Universität Zürich HS 2022
Institut für Informatik

Student Name:
Matrikel-Nr:

Numerical Methods in Informatics - Exercise 1

Hand out: 05.10.2022 - Due to: 18.10.2022 (End of day)

Please upload your solutions to the Olat system.

Practice

Code is code

If you spot mistakes or encounter errors, it's absolutely possible the given code has errors in it. If you spot anything that seems odd, you can always ask in OLAT or send me a mail.

1.1 CT Scanner

In a CT scanner, every slice of the 3D image gets created by shooting several x-rays from different angles and then reconstructing the slice from these rays. In this task, you are supposed to write the methods needed, to reconstruct a CT image slice from x-ray scans. Since medical images are a bit unaesthetic, in this task you'll "scan" a picture of a raccoon[1]. Even though, that's not a medical image, the method stays the same.

A bit of physics

Whenever an x-ray passes some material, its intensity decreases, depending on the materials density. The formula for the measured intensity (I_m) on the other side of the scanning-object is

$$I_m = I_0 \cdot e^{-d_0 l_0} \cdot e^{-d_1 l_1} \cdot \ldots = I_0 \cdot \prod_{i=0}^{\infty} e^{-d_i l_i} = I_0 \cdot e^{-\sum_{i=0}^{\infty} d_i l_i} \tag{1}$$

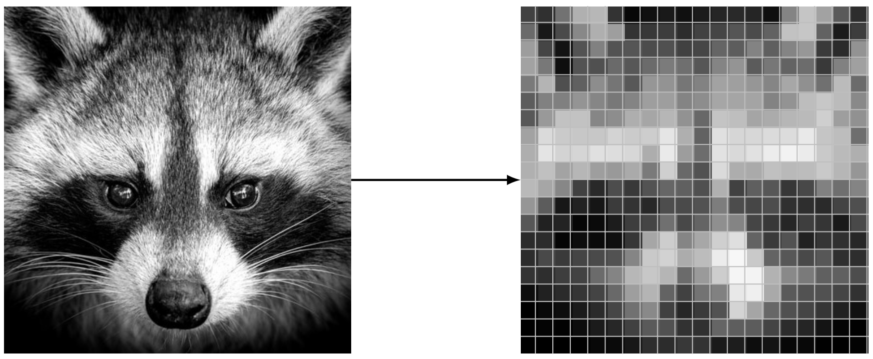
With d_i representing densities and l_i representing the distance, the x-ray travels through the material with the respective density d_i. I_0 represents the initial x-ray intensity.

Unfortunately, in a continuous world, there are assumably infinitely many materials with different densities, making it impossible to reconstruct everything correctly in a finite system (like a computer).

[1] Original photographer: Yannick Menard

A bit of practical stuff

In practice, this problem gets solved by discretization. By laying a grid over the continuous object and assuming fixed densities within each cell of the grid, the problem gets finite, as indicated in Figure 1.



(a) A continuous raccoon. (b) A discrete raccoon.

Figure 1: Discretization of a raccoon

Equation 1 is still not feasible to use yet. First of all, it's easiest to assume, that $I_0 = 1$, which makes the variable vanish from the whole equation. Then, using $\ln(e^x) = x$, allows us to transform the e^x term to a simple x one, making the overall equation more graspable. Applying all this to Equation 1 and multiplying afterwards by -1 yields

$$-\ln(I_m) = \sum_{i=0}^{N} d_i l_i \tag{2}$$

with N being the total amount of cells (in Figure 1b, $N = 20 \cdot 20 = 400$).

This equation can be used to set up a linear system of equations. The overall goal is to calculate the d_i, which are represented in Figure 1b as grey-scale values.

a) (20 Min, 2 Points) Determining the rank of a matrix

Please edit the `rowrank(matrix)` function in `backend.py`. In this function, you're supposed to determine the rowrank of the matrix given as an argument. The return value should be the rowrank.

The rank of a matrix is the number of linear independent rows.

b) (20 Min, 2 Points) Setting up the System

Please edit the `setUpLinearSystem(scanner)` function in `backend.py`. In this function, you're supposed to set up a solvable system of linear equations, which you can later use to reconstruct the image. The return values should be a matrix A, as well as the resulting vector $\vec{b}$.

Within this function, you can use the `scanner.shootRays(angle)` function (defined in `ctscanner.py`) to shoot a set of parallel rays from different angles into the image. As a result, the function returns three lists. Each list has as many entries, as rays have been shot. The first list contains the 1D indices of the cells the ray has hit. The second list contains the measured intensities per ray. The third list contains the lengths each ray has passed through the cells indexed in the first list.

Please note, that the measurements from an angle θ and $-\theta$ will yield the same values.

To elaborate a little bit more: By using the function `scanner.shootRays(angle)` and shooting one ray ($_m$) we get its measured intensity (-ln(I_m)) as given in equation 2. One ray may hit one or several cells. For each cell ($_i$) that was hit, we get the distance (l_i) the x-ray travelled through this cell. What we do NOT get by shooting one ray, is the density (d_i) of each cell that was hit. All the d_i (plural) are what we are eventually looking for in this practical exercise. So by shooting one ray, using the equation 2 and omitting cells that were not hit (since their distance is 0) we get for example:

$$-\ln(I_0) = d_3 l_3 + d_5 l_5 + d_{16} l_{16} \tag{3}$$

By shooting m x-rays we get for example:

$$\begin{aligned}
-\ln(I_0) &= d_3 l_3 + d_5 l_5 + d_{16} l_{16}\\
-\ln(I_1) &= d_6 l_6 + d_{25} l_{25}\\
-\ln(I_2) &= d_3 l_3 + d_{22} l_{22} + d_{89} l_{89} + d_{380} l_{380}\\
-\ln(I_3) &= d_{220} l_{220} + d_{221} l_{221} + d_{230} l_{230} + d_{233} l_{233}\\
&\dots\\
-\ln(I_m) &= d_0 l_0 + d_1 l_1 + \dots + d_n l_n
\end{aligned} \tag{4}$$

These equations can also be written down in matrix notation $Ax = b$. The left part of the equations 4 corresponds to b with m measurements (shot rays), all the d_i (plural) corresponds to $\vec{x}$ with n entries. And the l_i values are used to build the Matrix A. There are m rows in the matrix A, since we have m measurements (shot x-rays) and for each row and its measurement we have all the l_i values for all the n cells. Thus, the Matrix A has (m x n) dimensions, $\vec{x}$ has n entries and therefore (n x 1) dimensions and $\vec{b}$ has m entries and therefore (m x 1) dimensions.

$$A = \begin{pmatrix} l_{00} & l_{01} & l_{02} & l_{03}... & l_{0n} \\ l_{10} & l_{11} & l_{12} & l_{13}... & l_{1n} \\ l_{20} & l_{21} & l_{22} & l_{23}... & l_{2n} \\ ... & & & & \\ l_{m0} & l_{m1} & l_{m2} & l_{m3}... & l_{mn} \end{pmatrix} \text{ and } \vec{x} = \begin{pmatrix} d0 \\ d1 \\ d2 \\ ... \\ d_n \end{pmatrix} \text{ and } \vec{b} = \begin{pmatrix} -\ln(I_0) \\ -\ln(I_1) \\ -\ln(I_2) \\ ... \\ -\ln(I_m) \end{pmatrix}$$

To get the d_i (plural) and solve the system, we need a unique solution and we only get this, if our matrix has full rank. “Full rank” means the same as: We need to have at least as many linearly independent equations as unknown variables. When choosing the same resolution as in figure 1 we would need to have at least 400 measurements that are linearly independent. Depending on the angle that is used in `scanner.shootRays(angle)`, we get different measurements. `scanner.shootRays(angle)` shoots as many rays as you initially set for the resolution when calling the `CTScanner`s constructor. Let say you set it to 20, then 20 parallel rays for a given angle are shot. Thus, we get 20 equations each time we call `scanner.shootRays(angle)`. We can either shoot the rays from angle 0 to π with some increment c. Or we could just randomly select angles. In both cases, we need to gather all the data of all shot x-rays to build/extend our matrix A and $\vec{b}$ until we have full rank for matrix A.

As stated in the text above `scanner.shootRays(angle)` returns three lists. Each list has as many entries, as rays have been shot. The first list contains the **1D**[2] indices of the cells ($_i$) the ray has hit. The second list contains the measured intensity ($-\ln(I_m)$) per ray and the third list contains the lengths (l_i values) each ray has passed through the cells indexed in the first list.

Hints: To speed up your debugging and testing, you can reduce the resolution of the discretization, by handing it over to the `CTScanner`s constructor in `exercise_01.py`. Please note, that the number of cells is the squared resolution.
E.g. in Figure 1b, the resolution would have been set to 20.

Make use of the `rowrank(matrix)`, to make sure that your matrix is solvable (i.e. that the rowrank of the matrix equals the amount of variables it’s supposed to solve).

[2]Having a 20x20 Matrix results in 400 1D indices.
1D index: 0 = 2D index: [0,0]
1D index: 399 = 2D index: [19,19]

.

c) (60 Min, 6 Points) Solving the system

Please edit the solveLinearSystem(A, b) function in backend.py. In this function, you're supposed to implement the gaussian elimination method. The return value should be the vector $\vec{x}$, satisfying $A\vec{x} = \vec{b}$.

Hints:

To score all the points, your solver must be numerically stable, which you can achieve by adding full pivoting to your implementation.

Performance is not as important as accuracy. Still, there is a time limit to the evaluation of this task.

Handing in:

Please only include your backend.py in your hand in.