Symulacja tomografu komputerowego przy użyciu Transformaty Radona

Aleksandra Jarzyńska
Grzegorz Bryk

Spis treści

1 Wstęp

Celem projektu jest wykonanie symulacji Tomografu Komputerowego (CT) przy użyciu Transformaty Radona. Obraliśmy stożkowy model tomografu, z jednym emiterem i wieloma detektorami.

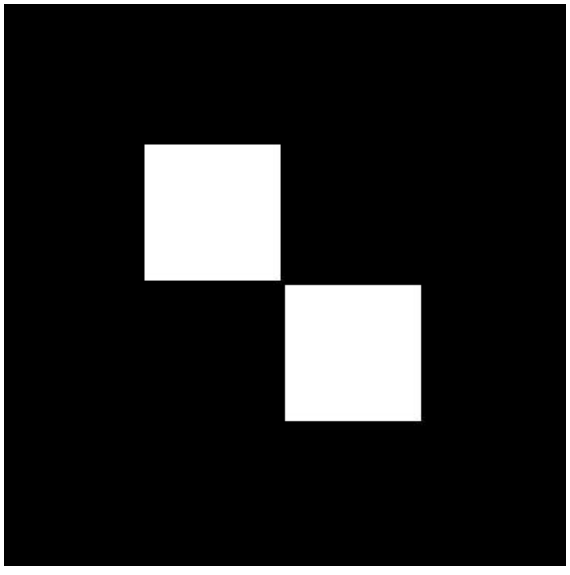
1.1 Technologie

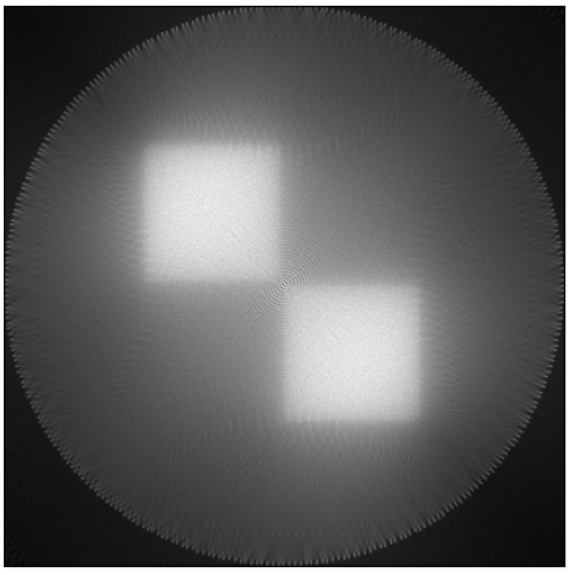
Do implementacji symulacji wykorzystaliśmy język Python w wersji 3.8. GUI zostało zaimplementowane w bibliotece Tkinter. Do obliczeń użyliśmy biblioteki NumPy, a do wizualizacji wyników Matplotlib.

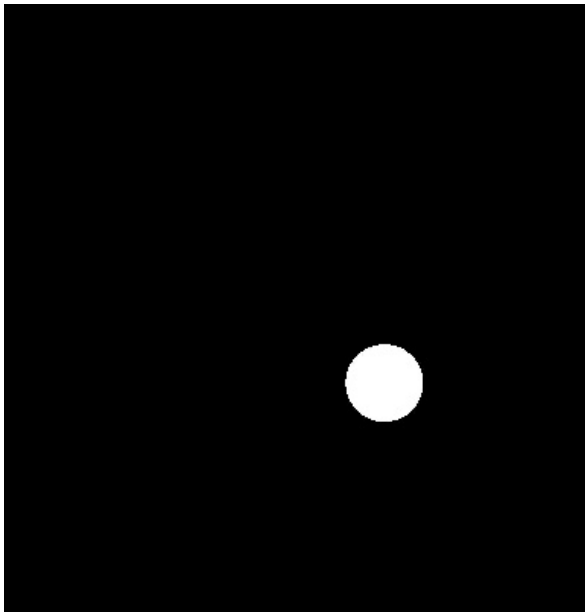
Ponieważ po profilowaniu pierwszego rozwiązania okazało się, że funkcja realizująca algorytm Brasenhama zajmuje około 70% czasu całego wykonania cyklu: obraz oryginalny → sinogram → rekonstrukcja obrazu, zdecydowaliśmy się wyekstrachować tę funkcję do osobnego pliku. W ten sposób powstała jej implementacja w języku C, zapisana w pliku brasenham.c, którą skompilowaliśmy do biblioteki brasenham.so; ta biblioteka jest importowana w kodzie Pythona i wykorzystywana do obliczania przebiegu linii. To pozwoliło skrócić czas wykonywania całości algorytmu o ok. 50%.

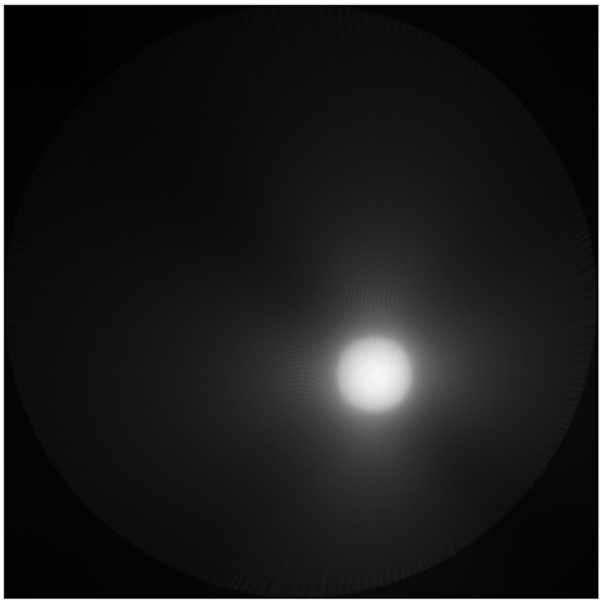
1.2 Przykładowe rezultaty

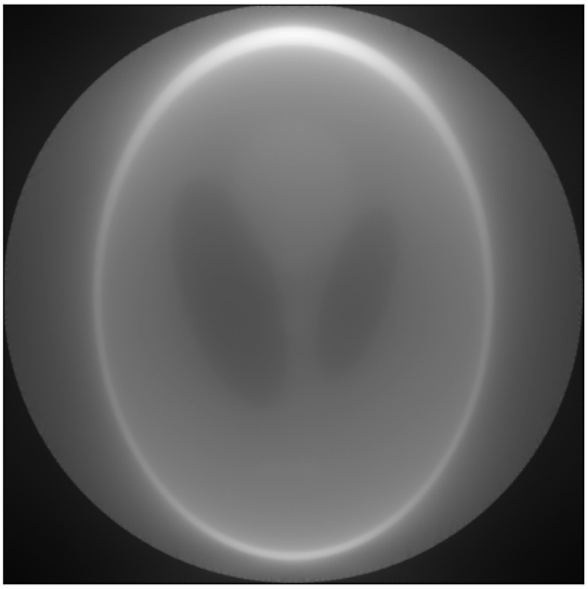
W wyniku działania algorytmu otrzymaliśmy następujące rekonstrukcje przykładowych obrazów:











2 Opis implementacji

2.1 Algorytm Brasenhama

Algorytm Brasenhama służy wyznaczeniu współrzędnych kolejnych pikseli odpowiadających dyskretnemu przybliżeniu przebiegu odcinka o określonym początku i końcu. W naszej implementacji podanie współrzędnych końca służy jedynie wyliczeniu kierunku biegu półprostej, a punkty są wyznaczane aż do osiągnięcia krawędzi obrazu. Algorytm zrealizowano w języku C, w osobnym pliku brasenham.c; oto jesgo zawartość:

```c
#include <stdbool.h>
#include <stdlib.h>

void brasenham(int height, int width, int y0, int x0, int y1, int x1, bool *result) {
    //result[y][x] = result[y * width + x];

    int dx = x1 - x0;
    int dy = y1 - y0;

    int xsign = dx / abs(dx);
    int ysign = dy / abs(dy);

    dx = abs(dx);
    dy = abs(dy);

    int xx, xy, yx, yy;

    if (dx > dy) {
        xx = xsign;
        xy = 0;
        yx = 0;
        yy = ysign;
    } else {
        int tdy = dy;
        dy = dx;
        dx = tdy;
        xx = 0;
        xy = ysign;
        yx = xsign;
        yy = 0;
    }

    int D = 2 * dy - dx;
    int x = 0, y = 0;

    int dy2 = 2 * dy;
    int dx2 = 2 * dx;

    int xr = x0 + x * xx + y * yx;
    int yr = y0 + x * xy + y * yy;

    while (0 <= xr && xr < width && 0 <= yr && yr < height) {
        result[yr* width + xr] = true;

        if (D >= 0) {
            y += 1;
            D -= dx2;
        }

        D += dy2;
        x += 1;

        xr = x0 + x * xx + y * yx;
        yr = y0 + x * xy + y * yy;
    }
}
```

Listing 1: brasenham.c

Początek pliku to importy potrzebnych bibliotek. *stdbool.h* dostarcza typ *bool* oraz stałe *true* i *false*. Plik *stdlib.h* dostarcza funkcję *int* ***abs****(int)*.
Poniżej następuje definicja funkcji brasenham;

2.1.1 Argumenty

- heigth – całkowita wysokość obrazu na którym wyznaczana jest prosta,
- width – całkowita szerokość obrazu na którym wyznaczana jest prosta,
- y0 – współrzędna pionowa punktu początkowego półprostej,
- x0 – współrzędna pozioma punktu początkowego półprostej,
- y1 – współrzędna pionowa punktu końcowego odcinka,
- x1 – współrzędna pionowa punktu końcowego odcinka,
- result – wskaźnik na obszar pamięci w którym zapisane zostają wyniki działania funkcji.

Ze względu na to, że funkcja jest przeznaczona do współpracy z kodem pythonowym, dość specyficznie zwraca rezultat swoich obliczeń. Przed wywołaniem, z poziomu Pythona alokowana jest tablica typu *bool* o rozmiarze *heigth*width*. Wspomniana tablica (wskaźnik na jej początek) jest przekazywana do kodu w C jako argument *result*. *W* ten sposób w tablicy mamy po jednym bicie na każdy piksel obrazu na którym pracujemy. Ponieważ użycie tablicy dwuwymiarowej wymagałoby tworzenia i rozwiązywania tablicy wskaźników, korzystamy z przekształcenia tablicy dwuwymiarowej na jednowymiarową zgodnie z wzorem *tab[y][x] = tab[y * width + x]*. Algorytm, wyznaczając kolejne wartości *y, x*, zamiast je zwracać, ustawia w strukturze *result* bit odpowiadający danemu miejscu w obrazie – *result[y][x]*.

Plik został skompilowany do pliku biblioteki dołączalnej *brasenham.so* z pomocą poniższego polecenia:
cc --std=c11 -fPIC -shared -o brasenham.so brasenham.c

2.2 Klasa Radon

Klasa Radon zawiera implementację symulacji tomografu, obejmuje to zarówno przekształcanie obrazu źródłowego w sinogram, jak i odwrotną transformatę Radona, z sinogramu do obrazu odtworzonego. Oprócz tego zawiera implementację filtrowania obrazu wynikowego oraz animacji zarówno sinaogramu jak i rekonstrukcji.

2.2.1 Konstruktor

Oto początek owej klasy:

```
class Radon:
    brasenham_lib = CDLL("/home/prance/PycharmProjects/IwM/CT/brasenham.so")

    def __init__(self, bitmap_path: str, da: float, detectors_no: int, span: float,
                 dicom: bool = False):  # da, span in radians

        if dicom:
            self._dicom = DICOMhandler().load(bitmap_path)
            self._bitmap = self._dicom.bitmap
        else:
            self._bitmap = plt.imread(bitmap_path).astype('float64')
            if len(self._bitmap.shape) == 3:
                self._bitmap = self._bitmap[:, :, 0]

        self._h, self._w = self._bitmap.shape
        self._sinogram = None
        self._center = np.array((self._h - 1, self._w - 1)) / 2
        self._da = da
        self._steps = int(2 * np.pi / da)
        self._initial_emitter_vector = np.array((0, (self._w - 1))) / 2
        self._rotation_angle = 0
        self._emitter = self._center + self._initial_emitter_vector
        self._detectors_no = detectors_no
        self._emitter_to_1st_detector = np.pi - (span / 2)
        self._detectors = np.zeros((detectors_no, 2))
        self._angle_between_detectors = span / (detectors_no - 1)
        self._calculate_detectors()
        self._reconstructed_bitmap = None
        self._reconstructed_unnormed = None
        self._c_array_type = c_bool * (self._h * self._w)
```

Listing 2: Konstruktor klasy Radon

Na początku, jako parametr wspólny dla wszystkich instancji klasy, ustawiany zostaje *brasenham_lib*. Jego wartością jest biblioteka zaimportowana z pliku *brasenham.so* – wynik kompilacji pliku *brasenham.c*.
Poniżej znajduje się konstruktor klasy.

Argumenty

- bitmap_path – ścieżka do obrazu źródłowego,
- da – Δα – krok układu emiter-detektory wyrażony w radianach,
- detectors_no – liczba detektorów,
- span – rozstaw detektorów wyrażony w radianach,
- dicom – parametr okreslający czy obraz podany w *bitmap_path* jest zapisany w formacie DICOM.

2.2.2 Ogólny zarys modelu i sposobu realizacji algorytmów

Zdecydowaliśmy się wykonywać wszelkie obliczenia w układzie współrzędnych bitmapy, to jest w układzie Y, X, gdzie Y to oś pionowa o współrzędnych rosnących „na południe”, a X to oś pozioma o współrzędnych rosnących „na wschód”. W tym układzie współrzednych środek obrazu, a co za tym idzie środek obrotu układu emiter-detektory wypada w punkcie (*height / 2, width / 2*). Jednak obrót wektora o początku w punkcie innym niż początek układu współrzędnych jest o wiele bardziej skomplikowany. Stąd wyliczanie pozycji emitera i detektorów w każdym kroku jest najpierw przeprowadzane przy założeniu że środek układu znajduje się w początku układu współrzędnych, a potem wszystkie wyliczone pozycje są przesuwane o wektor (*height / 2, width / 2*).

```
def _rotate(self):
    self._rotation_angle += self._da
    self._calculate_emitter()
    self._calculate_detectors()

def _calculate_emitter(self):
    mag = np.linalg.norm(self._initial_emitter_vector)
    s = np.sin(self._rotation_angle)
    c = np.cos(self._rotation_angle)
    self._emitter = (mag * s, mag * c)
    self._emitter += self._center

def _calculate_detectors(self):
    start_to_detector = self._rotation_angle + self._emitter_to_1st_detector
    mag = np.linalg.norm(self._initial_emitter_vector)
    for i in range(self._detectors_no):
        s = np.sin(start_to_detector)
        c = np.cos(start_to_detector)
        self._detectors[i] = (mag * s, mag * c)
        start_to_detector += self._angle_between_detectors
    self._detectors += self._center
```

Listing 3: Obliczanie pozycji emitera i detektorów

Na początku każdego skanu zmienna *rotation_angle* jestg powiększana o wartość *da*. W ten sposób zawsze przechowuje kąt między pozycją początkową, a obecną pozycją emitera. Funkcja `_calculate_emitter` służy wilczeniu pozycji emitera przy zadanym kącie obrotu względem pozycji początkowej. Dokonuje tego przy pomocy znanego wzoru (Lsinα, Lcosα), gdzie L to długość wektora.

Ponieważ początkową pozycję emitera ustaliliśmy na (*heigth* / 2, *width*), to wektor położenia emitera ma długość *width / 2* (jednak na wypadek zmiany położenia początkowego, każdorazowo jego długość jest wyliczana funkcją norm).
Funkcja cd wytlicza pozycje detektorów. Wykorzystuje do tego powyższy wzór, przy czym kąty obrotów są wyliczane poprzez dodanie do obecnego kąta obrotu kąta między emiterem a pierwszym detektorem i dodawanie odpowiedniej wielokrotności rozstawu detektorów podzielonego przez ich liczbę. Na koniec całość przesuwana jest o wektor do środka układu.

2.2.3 Integracja z biblioteką brasenham.so

```
def _brasenham(self, p0, p1):
    result = self._c_array_type()
    y0, x0 = map(int, np.round(p0))
    y1, x1 = map(int, np.round(p1))
    self.brasenham_lib.brasenham(self._h, self._w, y0, x0, y1, x1, byref(result))

    result = np.array(result)
    result.shape = self._h, self._w

    return result
```

Listing 4: _brasenham

Funkcja najpierw alokuje pamięć na wynik działania funkcji w C, następnie rozkłąda punkt początkowy i końcowy na współrzędne, po czym wywołuje funkcję biblioteczną z odpowiednimi parametrami. Po wypełenieniu przez bibliotekę, struktura wynikowa jest konwertowana do typu NumPy.Array, a następnie jej kształt jest dostosowywany z jednowymiarowej, do dwuwymiarowej zgodnej z wymiarami obrazu. Tak spreparowany wynik jest zwracany.

2.2.4 Konstrukcja sinogramu

```
def _sinogram_step(self, step: int, anim: bool = False):
    self._rotate()
    for i, d in enumerate(self._detectors):
        self._sinogram[i][step] = np.mean(self._bitmap[self._brasenham(self._emitter, d)])

    if anim:
        norm = self._sinogram.max()
        return self._sinogram / norm * 255.0

def sinogram(self):
    self._sinogram = np.zeros((self._detectors_no, self._steps), dtype='float64')
    for i in range(self._steps):
        self._sinogram_step(i)

    norm = self._sinogram.max()
    self._sinogram = self._sinogram / norm * 255.0
    return self._sinogram

def sinogram_animated(self):
    self._sinogram = np.zeros((self._detectors_no, self._steps), dtype='float64')
    fig = plt.figure()

    a = self._sinogram_step(0, anim=True)
    im = plt.imshow(a, interpolation='none', aspect='auto', vmin=0, vmax=1, cmap='gray')

    def animate_func(i):
        im.set_array(self._sinogram_step(i, anim=True))
        return [im]

    ani = anim.FuncAnimation(
        fig,
        animate_func,
        frames=range(1, self._steps),
        interval=1
    )
    plt.show()
```

Listing 5: Funkcje konstrukcji sinogramu

Sinogram jest konstruowany poprzez wyznaczenie odcinków od emitera do każdego z detektorów z pomocą algorytmu Brasenhama, a następnie zapisaniu jako wartości odczytanej na detektorze średniej wartości pikseli z tego odcinka. Normalizacja sinogramu to pomnożenie wartości każdego z jego pikseli przez 255 i podzielenie przez największą wartość z sinogramu. Wartym wytłumaczenia trickiem w powyższym kodzie jest indeksowanie bitmapy bezpośrednio wynikiem funkcji *_brasenham*. Wykorzystujemy tutaj interfejs klasy NumPy.Array, która pozwala na indeksowanie z pomocą struktury Boolowskiej, o wymiarach tożsamych z wymiarami struktury indeksowanej. Jest to wielokrotnie szybsze w wykonaniu niż iterowanie przez wszystkie pary (y, x) zwracane przez algorytm Brasenhama w wersji klasycznej i indeksowanie nimi bitmapy.

2.2.5 Rekonstrukcja obrazu i filtrowanie

Rekonstrukcja obrazu orygialnego przebiega bardzo podobnie do konstrukcji sinogramu.

```
def _reset(self):
    self._rotation_angle = 0
    self._emitter = self._center + self._initial_emitter_vector
    self._calculate_detectors()

def _reconstruction_step(self, step: int, anim: bool = False):
    self._rotate()
    for i, d in enumerate(self._detectors):
        line = self._brasenham(self._emitter, d)
        self._reconstructed_unnormed[line] += self._sinogram[i][step]

    if anim:
        print(step)
        norm = self._reconstructed_unnormed.max()
        return self._reconstructed_unnormed / norm * 255.0

def reconstruct(self, filter=True):
    self._reset()
    self._reconstructed_unnormed = np.zeros((self._h, self._w), dtype='float64')
    for i in range(self._steps):
        self._reconstruction_step(i)

    if filter:
        self.convolve()

    self._normalize()
    return self._reconstructed_bitmap

def _normalize(self):
    self._reconstructed_bitmap = self._reconstructed_unnormed * 255.0 /
self._reconstructed_unnormed.max()

def convolve(self, k=100, mode='constant'):
    k = np.array([[10, 10, 10],
                  [10, k, 10],
                  [10, 10, 10]])
    self._reconstructed_unnormed = convolve(self._reconstructed_unnormed, k, mode=mode)

    return self._reconstructed_unnormed
```

Listing 6: funkcje odpowiedzialne za rekonstrukcję obrazu

Na początku emiter i detektory są ustawiane w pozycji startowej. Następnie na każdej z pozycji, dla każdego detektora wyznaczany jest odcinek emiter-detektor, a wszystkie punkty na tym odcinku są rozjaśniane o wartość odczytu z detektora przy danym kącie obrotu. Potem odtworzony obraz jsplatany (jeśli filtr został włączony) z maską 3x3 i wagą oryginalnego piksela 10, a otaczających go 1. Na koniec obraz jest normalizowany.

3 Wyniki eksperymentu

W ramach eksperymentu, z pomocą powyższego kodu, oraz funkcji *mse* implementującej miarę RMSE

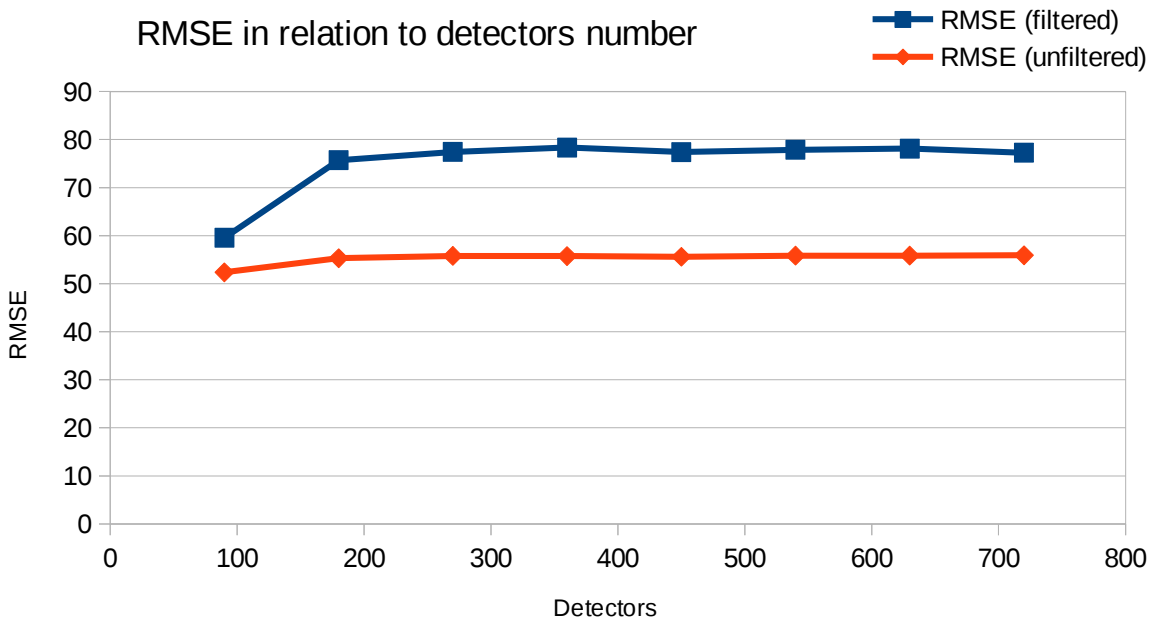
```
class Tests:
    @staticmethod
    def mse(orig, final):
        err = np.sum((orig - final) ** 2)
        err /= orig.shape[0] * orig.shape[1]
        res = np.sqrt(err)
        return res
```

Listing 7: początek klasy Tests

wykonano badanie wpływu parametrów symulacji na dokładność rekonstrukcji obrazu Shepp_logan.jpg – typowego testowego obrazu przedstawiajacego sztuczne przybliżenie przekroju głowy ludzkiej.

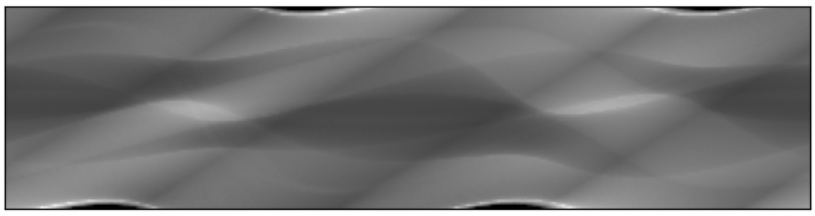
Rysunek 1: Shepp_logan.jpg

3.1 Zmiana RMSE w zależności od liczby detektorów

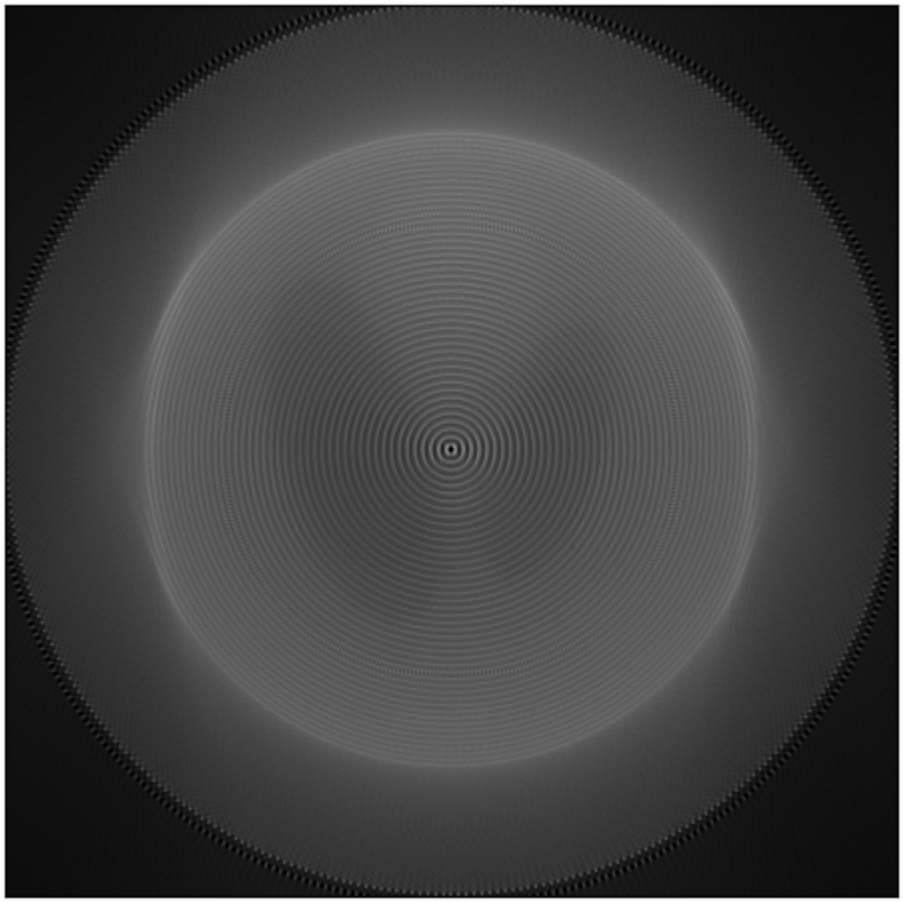


Wykres 1: RMSE w zalezności od liczby detektorów

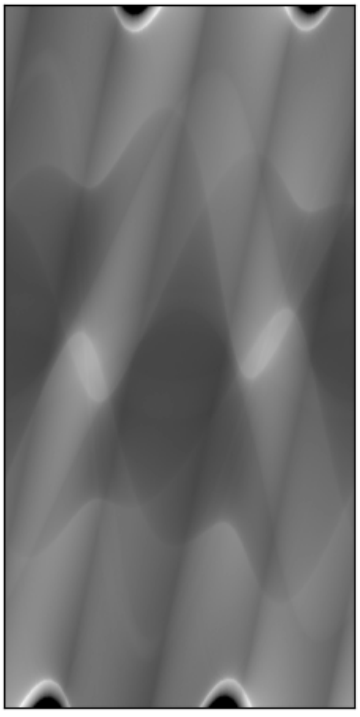
Jak widać, RMSE nie zmienia się znacznie wraz ze zmianą liczby detektorów, pomimo że różnica w subiektywnym odbiorze obrazu jest znaczna.



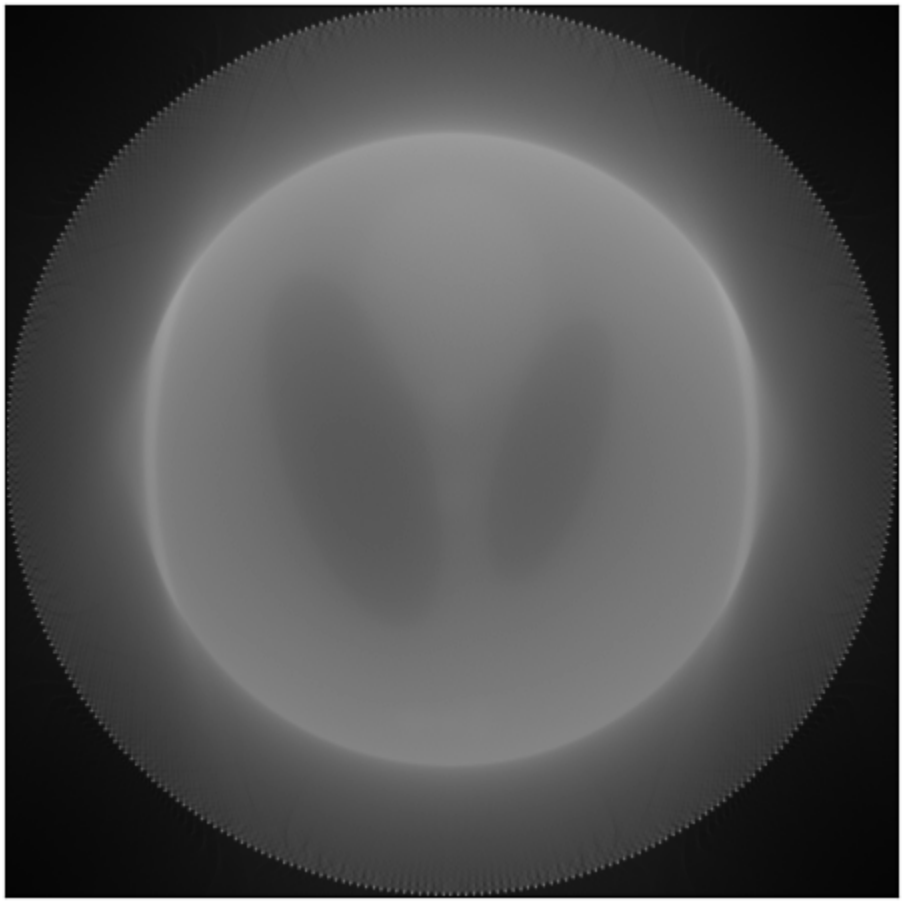
Rysunek 2: Sinogram dla 90 detektorów



Rysunek 3: Rekeonstrukcja dla 90 detektorów

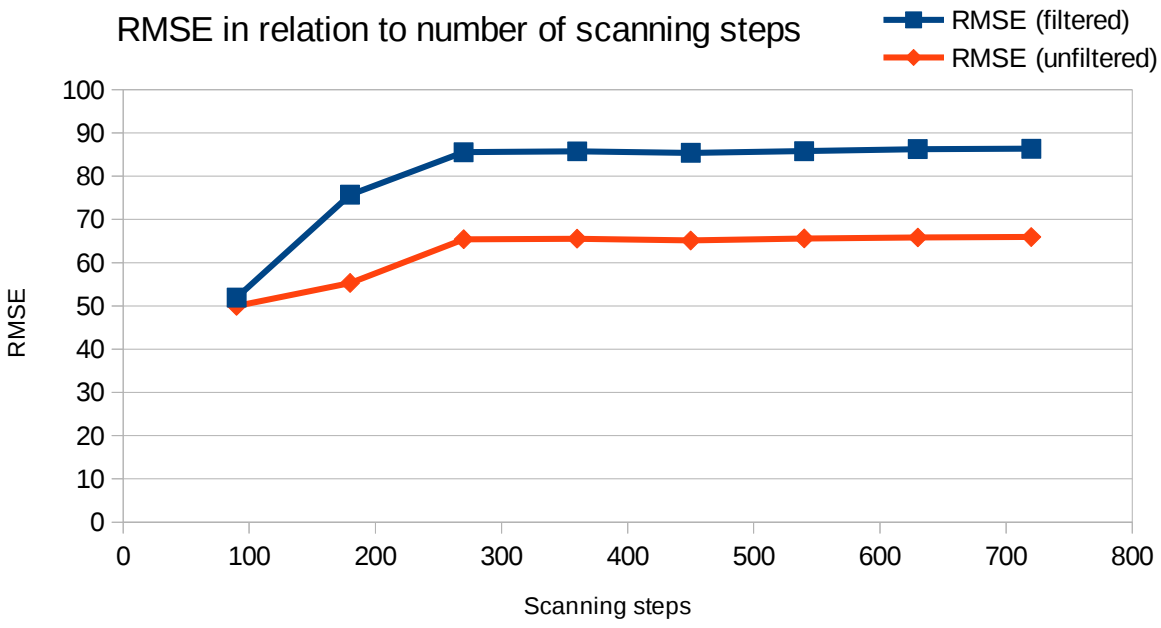


Rysunek 4: Sinogram dla 720 detektorów



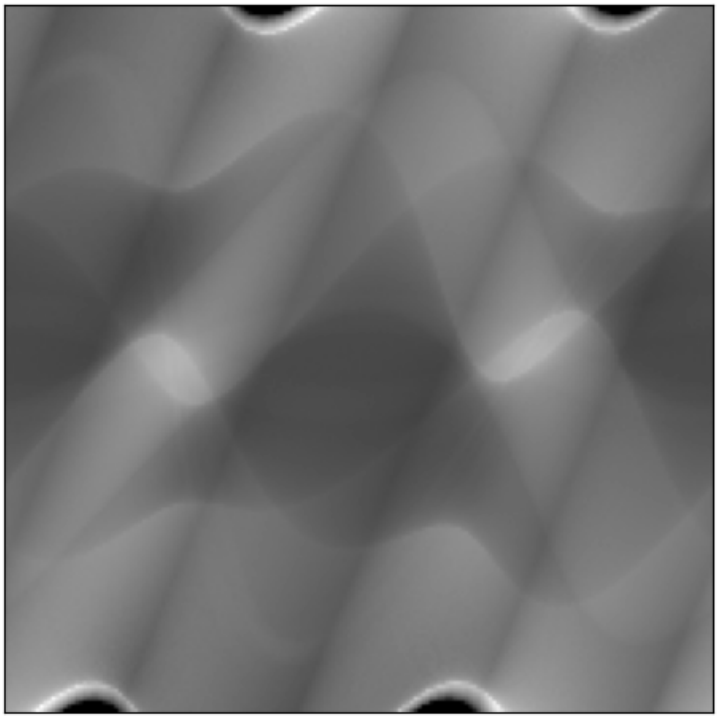
Rysunek 5: Rekonstrukcja przy 720 detektorach

3.2 Zmiana RMSE w zależności od liczby skanów

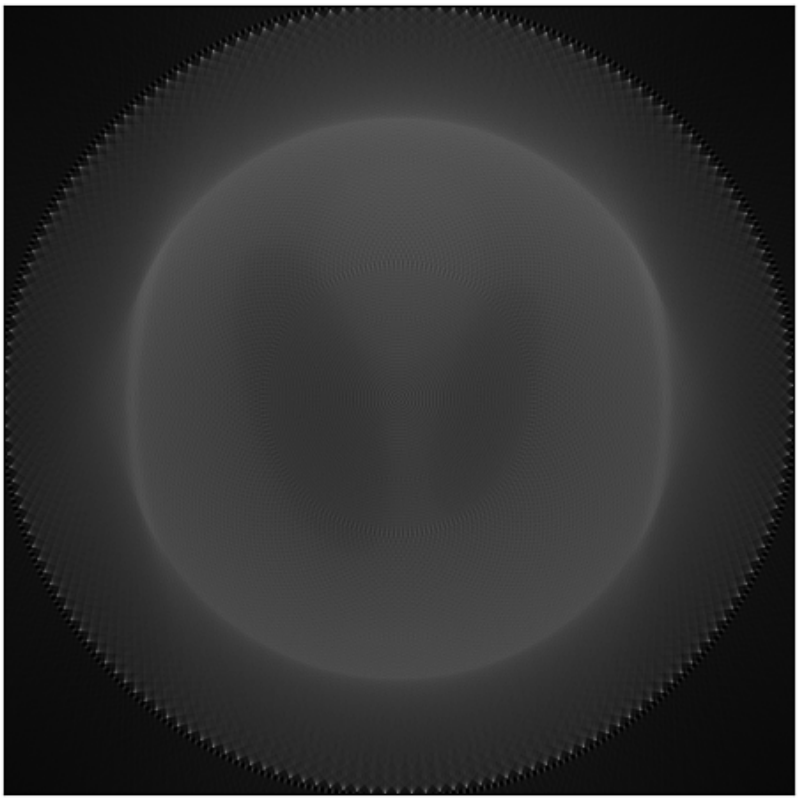


Wykres 2: RMSE w zależności od liczby skanów

Jak widać na wykresie, RMSE wbrew oczekiwaniom rośnie wraz z rosnącą liczbą skanów, aż w od ok. 300 skanów pozostaje na stałym poziomie. To po raz kolejny przeczy badaniu organoleptycznemu.

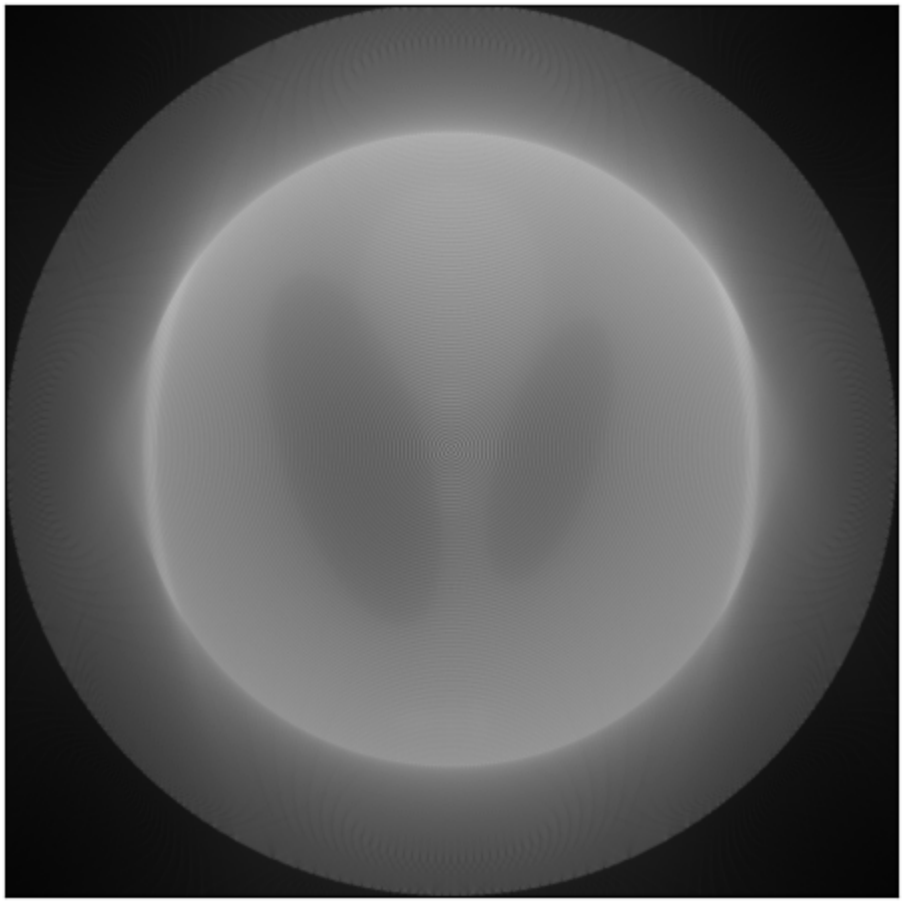


Rysunek 6: Sinogram dla 90 skanów



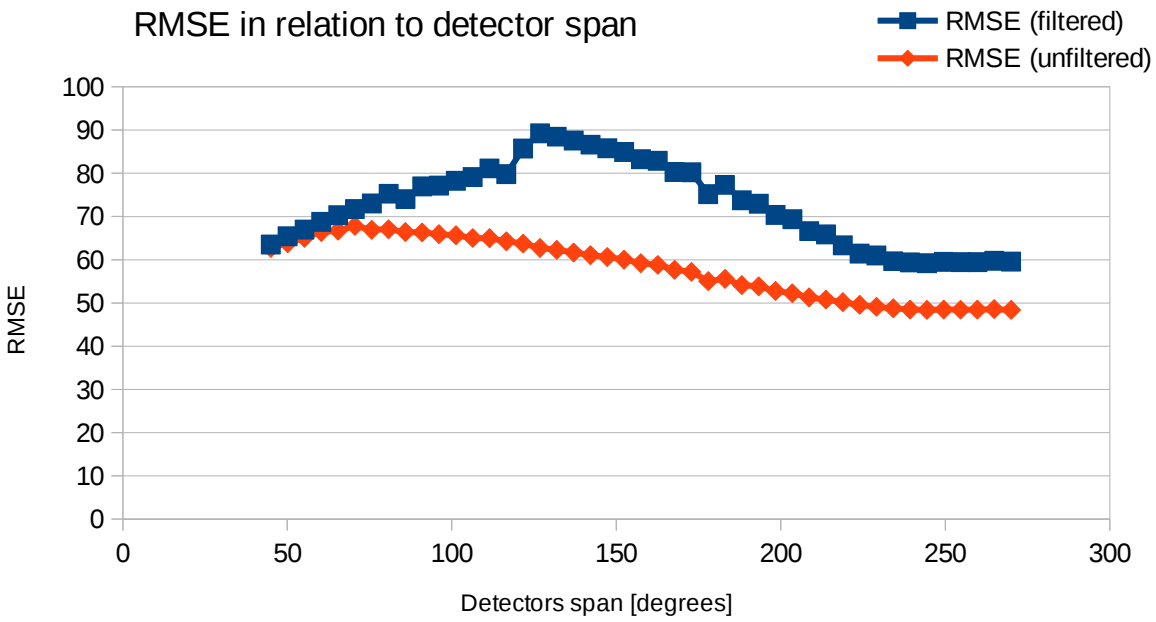
Rysunek 7: Rekonstrukcja dla 90 skanów

Rysunek 8: Sinogram dla 720 skanów



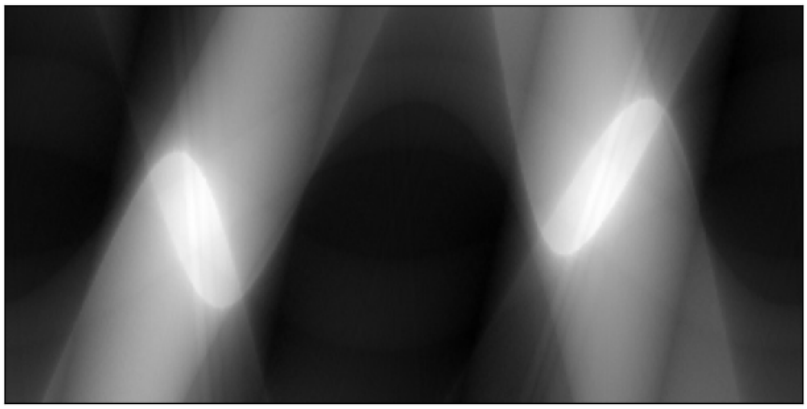
Rysunek 9: Rekonstrukcja dla 720 skanów

3.3 RMSE w zależności od rozstawu detektorów

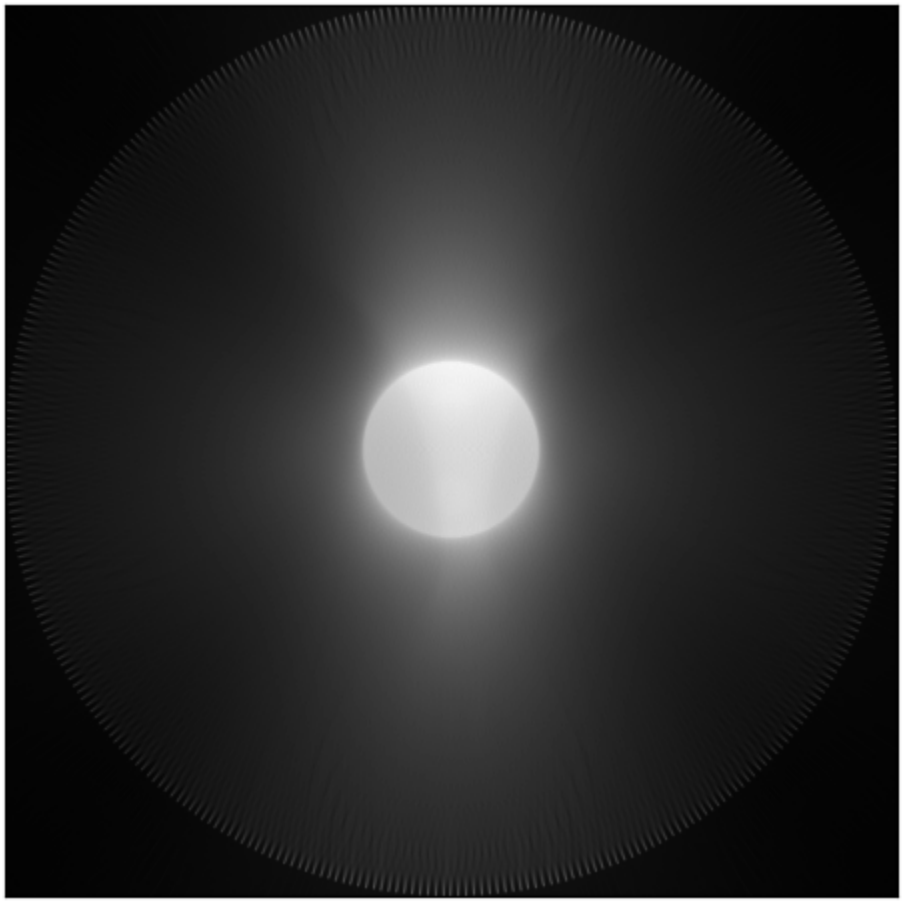


Wykres 3: RMSE w zależności od rozstawu detektorów

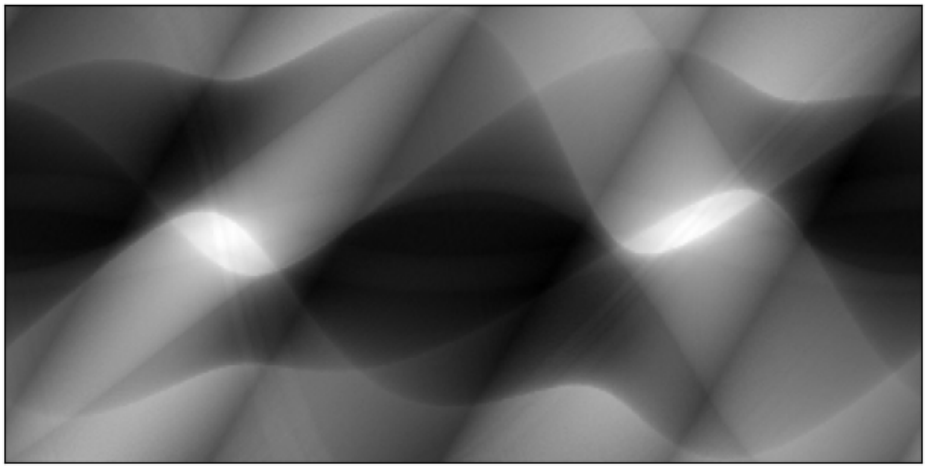
W tym przypadku otrzymujemy nieco niespójne wyniki – okazuje się że RMSE do pewniego momenstu rośnie, a od ok 120° zaczyna maleć w przypadku obrazu uśrednianego splotem, ale od początku konsekwentnie maleje dla obrazu niefiltrowanego. Generalnie można przyjąć, że w obu przypadkach najlepiej sprawdzają się bardzo duże rozstawy detektorów. Pokrywa się to z obserwacjami ludzkimi.



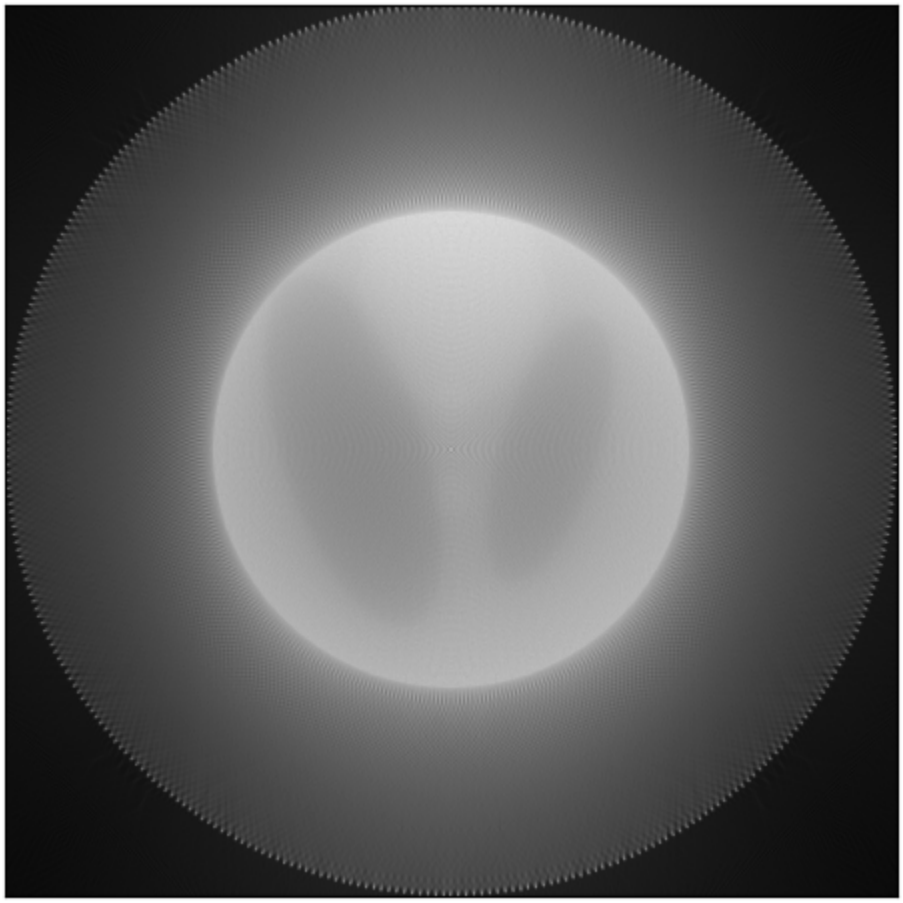
Rysunek 10: Sinogram przy rozwartości 45°



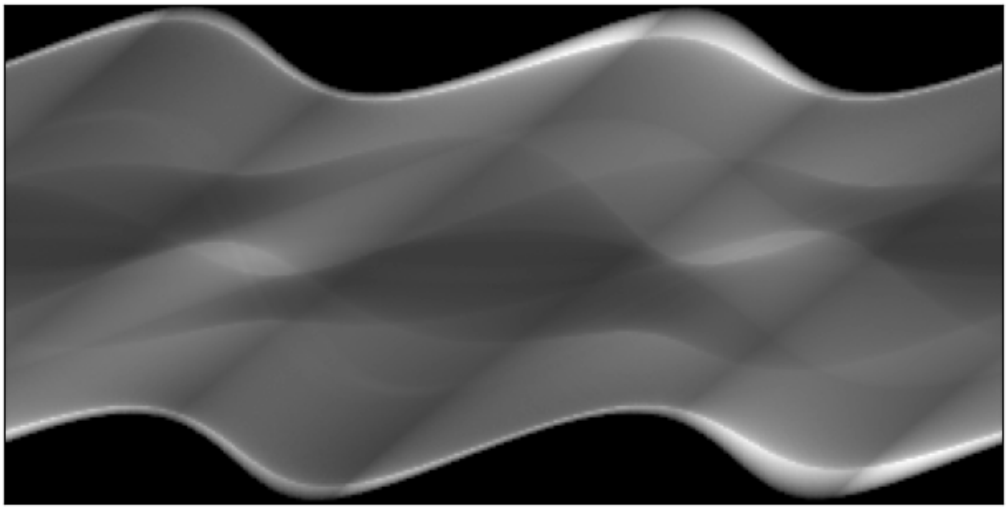
Rysunek 11: Rekonstrukcja przy rozwartości 45°



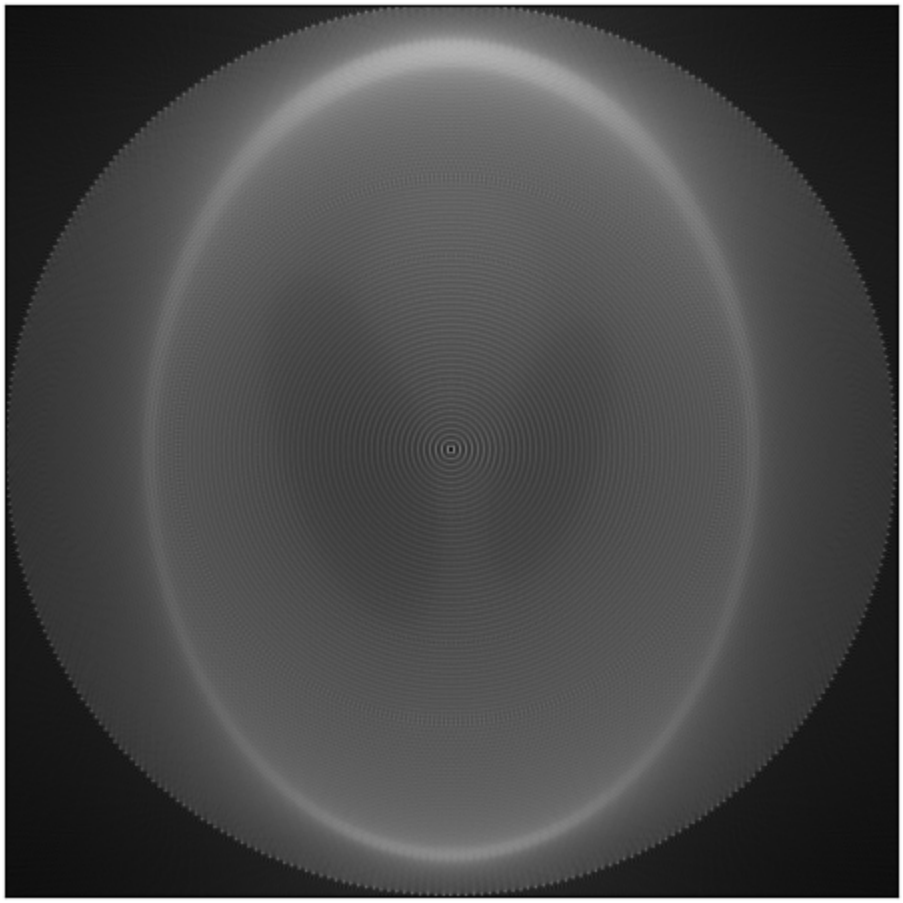
Rysunek 12: Sinogram przy rozwarości 128° (okolice maksymalnego RMSE przy filtrowanym obrazie)



Rysunek 13: Rekonstrukcja przy rozwartości 128° (okolice maksymalnego RMSE przy filtrowanym obrazie)

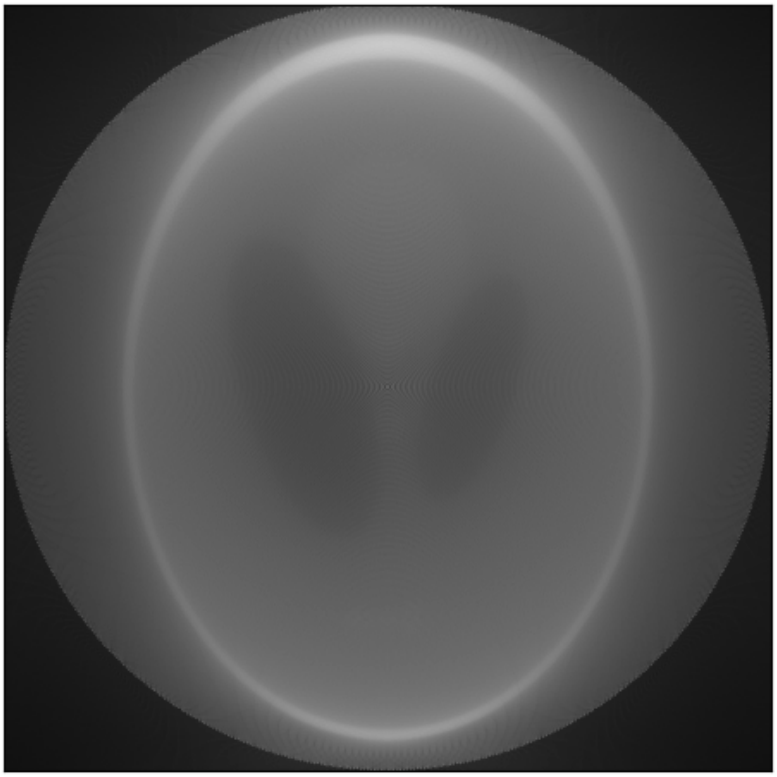


Rysunek 14: Sinogram przy rozwartości 270°

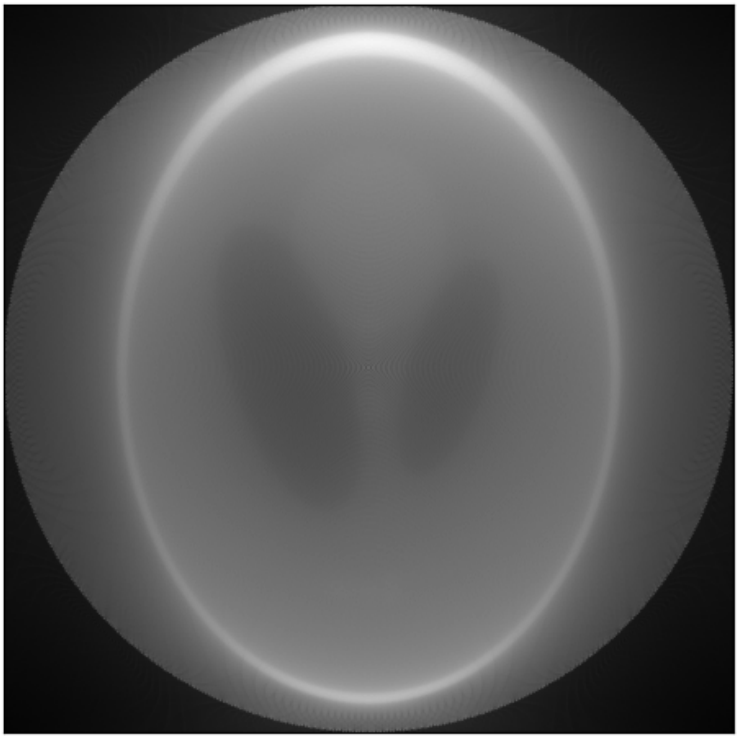


Rysunek 15: Rekonstrukcja przy rozwartości 270°

3.4 Porównanie RMSE obrazu przed i po konwolucji

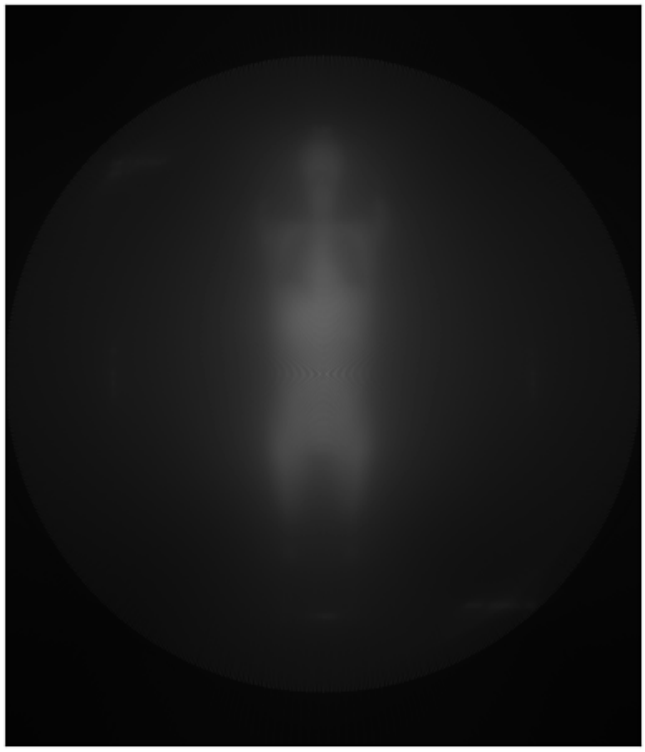


Rysunek 16: Rekonstrukcja bez konwolucji; RMSE = 67

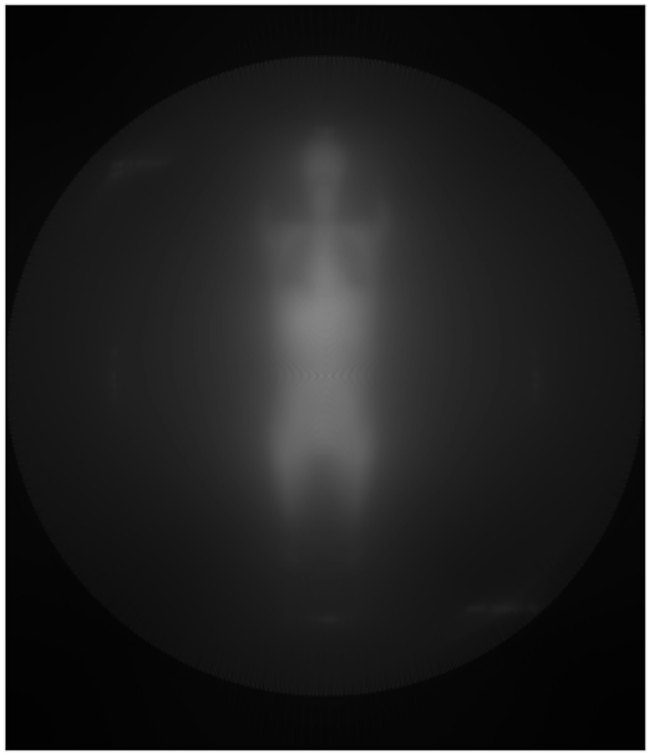


Rysunek 17: Rekonstrukcja po konwolucji; RMSE = 81,63

Jako drugi obraz wybraliśmy CT_ScoutView-large.jpg.



Rysunek 18: Rekonstrukcja przed konwolucją; RMSE = 28,08



Rysunek 19: Rekonstrukcja po konwolucji; RMSE = 32

4 Źródła

https://www.dicomlibrary.com/dicom/dicom-tags/

https://www.pclviewer.com/help/required_dicom_tags.htm

https://en.wikipedia.org/wiki/DICOM

https://gist.github.com/fubel/ad01878c5a08a57be9b8b80605ad1247

https://www.researchgate.net/publication/320616840_Implementation_of_the_Radon_transform_based_on_the_array_of_sources_and_reconstruction_with_Filtered_Back_Projection_algorithm

https://github.com/antego/radon/blob/master/radon.cpp

https://arxiv.org/pdf/1512.09140.pdf

http://www.dsp.agh.edu.pl/_media/pl:lukasz_mitka_10.01.2011.pdf

http://wmii.uwm.edu.pl/~panas/talks/inauguracja.pdf

https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/

https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html

https://github.com/encukou/bresenham