**Emre Ataklı - 2112944 24.12.2019**

Duration to complete Step-2: 10 days

**EE415 Step 2 - Inverse Problem**

# Algorithm

General structure of the algorithm:

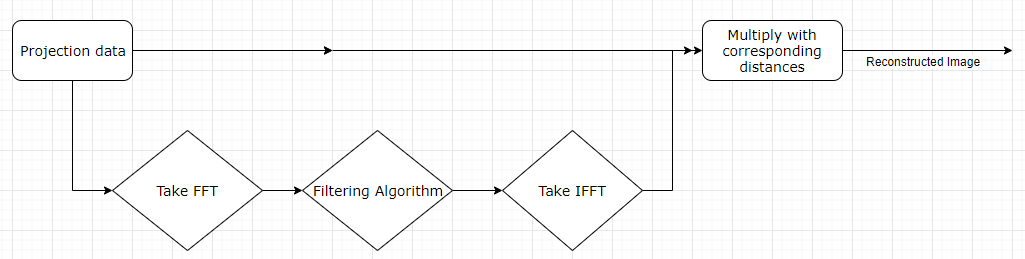
****

Figure : 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()

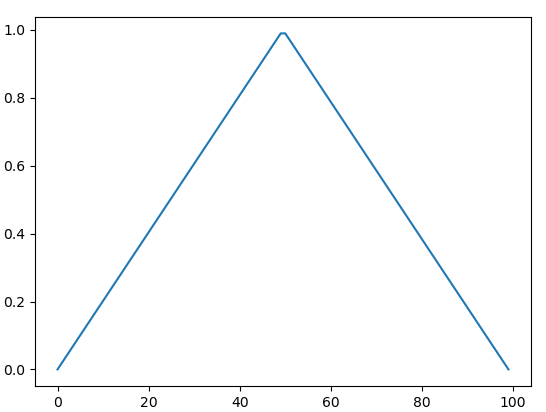


Figure : Implemented ramp filter.

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=. 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=. 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=. 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()*