Symulator tomografu komputerowego

Martyna Mirkiewicz 141285
Michał Olszewski 141292

Spis treści

1 Informacje dotyczące implementacji.

1.1 Zastowany model tomografu.

Nasz symulator używa stożkowego modelu tomografu.

1.2 Zastosowany język programowania.

Język programowania: Python

Bibloteki:

- open-cv
- numpy
- matplotlib
- pydicom
- skimage
- ipywidgets
- functools

2 Pozyskiwanie odczytów dla poszczególnych detektorów.

2.1 Pozyskanie uśrednionego koloru.

2.1.1 Opis.

Dla każdego detektora, iterujemy po pikselach, które znajdują się na linii wygenerowanej przez algorytm Bresenhama. Z każdego piksela pobieramy jego kolor. Następnie uzyskujemy średni kolor podanych pikseli.

2.1.2 Kod.

```
    def get_pixels_color(self, emitter_position, detectors_position):
        colors = []
        for detector_position in detectors_position:
            color = 0.0
            count_pixels = 0
            for x, y in self.bresenham_line(emitter_position, detector_position):
                if self.point_in_picture(x, y):
                    color += self.img[x][y]
                    count_pixels += 1
            if count_pixels:
                tmp = color / count_pixels
                colors.append(tmp/255)
            else:
                colors.append(np.float64(0))
        return colors
```

2.2 Algorytm Bresenhama.

2.2.1 Opis.

Algorytm Bresenhama służy do wyznaczenia punktów, które znajdują się między dwoma wybranymi punktami.

2.2.2 Kod.

```
    def bresenham_line(self, start, end):
        start_x, start_y = int(start[0]), int(start[1])
        end_x, end_y = int(end[0]), int(end[1])

        kx = 1 if start_x <= end_x else -1
        ky = 1 if start_y <= end_y else -1

        dx = abs(start_x - end_x)
        dy = abs(start_y - end_y)

        yield start_x, start_y

        if dx < dy:
            e = dy / 2
            for i in range(int(dy)):
                start_y += ky
                e = e - dx
                if e < 0:
                    start_x += kx
                    e = e + dy
                yield start_x, start_y
        else:
            e = dx / 2
            for i in range(int(dx)):
                start_x += kx
                e = e - dy
                if e < 0:
                    start_y += ky
                    e += dx
                yield start_x, start_y
```

3 Filtrowanie.

3.1 Opis

Do uzyskania lepszego obrazu, użyliśmy filtr, który niweluje skutki nierównej gęstości próbkowania. Filtrowanie wykonaliśmy przez splot sinogramu i stworzonej przez nas maski o rozmiarze 100.

3.2 Kod

```
    def filtruj(self, row_to_filter):
        if self.kernel is None:
            self.kernel = self.create_kernel()

        return np.convolve(row_to_filter, self.kernel, mode = 'same')

    def create_kernel(self, size=100):
        kernel = []
        for i in range(-size//2, size//2):
            if i == 0:
                kernel.append(1)
            elif i \% 2 ==0:
                kernel.append(0)
            else:
                kernel.append((-4/(np.pi**2))/(i**2))

        return kernel
```

4 Przetwarzanie i normalizacja obrazu.

4.1 Opis.

Normalizacja ma na celu przeniesienie wartości od 0 do 1. Ponadto zakres kolorów wynikowego obrazu zostaję zwiększony poprzez skalowanie, gdzie minimalny kolor zawsze wynosi 0, a maksymalny 1.

4.2 Kod.

```
    def normalizeArray(self, arr, x, y):
        normalizeArr = np.zeros((x, y))
        rowMax = 0
        rowMin = 999999
        for row in arr:
            if max(row) > rowMax:
                rowMax = max(row)
            if min(row) < rowMin:
                rowMin = min(row)
        diff = rowMax - rowMin
        for i, row in enumerate(arr):
            for j, x in enumerate(row):
                x -= rowMin
                x /= diff if diff > 0 else 0
                normalizeArr[i][j] = x
        return normalizeArr
```

5 Wyznaczanie wartości miary RMSE.

5.1 Kod.

```
def calculateRMSE(input_img, output_img):
    if np.amax(output_img) > 1:
        output_img = output_img / 255
    if np.amax(input_img) > 1:
        input_img = np.array(input_img) / 255
    print(np.amin(input_img), np.amax(input_img))
    print(np.amin(output_img), np.amax(output_img))
    rmse = 0.0
    n = input_img.shape[0] * input_img.shape[1]
    rmse = np.sum(np.concatenate((input_img - output_img)**2))
    return np.sqrt(rmse / n)
```

6 Obsługa plików DICOM

6.1 Wczytywanie.

6.1.1 Kod.

```
  def load_photo(self, img_name):
        if img_name.endswith(".dcm"):
            self.ds = pydicom.dcmread(img_name)
            self.img = self.ds.pixel_array
        else:
            self.img = cv2.imread(img_name, 0)
        self.add_padding()
```

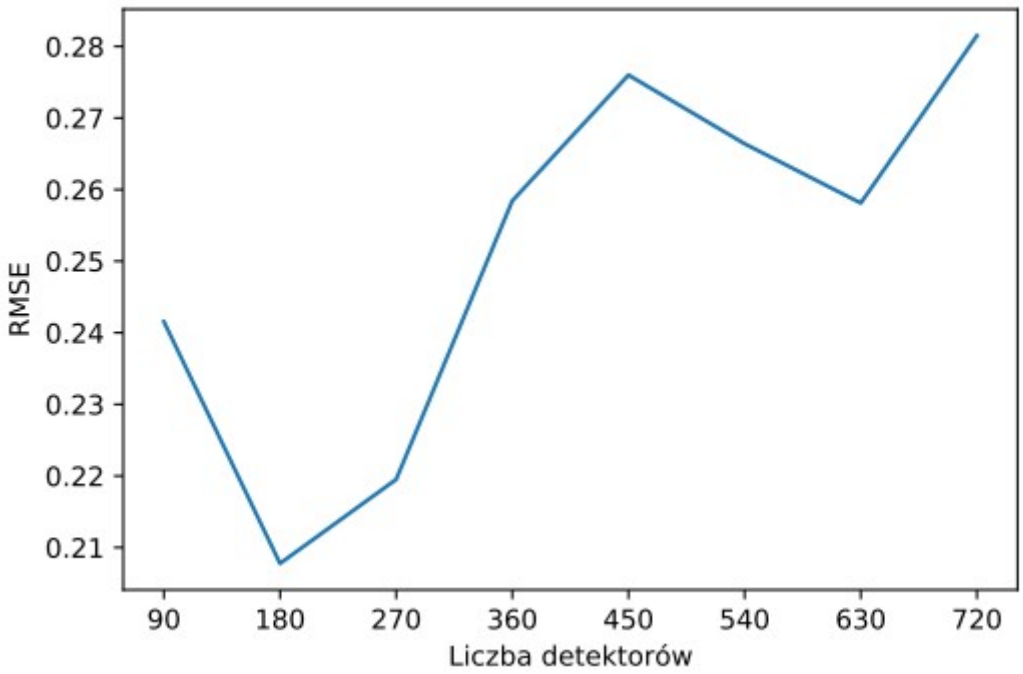
6.2 Zapisywanie.

6.2.1 Kod.

```
def write_as_dcm(img, filename, firstName, lastName, patientID, sex, birthDate, studyDate, comment):
    img = img_as_int(img)
    filect = get_testdata_file('CT_small.dcm')
    ds = pydicom.dcmread(filect)

    ds.PatientName = '{}^{}'.format(lastName, firstName)
    ds.PatientID = patientID
    ds.PatientSex = sex
    ds.PatientBirthDate = birthDate
    ds.StudyDate = studyDate

    ds.Rows, ds.Columns = img.shape[0], img.shape[1]
    ds.PixelRepresentation = 0
    ds.BitsAllocated = 16
    ds.BitsStored = 16
    ds.HighBit = 15
    ds.PixelData = img.tobytes()
    ds.ImageComments = comment

    ds.save_as(filename)
```

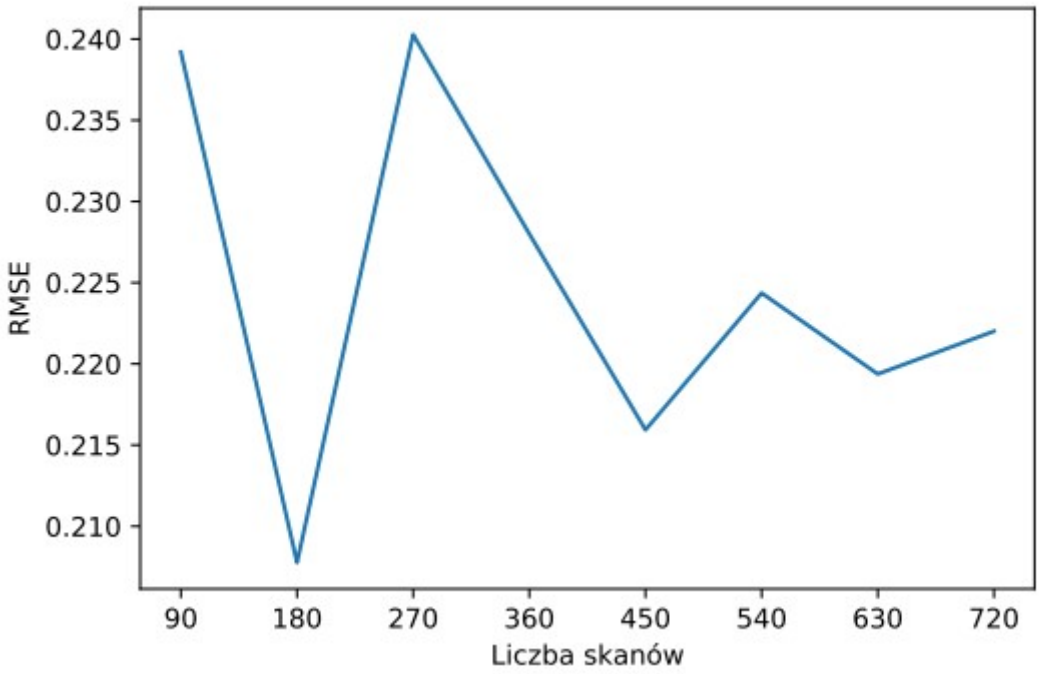
7 Wpływ zmiany parametrów na wynik końcowy.

7.1 Liczba detektorów.



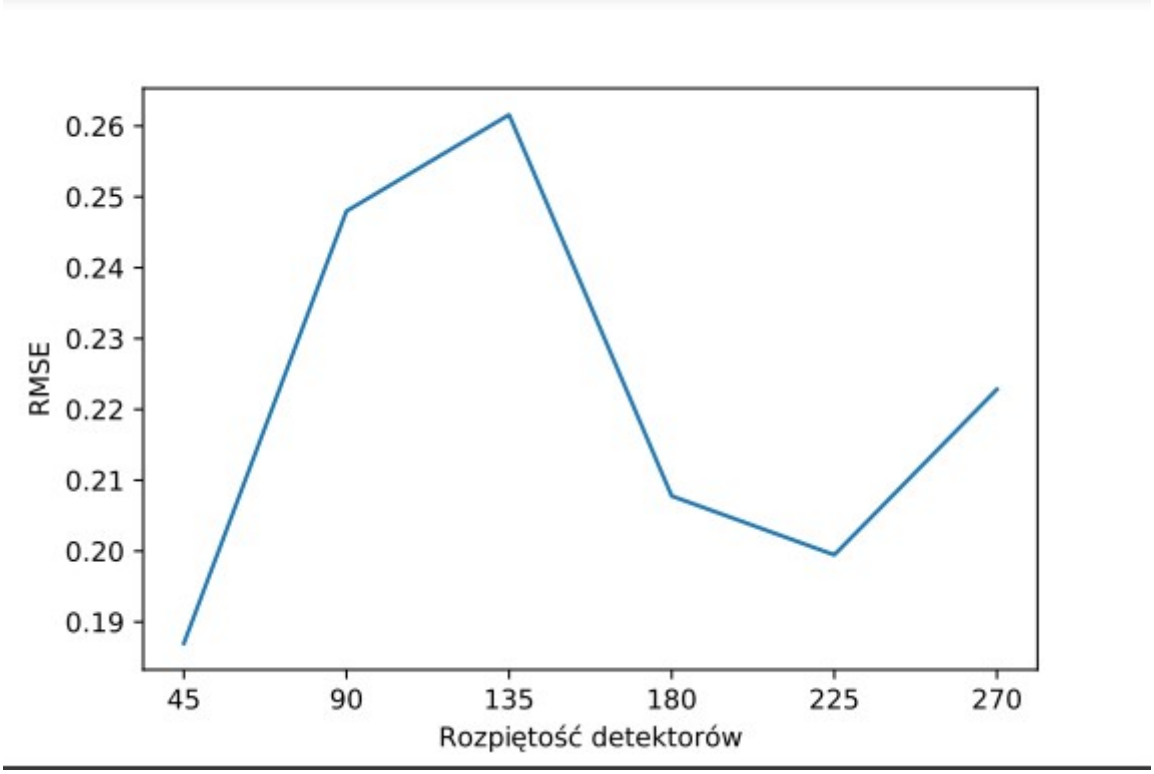
Można zauważyć, że najlepszy rezultat uzyskujemy, gdy liczba detektorów jest równa 180. Po przekroczeniu tej liczby wykres ma tendencję rosnącą.
W rzeczywistości, możemy jednak zauważyć, że im większa liczba detektorów, tym rezultat jest lepszy.

7.2 Liczba skanów.



Tak samo jak w poprzednim przypadku, najlepszy rezultat uzyskujemy dla liczby skanów równej 180. Możemy zauważyć, że wykres na przemian wzrasta i opada oraz, że dąży do wypłaszczenia wraz ze wzrostem liczby skanów.
Po obrazach możemy wywnioskować, że zmiana parametru, nie wpływa w dużym stopniu na uzyskany rezultat.

7.3 Rozpiętość detektorów.

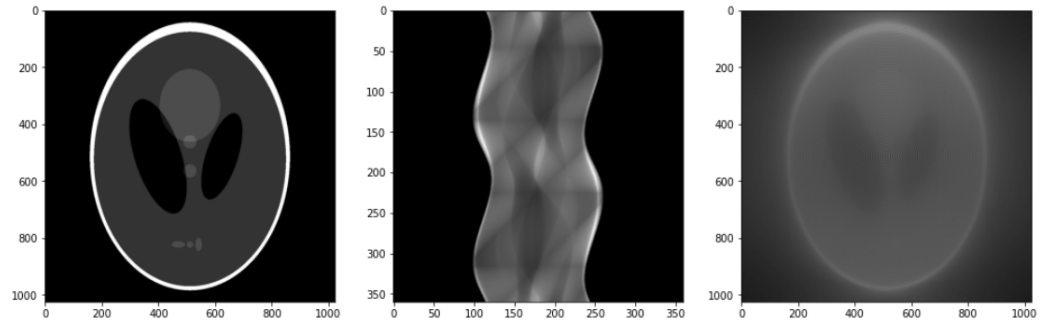


Z uzyskanego wykresu, ciężko wyciągnąć wnioski, gdyż nie ma konkretnej tendencji.
Po uzyskanych obrazach, możemy jednak wywnioskować, że im większy kat, tym lepiej odwzorowane zdjęcie dostaniemy.

8 Wpływ filtru na uzyskane wyniki.

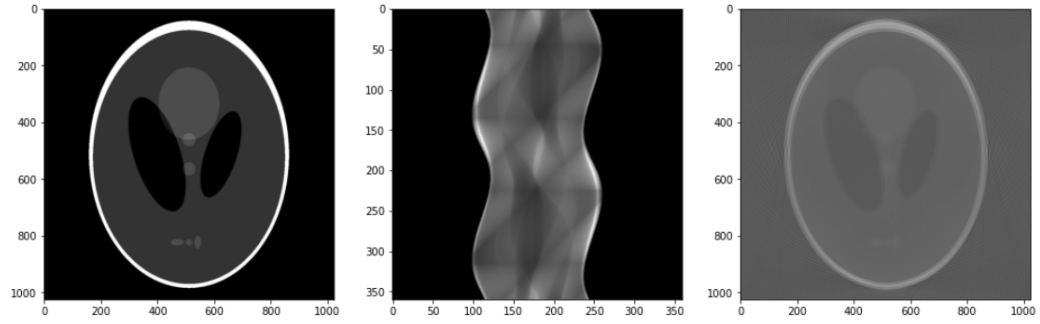
8.1 Obraz nr 1.

Rysunek 1: Obraz bez filtru (RMSE = 0.239)



```
0.23908853485350792
```

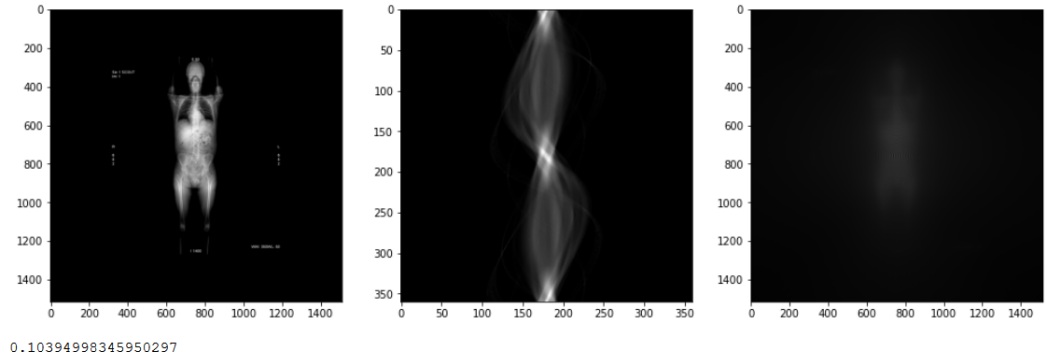
Rysunek 2: Obraz z filtrem (RMSE = 0.312)



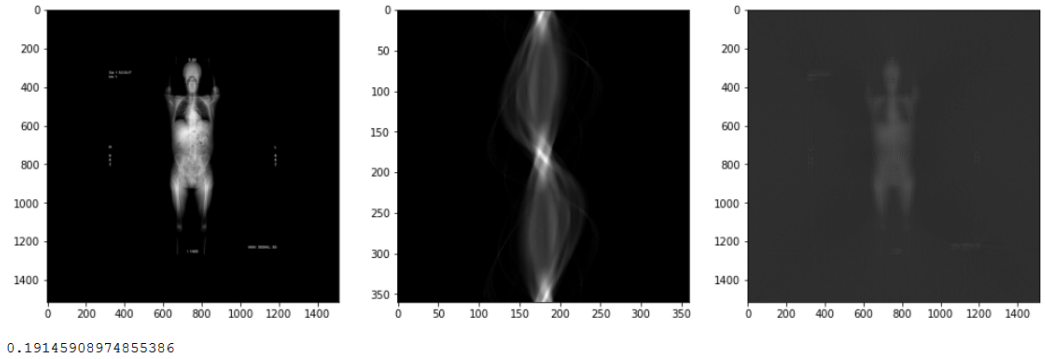
```
0.3118691757118549
```

8.2 Obraz nr 2.

Rysunek 3: Obraz bez filtru (RMSE = 0.104)



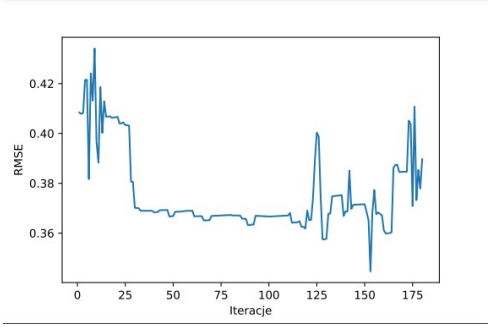
Rysunek 4: Obraz z filtrem (RMSE = 0.191)



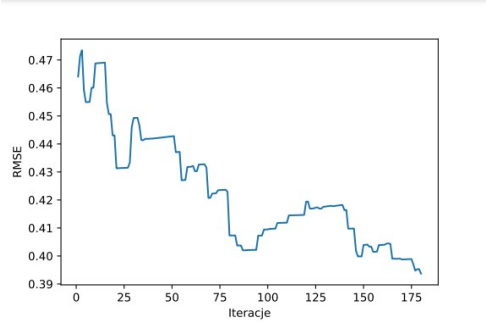
8.3 Wnioski

Możemy zauważyć, że z użyciem filtru obraz zachowuje więcej szczegółów, mimo tego że RMSE daje lepszy rezultat dla zdjęcia bez filtru. Dzieje się tak dla tego, gdyż nasze przefiltorwane zdjecie jest bardziej wyszarzone, co powoduje, że różnice między pikselami są większe. W tym przypadku miara RMSE nie jest dobrym wskaźnikiem do określania jakości rezultatu.

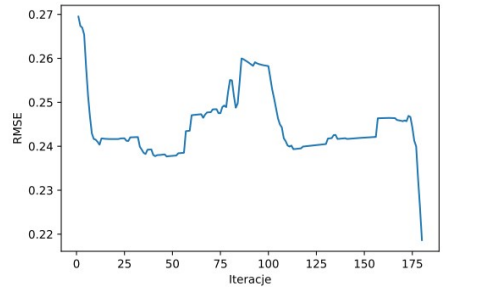
9 Wpływ liczby odbytych iteracji na wartość RMSE.



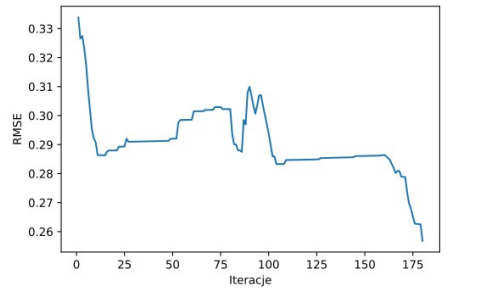
(a) Wykres nr 1.



(b) Wykres nr 2.



(c) Wykres nr 3.



(d) Wykres nr 4.

9.1 Wnioski.

Możemy zauważyć, że wraz ze wzrostem liczby iteracji, dostajemy coraz to dokładniejszy obraz.