exercise 1

working with bit strings and Boolean functions

solutions due

until **October 29, 2024** at **23:30** via **ecampus**

Note: Carefully read and follow the solution submission instructions which are detailed in task 1.8.

If your submission does not adhere to the guidelines in task 1.8, your solutions will not be accepted / graded.

general remarks

Your instructors are avid proponents of open science and education and therefore favor open source software. The coding problems for this course are thus to be solved using `python`, `numpy`, `scipy`, ... **Implementations in other languages will not be accepted**.

The practical problems on this and future exercise sheets are more or less simple; if you do not have any ideas for how to solve them, just look around for clues or ideas. In fact, you may ask modern AIs such as ChatGPT or Gemini for help. **If you do, please report your experiences.**

Remember that you have to achieve at least 50% of the exercise points to be eligible to the final exam. Your grades (and credits) will be decided on the exam, but you must first succeed in the exercises to get there.

All your solutions have to be *satisfactory* to count as a success. Your proofs, code, and results will be checked and need to be convincing. If they are, then your solutions are *satisfactory*. A *very good* solution (one that is rewarded full points) requires additional efforts w.r.t. mathematical precision and code readability. If your arguments are imprecise or your code is neither commented nor well structured or your discussions lack clarity, your solutions are not good! Striving for very good solutions should always be your goal!

mathematical preliminaries and notation

In this course and its exercises, we assume the set of natural numbers to be given by $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ That is, we explicitly consider 0 to be a natural number (this accords with the ISO 80000-2 standard and is thus quite common).

Below, we also invoke the notion of *bit strings*. We therefore recall that an alphabet $\mathcal{A}$ of size $|\mathcal{A}| = m$ is a set of symbols $\{a_1, \ldots, a_m\}$. A string s over $\mathcal{A}$ is a sequence of symbols $s \in \mathcal{A}^*$ where the Kleene star $\mathcal{A}^*$ denotes the set of all possible sequences (finite or infinite) of symbols contained in $\mathcal{A}$.

If a string s is a sequence of l symbols, it is of length $|s| = l$. It is also an element of $\mathcal{A}^l \subset \mathcal{A}^*$ which is the set of all strings of length l over $\mathcal{A}$. To explicitly denote the constituent symbols of a string, we may use the concatenation operator ":" and write $s = s_1 : s_2 : \ldots : s_l$.

For example, the bit string $b = 110110$ over the alphabet $\mathcal{B} = \{0, 1\}$ may also be written as $b = 1 : 1 : 0 : 1 : 1 : 0$.

task 1.1 [5 points]

bit strings and binary vectors

Recall that any non-negative integer

$$x \in \{0, 1, 2, \ldots, 2^n - 1\}$$

can be represented as a bit string

$$b = b_{n-1} : b_{n-2} : \ldots : b_1 : b_0$$

of length n whose constituent characters b_k are the digits 0 or 1. This works because we can express any such x in terms of a binary expansion

$$x = \sum_{k=0}^{n-1} 2^k \cdot b_k$$

For instance, letting $n = 3$, we have the following binary representations of the numbers 0 through 7

x	b
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Note: Above, we adhered to the convention where the least significant bit b_0 of the binary representation b of x is the rightmost character of the corresponding bit string. Later, we may also work with the "reverse convention" where b_0 is the leftmost character of the bit string b representing x. Mathematically, this this is perfectly acceptable; we just need to be aware of which convention is being followed.

Note: In this course, we will mainly consider binary vectors $\boldsymbol{z} \in \{0, 1\}^n$ rather than bit strings b in order to represent numbers x. Mathematically, this has the advantage that we can work with methods from linear algebra.

Here are your tasks:

a) **[2 points]** Implement a `numpy` function that takes a number $x \in \mathbb{N}$ as input and returns the number $n \in \mathbb{N}$ of bits required for its binary representation.
Run your function for the inputs $x \in \{0, 1, 7\}$ and report your results.

b) **[2 points]** Implement a `numpy` function that takes a natural number $0 \leq x \leq 2^n - 1$ and the parameter n as inputs and returns a binary vector $\boldsymbol{z} \in \{0, 1\}^n$ that represents x according to the most to least significant bit convention.

c) **[1 point]** Letting $n = 5$, use your code from task 1.1 b) to produce a table such as the one shown above.

task 1.2 [5 points]

Gray codes

In 1953, Frank Gray introduced another kind of ordering for the binary numeral system known as reflective binary code or Gray code. For instance, using a bith depth of $n = 3$, we have the following familiar binary codes $b(x)$ and Gray codes $g(x)$ of the natural numbers $x \in \{0, 1, \ldots, 7\}$

x	$b(x)$	$g(x)$
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Here are your tasks:

a) **[2 points]** How do the representations $b(x)$ and $g(x)$ of x differ? Is there anything noteworthy w.r.t. the Hamming distance between the Gray codes $g(x)$ and $g(x+1)$ of two consecutive numbers x and $x+1$? What advantages, if any, could Gray codes g have over the more familiar binary representations b? Discuss this in your own words.

b) **[2 points]** Implement a `numpy` function that takes a natural number x and the number n of bits required for its binary expansion and returns a vector $\boldsymbol{z} \in \{0, 1\}^n$ representing the Gray code of x.

c) **[1 point]** Letting $n = 5$, use your code from task 1.2 b) to produce a table such as the one shown above.

task 1.3 [5 points]

cellular automata

Recall that a one-dimensional (elemental) cellular automaton consists of

- an infinite sequence of cells $\{x_j\}_{j\in\mathbb{Z}}$ each of which is in either one of two states; usually these states are assumed to be 0 or 1, so that

$$x_j \in \{0,1\}$$

- an update rule $f : \{0,1\}^3 \to \{0,1\}$ for how to update a cell based on its and its two neighbors' current states

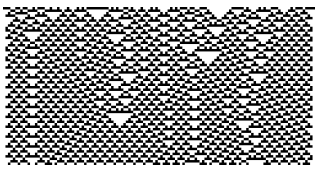
$$x_j[t+1] = f\Big(x_{j-1}[t], x_j[t], x_{j+1}[t]\Big)$$

At time $t = 0$, the x_j are (randomly) initialized, and, at any (discrete) time step t, all cells are updated simultaneously. Here is an illustrative example where we plot 0 as □ and 1 as ■

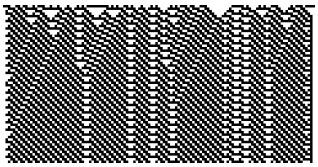
t ⋯□□□□□□□□□□□□□□□□□■□□□□□□□□□□□□□□□□□⋯
$t+1$ ⋯□□□□□□□□□□□□□□□□□■□■□□□□□□□□□□□□□□□□⋯

This update process continues forever and it is common to plot subsequent (finitely sized) state sequences below each other to visualize the evolution of an automaton under a certain rule. Here are some examples of possible evolutions starting from the same initial configuration

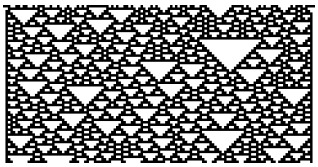
rule 32



rule 108



rule 110



rule 126

From now on, we remove notational clutter. This is to say that, instead of

$$x_j[t+1] = f\Big(x_{j-1}[t], x_j[t], x_{j+1}[t]\Big)$$

we henceforth simply write

$$y = f(\boldsymbol{x})$$

where $\boldsymbol{x} \in \{0,1\}^3$ and $y \in \{0,1\}$.

The naming convention for the rules of elemental cellular automata is due to Stephen Wolfram and illustrated in the following tables

$\boldsymbol{X}^\intercal$			$\boldsymbol{y}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

rule 110

$\boldsymbol{X}^\intercal$			$\boldsymbol{y}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

rule 126

With $\boldsymbol{x} \in \{0,1\}^3$, there are $2^3 = 8$ possible inputs for a rule. For each input, there are 2 possible outputs. Hence, there are $2^{2^3} = 256$ possible rules. We may collect the possible inputs in a transposed 8×3 matrix $\boldsymbol{X}^\intercal$ and the outputs of a rule in an 8 dimensional target vector $\boldsymbol{y}$. Converting a rule's binary target vector (read from bottom to top) into an integer gives the rule's name.

Here are your tasks:

a) **[3 points]** Implement code that produces pictures such as above. Make sure your code can handle any rule f_r for $r \in \{0, 1, \ldots, 255\}$.

Of course, we cannot practically work with infinite sequences of cells. Your implementation will thus have to consider only a finite number of cells, say $n_c = 127$. By the same token, we cannot realize an update process that runs forever. Hence, only consider a finite number of rows, say $n_r = 64$, when plotting the evolution of an automaton.

b) **[2 points]** Randomly initialize the individual cells of your automaton to 0 or 1 and then run it using rules 60 and 102. Show your results.

task 1.4 [5 points]

a crazy idea followed by a stupid one

Let us consider a seemingly crazy idea, namely the following substitution

$$0 \to +1$$
$$1 \to -1$$

People like this representation of binary states because there is a simple mapping from the ordered set $\{0,1\}$ to the ordered set $\{+1,-1\}$, namely

$$x \in \{0,1\} \quad \mapsto \quad x' = (-1)^x \in \{+1,-1\}$$

We can therefore convert binary functions $f_{01} : \{0,1\}^n \to \{0,1\}$ into bipolar functions $f_{+-} : \{+1,-1\}^n \to \{+1,-1\}$ by letting

$$f_{+-}\big((-1)^{x_1},\ldots,(-1)^{x_n}\big) = (-1)^{f_{01}(x_1,\ldots,x_n)}$$

Using all this, the above two tables for rule 110 and rule 126 will become

$\boldsymbol{X}^\intercal$			$\boldsymbol{y}$
$+1$	$+1$	$+1$	$+1$
$+1$	$+1$	-1	-1
$+1$	-1	$+1$	-1
$+1$	-1	-1	-1
-1	$+1$	$+1$	$+1$
-1	$+1$	-1	-1
-1	-1	$+1$	-1
-1	-1	-1	$+1$

rule 110

$\boldsymbol{X}^\intercal$			$\boldsymbol{y}$
$+1$	$+1$	$+1$	$+1$
$+1$	$+1$	-1	-1
$+1$	-1	$+1$	-1
$+1$	-1	-1	-1
-1	$+1$	$+1$	-1
-1	$+1$	-1	-1
-1	-1	$+1$	-1
-1	-1	-1	$+1$

rule 126

Here are your tasks:

a) **[3 points]** Given a matrix $\boldsymbol{X}^\intercal$ and a vector $\boldsymbol{y}$ as in the tables above, write code that first solves the least squares problem

$$\boldsymbol{w}_\star = \operatorname*{argmin}_{\boldsymbol{w}\in\mathbb{R}^3} \big\|\boldsymbol{X}^\intercal\boldsymbol{w} - \boldsymbol{y}\big\|^2$$

and then computes

$$\hat{\boldsymbol{y}} = \boldsymbol{X}^\intercal\boldsymbol{w}_\star$$

Tip: It is strongly suggested to work with the `numpy` function `lstsq` provided in the `linalg` module.

b) **[2 points]** Run your code for rules 110 and 126 and print the respective vectors $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$. Analyze your results. Do they suggest that rules 110 and 126 are linear functions $y = f(\boldsymbol{x})$ or not? Discuss this in your own words.

Don't be alarmed! What you were supposed to do here hardly makes any sense and was mainly intended to prepare ourselves for what comes next.

task 1.5 [5 points]

the Boolean Fourier transform

A Boolean domain $\mathbb{B}$ is a set containing exactly two elements that can be interpreted as `true` and `false`. The most common examples of such domains are $\mathbb{B} = \{0, 1\}$ and $\mathbb{B} = \{-1, +1\}$.

Given this definition, it is not surprising that functions of the following form

$$f : \mathbb{B}^n \to \mathbb{B}$$

are called Boolean functions. Moreover, functions of the form

$$f : \mathbb{B}^n \to \mathbb{R}$$

are called pseudo Boolean functions and we note that, for $\mathbb{B} \subset \mathbb{R}$, the set of pseudo Boolean functions includes the set of Boolean functions.

Above, we already saw that each possible rule $y = f(\boldsymbol{x})$ for the behavior of an elemental cellular automaton can be expressed as a Boolean function

$$f : \{\pm 1\}^n \to \{\pm 1\} \tag{1}$$

where $n = 3$. In what follows, we let $[n]$ denote the index set of the entries x_j of $\boldsymbol{x} \in \mathbb{B}^n$, that is

$$[n] = \big\{1, 2, \ldots, n\big\}.$$

The power set $2^{[n]}$ of $[n]$ is the set of all subsets $\mathcal{S}$ of $[n]$. In other words,

$$2^{[n]} = \big\{\emptyset, \{1\}, \{2\}, \ldots, \{1, 2\}, \ldots, \{1, 2, \ldots, n\}\big\}$$

and we note that $\big|2^{[n]}\big| = 2^n$.

Given these prerequisites, here is why Boolean functions of the form in (1) are interesting: “One can show” that every pseudo Boolean function

$$f : \{\pm 1\}^n \to \mathbb{R}$$

(and therefore every Boolean function of the form in (1)) can be uniquely expressed as a multilinear polynomial

$$f(\boldsymbol{x}) = \sum_{\mathcal{S} \in 2^{[n]}} w_{\mathcal{S}} \prod_{j \in \mathcal{S}} x_j \tag{2}$$

The expression in (2) is known as the Boolean Fourier series expansion of the function f. In contrast to the complex exponentials which form the basis functions for conventional Fourier analysis, here the basis functions are the monomials

$$\varphi_{\mathcal{S}}(\boldsymbol{x}) = \prod_{j \in \mathcal{S}} x_j$$

where by definition

$$\varphi_{\emptyset}(\boldsymbol{x}) = \prod_{j \in \emptyset} x_j = 1$$

Given these definitions of the $\varphi_{\mathcal{S}}(\boldsymbol{x})$, we can rewrite equation (2) as follows

$$f(\boldsymbol{x}) = \sum_{\mathcal{S} \in 2^{[n]}} w_{\mathcal{S}}\, \varphi_{\mathcal{S}}(\boldsymbol{x}) = \boldsymbol{w}^{\intercal} \boldsymbol{\varphi}(\boldsymbol{x}) \tag{3}$$

and recognize that any (pseudo) Boolean function of the kind that we are currently dealing with is an inner product between two vectors $\boldsymbol{w} \in \mathbb{R}^{2^n}$ and $\boldsymbol{\varphi}(\boldsymbol{x}) \in \{+1, -1\}^{2^n}$.

Here is your task:

Implement a `python` / `numpy` function that realizes the transformation

$$\boldsymbol{\varphi} : \{+1, -1\}^n \to \{+1, -1\}^{2^n}$$

which we implicitly introduced above.

To be specific, for an input vector $\boldsymbol{x} = [x_1, x_2, x_3]^{\intercal} \in \{+1, -1\}^3$, your code should produce the output vector

$$\boldsymbol{\varphi}(\boldsymbol{x}) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ x_1\, x_2 \\ x_1\, x_3 \\ x_2\, x_3 \\ x_1\, x_2\, x_3 \end{bmatrix}$$

However, implement you function in a more general manner, namely such that it works for arbitrary $n \in \mathbb{N}$ rather than just just for $n = 3$.

Tip: The python standard library contains the module `itertools` which, in turn, provides functionalities that may come in handy for this task.

task 1.6 [5 points]

a rather reasonable idea

Recall that the matrices $\boldsymbol{X}^\intercal$ in the two tables in task 1.4 are row matrices

$$\boldsymbol{X}^\intercal = \begin{bmatrix} — & \boldsymbol{x}_0^\intercal & — \\ — & \boldsymbol{x}_1^\intercal & — \\ & \vdots & \\ — & \boldsymbol{x}_7^\intercal & — \end{bmatrix}$$

whose rows are the transposed bipolar vectors $\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_7 \in \{\pm 1\}^3$ with

$$\boldsymbol{x}_0 = \begin{bmatrix} +1 \\ +1 \\ +1 \end{bmatrix} \qquad \boldsymbol{x}_1 = \begin{bmatrix} +1 \\ +1 \\ -1 \end{bmatrix} \qquad \cdots \qquad \boldsymbol{x}_7 = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

Using your code from the previous task 1.5, you should thus be able to compute a row matrix

$$\boldsymbol{\Phi}^\intercal = \begin{bmatrix} — & \boldsymbol{\varphi}_0^\intercal & — \\ — & \boldsymbol{\varphi}_1^\intercal & — \\ & \vdots & \\ — & \boldsymbol{\varphi}_7^\intercal & — \end{bmatrix}$$

where

$$\boldsymbol{\varphi}_j = \boldsymbol{\varphi}\big(\boldsymbol{x}_j\big)$$

Here is your task:

Given matrix $\boldsymbol{\Phi}^\intercal$ and the target vector $\boldsymbol{y}$ of a cellular automaton rule, write code that first solves the least squares problem

$$\boldsymbol{w}_\star = \operatorname*{argmin}_{\boldsymbol{w} \in \mathbb{R}^8} \big\| \boldsymbol{\Phi}^\intercal \boldsymbol{w} - \boldsymbol{y} \big\|^2$$

and then computes

$$\hat{\boldsymbol{y}} = \boldsymbol{\Phi}^\intercal \boldsymbol{w}_\star$$

Run your code for rules 110 and 126 and print the respective vectors $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$. What do you observe in comparison to your results in task 1.4? What is the “price” you had to pay to obtain these (hopefully) more reasonable results? Discuss this in your own words.

task 1.7 [20 points]

a subset sum problem

In lecture 02, we discussed subset sum problems (SSPs) and considered the following specific instance of such a problem

```
trgt = 8364
vecX = np.array([265, 453, 311, 876, 158, 344,  65, 411, 366, 314,
                 200, 816, 305, 716, 892, 787,  45, 258, 967, 669,
                 525, 638, 351, 438, 839, 438, 732, 675, 429, 278,
                 175, 192, 408, 243, 733, 176, 111, 570, 332, 766,
                 925, 680, 305, 805, 167, 162, 442, 404,  14, 603,
                 279, 636, 927, 757, 780, 892, 178, 148,  11, 766,
                 272, 520, 317, 573,  89, 776, 389, 444, 429, 984,
                 296,  68, 415, 257, 205, 409, 664, 472, 888,  25,
                 641, 367, 791, 687, 344, 320, 936, 520,  36, 494,
                 719, 362,  78, 437, 841, 163, 869, 945, 918, 840])
```

Here is your task:

Implement *numpy* code that finds solutions to the above SSP. Make sure your code includes and applies methods for validating purported solutions. Run your code, try to find at least 10 different solutions, and report them.

Remark: To this day, your instructors found more than $45,000$ solutions to the above SSP. This sounds like a lot, but note the following: The above array `vecX` encodes a (multi-)set $\mathcal{X}$ of 100 elements and the number of all subsets of a set of size 100 amounts to

$$2^{100} = 1,267,650,600,228,229,401,496,703,205,376$$

Given a search space of this magnitude, namely of size $O(10^{30})$, a solution space of size $O(10^4)$ pales in comparison. In other words, your code will have to search for the proverbial needle in a haystack ...

We are looking forward to your ideas and approaches for this problem!

task 1.8

submission of presentation and code

Prepare a presentation / set of slides about your solutions and results. **Submissions of only jupyter notebooks will not be accepted.**

Your slides should help you to give a scientific presentation of your work (i.e. to give a short talk in front of your fellow students and instructors and answer any questions they may have).

W.r.t. to formalities, please make sure that

- your presentation contains a title slide which lists the **names and matriculation numbers** of everybody in your team who contributed to the solutions.

W.r.t. content, please make sure that

- your presentation contains about 12 to 15 content slides but not more
- your presentation is concise and clearly structured
- your presentation answers questions such as
  - "what was the task / problem we considered?"
  - "what difficulties (if any) did we encounter?"
  - "how did we solve them?"
  - "what were our results?"
  - "what did we learn?"

Save / export your slides as a PDF file and upload it to eCampus.

Furthermore, please name all your code files in a manner that indicates which task they solve (e.g. `task-1-5.py`) and put them in an archive or a ZIP file.

Upload this archive / ZIP file with your code snippets to eCampus.