854e-17+0.j        ]
 [-4.37724326e-01+0.j          -4.92250943e-02+0.53613905j
  -4.92250943e-02-0.53613905j  5.77350269e-01+0.j        ]]
```

The *j* here is used in place of *i* for representing the imaginary number.

Hermitian Testing

Numpy has a really handy class called *np.matrix*, which can help you do different tests on the matrix.

```
In [8]: HRMatrix = ResultMatrix.getH()

        if np.all(ResultMatrix == HRMatrix):
            print("The matrix is Hermitian.")
        else:
            print("The matrix is not Hermitian")
```

Here I'm using the function *getH( )* which will return the complex conjugate transpose of the given matrix, which I've labeled *HRMatrix*. The *np.all* in the 'if' statement tells the program to look at each element of the original matrix and the new one to see if they are identical.

Using [0,1,2,3] as our integers again we find that the matrix is not Hermitian.

```
HRMatrix:     ResultMatrix:
 [[0 0 0 0]    [[0 0 0 0]
 [0 0 0 0]     [0 0 0 1]
 [0 0 1 1]     [0 0 1 0]
 [0 1 0 1]]    [0 0 1 1]]
```

If we instead use [1,2,4,8] as our integers, we find the matrix is Hermitian

```
HRMatrix:     ResultMatrix:
 [[0 0 0 1]    [[0 0 0 1]
 [0 0 1 0]     [0 0 1 0]
 [0 1 0 0]     [0 1 0 0]
 [1 0 0 0]]    [1 0 0 0]]
```

Unitary Testing

A matrix is said to be unitary if, when you multiply the original matrix by its conjugate transpose, you get the identity matrix.

```
In [9]: UniMatrix = ResultMatrix.dot(HRMatrix)

        if np.all(UniMatrix == np.identity(4)):
            print("The matrix is unitary.\n")
        else:
            print("The matrix is not unitary.\n")
```

I'm using the *dot* function to return the unitary matrix, which I'm calling *UniMatrix*. I'm then using *np.identity( )* to create a 4x4 identity matrix to compare to the unitary matrix.
Using [0,1,2,3] as our chosen integers, we see that the matrix is not unitary.

```
UniMatrix:   Identity:
 [[0 0 0 0]   [[1. 0. 0. 0.]
 [0 1 0 1]    [0. 1. 0. 0.]
 [0 0 1 1]    [0. 0. 1. 0.]
 [0 1 1 2]]   [0. 0. 0. 1.]]
```

If we instead use [1,2,4,8] as our integers again, we see that our matrix is unitary.

```
UniMatrix:   Identity:
 [[1 0 0 0]   [[1. 0. 0. 0.]
 [0 1 0 0]    [0. 1. 0. 0.]
 [0 0 1 0]    [0. 0. 1. 0.]
 [0 0 0 1]]   [0. 0. 0. 1.]]
```

Normal Testing

Next, it testing to see if the matrix is normal. A matrix is normal if it commutes with its conjugate transpose.

```
In [10]: NormMatrix1 = ResultMatrix.dot(HRMatrix)

         NormMatrix2 = HRMatrix.dot(ResultMatrix)

         if np.all(NormMatrix1 == NormMatrix2):
             print("The matrix is normal.")
         else:
             print("The matrix is not normal.")
```

I have two variables defined here, *NormMatrix1* for the condition where the original matrix is multiplied by its conjugate transpose and *NormMatrix2* for the condition that the conjugate transpose is multiplied by the original. I then have the program check to see if the two matrices are identical.
Using [0,1,2,3] as our integers, we see that their matrix is not normal.

```
NormMatrix1:          NormMatrix2:
 [[0 0 0 0]            [[0 0 0 0]
 [0 1 0 1]             [0 0 0 0]
 [0 0 1 1]             [0 0 2 1]
 [0 1 1 2]]            [0 0 1 2]]
```

Taking [1,2,4,8] to be our integers now, we see that their matrix is normal.

```
NormMatrix1:          NormMatrix2:
 [[1 0 0 0]            [[1 0 0 0]
 [0 1 0 0]             [0 1 0 0]
 [0 0 1 0]             [0 0 1 0]
 [0 0 0 1]]            [0 0 0 1]]
```