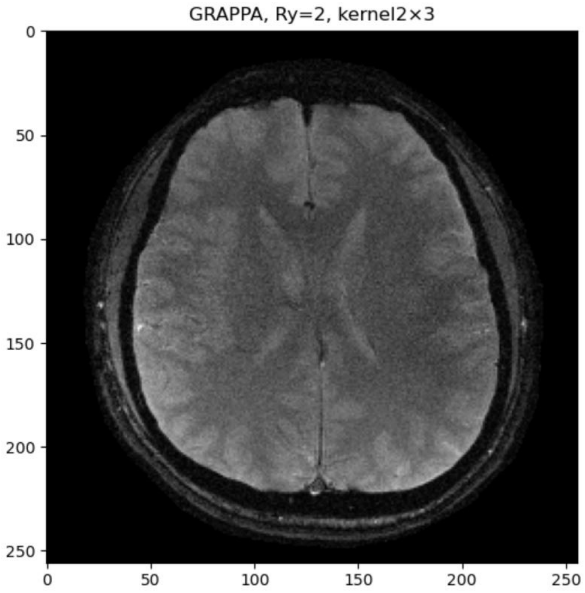
Computational MR imaging

Laboratory 6: k-space parallel imaging

Nan Lan

1. Simple GRAPPA reconstruction

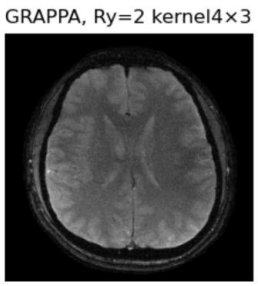
The results below shows simple grappa reconstruction, where acceleration factor Ry=2(in column direction), kernel size is 2x3.

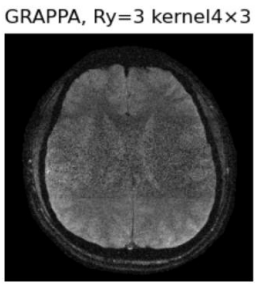


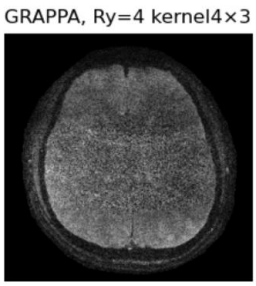
2. Modified GRAPPA reconstruction

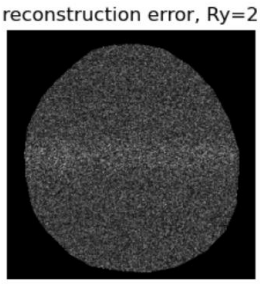
The results below shows simple grappa reconstruction, where acceleration factor Ry=2,3,4 (in column direction), kernel size is 4x3.
Reconstruction error is calculated through:

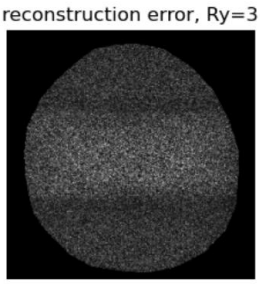
```
error_img = img_GRAPPA-img_least_square
```

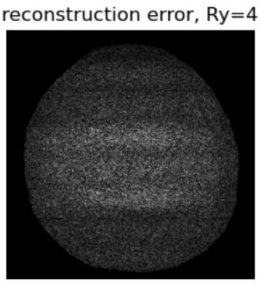












RMSE:

RMSE is calculated through:

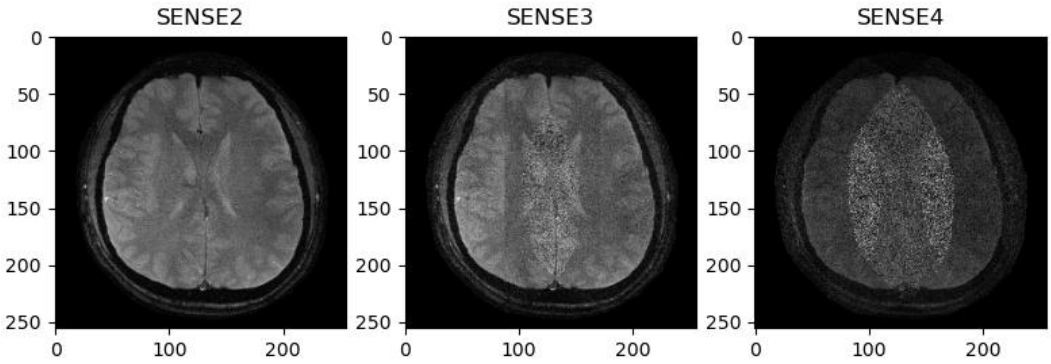
```
RMSE = np.sqrt(np.sum((np.abs(img_least_square)-np.abs(img_GRAPPA))**2)/np.size(img_GRAPPA))
```

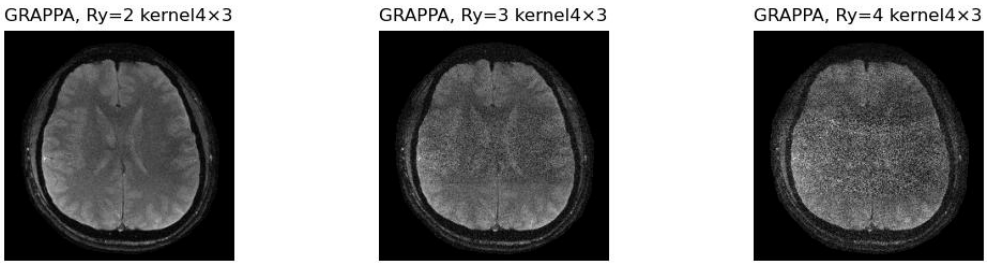
The result of RMSE is:

```
sum of reconstruction error(R=2) is  7.021437157439958
Root Mean Square Error(R=2) is 0.00011
sum of reconstruction error(R=3) is  19.15072551436612
Root Mean Square Error(R=3) is 0.00029
sum of reconstruction error(R=4) is  35.0618272491588
Root Mean Square Error(R=4) is 0.00054
```

From the GRAPPA reconstructed image and the RMSE, we can see that the noise and error increase with higher accelaration factor(R). The noise of GRAPPA mainly locate in the center of the brain region.

Comparison between SENSE and GRAPPA:





SENSE method reconstructs images from the individual coils and correction is performed in the spatial/image domain. GRAPPA method operate in the frequency domain; data is corrected in k-space before reconstruction.

The image quality of both algorithm are similar, when R=2,R=3. GRAPPA has a slight better reconstruction result than SENSE, when R =4.

The process of grappa is as follow:

(1) Create the undersampled kspace and the Autocalibration signal

```
def get_zf_kspace(R, fs_kdata):
    """""
    Input:
        R: acceleration factor
        fs_kdata: fully sampled kdata. (nx, ny, nc)
    Output:
        zf_kspace:fill every other line to zero, if R=2 (nx, ny, nc)
    """""
    zf_kspace = np.zeros_like(fs_kdata)
    for i in range(nx):
        if i % R == 0:  # even
            zf_kspace[i] = fs_kdata[i]
    return zf_kspace
```

```
def get_acs(num_cenLine, fs_kdata):
    acs = fs_kdata[(nx - num_cenLine) // 2: (nx + num_cenLine) // 2]
    return acs
```

(2) get source and target in ACS region

```
def extract(self):
    '''''
    from acs extract source and target
    '''''
    src = np.zeros((self.R-1, self.nb, self.nc * self.nk), dtype=np.complex64)
    targ = np.zeros((self.R-1, self.nb, self.nc), dtype=np.complex64)

    for i in range(self.R-1):
        src_idx = 0
        for col in range(self.num_col_acs - 2):
            for row in range(self.num_row_acs-(self.block_height-1)):  # for循环所有target点(nb)

                block = self.acs[row:row+self.block_height, col:col+self.block_width]

                src[i,src_idx] = block[::self.R].flatten()
                targ[i,src_idx] = acs[row+self.row_offset[i], col+1]

                src_idx += 1

    return src, targ
```

(3) calculate weight, based on the equation: target=source*weights

```
# get source and target in acs region, to calculate weight
ws = np.zeros((self.R-1,self.nk*self.nc, self.nc), dtype=complex)
src, targ = self.extract()  # 3维度
for i in range(self.R-1):
    ws[i] = np.dot(np.linalg.pinv(src[i]), targ[i])
```

(4) enlarge kspace with zero-pad, so that the data in the boundary can also be calculated

```
zp_up = nxacs-2
zp_dn = nxacs-1
zp_kdata = np.zeros((nx + zp_up+zp_dn, ny + 2, nc), np.complex64)
zp_kdata[zp_up:nx+zp_up, 1:ny + 1, :] = kdata
```

(5) Interpolate the lacking datapoint in the kspace

```
def interp(self, zp_kspace, ws):
    interpolated = np.array(zp_kspace)
    num_row_zpk, num_col_zpk, nc = zp_kspace.shape

    for i in range(self.R-1):
        src_idx = 0
        for col in range(num_col_zpk - 2):
            for row in range(0, num_row_zpk-(self.block_height-1), self.R):  # for循环所有target点(nb)

                block = zp_kspace[row:row+self.block_height, col:col+self.block_width]
                src = block[::self.R].flatten()

                interpolated[row+self.row_offset[i], col+1] = np.dot(src, ws[i])

                src_idx += 1

    return interpolated
```

(6) Get the kspace from the enlarged-reconstructed kspace.

```
# interpolation
interpolated = self.interp(zp_kdata, ws)
interpolated = interpolated[zp_up:self.num_row_zfKspace+zp_up, 1:self.num_col_zfKspace + 1, :]
```

(7) Use least square algorithm to combine the coil kspaces and return the reconstructed MRI image.

```
def get_Img(inpo_ksp):
    """
    combine kspace in diff coil and reconstruct image
    """
    m_coil_img = np.zeros_like(inpo_ksp)
    for i in range(nc):
        m_coil_img[..., i] = ifft2c(inpo_ksp[..., i])
    img_least_square, _ = least_square(m_coil_img, c_coil_sen)
    return img_least_square
```