Image Reconstruction from Sinogram

Team members:
Jufeng Yang ID: 20125011
Xingda Zhou ID: 19107471
Zhongen Qin ID: 19107579

Table of Content

1. Introduction

The main objective of this project is to reconstruct a sine wave image of a picture into the original picture that can be recognised by human vision. Most of the time, for example, hospital MRI's using X-rays tend to produce pictures that are various sine wave images. The principle of reconstructing these sine wave images into something recognisable to humans will be similar. Images can be optimised using filters in refactoring. For images that do not need to be filtered, refactoring can be done directly, but the results of refactoring are often not as good as expected. For the parts where filters are used, we first need to perform a Laplace transform on the sine wave graph. After filtering the data in the frequency domain using a ramp filter, the final data is finally subjected to an inverse Laplace transform. This will result in a grey-scale image of several channels, which will be combined to mean that all the work is done.

In this project, I will compare the effect between images reconstructed with or without the ramp filter, and those reconstructed with the hamming-windowed filter.

2. Data Process

2.1 Import Relative Libraries

Import imutils, which contains image reading functions, image display functions, channel reading functions, etc. numpy is used for array data. scipy.fftpack is used for fast Fourier transforms. Import rotate from scimage.transform for reconstructing images. import matplotlab for storing data. Import OpenCV to process images.

```
# Group Member:
#  Name                Student ID
#  Jufeng Yang         20125011
#  Xingda Zhou         19107471
#  Zhongen Qin         19107579

# Import useful libries.
import imutils
import numpy as np
import cv2
import scipy.fftpack as fft
from matplotlib import pyplot as plt
from skimage.transform import rotate
```

Figure.1: Useful libraries.

2.2 Single Channel Views

The original image needs to be read before the channel can be separated. The original picture is displayed.

```
# Check the Sinogram.
print("Original Sinogram")
sinogram2 = imutils.imread('sinogram.png')
imutils.imshow(sinogram2)
print(sinogram2.shape)

#Using the openCV to check the Sinogram.
sinogram = cv2.imread('sinogram.png')
cv2.imshow('Original Sinogram',sinogram)

cv2.waitKey(0)
# closing all open windows
cv2.destroyAllWindows()
```

Figure.2: Code for showing original images.

Using the image reading tool provided by imutils, the final image is displayed in 2-D. This does not get the single channel. Thus, I am using OpenCV to read the images and display them. Subsequent operations will be performed on the results returned by OpenCV. As shown in Figure.3

However, imutils can perform channel separation on the original image. Interestingly, the channels in imutils are not in the same order as the OpenCV channels. The images are stored as BGR in OpenCV, but RGB in imutils. I think it is possible to use either way, it just needs to be done in a different order for the final composition. The results of the separation are shown

in Figure 4 and Figure 5.

Original Pictures:

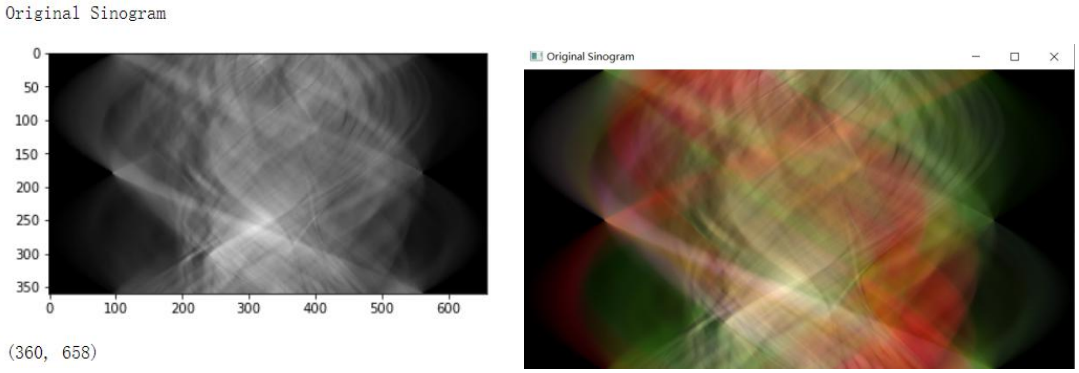


Figure.3: Original sinogram. Show by imutils (Left), Shown by OpenCV (Right).

Separate pictures by imutils:

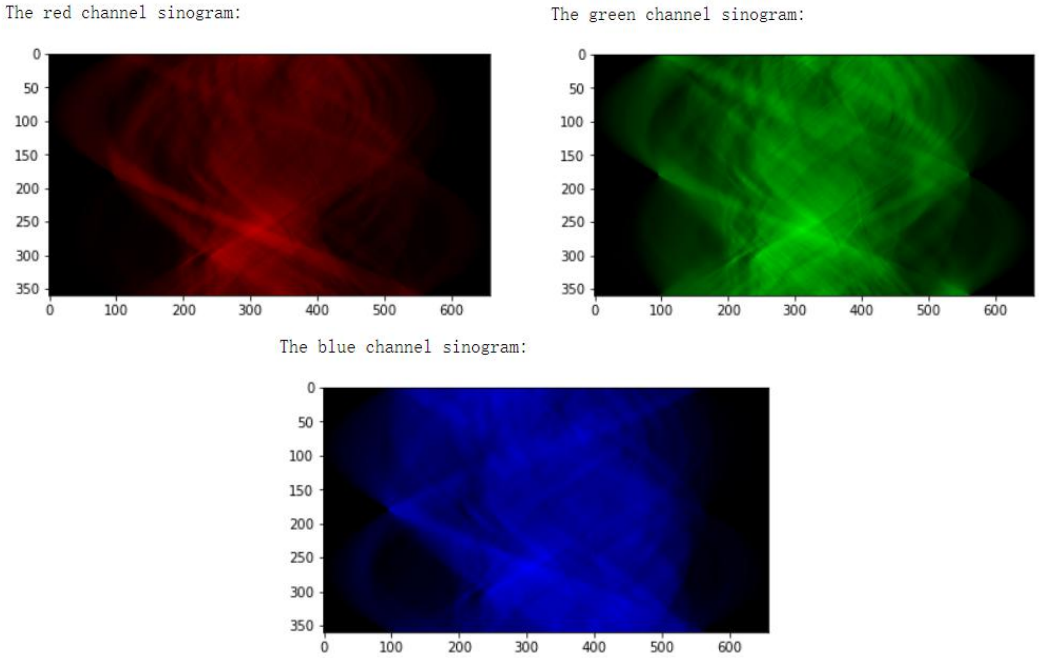
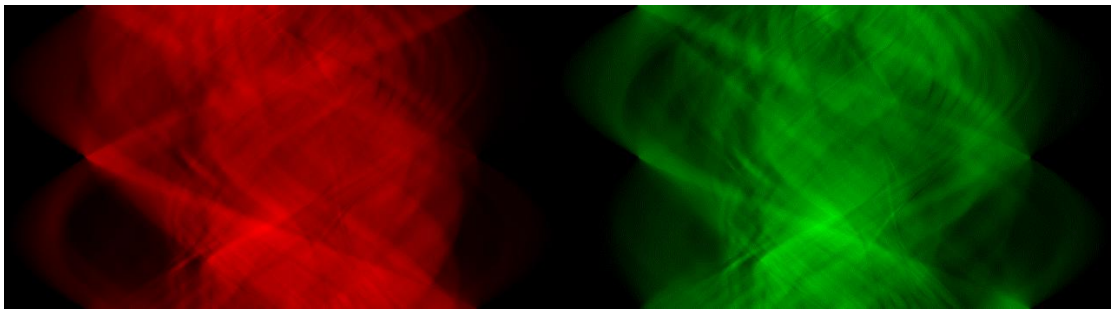


Figure.4: The 3 channels of originals generated by imutils.

Separate pictures by OpenCV:



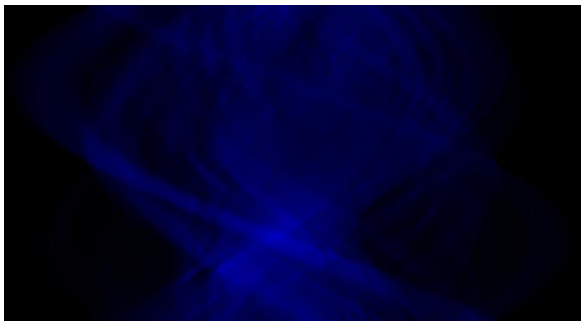


Figure.5: The 3 channels of originals generated byOpenCV.

From the perspective of the original image, OpenCV's order is more in line with the order of the original image.

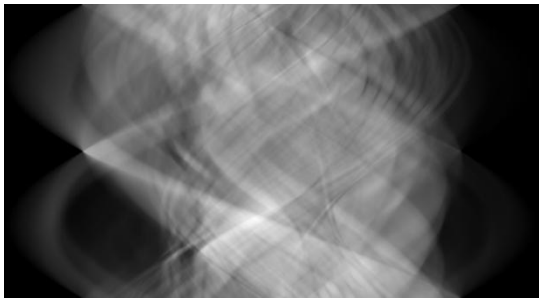
2.3 Grey – Scale Generation

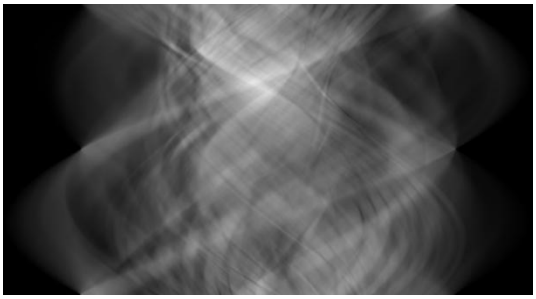
Channel slicing of the original image using OpenCV. After separating the channels, the grayscale images of the three channels will be obtained. Save the three greyscale images. Design this method as a function. Save them separately.

```
34 def creat_1_D_Grey_Sinogram(sinogram, color):
35     sinogram_bule, sinogram_green, sinogram_red = cv2.split(sinogram)
36     if color == 'red':
37         cv2.imshow('Sinogram Red Grey', sinogram_red)
38         cv2.waitKey(0)
39         cv2.destroyAllWindows()
40         cv2.imwrite('Sinogram_red_grey.png', sinogram_red)
41     elif color == 'blue':
42         cv2.imshow('Sinogram Blue Grey', sinogram_bule)
43         cv2.waitKey(0)
44         cv2.destroyAllWindows()
45         cv2.imwrite('Sinogram_bule_grey.png', sinogram_bule)
46     elif color == 'green':
47         cv2.imshow('Sinogram Green Grey', sinogram_green)
48         cv2.waitKey(0)
49         cv2.destroyAllWindows()
50         cv2.imwrite('Sinogram_green_grey.png', sinogram_green)
51     else:
52         print("Wrong parameter")
53
54 creat_1_D_Grey_Sinogram(sinogram, 'blue')
55 creat_1_D_Grey_Sinogram(sinogram, 'red')
56 creat_1_D_Grey_Sinogram(sinogram, 'green')
```

Figure.6: Split original pictures and generate gray scale pictures.

Since the method of image reconstruction cannot accept 3-D data, separating the channels to get a grey-scale image is a must. The library used in the separation is OpenCV, so the result after separation is similar to the previous single-channel image. This is shown in Figure 7.





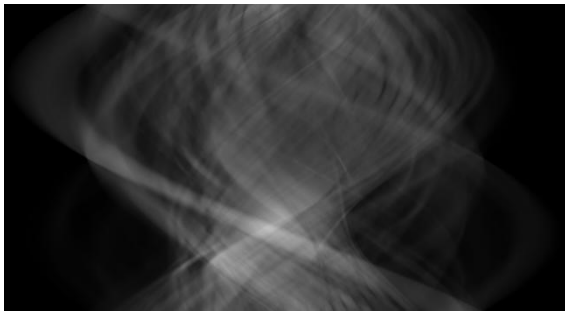


Figure.7: Gray scale pictures (2-D), R, G, B in order.

2.4 Methods Creation

Construct a Radon transform that will initially transform the image.

```
#Radon transform method - turns an image into a sinogram (Not used for reconstruction - this
#is how the original sinogram was generated

def radon(image, steps):
    #Build the Radon Transform using 'steps' projections of 'image'.
    projections = []        ## Accumulate projections in a list.
    dTheta = -180.0 / steps ## Angle increment for rotations.

    for i in range(steps):
        projections.append(rotate(image, i*dTheta).sum(axis=0))

    return np.vstack(projections) # Return the projections as a sinogram
```

Figure.8: Create a radon transform.

Provides a fast Fourier transform method that operates on data after a radon change.

```
#Translate the sinogram to the frequency domain using Fourier Transform
def fft_translate(projs):
    #Build 1-d FFTs of an array of projections, each projection 1 row of the array.
    return fft.rfft(projs, axis=1)
```

Figure.9: Return a fast Fourier result.

After the Fourier variation has been completed, all data is transformed to the frequency domain and the data in the frequency domain is processed using a ramp filter to remove the unnecessary amount of interference. The implementation of the method is shown in Figure 10.

```
#Filter the projections using a ramp filter
def ramp_filter(ffts):
    #Ramp filter a 2-d array of 1-d FFTs (1-d FFTs along the rows).
    ramp = np.floor(np.arange(0.5, ffts.shape[1]//2 + 0.1, 0.5))
    return ffts * ramp
```

Figure.10: Ramp filter implement.

After completing the filtering in the frequency domain, the frequency domain is reconverted back to the time domain using inverse Fourier variation. The Fourier change method is shown in the figure

```python
#Return to the spatial domain using inverse Fourier Transform
def inverse_fft_translate(operator):
    return fft.irfft(operator, axis=1)

```

Figure.11: Inverse fast Fourier transform.

After returning the filtered results from the time range, the image is reconstructed using an inverse transform. The reconstruction is done in the opposite way to turning the original image into a sine wave. The design of the method is shown in Figure 12.

```python
#Reconstruct the image by back projecting the filtered projections (UNFINISHED)
def back_project(operator):
    laminogram = np.zeros((operator.shape[1],operator.shape[1]))
    dTheta = 180.0 / operator.shape[0]
    for i in range(operator.shape[0]):
        temp = np.tile(operator[i],(operator.shape[1],1))
        temp = rotate(temp, dTheta*i)
        laminogram += temp
    return laminogram
```

Figure.12: Back project method to reconstruct the pictures.

3. Image Reconstruction

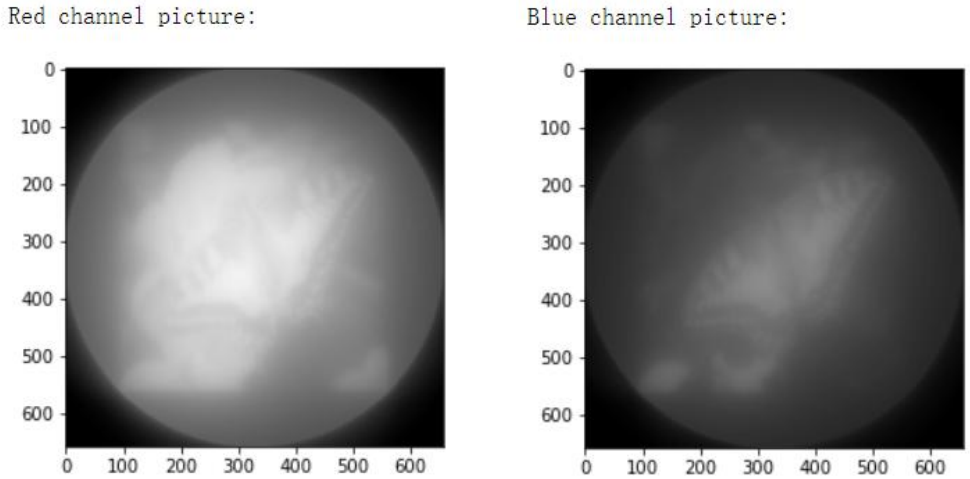
3.1 Reconstruct without filter

The sinogram is reconstructed directly into the original image, without any filters, and will not need to undergo any Fourier transform, and inverse Fourier transform. The grayscale images saved during the data processing stage are read directly, and the grayscale images are reconstructed, which will return three single-channel images that will be displayed. The three images are finally combined using numpy.dstack. Normalise all the data so that the luminance is displayed properly.

```
# NO ANY FILTERS RECONSTRUCTION.
# Reconstruct without any filter.
# Reconstruct using the grey pictures.
sino_bule = imutils.imread('Sinogram_bule_grey.png')
Reconstruct_nofilter_blue = back_project(sino_bule)

sino_red = imutils.imread('Sinogram_red_grey.png')
Reconstruct_nofilter_red = back_project(sino_red)

sino_green = imutils.imread('Sinogram_green_grey.png')
Reconstruct_nofilter_green = back_project(sino_green)

# Merge the 3 channels picture into a picture.
Reconstruct_nofilter = np.dstack((Reconstruct_nofilter_red, Reconstruct_nofilter_green, Reconstruct_nofilter_blue))
#Normalize the picture to avoid picture become too bright.
Reconstruct_nofilter = cv2.normalize(Reconstruct_nofilter, 0, 255, cv2.NORM_MINMAX)

# Show all single channel pictures and finnal result.
print("Red channel picture:")
imutils.imshow(Reconstruct_nofilter_red)

print("Blue channel picture:")
imutils.imshow(Reconstruct_nofilter_blue)

print("Green channel picture:")
imutils.imshow(Reconstruct_nofilter_green)

print("The merged pictrue:")
imutils.imshow(Reconstruct_nofilter)

```

Figure.13: Reconstruction without any filter.

The output of the three images without filters is displayed as shown in Figure 13.



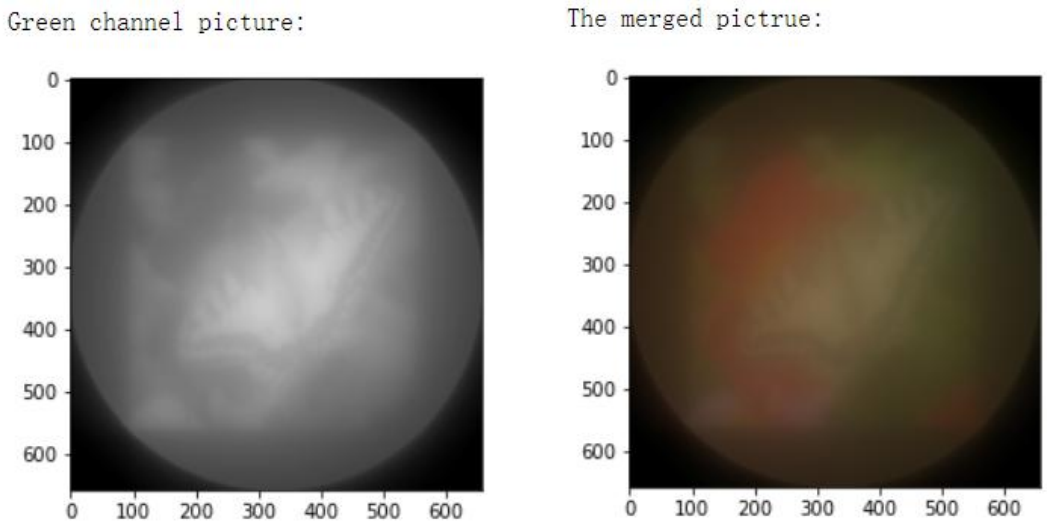


Figure.13: Single channel reconstructed picture (First 3 pictures) and merged pictures (Last pictures.)

It is easy to see from the picture that without any filter the picture is very confusing. Although the main content of the image can be seen, it is no longer possible to distinguish the details. In short, image reconstruction without any filters involved does not work well.

3.2 Reconstruct with ramp filter

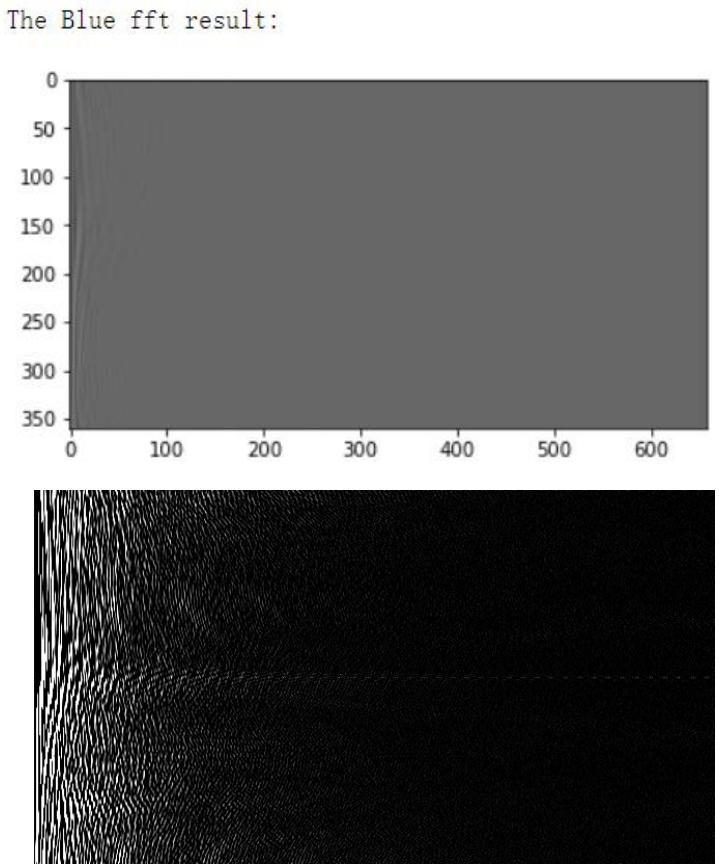
Unlike reconstruction without a filter, reconstruction with a ramp filter first requires a fast Fourier transform of the image data, and the filter can only be applied to the Fourier transformed data. After the filter, the data is inverse Fourier transformed and the final processed data is fed into the reconstruction function to obtain the final input image. In this sub-chapter I will demonstrate each step of the process in detail.

The First step is FFT. Code as shown in Figure 14.

```
# THROUGH THE FFT TRANS AND THE RAMP FILTER.

# Read three channel grey pictures from PC
sino_blue = imutils.imread('Sinogram_bule_grey.png')
sino_red = imutils.imread('Sinogram_red_grey.png')
sino_green = imutils.imread('Sinogram_green_grey.png')

# Fourier fast tranform for the 3 channel pictures.
sino_blue_fft = fft_translate(sino_blue)
sino_red_fft = fft_translate(sino_red)
sino_green_fft = fft_translate(sino_green)

#Show the transform results(pictures).
print("the fft result:")
imutils.imshow(sino_blue_fft)
imutils.imshow(sino_red_fft)
imutils.imshow(sino_green_fft)

#show the result use openCV.
cv2.imshow('sino_blue_fft',sino_blue_fft)
cv2.imshow('sino_red_fft',sino_red_fft)
cv2.imshow('sino_green_fft',sino_green_fft)

#cv2.imwrite('sino_blue_fft.png',sino_blue_fft)
#cv2.imwrite('sino_red_fft.png',sino_red_fft)
#cv2.imwrite('sino_green_fft.png',sino_green_fft)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure.14: Code for FFT.

The result after the Fast Fourier Transform is shown in Figure 15. Two displays are used to show the results. imutils shows the results less legibly. While OpenCV is very clear. Only the results for Blue's image are shown below. However, the results for the other two channels are similar.



Figuer.15: FFT result. Shown by imutils (Upper), Shown by OpenCV (Dowm).

Once the Fourier transform has been completed, it will be possible to pass all the data through the filter and display the filter results through imutils and OpenCV. The code is shown in Figure 16. The BLUE filtered results are shown in Figure 17.

```
#Through the transformed picture after Ramp Filter
filtered_sino_blue_fft = ramp_filter(sino_blue_fft)
filtered_sino_red_fft = ramp_filter(sino_red_fft)
filtered_sino_green_fft = ramp_filter(sino_green_fft)

# Show the filtered results(pictures)
print("Filtered FFT result:")
imutils.imshow(filtered_sino_blue_fft)
imutils.imshow(filtered_sino_red_fft)
imutils.imshow(filtered_sino_green_fft)

# Show filtered pictures useing openCV
cv2.imshow('f_blue',filtered_sino_blue_fft)
cv2.imshow('f_red',filtered_sino_red_fft)
cv2.imshow('f_green',filtered_sino_green_fft)

#cv2.imwrite('f_blue.png',filtered_sino_blue_fft)
#cv2.imwrite('f_red.png',filtered_sino_red_fft)
#cv2.imwrite('f_green.png',filtered_sino_green_fft)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure.16: Code use to filter FFT result.

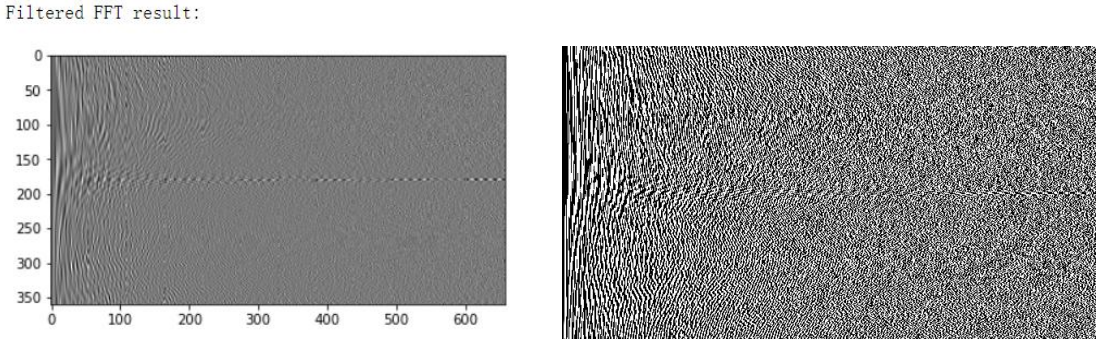
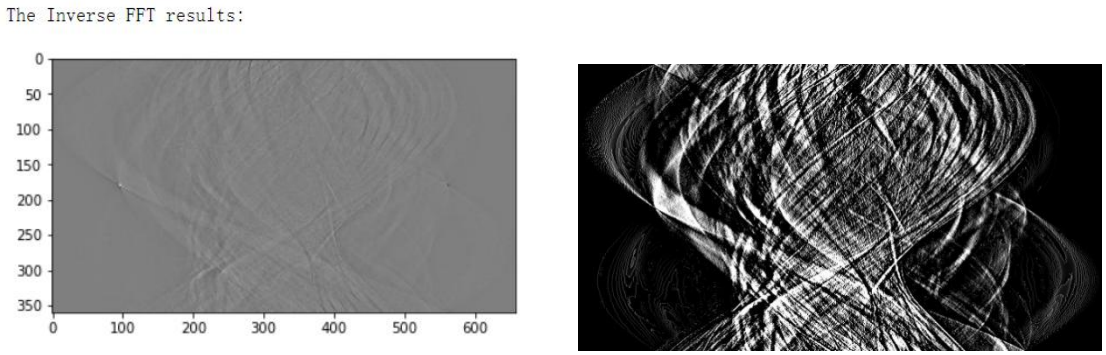


Figure.17: The filtered results (BLUE). Shown in imutils (Left), Shown in OpenCV (Right).

The next step is to perform an inverse Fourier transform on the filtered result to transform the result back to sinogram. the code implementation is shown in Figure 18 and the BLUE result is shown in Figure 19.

```
# Reverse FFT
filtered_sino_blue = inverse_fft_translate(filtered_sino_blue_fft)
filtered_sino_red = inverse_fft_translate(filtered_sino_red_fft)
filtered_sino_green = inverse_fft_translate(filtered_sino_green_fft)

# Show the reverse FFT results.
print("The Inverse FFT results: ")
imutils.imshow(filtered_sino_blue)
imutils.imshow(filtered_sino_red)
imutils.imshow(filtered_sino_green)

# Show the reverse FFT results using openCV.
cv2.imshow('new blue',filtered_sino_blue)
cv2.imshow('new red',filtered_sino_red)
cv2.imshow('new green',filtered_sino_green)

#cv2.imwrite('new blue.png',filtered_sino_blue)
#cv2.imwrite('new red.png',filtered_sino_red)
#cv2.imwrite('new green.png',filtered_sino_green)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure.18: Code for inverse FFT.

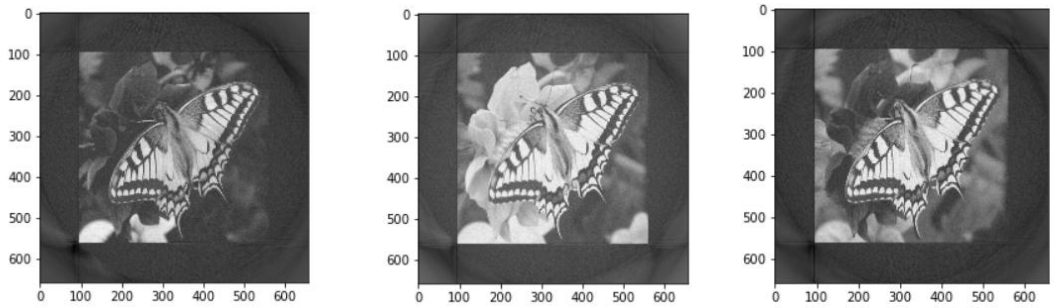


Figre.19: The inverse FFT (BLUE). Shown by imutils (Left), Shown by OpenCV (Right).

The final step reconstructs the graph after the inverse Fourier transform. This will give the original map for the three channels. The plots of the three channels are then combined and the final plots are regularised. The code implementation is shown in Figure 20 and the three channel result is shown in Figure 21. The merged and cropped results are shown in Figure 22.

```
# Reconstruct image using the filter
filtered_blue = back_project(filtered_sino_blue)
filtered_red = back_project(filtered_sino_red)
filtered_green = back_project(filtered_sino_green)

# Use last block generated reconstruct data.
Red_channel = filtered_red
Blue_channel = filtered_blue
Green_channel = filtered_green

# Show the data generated before.
imutils.imshow(Blue_channel)
imutils.imshow(Red_channel)
imutils.imshow(Green_channel)

# Merge those 3 channel into a BGR graph
Reconstruct_nofilter_with_filtered = np.dstack((Red_channel, Green_channel, Blue_channel))
print(Reconstruct_nofilter_with_filtered.shape)
Reconstruct_nofilter_with_filtered = cv2.normalize(Reconstruct_nofilter_with_filtered, 0, 255, cv2.NORM_MINMAX)
Reconstruct_nofilter_with_filtered = np.clip(Reconstruct_nofilter_with_filtered, 0,1)

Reconstruct_nofilter_with_filtered_ = Reconstruct_nofilter_with_filtered[97:561, 97:561, :]

# Show the reconstruct filtered fictures(Color)
imutils.imshow(Reconstruct_nofilter_with_filtered)
cv2.imshow('Reconstruct_nofilter_with_filtered', Reconstruct_nofilter_with_filtered)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure.20: Reconstruct code implement.



Figrue.21: 3 single channel reconstructed picture. B, R, G is order.

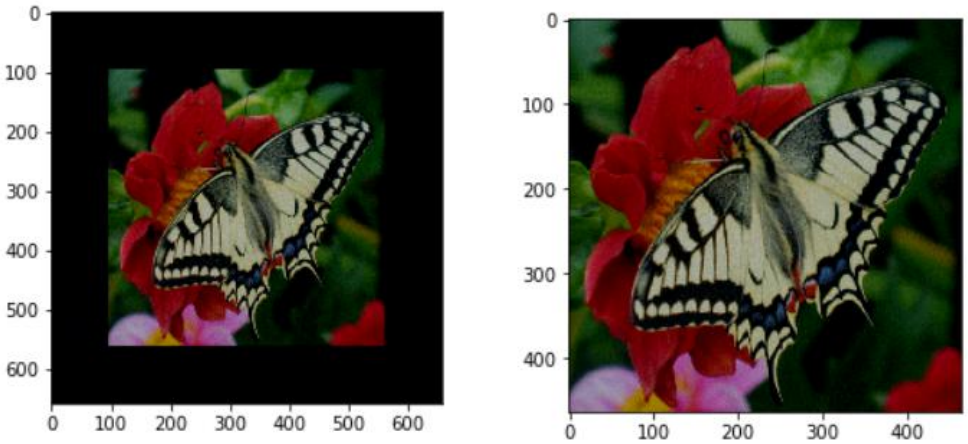


Figure.22: Merge 3 single channel pictures.

As can be seen in Fig. 20, the plots are different for all three channels and are all very clear. As can be seen in Fig. 21, the merged diagram has a wide black edge which needs to be partially cropped out. As the original diagram is square, the crop only needs to be a symmetrical section on both the left and right and the top and bottom.

3.3 Reconstruct with Hamming – windowed filter.

The Hamming-windowed filter is a very good filter. In some ways it actually works better than the ramp filter, as will be demonstrated later on. The Hamming-windowed filter is implemented in the same way as the ramp filter. The code implementation is shown in Figure 23. The output is shown in Figure 24.

```
# Using the Hamming filter to process pictures.
print("Hamming-Windowed reconstructed image")
window = np.hamming(658)

Hamming_filtered_sino_blue_fft = window * sino_blue_fft
Hamming_filtered_sino_red_fft = window * sino_red_fft
Hamming_filtered_sino_green_fft = window * sino_green_fft

# Reverse FFT
Hamming_filtered_sino_blue = inverse_fft_translate(Hamming_filtered_sino_blue_fft)
Hamming_filtered_sino_red = inverse_fft_translate(Hamming_filtered_sino_red_fft)
Hamming_filtered_sino_green = inverse_fft_translate(Hamming_filtered_sino_green_fft)

# Reconstruct image using the filter
Hamming_filtered_blue = back_project(filtered_sino_blue)
Hamming_filtered_red = back_project(filtered_sino_red)
Hamming_filtered_green = back_project(filtered_sino_green)

# Merge all 3 channel pictures.
hamming_new = np.dstack((Hamming_filtered_red, Hamming_filtered_green, Hamming_filtered_blue))
hamming_new = cv2.normalize(new, 0, 255, cv2.NORM_MINMAX)
imutils.imshow(hamming_new)
```

Figure.23: The code implementation of hamming – windowed filter.

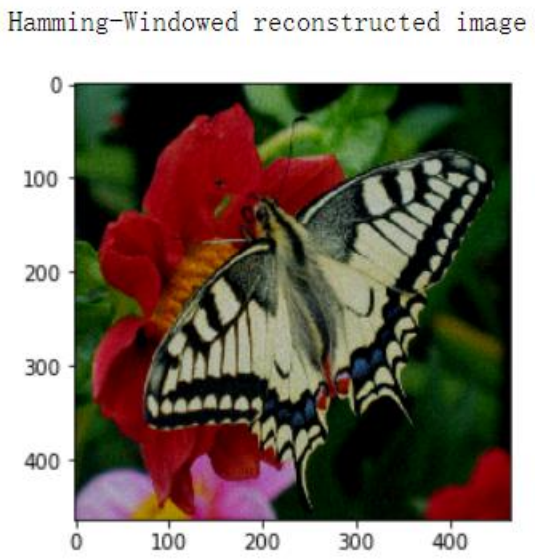


Figure.24: Hamming – windowed filtered result.

From the results, the results of the Hamming window do not need to be cropped; the function library itself will output the optimal result. Therefore the hamming - windowed result is optimal.

4. Conclusion

At this point, all the steps have been completed. We can draw some conclusions by comparing the three refactorings. The reconstruction method without any filters involved yields an output graph that is very blurred and granular. It can be guessed that the sinogram contains too many disturbing elements. Therefore, filters are a necessary part of the reconstruction method. The final output of the reconstruction method with the ramp filter is very clear, but the only disadvantage is that it contains a lot of black borders, which need to be cropped artificially. After cropping, the quality of the image is reduced, but the hamming - windowed filter will give you the final image directly without cropping. This is more than can be said for the ramp filter.

In my opinion, the advantage of the hamming - windowed filter over the ramp filter may not be fully appreciated in this case. hamming - windowed results will depend on the quality of the photo before it becomes a sine wave, so the final output is limited in terms of top line. The ramp filter also seems to give the most original image. Therefore, the hamming - windowed filter is the best, followed by the ramp filter.

5. Reference

[1] C. Flanagan. Image Reconstruction from a Sinogram [PDF]. Available: https://sulis.ul.ie/access/content/group/82edf4ed-f03d-4ab6-bf99-2773fa7cd801/Test%20Images/Project%20_1_%3A%20Image%20Reconstruction%20from%20a%20Sonogram/project1-sinogram.4up.pdf