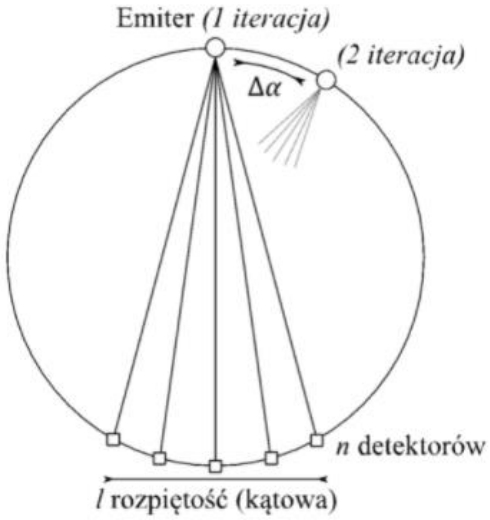
Raport z projektu: Symulator tomografu komputerowego	
Skład grupy: **Mateusz Matkowski 145432** **Damian Mielczarek 145388**	Data wykonania: **28.03.2022**

A) Zastosowany model tomografu: *stożkowy*



B) Zastosowany język programowania oraz dodatkowe biblioteki:

Do napisania programu użyliśmy języka python w środowisku Jupyter Notebook.
Dodatkowe biblioteki:

- Numpy
- Skimage
- Math
- Pydicom
- Datatime
- Time
- Ipywidgets
- Warnings

C) Opis głównych funkcji programu:

a) Pozyskiwanie odczytów dla poszczególnych detektorów:

```
20    #przesunięcie emitera i detektorów (model stożkowy)
21    for i in range(steps):
22       #przesunięcie emitera
23       x = center_x + int(radius_x * np.cos(angle))
24       y = center_y + int(radius_y * np.sin(angle))
25       #wyliczenie "położenia kątowego" pierwszego detektora
26       tmpDetectorAngle = m.radians(180 - l/2) + angle
27       detectorStep = m.radians(l / (detectors - 1))
28       for j in range(detectors):
29          #przesunięcie detektorów
30          xDet = center_x + int(radius_x * np.cos(tmpDetectorAngle))
31          yDet = center_y + int(radius_y * np.sin(tmpDetectorAngle))
32          tmpDetectorAngle += detectorStep
33          #wykorzystanie algorytmu Brenshema
34          if(inverse==False):
35             sinog[i][j]=BresenhamAlgorithm(img, x, y, xDet, yDet, inverse)
36          else:
37             sinog = BresenhamAlgorithm(sinog, x, y, xDet, yDet, inverse, img[i][j])
```

W drugiej pętli ustawiamy współrzędne poszczególnych detektorów i używamy algorytmu Bresenham'a do liniowego przejścia przez kolejne pixele obrazu dyskretnego pomiędzy emiterem a detektorem. Funkcja warunkowa ze zmienną 'inverse' determinuje działanie całej funkcji (transformata Radona lub odwrotna transformata Radona).

b) Filtrowanie sinogramu:

```
1   def filtredSinogram(sinog, kernelSize=21):
2      mask=[]
3      for i in range((int)(kernelSize/2)+1):
4         if(i==0):
5            mask.append(1.0)
6         elif(i%2==0):
7            mask.append(0.0)
8         else:
9            mask.append((-4)/np.pi**2/i**2)
10     mask=mask[::-1]+mask[1:]
11     for i in range(len(sinog)):
12        sinog[i,:]=np.convolve(sinog[i,:],mask, mode='same')
13     return sinog
```

Do filtrowania użyliśmy konwolucji z biblioteki numpy, ustawiając rozmiar maski na 21.

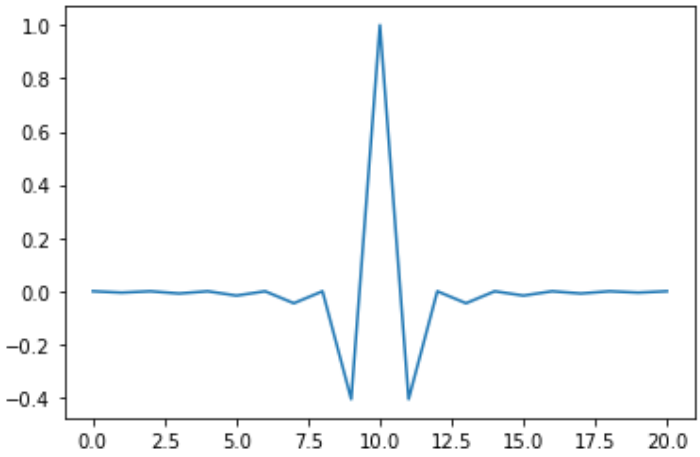
Do wygenerowania maski użyliśmy funkcji:

$$h[0] = 1$$

$$h[k] = 0 \qquad \text{dla } k \text{ parzystych}$$

$$h[k] = \frac{-4/\pi^2}{k^2} \qquad \text{dla } k \text{ nieparzystych}$$

Wykres wartości maski:



c) Ustalanie jasności poszczególnych punktów obrazu wynikowego oraz jego przetwarzanie końcowe:

Korzystamy z uśredniania w algorytmie Bresenham'a:

```
1  def plotLineHigh(img, suma,dx,dy,x,y,sx,sy,y2, inverse, brightness):
2      #usprawnienie obliczeń
3      a = (dx - dy) * 2
4      b = dx * 2
5      D = b - dy
6      #licznik iteracji
7      it=0
8
9      while y != y2:
10         if(inverse==False):
11             #pobranie koloru
12             suma += img[x][y]
13             it+=1
14         else:
15             #wzmocnienie obrazu przy użyciu sinogramu
16             img[x][y] += brightness
17
18         if D >= 0:
19             x += sx
20             y += sy
21             D += a
22         else:
23             D += b
24             y += sy
25
26     if(inverse==False):
27         return suma/it
28     else:
29         return img
```

Normalizacja:

```
1  def normalize(sinog):
2      maksimum = np.max(sinog)
3      minimum = np.min(sinog)
4      if (maksimum-minimum)!=0:
5          sinog = (sinog-minimum)/(maksimum-minimum)
6      return sinog
```

```
52      #normalizacja oraz poprawa wyniku po odwrotnej transformacie Radona
53      sinog= normalize(sinog**2)**0.5
```

Funkcja normalizacji jest ostatnim krokiem odwrotnej transformaty Radona. W naszym przypadku podnosimy wartości wynikowe odwrotnej transformaty do kwadratu, aby wyegzekwować odworowywane kształty z obrazu wejściowego. Pierwiastek po normalizacji rozjaśnia piksele poszukiwanych kształtów.

d) Wyznaczanie wartości miary RMSE na podstawie obrazu źródłowego oraz wynikowego:

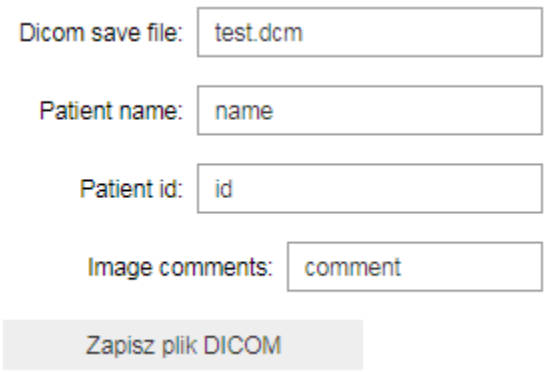
```
1  def RootMeanSquaredError(img, img2):
2      err = np.sum((normalize(img) - img2) ** 2)
3      err /= (img.shape[0] * img.shape[1])
4      return m.sqrt(err)
```

Przy czym: img – obraz wejściowy, img2 – obraz wyjściowy.
Normalizujemy obraz wejściowy, żeby błąd był znormalizowany (prawidłowy).

e) Odczyt i zapis plików DICOM:
Funkcja do wczytywania pliku DICOM, wypisywania wszystkich informacji w nim zawartych oraz wyświetlania znajdującego się tam obrazu:

```
1  def read_dicom(file_name):
2      ds = dcmread(file_name, force=True)
3      print(ds)
4      io.imshow(file_name)
```

Graficzny interfejs do wpisywania informacji o pacjencie i wyboru nazwy pliku:



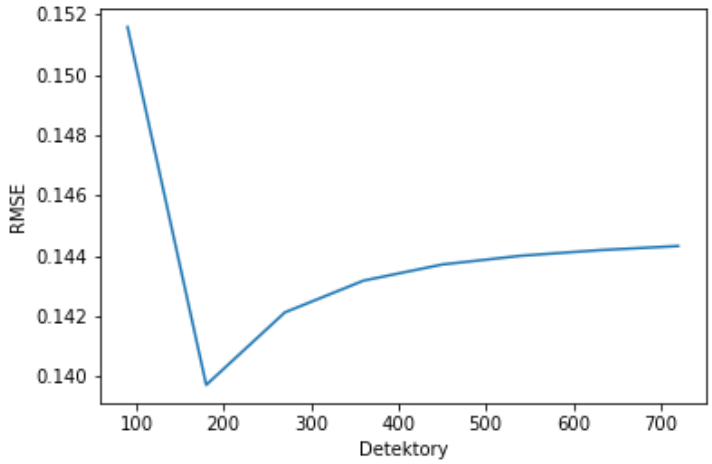
Funkcja do zapisywania obrazu wyjściowego programu i informacji o pacjencie do pliku DICOM:

```
def save_as_dicom(file_name, img, patient_data):
    img_converted = img_as_ubyte(rescale_intensity(img, out_range=(0.0, 1.0)))

    # Populate required values for file meta information
    meta = Dataset()
    meta.MediaStorageSOPClassUID = pydicom._storage_sopclass_uids.CTImageStorage
    meta.MediaStorageSOPInstanceUID = pydicom.uid.generate_uid()
    meta.TransferSyntaxUID = pydicom.uid.ExplicitVRLittleEndian

    ds = FileDataset(file_name, {}, preamble=b"\0" * 128)
    ds.file_meta = meta

    ds.is_little_endian = True
    ds.is_implicit_VR = False

    ds.SOPClassUID = pydicom._storage_sopclass_uids.CTImageStorage
    ds.SOPInstanceUID = meta.MediaStorageSOPInstanceUID

    ds.PatientName = patient_data["PatientName"]
    ds.PatientID = patient_data["PatientID"]
    ds.ImageComments = patient_data["ImageComments"]
    ds.ContentDate = str(datetime.date.today()).replace('-','')
    ds.ContentTime = str(time.time())

    ds.Modality = "CT"
    ds.SeriesInstanceUID = pydicom.uid.generate_uid()
    ds.StudyInstanceUID = pydicom.uid.generate_uid()
    ds.FrameOfReferenceUID = pydicom.uid.generate_uid()

    ds.BitsStored = 8
    ds.BitsAllocated = 8
    ds.SamplesPerPixel = 1
    ds.HighBit = 7

    ds.ImagesInAcquisition = 1
    ds.InstanceNumber = 1

    ds.Rows, ds.Columns = img_converted.shape

    ds.ImageType = r"ORIGINAL\PRIMARY\AXIAL"

    ds.PhotometricInterpretation = "MONOCHROME2"
    ds.PixelRepresentation = 0

    pydicom.dataset.validate_file_meta(ds.file_meta, enforce_standard=True)

    ds.PixelData = img_converted.tobytes()

    ds.save_as(file_name, write_like_original=False)
```

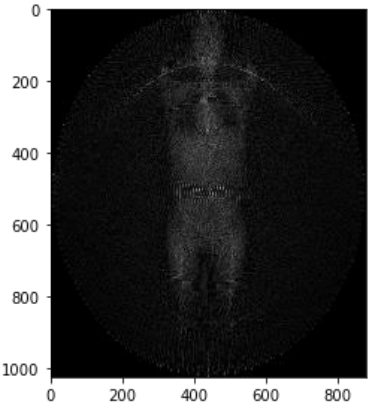
D) Wyniki eksperymentów:

Poniższe wykresy zostały wykonane przy użyciu filtra.

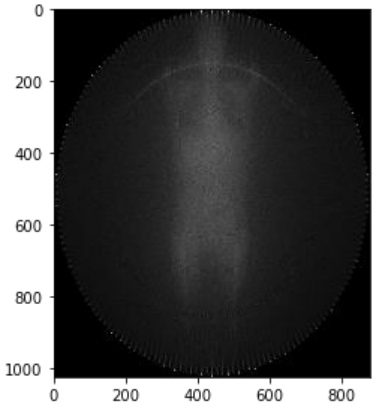
Wykres zależności RMSE od ilości detektorów:



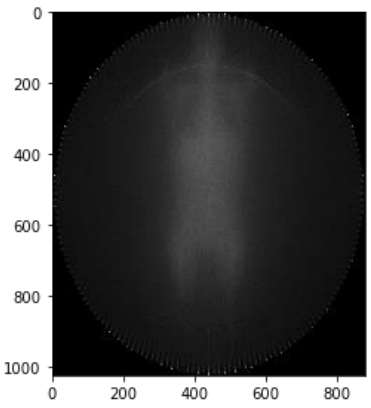
Detektory: 90



Detektory: 450

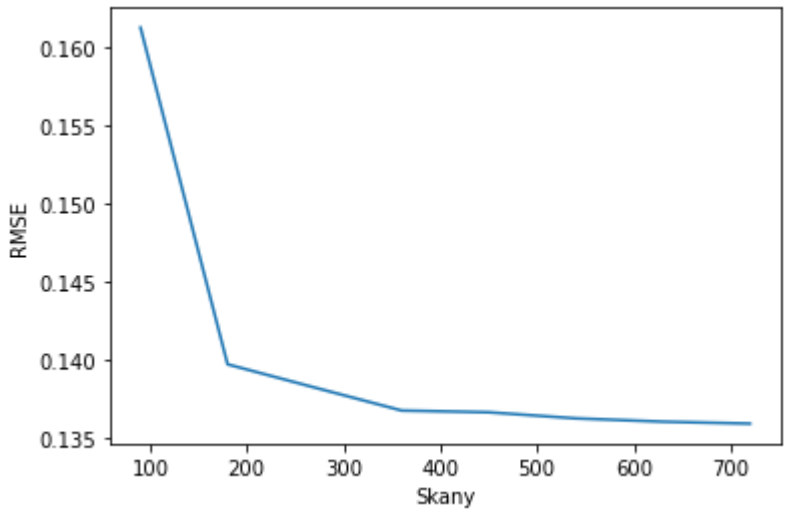


Detektory: 720

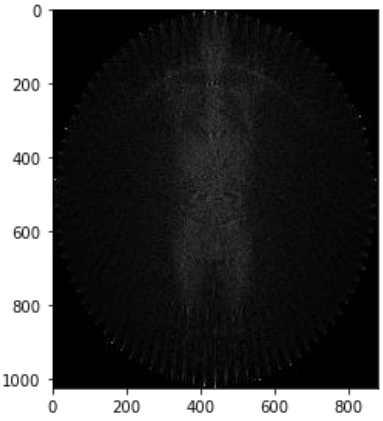


Z początku wartości RMSE gwałtownie maleją, ale od pewnego momentu, przy kolejnych zwiększaniach liczby detektorów, błąd zaczyna wzrastać. Dzieje się tak dlatego, że w pewnym momencie zaczyna pojawiać się iluminacja wokół szkieletu, powstała wskutek nałożenia się jasnych lini na obraz w wyniku odwrotnej transformaty Radona. Mimo pogorszenia się wartości błędu RMSE, jakość obrazu jednak nie ulega degradacji. Ciągłe powiększanie liczby detektorów nie niesie za sobą lepszych rezultatów (jakość obrazu wynikowego nie poprawia się).

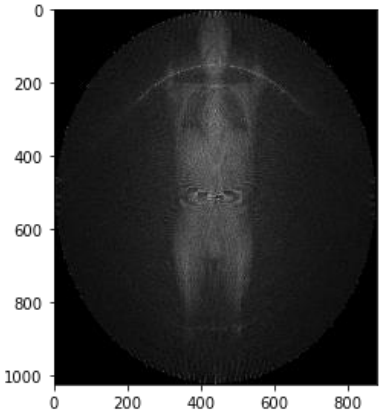
Wykres zależności RMSE od ilości skanów (kroków):



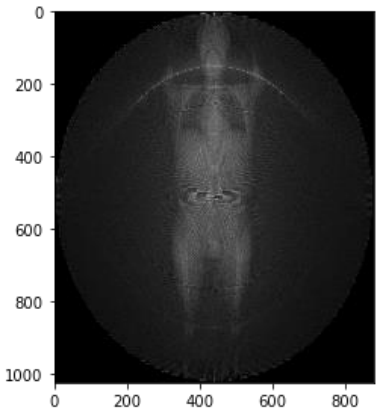
Skany: 90



Skany: 450

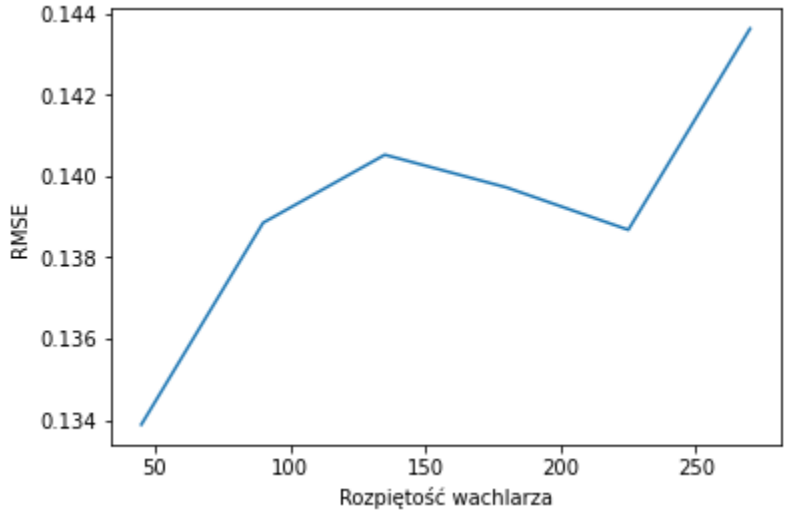


Skany: 720

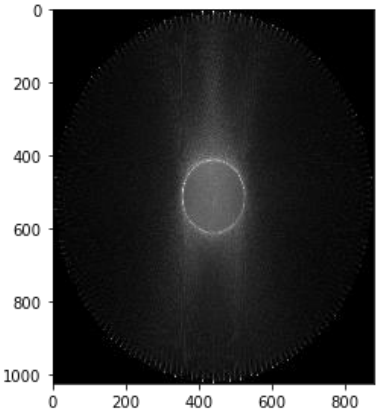


Według wykresu i obrazów wyjściowych wzrost liczby skanów jest jednoznaczny ze zmniejszaniem się błędu RMSE. Jakość obrazów ulega polepszeniu, przy czym coraz bardziej widoczna jest poświata wokół szkieletu.

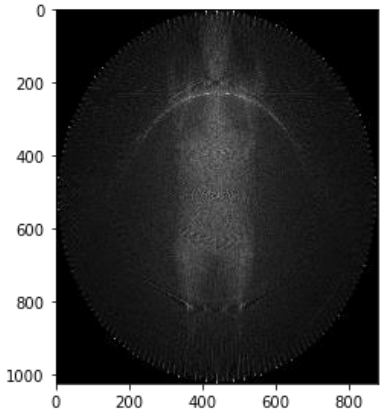
Wykres zależności RMSE od kąta rozpiętości wachlarza:



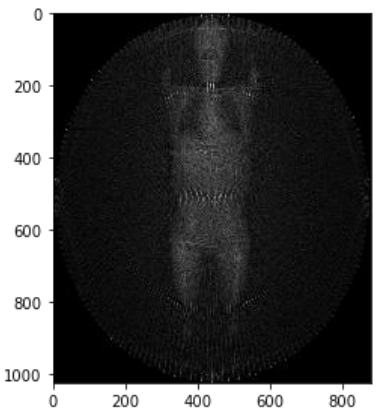
Rozpiętość 45°



Rozpiętość 135 °



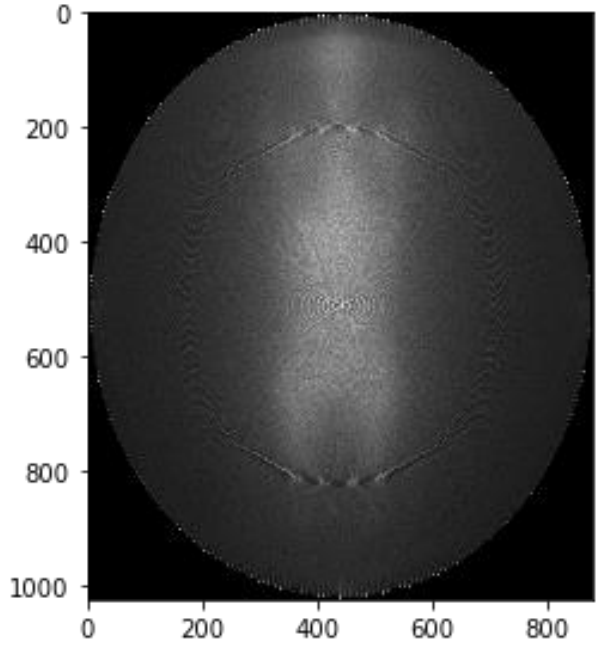
Rozpiętość 270 °



W ogólności błąd RMSE rośnie wraz ze wzrostem kąta rozpiętości wachlarza. Momentalny spadek widoczny na wykresie może być spowodowany zakłóceniami wynikającymi z jasnej elipsy, która przy większych wartościach rozpiętości się rozpływa. Jakość obrazów rośnie wraz ze wzrostem rozpiętości.

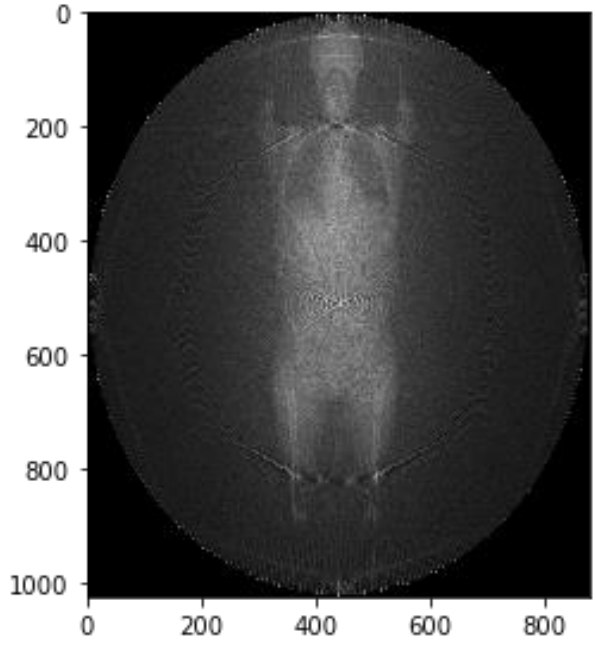
Porównanie wyników symulacji z i bez filtra:

CT_ScoutView.jpg (bez filtra):



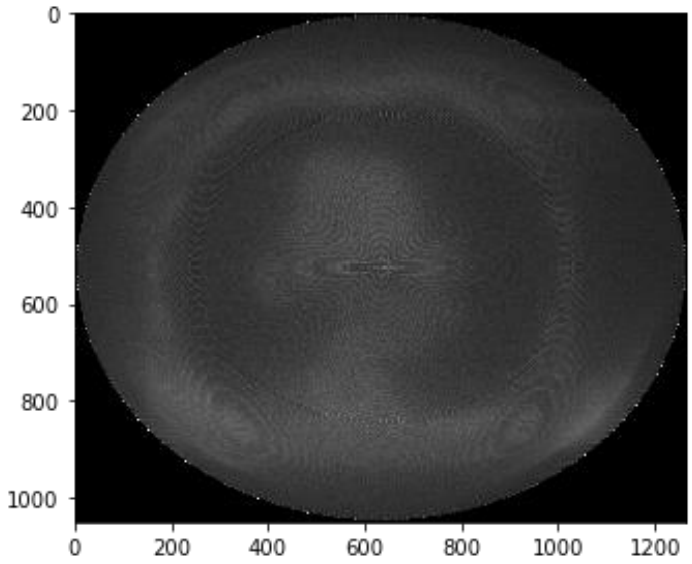
RMSE: 0.17846079036241605

CT_ScoutView.jpg (z filtrem)



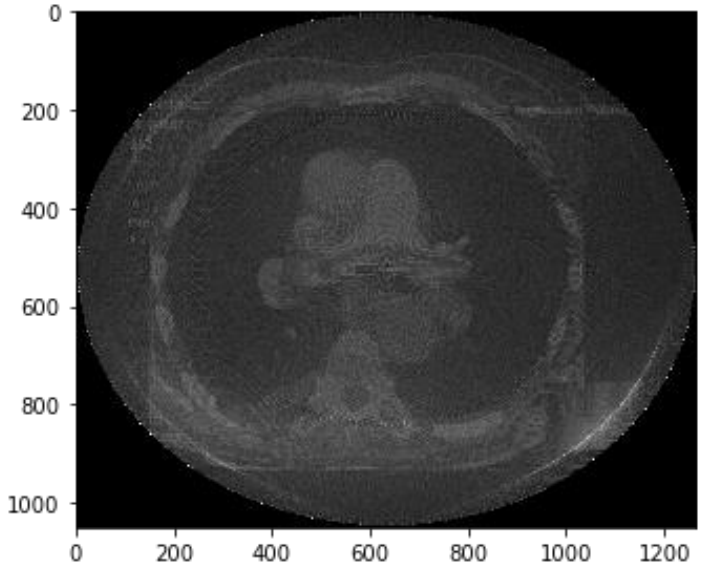
RMSE: 0.15731643521880154

SADDLE_PE-large.jpg (bez filtra)



RMSE: 0.2418653519614973

SADDLE_PE-large.jpg (z filtrem)



RMSE: 0.2314242336063266

Nałożenie filtra pozytywnie wpływa na jakość obrazów wyjściowych i zmniejsza błąd RMSE w każdym przypadku. Przy użyciu filtra można dostrzec więcej szczegółów na obrazach wynikowych, co pomoże przy późniejszej analizie medycznej.