exercise 2

important transformations

solutions due

until **November 12, 2024** at **23:30** via **ecampus**

⚠

Note: Carefully read and follow the solution submission instructions which are detailed in task 2.6.

If your submission does not adhere to the guidelines in task 2.6, your solutions will not be accepted / graded.

general remarks

These exercise sheets expose you to ideas and objects which are of truly fundamental importance in quantum computing.

If you have not yet seen these ideas and objects or never worked with them before, we encourage you to make an effort to really understand what you are doing when you solve the following tasks.

task 2.1 [10 points]

Hadamard transforms

In the last exercises, we worked with Boolean functions $f : \mathbb{B}^n \to \mathbb{B}$ over a Boolean domain $\mathbb{B}$. We also had fun with elemental cellular automata whose update rules can be thought of a Boolean functions

$$f : \mathbb{B}^3 \to \mathbb{B} \qquad \text{where} \qquad \mathbb{B} = \{0, 1\}$$

For instance, here are the definitions of rules 110 and 126 in a slightly different notation than last time:

x	$\boldsymbol{b}(x)$			$f(\boldsymbol{b}(x))$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

rule 110

x	$\boldsymbol{b}(x)$			$f(\boldsymbol{b}(x))$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

rule 126

See what we did here? We listed the numbers $x \in \{0, 1, \ldots 2^3 - 1\}$ and wrote them as big-endian Boolean vectors $\boldsymbol{b} = [b_3\, b_2\, b_1]^\intercal \in \mathbb{B}^3$ for which we then gave the function value as $f(\boldsymbol{b}(x))$. But this is to say that we may also think of Boolean functions $f : \mathbb{B}^n \to \mathbb{B}$ as partial functions from $\mathbb{N}$ into $\mathbb{B}$, namely

$$f : \{0, 1, \ldots, 2^n - 1\} \to \mathbb{B}$$

If we next interpret the domain $\{0, 1, \ldots, 2^n - 1\}$ as an index set, we realize that every Boolean function can be represented as a Boolean vector

$$\boldsymbol{f} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{2^n-1} \end{bmatrix} \in \mathbb{B}^{2^n}$$

For instance, for the above rule 110, we would have $\boldsymbol{f}_{110} = [0\,1\,1\,1\,0\,1\,1\,0]^\intercal$.

Given these prerequisites, we now turn to what this task is all about and meet a family of objects which are of fundamental importance in quantum computing . . .

Every 2^n-dimensional vector $\boldsymbol{f}$ over $\mathbb{C}$, that is not just every Boolean vector, has a Hadamard transform

$$\boldsymbol{f} = \boldsymbol{H}_n \boldsymbol{w} \tag{1}$$

where the $2^n \times 2^n$ matrix $\boldsymbol{H}_n$ has a very special structure.
For instance, for the particular case where $n = 2$, we have

$$\boldsymbol{H}_2 = \frac{1}{\sqrt{2^2}} \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix}$$

In general, i.e. for any $n \geq 1$, Hadamard matrices can famously be constructed recursively, either like this

$$\boldsymbol{H}_0 = [1]$$
$$\boldsymbol{H}_n = \tfrac{1}{\sqrt{2}} \begin{bmatrix} +\boldsymbol{H}_{n-1} & +\boldsymbol{H}_{n-1} \\ +\boldsymbol{H}_{n-1} & -\boldsymbol{H}_{n-1} \end{bmatrix}$$

or like this

$$\boldsymbol{H}_1 = \tfrac{1}{\sqrt{2}} \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix}$$
$$\boldsymbol{H}_n = \boldsymbol{H}_1 \otimes \boldsymbol{H}_{n-1}$$

where "$\otimes$" denotes the Kronecker product.

Here are your tasks:

a) **[4 points]** Implement a *numpy* function that takes a number $n \in \mathbb{N}$ as input and returns the corresponding Hadamard matrix $\boldsymbol{H}_n$.

b) **[2 points]** Compute the Hadamard transforms of rules 110 and 126. That is, solve $\boldsymbol{f}_{110} = \boldsymbol{H}_3 \boldsymbol{w}_{110}$ and $\boldsymbol{f}_{126} = \boldsymbol{H}_3 \boldsymbol{w}_{126}$ for $\boldsymbol{w}_{110}$ and $\boldsymbol{w}_{126}$, respectively. Verify that your results are correct and print them.

c) **[2 points]** Compute the expression $\boldsymbol{H}_n^\dagger \boldsymbol{H}_n$ for several choices of n where $\boldsymbol{H}_n^\dagger$ denotes the conjugate transpose of $\boldsymbol{H}_n$. Discuss what you observe. What does your observation imply for the columns and/or rows of matrix $\boldsymbol{H}_n$? Why does this implication guarantee that every 2^n-dimensional vector has a Hadamard transform?

Note: If you use your computer to compute $\boldsymbol{H}_n^\dagger \boldsymbol{H}_n$ then be aware that it may run into numerical issues. In other words, be very careful or be smart when interpreting the result of $\boldsymbol{H}_n^\dagger \boldsymbol{H}_n$ according to your computer.

d) **[2 points]** Use your insights from subtask c) to solve $\boldsymbol{f}_{110} = \boldsymbol{H}_3 \boldsymbol{w}_{110}$ and $\boldsymbol{f}_{126} = \boldsymbol{H}_3 \boldsymbol{w}_{126}$ for $\boldsymbol{w}_{110}$ and $\boldsymbol{w}_{126}$ in a clever and efficient manner.

task 2.2 [10 points]

Fourier transforms

Every 2^n-dimensional vector $\boldsymbol{f}$ over $\mathbb{C}$, that is not just every Boolean vector, has a Fourier transform

$$\boldsymbol{f} = \boldsymbol{F}_n \boldsymbol{w} \tag{2}$$

where the $2^n \times 2^n$ matrix $\boldsymbol{F}_n$ has a very special structure.
For instance, for the particular case where $n = 2$, we have

$$\boldsymbol{F}_2 = \frac{1}{\sqrt{2^2}} \begin{bmatrix} i^0 & i^0 & i^0 & i^0 \\ i^0 & i^1 & i^2 & i^3 \\ i^0 & i^2 & i^4 & i^6 \\ i^0 & i^3 & i^6 & i^9 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & +i & -1 & -i \\ +1 & -1 & +1 & -1 \\ +1 & -i & -1 & +i \end{bmatrix}$$

In general, i.e. for any $n \geq 1$, the entries of Fourier matrices are given by

$$\bigl[\boldsymbol{F}_n\bigr]_{jk} = \tfrac{1}{\sqrt{2^n}} \exp\left(\tfrac{i\,2\,\pi\,j\,k}{2^n}\right)$$

where we start counting row indices (j) and column indices (k) from zero, i.e. where we assume

$$j \in \bigl\{0, 1, \ldots, 2^n - 1\bigr\}$$
$$k \in \bigl\{0, 1, \ldots, 2^n - 1\bigr\}$$

Here are your tasks:

a) **[4 points]** Implement a *numpy* function that takes a number $n \in \mathbb{N}$ as input and returns the corresponding Fourier matrix $\boldsymbol{F}_n$.

b) **[2 points]** Compute the Fourier transforms of rules 110 and 126. That is, solve $\boldsymbol{f}_{110} = \boldsymbol{F}_3 \boldsymbol{w}_{110}$ and $\boldsymbol{f}_{126} = \boldsymbol{F}_3 \boldsymbol{w}_{126}$ for $\boldsymbol{w}_{110}$ and $\boldsymbol{w}_{126}$, respectively. Verify that your results are correct and print them.

c) **[2 points]** Reproduce your results by using function *fft* in module *numpy.fft*.

   Note: Carefully choose *fft*'s parameter `norm` to get the right result.

d) **[2 points]** Compute the expression $\boldsymbol{F}_n^\dagger \boldsymbol{F}_n$ for several choices of n where $\boldsymbol{F}_n^\dagger$ denotes the conjugate transpose of $\boldsymbol{F}_n$. Discuss what you observe. What does your observation imply for the columns and/or rows of matrix $\boldsymbol{F}_n$? Why does this implication guarantee that every 2^n-dimensional vector has a Fourier transform?

task 2.3 [10 points]

vector spaces over Boolean rings

Consider the Boolean ring R on $\mathbb{B} = \{0, 1\}$, that is, consider the following algebraic structure

$$R = \big(\mathbb{B}, \oplus, \cdot\big) \tag{3}$$

for which the multiplication $x \cdot y$ and addition $x \oplus y$ of two objects $x, y \in \mathbb{B}$ are defined according to the following tables

$\cdot$	0	1
0	0	0
1	0	1

$\oplus$	0	1
0	0	1
1	1	0

In other words, the multiplication "$\cdot$" behaves like the usual multiplication but the addition "$\oplus$" behaves like the usual addition mod 2. In Boolean logic, this addition is also known as the XOR function.

Here are your tasks:

a) **[5 points]** Show that $\big(\mathbb{B}^n, \oplus, \cdot\big)$ for $n > 1 \in \mathbb{N}$ is a vector space over R.

b) **[5 points]** Letting $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{B}^n$, show that

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle \equiv \bigoplus_{j=1}^{n} x_j \cdot y_j$$

is an inner product for the vector space $\big(\mathbb{B}^n, \oplus, \cdot\big)$.

Remarks for mathematicians: Your instructors are well aware of the fact that what we call R is also known as the Galois field $GF(2) = \mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$.

We are also know that mathematicians would prefer to call $(\mathbb{B}^n, \oplus, \cdot)$ a "module" rather than a "vector space" but we try to keep things simple for those who do not study math . . .

task 2.4 [10 points]

Bool-Möbius transforms

⚠ ⚠ ⚠

Note: On this and on the next page, matrix-vector and matrix-matrix products are to be understood as

$$\boldsymbol{a} = \boldsymbol{B}\,\boldsymbol{c} \quad \Leftrightarrow \quad a_j = \bigoplus_{l=1}^{2^n} B_{jl} \cdot c_l \tag{4}$$

$$\boldsymbol{A} = \boldsymbol{B}\,\boldsymbol{C} \quad \Leftrightarrow \quad A_{jk} = \bigoplus_{l=1}^{2^n} B_{jl} \cdot C_{lk} \tag{5}$$

Every 2^n-dimensional vector $\boldsymbol{f}$ over R has a Bool-Möbius transform

$$\boldsymbol{f} = \boldsymbol{S}_n \boldsymbol{w} \tag{6}$$

where the $2^n \times 2^n$ matrix $\boldsymbol{S}_n$ in has a very special structure.
For instance, for the particular case where $n = 2$, we have

$$\boldsymbol{S}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

In general, i.e. for any $n \geq 1$, Sierpinski matrices can be constructed recursively as

$$\boldsymbol{S}_1 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$$\boldsymbol{S}_n = \boldsymbol{S}_1 \otimes \boldsymbol{S}_{n-1}$$

where "$\otimes$" denotes the Kronecker product.

Here are your tasks:

a) **[2 points]** Implement a `numpy` function that takes a number $n \in \mathbb{N}$ as input and returns the corresponding Sierpinski matrix $\boldsymbol{S}_n$.

b) **[4 points]** Pay attention to (5) and compute the product $\boldsymbol{S}_n^2 = \boldsymbol{S}_n \boldsymbol{S}_n$. Discuss what you observe! Have you seen a result like this before? If so, please explain where and in which context.

c) **[4 points]** Use your result from subtask b) to compute the Bool-Möbius transforms of rules 110 and 126. That is, solve $\boldsymbol{f}_{110} = \boldsymbol{S}_3 \boldsymbol{w}_{110}$ and $\boldsymbol{f}_{126} = \boldsymbol{S}_3 \boldsymbol{w}_{126}$ for $\boldsymbol{w}_{110}$ and $\boldsymbol{w}_{126}$, respectively. Verify that your results are correct and print them.

task 2.5 [5 + 5 points]

yet another feature transform

In the last exercises, you were given an n-dimensional vector $\boldsymbol{x}$ (over $\mathbb{C}$) and had to implement a function that maps it to a 2^n-dimensional vector $\varphi(\boldsymbol{x})$. We introduced this mapping using this 3-dimensional example

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \xrightarrow{\varphi} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ x_1x_2 \\ x_1x_3 \\ x_2x_3 \\ x_1x_2x_3 \end{bmatrix} \tag{7}$$

Such high-dimensional vectors play an absolute central role in quantum computing. To be more specific, what is so absolutely central about these vectors are their entries and *not* the order in which we list them.

We may therefore consider another mapping from $\mathbb{C}^n$ to $\mathbb{C}^{2^n}$ which we again illustrate by means of a 3-dimensional example

$$\begin{bmatrix} x_3 \\ x_2 \\ x_1 \end{bmatrix} \xrightarrow{\phi} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1x_2 \\ x_3 \\ x_1x_3 \\ x_2x_3 \\ x_1x_2x_3 \end{bmatrix} \tag{8}$$

To see why this alternative mapping is of considerable interest, we recall that, for any $x \in \mathbb{C}$, we have $x^0 = 1$ and $x^1 = x$. But this means that we have the following equivalence

$$\begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1x_2 \\ x_3 \\ x_1x_3 \\ x_2x_3 \\ x_1x_2x_3 \end{bmatrix} \Leftrightarrow \begin{bmatrix} x_3^0\, x_2^0\, x_1^0 \\ x_3^0\, x_2^0\, x_1^1 \\ x_3^0\, x_2^1\, x_1^0 \\ x_3^0\, x_2^1\, x_1^1 \\ x_3^1\, x_2^0\, x_1^0 \\ x_3^1\, x_2^0\, x_1^1 \\ x_3^1\, x_2^1\, x_1^0 \\ x_3^1\, x_2^1\, x_1^1 \end{bmatrix} \tag{9}$$

Ponder this equivalence and see what we did here. Connect it to what we did in task 2.1 and convince yourself that we can get corresponding mappings from $\boldsymbol{x}$ to $\boldsymbol{\phi}(\boldsymbol{x})$ for arbitrary choices of n.

Here is your task:

Implement a *numpy* function that realizes the transformation

$$\boldsymbol{\phi} : \mathbb{C}^n \to \mathbb{C}^{2^n}$$

we just discussed. To be specific, given a vector $\boldsymbol{x} = [x_3 \, x_2 \, x_1]^\intercal \in \mathbb{C}^3$, your code should produce the vector in (9). However, implement your function in a more general manner, namely such that it works for arbitrary $n \in \mathbb{N}$ rather than just just for $n = 3$.

Note: If you can successfully solve this task, you will be awarded **5 points**.

Note: There is a very compelling solution to this task which involves the Kronecker product "$\otimes$". Yet, this solution requires creative out-of-the-box thinking. If you can discover this solution and successfully implement it using *numpy.kron*, you will be awarded **5 additional points**.

task 2.6

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.