MRI Recon Project

December 2023

洪浩天, 2023291007, doge@shanghaitech.edu.cn

https://github.com/bughht/MRI_ReconLab_ShanghaiTech

Iterative Sense

Consider the fully sampled image m.mat and the coil sensitivity maps C.mat from an 8-channel brain coil (Nc = 8). Apply the coil sensitivity information to the image data creating the coil images m c obtained by each of the coils.

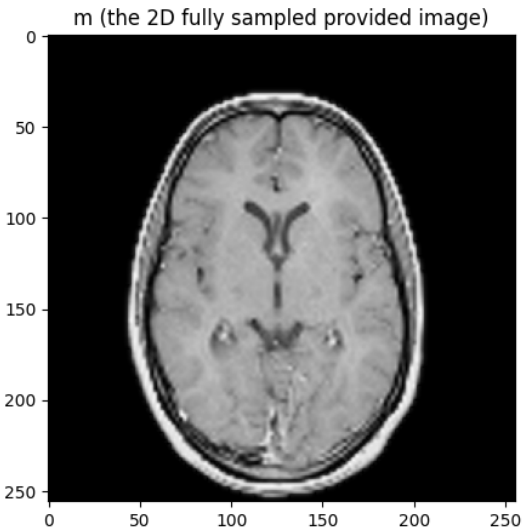
```
import numpy as np
from numpy.fft import *
from scipy.io import loadmat
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
from tqdm import tqdm

# Load CSM and images

C = loadmat('SUBMIT/C.mat')['C']
m = loadmat('SUBMIT/M.mat')['M']
Nc = 8
```

Here we load the required packages and the data from the .mat files.

Assignment A

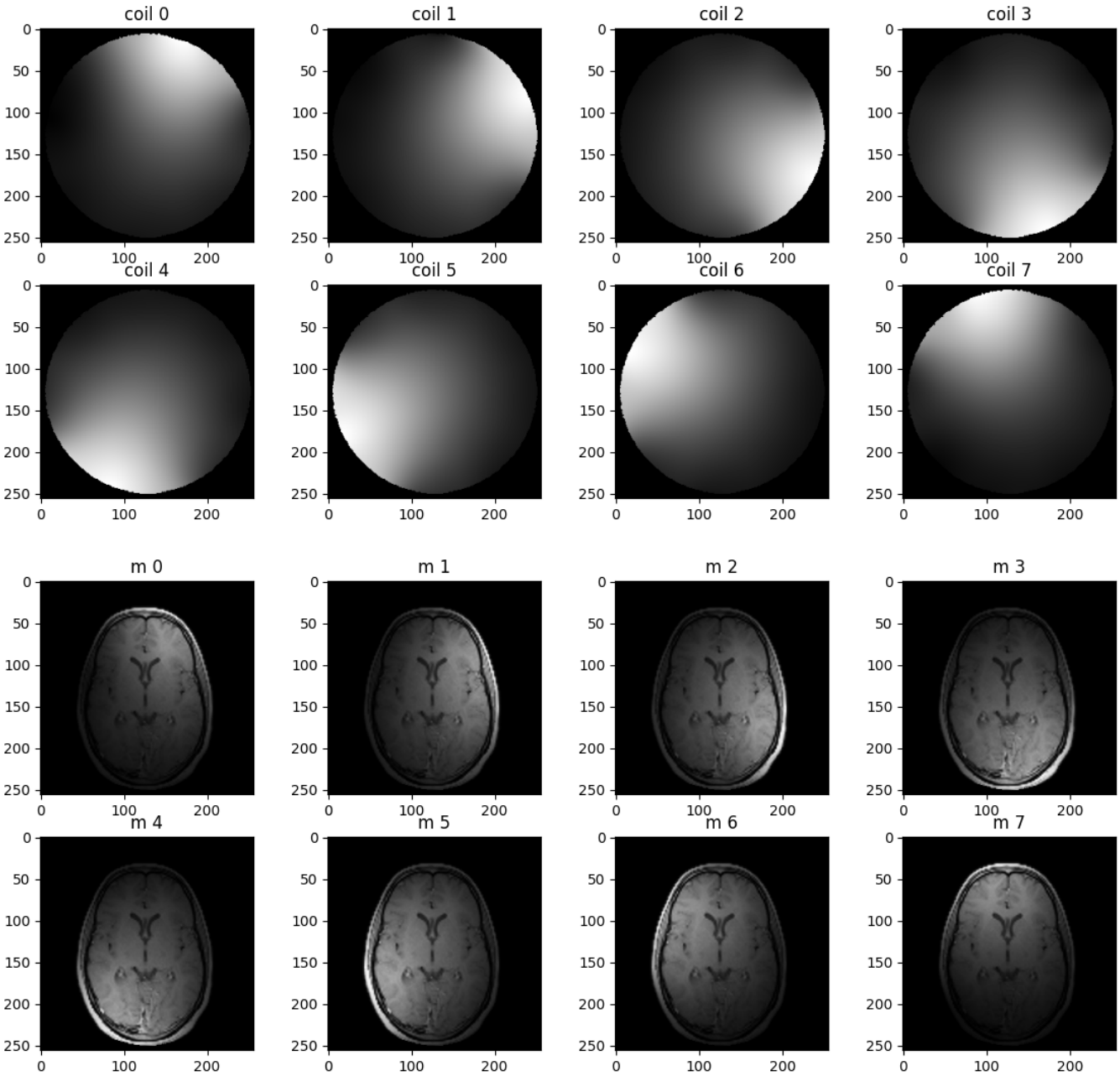
(4marks) Determine and depict m (the 2D fully sampled provided image), $C_i = 1, 2, \cdots, N_c$ and $m_i = 1, 2, \cdots, N_c$. Employ the root-sum-of-square approach and the weighted coil sensitivity
approach to combine the data from the individual coils. Depict and compare the combined images with respect to the original fully sample image m.

```
# original image
plt.figure(figsize=(5, 5))
plt.imshow(np.abs(m), cmap='gray')
plt.title("m (the 2D fully sampled provided image)")
plt.show()
```



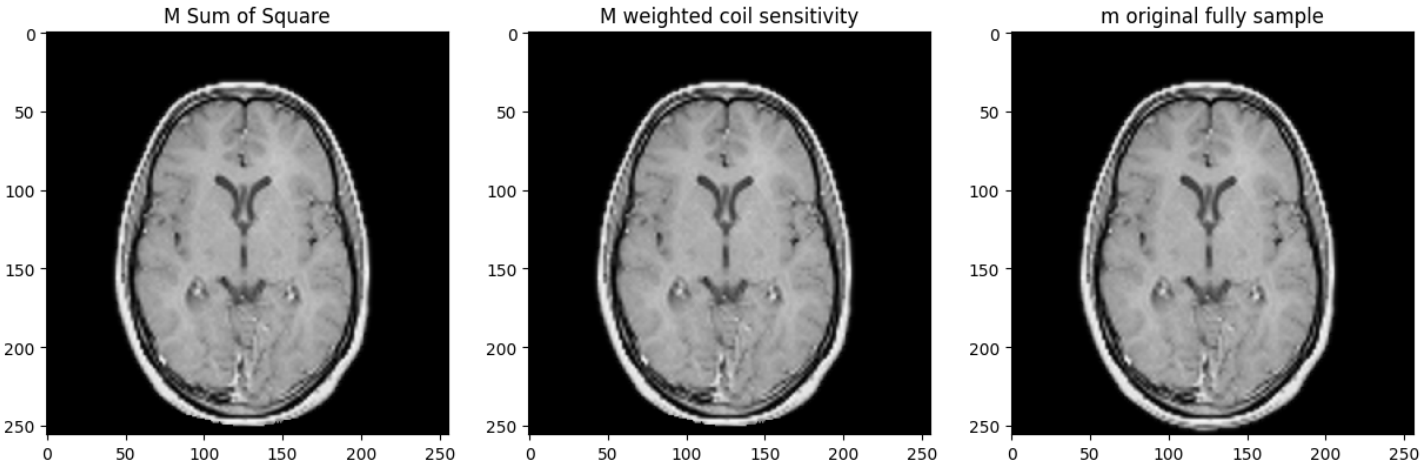
2D fully sampled image m was depicted above

```
# Coil sensitivity maps and coil images
C_rss = np.sqrt(np.sum(np.abs(C)**2, axis=2))+1e-11
# C_rss = np.sqrt(np.sum(C*C.conj(), axis=2))+1e-11
C /= C_rss[:, :, np.newaxis]

plt.figure(figsize=(14, 6))
for i in range(Nc):
    plt.subplot(2, 4, i+1)
    plt.imshow(np.abs(C[:, :, i]), cmap='gray')
    plt.title("coil "+str(i+0))
plt.show()

M = np.reshape(np.repeat(m, Nc, axis=1), C.shape)*C
plt.figure(figsize=(14, 6))
for i in range(Nc):
    plt.subplot(2, 4, i+1)
    plt.imshow(np.abs(M[:, :, i]), cmap='gray')
    plt.title("m "+str(i+0))
plt.show()
```



Here we normalized the coil sensitivity maps to ensure $C^H C = I$. Then we applied them to the image data to simulated the coiled images.

```
# root-sum-of-square
M_sos = np.sqrt(np.sum(np.abs(M)**2, axis=2))
# weighted coil sensitivity
M_wcs = np.sum(M*C.conj(), axis=2)

plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(np.abs(M_sos), cmap='gray')
plt.title("M Sum of Square")
plt.subplot(1, 3, 2)
plt.imshow(np.abs(M_wcs), cmap='gray')
plt.title("M weighted coil sensitivity")
plt.subplot(1, 3, 3)
plt.imshow(np.abs(m), cmap='gray')
plt.title("m original fully sample")
plt.show()
```



Two types of combination were used: root-sum-of-square approach and the weighted coil sensitivity approach.

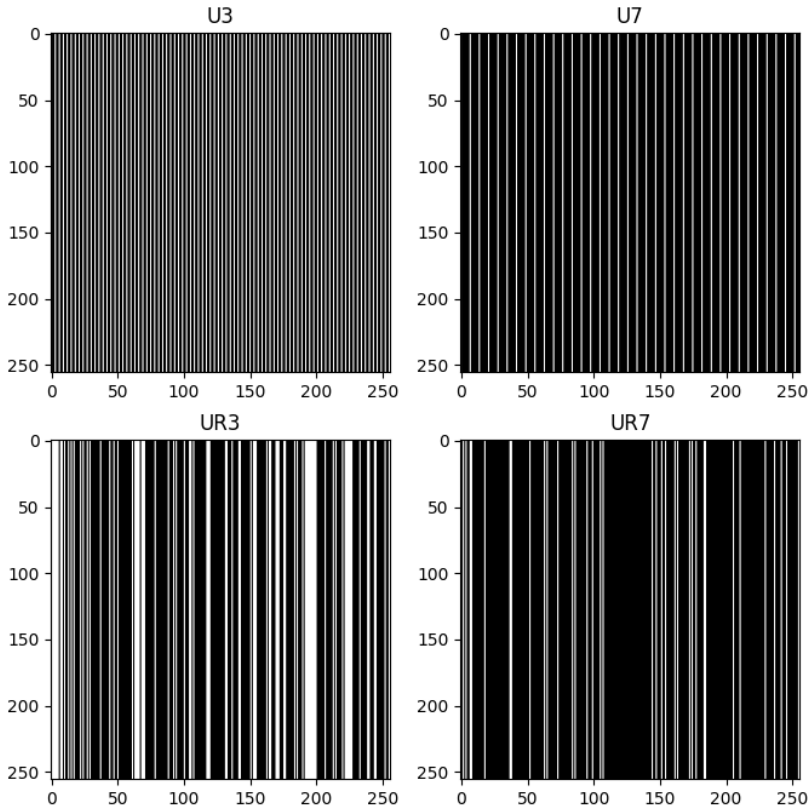
Root-sum-of-square approach: $M_{recon} = \sqrt{\sum_{i=1}^{N_c} |m_i|^2}$

Weighted coil sensitivity approach: $M_{recon} = \frac{\sum_{i=1}^{N_c} C_i^H m_i}{\sum_{i=1}^{N_c} C_i^H C_i}$, *considering that* $C^H C = I$, *here we simplify it into* $M_{recon} = \sum_{i=1}^{N_c} C_i^H m_i$

Assignment b

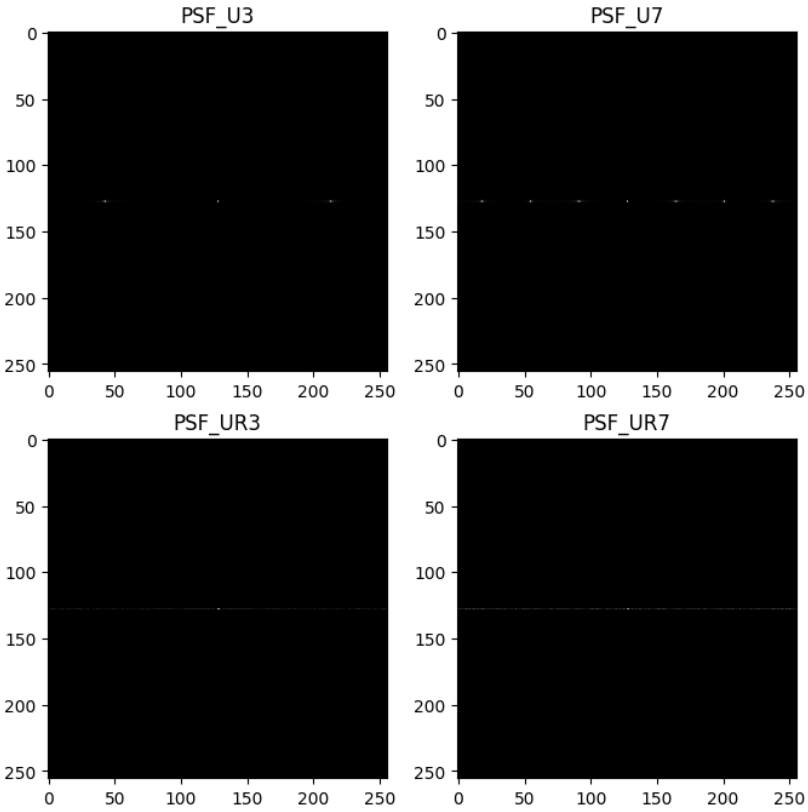
(6 marks) Generate 2 uniform undersampling patterns with acceleration factors of 3 and 7 (U_3 and U_7) and 2 random undersampling patterns (U_{R3} and U_{R7}). Each sampling pattern must be a matrix with 1s in the sampled positions and 0s in the remaining entries. Obtain the corresponding point spread functions (PSFs) and comment about the expected aliasing generated by these undersampling patterns.

```
U3 = np.zeros_like(M)
U7 = np.zeros_like(M)
UR3 = np.zeros_like(M)
UR7 = np.zeros_like(M)

U3[:, np.arange(0, 256, 3, dtype=int), :] = 1
U7[:, np.arange(0, 256, 7, dtype=int), :] = 1
UR3[:, np.random.choice(256, 256//3, replace=False)] = 1
UR7[:, np.random.choice(256, 256//7, replace=False)] = 1

plt.figure(figsize=(8,8))
plt.subplot(2,2,1)
plt.imshow(np.abs(U3[:,:,0]), cmap='gray')
plt.title("U3")
plt.subplot(2,2,2)
plt.imshow(np.abs(U7[:,:,0]), cmap='gray')
plt.title("U7")
plt.subplot(2,2,3)
plt.imshow(np.abs(UR3[:,:,0]), cmap='gray')
plt.title("UR3")
plt.subplot(2,2,4)
plt.imshow(np.abs(UR7[:,:,0]), cmap='gray')
plt.title("UR7")
plt.show()
```



Undersampling masks were demonstrated above, for acceleration factor of 3 and 7 of uniform and random sampling

```
PSF_U3 = np.fft.fftshift(np.fft.fft2(np.fft.ifftshift(U3[:,:,0])))
PSF_U7 = np.fft.fftshift(np.fft.fft2(np.fft.ifftshift(U7[:,:,0])))
PSF_UR3 = np.fft.fftshift(np.fft.fft2(np.fft.ifftshift(UR3[:,:,0])))
PSF_UR7 = np.fft.fftshift(np.fft.fft2(np.fft.ifftshift(UR7[:,:,0])))

plt.figure(figsize=(8,8))
plt.subplot(2,2,1)
plt.imshow(np.abs(PSF_U3), cmap='gray')
plt.title("PSF_U3")
plt.subplot(2,2,2)
plt.imshow(np.abs(PSF_U7), cmap='gray')
plt.title("PSF_U7")
plt.subplot(2,2,3)
plt.imshow(np.abs(PSF_UR3), cmap='gray')
plt.title("PSF_UR3")
plt.subplot(2,2,4)
plt.imshow(np.abs(PSF_UR7), cmap='gray')
plt.title("PSF_UR7")
plt.show()
```



The PSFs for undersampling patterns were obtained by taking the inverse Fourier transform of the sampling patterns.

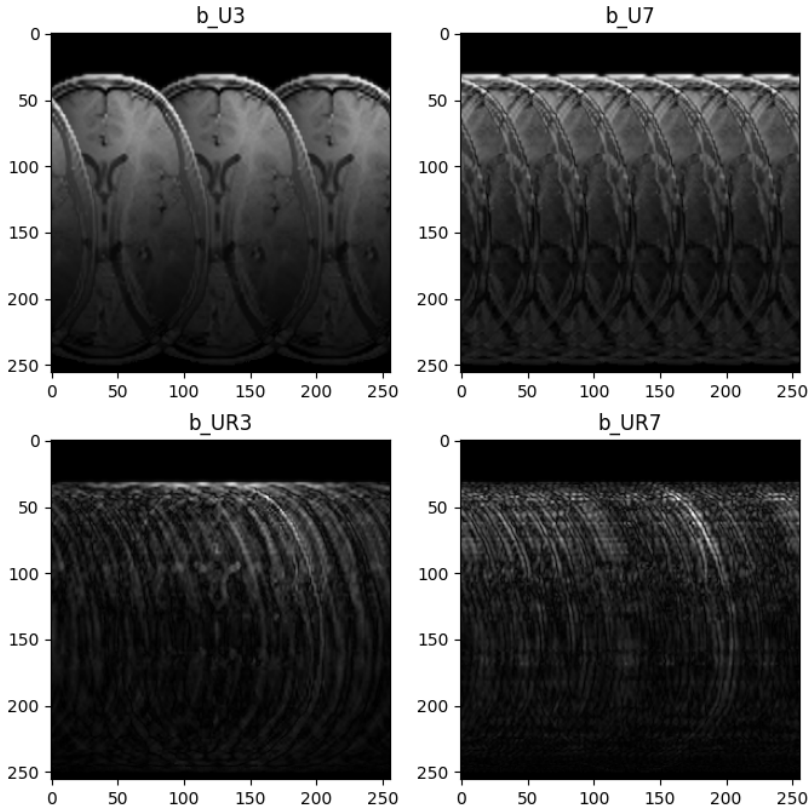
Assignment c

(4 marks) Obtain the aliased images for each coil as a result of undersampling with the generated patterns. For this you should use:

$$b_i = F^{-1}UFC_im$$

where U is the corresponding undersampling pattern, F is the Fourier transform and b i are the aliased images for each coil i= 1,2,...,Nc. Depict and compare the aliased images for the different undersampling factors and patterns.

```
b_U3 = fftshift(ifft2(ifftshift(U3*fftshift(fft2(ifftshift(M))))))
b_U7 = fftshift(ifft2(ifftshift(U7*fftshift(fft2(ifftshift(M))))))
b_UR3 = fftshift(ifft2(ifftshift(UR3*fftshift(fft2(ifftshift(M))))))
b_UR7 = fftshift(ifft2(ifftshift(UR7*fftshift(fft2(ifftshift(M))))))

plt.figure(figsize=(8,8))
plt.subplot(2,2,1)
plt.imshow(np.abs(b_U3)[:,:,0], cmap='gray')
plt.title("b_U3")
plt.subplot(2,2,2)
plt.imshow(np.abs(b_U7)[:,:,0], cmap='gray')
plt.title("b_U7")
plt.subplot(2,2,3)
plt.imshow(np.abs(b_UR3)[:,:,0], cmap='gray')
plt.title("b_UR3")
plt.subplot(2,2,4)
plt.imshow(np.abs(b_UR7)[:,:,0], cmap='gray')
plt.title("b_UR7")
plt.show()
```



Here we depicted the zero-filled recon result of k-space undersampled images of the first coil for 4 types of undersampling patterns mentioned previously.

Assignment d

(16 marks) The SENSE undersampled reconstruction can be written as a linear problem:

$$Em = B$$

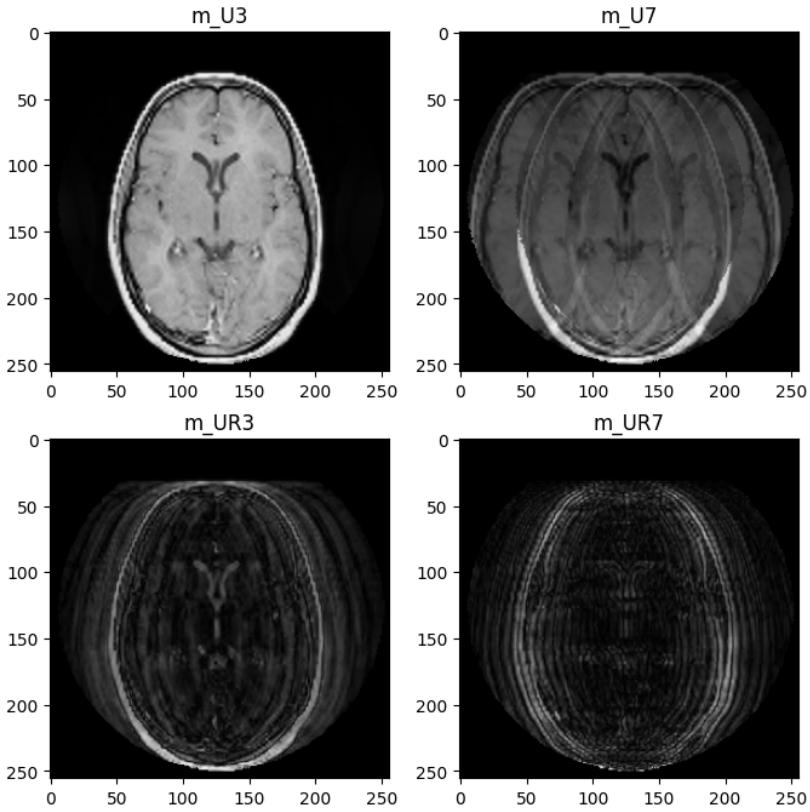
where m is the image to be reconstructed and the encoding matrix E = UFC corresponds to the forward sampling operator, with U the undersampling operator, F the Fourier transform operator, C the coil sensitivity maps and B the undersampled k-space data.

Iterative SENSE reconstruction is obtained by solving the linear problem Em = B as a least square minimization $\min_m \|Em - B\|_2^2$. Implement the Gradient Descent method to solve the above problem. To do this, you can use the forward E and conjugate transpose E H SENSE encoding operators you implemented in MRI Recon Lab.

Show and compare your results for the undersampling patterns generated in part b. What can you conclude from them? How many iterations are needed to reconstruct acquisitions with the different sampling patterns?

```
def E(U, C, m):
    return U*fftshift(fft2(ifftshift(C*m[:, :, np.newaxis]), axes=(0, 1)))


def EH(C, b):
    res = np.sum(fftshift(ifft2(ifftshift(b), axes=(0, 1)))
                 * C.conj(), axis=2)
    return res


def MSE(m, m_):
    e = m-m_
    return np.sum(np.abs(e)**2)/np.shape(m)[0]/np.shape(m)[1]


def Gradient_Descent(U, C, b, maxit):
    m_ = EH(C, b)
    r = m_-EH(C, E(U, C, m_))
    mse_history = [MSE(m, m_)]
    for i in tqdm(range(maxit)):
        a = np.abs(np.sum(r.conj()*r)/np.sum(r.conj()*EH(C, E(U, C, r))))
        m_ += a*r
        mse_history.append(MSE(m, m_))
        # Early Stopping
        err = np.sum(np.abs(r))
        if err < 1e-3:
            print('Converged at', i, 'with error', err)
            break
        r -= a*EH(C, E(U, C, r))
    return m_, mse_history


b_U3 = E(U3, C, m)
b_U7 = E(U7, C, m)
b_UR3 = E(UR3, C, m)
b_UR7 = E(UR7, C, m)

m_U3, mse_U3_GD = Gradient_Descent(U3, C, b_U3, 20)
m_U7, mse_U7_GD = Gradient_Descent(U7, C, b_U7, 20)
m_UR3, mse_UR3_GD = Gradient_Descent(UR3, C, b_UR3, 50)
m_UR7, mse_UR7_GD = Gradient_Descent(UR7, C, b_UR7, 50)
# m_U3, mse_U3_GD = Gradient_Descent(U3, C, b_U3, 200)
# m_U7, mse_U7_GD = Gradient_Descent(U7, C, b_U7, 200)
# m_UR3, mse_UR3_GD = Gradient_Descent(UR3, C, b_UR3, 500)
# m_UR7, mse_UR7_GD = Gradient_Descent(UR7, C, b_UR7, 500)

print("Gradient Descend")
plt.figure(figsize=(8, 8))
plt.subplot(2, 2, 1)
plt.imshow(np.abs(m_U3), cmap='gray')
plt.title("m_U3")
plt.subplot(2, 2, 2)
plt.imshow(np.abs(m_U7), cmap='gray')
plt.title("m_U7")
plt.subplot(2, 2, 3)
plt.imshow(np.abs(m_UR3), cmap='gray')
plt.title("m_UR3")
plt.subplot(2, 2, 4)
plt.imshow(np.abs(m_UR7), cmap='gray')
plt.title("m_UR7")
plt.show()
```

```
100%|██████████| 20/20 [00:02<00:00,  7.32it/s]
100%|██████████| 20/20 [00:02<00:00,  7.40it/s]
100%|██████████| 50/50 [00:06<00:00,  7.25it/s]
100%|██████████| 50/50 [00:06<00:00,  7.47it/s]


Gradient Descend
```



Iterative SENSE solves linear problem $E\hat{m} = b$ as a least square minization $\min_m \|Em - b\|_2^2$. Applying the gradient descent method, the iterative SENSE algorithm is given by solving $\min_m \|(E^H E)m - E^H b\|_2^2$.

Gradient Descent algorithm:

Initialize:

$$m_0 = E^H b$$
$$r_0 = m_0 - E^H E m_0$$

Iteration:

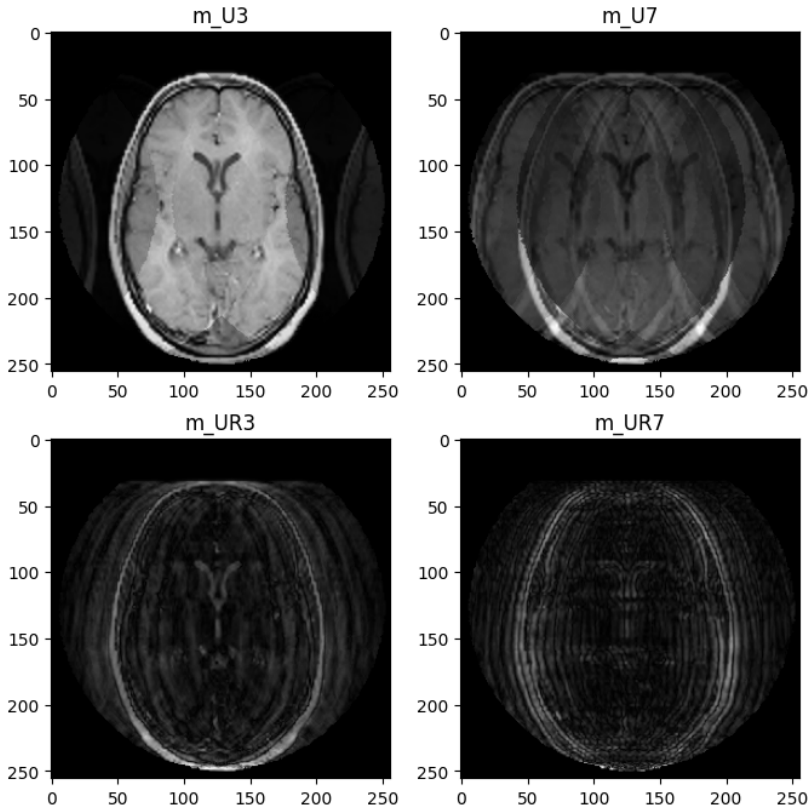
$$\alpha_k = \frac{r_k^T r_k}{r_K^T E^H E r_k}$$
$$m_{k+1} = m_k + \alpha_k r_k$$
$$r_{k+1} = r_k - \alpha_k E^H E r_k$$

20 steps iterations were applied to the uniform undersampling patterns U3 and U7, while 50 steps iterations were applied to the random undersampling patterns UR3 and UR7.

Early stopping technique was utilized to determine the number of iterations for each undersampling pattern. The stopping criteria was set to be the L1 norm of the difference between the current and previous iteration $\sum \|r_k\|_1$.

```python
def Conjugate_Gradient_Descent(U, C, b, maxit):
    m_ = EH(C, b)
    r = m_-EH(C, E(U, C, m_))
    p = r.copy()
    mse_history = [MSE(m, m_)]
    for i in tqdm(range(maxit)):
        a = np.abs(np.sum(r*r.conj())/np.sum(p.conj()*EH(C, E(U, C, p))))
        m_ += a*p
        r_ = r
        r -= a*EH(C, E(U, C, p))
        mse_history.append(MSE(m, m_))
        # Early Stopping
        err = np.sum(np.abs(r))
        if err < 1e-1:
            print('Converged at', i, 'with error', err)
            break
        b = np.abs(np.sum(r*r.conj())/np.sum(r_*r_.conj()))
        p = r + b*p
    return m_, mse_history

b_U3= E(U3, C, m)
b_U7= E(U7, C, m)
b_UR3= E(UR3, C, m)
b_UR7= E(UR7, C, m)

m_U3, mse_U3_CGD  = Conjugate_Gradient_Descent(U3, C, b_U3, 20)
m_U7, mse_U7_CGD = Conjugate_Gradient_Descent(U7, C, b_U7, 20)
m_UR3, mse_UR3_CGD = Conjugate_Gradient_Descent(UR3, C, b_UR3, 50)
m_UR7, mse_UR7_CGD  = Conjugate_Gradient_Descent(UR7, C, b_UR7, 50)

print("Conjugate Gradient Descend")
plt.figure(figsize=(8, 8))
plt.subplot(2, 2, 1)
plt.imshow(np.abs(m_U3), cmap='gray')
plt.title("m_U3")
plt.subplot(2, 2, 2)
plt.imshow(np.abs(m_U7), cmap='gray')
plt.title("m_U7")
plt.subplot(2, 2, 3)
plt.imshow(np.abs(m_UR3), cmap='gray')
plt.title("m_UR3")
plt.subplot(2, 2, 4)
plt.imshow(np.abs(m_UR7), cmap='gray')
plt.title("m_UR7")
plt.show()
```

```
100%|██████████| 20/20 [00:02<00:00,  7.01it/s]
100%|██████████| 20/20 [00:02<00:00,  7.23it/s]
100%|██████████| 50/50 [00:06<00:00,  7.18it/s]
100%|██████████| 50/50 [00:06<00:00,  7.32it/s]


Conjugate Gradient Descend
```



To investigate the performance between Gradient Descent (GD) and Conjugate Gradient Descent (CGD), we also implemented CGD algorithm for the same undersampling patterns. The CGD algorithm is given by:

Conjugate Gradient Descent

Initialize

$$
\begin{aligned}
m_0 &= E^H b \\
r_0 &= m_0 - E^H E m_0 \\
p_0 &= r_0
\end{aligned}
$$

Iteration:

$$
\begin{aligned}
\alpha_k &= \frac{r_k^T r_k}{p_K^T E^H E p_k} \\
m_{k+1} &= m_k + \alpha_k p_k \\
r_{k+1} &= r_k - \alpha_k E^H E r_k \\
\beta_k &= \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \\
p_{k+1} &= r_{k+1} + \beta_k p_k
\end{aligned}
$$

Maximum iteration steps and early stopping criteria were the same as GD.

According to the reconstruction image for GD and CGD SENSE, it's apparent that the result from GD for uniform undersampling patterns is better than that from CGD, especially for the acceleration factor of 3.

Assignment e

(5 marks) Define the reconstruction error as the difference between the fully sampled image {m} and your reconstructions $\hat{m}$ as:

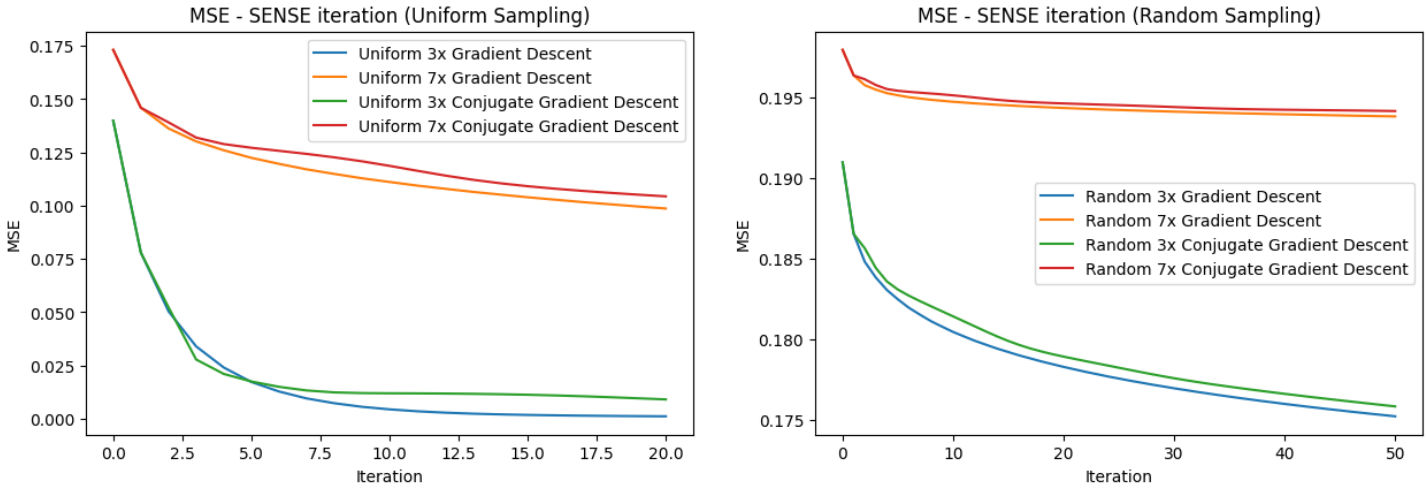
$$e = m - \hat{m}$$

and the mean square error (MSE) of the reconstruction as

$$\epsilon = \frac{1}{N} \sum_{i=1}^{N} |m(i) - \hat{m}(i)|^2$$

where $i = 1 \cdots N$ indicate each pixel in the image.
For all reconstructions (i.e. uniform and random 3x and 7x) plot the MSE w.r.t the number of iterations for the Gradient descent iterative SENSE method implemented. Comment about the convergence of the method.

```
plt.figure(figsize=(15, 10))
plt.subplot(2, 2, 1)
plt.plot(mse_U3_GD, label='Uniform 3x Gradient Descent')
plt.plot(mse_U7_GD, label='Uniform 7x Gradient Descent')
plt.plot(mse_U3_CGD, label='Uniform 3x Conjugate Gradient Descent')
plt.plot(mse_U7_CGD, label='Uniform 7x Conjugate Gradient Descent')
plt.legend()
plt.xlabel("Iteration")
plt.ylabel("MSE")
plt.title("MSE - SENSE iteration (Uniform Sampling)")

plt.subplot(2, 2, 2)
plt.plot(mse_UR3_GD, label='Random 3x Gradient Descent')
plt.plot(mse_UR7_GD, label='Random 7x Gradient Descent')
plt.plot(mse_UR3_CGD, label='Random 3x Conjugate Gradient Descent')
plt.plot(mse_UR7_CGD, label='Random 7x Conjugate Gradient Descent')
plt.legend()
plt.xlabel("Iteration")
plt.ylabel("MSE")
plt.title("MSE - SENSE iteration (Random Sampling)")
plt.show()
```



MSE-iteration relationship was plotted above for uniform and random undersampling patterns with acceleration factor of 3 and 7.

We can figure out that the MSE decreases with the increase of iterations. With regards to the convergence of the curves, the MSE of uniform undersampling patterns converges faster than that of random undersampling patterns. The MSE of acceleration factor of 3 decreases faster than that of acceleration factor of 7.

It's quite superising that the MSE of CGD converges rougher and slower than that of GD, with a relatively larger convergence error. The performance of CGD is not as good as that of GD, which could also be observed from the reconstructed images.

Compressed Sensing

```
import numpy as np
from numpy.fft import *
from scipy.io import loadmat
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
from tqdm import tqdm
import pywt

# Load Image
m = loadmat('SUBMIT/M.mat')['M']
```

Here we load the required packages and data from .mat files.

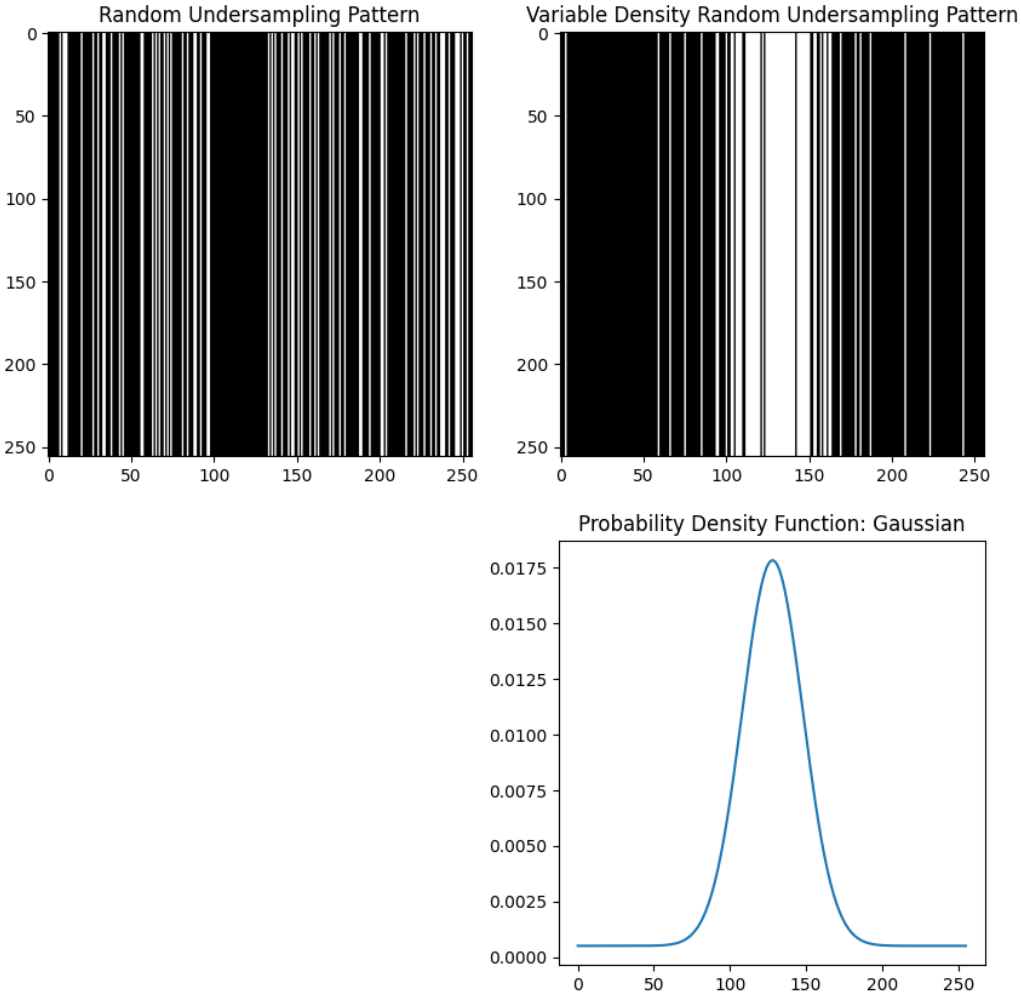
Assignment a

(4 marks) Generate a random undersampling pattern (U_R) and a variable density random undersampling pattern (U_{VDR}), both with acceleration factor of 4. Each sampling pattern must be a matrix with 1s in the sampled positions and 0s in the remaining ones. Obtain the corresponding PSFs and compare them. Obtained the aliased images as a result of undersampling with the generated patterns. For this you should use:

$$b = F^{-1}UFm$$

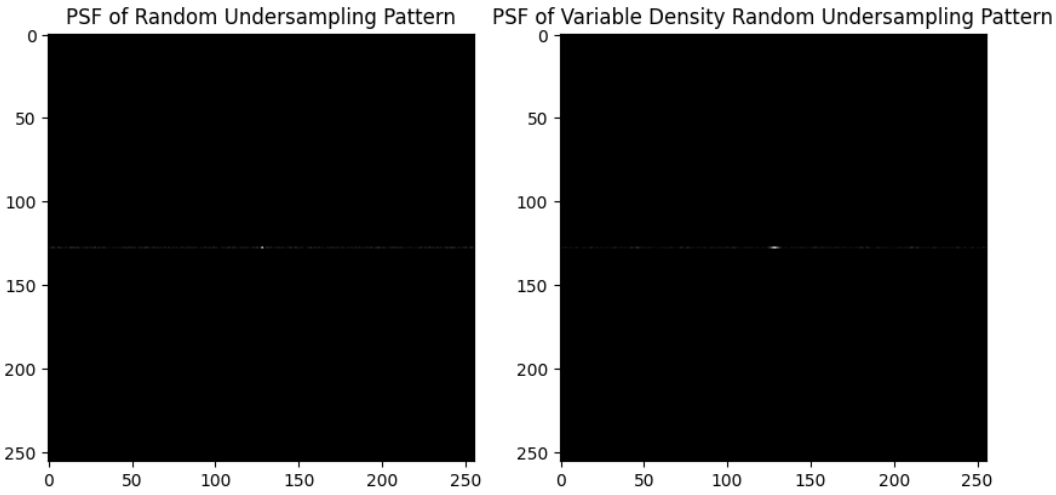
where m is the fully sampled image, U is the corresponding undersampling pattern, F is the Fourier transform and b is the aliased image. Depict the aliased images for the different undersampling patterns and compare against the fully sampled image.

```
UR = np.zeros_like(m)
UVDR = np.zeros_like(m)

UR[:, np.random.choice(256, 256//4, replace=False)] = 1
sigma = 2e1
bias = 3e-2
p = np.exp(-(np.arange(256)-128)**2/(2*sigma**2))+bias
p /= np.sum(p)
UVDR[:, np.random.choice(256, 256//4, replace=False, p=p)] = 1

plt.figure(figsize=(10, 10))
plt.subplot(2, 2, 1)
plt.imshow(UR, cmap='gray')
plt.title("Random Undersampling Pattern")
plt.subplot(2, 2, 2)
plt.imshow(UVDR, cmap='gray')
plt.title("Variable Density Random Undersampling Pattern")
plt.subplot(2, 2, 4)
plt.title("Probability Density Function: Gaussian")
plt.plot(p)
plt.show()
```



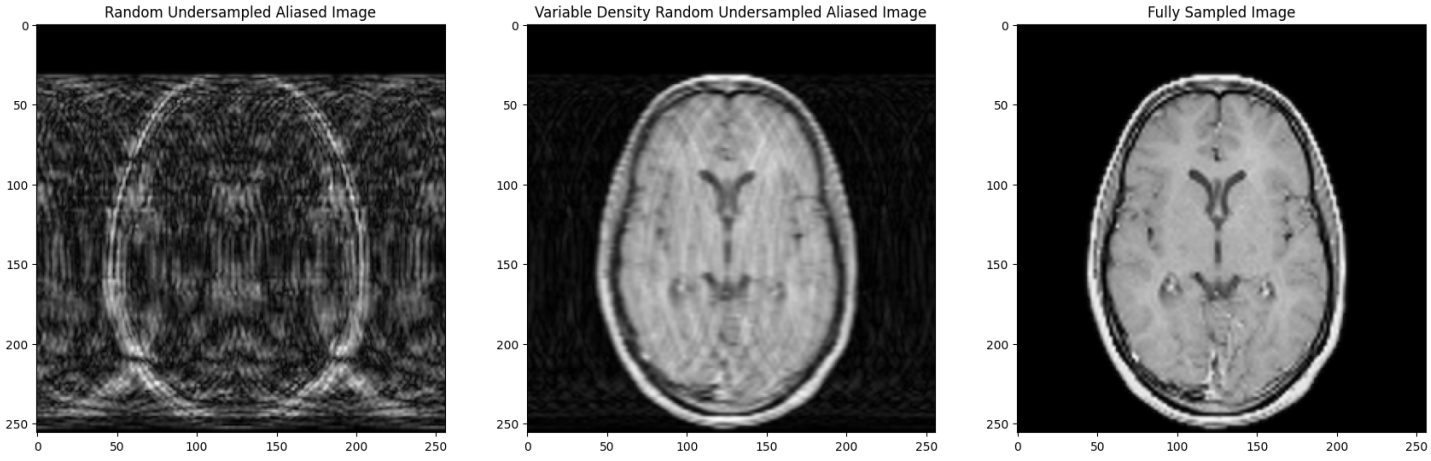
Two types of random undersampling patterns were generated, one with uniform random sampling (UR) and another one with variable density sampling (UVDR). The probablity density function (PDF) for the UVDR follows a normalized gaussian distribution with standard deviation of 2 and bias of 3e-2.

```
PSF_UR = np.fft.fftshift(np.fft.fft2(np.fft.ifftshift(UR)))
PSF_UVDR = np.fft.fftshift(np.fft.fft2(np.fft.ifftshift(UVDR)))

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(np.abs(PSF_UR), cmap='gray')
plt.title("PSF of Random Undersampling Pattern")
plt.subplot(1, 2, 2)
plt.imshow(np.abs(PSF_UVDR), cmap='gray')
plt.title("PSF of Variable Density Random Undersampling Pattern")
plt.show()
```



The PSFs for the UR and UVDR patterns depicted above were obtained by applying iFFT to the sampling patterns. It's observed that the PSF for the UVDR pattern is more concentrated than the PSF for the UR pattern. This is due to the fact that the UVDR pattern has a higher probability of sampling the center of the image than the UR pattern.

```
b_UR = fftshift(ifft2(ifftshift(UR*fftshift(fft2(ifftshift(m))))))
b_UVDR = fftshift(ifft2(ifftshift(UVDR*fftshift(fft2(ifftshift(m))))))

plt.figure(figsize=(21, 7))
plt.subplot(1, 3, 1)
plt.imshow(np.abs(b_UR), cmap='gray')
plt.title("Random Undersampled Aliased Image")
plt.subplot(1, 3, 2)
plt.imshow(np.abs(b_UVDR), cmap='gray')
plt.title("Variable Density Random Undersampled Aliased Image")
plt.subplot(1, 3, 3)
plt.imshow(np.abs(m), cmap='gray')
plt.title("Fully Sampled Image")
plt.show()
```



Above we compare the aliased images for the different undersampling patterns with the fully sampled image. It's apparent that the aliased image for the UR pattern is much more distorted than that for the UVDR pattern. This is due to the fact that UVDR undersampling pattern has a higher probability of sampling the lower frequencies components (center of the K-space) than the UR pattern.

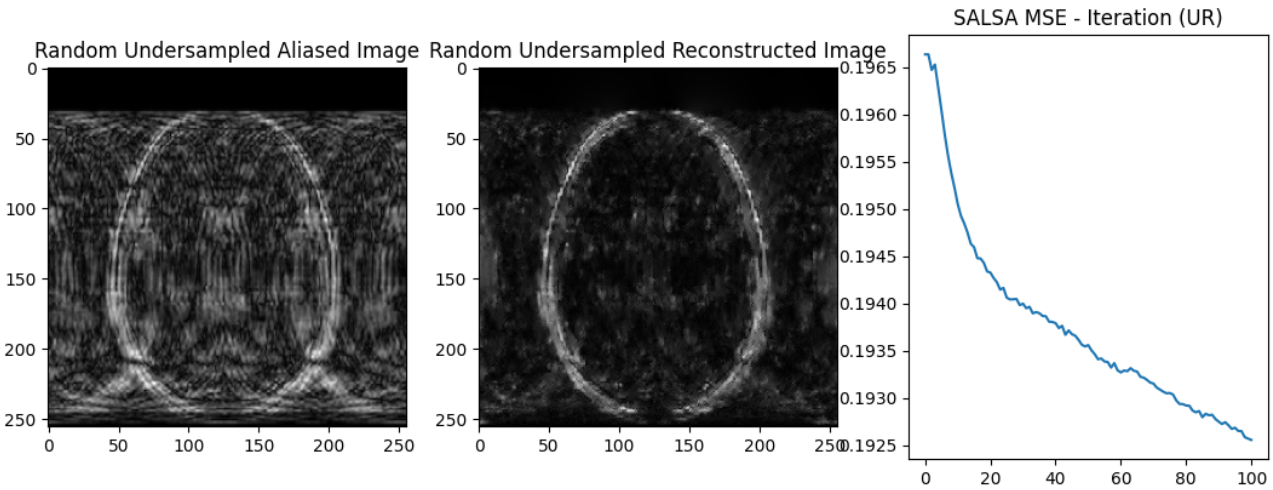
Assignment b

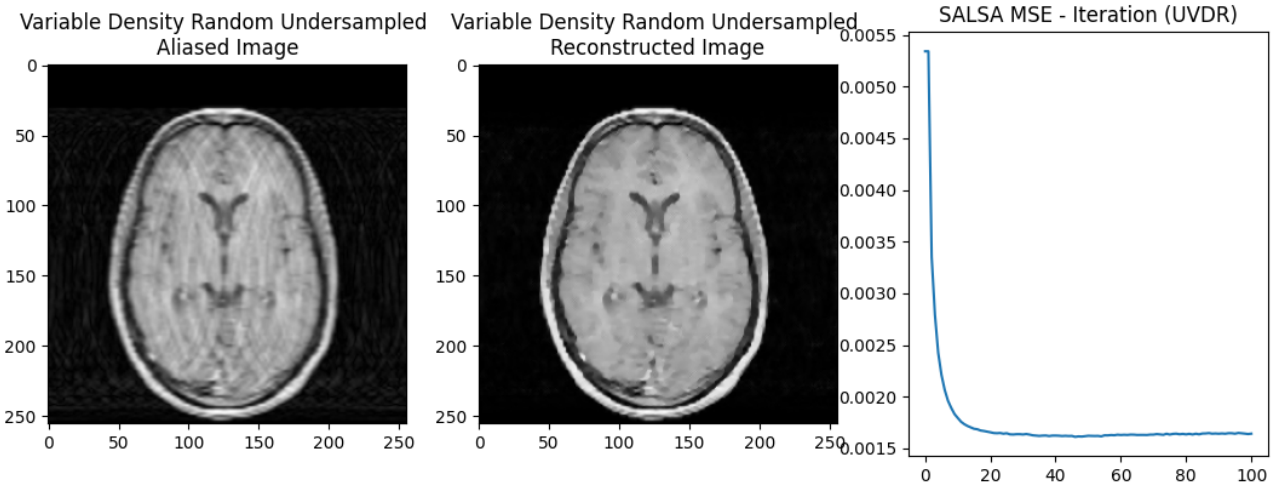
(6 marks) Select one algorithm, among those available on internet (some of these are listed below, and you can use any other algorithms you found), to solve the Compressed Sensing problem with Total Variation regularization. Download the software and briefly describe each of the algorithms, you can use the test examples, usually provided together with these softwares, for your description.
Examples of algorithms:

- L1-magic https://statweb.stanford.edu/~candes/software/l1magic/
- NESTA https://statweb.stanford.edu/~candes/software/nesta/
- TWIST http://www.lx.it.pt/~bioucas/TwIST/TwIST.htm
- **SALSA http://cascais.lx.it.pt/~mafonso/salsa.html**

```
from utils import *

def PH(m):
    return m


def E(m, U=UVDR):
    return U*fftshift(fft2(ifftshift(m)))


def EH(b):
    return fftshift(ifft2(ifftshift(b), axes=(0, 1)))


def Psi(m, threshold):
    return denoiseTV(m, threshold, 20)


def invLS_UVDR(m, mu):
    return (m-(1/(1+mu))*EH(E(m, UVDR)))/mu


def invLS_UR(m, mu):
    return (m-(1/(1+mu))*EH(E(m, UR)))/mu

def SALSA(b, m_gold, EH, Psi, PH, invLS, threshold=0.5, mu=0.1, maxit=100):
    # Solving argmin_m||Em-b||_2^2 + lambda Phi(m)
    EHb = EH(b)
    m_ = EHb
    PHm = EHb
    u = PHm
    bu = np.zeros_like(u)
    threshold = threshold

    r = invLS(m_, mu)

    mse_history=[MSE(m_,m_gold)]
    for iteration in tqdm(range(maxit)):
        r = EHb + mu*(u+bu)
        m_ = invLS(r, mu)
        u = Psi(m_-bu, threshold)
        PHm = PH(m_)
        bu += (u-PHm)
        mse_history.append(MSE(m_,m_gold))
    return m_, mse_history

img_recon_UR, mse_UR = SALSA(E(m, UR), m, EH, Psi, PH, invLS_UR,
                     threshold=1e-1, mu=1e-5, maxit=100)
img_recon_UVDR, mse_UVDR = SALSA(E(m, UVDR), m, EH, Psi, PH,
                       invLS_UVDR, threshold=1e-1, mu=1e-5, maxit=100)

plt.figure(figsize=(13, 10))
plt.subplot(2, 3, 1)
plt.imshow(np.abs(b_UR), cmap='gray')
plt.title("Random Undersampled Aliased Image")
plt.subplot(2, 3, 2)
plt.imshow(np.abs(img_recon_UR), cmap='gray')
plt.title("Random Undersampled Reconstructed Image")
plt.subplot(2,3,3)
plt.plot(mse_UR)
plt.title("SALSA MSE - Iteration (UR)")
plt.subplot(2, 3, 4)
plt.imshow(np.abs(b_UVDR), cmap='gray')
plt.title("Variable Density Random Undersampled \nAliased Image")
plt.subplot(2, 3, 5)
plt.imshow(np.abs(img_recon_UVDR), cmap='gray')
plt.title("Variable Density Random Undersampled \nReconstructed Image")
plt.subplot(2,3,6)
plt.plot(mse_UVDR)
plt.title("SALSA MSE - Iteration (UVDR)")
plt.show()
```

```
100%|██████████| 100/100 [00:09<00:00, 10.01it/s]
100%|██████████| 100/100 [00:09<00:00, 10.36it/s]
```





Here we select the SALSA algorithm to solve the Compressed Sensing problem with Total Variation regularization. Due to the fact that official demo code for SALSA is written in Matlab, here we implement the algorithm in Python and apply it to our single coil CS reconstruction problem.

SALSA (split augmented Lagrangian shrinkage algorithm) algorithm is a fast and robust algorithm for solving formulations of a unconstrained optimization problem with a non-smooth regularization term. SALSA is based on variable splitting and augmented lagrangian methods.

The CS reconstruction problem is formulated as follows:

$$\min_{m,\theta} \|Em - b\|_2^2 + \lambda\Phi(\theta)$$
$$\text{s.t.} \quad m = \theta$$

Where $E = UF$ is the forward operator for image to k-space and undersampling, b is the undersampled k-space data, m is the image to be reconstructed, θ is the auxiliary variable introduced by the variable splitting method, Φ is the regularization term (Total Variation), and λ is the regularization parameter.

The algorithm is given by the following steps:

Initialization:

$$choose \quad \mu > 0, m_0, \theta_0, d_0$$
$$\bar{b} = E^H b$$

Iteration

$$\begin{aligned} m'_k &= \theta_k + d_k \\ r_k &= \bar{b} + \mu m'_k \\ m_{k+1} &= \frac{1}{\mu}(I - E^H E) r_k \\ \theta'_k &= m_{k+1} - d_k \\ \theta_{k+1} &= \Psi_{r\Phi/\mu}(\theta'_k) \\ d_{k+1} &= d_k - \beta_{k+1} + \theta_{k+1} \end{aligned}$$

Where $\Psi_{r\Phi/\mu}$ denotes shrinkage/thresholding function for denoiseing the image

Assignment c

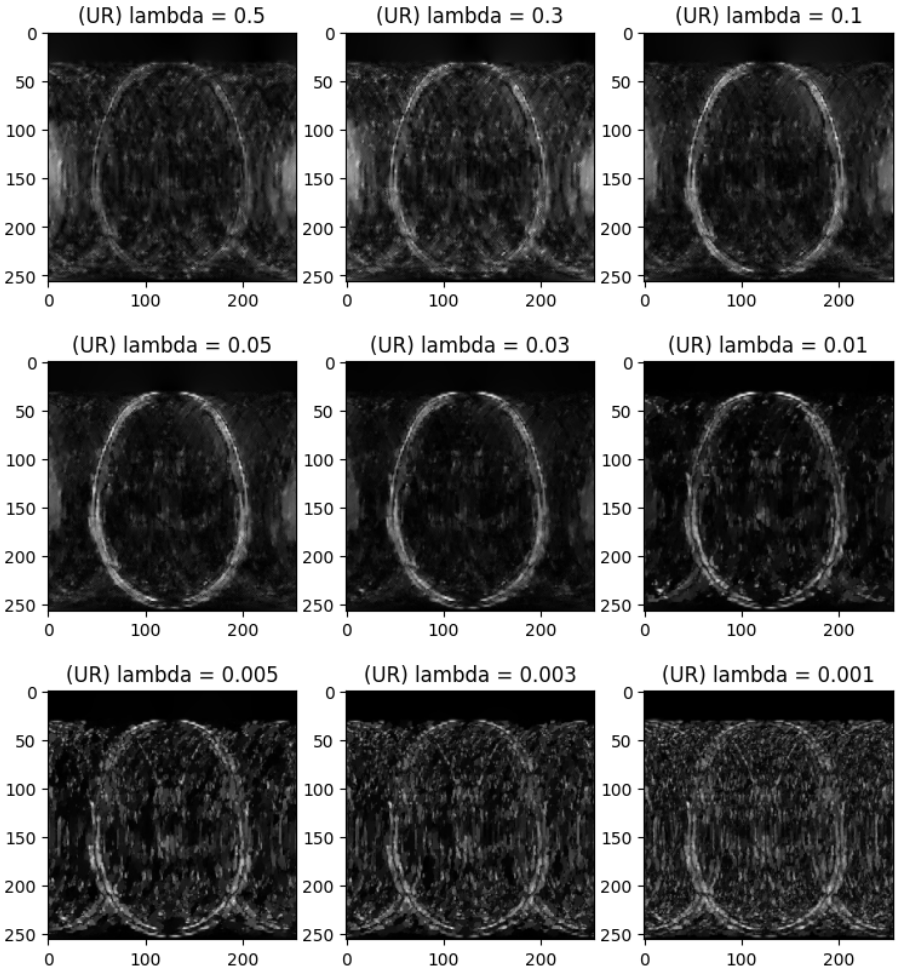
(15 marks) Employ one of the algorithms to solve the undersampled problem in MRI for both undersampling patterns, i.e.:

$$\min_m \|Em - b\|_2^2 + \lambda TV(m)$$

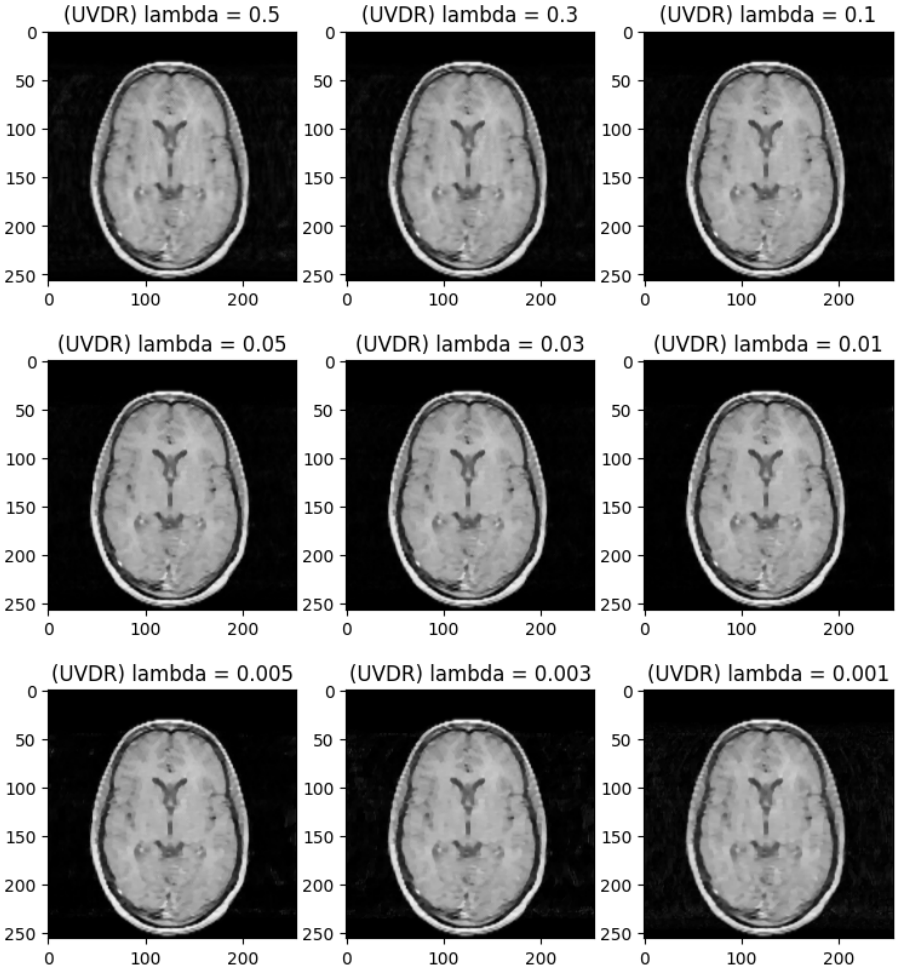
where E = UF corresponds to the forward sampling operator, with U the undersampling operator, F the Fourier transform operator, B the undersampled k-space data and TV the spatial finite difference operator. Show and compare your results for different values of the parameter λ, including those with the near optimal parameters.

```
lambdas = [0.5, 0.3, 0.1, 0.05, 0.03, 0.01, 0.005, 0.003, 0.001]
mse_UR_list = []
mse_UVDR_list = []
plt.figure(figsize=(9, 10))
print("SALSA with UR")
for i, lm in enumerate(lambdas):
    mu = 1e-4
    img_recon_UR, mse_UR = SALSA(E(m, UR), m, EH, Psi, PH,
                          invLS_UR, threshold=lm, mu=mu, maxit=500)
    mse_UR_list.append(mse_UR)
    plt.subplot(3, 3, i+1)
    plt.imshow(np.abs(img_recon_UR), cmap='gray')
    plt.title("(UR) lambda = {}".format(lm, mu))
plt.show()
plt.figure(figsize=(9, 10))
print("SALSA with UVDR")
for i, lm in enumerate(lambdas):
    mu = 1e-4
    img_recon_UVDR, mse_UVDR = SALSA(E(m, UVDR), m, EH, Psi, PH,
                          invLS_UVDR, threshold=lm, mu=mu, maxit=500)
    mse_UVDR_list.append(mse_UVDR)
    plt.subplot(3, 3, i+1)
    plt.imshow(np.abs(img_recon_UVDR), cmap='gray')
    plt.title("(UVDR) lambda = {}".format(lm, mu))
plt.show()
```

```
SALSA with UR
```



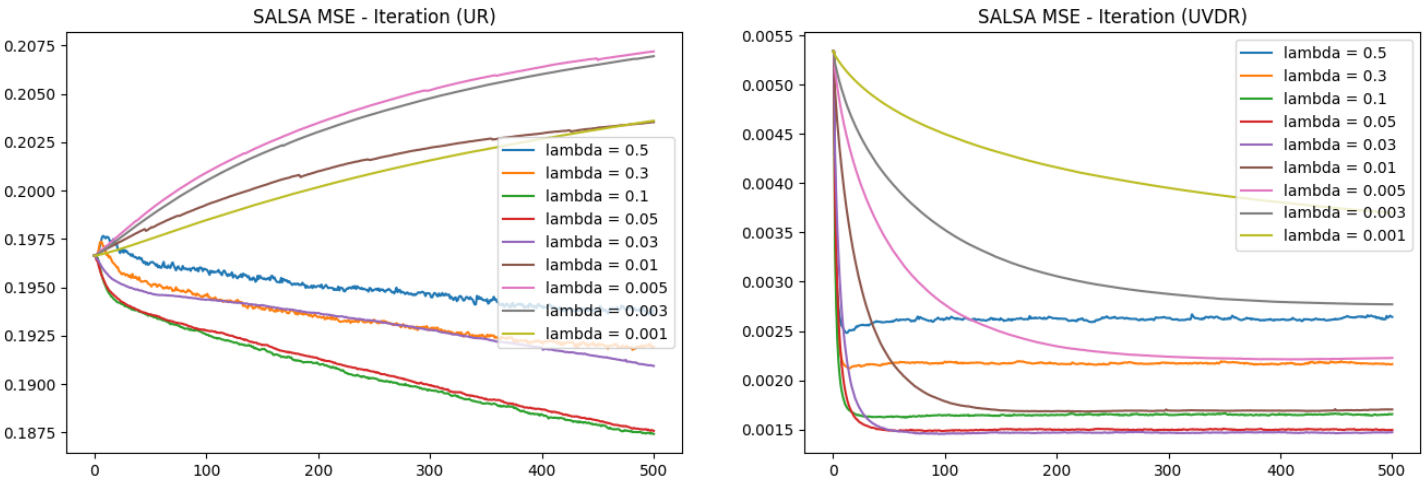

```
SALSA with UVDR
```



Above we depict the reconstructed images for the UR and UVDR undersampling patterns with different values of the regularization parameter λ. It's observed that the reconstructed images for the UVDR pattern are much better than those for the UR pattern. This is due to the fact that the UVDR pattern has a higher probability of sampling the lower frequencies components (center of the K-space) than the UR pattern.

With regards to the regularization parameter λ, it's apparent that the reconstructed image quality is highly dependent on λ. When λ is too small or too large, aliasing artifacts couldn't be removed. For the UVDR pattern, the optimal value of λ is around 0.03

```
plt.figure(figsize=(16, 5))
plt.subplot(1, 2, 1)
for i in range(len(lambdas)):
    plt.plot(mse_UR_list[i], label='lambda = {}'.format(lambdas[i]))
plt.title("SALSA MSE - Iteration (UR)")
plt.legend()
plt.subplot(1, 2, 2)
for i in range(len(lambdas)):
    plt.plot(mse_UVDR_list[i], label='lambda = {}'.format(lambdas[i]))
plt.title("SALSA MSE - Iteration (UVDR)")
plt.legend()
plt.show()
```



MSE criteria is collected to analyze the reconstruction performance during SALSA iterations.

With regards to the UR pattern, considering that the image contrast for the undersampled image is extremely low (due to the fact that the UR pattern has a very low probability of sampling the lower frequencies components), MSE converges to a relatively high value (>1.8) after about 50 iterations for lambda range from 0.03 to 0.5, and for lambda below 0.3, MSE increases with iterations, which indicates that the reconstructed image quality is getting worse with iterations.

With regards to the UVDR pattern undersampling, aliasing artifacts are much less severe than those for the UR pattern. MSE converges to a extremely low value around 0.0015 after about 50 iterations for the optimal lambda value of 0.03. For lambda below 0.3, the steps required for MSE to converge to the optimal value increases with lambda.

Assignment d

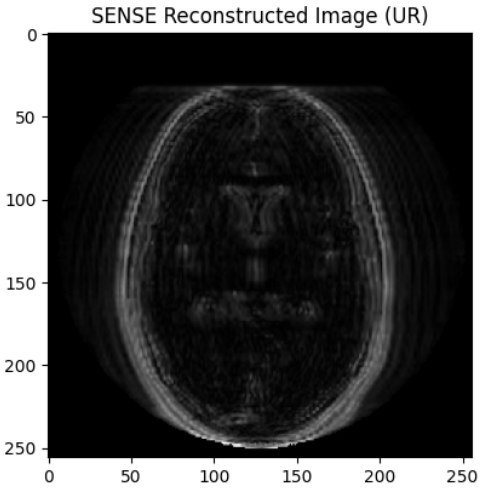
(10 marks) Employ the iterative SENSE reconstruction implemented in Question 1 to reconstruct the undersampled problem in MRI for both undersampling patterns (considering a parallel imaging acquisition with coil maps given in Question 1). Compare the algorithms and performance of iterative SENSE versus the Compressed Sensing (single coil) method you investigated in (c). This comparison could consider e.g. usability, number of iterations, convergence, root mean square error, computation time, etc.

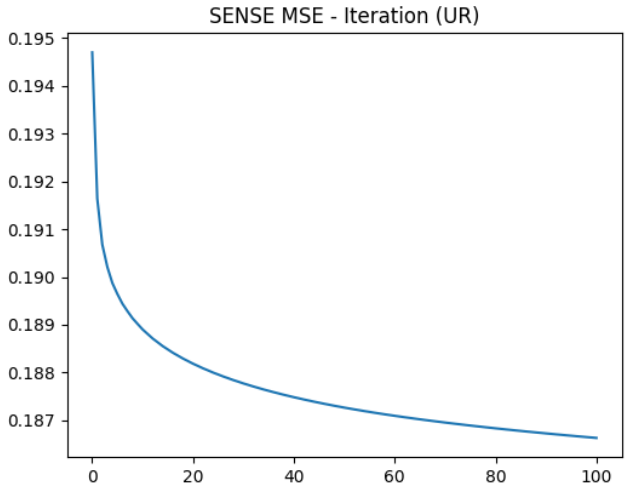
```
Nc = 8
C = loadmat('SUBMIT/C.mat')['C']
C_rss = np.sqrt(np.sum(np.abs(C)**2, axis=2))+1e-11
C /= C_rss[:, :, np.newaxis]
M = np.reshape(np.repeat(m, Nc, axis=1), C.shape)*C

def E_c(U, C, m):
    return U[:,:,np.newaxis]*fftshift(fft2(ifftshift(C*m[:, :, np.newaxis]), axes=(0, 1)))


def EH_c(C, b):
    return np.sum(fftshift(ifft2(ifftshift(b), axes=(0, 1)))
                  * C.conj(), axis=2)


def Gradient_Descent(U, C, b, maxit):
    m_ = EH_c(C, b)
    r = m_-EH_c(C, E_c(U, C, m_))
    mse_history = [MSE(m, m_)]
    for i in tqdm(range(maxit)):
        a = np.abs(np.sum(r.conj()*r)/np.sum(r.conj()*EH_c(C, E_c(U, C, r))))
        m_ += a*r
        mse_history.append(MSE(m, m_))
        # Early Stopping
        err = np.sum(np.abs(r))
        if err < 1e-3:
            print('Converged at', i, 'with error', err)
            break
        r -= a*EH_c(C, E_c(U, C, r))
    return m_, mse_history

b_c_UR = E_c(UR, C, m)
b_c_UVDR = E_c(UVDR, C, m)

m_UR, mse_UR = Gradient_Descent(UR, C, b_c_UR, 100)
m_UVDR, mse_UVDR = Gradient_Descent(UVDR, C, b_c_UVDR, 100)

plt.figure(figsize=(13, 10))
plt.subplot(2, 2, 1)
plt.imshow(np.abs(m_UR), cmap='gray')
plt.title("SENSE Reconstructed Image (UR)")

plt.subplot(2, 2, 2)
plt.plot(mse_UR)
plt.title("SENSE MSE - Iteration (UR)")

plt.subplot(2, 2, 3)
plt.imshow(np.abs(m_UVDR), cmap='gray')
plt.title("SENSE Reconstructed Image (UVDR)")

plt.subplot(2, 2, 4)
plt.plot(mse_UVDR)
plt.title("SENSE MSE - Iteration (UVDR)")
```

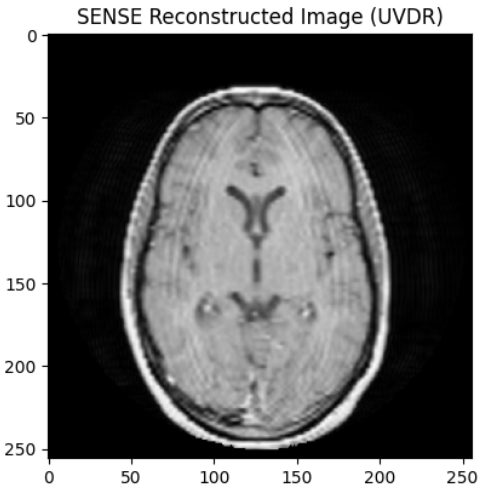
```
100%|██████████| 100/100 [00:10<00:00,  9.62it/s]
100%|██████████| 100/100 [00:10<00:00,  9.66it/s]
```

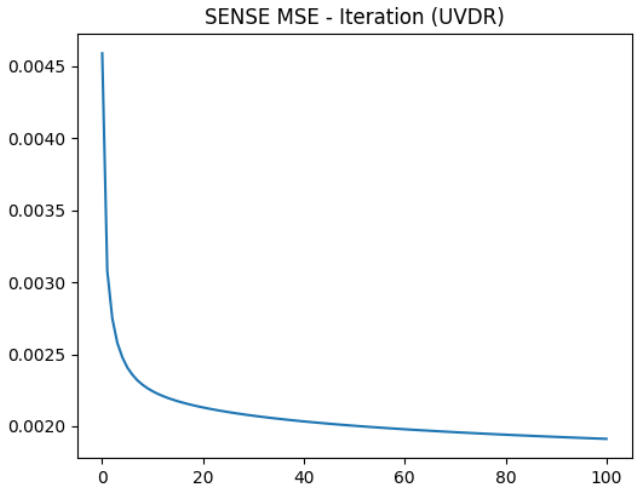


```
Text(0.5, 1.0, 'SENSE MSE - Iteration (UVDR)')
```









To compare the performance of iterative SENSE with SINA, we implement the iterative SENSE reconstruction for the UR and UVDR undersampling patterns with coil sensitivity. We analyze the algorithm performance from the following aspects:

Usability: *Iterative SENSE is much easier to implement than SALSA, it requires doesn't have any hyperparameters to tune, and it converges extremely fast. However, iterative SENSE requires coil sensitivity maps and parallel imaging with multiple coils. SALSA doesn't require coil sensitivity maps and it can be applied to single coil CS reconstruction.*

Convergence: *Iterative SENSE converges extremely fast, it only requires about 20 iterations to converge. SALSA requires about 100 iterations to converge. But SALSA with optimal lambda value reaches a better reconstruction performance than iterative SENSE.*

Image quality: *Although image quality could be assessed by MSE, some slight artifacts could be discovered by visual inspection. For the UVDR pattern, image reconstructed by iterative SENSE has vertical stripe artifacts, while image reconstructed by SALSA seems to be much smoother.*

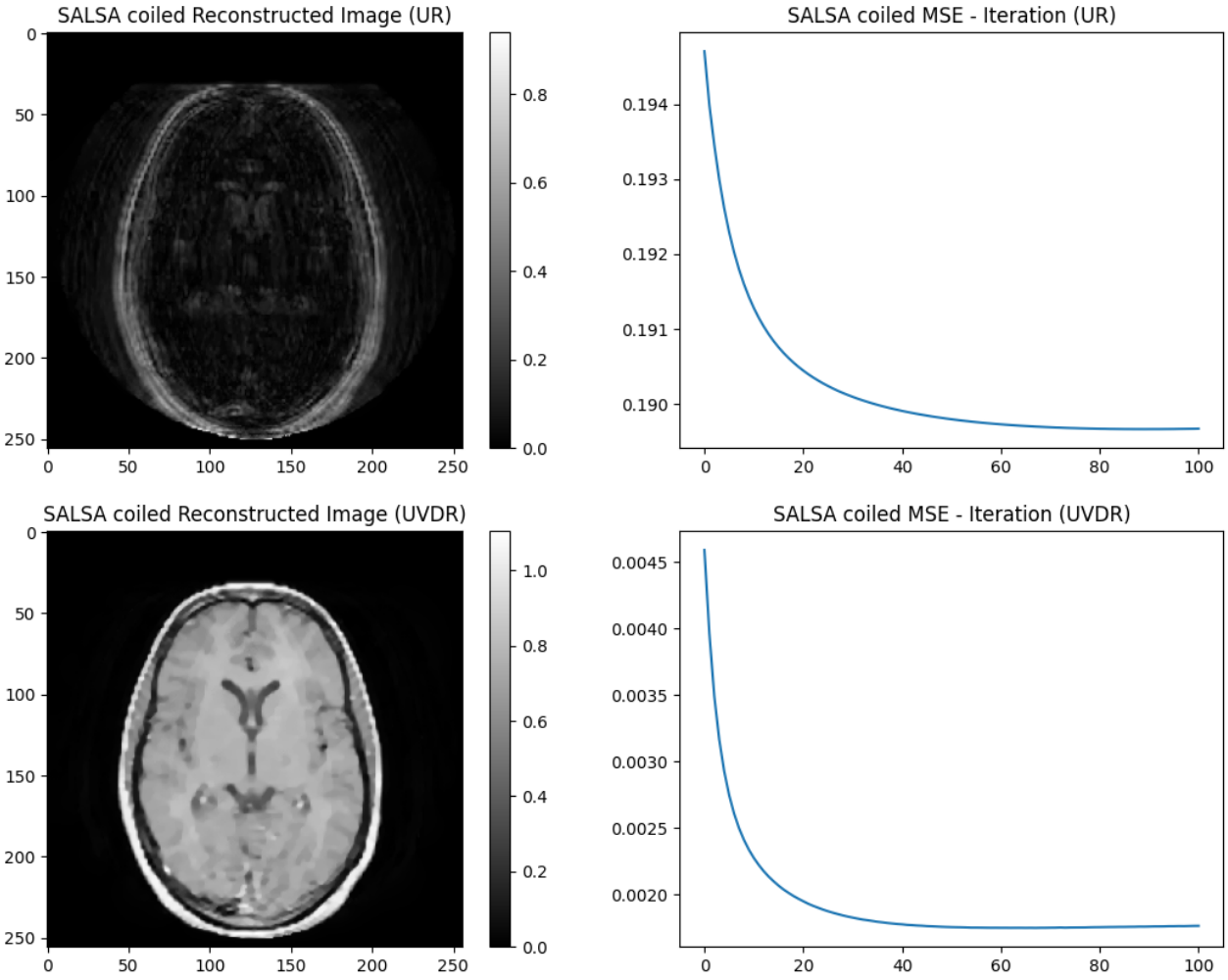
Computation time: *Iterative SENSE and SALSA are both extremely fast. Although SALSA requires more iterations, each iteration is faster than that for iterative SENSE. For the optimal lambda value, SALSA and iterative SENSE both requires about 10 seconds to converge for the UVDR pattern.*

Assignment e

(15 marks) Considering now a parallel imaging acquisition with coil maps given in Question 1 for Compressed Sensing, modify the algorithm you investigated in (c) to combine Compressed Sensing and Parallel Imaging, i.e. now E = UFC. Compare the performance of this reconstruction with your results obtained in (d).

```
def SALSA_c(b, C, m_gold, EH_c, Psi, PH, invLS, threshold=0.5, mu=0.1, maxit=100):
    # SALSA for coil sensitivity (E = UFC)
    EHb = EH_c(C, b)
    m_ = EHb
    u = m_
    bu = np.zeros_like(u)

    r = invLS(m_, mu)

    mse_history=[MSE(m_,m_gold)]
    for iteration in tqdm(range(maxit)):
        r = EHb + mu*(u+bu)
        m_ = invLS(r, mu)
        u = Psi(m_-bu, threshold)
        bu += (u-m_)
        mse_history.append(MSE(m_,m_gold))
    return m_, mse_history

def E_c(U, C, m):
    return U[:,:,np.newaxis]*fftshift(fft2(ifftshift(C*m[:, :, np.newaxis]), axes=(0, 1)))


def EH_c(C, b):
    return np.sum(fftshift(ifft2(ifftshift(b), axes=(0, 1)))
                  * C.conj(), axis=2)

def invLS_c_UVDR(m, mu):
    return (m-1/(1+mu)*EH_c(C, E_c(UVDR, C, m)))/mu


def invLS_c_UR(m, mu):
    return (m-(1/(1+mu))*EH_c(C, E_c(UR, C, m)))/mu

img_recon_UR, mse_UR = SALSA_c(b_c_UR, C, m, EH_c, Psi, PH,
                       invLS_c_UR, threshold=1e-3, mu=3, maxit=100)
img_recon_UVDR, mse_UVDR = SALSA_c(b_c_UVDR, C, m, EH_c, Psi, PH,
                       invLS_c_UVDR, threshold=5e-3, mu=2, maxit=100)

plt.figure(figsize=(13, 10))
plt.subplot(2, 2, 1)
plt.imshow(np.abs(img_recon_UR), cmap='gray')
plt.colorbar()
plt.title("SALSA coiled Reconstructed Image (UR)")

plt.subplot(2, 2, 2)
plt.plot(mse_UR)
plt.title("SALSA coiled MSE - Iteration (UR)")

plt.subplot(2, 2, 3)
plt.imshow(np.abs(img_recon_UVDR), cmap='gray')
plt.colorbar()
plt.title("SALSA coiled Reconstructed Image (UVDR)")

plt.subplot(2, 2, 4)
plt.plot(mse_UVDR)
plt.title("SALSA coiled MSE - Iteration (UVDR)")
```

```
100%|██████████| 100/100 [00:14<00:00,  6.93it/s]
100%|██████████| 100/100 [00:14<00:00,  7.04it/s]

Text(0.5, 1.0, 'SALSA coiled MSE - Iteration (UVDR)')
```



To combine Compressed Sensing and Parallel Imaging, we modify the SALSA algorithm by adding a coil sensitivity map to the forward operator $E = UFC$, and the corresponding inverse operator $E^H = C^H F^H$.

The reconstructed images and MSE-iteration curves for the UR and UVDR undersampling patterns are depicted above. In comparison with the single coil SALSA reconstruction, the multi-coil SALSA reconstruction converges much faster and reaches a better reconstruction performance. For the UVDR pattern, the multi-coil SALSA reconstruction reaches a MSE value of 0.0017 after about 20 iterations.

In comparison with the iterative SENSE reconstruction, the multi-coil SALSA reconstruction converges a little bit slower than the iterative SENSE reconstruction, but it reaches a lower MSE. Based on the MSE-iteration curves, the multi-coil SALSA reconstruction reaches a MSE value of 0.0017 after about 20 iterations, while the iterative SENSE reconstruction reaches a MSE value of 0.002 after about 20 iterations. Based on visual inspection, stripe artifacts are less significant for the multi-coil SALSA reconstruction than the iterative SENSE reconstruction.