# Symulator tomografu komputerowego

Mikołaj Kiszka 155973

Maja Komorowska 155844

1. **Cel projektu**

Celem niniejszego projektu było stworzenie symulatora tomografu komputerowego, którego zadaniem jest odwzorowanie podstawowego działania rzeczywistego tomografu. W symulatorze tomografu zastosowano model stożkowy, własną implementację algorytmu Brehensama, transformaty Radona oraz odwrotnej transformaty Radona. W celu zobrazowania działania symulatora stworzono graficzny interfejs pozwalający na testowanie jego działa, a oszacowaniu jakości jego działania pomaga funkcja obliczająca pierwiastek błędu średniokwadratowego. W raporcie zaprezentowano także przykłady działania symulatora oraz ocenę jakości jego działania ze względu na różne parametry wybrane w symulacji.

1. **Opis głównych funkcji programu**

Poniżej opisane są fragmenty kodu zastosowane do stworzenia symulatora tomografu komputerowego. Symulator tomografu komputerowego wykonany został w Pythonie.

1. Użyte biblioteki niestandardowe:

* PIL – otwieradnie, edytowanie i zapisywanie obrazów
* matplotlib – tworzenie wykresów i wizualizacji danych
* IPython – interaktywne środowisko do wizualizacji w Jupyter Notebook
* numpy – operacje na macierzach
* pydicom – odczyt i zapisów plików medycznych w formacie dicom
* ipywidgets – tworzenie interaktywnych kontrolek w Jupyter Notebook

1. Odczyt plików jpg i dicom

Funkcje obsługujące odczyt plików jpg i dicom z zastosowaniem odpowiedniego formatowania i skalowania w celu normalizacji i ułatwienia kolejnych obliczeń

def cutThirdDimension(bitmap):  
 if bitmap.ndim == 3:  
 return bitmap[..., 0]  
 else:  
 return bitmap  
  
def scaleBitmap(bitmap):  
 return bitmap/255.0  
  
def turnSingleJpgIntoBitmap(single\_file):  
 directory = r"Tomograf-images"  
 jpg\_file = os.path.join(directory, single\_file)  
 if single\_file.lower().endswith('.jpg'):  
 img = Image.open(jpg\_file)  
 bitmap = np.array(img)  
 return scaleBitmap(cutThirdDimension(bitmap))  
 else:  
 print('Wrong file format')  
 exit()  
  
def turnSingleDicomIntoBitmap(single\_file):  
 directory = r"Tomograf-dicom"  
 dicom\_file = os.path.join(directory, single\_file)  
  
 if single\_file.lower().endswith('.dcm'):  
 dicom\_data = pydicom.dcmread(dicom\_file)  
 bitmap = dicom\_data.pixel\_array.astype(np.float32)  
 return scaleBitmap(cutThirdDimension(bitmap))  
 else:  
 print('Wrong file format')  
 exit()

1. Zapis do jpg i dicom

Funkcje obsługujące zapis do formatów jpg i dicom. W trakcie zapisu do formatu dicom, zapisywane są także podane bądź wygenerowane metadane obrazu.

def saveBitmapAsJpg(bitmap, filename):  
 img = Image.fromarray((bitmap \* 255).astype(np.uint8)) # Normalize  
 img.save(filename, format="JPEG")  
  
def saveBitmapAsDicom(bitmap, filename, patient\_name, patient\_id, date, comment="Generated DICOM image"):  
 ds = Dataset()  
  
 # Set required DICOM fields (some set to static or random values)  
 ds.PatientName = patient\_name  
 ds.PatientID = patient\_id  
 ds.Modality = "CT"  
 ds.StudyDate = datetime.datetime.strptime(date, "%d.%m.%Y")  
 ds.StudyInstanceUID = pydicom.uid.generate\_uid()  
 ds.SeriesInstanceUID = pydicom.uid.generate\_uid()  
 ds.SOPInstanceUID = pydicom.uid.generate\_uid()  
 ds.SOPClassUID = pydicom.uid.CTImageStorage  
 ds.ImageType = ["ORIGINAL", "PRIMARY", "AXIAL"]  
 ds.InstanceNumber = str(random.randint(1, 100))  
 ds.ImagesInAcquisition = "1"  
 ds.FrameOfReferenceUID = pydicom.uid.generate\_uid()  
  
 # Convert bitmap to DICOM pixel array  
 ds.Rows, ds.Columns = bitmap.shape  
 ds.PhotometricInterpretation = "MONOCHROME2"  
 ds.SamplesPerPixel = 1  
 ds.BitsAllocated = 16  
 ds.BitsStored = 16  
 ds.HighBit = 15  
 ds.PixelRepresentation = 0  
 ds.PixelData = (bitmap.astype(np.uint16)).tobytes()  
  
 # Set DICOM file meta information  
 file\_meta = pydicom.dataset.FileMetaDataset()  
 file\_meta.MediaStorageSOPClassUID = pydicom.uid.SecondaryCaptureImageStorage  
 file\_meta.MediaStorageSOPInstanceUID = pydicom.uid.generate\_uid()  
 file\_meta.TransferSyntaxUID = pydicom.uid.ExplicitVRLittleEndian  
 file\_meta.ImplementationClassUID = pydicom.uid.generate\_uid()  
 file\_meta.ImplementationVersionName = "PYDICOM 2.0.0"  
  
 ds.file\_meta = file\_meta  
  
 ds.ImageComments = comment  
  
 # Save as DICOM file  
 ds.is\_little\_endian = True  
 ds.is\_implicit\_VR = False  
 ds.save\_as(filename, write\_like\_original=False)

1. Algorytm Brehensama

Funkcja zawierająca implementacją algorytmu Brehensama służącego do symulacji promieni od emiterów do detektorów

def bresenhamAlgorithm(x1, y1, x2, y2):  
 points = []  
 x = x1  
 y = y1  
 dx = abs(x2 - x1)  
 dy = abs(y2 - y1)  
  
 # Direction of the algorithm  
 xi = 1 if x1 < x2 else -1  
 yi = 1 if y1 < y2 else -1  
  
 # First pixel  
 points.append((x,y))  
  
 if dx > dy:  
 # Algorithm operates on the leading axis OX  
 d = 2\*dy - dx  
 while x != x2:  
 # Incrementing D according to algorithm  
 if d >= 0:  
 # Move diagonally  
 x += xi  
 y += yi  
 d += 2\*(dy-dx)  
 else:  
 # Move horizzontally  
 d += 2\*dy  
 x += xi  
 points.append((x,y))  
 else:  
 # Algorithm operates on the leading axis OY  
 d = 2\*dx - dy  
 while y != y2:  
 # Incrementing D according to algorithm  
 if d >= 0:  
 # Move diagonally  
 x += xi  
 y += yi  
 d += 2\*(dx-dy)  
 else:  
 # Move vertically  
 d += 2\*dx  
 y += yi  
 points.append((x,y))  
 return points

1. Zmiana rozmiaru obrazu wejściowego

Funkcja rozszerzająca obrazy prostokątne do kwadratowych w celu normalizacji obliczeń na współrzędnych. W innych funkcjach, które zostały użyte w symulatorze założone zostało, że praca wykonywana jest na obrazie kwadratowym. Jeżeli więc obraz jest prostokątny to piksele poza oryginalnym obrazem wypełniane są zerami (kolorem czarnym)

def resizeImage(bitmap):  
 # Get the dimensions of the input bitmap (rectangle)  
 a, b = bitmap.shape  
  
 # If already square just return unchanged  
 if a == b:  
 return bitmap  
   
 size = max(a, b)  
  
 # Create a bitmap with the desired dimensions  
 big\_bitmap = np.zeros((size, size), dtype=bitmap.dtype)  
  
 # Compute the offsets to center the image  
 start\_x = (size - a) // 2  
 start\_y = (size - b) // 2  
  
 # Copy the original bitmap into the center of the new bitmap  
 big\_bitmap[start\_x:start\_x + a, start\_y:start\_y + b] = bitmap  
  
 return big\_bitmap

1. Transformata Radona

Funkcja implementująca transformatę Radona. Zastosowany został model stożkowy z symulacją ruchu emitera i detektorów. Poruszają się one po okręgu będącym okręgiem wpisanym w kwadrat (obraz oryginalny)

def radonTransform(bitmap , step, detectors\_span, detectors\_number):  
 global singoram\_steps  
 singoram\_steps = []  
 bitmap = resizeImage(bitmap)  
  
 size = max(bitmap.shape)  
 center = size//2  
 radius = size//2 - 1  
  
 angles = np.deg2rad(np.arange(0,FULL\_SCAN\_ANGLE,step))  
 span\_rad = np.deg2rad(detectors\_span)  
  
 sinogram = np.zeros((len(angles), detectors\_number), dtype=bitmap.dtype)  
   
 for a\_i, angle in enumerate(angles):  
 # Emitter coordinates  
 x = int(center + radius\*np.cos(angle))  
 y = int(center + radius\*np.sin(angle))  
  
 # Calculate detectors placements for each emiiter   
 for d\_i in range(detectors\_number):  
 # Detector coordinates  
 x\_d = int(center + radius\*np.cos(angle + np.pi - span\_rad/2 + (d\_i\*span\_rad)/(detectors\_number-1)))  
 y\_d = int(center + radius\*np.sin(angle + np.pi - span\_rad/2 + (d\_i\*span\_rad)/(detectors\_number-1)))  
  
 # Ray between emitter and detector  
 bresenham\_points = bresenhamAlgorithm(x, y, x\_d, y\_d)  
   
 # Calculate sinogram single point  
 total\_intensity = np.mean([bitmap[x\_b, y\_b] for x\_b, y\_b in bresenham\_points if 0 <= x < size and 0 <= y < size])  
 sinogram[a\_i, d\_i] = total\_intensity

singoram\_steps.append(sinogram.copy())  
   
 return sinogram, bitmap.shape[0]

1. Filtrowanie

Funkcja implementująca filtrowanie z zastosowaniem konwolucji z domyślą maską (jądrem) 21

def filtering(bitmap, kernel\_size = 21):  
 # Ensure that the kernel size is odd  
 if kernel\_size%2==0:  
 kernel\_size += 1  
 kernel = np.zeros(kernel\_size)   
  
 # Calculate middle index (kernel is indexed from -k to k)  
 middle\_idx = kernel\_size//2  
 kernel[middle\_idx] = 1  
  
 # Calculate kernel values for negative k:  
 # h[k] = 0, if k is even  
 # h[k] = (-4/pi^2)/k^2, if k is odd  
 for i in range(1, middle\_idx, 2):  
 kernel[i] = (-4/(np.pi\*\*2)) / ((i-middle\_idx)\*\*2)   
  
 # Mirror first half of kernel into the second half  
 kernel[-middle\_idx:] = kernel[:middle\_idx][::-1]  
  
 # Use convolution  
 for it, i in enumerate(bitmap):  
 bitmap[it] = np.convolve(i, kernel, mode='same')  
 return bitmap

1. Odwrotna transformata Radona

Funkcja implementująca odwrotną transformatę Radona wykonana analogicznie do transformaty Radona i pozwalająca na wstępne odtworzenie obrazu oryginalnego

def inverseRadonTransform(sinogram, resized\_size, detectors\_span):  
 global inverse\_radon\_steps  
 inverse\_radon\_steps = []

step = FULL\_SCAN\_ANGLE/sinogram.shape[0]  
 detectors\_number = sinogram.shape[1]  
 center = resized\_size//2  
 radius = resized\_size//2 - 1  
 angles = np.deg2rad(np.arange(0,FULL\_SCAN\_ANGLE,step))  
 span\_rad = np.deg2rad(detectors\_span)  
  
 result\_image = np.zeros((resized\_size, resized\_size), dtype=sinogram.dtype)  
  
 for a\_i, angle in enumerate(angles):  
 # Emitter coordinates  
 x = int(center + radius\*np.cos(angle))  
 y = int(center + radius\*np.sin(angle))  
  
 for d\_i in range(detectors\_number):  
 # Detector coordinates  
 x\_d = int(center + radius\*np.cos(angle + np.pi - span\_rad/2 + (d\_i\*span\_rad)/(detectors\_number-1)))  
 y\_d = int(center + radius\*np.sin(angle + np.pi - span\_rad/2 + (d\_i\*span\_rad)/(detectors\_number-1)))  
  
 # Ray between emitter and detector  
 bresenham\_points = bresenhamAlgorithm(x, y, x\_d, y\_d)  
 # Calculate sinogram single point  
 for x\_b, y\_b in bresenham\_points:  
 if 0 <= x\_b < resized\_size and 0 <= y\_b < resized\_size:  
 result\_image[x\_b, y\_b] += sinogram[a\_i, d\_i]  
 inverse\_radon\_steps.append(result\_image.copy())  
 return result\_image

1. Przywrócenie oryginalnego rozmiaru obrazu

Funkcja przywracająca oryginalny rozmiar obrazu. Oznacza to, że obrazy oryginalnie prostokątne, który zostały rozszerzone do kwadratowych zostaną ponownie przycięte do oryginalnych rozmiarów

def cropToOriginal(resized\_bitmap, a, b):  
 size = resized\_bitmap.shape[0] # Square size  
  
 # Compute the offsets used during centering  
 start\_x = (size - a) // 2  
 start\_y = (size - b) // 2  
  
 # Crop the image back to the original size  
 cropped\_bitmap = resized\_bitmap[start\_x:start\_x + a, start\_y:start\_y + b]  
  
 return cropped\_bitmap

1. Normalizacja obrazu wynikowego

Funkcja obsługująca usunięcie ujemnych wartości i przeskalowanie go do zakresu [0,1] na podstawie 99.9. percentyla wartości pikseli

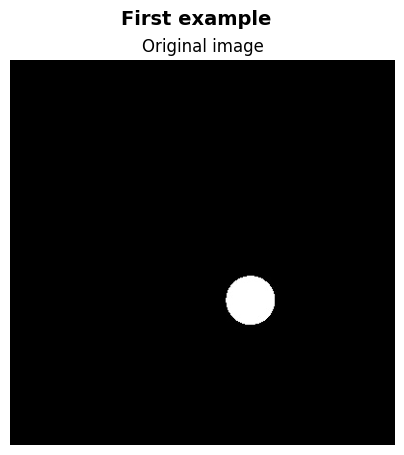
def normalizeImage(bitmap):  
 bitmap = np.maximum(bitmap, 0) # Remove negative values  
 max\_val = np.quantile(bitmap, 0.999) # Use the 99.9th percentile   
 bitmap = np.clip(bitmap / max\_val, 0, 1) if max\_val > 0 else np.zeros\_like(bitmap)  
 return bitmap

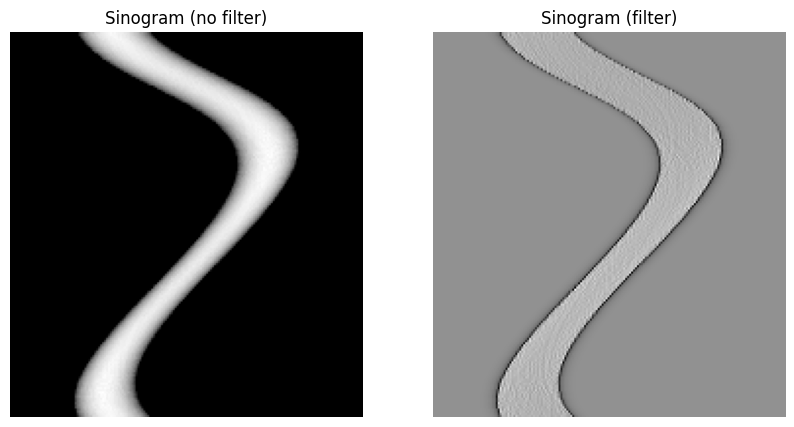
1. Obliczanie pierwiastka błędu średniokwadratowego

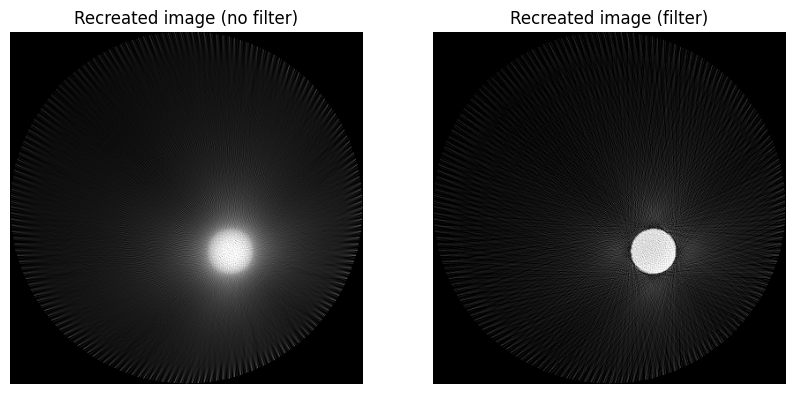
Funkcja obliczające pierwiastek błędu średniokwadratowego między 2 obrazami

def rootMeanSquaredError(img1, img2):  
 if img1.shape != img2.shape:  
 raise ValueError("Obrazy muszą mieć ten sam rozmiar")  
   
 # Obliczanie RMSE  
 rmse = (np.mean((img1 - img2) \*\* 2))\*\*0.5  
 return rmse

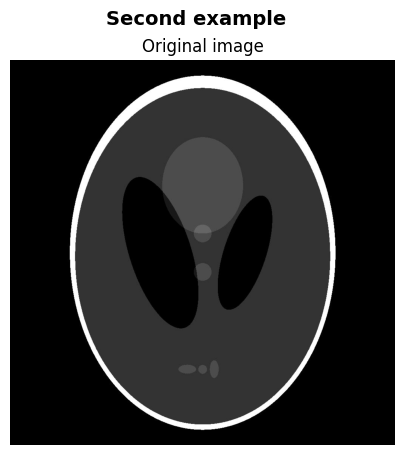
1. **Przykłady działania symulatora**
2. Obraz prosty

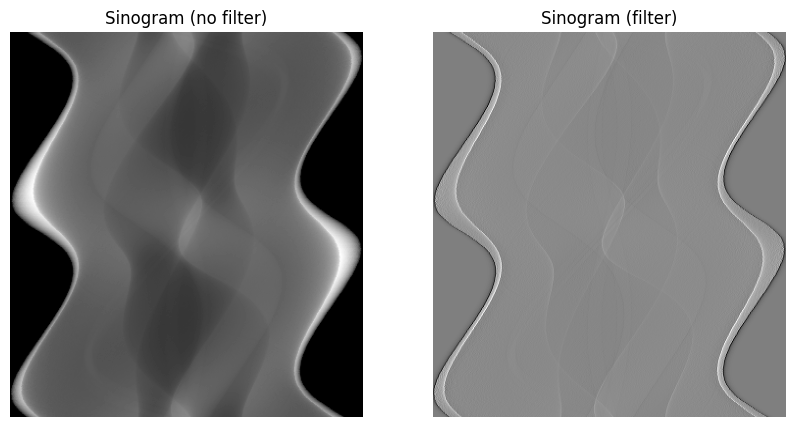


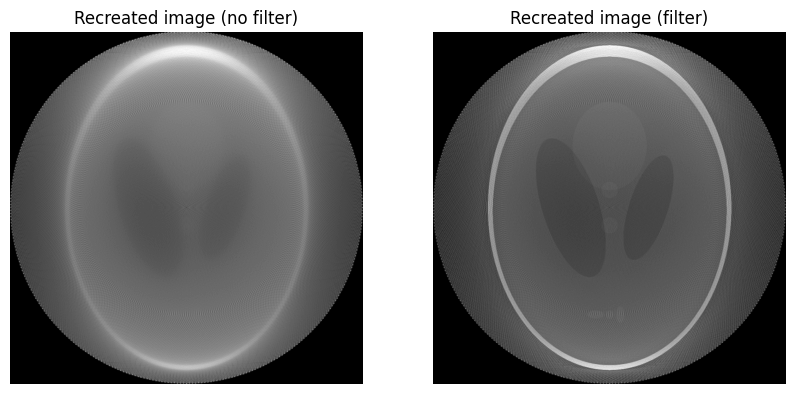




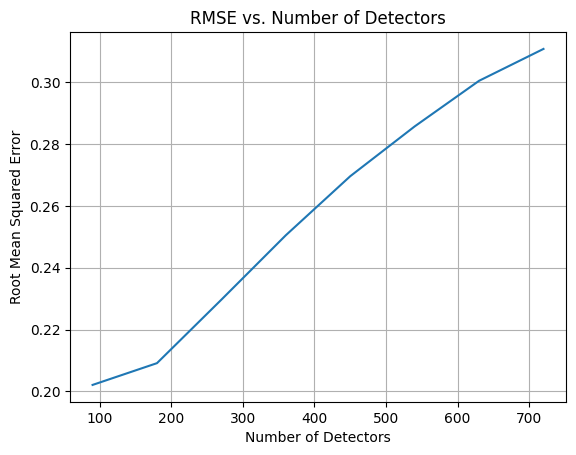
1. Obraz złożony





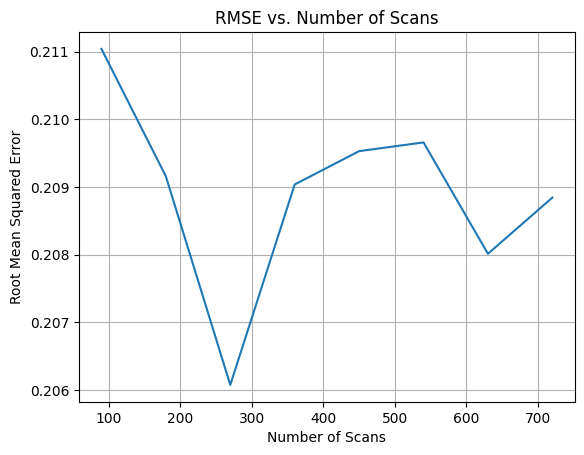


1. **Badanie wpływu parametrów na jakość obrazu wynikowego**
2. Liczba detektorów



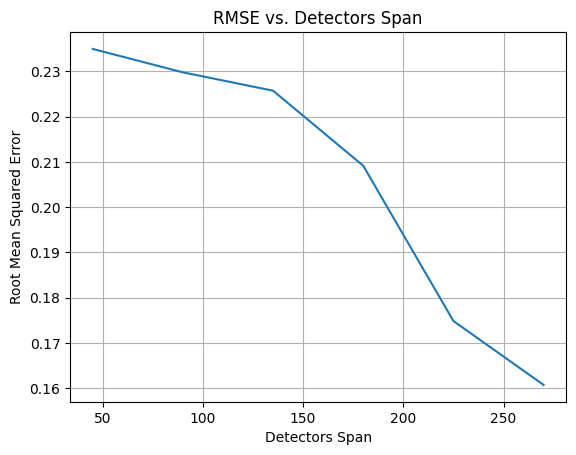
Obliczenia RMSE zgadzają się z subiektywną oceną obrazu. Zwiększanie liczby detektorów powoduje coraz mocniejsze rozświetlenie obrazu i coraz widoczniejsze stają się promienie dookoła oryginalnego obrazu. Przy obecnej implementacji lepiej sprawdzają się mniejsze liczby detektorów (do minimalnej wartości 90 - poniżej tej wartości obrazy są niewyraźne).

1. Liczba skanów



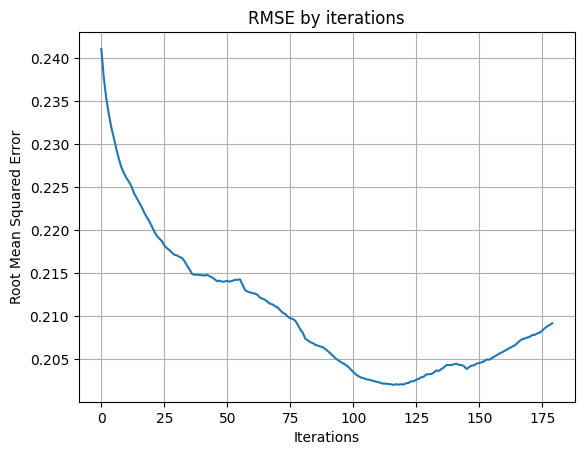
Subiektywnie obrazy są bardzo zbliżone do siebie (poza wyraźnie gorszym obrazem dla minimalnej liczby skanów - 90). Ciężko więc zobaczyć zależność RMSE wynikającą z obliczeń.

1. Rozpiętość wachlarza



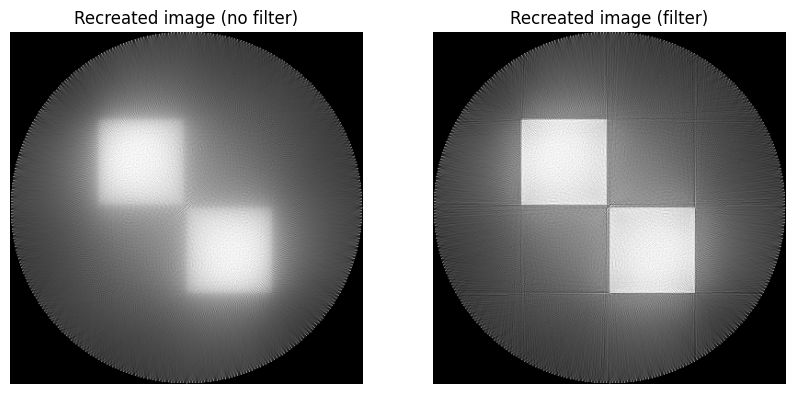
Zwiększanie rozpiętości wachlarza wyraźnie poprawia jakość wynikowego obrazu. Przy wartościach poniżej 180 nie widać nawet wszystkich szczegółów obrazu. Obserwacje te są zgodne z obliczonymi wartościami RMSE.

1. Liczba iteracji

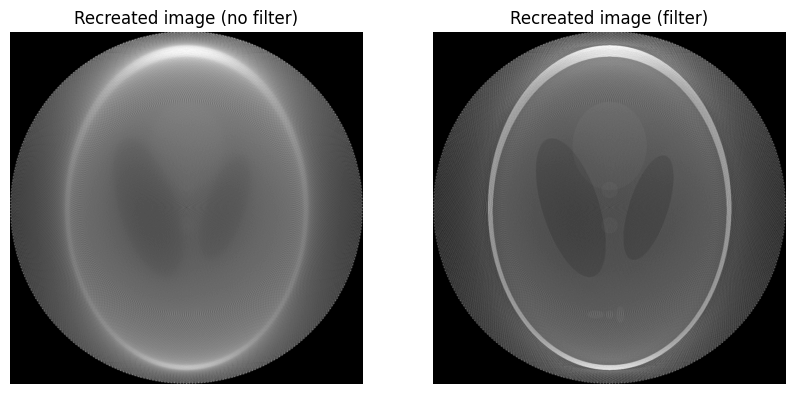


Wykonanie kolejnych iteracji odwrotnej transformaty Radona poprawia jakość wynikowego obrazu. Zgadza się to w większości z wykreśloną zależnością na wykresie. Zaskakujący jest jednak wzrost błędu powyżej iteracji 125. Może to wynikać ze zwiększenia widoczności promieni dookoła analizowanego obrazu wraz ze wzrostem iteracji.

1. Filtracja



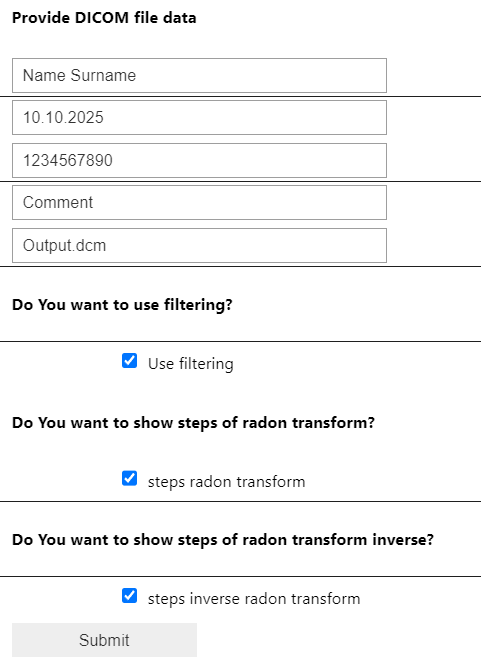
Simple image filter influence  
RMSE without filter: 0.3229776435111086  
RMSE with filter: 0.30205610031935326



Complex image filter influence  
RMSE without filter: 0.2918063436705631  
RMSE with filter: 0.22675962751320947

Filtrowanie obrazu wyraźnie poprawia jakość wynikowego obrazu. Dla prostych obrazów brak filtracji powoduje, że obraz staje się nieostry. Jeżeli nie zastosowano filtracji to dla bardziej skomplikowanych obrazów wejściowych, obraz wyjściowy staje się bardzo niewyraźny a szczegóły całkiem niewidoczne. Obserwacje te są zgodne z obliczonymi wartościami RMSE

1. Interfejs graficzny

W ramach przygotowaniu projektu stworzony został interfejs graficzny do testowania oraz obrazowania działania symulatora tomografu komputerowego. Program pozwala na dostosowanie parametrów symulatora i wybór pliku wejściowego oraz wyjściowego wraz z metadanymi. Program pozwala także na wybranie opcji użycia filtrowania oraz pokazania kolejnych kroków transformaty bądź odwrotnej transformaty Radona w postaci animacji. Poniżej przedstawiony jest interfejs wraz z jego wynikiem działania. 