Sprawozdanie: Tomograf

Eryk Szpotański nr indeksu 136811

1 Skład grupy

Eryk Szpotański nr indeksu 136811

2 Zastosowany model tomografu

stożkowy addytywny

3 Zastosowany język programowania

transformata + algorytm Bresenhama: C (podstawowe biblioteki: stdio.h, math.h, stdlib.h, string.h, omp.h)

GUI: Python + Streamlit (bilbioteki: streamlit, ctypes, math, numpy, skimage, os, copy)

4 Opis głównych funkcji programu

pozyskiwanie odczytów:

```
for (unsigned int i = 0; i < width; ++i) {
        alpha = i * step;
        E = point_on_circle(C, alpha);

        beta = alpha - spread / 2 + M_PI;
        for (int j = 0; j < decod; ++j) {
                beta += step_beta;
                D = point_on_circle(C, beta);
                if (E.x <= D.x)
                        result[i][j] = line_bres(E.x, E.y, D.x, D.y);
                else
                        result[i][j] = line_bres(D.x, D.y, E.x, E.y);
        }
}
```

Gdzie alpha i beta to kąty (w radianach) odpowiadające pozycji emitera i aktualnego dekodera na wcześniej obliczonym okręgu C (opisanego na obrazie). 'step' to podana zmiana kąta (również w radianach) odpowiadająca jednemu krokowi, spread to podana rozpiętość emiterów, a decod to podana ilość dekoderów. Width to ilość kroków (inaczej szerokość sinogramu), obliczane tutaj E oraz D to pozycje odpowiednio emitera oraz aktualnego dekodera, a wywoływana funkcja line_bres zwraca sumę poszczególnych piskeli wybranych przez algorytm Bresenhama. Podany kod odpowiada fragmentowi z pliku C/transform.c linii 464.

filtrowanie:

```
const float rev_pi = -(float) (M_2_PI * M_2_PI);
float filtered(unsigned int row, int col) {
        float sum = 0.0f;
        for (int i = -11; i <= 11; i += 2) {
                if (i + col >= 0 && i + col < result_b)
                        sum += (rev_pi / (float) (i * i))
                                * result[row][col + i];
        }
        sum += result[row][col];
        return sum;
}
```

Zmienna globalna result zawiera sinogram. Podany kod odpowiada fragmentowi z pliku C/transform.c linia 602.
Rozmiar maski wynosi 23.

przetwarzanie końcowe:

```
res = gaussian(res, sigma=1)
if fil:
        ma = np.amin(res)
        print(ma)
        res = res - ma
        ma = np.amax(res)
        print(ma)
        res = res / ma
else:
        ma = np.amax(res)
        print(ma)
        res = res / ma
```

'gaussian' jest funkcją z skimage, a fil przybiera wartość True, jeśli obraz jest filtrowany, a res to sam otrzymany obraz. Kod ten odpowiada fragmentowi z pliku Python/main.py linii 86.

obliczanie RMSE:

```
math.sqrt(((self.cut_all(image) - rev) ** 2).mean())
```

Gdzie image jest pierwotnym obrazem, przekazywanym do funkcji 'cut_all' zwracającej ten sam obraz uprzednio upewniając się, że posiada jedynie wartości z przedziału [0,1]. Zmienna rev wskazuje na uzyskany obraz z odwrotnej transformaty Radona. Powyższy kod odpowiada linii 198 z pliku Python/main.py.

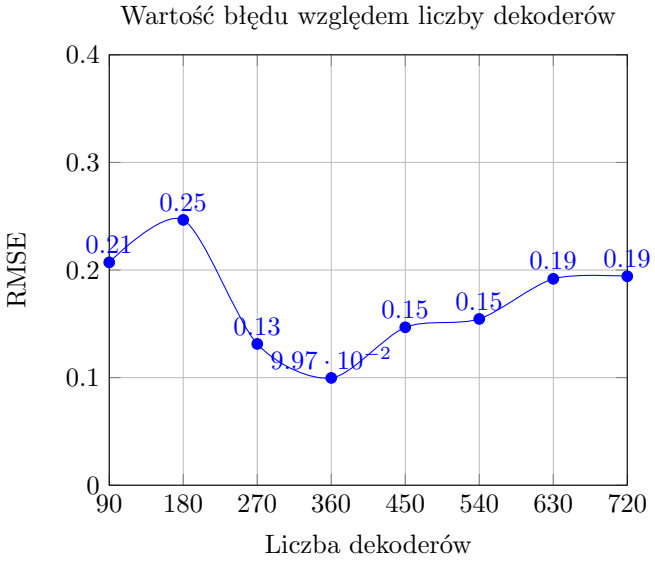
5 Wyniki eksperymentu

Skrypt generujący wyniki znajduje się w Python/tests.py oraz dokładne wyniki znajdują się w folderze Python/wynik.

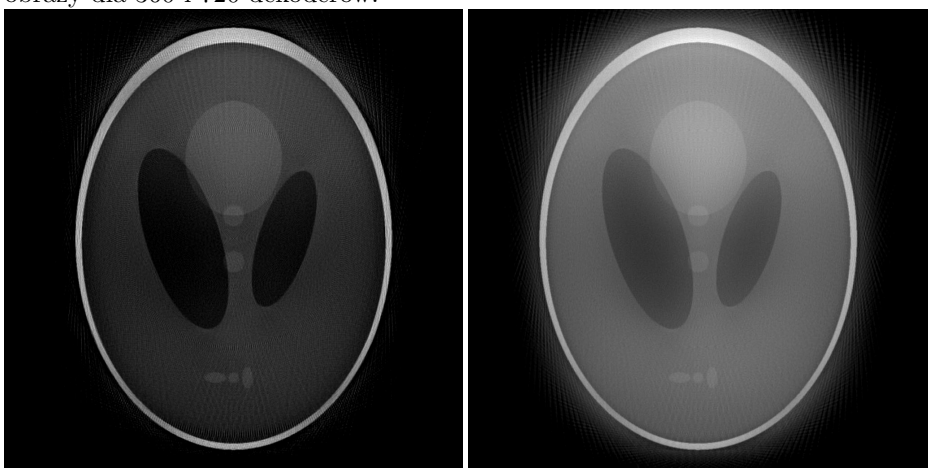
5.1 Część pierwsza

Eksperyment został przeprowadzony dla następującego obrazu (filtrowanie zawsze było włączone):

5.1.1 Zmienna liczba dekoderów

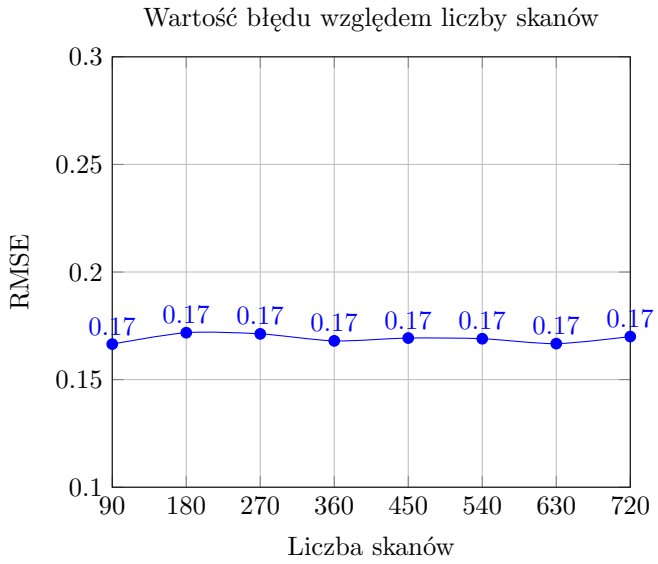


Wykres przebiega dość zaskakująco, gdyż spodziewałem się zobaczyć coś bardziej przypominającego funkcję nierosnącą. Lecz gdy spojrzymy na otrzymane obrazy dla 360 i 720 dekoderów:

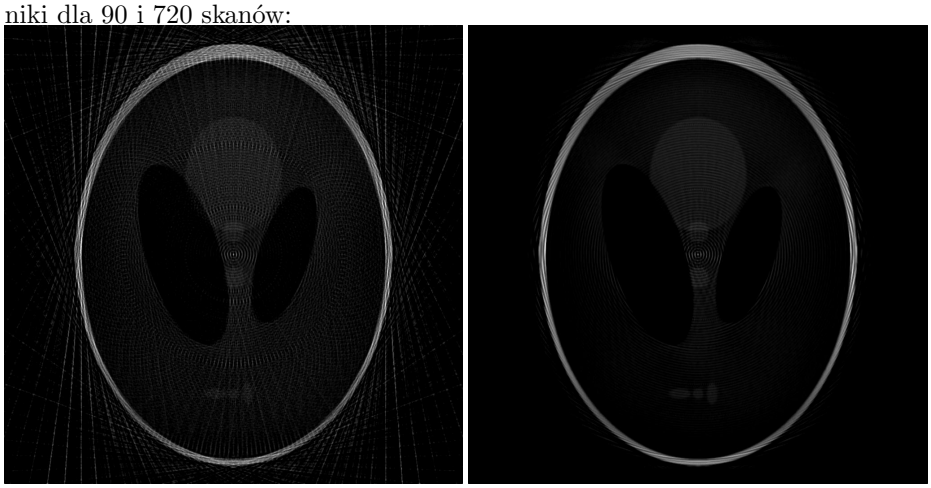


można zaobserwować, że mimo bardziej widocznych artefaktów, przy mniejszej ilości dekoderów, obraz ten jest bardziej zbliżony kolorystycznie do oryginału, co przekłada się na mniejsze RMSE, ale nie koniecznie na subiektywną jakość obrazu (która zwiększa się wraz ze wzrostem liczby dekoderów). Wynika z tego, że przetwarzanie końcowe tych obrazów jest dalekie od ideału i powinienem je uzależnić od jasności obrazu lub sumy wiersza sinogramu.

5.1.2 Zmienna liczba skanów

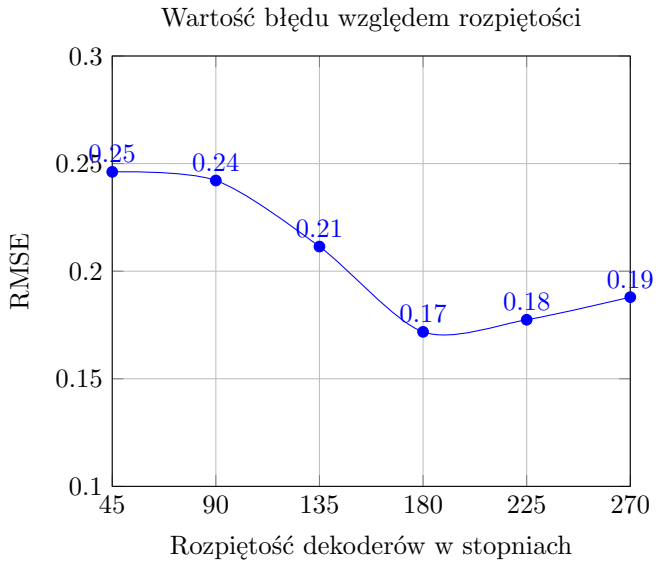


Obserwując przebieg wykresu można wysunąć wniosek, że liczba kroków nie ma znaczącego wpływu na wielkość obliczanego błędu, lecz porównując skrajne wyniki dla 90 i 720 skanów:

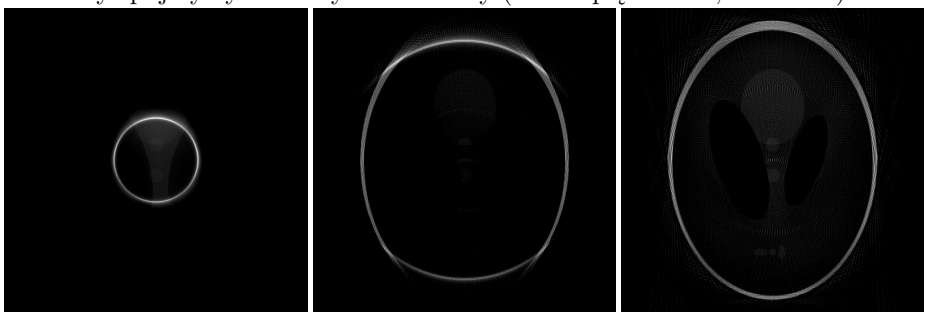


można zaobserwować poprawę jakości obrazu.

5.1.3 Zmiana rozpiętości



Na tym wykresie można zaobserwować spory spadek RMSE koło rozpiętości 135. Gdy spojrzymy na otrzymane obrazy (dla rozpiętości 45, 135 i 270):

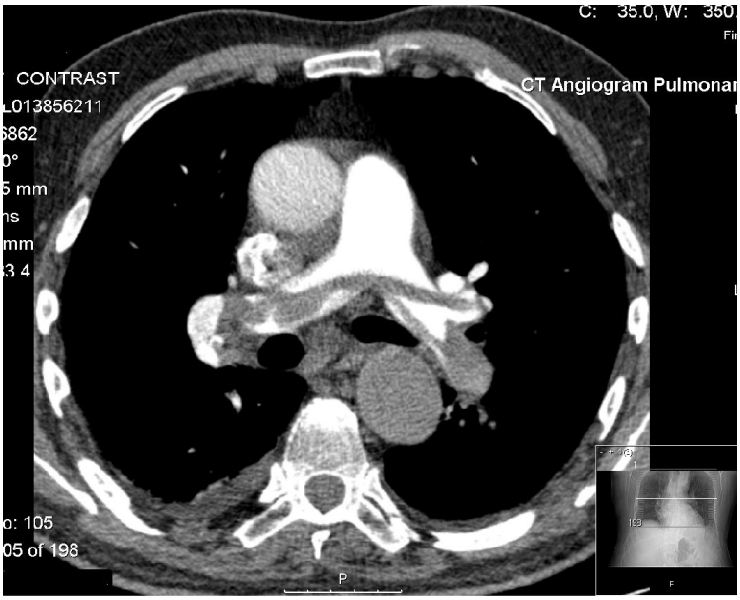


wyraźnie widać, że dopiero dla rozpiętości 180 cały obraz zostaje odtworzony, co wyjaśniałoby zachowanie wykresu. Ponadto dla wartości większych od 180 występują coraz bardziej widoczne linie, co też jest odwzorowane na wykresie niewielkim wzrostem. Można więc powiedzieć, że w przypadku zmian rozpiętości błąd RMSE dobrze odwzorowuje subiektywną ocenę obrazu.

5.2 Część druga

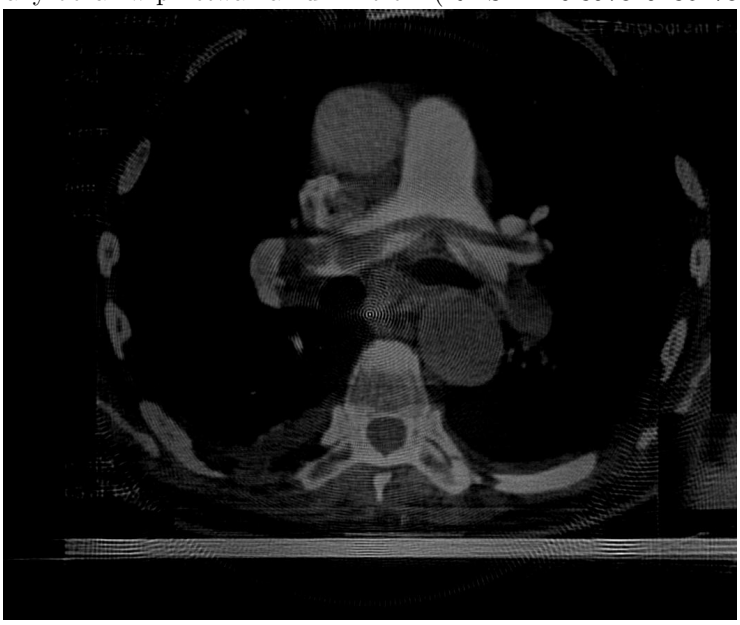
Wszystkie obrazy były przetwarzane dla 360 dekoderów, 360 skanów oraz rozpiętości równej 270 stopni.

Pierwszy obraz:



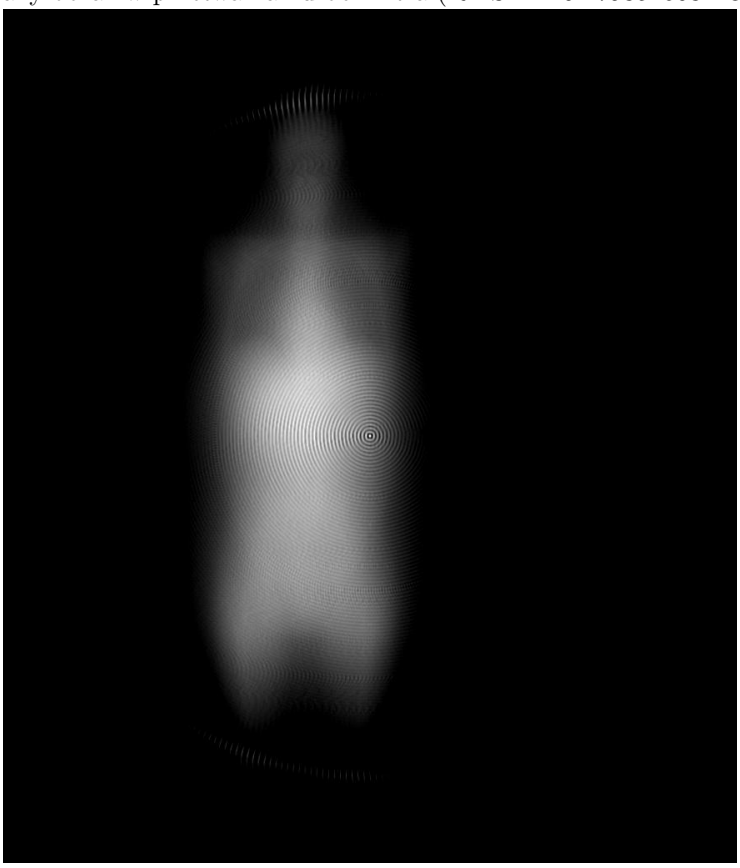
Otrzymany obraz w przetwarzaniu bez filtra (RMSE = 0.3921467677326692):

Otrzymany obraz w przetwarzaniu z filtrem (RMSE = 0.39781943647810575):

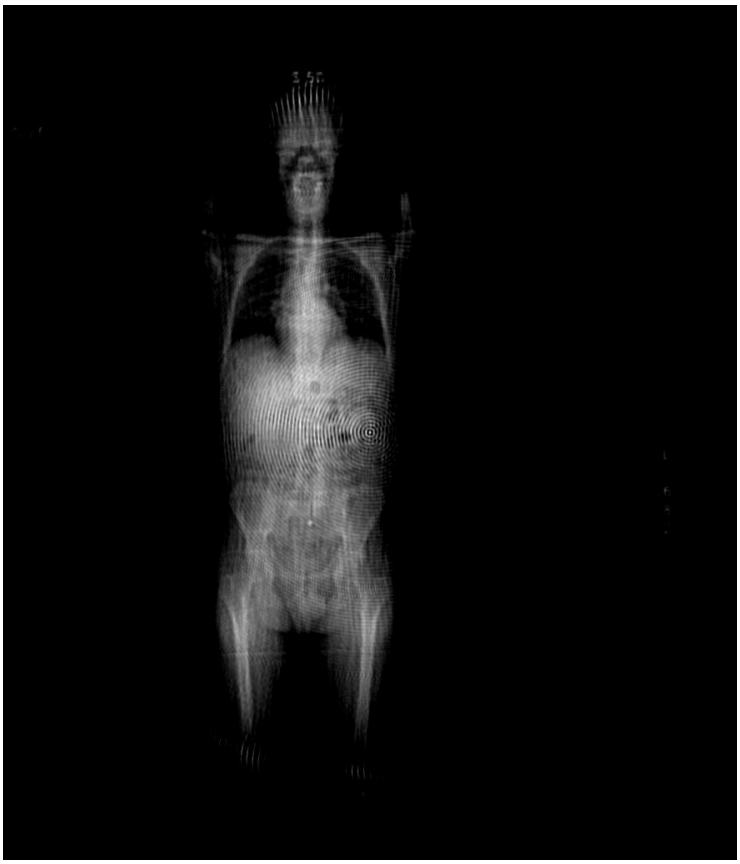


Drugi obraz:

Otrzymany obraz w przetwarzaniu bez filtra (RMSE = 0.17935460341877366):



Otrzymany obraz w przetwarzaniu z filtrem (RMSE = 0.14339777897960196):



Jak można zauważyć zastosowanie filtra w obu przypadkach znacząco poprawiło jakość obrazu oraz umożliwiło rozpoznanie jakichkolwiek szczegółów. Co ciekawe w pierwszym przypadku RMSE po zastosowaniu filtra minimalnie wzrosło, a w drugim znacząco się zmniejszyło. Może to być spowodowane różnicami w obrazach, mianowicie rozłożeniem jasnych i ciemnych elementów.

6 Informacje dodatkowe

GUI należy uruchamiać z folderu Python komendą:

```
streamlit run gui.py
```

W innym przypadku program nie będzie mógł wczytać biblioteki utworzonej z programu napisanego w C (transf.so).