Exercise 5

Deadline: 06.12.2017, 10:00

This exercise is dedicated to linear regression and its efficient implementation for large sparse matrices. We will test this method on the relevant application of computer tomography.

Regulations

Please hand in your solution as a Jupyter notebook `linear-regression.ipynb`, accompanied with exported PDF `linear-regression.pdf`. Zip all files into a single archive with naming convention (sorted alphabetically by last names)

`lastname1-firstname1_lastname2-firstname2_exercise05.zip`

or (if you work in a team of three)

`lastname1-firstname1_lastname2-firstname2_lastname3-firstname3_exercise05.zip`

and upload it to Moodle before the given deadline. We will give zero points if your Zip file does not conform to the naming convention.

Setting

You find yourself in a two-dimensional world in the role of a medical doctor. A patient (we will call him H.S. here for anonymity) walks into your hospital and complains about a headache. A close examination of the head provides no exterior clues as to the cause of the pain, and you soon realize that you will have to perform computer tomography. Unfortunately, the computer for processing recorded data is broken. Nevertheless, you let H.S. enter the tomograph and take X-ray images of his head from several angles. You send H.S. back to the waiting room and start to think about how you might process the scans using only your private laptop.

Preliminaries

For the scans, H.S.'s 2D head was placed between an X-ray source and a 1D sensor array. The sensor elements record the accumulated signals of many parallel rays passing through the head (see left side of figure 1). To reconstruct all the internal structures, multiple measurements need to be taken from different angles α. The resulting tomogram is an image $\boldsymbol{\mu} \in \mathbb{R}^{M \times M}$ where each pixel encodes the local X-ray absorption coefficient of the patient's head (bright for bones, gray for soft tissue, black for air). The least-squares reconstruction algorithm will create a flattened version $\boldsymbol{\beta} \in \mathbb{R}^D$ of $\boldsymbol{\mu}$ that needs to be reshaped into a 2D image to be displayed. The pixels $\boldsymbol{\mu}_{j_a,j_b}$ are related to the vector elements $\boldsymbol{\beta}_j$ by the formula $j = j_a + M j_b$, as shown in figure 1.

The sensor consists of a 1-dimensional array of N_p sensor elements that register parallel X-rays, and the emitter-detector setup is rotated into N_o different orientations α_{i_o}. The measured intensities are commonly displayed as a 2D image called a *sinogram*, see figure 3. For least-squares reconstruction, we flatten the sinogram $\boldsymbol{s} \in \mathbb{R}^{N_o \times N_p}$ into the response vector $\boldsymbol{y} \in \mathbb{R}^N$. Its elements $\boldsymbol{y}_i$ are defined by the relation $i = i_p + N_p i_o$, where i_p and i_o are the indices of the sensor elements and sensor orientations respectively.

We can now describe the relation between the tomogram $\boldsymbol{\beta}$ and the measured intensities $\mathbf{y}$ as a linear projection with projection matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$:

$$\mathbf{X} \cdot \boldsymbol{\beta} = \mathbf{y} \tag{1}$$

Intuitively, entry $\mathbf{X}_{ij}$ should be interpreted as a weight which encodes how much the sensor response $\mathbf{y}_i$ is influenced by the absorption at pixel $\boldsymbol{\beta}_j$.

In the continuous domain, the measured intensity I_{sensor} of a single ray $\mathbf{r}$ depends on how much of the original radiation I_{source} is absobed by the material between emitter and sensor. This can be modelled as

$$I_{\text{sensor}} = I_{\text{source}} \cdot \exp\left(-\int_{a,b\in\mathbf{r}} \mu(a,b)\,\mathrm{d}a\,\mathrm{d}b\right), \tag{2}$$

where μ describes the material's absorption properties at each spatial position (a, b) and can be seen as a continuous version of the discrete tomography image $\boldsymbol{\mu}$ defined above. For simplicity, we assume that the raw sensor measurements are already subjected to preprocessing which computes the logarithm of this formula, so that the response is defined by

$$y = \ln\left(\frac{I_{\text{source}}}{I_{\text{sensor}}}\right) \tag{3}$$

$$= \int_{a,b\in\mathbf{r}} \mu(a,b)\,\mathrm{d}a\,\mathrm{d}b. \tag{4}$$

For practical computation, we need to discretize this formula. Each sensor element is essentially a bin collecting radiation from many parallel rays. We assume that a ray intersecting the sensor array inbetween two bins distributes its intensity among both. The ratio of distribution is determined in a linear fashion by how close the intersection is to the center of either bin.

Example: Suppose, ray i intersects the sensor array at position 4.35, that is, between sensor elements 4 and 5. In this case, sensor element 4 captures 65% of the ray's intensity and sensor element 5 captures 35%. This is encoded in the weight matrix $\mathbf{X}$ by setting $\mathbf{X}_{i4} = 0.65$ and $\mathbf{X}_{i5} = 0.35$.

Thanks to this definition, the integral in equation 4 can be discretized into a sum, and writing the sum in matrix notation gives the linear system of equation 1.

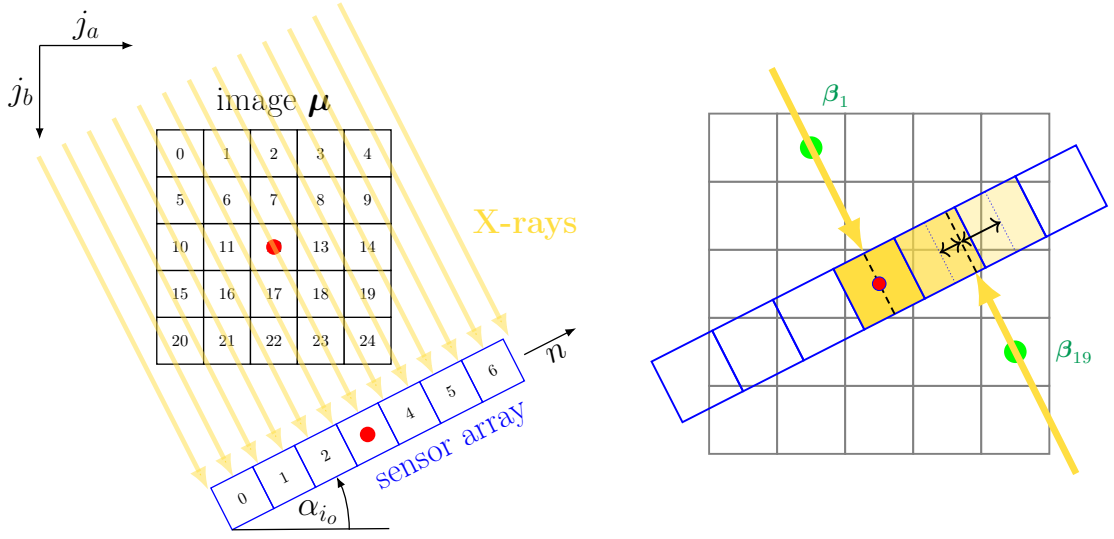

Figure 1: Sketch of the projection of a 5×5-image $\boldsymbol{\mu}$ onto a 7 pixel sensor array that has been rotated around the center of the image by the angle α_{i_o} *(left)*. Small numbers reflect the indices j within $\boldsymbol{\beta}$ and i_p within one slice of $\mathbf{s}$, respectively. While both the X-ray source and the sensors are naturally situated outside the tissue, the projection can be made easier by virtually placing the sensor at the image center and considering the rays to come from both directions *(right)*. In any case, for each individual coordinate in $\boldsymbol{\beta}$, we trace the ray passing through that point and calculate how much of its intensity is collected by which sensor pixels.

1 Constructing the matrix X (25 points)

With only one projection, solving equation 1 is ill-posed and a useful reconstruction of $\boldsymbol{\beta}$ is impossible. The problem becomes well-posed by using sufficiently many projections of $\boldsymbol{\beta}$ under different angles α, whose recordings are just concatenated in the flattened measurement vector $\mathbf{y}$ or the 2D sinogram (figure 3), respectively. Constructing the corresponding weight matrix $\mathbf{X}$ is the main difficulty of this exercise.

Implement a function `X = construct_X(M, alphas, Np = None)` which, given the desired tomogram size $D = M \times M$, a list of measurement angles $(\alpha_0, \alpha_1, \ldots)$ (in degrees) and an optional sensor resolution N_p, returns the matrix $\mathbf{X}$, which completely describes the setup of our simplified CT scanner.

If `Np` is not given, choose a value large enough to fit the diagonal of the image $\boldsymbol{\beta}$, like $N_p = \lceil \sqrt{2} \cdot M \rceil$. Choosing an odd number of sensor elements will make it easier to align the image coordinates with the coordinates of the sensor array: You can define a common coordinate origin and then find ray intersections by projecting the tomogram's pixel coordinates along the current ray orientation onto the rotated sensor array, see figure 1.

In the examination scenario for patient H.S., we have $M = 195$, i.e. the tomogram has $D = 38025$ pixels. The sensor has $N_p = 275$ bins, and intensities have been measured at $N_o = 179$ projection angles, resulting in a response vector of size $N = 49225$. Thus, matrix $\mathbf{X}$ has shape 49225×38025 (almost 2 billion entries), and you will run into trouble if you implement `construct_X()` naively.

Fortunately, the intensity of each ray is only distributed among the *two* sensor elements closest to the ray's intersection with the sensor. Consequently, each image pixel can contribute to no more than two sensor elements under any given measurement angle (see right side of figure 1) and the vast majority of entries of $\mathbf{X}$ are actually zero. This is called a *sparse matrix*, and the scipy module `scipy.sparse`[1] provides powerful tools to take advantage of this property: it allows you to save memory by offering spare matrix classes that store only the non-zero entries, and to skip unnecessary calculations by offering specialized implementations of linear algebra functionality. Specifically, the class `coo_matrix` ("COOrdinate" format) is recommended to create $\mathbf{X}$, and the class `csc_matrix` ("Compressed Sparse Column" format) is best for solving the linear system. Converting between different matrix types is easy and results in faster code than using the same type throughout.

However, using a sparse matrix is not enough to make your code efficient. You should also vectorize the function `construct_X()` (cf. exercise 1). A good way of doing this is to create an array $\mathbf{C} \in \mathbb{R}^{2 \times D}$ holding the coordinates of the tomogram's pixel centers, i.e.

$$\mathbf{C}_{0j} = a_0 + j_a \cdot h \text{ (the } a \text{ coordinate of pixel } j)$$
$$\mathbf{C}_{1j} = b_0 + j_b \cdot h \text{ (the } b \text{ coordinate of pixel } j)$$

with h being the pixel distance and $j = j_a + M j_b$ as defined earlier (`numpy.mgrid` is useful here). The current orientation can be expressed by a unit vector $\mathbf{n}$ along the rotated sensor, and the projection $\mathbf{p}$ of each pixel onto the sensor is simply a linear projection according to

$$\mathbf{p} = \mathbf{n}^\top \mathbf{C} + s_0,$$

where s_0 is the position of sensor element 0 relative to the sensor's coordinate origin. The weights $\mathbf{X}$ for each angle can now be computed from $\mathbf{p}$ by numpy's vector functions. In the end, the valid elements of $\mathbf{X}$ are represented by their indices (i, j) and the corresponding weights. The constructor of class `coo_matrix` expects this information to be provided by three 1-dimensional arrays `i_indices`, `j_indices`, and `weights`, resulting in the call

```
X = coo_matrix((weights, (i_indices, j_indices)), shape=(N, D), dtype = numpy.float32),
```

where we used element type `float32` to save even more memory. Check the `scipy` documentation

[1] Install `scipy` via `conda` as usual.

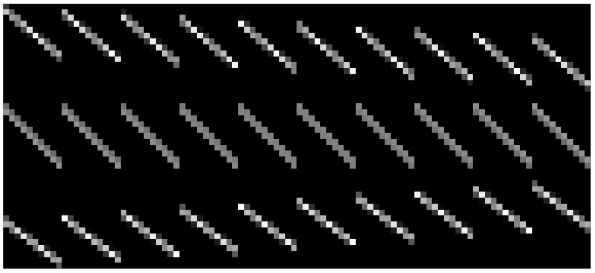

Figure 2: Visualization of matrix $\mathbf{X}$ for $M = 10$, $N_p = 9$ and three projections at angles $(-33, 1, 42)$. You can also find this matrix as `'X_example.npy'` among the uploaded files.

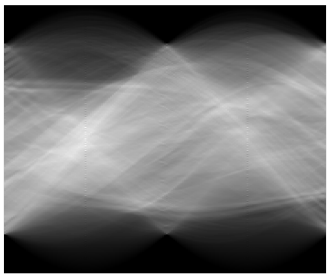

Figure 3: Sinogram visualization of $\mathbf{y}$. Each column shows the sensor response for a different angle $\alpha \in [-90, \ldots, 90]$.

for more details about how this works. In our example implementation, constructing the sparse 49225×38025 matrix takes about 5 seconds on a standard laptop.

As in real-world scenarios, there is some ambiguity left to this task: Where is the origin of the coordinate system, which direction is $\alpha = 0$, and how is $\mathbf{y}$ oriented? The answers to these questions determine the appropriate indexing order and sign of the expressions, and figuring out these details is part of your task (emulating what often happens in the real world). To check whether your matrix construction is correct, visualize the result of `construct_X(10, [-33, 1, 42])` with `pyplot.imshow()` and compare it to the matrix shown in figure 2, which you can download from Moodle as `'hs_tomography/X_example.npy'`[2]. Note that you will have to convert your sparse matrix to a dense `numpy`-array for visualization.

2 Recovering the image (6 points)

The material for this sheet contains the list of angles and the measured sensor data $\mathbf{y}$ for two versions of the experiment. The smaller one was created with $M = 77$, $N_p = 109$ and 90 projection angles. The larger one was created with $M = 195$, $N_p = 275$ and 179 projection angles. The exact set of angles and the corresponding response vectors $\mathbf{y}$ can be downloaded from Moodle in the files `'alphas_77.npy'` and `'y_195.npy'` resp. `'alphas_195.npy'` and `'y_77.npy'` (in folder `'hs_tomography'`). We recommend the smaller version of the data for debugging your code. However, its resolution is insufficient to diagnose the cause for your patient's headache. If you can recognize the head in the small tomogram, your code is probably correct, and you should switch to the larger version.

Exactly how many non-zero entries does $\mathbf{X}$ have? Use `scipy`'s tools to find out and report the *sparsity* of $\mathbf{X}$.

With the ability to construct the matrix $\mathbf{X}$ and having obtained a set of projections $\mathbf{y}$, you are now able to reconstruct the original image $\boldsymbol{\beta}$. You could try to solve the equation system directly via the pseudo-inverse

$$\boldsymbol{\beta} = (\mathbf{X}^\top \mathbf{X})^{-1} \cdot \mathbf{X}^\top \cdot \mathbf{y},$$

but this ignores sparsity and is very slow.

Fortunately, `scipy` already gives you access to efficient solvers for sparse linear equation systems! Find out how to use `scipy.sparse.linalg.lsqr()` to obtain the least-squares solution to your problem. The solver's low default tolerance parameters `atol = 1e-08` and `btol = 1e-08` lead to high quality solutions, but also long computation times on the large version of the data. A

[2] `'.npy'` is numpy's matrix file format. You can load it with `X_ex = np.load('hs_tomography/X_example.npy')`

good trade-off might be `atol = btol = 1e-05`, but you can decrease the tolerance once your are confident that your code is correct.

Reconstruct the tomogram and plot it as a 2D image. Give a diagnosis on what causes H.S.'s headache and propose a treatment.

(If you didn't manage to correctly construct $\mathbf{X}$ in task 1, you can find a precomputed weight matrix for the smaller version of the experiment in `'hs_tomography/X_77.npy'` and use it for this task.)

3 Minimizing the radiation dose (9 points)

Obviously, as a doctor you do not want to expose your patients to unnecessary radiation. Try to reduce the number of projection angles in a sensible way and visualize how this changes the quality of the reconstruction. In the case of H.S., what would you say is the minimal number of projections that still allows you to resolve the cause of his headache?