Advanced simulation of CT measurements using physical coordinate systems

Goal:

This exercise sheet has two goals: 1) introduce you to the idea of how to overlay a normal measurement without units with a physical coordinate system and 2) introduce you to a better way to simulate the measurement process in CT.

Please note:

Although this exercise sheet is voluntary, you should **really** try to solve it. The projects you have to solve in a few weeks will be slightly more complicated, so please use this as an opportunity to evaluate and improve your programming skills.

Overview:

The measurement process
In 1917 Johan Radon showed that it is possible to reconstruct the inner structure of an object if a complete set of *line integrals* (or projections) through this objects is available. This insight is the basis of all tomographic imaging modalities.

In the general context of tomography, an integral (in the form of a straight line or even a curve or a surface) is something which "adds" or "collects" a specific material property of the object along the line or curve.

In the case of CT (X-ray Computed Tomography), the material property of interest is called the X-ray attenuation coefficient. It is related to the amount of attenuation of an X-ray beam on its way through the object. The distribution of this attenuation coefficient in CT is usually given as $\mu(x)$, which describes the attenuation coefficient at position x inside the object. Since the X-ray attenuation is directly related to the morphological structure of the object its exact spatial distribution can be used to look into the object.

Therefore, the main goals of CT are 1) to measure a complete set of projections through the object to gather information about the distribution of the X-ray attenuation and 2) to actually reconstruct the attenuation distribution from the measurement to allow the user to look into the object.

Unfortunately, it is often not possible to measure these integrals directly. Therefore, for each tomographic application one has to find an indirect way of measuring them. In CT this is done with Beer's Law. It describes how many photons arrive at the detector after crossing an object on a certain line:

$$I = I_0 * \exp\left(-\int_L \mu(\vec{x})\, dx\right)$$

In this equation…:

- L describes the line between the source and a single dexel (short version of: detector element),
- I is the number of photons arriving at the detector
- I_0 is the number of initial photons leaving the tube
- μ is the attenuation coefficient at position $\vec{x}$.

If I and I_0 are measured, then the so-called projection value can be calculated:

$$p = \int_L \mu(\vec{x})dx = -\log\left(\frac{I}{I_0}\right).$$

Based on the definition of the projection value, the forward projection for a single pencil beam through an image μ can be seen as a line integral, which 'collects' all the attenuation values along the path of the beam.

In this exercise, it is your task to implement this integration/measurement function in order to create a complete sinogram of an arbitrary image.

Coordinate systems
In order to create a somehow realistic forwardprojection, it is important to understand how to connect a simple unit-less Matlab array of numbers to the "real world", in which sizes and distances are given in *mm* or c*m* and not in *pixel*.

A good example for this would be a reconstructed image from a clinical CT: while the size of these reconstructed images is almost always 512*512 *pixel*, their actual size in *mm* depends on the region of interest which is reconstructed. It might be 100*100 mm, or 350*350 mm or even 50*245 mm. Therefore, the size of an image in pixel is not enough information for correct image processing.

In this exercise you will learn one way to solve this problem.

Part 1: The base coordinate system and its orientation

A simple 2D array in Matlab is stored as a one dimensional stream of numbers. Since such a stream of numbers does not have an inherent concept of up/down/left/right, one can align the image in arbitrary and non-standard ways. Fortunately, instead of establishing its own system, Matlab basically uses the same matrix conventions as everyone else. This means that the upper left element of an image/array has the index (1, 1) and the element in the *r*-th row and the *c*-th column has the index (*r*, *c*). A call of the function `size()` for a 2D array returns [n_{rows}, n_{cols}].

It should be noted that *r* and *c* are used as indices instead of the usual *i* and *j* because it makes the code more readable and greatly reduces the likelihood of accidently swapping the axes.

Based on this matrix notation we now introduce the first coordinate system which is called the r/c systems (related to the concept of rows and columns in an array). In simple terms, the r/c system is just a continuous extension of the normal array indices.

The exact orientation of the r/c system is shown on the left side of Figure 1. As it can be seen its "origin" has the position (1, 1) and it is centered exactly in the middle of the upper left pixel. This means that the position of the center of each pixel is positioned exactly at (r_idx, c_idx) in the r/c system. This means that the center of the pixel (4, 3) has the position (4, 3) in the r/c system. It also means that its lower left corner of this pixel has the position (4.5, 2.5) in the r/c system (see Figure 1, left side)

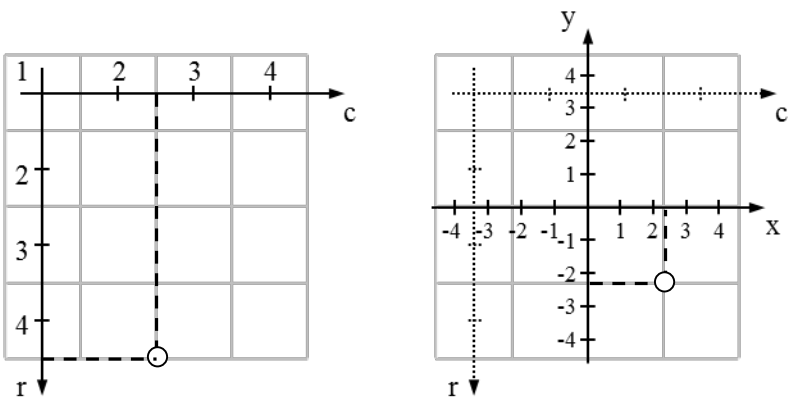

Figure 1: Left side: the *r*/*c* system, which is used to access the array elements. Right side: The world coordinate system *x*/*y*.

A very important coordinate system, which is added on top of the r/c system, is shown on the right hand side of Figure 1. It is called the world coordinate system and uses the indices x and y. It is centered exactly in the middle of the pixel image and has the unit of mm. The link between the two systems is the width of a pixel in mm. In addition, the origin of the world coordinate system is also the rotation center of the X-ray tube and detector.

If the size of the image in pixel is known and the size of a single pixel in mm is defined then the position and size of the world coordinate system on top of the r/c system is unique. Therefore, we can assign each point in the pixel image to a position in the real world and the other way around.

Although initially this seems to be a complicated solution, it actually makes life much easier. For example the geometry of the scanner can now be given in mm instead of pixel. In addition, the rotation of the source can be easily implemented with a simple rotation matrix since the origin of the coordinate system and the rotation axis of the scanner are in the same position.

In order to connect the coordinate systems, several helper functions will be developed, which convert the coordinate of a single point from one system to the other. For example the position of the point in the right image in Figure 1 is x = 2.2 mm and y = -2.2 mm. Therefore, in the r/c coordinate system it has the position r = 3.5 and c = 3.5.

Part 2: Adding the *x*/*y* system

In this step you will create some helper functions, which convert coordinates from the world coordinate system into the r/c system.

Create a file called x_to_c.m, which contains this function:

```
function pos_c = x_to_c(pos_x, data, pixel_size_mm)
```

In addition create a file called y_to_r.m with this function:

```
function pos_r = y_to_r(pos_y, data, pixel_size_mm)
```

Both functions take a position on one axis of the x/y system and calculate the position on the corresponding axis in the r/c system. Implement this functionality and the two inverse cases.

Hint: The size of a pixel is 2.2 mm in Figure 1, left image. Therefore, the position x = 0 mm, y = 0 mm should be converted into r = 2.5, c = 2.5. The position x = -3.3 mm, y = 3.3 mm should be converted into r = 1, c = 1.

For debugging purpose create a Matlab script similar to this:

```
data = zeros(5,5)
x_to_c( 6.0, data, 3.0) % result: 5
x_to_c( 0.0, data, 3.0) % result: 3
x_to_c(-7.5, data, 3.0) % result: 0.5
y_to_r( 0.0, data, 6.0) % result: 3
y_to_r(15.0, data, 6.0) % result: 0.5
```

Part 3: Create the X-ray source

In part 3 to 6 you will implement the three ‘components’ of a CT: The source, the detector (which consists of a lot of small detector elements) and the measurement function (for the beam between the tube and a single detector element). In part 7 to 9 you will then combine all three components to get a fully working virtual CT.

In order to simplify the task you first can create a function called rotate.m, which takes a coordinate and rotates it around a given angle. To rotate the coordinates use a 2*2 rotation matrix, which can be found on Wikipedia. The rotation should be clockwise as in Figure 2.

```
function [new_x, new_y] = rotate_xy(x, y, angle_deg)
```

A second helper function should calculate the position of the X-ray source for a given rotation angle in the world coordinate system. This simulates the rotation of the tube around the main axis. Create a Matlab-file called tube_position_xy.m which contains the function

```
function [tube_x, tube_y] = tube_position_xy(FCD_mm, angle_deg)
```

`FCD_mm` is the Focus-center-distance and describes the distance between the rotation center (the origin of the world coordinate system) and the source. At 0°, the tube position is `[0, FCD_mm]`.

For debugging, set the variable `FCD_mm` to 10. For `angle_deg=0` the result should be `[0, 10]`. For `angle_deg=135` the result should be approx. `[7.07, -7.07]`. For `angle_deg=270` the result should be `[-10, 0]`. For `angle_deg=-45` the result should be approx. `[-7.07, 7.07]`.

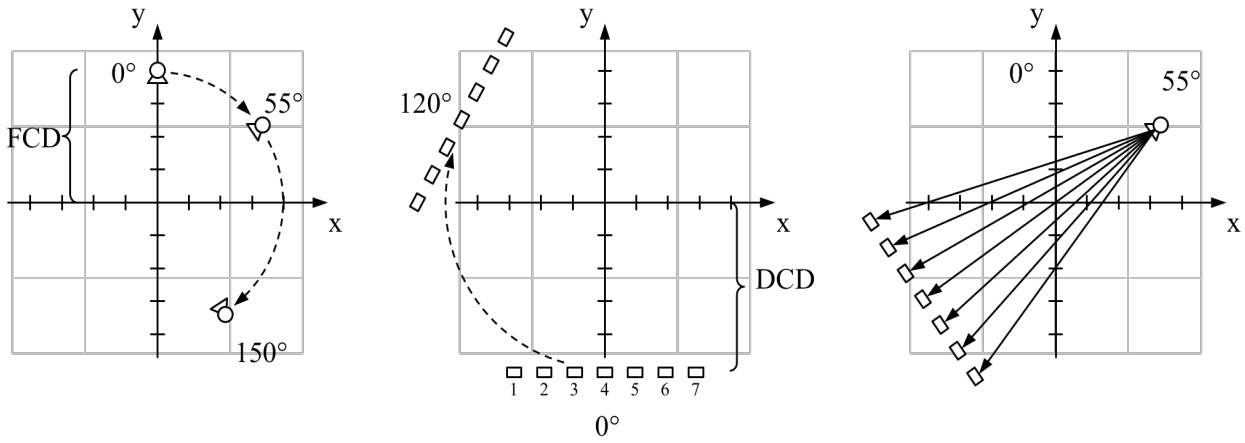

Figure 2: Left image (Part 3): The basic geometry of the X-ray source. At 0°, the source sits at x = 0 mm and y = FCD mm. For increasing angles, it rotates clockwise. Middle image (Part 4): The basic geometry of the detector. For 0° the y coordinate of all dexel is y = - DCD mm. Right image (Part 4): A complete example for the view angle of 55°.

Part 4: Create the detector

The detector array in our virtual CT consists of a number of small detector elements. To avoid confusion about the meaning of the word ‘detector’, in this exercise ‘detector’ describes the complete detector array. When we deal with a single element of the detector array, then this element is called a detector element or a dexel (similar to a pixel).

In real life application, a dexel always has a certain width. Since this is complicated to work with, we idealize the systems and concentrate only on the center of a dexel. This means that we make the ‘measurement’ between the position of the X-ray source and the center of the detector element. In order to make a more realistic simulation one could simulate several beams per detector element and do an averaging of the values.

The width of a dexel is still important because it is responsible for the spacing between the single dexel. If for example the width of a dexel is given as 2 mm, then the centers of two neighboring dexel are separated by 2 mm.

In this part you should write a function which creates the position of all the dexel of the detector for a given rotation angle. This simulates the rotation of the detector around the main axis. For an exact description of the detector, its orientation and the rotation direction see Figure 2.

Create a function called detector_position_xy.m, which contains the function:

```
function [det_x, det_y] = detector_position_xy(DCD_mm,
                                               angle_deg,
                                               n_dexel,
                                               dexel_size_mm)
```

`det_x` and `det_y` are 1D vectors with n entries, which contain the positions of each dexel. `DCD_mm` is the Detector-center-distance and describes the distance between the rotation center and the detector.

`n_dexel` is the number of dexel and `dexel_size_mm` is the width of a detector element. Therefore, the whole width of the detector is defined as `dexel_size_mm*n_dexel`.

In order to implement this function, in a first step, you always create the detector for 0 degree. You know that the y-component of the position of each detector element is – DCD. The x-position of each detector element you can calculate by using `linspace()` and the information provided above. In a second step you rotate these coordinates with a standard 2*2 rotation matrix around the desired angle.

For debugging purposes use pen and pencil and draw an example for the case `DCD_mm = 100`, `angle_deg = 45`, `n_dexel=10`, `dexel_size_mm=10` and compare this to your results. Do the same for `angle_deg=235`. It is also helpful to use the functions `hold all`, `plot(x,y)`, `xlim([-150, 150])` and `ylim([-150, 150])` to plot the detectors for several angles.

Part 5: A simple line integral through an array

After you have implemented the source and detector part of the CT, you will now implement the actual measurement function. To simplify the implementation, the measurement function will only work for a combination of one source position and one detector element position. The implementation for the full detector with all elements is done in part 6 and 7.

In this part you will implement the actual measurement process for a single beam by implementing the line integral equation from the introduction on the simple r/c grid. Since it cannot be implemented in its continuous form, it must be approximated by a sum:

$$p = \int_L \mu(\vec{x})dx \approx \sum_N \mu(\vec{x})\Delta s$$

Again, L describes the beam and μ is the attenuation coefficient at position $\vec{x}$. The term Δs is a step size and controls the degree of approximation. If the step size is decreased, the approximation becomes more accurate. N is the number of data points, which are collected. It is calculated by dividing the source-dexel distance by the step size Δs. The basic principle of this summation is shown in Figure 3.

One way to implement this sum is to describe the beam with the equation

$$\vec{p} = \vec{b} + s * \vec{d},$$

where $\vec{p}$ is an arbitrary point on the line, $\vec{b}$ is the starting point, $\vec{d}$ is a unit vector which describes the direction of the line and s is an arbitrary number. By changing s, one can reach any point on the line.

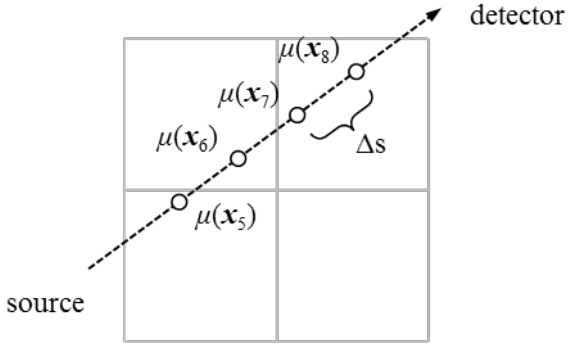

Figure 3: The continuous line integral is replaced by a sum of values. The color μ at a point $\vec{x}$ is multiplied with the step size Δs and the result is added to p. If this is done over the whole beam, the line integral is approximated. Note: The colors at position 7 and 8 are the same as the positions are in the same pixel.

As a first step implement a function, which does the summation on an image for a given start- and endpoint. Create a file called line_integral_rc.m which contains this function:

```
function p = line_integral_rc(data,
                              source_r, source_c,
                              dexel_r, dexel_c)
```

The input of this function is a 2D image in the form of an array, the position of the source in r/c coordinates and the position of a single detector element in r/c coordinates. The output is the calculated projection value.

In order to calculate *p*, you should roughly follow these steps:

- Initialize *p* to 0.
- Calculate the vector pointing from the source to the detector.
- Use this vector to calculate the distance between source and detector.
- Use both information to create a normalized direction vector $\vec{d}$.
- Choose a step size Δs. It should be small enough that each pixel is hit at least three or four times.
- Use a `for` or `while` loop to increment *s* (see the beam equation above) from 0 to the source-detector distance. For each increment of *s*
    - calculate the current position on the line
    - check if this point is inside the image.
        - If it is inside the image, then look up the color at this position (use `round()` to get the array indices), multiply it by Δs and add it to *p*.
        - If it is outside the image, do nothing and continue the loop.
- Return p.

For debug purposes write a Matlab script similar to this

```
data = [0, 0, 0, 0
        0, 5, 2, 0
        0, 1, 3, 0
        0, 0, 0, 0];
p = line_integral_rc(data, source_r, source_c, dexel_r, dexel_c);
```

and then check if the following starting/end points result in the given projection value

```
source_r=1.7; source_c=4.5; dexel_r=1.7; dexel_c=0.5; % p -> 7.15
source_r=3.0; source_c=0.5; dexel_r=3.0; dexel_c=4.5; % p -> 4.15
source_r=4.5; source_c=0.5; dexel_r=0.5; dexel_c=4.5; % p -> 4.20
source_r=4.5; source_c=3.5; dexel_r=0.5; dexel_c=1.5; % p -> 9.05
```

Please note: Your number might deviate from the above number. Depending on the step size and your implementation you might have (slightly) different results. The step size Δs should be rather small (0.05 or so) to get good results.

Part 6: Extend the integration function

Since the position of the source and detector is given in mm and not in pixel it would be helpful if the function `line_integral_rc()` could accept also coordinates in the world coordinate systems. In order to do this, you should create a file called `line_integral_xy.m` which contains this function:

```
function p = line_integral_xy(data, pixel_size_mm,
                              source_x, source_y,
                              dexel_x, dexel_y)
```

Use the coordinate-system-converter-function to convert the coordinates from x/y to r/c and then call the function `line_integral_rc()`. In order to let the image values represent attenuation values per cm, multiply the result with `pixel_size_mm/10`.

Part 7: Do simulation for one view

Since you are now able to 1) simulate the position of the tube and the detector for a given configuration and to 2) simulate the measurement for an arbitrary line it is now time to combine all three functions into a CT simulator. Instead of writing a function which creates all measurements from all directions, it makes more sense to first write a program which calculates all measurements for a single view or a single angle.

To do this, write a function with the definition:

```
function P = view(image_data, FCD_mm, DCD_mm, angle_deg, n_dexel,
                  dexel_size_mm, pixel_size_mm)
```

From the parameters first calculate the position of the tube and all detector elements. In a next step you should write a loop, which iterates over all detector elements and measures from the current detector element to the source. Store each single measurement and return them as a 1D vector.

If all parameter are constant and you only change `angle_deg`, this results in the rotation of the source/detector combination around the object.

There is a script on the moodle page called `introduction_3_test_view.m`, which you can download to your working folder. If you also add the image from the last exercise and run the program, the result should look like the plot in Figure 4. This is the projection data the virtual CT scanner measures from one specific direction.

Part 8: Do a complete simulation

Add a final function, which does a full simulation for a given number of angles

```
function sinogram = simulation(image_data, FCD_mm, DCD_mm,
                               angles_deg, n_dexel,
                               dexel_size_mm,
                               pixel_size_mm)
```

`angles_deg` is a vector of all angles which should be used. Usually it is equal to 0:359 (or 0:5:359 to make it run faster). In order to implement this function create a loop, which iterates over all angles, calls `view()` with the appropriate angle and stores the results in a 2D sinogram as in the previous exercise.

The result of the test-script `introduction_3_test_simulation.m` from the moodle page should look like the sinogram in Figure 5.

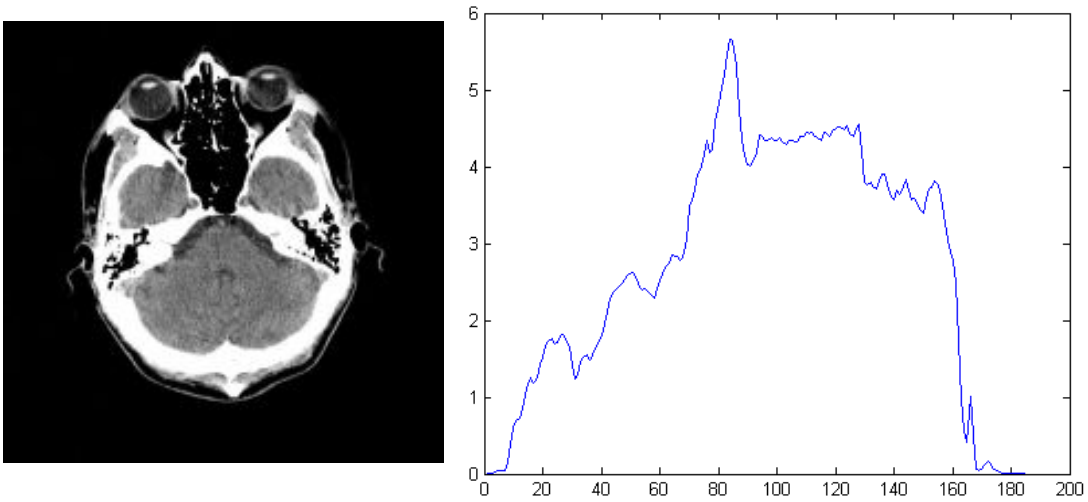

Figure 4: The resulting plot of the test script.

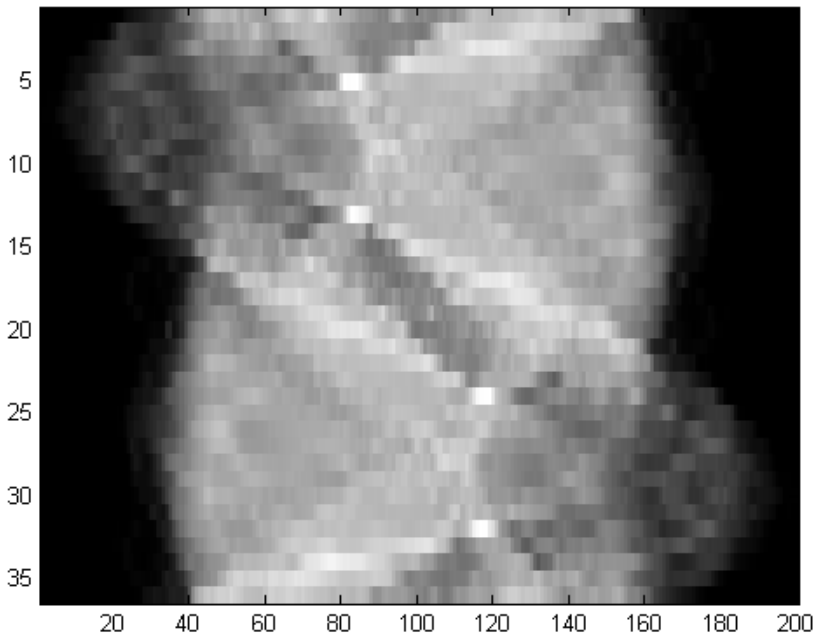

Figure 5: The resulting sinogram generated by the test script.