Solving advent of code 2021's day 8 with linear algebra

Stefan Volz

December 8, 2021

```
 0:      1:      2:      3:      4:
 aaaa    ....    aaaa    aaaa    ....
b    c  .    c  .    c  .    c  b    c
b    c  .    c  .    c  .    c  b    c
 ....    ....    dddd    dddd    dddd
e    f  .    f  e    .  .    f  .    f
e    f  .    f  e    .  .    f  .    f
 gggg    ....    gggg    gggg    ....

  5:      6:      7:      8:      9:
 aaaa    aaaa    aaaa    aaaa    aaaa
b    .  b    .  .    c  b    c  b    c
b    .  b    .  .    c  b    c  b    c
 dddd    dddd    ....    dddd    dddd
.    f  e    f  .    f  e    f  .    f
.    f  e    f  .    f  e    f  .    f
 gggg    gggg    ....    gggg    gggg
```

1 Encoding the 7-segment display

We want to encode a 7-segment display somehow. To do this lets first give a mathematical model of our display: note that we can describe any *display state* using a string of letters as described on the problem page. The presence of some letter in such a string means that the corresponding *segment* in the display is lit (so mathematically: we have some alphabet A of length 7 and any subset $S \subseteq A$ corresponds to precisely one state of the seven segment display). But we can also find an alternate description: enumerate all the segments in some way and just note down whether the k-th segment is turned on. So if we for example enumerate them in lexicographic order ($a \mapsto 1, b \mapsto 1, ..., g \mapsto 7$) then we could describe the state `"adef"` as "first segment turned on, second and third segment turned off, fourth, fifth and

sixth segment turned on and seventh off" - or a bit more compactly we could just write `1001110` or $(1,0,0,1,1,1,0)$ for such a state.
These strings of values seem quite familiar from binary etc. and are commonly modelled using $\mathbb{Z}/2\mathbb{Z}$ - the set of integers modulo 2 which is a set that has two elements that are usually refered to as 0 and 1. This set also comes with its own set of arithmetic rules: basically just do your calculations like normal but when you're done divide by two and take the remainder as your result. So we get

$$1+0=1, 1+1=0+0=0$$
$$1\cdot 1=1, 1\cdot 0=0\cdot 0=0.$$

Note that we also have $-1=1$ and thus $x-y=x+y$ for all $x,y\in\mathbb{Z}/2\mathbb{Z}$. Funnily enough these operations work out to behave precisely like the boolean operations of *xor* (addition) and *and* (multiplication).
The cool thing about this set from a mathematical point is, is that it forms a so called *field* - something that behaves kinda like the regular numbers we're used to: we can add, subtract, multiply and divide, have something that acts like a 0 and something that acts like a 1 and all the operations play together "nicely". Because these operations and elements interact so nicely, we can also use a lot of the techniques that we've developed for regular numbers - in particular huge parts of linear algebra can be developed for these fields [1]
To start applying linear algebra we first note that our string of seven 0s and 1s can now be interpreted as an element of $\mathbb{Z}_2^7$ - the vector space of tuples of length seven over $\mathbb{Z}_2$. What does this mean? It means that we have some way of adding our strings of 1s and 0s: we just add each component seperately. We can also do more stuff like multiply each element by a scalar (so "a number") - but in $\mathbb{Z}_2$ this is kinda boring since we'll either end up with our original tuple or the one containing just zeroes.
An important concept related to these vector spaces is that of a basis: we call some collection of elements a basis of some vector space if we can write each element of that space as a *linear combination* of these elements. This is best visualized using an example: The space $\mathbb{Z}_2^2$ is spanned by $e_1=(1,0), e_2=(0,1)$ since we can write each element $v\in\mathbb{Z}_2^2$ (uniquely) as $v=a_1e_1+a_2e_2$ where $a_1,a_2\in\mathbb{Z}_2$. We call a_1,a_2 the coordinates of v with respect to that basis. But note that there are more possible bases: we could either change around e_1 and e_2 (so that we write each one as $a_1e_2+a_2e_1$) or we could choose different elements altogether and go with for example e_1

[1] Note that there are more of these fields - in fact one for each power q of a prime there's a field $\mathbb{F}_q$. In the case where q itself is prime this field is precisely the $\mathbb{Z}/q\mathbb{Z}$ as defined above [and often times denoted by $\mathbb{Z}_q$] and in the case where it's not prime it's a bit more complicated. These (finite) fields have their origin with a mathematician called Évariste Galois and thus are also sometimes called Galois fields - which explains the `galois` import in python - it's a library that allows us to work with these fields.

and $e_1 + e_2$. Importantly these bases will always have the same number of elements (always the dimension of the space) and we can always find some *linear* way to change our coordinates from one to the others. We'll get back to this *changing over* later because this is basically the mechanism we'll use. So we can now encode each string - each state of the seven segment display - as a vector that we can do some calculations with via the following mapping:

$$\phi : P(\{a, b, c, ..., g\}) \to \mathbb{Z}_2^7 \tag{1}$$

$$S \mapsto \left(\begin{cases} 1, & a \in S \\ 0, & \text{otherwise} \end{cases}, ..., \begin{cases} 1, & g \in S \\ 0, & \text{otherwise} \end{cases} \right). \tag{2}$$

2 Translating the problem

Using this we can directly translate all our numbers into codes:

```
import numpy as np
from galois import GF2 as Z2
import re

CODES = {
    "abcefg": 0,
    "cf": 1,
    "acdeg": 2,
    "acdfg": 3,
    "bcdf": 4,
    "abdfg": 5,
    "abdefg": 6,
    "acf": 7,
    "abcdefg": 8,
    "abcdfg": 9}

# We want to interpret the 7-segment display as a 7-dimensional vector over Z/2Z=Z .
# To do this we start by mapping a-g to e_1, ..., e_7.
ident = np.eye(7, dtype=Z2)
ident_z2 = Z2(np.eye(7, dtype=Z2))
mapping = dict(zip("abcdefg", ident))


# we now encode each of our strings with characters from a-g as a combination
# of our basis vectors.
def encode(string):
    return sum(mapping[char] for char in string)
```

```
c = np.array([tuple(encode(k)) for k in CODES])
print(c)
```

Which outputs

```
array([[1, 1, 1, 0, 1, 1, 1],
       [0, 0, 1, 0, 0, 1, 0],
       [1, 0, 1, 1, 1, 0, 1],
       [1, 0, 1, 1, 0, 1, 1],
       [0, 1, 1, 1, 0, 1, 0],
       [1, 1, 0, 1, 0, 1, 1],
       [1, 1, 0, 1, 1, 1, 1],
       [1, 0, 1, 0, 0, 1, 0],
       [1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 0, 1, 1]])
```

This will print us the array `c` where the first row holds the coordinates of the string belonging to 0, the second one those belonging to 1 and so on. Mathematically this means we now have a matrix $c \in \mathbb{Z}_2^(10 \times 7)$ and this matrix has an important property: it has *full column-rank* of 7 - this means that all it's columns are linearly independent and thus are the basis of a 7 dimensional subspace of $\mathbb{Z}_p^{10}$ (10 because we have 10 rows) which through some roundabout way also asserts us that 7 of it's rows are linearly independent and thus a basis of the space $\mathbb{Z}_2^7$ we actually wanna use. What does this mean for the code: we can somehow overlay the different numbers to find our basis vectors.
Consider this example where we overlay 1 and 7:

```
 1:          7:          a:
 ....       aaaa        aaaa
.    c     .    c      .    .
.    c     .    c      .    .
 ....   +   ....   =    ....
.    f     .    f      .    .
.    f     .    f      .    .
 ....        ....        ....
```

So in our encoding this means(I've ommited the commas between the 0s and 1 to make it more legible):

$$\begin{aligned}&(0010010)\\+&(1010010)\\=&(1000000) = e_1.\end{aligned}$$

Importantly this works regardless of which basis we're working in: if we add 1 and 7 we'll always end up with the vector representing exactly this `a` segment.
So to sum this up(pun intended): we can somehow combine our digits to find basis vectors.
Now think about what happens if our elements get all jumbled up as is the case in the problem that we want to solve. Let's say we've jumbled a to b, b to c and so on until we've finally jumbled g to a. Then we could still express each digit as a list of which digits are turned on - we've just changed the way in which we enumerate them. If we for example started with `adg` our jumbled version would be `bea` or in the standard basis of $e_1, ..., e_7$ we've defined above we could write them as (1001001) and (1100100). Playing this jumbling through for all kinds of codes we may notice the following: if we jumble some code like $(1000001) = e_1 + e_7$ then we get the same thing as if we we'd first jumbled e_1 and e_7 separately and added them up afterwards. So if σ is the jumbling map then

$$\sigma(e_1 + e_7) = \sigma(e_1) + \sigma(e_7).$$

We call this property linearity and it's at the very basis of linear algebra. The neat thing about linear maps is that we can express any such map as a matrix once we've chosen a set of coordinates. In this example we get

$$P = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

and you can easily verify that we do indeed get the desired behaviour: write (1000001) as a column vector $\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$ and multiply this from the left by P

matrix using matrix multiplication and you'll end up with $\begin{pmatrix}1\\1\\0\\0\\0\\0\\0\\0\end{pmatrix}$ [2].

3 Finding a solution

Let's remember that we've noticed that we can somehow combine the elements of our space (regardless of their coordinates) and end up with the vectors representing our segments (although we haven't yet nailed down how to actually calculate them - we'll get to that later). We also noticed that we can somehow express the jumbling as a permutation-matrix. This means that we basically know for example e_1 and the jumbled version of e_1, which we could write as Pe_1 for some matrix P. Our problem is that we don't know P. The trick we now use to solve the problem is the following: note that we can write any vector $v \in \mathbb{Z}_2^7$ as a linear combination of the basis elements: $v = \sum_{k=1}^{7} a_k e_k$. We noticed that our jumbling is linear so we have $\sigma(v) = \sigma(a_1e_1 + a_2e_2 + ... + a_7e_7) = \sigma(a_1e_1) + \sigma(a_2e_2) + ... + \sigma(a_7e_7) = a_1\sigma(e_1) + a_2\sigma(e_2) + ... + a_7\sigma(e_7)$. The important bit is: if we knew $\sigma(e_1), ..., \sigma(e_7)$ (using our matrix these are just $Pe_1, ..., Pe_7$) then we could calculate $\sigma(v)$ for any v (mathematically we say that a linear map is uniquely determined by it's action on a basis)! And we can also go the other way around and "dejumble" the letters this way - and for a permutation this is in fact quite easy: just mirror the matrix along it's diagonal (this is called taking the *transpose*) and you're done (if you're super bored you can take the matrix P from earlier, mirror it and multiply P by the result [using matrix multiplication] - you'll find that you get out the *identity matrix* [usually denoted I_7 or E_7] which is a matrix that does nothing when multiplied with any vector.). But we do know $Pe_1, ..., Pe_7$! They are precisely the vectors we'll calculate by "overlaying" the jumbled up versions of our digits.
So we now have a basic plan:

1. Combine the jumbled up digits in such a way that we end up with vectors representing single segments in our display. We have to do this in such a way that we know which segment each vector belongs to (in the earlier example we for example combined 1 and 7 and knew the resulting vector *has* to correspond to the segment `a`).

[2] Mathematically this "jumbling" we've described is called a permutation and the matrices belonging to these permutations have some nice properties.

2. Find a matrix that takes each segment expressed in the default basis (so each e_k) to the right jumbled vector.
3. Invert this matrix
4. Apply the inverted matrix to the output values.
5. Decode the output values from vectors back to actual digits.

4 Implementing it

Before we start we'll note one last thing: the code belonging to 8 is worthless to us since adding it to any state of the display will just flip everything (in logic terms: xoring with `0xFF` is the same as bitwise negation) and (pointwise) multiplication with it will not change anything (in logic: anding with `0xFF` doesn't change anything). So we'll discard it and save some memory.
Okay let's start easy and not overcomplicate things:

```
# I'm assuming that all the definitions and imports from the earlier
# listing are already done here.

# we start with one line of our input that we manually split up
signal_pattern = "dcbgefa cebd bfega eadbf db cdfaeb dba bfcgda egadcf aedcf"
output_values = "egadcfb eafcd db debc"
# we'll encode this by mapping each string to it's corresponding vector
encoded_patterns = np.array(
    [encode(string) for string in signal_pattern.split(" ")
    if len(string) != 7])
# and the same thing for the output_values - although we won't need them
# for a while.
encoded_right_side = np.array(
    [encode(string) for string in output_values.split(" ")])
```

Printing these we have

```
In [13]: encoded_patterns
Out[13]:
array([[0, 1, 1, 1, 1, 0, 0],
       [1, 1, 0, 0, 1, 1, 1],
       [1, 1, 0, 1, 1, 1, 0],
       [0, 1, 0, 1, 0, 0, 0],
       [1, 1, 1, 1, 1, 1, 0],
       [1, 1, 0, 1, 0, 0, 0],
       [1, 1, 1, 1, 0, 1, 1],
```

```
        [1, 0, 1, 1, 1, 1, 1],
        [1, 0, 1, 1, 1, 1, 0]], dtype=object)

In [15]: encoded_right_side
Out[15]:
array([[1, 1, 1, 1, 1, 1, 1],
    [1, 0, 1, 1, 1, 1, 0],
    [0, 1, 0, 1, 0, 0, 0],
    [0, 1, 1, 1, 1, 0, 0]], dtype=object)

    # We'll also define the Z versions of those arrays. Essentially the
    # same thing, the just behave differently under addition etc.
    encoded_patterns_z2 = Z2(encoded_patterns)
    right_sides_z2 = Z2(right_sides)

    # we now calculate the so called weight of each vector (so each row):
    # the number of non-zero entries. We're doing this by abusing the sum
    # operation.
    lengths = encoded_patterns.sum(axis=1)
    # lengths == array([4, 5, 5, 2, 6, 3, 6, 6, 5], dtype=object)
```

Now things get a bit messier: we can group the digits into "classes" depending on how many segments are active in them. The classes are as follows:

```
length | values
    2  | 1
    3  | 7
    4  | 4
    5  | 2, 3, 5
    6  | 0, 6, 9
    7  | 8
```

We make theses classes (as matrices) accessible as a list `l` where the k-th entry is the class with elements of length k.

```
    l = [encoded_patterns_z2[lengths == k] for k in range(8)]
```

And now we can start calculating our basis vectors. We do this by combining the elements of each class in such a way, that we don't have to know which element in the class is actually which digit. If we for example multiply all the elements from the class 5 (pointwise) we end up with a vector that has three entries: one for a, one for d and one for g. We can now subtract a (or add it - remember that addition and subtraction are the same thing in $\mathbb{Z}_2$) to get just d and g which we can then process further. One possible solution

is the following (note that I have not tried simplifying this at all - there's likely some huge redundancies in there)(also note that you could very likely phrase this as some linear system and get the solution through that):

```
    # `fivers`` and `sixers` are sums over all the codes of length 5/6.
    # Summing like this cancels out some unknowns and allows us to
    # get some more basis vectors pretty easily.
    fivers = l[5].sum(axis=0)
    sixers = l[6].sum(axis=0)
    e_1 = l[3] - l[2]  # a
    e_6 = (fivers + sixers) * l[2]  # f
    e_3 = l[2] - e_6  # c
    e_2 = (l[4] + l[2]) * sixers  # b
    e_4 = l[4] - l[2] - e_2  # d
    e_7 = l[5].prod(axis=0) - e_1 - e_4  # g
    e_5 = fivers + sixers - e_4 - e_6  # e
    transform = np.array([e_1, e_2, e_3, e_4, e_5, e_6, e_7]).reshape(7,7)
```

which will leave us with

```
In [41]: transform.reshape(7,7)
Out[41]:
array([[1, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 1],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0]], dtype=uint8)
```

which is the inverse of the permutation matrix that jumbled our input. As described earlier we now want to multiply this from the left with each jumbled value - because matrices are nice we can do all of them at once by just writing one besides the other into a matrix. We earlier defined these "rigth sides" as a matrix where the rows are the vectors we want but we need these vectors to be the columns - so we take the transpose to flip things around.

```
    original_encoding = (transform @ encoded_right_side.T)
```

```
In [52]: original_encoding_z2
Out[52]:
GF([[1, 1, 0, 0],
    [1, 1, 0, 1],
    [1, 0, 1, 1],
```

```
    [1, 1, 0, 1],
    [1, 0, 0, 0],
    [1, 1, 1, 1],
    [1, 1, 0, 0]], order=2)
```

At this point we're essentially done. All that's left to do is decoding:

```
    # create a table where we can plug in some string of 1s and 0s
    # and get out the corresponding digit.
    decoding_table = {tuple(encode(k)): v for (k, v) in CODES.items()}
    digits = np.array([decoding_table[tuple(row)] for row in original_encoding.T])
    # digits == [8, 5, 1, 4]
    # and we can convert this to a single decimal number by taking the dot product
    # with (1000, 100, 10, 1)
    print(digits @ (10 ** np.arange(3, -1, -1)))
```

Which prints the desired result. Now just loop over all entries (or do them all at once by working with arrays) and take the sum at the end.
Also note that you could do the whole thing in vanilla python by just using booleans and their logical connectives, lists etc..