Submission for Programming Assignment 2

CS 427: Mathematics for Data Science, Autumn 2020-21

Group 5: K. Sai Anuroop (170030035), Anubola Sai Abhinay (180010006)

December 16, 2020

Solution 1

We are given $s \in \mathbb{R}^n$ which is a sparse representation of $x \in \mathbb{R}^n$, $y \in \mathbb{R}^q$ $(q < n)$ which is obtained by a linear transformation on x, and $C \in \mathbb{R}^{q \times n}$ which is some transformation matrix. We are required to explain why $\min_s \|s\|_0$, such that $\|y - Cs\|_2^2 = 0$ is not a convex optimisation problem.
Too see why, we now show that l_0 norm, which is the objective of our optimisation problem, is not convex.
Consider the following simple counterexample in $\mathbb{R}^2$. Let $s_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $s_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Then for any $\theta \in (0, 1)$,

$$\|\theta s_1 + (1-\theta)s_2\|_0 = \left\| \begin{pmatrix} \theta \\ 1-\theta \end{pmatrix} \right\|_0 = 2$$

$$\theta\|s_1\|_0 + (1-\theta)\|s_2\|_0 = \theta \left\| \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\|_0 + (1-\theta) \left\| \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\|_0 = \theta(1) + (1-\theta)(1) = 1$$

$$\Rightarrow \|\theta s_1 + (1-\theta)s_2\|_0 > \theta\|s_1\|_0 + (1-\theta)\|s_2\|_0$$

Therefore l_0 norm is not convex.
Since the objective is not convex, the optimisation problem is not convex. □

Solution 2

We first show that $\min_s \|s\|_1$, such that $\|y - Cs\|_2^2 = 0$ is a convex relaxation of the previous problem.
We make use of the following lemma:
The l_1 norm ball is the convex hull of the intersection between the l_0 norm ball and the l_∞ norm ball.
Further, if we restrict $\|s\|_\infty \leq 1$, then l_1 norm is the tightest convex relaxation of l_0 norm, from the lemma.
Therefore, as we have replaced the original objective $\|s\|_0$ with $\|s\|_1$ in the optimisation problem, and since l_1 norm is a convex relaxation of l_0 norm, the re-formulated optimisation problem is a convex relaxation of the original optimisation problem.

We next note that the Lagrangian function of the optimisation problem is

$$L(s, \nu) = \|s\|_1 + \nu^T \|y - Cs\|_2^2$$

However, from the definition of any p norm, we know that $\|\alpha\| = 0$ if and only if $\alpha = 0$. So, we have the following if and only conditions:

$$\|y - Cs\|_2^2 = 0 \iff \|y - Cs\|_2 = 0$$

$$\|y - Cs\|_2 = 0 \iff y - Cs = 0$$

$$y - Cs = 0 \iff y = Cs$$

We therefore replace the norm-squared constraint with the affine constraint, i.e., our optimisation problem is now recast as $\min_s \|s\|_1$, such that $Cs = y$.
The Lagrangian of the recast optimisation, denoted by $\widetilde{L}$, problem is

$$\widetilde{L}(s, \nu) = \|s\|_1 + \nu^T(y - Cs)$$

The Lagrange dual function of the recast optimisation problem is given by

$$g(\nu) = \inf_{s \in \mathbb{R}^n} \widetilde{L}(s, \nu)$$

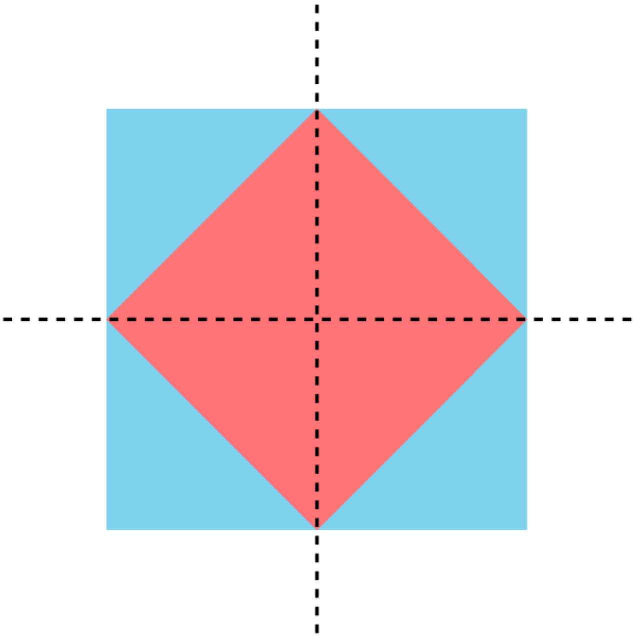

Figure 1: Considering $\mathbb{R}^2$, the l_0 norm ball is shown in dotted-black lines, the l_∞ norm ball in blue and the l_1 norm ball in red.

$$= \inf_{s\in\mathbb{R}^n} \left(\|s\|_1 + \nu^T(y - Cs) \right)$$

$$= \inf_{s\in\mathbb{R}^n} \left(\|s\|_1 - (C^T\nu)^T s + \nu^T y \right)$$

Now, if $(C^T\nu)_i > 1 \; \forall \; i \in \{1, \cdots, n\}$, we can set s_i arbitrarily large ($\forall \; i \in \{1, \cdots, n\}$) so that $g(\nu) \to -\infty$. Also, observe that the same happens if $(C^T\nu)_i < 1 \; \forall \; i \in \{1, \cdots, n\}$. Now if $\left\|C^T\nu\right\|_\infty \leq 1$, then by Hölder's inequality we get

$$\left|(C^T\nu)^T s\right| \leq \|s\|_1 \left\|C^T\nu\right\|_\infty \leq \|s\|_1$$

So, the Lagrangian is minimised by setting $s = 0$ and the dual function thus is $g(\nu) = \nu^T y$. The dual problem therefore is, $\max_{\nu\in\mathbb{R}^q} \nu^T y$ such that $\left\|C^T\nu\right\|_\infty \leq 1$.

We next note Slater's theorem, which states that strong duality holds if Slater's condition holds. We further observe that for an optimisation problem with a convex objective, Slater's condition reduces to feasibility when the constraints are all linear equalities and inequalities, and domain of the objective function is open. In the current case too, we see that the constraints are all linear equalities. So by Slater's theorem, strong duality holds since $y \in \mathbf{col}C$ where **col** refers to the column space of a matrix. $\square$

Solution 3

```python
"""
This code recovers an image x, given y, an incomplete measurement of x; C, a transformation matrix; and A_inv, inverse of the sensing matrix A
For achieving this, it solves an optimisation problem:
minimise(1-norm of s) such that 2-norm-squared(y-Cs)=0
Since the norm-squared constraint can be recast as an affine constraint, we provide the affine equality as our constraint instead.
"""

# import libraries
import cvxpy as cp # this is convex optimisation library
import numpy as np
import matplotlib.pyplot as plt

# loading the required matrices and vectors
C = np.load("./C.npy") # load the transformation matrix
A_inv = np.load("./A_inv.npy") # load the inverse of the sensing matrix A
y = np.load("./y.npy") # load the incomplete measurement of x
s = cp.Variable(10000) # declaring the optimising variable

# optimisation

objective = cp.Minimize(cp.norm(s, 1)) # declaring the objective as minimisation of l1-norm of s
constraints = [C@s == y.reshape(len(y),)] # declaring the constraint as an affine equality
prob = cp.Problem(objective, constraints) # declaring the optimisation problem

# solving the optimisation problem using ECOS solver
obj = prob.solve(verbose=True, solver=cp.ECOS, max_iters=50, abstol=1e-10, reltol=1e-10, feastol=1e-10)

# reconstruction
recon_img = A_inv@s.value # reconstruct the original image by multiplying the s value with the sensing matrix

# save numpy arrays so that they can be worked upon in future
np.save('./s_value',s.value)
np.save('./recon_img_array',recon_img)

# plot the reconstructed image
plt.imshow(recon_img.reshape(100,100).T, cmap='gray')
plt.axis('off') # not printing axis to focus only on the image
plt.savefig('./recon_gray'+'.png')
plt.close()

plt.imshow(recon_img.reshape(100,100).T) # cmap is removed to prevent the mapping of colours to grayscale
plt.axis('off') # not printing axis to focus only on the image
plt.savefig('./recon_colour'+'.png')
plt.close()
```

compressed_sensing.py script to reconstruct the image

Solution 4

During reconstruction, we used the ECOS solver provided by the `cvxpy` library for convex optimisation with the following parameters: `max_iters=50`, `abstol=1e10`, `reltol=1e10`, `feastol=1e10`. Reconstruction took about 30 minutes to complete, stopping after the 25^{th} iteration.
We see that the reconstructed image, though barely discernible, gives an image of an elephant, probably with its calf on the left.
Note: The images corresponding to a result are placed on a separate page in this document, after the commentary, for clarity.

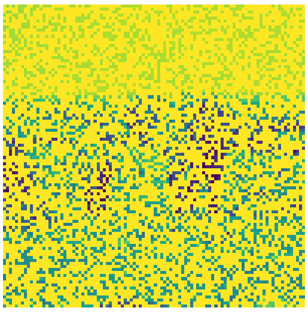

Figure 2: Masked image

Figure 3: Reconstructed image (grayscale)

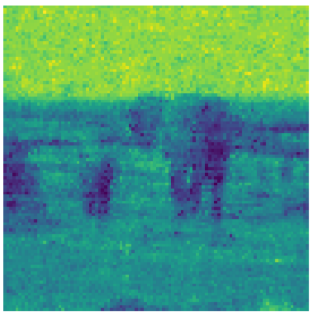

Figure 4: Reconstructed image

Solution 5

We did the following:

- Took some coloured images on the internet.
- Using the link provided in the assignment, resized those pictures to 100×100 pixels.
- Modified the `create_data_for_assignment.py` to create a new script named `create_data_for_assignment_colour.py` which performed the following.
  1. Read the resized coloured image.
  2. Split the image into the constituent Red, Green and Blue channels.
  3. For each image of the constituent channel, corrupt it and output the files required for performing optimisation.
- Modified the `compressed_sensing.py` script to perform optimisation on each of the three different constituents of the coloured image obtained in the above step.
- Created `mix_channels.py` to combine the reconstructed images of the three different constituents of the coloured image obtained in the above step, into a single coloured image.

We have modularised the code into three different scripts because doing so made it easier for us to work and tinker with the output files produced by each of these scripts. In principle, we could have clubbed all of these scripts into a single one, but we chose not to do so for the reason stated earlier and due to the fact that modularising the code is a good practice to follow.
The masks were created by retaining only the randomly sampled pixels in the original image and setting the other pixels to 255. We ensured that the pixels that were randomly sampled remained the same across each of the three channels, otherwise it would lead to an unfair reconstruction due to the fact that each channel has provides a different information in that case. The code for creating the output files used for reconstruction remains unchanged.
We now present the results of this section.

Pandemonium of parrots

We have corrupted the image using the default corruption rate of 0.7 and zoom-out value of 0.9999999. During reconstruction, we used the ECOS solver provided by the `cvxpy` library for convex optimisation with the following parameters: `max_iters=30`, `abstol=1e6`, `reltol=1e6`, `feastol=1e6`. Reconstruction took about 1 hour 15 minutes to complete.
We observe here that the original image contains many colours (we have chosen such an image that has many colours so as to check the efficacy of our reconstruction algorithm!). The red channel shows good contrast along with the green channel, however the blue channel offers little contrast.
The reconstructed image, though blurry, is sufficient for the naked eye to discriminate between the three parrots in the image.

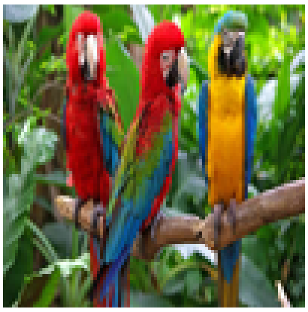

Figure 5: Original 100 × 100 coloured image

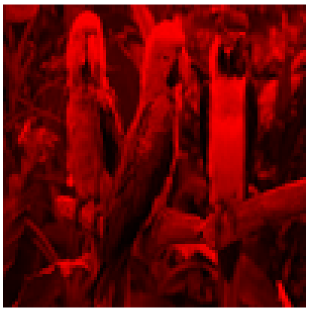

Figure 6: Red channel of the original 100 × 100 coloured image

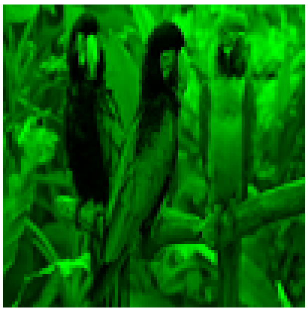

Figure 7: Green channel of the original 100 × 100 coloured image

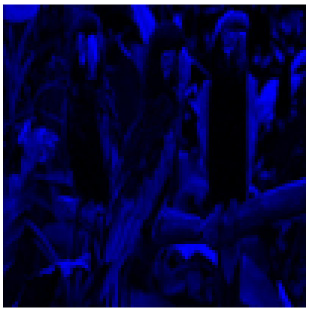

Figure 8: Blue channel of the original 100 × 100 coloured image

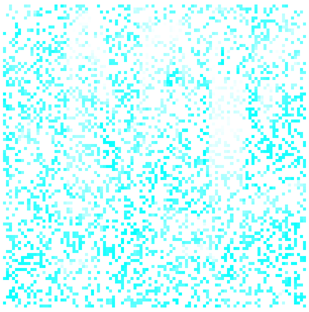

Figure 9: Mask of the red channel of the original 100 × 100 coloured image

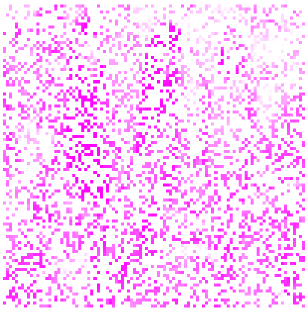

Figure 10: Mask of the green channel of the original 100 × 100 coloured image

Figure 11: Mask of the blue channel of the original 100 × 100 coloured image

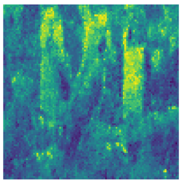

Figure 12: Reconstruction of the red channel of the original 100 × 100 coloured image

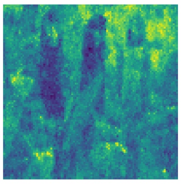

Figure 13: Reconstruction of the green channel of the original 100 × 100 coloured image

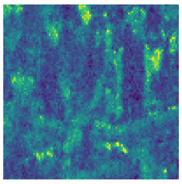

Figure 14: Reconstruction of the blue channel of the original 100 × 100 coloured image

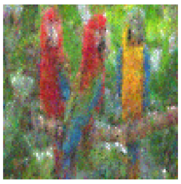

Figure 15: Complete reconstruction of the original 100 × 100 coloured image

Audi car

We have corrupted the image using the default corruption rate of 0.7 and zoom-out value of 0.9999999. During reconstruction, we used the ECOS solver provided by the `cvxpy` library for convex optimisation with the following parameters: `max_iters=30`, `abstol=1e6`, `reltol=1e6`, `feastol=1e6`. Reconstruction took about 1 hour 10 minutes to complete.
We observe here that the original image contains predominantly yellow colour, whose various hues are obtained by mixing of red, green and blue colours in various proportions. This image thus is a good test for our reconstruction algorithm.
It is seen from the individual channel reconstructions that the red and green constituents offer a greater amount of information when compared to the blue constituent. The reconstructed image approximates the original image quite well. It also preserves the shiny gloss which can be seen on the car in the original image!

Figure 16: Original 100×100 coloured image

Figure 17: Red channel of the original 100×100 coloured image

Figure 18: Green channel of the original 100×100 coloured image

Figure 19: Blue channel of the original 100×100 coloured image

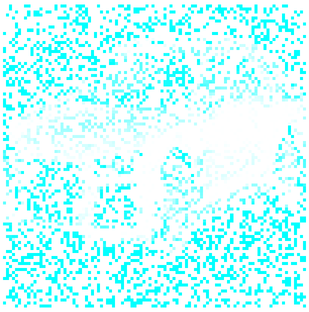

Figure 20: Mask of the red channel of the original 100×100 coloured image

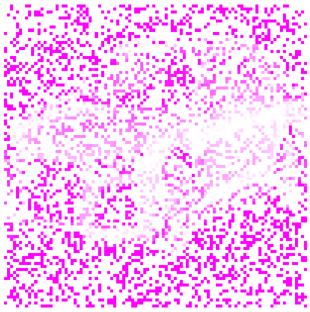

Figure 21: Mask of the green channel of the original 100×100 coloured image

Figure 22: Mask of the blue channel of the original 100×100 coloured image

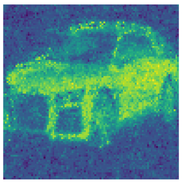

Figure 23: Reconstruction of the red channel of the original 100×100 coloured image

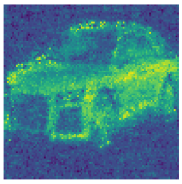

Figure 24: Reconstruction of the green channel of the original 100×100 coloured image

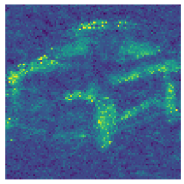

Figure 25: Reconstruction of the blue channel of the original 100×100 coloured image

Figure 26: Complete reconstruction of the original 100×100 coloured image

Image of a person

We have corrupted the image using a corruption rate of 0.6 and zoom-out value of 0.9999999. During reconstruction, we used the ECOS solver provided by the `cvxpy` library for convex optimisation with the following parameters: `max_iters=30`, `abstol=1e6`, `reltol=1e6`, `feastol=1e6`. Reconstruction took about 1 hour 10 minutes to complete.
We observe that the image is of a grinning man in his early twenties who wears a bright red t-shirt. We chose this image as we wanted to test our reconstruction algorithm on images containing the faces of people, and see if the face is discernible at the given corruption rate and tolerances in our optimisation code.
It is observed that the face of the person is discernible. This means that the reconstruction algorithm does well even on those images concerning faces of the people.

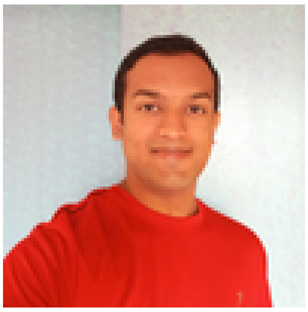

Figure 27: Original 100×100 coloured image

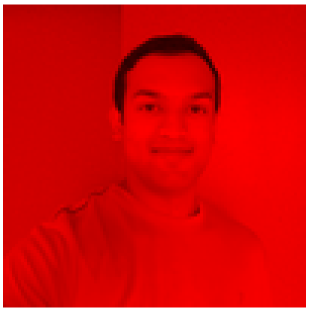

Figure 28: Red channel of the original 100×100 coloured image

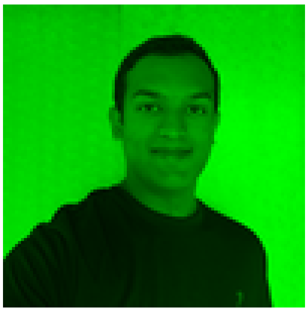

Figure 29: Green channel of the original 100×100 coloured image

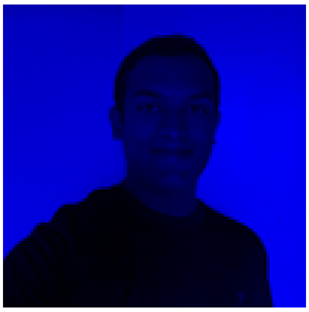

Figure 30: Blue channel of the original 100×100 coloured image

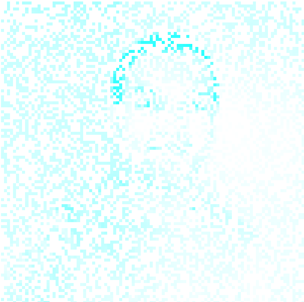

Figure 31: Mask of the red channel of the original 100×100 coloured image

Figure 32: Mask of the green channel of the original 100×100 coloured image

Figure 33: Mask of the blue channel of the original 100×100 coloured image

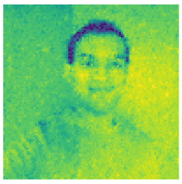

Figure 34: Reconstruction of the red channel of the original 100×100 coloured image

Figure 35: Reconstruction of the green channel of the original 100×100 coloured image

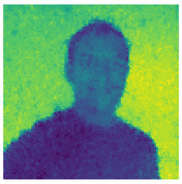

Figure 36: Reconstruction of the blue channel of the original 100×100 coloured image

Figure 37: Complete reconstruction of the original 100×100 coloured image

```python
"""
This code recovers an image x, given y, an incomplete measurement of x; C, a transformation matrix; and A_inv, inverse of the sensing matrix A
For achieving this, it solves an optimisation problem:
minimise(1-norm of s) such that 2-norm-squared(y-Cs)=0
Since the norm-squared constraint can be recast as an affine constraint, we provide the affine equality as our constraint instead.
The optimisation is run for the three constituent channels of Red, Green and Blue. So the corresponding input files are expected by this code.
"""

# import libraries
import cvxpy as cp # this is convex optimisation library
import numpy as np
import matplotlib.pyplot as plt

# run the optimisation for each constituent channel: Red, Green, Blue
for i in range(0,3):

  # loading the required matrices and vectors
  C = np.load("./C"+str(i)+".npy") # load the transformation matrix
  A_inv = np.load("./A_inv"+str(i)+".npy") # load the inverse of the sensing matrix A
  y = np.load("./y"+str(i)+".npy") # load the incomplete measurement of x
  s = cp.Variable(10000) # declaring the optimising variable

  # optimisation
  objective = cp.Minimize(cp.norm(s, 1)) # declaring the objective as minimisation of l1-norm of s
  constraints = [C@s == y.reshape(len(y),)] # declaring the constraint as an affine equality
  prob = cp.Problem(objective, constraints) # declaring the optimisation problem

  # solving the optimisation problem using ECOS solver
  obj = prob.solve(verbose=True, solver=cp.ECOS, max_iters=30, abstol=1e-6, reltol=1e-6, feastol=1e-6)

  # reconstruction
  recon_img = A_inv@s.value # reconstruct the original image by multiplying the s value with the sensing matrix

  # save numpy arrays so that they can be worked upon in future
  np.save('./s_value'+str(i),s.value)
  np.save('./recon_img_array'+str(i),recon_img)

  # plot the reconstructed image
  plt.imshow(recon_img.reshape(100,100).T)
  plt.axis('off') # not printing axis to focus only on the image
  plt.savefig('./recon'+str(i)+'.png')
  plt.close()
```

Modified `compressed_sensing.py` script to reconstruct each constituent channel of the image

```python
# import libraries
import numpy as np
import matplotlib.pyplot as plt
import imageio

# load the image arrays corresponding to each of three constituent channels
r = np.load("./recon_img_array0.npy")
g = np.load("./recon_img_array1.npy")
b = np.load("./recon_img_array2.npy")

# reshape the flat arrays into a 100 x 100 matrix
r = r.reshape(100,100).T
g = g.reshape(100,100).T
b = b.reshape(100,100).T

# stack different channels into a single image
mix = np.dstack((r,g,b))

# perform normalisation of pixel values
mix = abs((mix-mix.min())/(mix.max()-mix.min()))

# plot the reconstructed image
plt.imshow(mix)
plt.axis('off')
plt.savefig("recon_color.png")
plt.close()

```

`mix_channels.py` script to combine the reconstructed images of the three different constituents of the coloured image, into a single coloured image

```python
"""
This is a modified version of create_data_for_assignment.py script provided, which does the following:
* Read the resized coloured image.
* Split the image into the constituent Red, Green and Blue channels.
* For each image of the constituent channel, corrupt it and output the files required for performing optimisation.
"""
""" changes to the original code are indicated by ----> """
import numpy as np
import matplotlib.pyplot as plt
import scipy.fftpack as spfft
import scipy.ndimage as spimg
import imageio
#%% Discrete Cosine Transform
def dct2(x):
    return spfft.dct(spfft.dct(x.T, norm='ortho', axis=0).T, norm='ortho', axis=0)

def idct2(x):
    return spfft.idct(spfft.idct(x.T, norm='ortho', axis=0).T, norm='ortho', axis=0)

#%% VARIABLES FOR YOU TO CHANGE
path_to_your_image="./rsz_profile.png"
zoom_out=0.9999999 #Fraction of the image you want to keep.
corruption=0.9#Fraction of the pixels that you want to discard
#%% Get image and create y
# read original image and downsize for speed
orig =imageio.imread(path_to_your_image) # ----> read in color
np.random.seed(0) # ----> fix the random seed so that it'll be easier to check
# extract small sample of signal
X = spimg.zoom(orig, zoom_out)
ny,nx,ncolor = X.shape[0],X.shape[1],X.shape[2]
corruption=1-corruption
k = round(nx * ny * corruption)
ri = np.random.choice(nx * ny, k, replace=False) # random sample of indices

for i in range(0,3): # ----> create output files for each of the three constituent channels
  Xorig = orig.copy() # ----> make a copy of the original image each time
  # ----> mult_factor decides the offset
  """ ---->(note that since the image is flattened, each time we sample indices with respect to a channel,
  we need to choose the pixel value belonging to 'that' channel, hence the offset is required) """

  if i==0: # red ----> indexing red with 0 and setting the other channel pixel values to 0 (to extract red only)
    mult_factor = 1
    blank_1 = 1
    blank_2 = 2
  elif i==1: # green ----> indexing green with 1 and setting the other channel pixel values to 0 (to extract green only)
    mult_factor = 2
    blank_1 = 0
    blank_2 = 2
  else: # blue ----> indexing blue with 2 and setting the other channel pixel values to 0 (to extract blue only)
    mult_factor = 3
    blank_1 = 0
    blank_2 = 1
  # ----> set the other colors to
  Xorig[:,:,blank_1]=0
  Xorig[:,:,blank_2]=0

  #Downsize image
  X = spimg.zoom(Xorig, zoom_out)

  # extract small sample of signal
  b = X.T.flat[ri+(10000*(mult_factor-1))] # ----> note the offset here!
  b = np.expand_dims(b, axis=1)

  #%% CREATE A inverse and C
  # ********************************************************************************************
  """This part consumes a lot of memory. Your PC might crash if the images you load are larger than 100 x 100 pixels """
```

```
# create dct matrix operator using kron (memory errors for large ny*nx)
Aa = np.kron(
    np.float16(spfft.idct(np.identity(nx), norm='ortho', axis=0)),
    np.float16(spfft.idct(np.identity(ny), norm='ortho', axis=0))
    )
A = Aa[ri,:] # same as B times A
# ******************************************************************************************
# create images of mask (for visualization)
Xm = 255 * np.ones(X.shape)
Xm.T.flat[ri+(10000*(mult_factor-1))] = X.T.flat[ri+(10000*(mult_factor-1))] # ----> note the offset here!
Xm = Xm.astype(np.uint8)

plt.imshow(Xorig)
plt.title("Original")
plt.show()

plt.imshow(Xm)
plt.title("Incomplete")
plt.show()
#%% SAVE MATRICES TO DRIVE

import os
dir_name="Try12" # ----> you could name this directory as you like!
try:
    os.mkdir(dir_name)
except Exception as e:
    pass

np.save(dir_name+'/C'+str(i),A)
np.save(dir_name+'/A_inv'+str(i),Aa)
np.save(dir_name+'/y'+str(i),b)
plt.imsave(dir_name+'/incomplete'+str(i)+'.png',Xm)
plt.imsave(dir_name+'/original_with_crop'+str(i)+'.png',X)

# ----> need to clear the arrays or else artefacts will end up in the reconstructed images!
X = []
b = []
A = []
Aa = []
Xm = []
```

create_data_for_assignment_colour.py

Scan this QR code to access the GitHub repository of my homework solutions at https://github.com/ksanu1998/MDS_HW_Solutions