Exercise Sheet 2 - Parallel-Beam

Yixing Huang, Fabian Wagner, Andreas Maier

May 12, 2022

In this exercise, we will implement a parallel-beam filtered back-projection (FBP) algorithm, and test it on the phantom which we created in Exercise 1. It is recommended to use smaller grid sizes for development (e.g. [64, 64]) to avoid long execution times, and test with a full scale phantom at the end.

1. **Sinogram Generation:** Before you can perform a back-projection, you first need to generate a sinogram of the phantom, i.e. simulate an acquisition. For this purpose, implement the Radon transform. Projections should be generated using the ray-driven approach. The following input parameters are mandatory: number of projections, detector spacing, number of detector pixels and the angular range. You can assume that your source-to-detector distance is large enough to cover the full phantom. To read out your phantom at arbitrary positions, you need interpolation. Set the origin of the sinogram Grid to 0 degrees in one dimension and to the middle of the detector in the other dimension (you can e.g. add a method `set_origin()` to your Grid class).

   Task: Create and implement the method
   `create_sinogram(phantom, number_of_projections, detector_spacing, detector_sizeInPixels, angular_scan_range)`

2. **Backprojection**: Now that you have created the sinogram, you can implement a pixel-driven back-projector. For each rotation angle, the back-projector needs to project the position of each pixel in the result image onto the detector, read-out the value and add it to the corresponding pixel. How does the backprojection result look like? Explain this effect.

   The back-projection should be able to deal with the sinograms you generated in Exercise 1. That means you should incorporate, e.g., the detector spacing, and other variables from your sinogram. It takes the size and pixel spacing of the reconstructed image as inputs. The origin can be assumed to be in the center of the image.

   Task: Create and implement the method
   `backproject(sinogram, reco_size_x, reco_size_y, spacing)`

3. **Ramp and RamLak filter**: Implement a row-wise ramp and a RamLak filter on your Grid class. Start implementing the ramp filter in Fourier

domain. Afterwards, implement the RamLak filter in spatial domain. Compare both implementations.

Ramp filtering is a convolution of each detector line of your sinogram with the ramp-filtering kernel. Because the ramp filtering kernel is best known in Fourier domain we perform the convolution by element-wise multiplication in the Fourier domain. Some details are important to define the ramp-filtering kernel:

(a) FFT algorithms swap the positive and negative frequency axes. The vector you obtain by the forward FFT starts with the zero frequency up to the positive maximum located in the center of the vector, then it continues from the negative maximum to almost zero at the end of the vector. (Hint: That means if you visualize your kernel using the `show()` method, it should look like a pyramid.)

(b) For proper ramp-filtering we need to apply zero padding. Bear in mind that the ramp filter needs to be defined over the full length in Fourier domain, i.e., the 1024 values. The same holds for the Ram-Lak filter except that it is implemented in spatial domain and then Fourier-transformed.

(c) To compute the ramp kernel you need to know the spacing of your frequency axis. This depends on the amount of zero-padding and your detector spacing. It can be computed by:

$$\Delta f = \frac{1}{\Delta s \cdot K} \ , \qquad (1)$$

where Δf is the frequency spacing, Δs is the detector spacing and K is the length of your signal after zero-padding.

(d) Recall complex multiplication!

(e) RamLak filters are initialized in spatial domain. Use the formula from the lecture to initialize the filter.

Task: Create a helper method `next_power_of_two(value)`. This method should round up `value` to the next power of two, multiply that value by two, and return that value.

Task: Create and implement the method
`ramp_filter(sinogram, detector_spacing)`

Task: Create and implement the method
`ramlak_filter(sinogram, detector_spacing)`

4. **Filtered Back Projection**: Combine the backprojector and the filter to create a reconstruction from your sinogram.

   Task: Combine your previously implemented methods to obtain a reconstruction: sinogram generation, filtering, and backprojection.