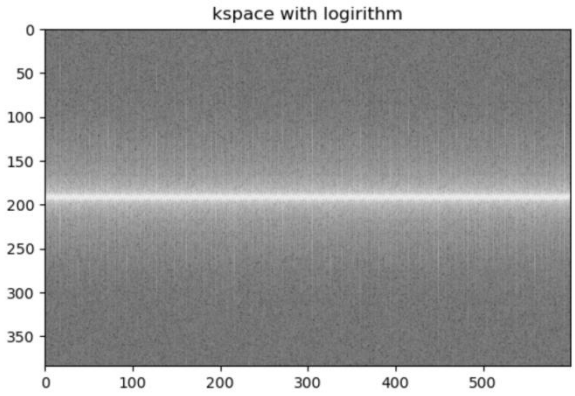
Computational MR imaging

Lab 4: Reconstruction of non-Cartesian k-space data

Nan Lan

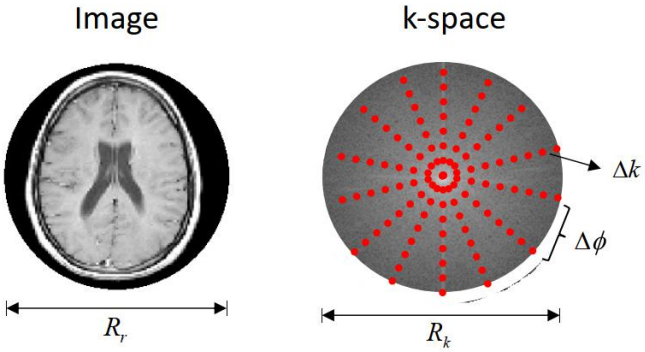
1. Radial sampling pattern

The result below shows the kspace with logorithm. The shape of kspace is 384×600. There are 600 radial lines. Each column corresponds to the readout dimension for each radial line.



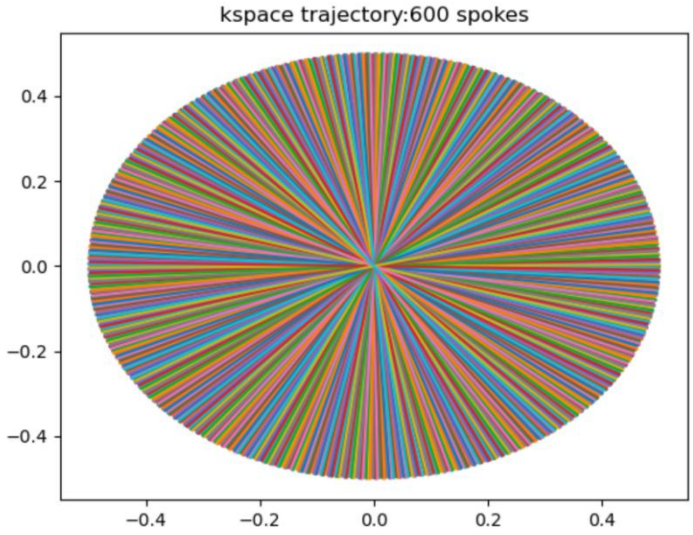
~~If the matrix size for Cartesian imaging is 384x384, the number of radial lines corresponding to the Nyquist rate is 600~~.

According to the Nyquist rate(refer to the screenshot below), N_radial = pai/2*N_cartesian=pai/2*384≈603. (△k is the max sampling distance in kspace)



Nyquist rate: $N_{radial} = \frac{\pi}{2} N_{Cartesian}$

The image below shows a sampling trajectory that corresponds to the above kspace data for the reconstruction. This radial trajectory is generated with golden angle increment(111.246117975°) and the first angle is at pi/2.



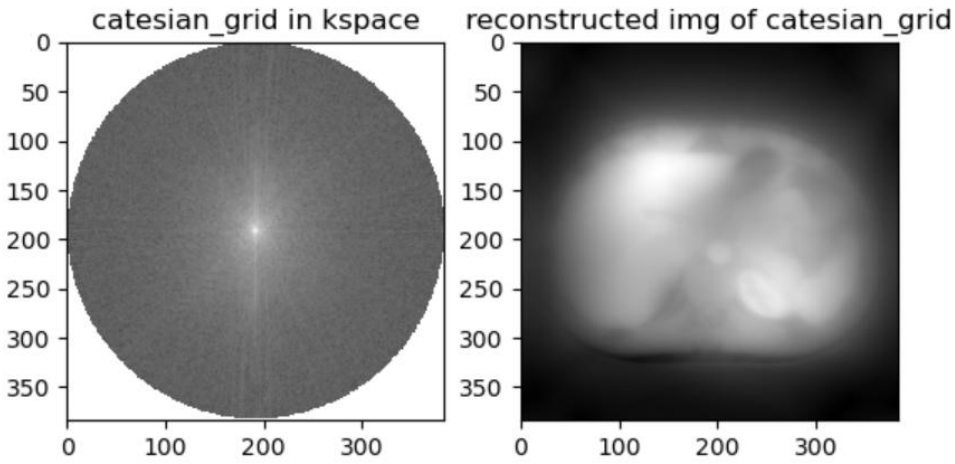
The process of this ratial trajectory is as follow:

```
#######################################################
###### 1.Radial sampling pattern ######
#######################################################
def radial_trajectory(delta_ang, start_ang, kspace):
    lenth, num_lin = np.shape(kspace) #384,600
    sample_points = np.zeros_like(kspace,dtype=complex) # 注意dtype，否则下方的语句执行后Complex_Mat中只存放实数

    for j in range(num_lin):  # 600
        angle = np.deg2rad(j * delta_ang + start_ang)

        for l in range(lenth):  # 384
            x = (l - (lenth-1)/2) * np.cos(angle)/lenth # scale from -0.5 to 0.5
            y = (l - (lenth-1)/2) * np.sin(angle)/lenth
            sample_points[l, j] = x+1j*y

        plt.plot([np.real(sample_points[0, j]), np.real(sample_points[-1, j])],
                 [np.imag(sample_points[0, j]), np.imag(sample_points[-1, j])])

    return sample_points

golden_angle_increment = 111.246117975
trajectory = radial_trajectory(golden_angle_increment, 90, kspace)
```

2. Basic gridding reconstruction

The image below shows the Cartesian in kspace and the corresponding reconstructed image, based on the grid function. The grid function grids 2D non-Cartesian k-space data to Cartesian k-space data using triangular gridding kernel of width 2.

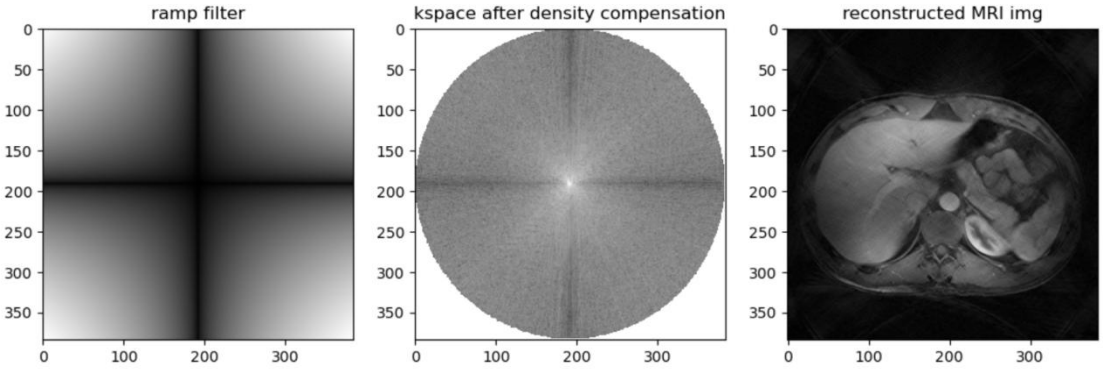
The reconstructed MRI image is the abdomen along the transverse plane. The reconstructed MRI image is quite blur. The reason is that the center region of kspace is oversampling, if radial trajectory is applied in kspace.



3. Density compensation

Density compensation is to deal with the problem of oversampling in kspace center. The images below show the kspace after density compensation, using 2D ramp filter. The reconstructed MRI image has obviously higher spatial resolution after density compensation.

Or first convolute the ungridding kspace with ramp filter, then map to Cartesian grid(This will be more computational efficient)



The process of the density compensation is as follow:

1) Create a 2D density filter

```
kp_sz = len(kspace)
ramp1D_half = np.linspace(1,1/kp_sz,kp_sz//2)
ramp1D = np.concatenate((ramp1D_half, ramp1D_half[::-1]))
ramp2D = np.sqrt(np.outer(ramp1D, ramp1D))
```

2) Multiply kspace with density filter and do inverse Fourier transform

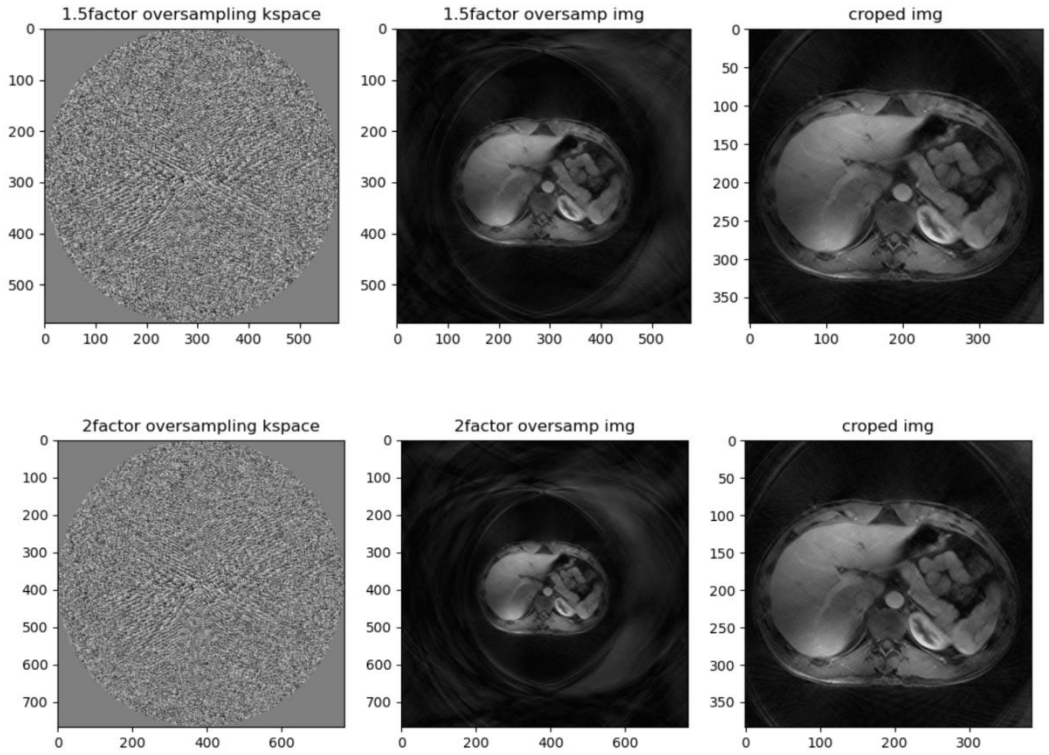
```
kspace_denComp = kspace * ramp2D
img_denComp = ifft2c(kspace_denComp)
```

4. Oversampling

Field of View(FoV) is inversely proportional to the interval of kspace (Fov ~ 1/△k). Oversampling increase the sampling points, which means decrease the interval of kspace. Therefore, oversampling will produce a larger field of view, which also require more data to be stored and processed.

But oversampling moves the replica sidelobes out after cropping the reconstructed MRI image, reducing aliasing, and allowing less apodization.

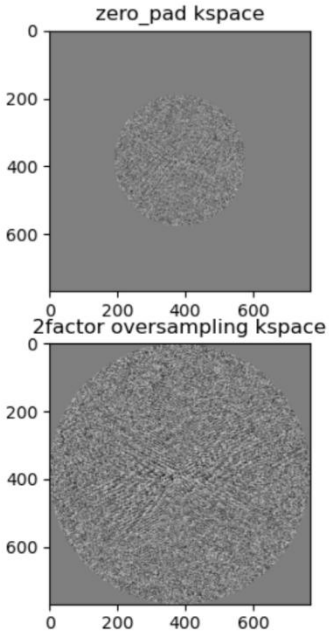
The results below show the kspace, MRI image and crop MRI image, with oversampling factors of 1.5 and 2. The Fov increase with factors of 1.5 and 2 correspondingly.

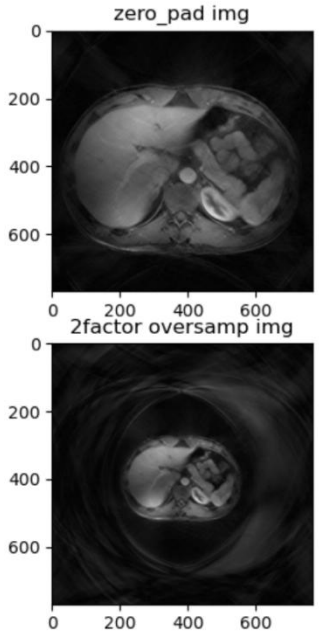


PS: Comparison between oversampling and zero padding

The results below show the comparison between oversampling and zero padding.

Oversampling increases FoV, but zeropadding won't increase Fov, because zeropadding doesn't increase the sampling points i kspace.



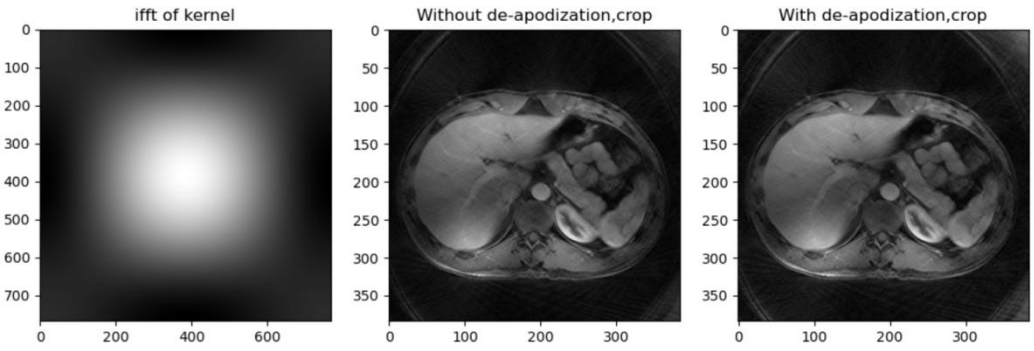


5. De-apodization

When the non-cartesian gridding is mapped to the basic cartesian gridding through convolution with the gridding kernel, apodization arises. Generally, apodization reduces the resolution of an optical image; however, it also reduces side lobes in kspace.

Deapodization can weaken the influence of apodization.

The results below show the inverse Fourier transform of the gridding kernel and the effect of deapodization. After the de-apodization, the reconstructed MRI image have slightly higher resolution.



The process of the density compensation is as follow:

1) Create 2D gridding triangle kernel (the same as previous kernel) and zero pading to the same shape of reconstructed image.

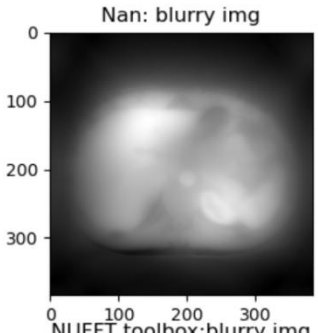
```
tri1D = [0, 0.5, 1, 0.5, 0]
tri2D = np.sqrt(np.outer(tri1D, tri1D))
pad_w = (sz_overSam2-len(tri1D))//2
tri2D_pad = np.pad(tri2D, ((pad_w+1,pad_w), (pad_w+1,pad_w)), 'constant', constant_values=0)
```

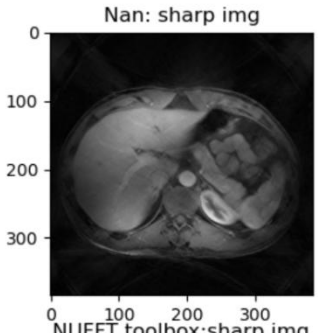
2) Divide the reconstructed image by the inverse Fourier transform of the gridding kernel.

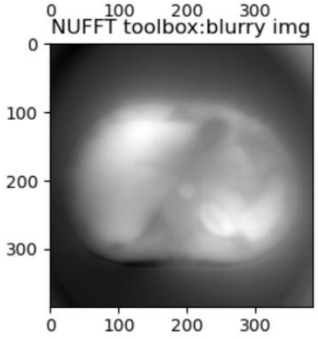
```
tri_ift = ifft2c(tri2D_pad)+10**(-5)
img_deApo = img_overSam2/tri_ift
```

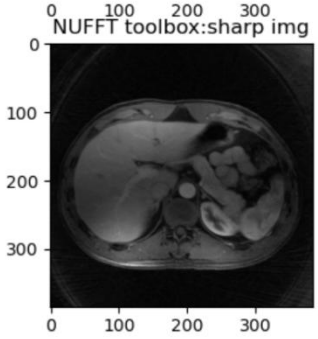
6. NUFFT toolbox

The results below show the reconstructed MRI image by myself and by Torch KB-NUFFT toolbox. This toolbox implements a non-uniform Fast Fourier Transform with Kaiser-Bessel gridding in PyTorch. The sharp image has the additional density compensation step.









The process is as follow:

```
spokelength, nspokes = np.shape(kspace_radial)
ga = np.deg2rad(golden_angle_increment)
kx = np.zeros(shape=(spokelength, nspokes))
ky = np.zeros(shape=(spokelength, nspokes))
ky[:, 0] = np.linspace(-np.pi, np.pi, spokelength)
for i in range(1, nspokes):
    kx[:, i] = np.cos(ga) * kx[:, i - 1] - np.sin(ga) * ky[:, i - 1]
    ky[:, i] = np.sin(ga) * kx[:, i - 1] + np.cos(ga) * ky[:, i - 1]
ky = np.transpose(ky)
kx = np.transpose(kx)

ktraj = np.stack((ky.flatten(), kx.flatten()), axis=0)
ktraj = torch.tensor(ktraj)

kdata = kspace_radial.transpose()
kdata = kdata.reshape((1,-1))
kdata = torch.tensor(kdata).unsqueeze(0)

adjnufft_ob = tkbn.KbNufftAdjoint(im_size=(sz_crop,sz_crop))

image = adjnufft_ob(kdata, ktraj)
```