Emre Ataklı - 2112944 **24.12.2019**

Duration to complete
Step-2: 10 days

EE415 Step 2 - Inverse Problem

Algorithm

General structure of the algorithm:

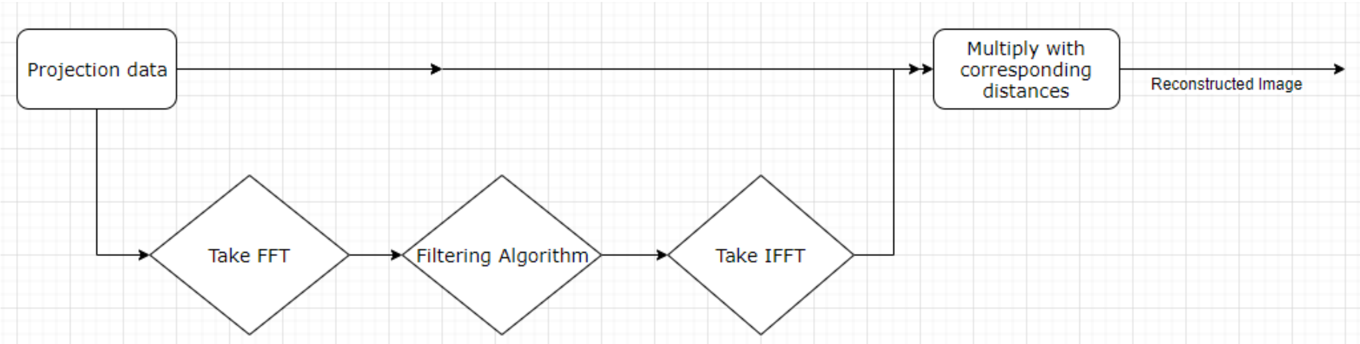


Figure 1: A brief flowchart of the overall algorithm.

Filtering part from the software:

```
def ramp_filter():
    fft_of_projection = fft2(image_to_be_reconstructed)          # Take FFT of the projection data
    t = np.linspace(0, 1, number_of_sampling_points)             # Design the ramp filter
    high_pass_filter = abs(abs(signal.sawtooth(2 * pi * t))-1)

    filtered_fft_of_projection = fft_of_projection * high_pass_filter    # Filter out the low frequency content

    ifft_of_projection = ifft2(filtered_fft_of_projection)       # Take IFFT of the filtered projection data
    return ifft_of_projection.tolist()
```

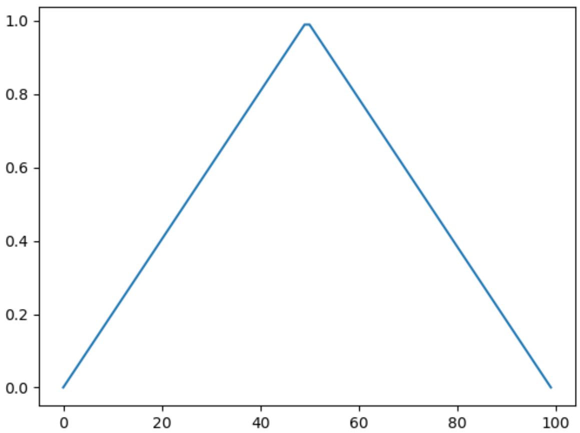
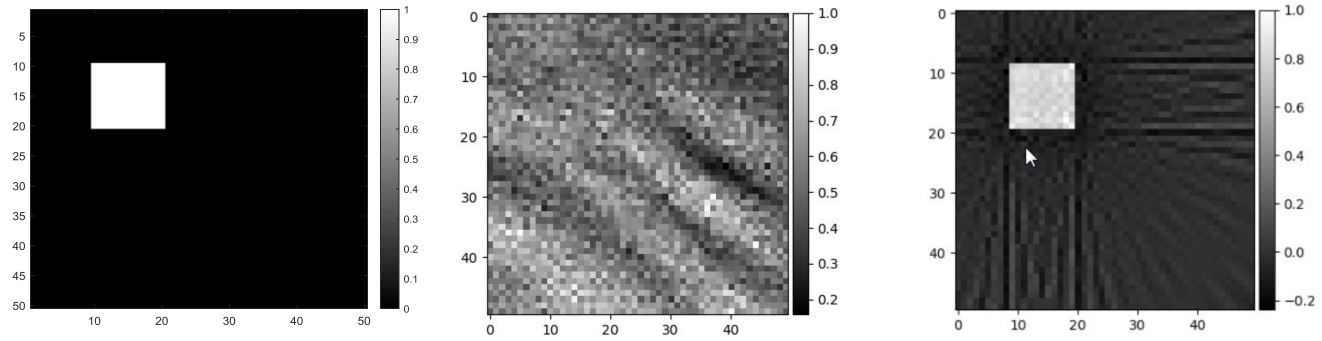


Figure 2: Implemented ramp filter.

The procedure of filtering part is very straightforward. To design the filter, I produced a ramp shaped curve. After multiplying the filter with FFT of the projection data, I have taken IFFT of the result and the back-projection data is formed. The plot of the implemented filter is given in Figure 2.

Results and Comments

Output 1:



(a) (b) (c)

Figure 3: (a) The square image (b) Reconstructed image without filtering (c) Reconstructed image with filtering when the number of beams=100 and step size=6°. Ramp type filter is used.

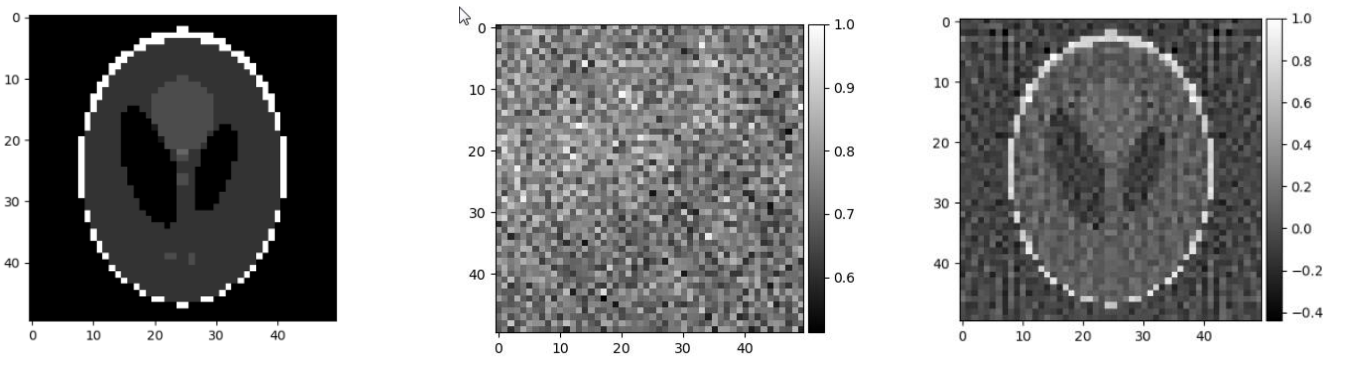
Comment-1:

The image reconstructed without filter is severely blurred because low frequency content of the image gets amplified while the high frequency content gets diminish by being weighted non-uniformly (Actually still it should have had resemblance to the original image but an unknown reason causes this condition, possibly due to a mistake in the code). The effect of the filter is the compensation the non-uniform weighting in the 2D Fourier space. We use basically a high-pass filter.

To quantitatively compare the reconstructed image with the original image, we can use mean squared error approach (MSE). The closer the MSE to zero, the better the image is reconstructed.

MSE values for filtered and unfiltered cases are as follows, respectively: 0.063, 0.712

Output 2:



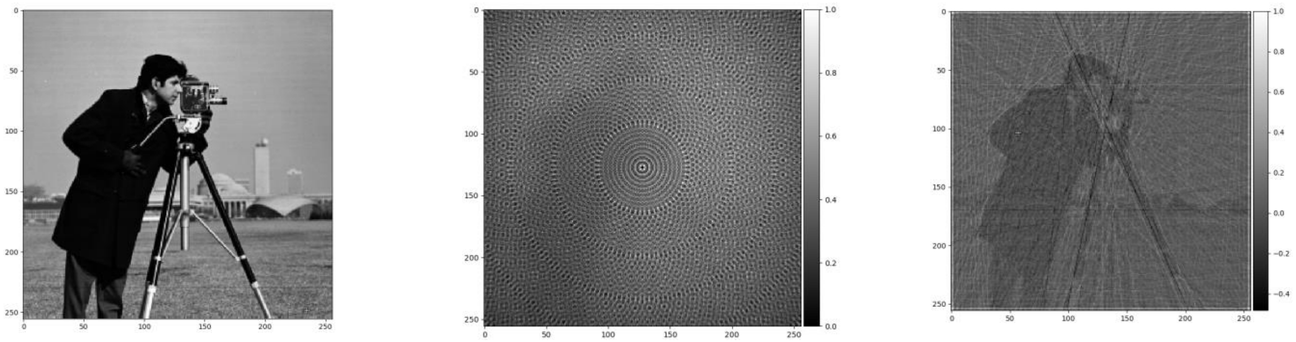
(a) (b) (c)

Figure 4: (a) The Shepp - Logan image (b) Reconstructed image without filtering (c) Reconstructed image with filtering when the number of beams = 100 and step size=6°. Ramp type filter is used.

Comment-2:

MSE values of the filtered and unfiltered back-projections are the following, respectively: 0.084 and 1.88.

Output 3:



(a) (b) (c)

Figure 5: (a) The cameraman image (b) Reconstructed image without filtering (c) Reconstructed image with filtering when the number of beams=100 and step size=6°. Ramp type filter is used.

Comment-3:

MSE values of the filtered and unfiltered back-projections are the following, respectively: 0.28, 0.28. Due to an unknown reason, MSE did not give an acceptable comparison between the original image and the reconstructed image in this case.

Appendix

The code:

```
import PySimpleGUI as sg
import sys,time,pickle
from mplcursors import cursor
import scipy.io as sio
from scipy import signal
import numpy as np
import matplotlib.pyplot as plt
from numpy.fft import fft2,ifft2
from mpl_toolkits.axes_grid1 import make_axes_locatable
sg.change_look_and_feel('DefaultNoMoreNagging')
layout = [                                            # Here's for the GUI window
        [sg.Text('Choose where you get the projection data from:')],
  [sg.Radio('From text file     ', "RADIO2"), sg.Radio('From mat file     ', "RADIO2"),
        sg.Radio('Do new projection         ', "RADIO2", default=True)],
        [sg.Text('Enter the number of beams:')],
        [sg.InputText('100')],
        [sg.Text('Enter the step size:')],
        [sg.InputText('30')],
        [sg.Text('kare_kosede_50ye50.mat is the default')],
        [sg.Listbox(values=['cameraman_256_256.mat','bird_472_472.mat',                      'lena_256ya256.mat',
'horse_400_400.mat', 'Shepp-Logan.mat'],
        default_values=['kare_kosede_50ye50.mat'], size=(30, 5))],
        [sg.Text('Choose filter type:')],
        [sg.Radio('Ramp     ', "RADIO3", default=True), sg.Radio('Hanning     ', "RADIO3"),
        sg.Radio('Cosine     ', "RADIO3"), sg.Radio('No filter     ', "RADIO3")],
        [sg.Checkbox('Do only projection', default=False),sg.Text(' '*15+'Enter the projection angle:'),
```

```
        sg.InputText(size=(5,1))],#sg.Checkbox('Show Error Image')],
        # [sg.Checkbox('Both filtered and none-filtered versions', default=True)],
        [sg.Submit(), sg.Cancel()]]
window  = sg.Window('Projection GUI', auto_size_text=True, default_element_size=(40, 1)).Layout(layout)
while True:
        event, values = window.Read()
        if event == 'Submit':
                break
        elif event == 'Cancel':
                sys.exit()

if event == 'Submit':
        window.Close()
elif event == 'Cancel':
        sys.exit()
pi = np.pi
if values[6] == True:
        filter = 6
        filter_name = 'Ramp Filter'
else:
        if values[7] == True:
                filter = 7
                filter_name = 'Hanning Filter'
        elif values[8] == True:
                filter = 8
                filter_name = 'Cosine Filter'
        elif values[9] == True:
                filter = 0
                filter_name = 'No Filter'
def project():
        pro_bas = time.time()
        y_values = x_values = np.arange(-size/2, size/2+1)                              # determine x & y values
on the image
        t = np.linspace(-size/pow(2,1/2), size/pow(2,1/2),number_of_beams)
        carp = size * np.sqrt(2)
        karsi_uz = np.where(teta <= 90,carp * np.cos((45-teta) * pi/180),carp * np.cos((135-teta) * pi/180))
        # 5. step: Find all intersection points for all beams for all projection angles using line equation:
        result=[]
        for aci in teta_degree:
                tan = np.tan(aci)
                cos = np.cos(aci)
                for t_degeri in t:
                        for x_degeri in x_values:
                                resulted_y_values = tan * x_degeri + t_degeri / cos #line equation
                                result.append([aci,t_degeri,x_degeri,resulted_y_values])
        for aci in teta_degree:
                cos = np.cos(aci)
                sin = np.sin(aci)
                for t_degeri in t:
                        for y_degeri in y_values:
                                if aci==0 and y_degeri==t_degeri:                                    # in case of
0 in the denominator
                                        for x_degeri in x_values:
                                                result.append([aci,t_degeri,x_degeri,y_degeri])
                                                # np.where(aci==0 and y_values == t_degeri,)
```

```
                                elif aci != 0:
                                        resulted_x_values = (y_degeri * cos - t_degeri)/sin # line equation
                                        result.append([aci,t_degeri,resulted_x_values,y_degeri])
        # Remove the repeated points:
        final_result = [list(t) for t in set(tuple(np.round(element,1)) for element in result)]
        son = []
        # 6. Step: Remove the points which are irrelevant to the object:
        # Bu işlemle irrelevant noktaları attığımız için mesela 0 derece t=sqrt(-2) noktaları gitti
        bakk = time.time()
        for element in final_result:                     # 6.5 saniye
                if ((element[2]) <= (x_values[-1]) and (element[2]) >= (x_values[0]) and (element[3]) <= (y_values[-
1]) and (element[3]) >= (y_values[0])):
                        son.append(element)
        son=sorted(son)                                                         # 7. Step: Sort the relevant points

# Below, I grouped the elements of 'son' variable with respect to their angle and t values while it had one row only
before this işlem
        temp_aci_t_degeri = son[0][0:2]
        alt_liste=[son[0]]
        son_son=[]
        for i in son[1:]:
                if i[0:2] == temp_aci_t_degeri:
                        alt_liste.append(i)
                        temp_aci_t_degeri = i[0:2]
                else:
                        son_son.append(alt_liste)
                        alt_liste = []
                        alt_liste.append(i)
                        temp_aci_t_degeri = i[0:2]
        son_son.append(alt_liste)
        # 8. Find the midpoint and the length of line segments:
        midX=[]
        midY=[]
        distance_son_son=[]
        for i in son_son:
                temp=i[0]
                distance=[]
                for j in i[1:]:
                        temp_midX=((j[2]+temp[2])/2)
                        temp_midY=((j[3]+temp[3])/2)
                        dist_temp = pow((j[2]-temp[2])*(j[2]-temp[2])+(j[3]-temp[3])*(j[3]-temp[3]),1/2)
                        # dist_temp = abs( j[2]-temp[2] + (j[3]-temp[3])*1j )
                        midX.append(temp_midX)
                        midY.append(temp_midY)
                        distance.append(dist_temp)
                        temp = j
                distance_son_son.append(distance)

        # 9. Detect the address (row and column data) by using the midpoint data.
        rowdata = (np.ceil(size/2 - np.floor(midY))-1)
        columndata = (np.ceil(size/2 - np.floor(midX))-1)
        # 10. Sum all pixel value and distance products
        say = 0
        projection = []
        for i in distance_son_son:
```

```
            toplam=0
            for j in i:
                toplam += (j * img[int(rowdata[say])][int(columndata[say])])
                say=say+1
            projection.append(toplam)
        grup = []
        sa = 0
        for te in teta:
            if ( int(te) == 45 or int(te) == 135):
                grup.append(number_of_beams)
            else:
                k = 0
                for i in range(len(t)):
                    if abs(t[i]) > karsi_uz[sa]/2:
                        k+=1
                    else:
                        break
                grup.append(number_of_beams-k*2)
            sa += 1
        # print(grup)
        # açılara göre gruplu projection:
        Inputt = iter(projection)
        son_projection_with_zeros = [list(__import__('itertools').islice(Inputt, elem)) for elem in grup]
        # açılara göre gruplu distance:
        Inputt = iter(distance_son_son)
        son_distance_with_zeros = [list(__import__('itertools').islice(Inputt, elem)) for elem in grup]
        # pad the projection with 0s which occur when the teta values other than 45 and 90 degrees
        grup_say=0
        for pro in son_projection_with_zeros:
            if (len(pro) < number_of_beams):
                for i in range(int((number_of_beams - grup[grup_say])/2)):
                    pro.insert(0,0)
                    pro.insert(len(pro),0)
            grup_say+=1
        grup_say=0
        for pro in son_distance_with_zeros:
            if (len(pro) < number_of_beams):
                for i in range(int((number_of_beams - grup[grup_say])/2)):
                    pro.insert(0,0)
                    pro.insert(len(pro),0)
            grup_say+=1
        with open('projection_data.txt','w') as dosya_txt:
            dosya_txt.write(str(number_of_projections)+'\n'+str(number_of_sampling_points)+'\n')
            for k in range(len(son_projection_with_zeros)):
                dosya_txt.write(str(k+1)+'\n')
                for j in son_projection_with_zeros[k]:
                    dosya_txt.write(str(j)+'\n')
        mat_array=np.array(son_projection_with_zeros)          #list to ndarray conversion
        column_array=np.array(columndata)
        row_array=np.array(rowdata)
        with open('distance_list.obj','wb') as dist:
            pickle.dump(son_distance_with_zeros,dist)
        sio.savemat(values[5][0][:-4]+'_projection_data.mat', mdict={   'projection':
mat_array,'columndata':column_array,
```

```
        'rowdata':row_array,'size':size, 'original':img     })
        print('projection time: ',time.time() - pro_bas)
        if values[10] == True:                                                    # If we do projection only
                plot_projection(t,son_projection_with_zeros,number_of_sampling_points,step_size)
        return son_projection_with_zeros,son_distance_with_zeros,rowdata,columndata
def plot_projection(t,projection,number_of_sampling_points,step_size):
        if values[11] == '':
                fig, axs = plt.subplots(2,3)
                sayyy = 0
                for i in axs.flatten():
                        i.plot(t.round(2),projection[sayyy])
                        sayyy += 1
                plt.suptitle('Projections        for        '+'\nNumber        of        sampling        points:
'+str(number_of_sampling_points)+'\n'+' Step size: '+str(step_size))
                plt.figure()
                plt.imshow(img,cmap='gray')
                plt.title('Original image')
        else:
                cizdirilecek_aci = float(values[11])
                cizdirilecek_acinin_indexi = np.where(teta==cizdirilecek_aci)[0][0]
                fig, axes = plt.subplots(1,2)
                axes[1].plot(t.round(2),projection[cizdirilecek_acinin_indexi],'ro')
                axes[1].set_xlabel('The beam which going through the image (t coordinates)')
                axes[1].set_ylabel('Projection value for the beams')
                axes[1].set_title('Projections  for  '+str(cizdirilecek_aci)+'$^\circ$'+'\nNumber  of  sampling  points:
'+str(number_of_sampling_points)+'\n'+' Step size: '+str(step_size))
                axes[0].set_title('Original')
                axes[0].imshow(img,cmap='gray')
        cursor(multiple=True)
        plt.subplots_adjust(left=0.125, bottom=0.1, right=0.9 , top=0.85 , wspace=0.4, hspace=0.2)
        plt.show()
def ramp_filter():
        fft_of_projection = fft2(image_to_be_reconstructed)
        t = np.linspace(0, 1, number_of_sampling_points)
        high_pass_filter = abs(abs(signal.sawtooth(2 * pi * t))-1)
        filtered_fft_of_projection = fft_of_projection * high_pass_filter
        ifft_of_projection = ifft2(filtered_fft_of_projection)
        return ifft_of_projection.tolist()
def hanning_filter():
        fft_of_projection = fft2(image_to_be_reconstructed)
        high_pass_filter = np.hanning(number_of_sampling_points)
        filtered_fft_of_projection = fft_of_projection * high_pass_filter
        ifft_of_projection = ifft2(filtered_fft_of_projection)
        return ifft_of_projection.tolist()
def hamming_filter():
        fft_of_projection = fft2(image_to_be_reconstructed)
        high_pass_filter = np.hamming(number_of_sampling_points)
        filtered_fft_of_projection = fft_of_projection * high_pass_filter
        ifft_of_projection = ifft2(filtered_fft_of_projection)
        return ifft_of_projection.tolist()
def cosine_filter():
        fft_of_projection = fft2(image_to_be_reconstructed)
        high_pass_filter = signal.cosine(number_of_sampling_points)
        filtered_fft_of_projection = fft_of_projection * high_pass_filter
```

```
        ifft_of_projection = ifft2(filtered_fft_of_projection)
        return ifft_of_projection.tolist()
def back_projection(getir=None):
        back_pro_bas = time.time()
        if getir == None:
                getir = image_to_be_reconstructed
        # Multiply the filtered projection data with the distance:
        netice = []
        for i in getir:
                o = []
                for k in i:
                        o.append(k * np.array(distance[getir.index(i)][i.index(k)]))
                netice.append(o)
        kl=np.array([1.6024768-0.52718694j, 1.6024768-0.52718694j])
        tur = type(kl)
        son_netice = []
        for i in netice:
                ara_netice=[]
                for k in i:
                        if type(k) == tur:
                                ara_netice.append(k.tolist())
                        else:
                                ara_netice.append(k)
                son_netice.append(ara_netice)

        img_back = np.zeros((size,size))
        say = 0
        for i in son_netice:
                for j in i:
                        if not j == 0:
                                for k in j:
                                        img_back[int(rowdata[say])][int(columndata[say])] += k.real
                                        say += 1
        max_img = np.amax(img_back)
        img_normalized = img_back / max_img
        error_img = img - img_normalized                    # if you want error_img to be included,
uncomment related parts
        img_normalized_er = error_img / np.amax(error_img)
        av_err = np.mean(img_normalized_er)
        mse = np.mean(np.square(img_normalized_er))
        print('back projection time: ',time.time() - back_pro_bas)
        print('av_err :',av_err)
        print('mse :',mse)
        fig,(original,back) = plt.subplots(1,2) #,error)
        plt.subplots_adjust(left=0.125, bottom=0.1, right=0.9 , top=0.9 , wspace=0.4, hspace=0.2)
        original.imshow(img,cmap='gray')
        # im_err = error.imshow(img_normalized_er,cmap='gray')                    # error related,
comment/uncomment
        im_back = back.imshow(img_normalized,cmap='gray')
        divider_b = make_axes_locatable(back)
        # divider_e = make_axes_locatable(error)                                  # error related,
comment/uncomment
        cax1 = divider_b.append_axes("right", size="5%", pad=0.05)
        # cax2 = divider_e.append_axes("right", size="5%", pad=0.05)   # error related, comment/uncomment
```

```
        # original.set_title('Original image')                                   # error related,
comment/uncomment
        # back.set_title('Back projected image')                              # error related,
comment/uncomment
        # error.set_title('Error')                                                   # error
related, comment/uncomment
        fig.colorbar(im_back,cax=cax1)
        # fig.colorbar(im_err,cax=cax2)                                              # error
related, comment/uncomment
        fig_name = "number_of_sampling_points: "+str(number_of_sampling_points)+" step_size:
"+str(step_size)+" "+filter_name+".png"
        # plt.savefig(fig_name)                    # anlamadım hatayı
        plt.suptitle('number_of_sampling_points: '+str(number_of_sampling_points)+'\n'+' step_size:
'+str(step_size)+'\n'+filter_name)
        plt.show()

if values[2] == True:
        # If "Do new projection" is chosen
        if values[5] == []:
                mat = sio.loadmat('kare_kosede_50ye50.mat')                            # 1. step:
        load the default image
                values[5] = ['kare_kosede_50ye50.mat']
        else:
                        # or other image
                mat = sio.loadmat(values[5][0])
        img = list(mat.values())[3]#:][0]
        # img1=np.zeros((50,50))
        # img1[29:40,9:20] = img[9:20,9:20]
        # img1[9:20,29:40] = img[9:20,9:20]
        # img = img1
        size = img.shape[0]
                # 2. step:      determine the size of the image
        number_of_sampling_points = number_of_beams = int(values[3])          # 3. step:     get
number of beams
        step_size = float(values[4])                                                   #
                        get step_size
        teta = np.arange(0,180,step_size)                                              #
specify angle values according to the step size
        teta_degree = teta*pi/180
        number_of_projections = teta_adedi = teta.shape[0]
        if values[10] == True:
        # Do only projection
                project()
        else:
                image_to_be_reconstructed,distance,rowdata,columndata = project()
                if filter == 6:
                        back_projection(ramp_filter())
                elif filter == 7:
                        back_projection(hanning_filter())
                elif filter == 8:
                        back_projection(cosine_filter())
                elif filter == 0:                                              # no filter
                        back_projection()
```

```
else:                                                                                    #
Use ready projection data (txt or mat)
    if values[0] == True:                                              # from txt
        with open('projection_data.txt') as dosya_txt:
            # data_from_txt = dosya_txt.read()
            lines_from_txt = dosya_txt.readlines()
            number_of_projections = int(lines_from_txt[0])
            number_of_sampling_points = int(lines_from_txt[1])

        image_to_be_reconstructed = image_to_be_reconstructed.tolist()
        step_size = 180/number_of_projections
        size = mat_liste[6][0][0]
        columndata = mat_liste[4].tolist()[0]
        # print(type(columndata))
        # print('size: ',columndata)
        rowdata = mat_liste[5].tolist()[0]
        # distance_arr = mat_liste[6]
        img = mat_liste[7]
        with open('distance_list.obj','rb') as dist:
            distance = pickle.load(dist)
    elif values[1] == True:                                        # from mat
        if values[5] == []:
            values[5] = ['kare_kosede_50ye50.mat']
        mat = sio.loadmat(values[5][0][:-4]+'_projection_data.mat')
        mat_liste = list(mat.values())
        image_to_be_reconstructed = mat_liste[3]
        number_of_projections = image_to_be_reconstructed.shape[0]
        number_of_sampling_points = number_of_beams = image_to_be_reconstructed.shape[1]
        image_to_be_reconstructed = image_to_be_reconstructed.tolist()
        step_size = 180/number_of_projections
        size = mat_liste[6][0][0]
        columndata = mat_liste[4].tolist()[0]
        rowdata = mat_liste[5].tolist()[0]
        # distance_arr = mat_liste[6]
        img = mat_liste[7]
        with open('distance_list.obj','rb') as dist:
            distance = pickle.load(dist)
    if filter == 6:
        back_projection(ramp_filter())
    elif filter == 7:
        back_projection(hanning_filter())
    elif filter == 8:
        back_projection(cosine_filter())
    elif filter == 0:                                  # no filter
        back_projection()
```