Homework sheet 7 2023-06-27

Due date: 2023-07-06 17:59

Delivery format

- Create a folder called "hw07" under the "homework" folder in your git repository.
- Answer all questions to the related to the homework in a file called "README.md" in the above created folder. **The readme must be a complete and standalone solution**. You are highly encouraged to provide instructions on how to recreate your results (e.g. run this script with this dataset, or run through this Jupyter notebook).
- Put all scripts and images you've created to reproduce and comprehend your answers in the same folder. Examples for this, are scripts to create plots, visualization, certain reconstructions to compare different strengths and weaknesses and similar things.
- The concrete implementations of the homework, must be put into the "aomip" folder. You are free to choose the structure, layout and form of your implementation.
- Once you've finished your homework, commit your changes and create a git tag with `git tag -a "hw07" -m "Tag for homework 7"` and push your changes. This needs to be tagged and pushed to your "master" branch before the deadline ends. Please use this exact command.
- Finally, once you push the changes with the tag to your repository. Create a release in GitLab. You find it in your project, under the menu 'Deployments'.

Note

Generally, we expect and reward both explorativ work and good software engineering. Hence, we highly encourage you to look at the provided resources, do your own research and try different things and play around with the tools provided to you during the course.

Without explicitly permissions you are not allowed to pull any other dependencies. If you need dependencies to load or import a dataset, please get in touch with us and let us know the limitations. If you need to open `TIFF` files, use the already included dependency `tifffile`. Use it with `tifffile.imread`. See some examples at https://pypi.org/project/tifffile/#examples. This can also handle 3D TIFF-files, which makes this quite handy.

Homework 1: More Proximal Operators

As we have often already done, we can look at 2D images and 3D volumes as linearized vectors of size n (where, $n = n_x * n_y$ for 2D images and $n = n_x * n_y * n_z$ for 3D volumes). Further let d be the dimension of the image or volume The gradient of such an image or volume f is then given as:

$$\nabla f = \begin{bmatrix} \nabla f_1 \\ \nabla f_2 \\ \vdots \\ \nabla f_d \end{bmatrix} \tag{1}$$

with $\nabla f_i \in \mathbb{R}^n$. Then $\nabla f \in \mathbb{R}^{d \times n}$ I.e. ∇f can be seen as a matrix. Given such a matrix, once can compute the $\ell_{p,1}$-norms:

$$\lambda ||X||_{p,1} = \sum_{j=0}^{m} ||x_j||_p \tag{2}$$

where, x_j is the j-th column, i.e. one computes the p-norm of each column, and then the sum of these colume-wise sums.

So why is that interesting? The special cases for the $\ell_{2,1}$ and $\ell_{1,1}$ are commonly used for anisotropic and isotropic TV. Hence, this will become important later. Now we need the proximal operator of the $\ell_{2,1}$-norm is:

$$\text{prox}_{\lambda||\cdot||_{2,1}}(x_j) = (1 - \frac{\lambda}{\max\{||x_j||, \lambda\}})x_j \quad \forall j \tag{3}$$

The proximal operator of the $\ell_{1,1}$ is simply:

$$\text{prox}_{\lambda||\cdot||_{1,1}}(x) = \text{SoftThreshold}(x, \lambda) \tag{4}$$

This is due to the separability of the ℓ_1 norm.

Homework 2: ADMM

As you have seen in the lecture ADMM solves quite general kind of problems:

$$\min f(x) + g(z) \quad \text{s.t. } Ax + Bz = c \tag{5}$$

However, it is very hard to solve this general problem. We will focus on a special case, the so called linearized ADMM. The linearized version assumes $A = K$ (we use K to have a different name than A) to be any linear operator, $B = -I$ and $c = 0$. Hence, the optimization problem reduces to:

$$\min f(x) + g(Kx) \tag{6}$$

The update steps for the linearized version is given as:

$$\begin{aligned} x_{k+1} &= \text{prox}_{\mu f}(x_k - \frac{\mu}{\tau} K'(Kx_k - z_k + u_k)) \\ z_{k+1} &= \text{prox}_{\tau g}(Kx_{k+1} + u_k) \\ u_{k+1} &= u_k + Kx_{k+1} - z_{k+1} \end{aligned} \tag{7}$$

To ensure convergence, $0 < \mu \leq \frac{\tau}{||K||_2^2}$ (see below for some practical tips).

So how can you solve an X-ray CT problem with this? Let us start with an easier problem, like LASSO.

iii) LASSO Problem

We want to solve

$$\frac{1}{2}||Ax - b||_2^2 + \beta||x||_1 \tag{8}$$

using ADMM implemented as above. We choose the simple approach: let $f(x) = \beta||x||_1$, and $g(x) = \frac{1}{2}||Ax - b||_2^2$, here $K = A$ (i.e. the X-ray transform).

For the LASSO problem, run the following experiment:

1. Compute the norm of K using the power method

2. set τ to some value
3. Use $\lambda = \frac{0.95\tau}{||K||_2^2}$,
4. Perform the reconstruction using these parameters

Do this for a range e.g. `taus = np.logspace(-3, 6, 10)` (Maybe you need to adjust that, or make it more finegrained). Show the reconstructed images using this method for both formulations and each τ.

In theory, you can do something similar, fix λ, compute a τ based on that and perform the reconstruction, and sometimes that works better. If in doubt, you can try this, but is left optional for this exercise.

Use a dataset you know already, such as the challenge dataset. Most likely you should use a 2D dataset, as you need to perform many reconstructions.

So, but we can already solve the LASSO problem, we want to solve more interesting problems. Lets work on that.

ii) TV regularization

We want to solve

$$\frac{1}{2}||Ax - b||_2^2 + \beta||\nabla x||_1 \tag{9}$$

However, now we can not easily solve it with the approach above, so we need some more. Namely, we need stacked operators and separable functions.

Stacked operators Similar to the transformation performed for CG with Tikhonov regularization, we define K to be a stack of k operators:

$$K = \begin{bmatrix} K_1 \\ \vdots \\ K_k \end{bmatrix} \tag{10}$$

Applying a vector to to K is given by

$$Kx = \begin{bmatrix} K_1 x \\ \vdots \\ K_k x \end{bmatrix} \tag{11}$$

Applying a vector to the adjoint is given as:

$$K'y = K' \begin{bmatrix} y_1 \\ \vdots \\ y_k \end{bmatrix} = \sum_{i=1}^{k} K_i' y_i \tag{12}$$

Implement this functionality.

Separable Sum The separable sum h of the functionals n functionals f_i is defined as:

$$h(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} f_i(x_i) \tag{13}$$

It is defined for functions with the same range (in our case $\mathbb{R}$). Importantly, the proximal operator is defined as:

$$\text{prox}_{\sigma h}(x_1, x_2, \ldots, x_k) = \begin{bmatrix} \text{prox}_{\sigma f_1(x_1)}, \\ \text{prox}_{\sigma f_2(x_2)}, \\ \vdots \\ \text{prox}_{\sigma f_k(x_k)} \end{bmatrix} \tag{14}$$

Note that here, x_i is a vector, not a scalar.

Implement both, and specifically the proximal operator.

Reconstructions Now with the stacked operator and the separable sum, you can define an operator as e.g.

$$K = \begin{bmatrix} A \\ \nabla \end{bmatrix} \tag{15}$$

(here A is the Xray transform). With $g_1(x) = \frac{1}{2}||x - b||_2^2$ and $g_2(x) = \lambda||x||_{1,1}$, you can define the separable sum (and it's proximal operator), such that you can solve the anisotropic TV problem. Replace the $\ell_{1,1}$ by the $\ell_{2,1}$ norm to solve with isotropic TV. You can set $f(x) = 0$, or use a non-negativity constraint.

For this part of the homework, try to find both synthetic and real datasets, where you can spot a difference between isotropic and anisotropic TV. Combine this with a discussion on the influence of the parameters τ and λ. As a final part, make a convergence analysis for the synthetic cases. Plot the reconstruction error and analyise the convergence rate of ADMM.

A good synthetic phantoms to spot TV regularization and specifically over regularization, are usually smooth and have slow transitions from one intensity to another, rather than sharp edges.A translated example from the MATLAB package AIR Tools II, by Per Christian Hansen, is given below. However, you are encourage to find or design other well suited functions as well. You can see it in Figure 1.

```
def smooth(N):
    I, J = np.meshgrid(np.arange(N), np.arange(N))
    sigma = 0.25 * N
    c = np.array([[0.6*N, 0.6*N], [0.5*N, 0.3*N], [0.2*N, 0.7*N], [0.8*N, 0.2*N]])
    a = np.array([1, 0.5, 0.7, 0.9])
    img = np.zeros((N, N))
    for i in range(4):
        term1 = (I - c[i, 0])**2 / (1.2 * sigma )**2
        term2 = (J - c[i, 1])**2 / sigma**2
        img += a[i] * np.exp(-term1 - term2)
    return img
```

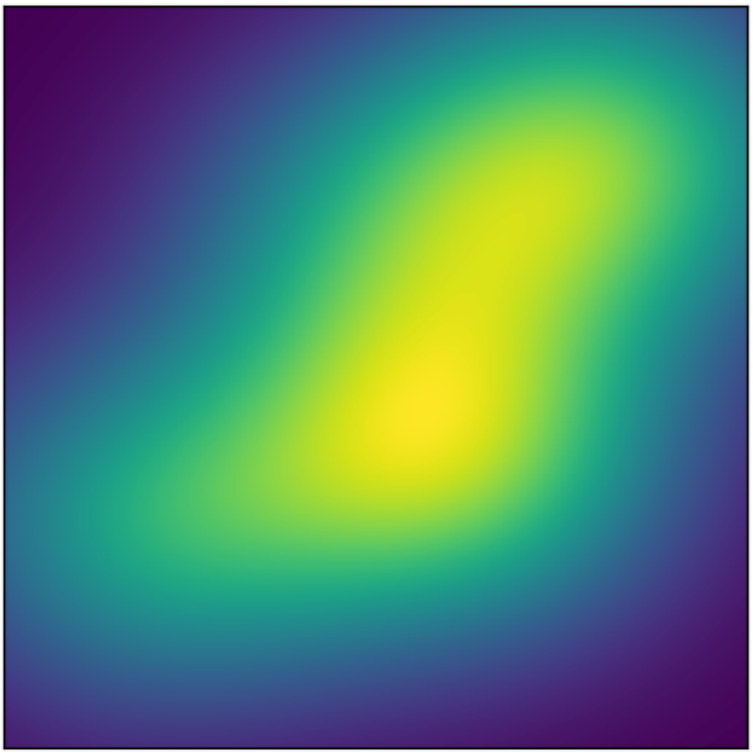

Figure 1: Example a smooth function to test TV regularization on

Homework 4: Challenge data

Apply ADMM and TV-regularization to the challenge dataset. Try to achieve even better scores in limited angle cases (i.e. start playing around with 60° and 30°). Show your results and the parameters you have used. Also go in depth on the parameter search, similar to the other parts of this exercise.

Appendix: Finite Differences

As we are dealing with the gradient in this homework a lot, I want to ensure, that you have a well working finite difference implementation going. As far as I gathered from some feedback, some have struggled with it due to large memory consumption and so on. The code below gives a brief snipped of an operator like class, that you can use to apply the gradient to an n-D array. For many cases you might want to adapt it, but that is more boilerplate and left to you as an exercise. Further, note that the code may contain bugs, but you should be able to extract the idea from it!

```
def forward_diff(x, axis=-1, sampling=1):
    """Apply forward differences to any chosen axis of the given input array

    Notes
    -----
```

```
    For simplicity, given a one dimensional array, the first-order forward
    stencil is:

    .. math::
        y[i] = (x[i+1] - x[i]) / \Delta x
    """
    # swap the axis we want to calculate finite differences on,
    # to the last dimension
    x = x.swapaxes(axis, -1)

    # create output vector
    y = np.zeros_like(x)

    # compute finite differences
    y[..., :-1] = (x[..., 1:] - x[..., :-1]) / sampling

    # swap axis back to original position
    return y.swapaxes(axis, -1)


def adjoint_forward_diff(x, axis=-1):
    """Apply the adjoint of the forward differences to any chosen axis of
    the given input array
    """
    x = x.swapaxes(axis, -1)
    y = np.zeros_like(x)
    y[..., :-1] -= x[..., :-1]
    y[..., 1:] += x[..., :-1]
    return y.swapaxes(axis, -1)


class FirstDerivative:
    """Operator which applies finite differences to the given dimensions

    Notes
    -----

    This operator computed the finite differences lazily, i.e. doesn't construct
    a matrix and consumes unnecessary memory
    """

    def apply(self, x):
        dim = x.ndim
        grad = np.zeros((dim,) + x.shape)
        for i in range(dim):
            grad[i, ...] = forward_diff(x, axis=i).copy()
        return grad

    def applyAdjoint(self, x):
        grad = np.zeros(x.shape[1:])
        for i in range(x.shape[0]):
```

```
            grad[...] += adjoint_forward_diff(x[i], axis=i).copy()
        return grad

# usage:
grad = FirstDerivative()
dx = grad.apply(x)
adx = grad.applyAdjoint(dx)
```