# Sprawozdanie

## Informatyka w medycynie

## Symulator tomografu komputerowego

Łukasz Frątczak 141213

Mateusz Żelazowski 140810

1. **Model tomografu**, który zastosowaliśmy to model równoległy. Do napisania symulatora użyliśmy języka python wraz z takimi bibliotekami jak OpenCV, numpy, tkinter, scipy oraz pydicom.

2. **Opis głównych funkcji programu**
   a) Dla poszczególnych detektorów pozyskujemy odczyty poprzez wyznaczenie współrzędnych emitorów i detektorów a następnie odczytujemy linię za pomocą algorytmu Bresenhama.

```
def calculate_positions(alfa, phi, n, radius):
    positions = []
    for i in range(n):
        deg1 = alfa + phi / 2 - i * phi / (n - 1)
        x1 = radius * np.cos(deg1) + radius - 1
        y1 = radius * np.sin(deg1) + radius - 1
        deg = alfa + np.pi - phi / 2 + i * phi / (n - 1)
        xd = radius * np.cos(deg) + radius - 1
        yd = radius * np.sin(deg) + radius - 1
        positions.append([x1, y1, xd, yd])
    return positions
```

Dla każdej linii obliczany jest jej kolor. Dla zachowania odpowiedniej jasności dla różnych obrazów, program dostosowuje kolory sinogramu na podstawie średniej wartości koloru oryginalnego obrazu

```
divider = (15 * mean_color_on_img ** 0.7)

color = sum(img[line[:, 0], line[:, 1]]) / len(line) / divider
```

   **b) Filtrowanie sinogramu**
   Filtrowanie odbywa się poprzez wykonanie transformaty Fouriera dla każdej linii obrazu. Mnożenie i potęgowanie zostało zastosowane w celu zwiększenia kontrastu, a odejmowanie pomaga przy usunięciu szumu wokół skanowanego obiektu. Następnie wynik mnożony jest przez okno Hamminga. Na koniec wykonujemy odwrotną transformatę Fouriera i zostawiamy tylko rzeczywistą część wyników.

```
for i in range(len(sinogram)):
    sinogram[i] = np.real(scipy.fft.ifft(scipy.fft.fft((2*sinogram[i])**1.25-0.3) * np.hamming(len(sinogram[i]))))
```

c) **Ustalanie jasności poszczególnych punktów obrazu** wynikowego odbywa się poprzez nakładanie na obraz linii na podstawie wcześniej obliczonego koloru na oraz obliczanie dzielnika, który normalizuje wynik dla zachowania odpowiedniej jasności na podstawie średniej wartości koloru sinogramu, liczby detektorów oraz kroku co jaki przesuwane są emitery i detektory.

**Dla wersji bez filtrowania:**

```
ratio = 1 / angle2
divider = n / 40 / np.mean(sinogram) * ratio *(180/l)**2
for angle in angles:
    i = 0
    for line in lines[int(angle * ratio)]:
        color = sinogram[int(angle * ratio)][i]
        i += 1
        img[line[:, 0], line[:, 1]] += color / divider
```

**Dla wersji z filtrowaniem:**

```
ratio = 1 / angle2
divider = n / 750 / np.mean(sinogram)**1.8*ratio/2.5 *(180/l)**2
for i in range(len(sinogram)):
    sinogram[i] = np.real(scipy.fft.ifft(scipy.fft.fft((2*sinogram[i])**1.25-0.3)
                                         * np.hamming(len(sinogram[i]))))

for angle in angles:
    i = 0
    for line in lines[int(angle * ratio)]:
        color = sinogram[int(angle * ratio)][i]
        i += 1
        img[line[:, 0], line[:, 1]] += color/divider
```

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

Na początku obraz jest poddawany modyfikacji poprzez zamianę wartości większych od 1 na 1 oraz mniejszych od 0 na 0, w celu zachowania odpowiedniej skali kolorów. Operacja ta nie wpływa na wizualną zmianę obrazu, gdyż funkcją rysująca obraz robiła to automatycznie. Brak tej funkcji mógłby powodować zakłamanie wartości miary RMSE mimo poprawnego obrazu. Kolory obrazu wejściowego wczytane są w skali 0-255, dlatego punkty z naszego obrazu, które są w skali 0-1 są mnożone razy 255.

```
rev_filter[rev_filter < 0] = 0
rev_filter[rev_filter > 1] = 1
MSE = 0
for i in range(len(rev_filter)):
    for j in range(len(rev_filter[i])):
        MSE += (img[i][j] - rev_filter[i][j]*255)**2
MSE = MSE / (len(rev_filter) * len(rev_filter[0]))
RMSE = MSE**0.5
```

e) **Pliki DICOM** odczytywane są za pomocą biblioteki pydicom, następnie uzupełniamy obraz kolorem czarnym, aby obraz był kwadratowy.

```
img1 = pydicom.dcmread('tomograf-zdjecia/' + file)
s = max(img1.pixel_array.shape)
img2 = np.zeros((s, s), np.uint8)
ax, ay = (s - img1.pixel_array.shape[1]) // 2, (s - img1.pixel_array.shape[0]) // 2
img2[ay:img1.pixel_array.shape[0] + ay, ax:ax + img1.pixel_array.shape[1]] = img1.pixel_array
```

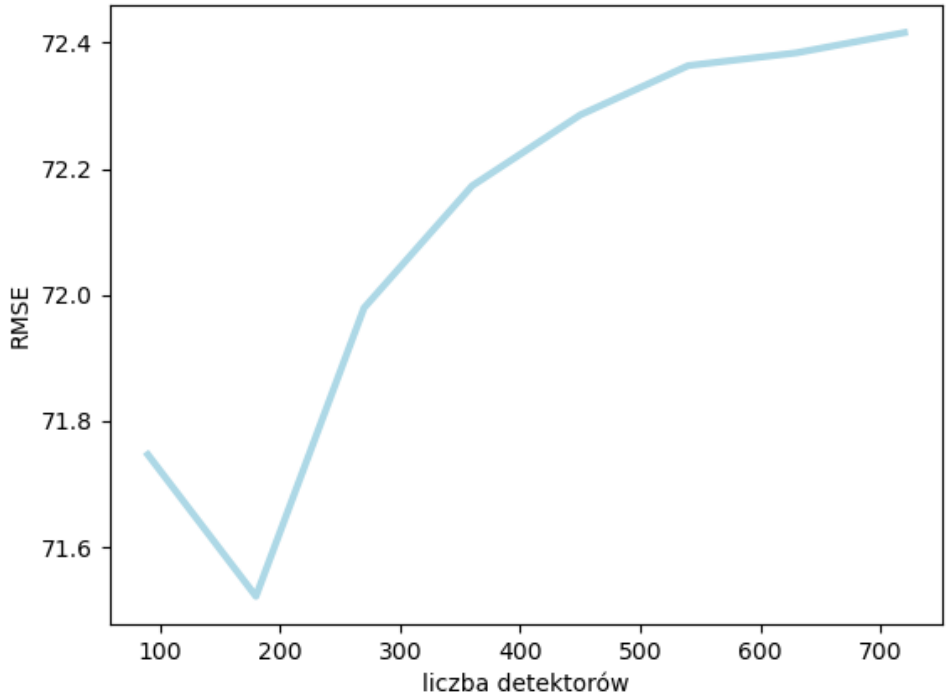
Do zapisu wygenerowanego obrazu również używamy biblioteki pydicom. Przed zapisem nadajemy konieczne atrybuty potrzebne do odczytu pliku DICOM.

```
pydicom.dataset.validate_file_meta(ds.file_meta, enforce_standard=True)
ds.PixelData = img.tobytes()
Path("output_dicom").mkdir(parents=True, exist_ok=True)
ds.save_as('output_dicom/' + full_filename, write_like_original=False)
```

## 3. Wynik eksperymentu

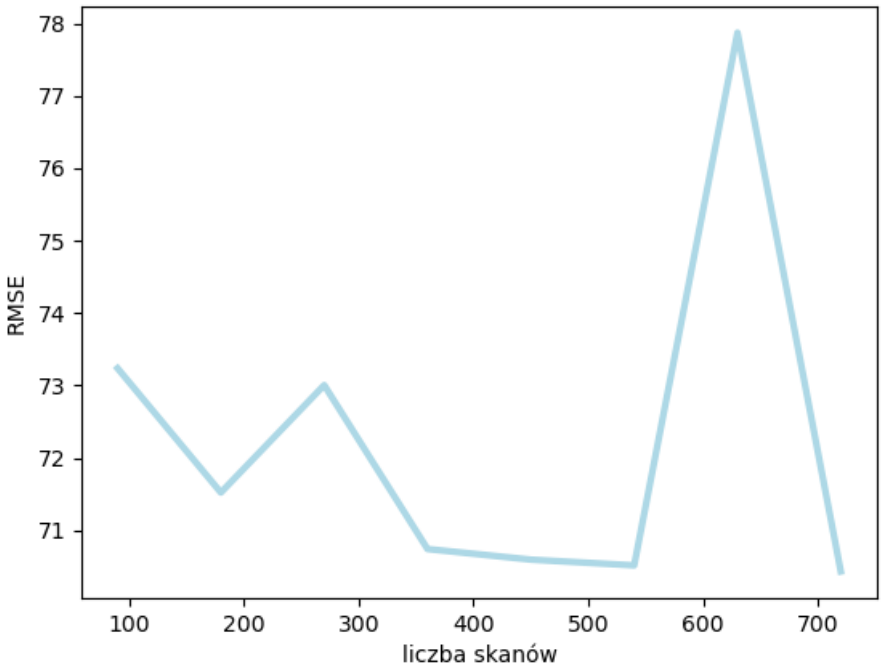
Eksperyment przeprowadziliśmy na przykładowym obrazie SADDLE_PE.jpg

a) Zmienna liczba detektorów



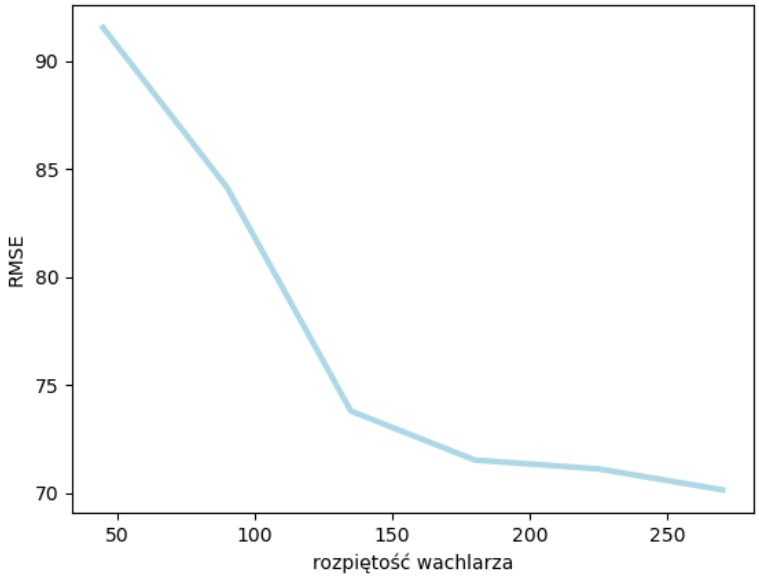

Mniejsza liczba detektorów wg przebiegu zapewnia lepszą jakość obrazu lecz subiektywna ocena czytelności obrazu wskazuje, że im więcej detektorów tym jakość jest o wiele lepsza.

**b) Zmienna liczba skanów**




Większa liczba skanów w dużym stopniu poprawia czytelność obrazu wynikowego co jest pokrywa się z wykresem.

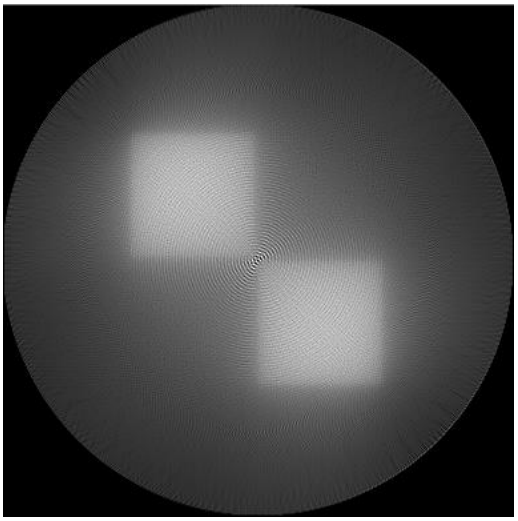
**c) Zmienna rozpiętość wachlarza**



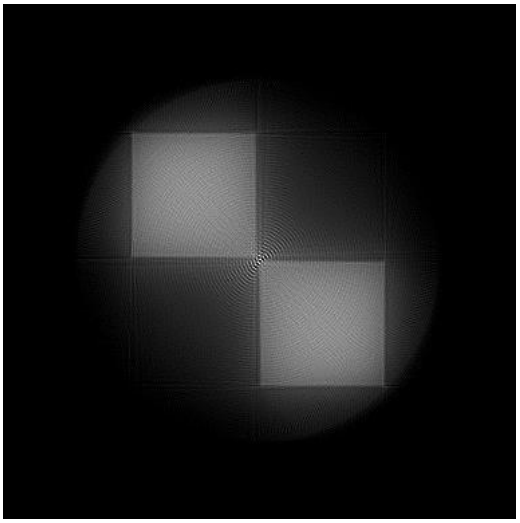

Zgodnie z przebiegiem obraz jest lepszy wraz ze zwiększeniem rozpiętości wachlarza co odpowiada subiektywnej ocenie obrazu.

**d) Porównanie dwóch obrazów przy stałych parametrach**

**- Obraz Kwadraty2.jpg**



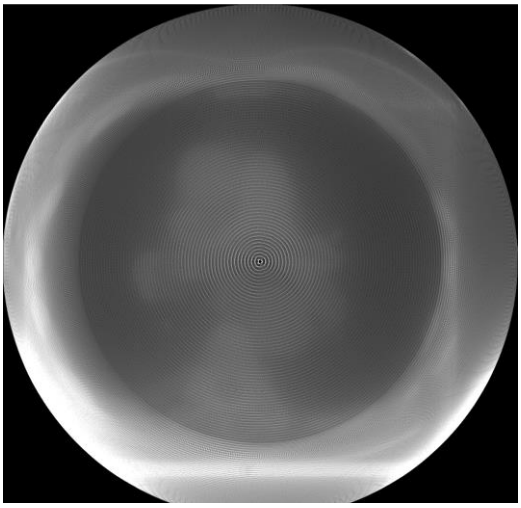
RMSE bez filtra: 69.459



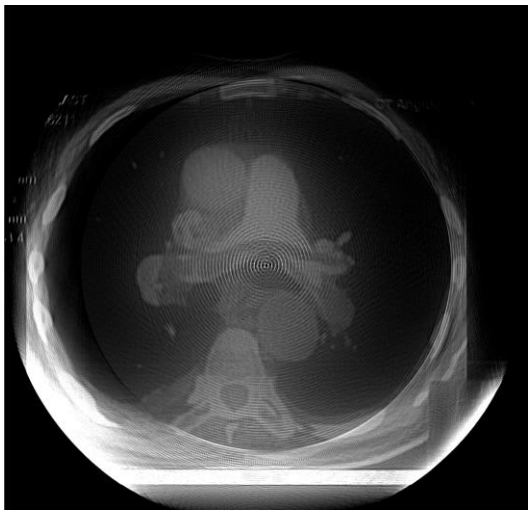
RMSE z filtrem: 50.111

Obraz z filtrem jest znacznie czytelniejszy, krawędzie są ostre, a tło nie jest tak zaszumione jak obraz bez filtrowania, co odzwierciedla również miara RMSE.

**- Obraz SADDLE_PE.jpg**



RMSE bez filtra: 91.038



RMSE z filtrem: 69.128

Zgodnie z miarą RMSE obraz bez filtra jest słabej jakości, kontury są bardzo rozmazane. Obraz z filtrem jest czytelniejszy, posiada widoczne kontury oraz jest większy kontrast między poszczególnymi elementami.