Part 1: Simple plotting with matplotlib

a) Plotting the middle z-slice for each image type

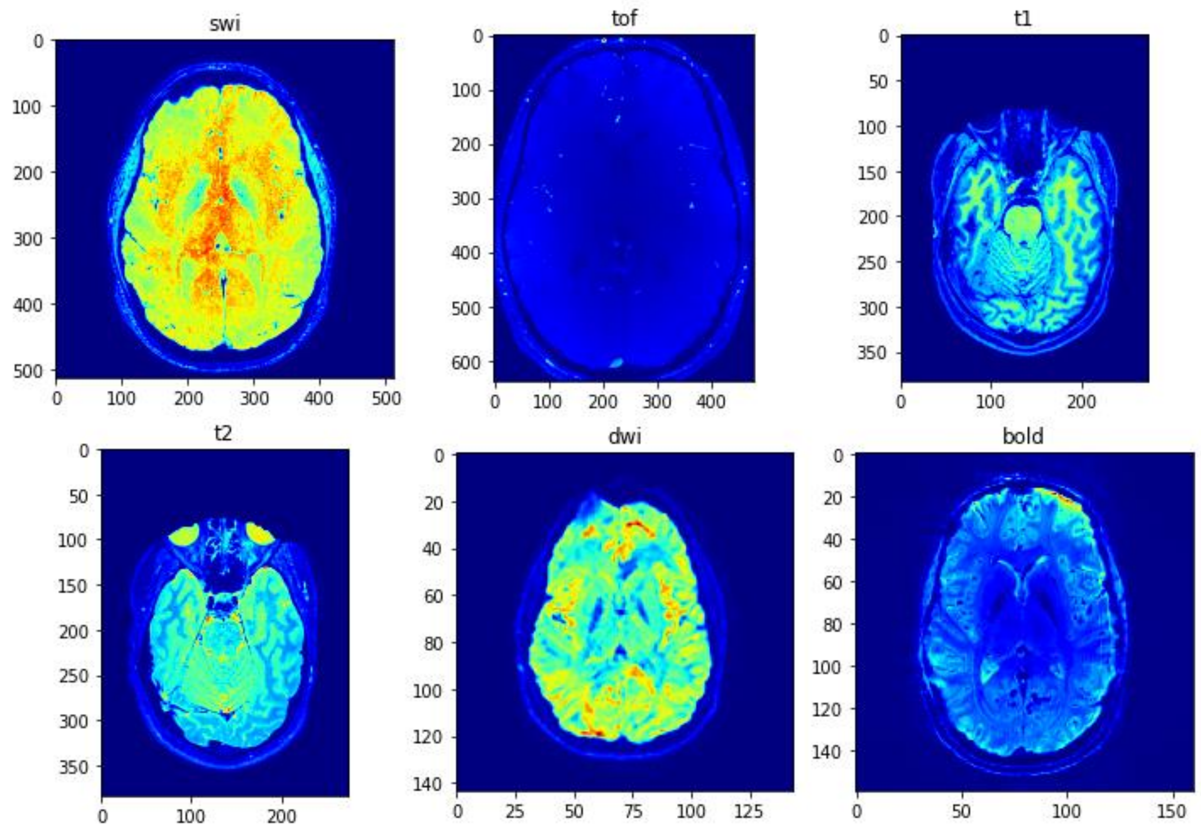
In this task, we are loading each nii image and taking the middle z-slice for them using numpy and plotting them using matplotlib.

```
import nibabel as nib
import matplotlib.pyplot as plt
import numpy as np

# Load the nii image
nii_image_swi = nib.load(
    "images/sub-01_ses-forrestgump_anat_sub-01_ses-forrestgump_acq-
mag_veno.nii"
)
nii_image_tof = nib.load(
    "images/sub-01_ses-forrestgump_anat_sub-01_ses-forrestgump_angio.nii"
)
nii_image_t1 = nib.load(
    "images/sub-01_ses-forrestgump_anat_sub-01_ses-forrestgump_T1w.nii"
)
nii_image_t2 = nib.load(
    "images/sub-01_ses-forrestgump_anat_sub-01_ses-forrestgump_T2w.nii"
)
nii_image_dwi = nib.load(
    "images/sub-01_ses-forrestgump_dwi_sub-01_ses-forrestgump_dwi.nii"
)
nii_image_bold = nib.load(
    "images/sub-01_ses-forrestgump_func_sub-01_ses-forrestgump_task-
forrestgump_acq-dico_run-01_bold.nii"
)

# Get the image data from nii_image
swi_img = nii_image_swi.get_fdata()
tof_img = nii_image_tof.get_fdata()
t1_img = nii_image_t1.get_fdata()
t2_img = nii_image_t2.get_fdata()
dwi_img = nii_image_dwi.get_fdata()
bold_img = nii_image_bold.get_fdata()

# Slicing the middle z-slice for swi image and formatting it
swi_slice_image = np.rot90(swi_img[:, :, swi_img.shape[2] // 2])

# Slicing the middle z-slice for tof image and formatting it
tof_slice_image = np.fliplr(
    np.rot90(np.transpose(tof_img[:, :, tof_img.shape[2] // 2]), 2)
)
```

```
# Slicing the middle z-slice for t1 image and formatting it
t1_slice_image = np.fliplr(
    np.rot90(np.transpose(t1_img[:, :, t1_img.shape[2] // 2]), 2)
)

# Slicing the middle z-slice for t2 image and formatting it
t2_slice_image = np.fliplr(
    np.rot90(np.transpose(t2_img[:, :, t2_img.shape[2] // 2]), 2)
)

# Slicing the middle z-slice for dwi image and formatting it
dwi_slice_image = np.fliplr(
    np.rot90(
        np.transpose(dwi_img[:, :, dwi_img.shape[2] // 2, dwi_img.shape[3] //
2]), 2
    )
)

# Slicing the middle z-slice for bold image and formatting it
bold_slice_image = np.fliplr(
    np.rot90(
        np.transpose(bold_img[:, :, bold_img.shape[2] // 2, bold_img.shape[3]
// 2]), 2
    )
)

# Plot the figure
fig = plt.figure(figsize=(10, 7))

# Plotting swi
fig.add_subplot(2, 3, 1)
plt.imshow(swi_slice_image, cmap="jet")
plt.title("swi")

# Plotting tof
fig.add_subplot(2, 3, 2)
plt.imshow(tof_slice_image, cmap="jet")
plt.title("tof")

# Plotting t1
fig.add_subplot(2, 3, 3)
plt.imshow(t1_slice_image, cmap="jet")
plt.title("t1")

# Plotting t2
fig.add_subplot(2, 3, 4)
plt.imshow(t2_slice_image, cmap="jet")
plt.title("t2")
```

```
# Plotting dwi
fig.add_subplot(2, 3, 5)
plt.imshow(dwi_slice_image, cmap="jet")
plt.title("dwi")

# Plotting bold
fig.add_subplot(2, 3, 6)
plt.imshow(bold_slice_image, cmap="jet")
plt.title("bold")


plt.tight_layout()
plt.show()
```

Result:

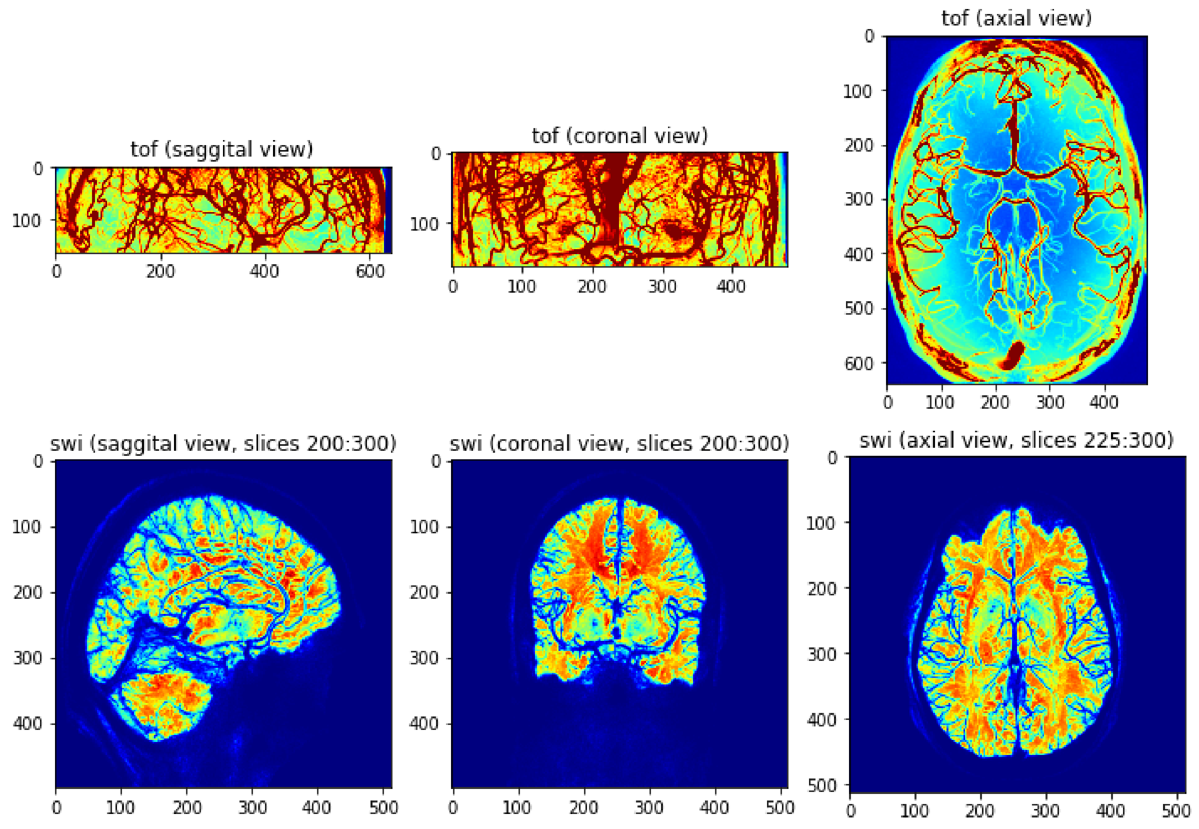


b) Plotting the maximum intensity projection and minimum intensity projection for TOF and SWI

For this question, we have loaded the tof and swi image. For tof, we have projected the maximum intensity projection (np.max) using vmax=300. For swi, we have projected the minimum intensity projection (np.min) taking the image slices ~200 to 300 slices.

```
import numpy as np
import matplotlib.pyplot as plt
import nibabel as nib

# Load the nii image
nii_image_swi = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_acq-mag_veno.nii')
nii_image_tof = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_angio.nii')

# Get the image data from nii_image
swi_img = nii_image_swi.get_fdata()
tof_img = nii_image_tof.get_fdata()

# Plot the figure
fig = plt.figure(figsize=(10, 7))

# Plot maximum intensity projections for tof image
fig.add_subplot(2, 3, 1)
plt.imshow(np.rot90(np.max(tof_img, axis=0)), cmap='jet', vmax=300)
plt.title('tof (saggital view)')

fig.add_subplot(2, 3, 2)
plt.imshow(np.rot90(np.max(tof_img, axis=1)), cmap='jet', vmax=300)
plt.title('tof (coronal view)')

fig.add_subplot(2, 3, 3)
plt.imshow(np.rot90(np.max(tof_img, axis=2)), cmap='jet', vmax=300)
plt.title('tof (axial view)')

# Plot minimum intensity projections for swi image
fig.add_subplot(2, 3, 4)
plt.imshow(np.rot90(np.min(swi_img[200:300,:,:], axis=0)), cmap='jet')
plt.title('swi (saggital view, slices 200:300)')

fig.add_subplot(2, 3, 5)
plt.imshow(np.rot90(np.min(swi_img[:,200:300,:], axis=1)), cmap='jet')
plt.title('swi (coronal view, slices 200:300)')

fig.add_subplot(2, 3, 6)
plt.imshow(np.rot90(np.min(swi_img[:,:,225:300], axis=2)), cmap='jet')
plt.title('swi (axial view, slices 225:300)')

plt.tight_layout()
plt.show()
```

Result:


tof (saggital view)
tof (coronal view)
tof (axial view)
swi (saggital view, slices 200:300)
swi (coronal view, slices 200:300)
swi (axial view, slices 225:300)

Part 2: Contrast estimation

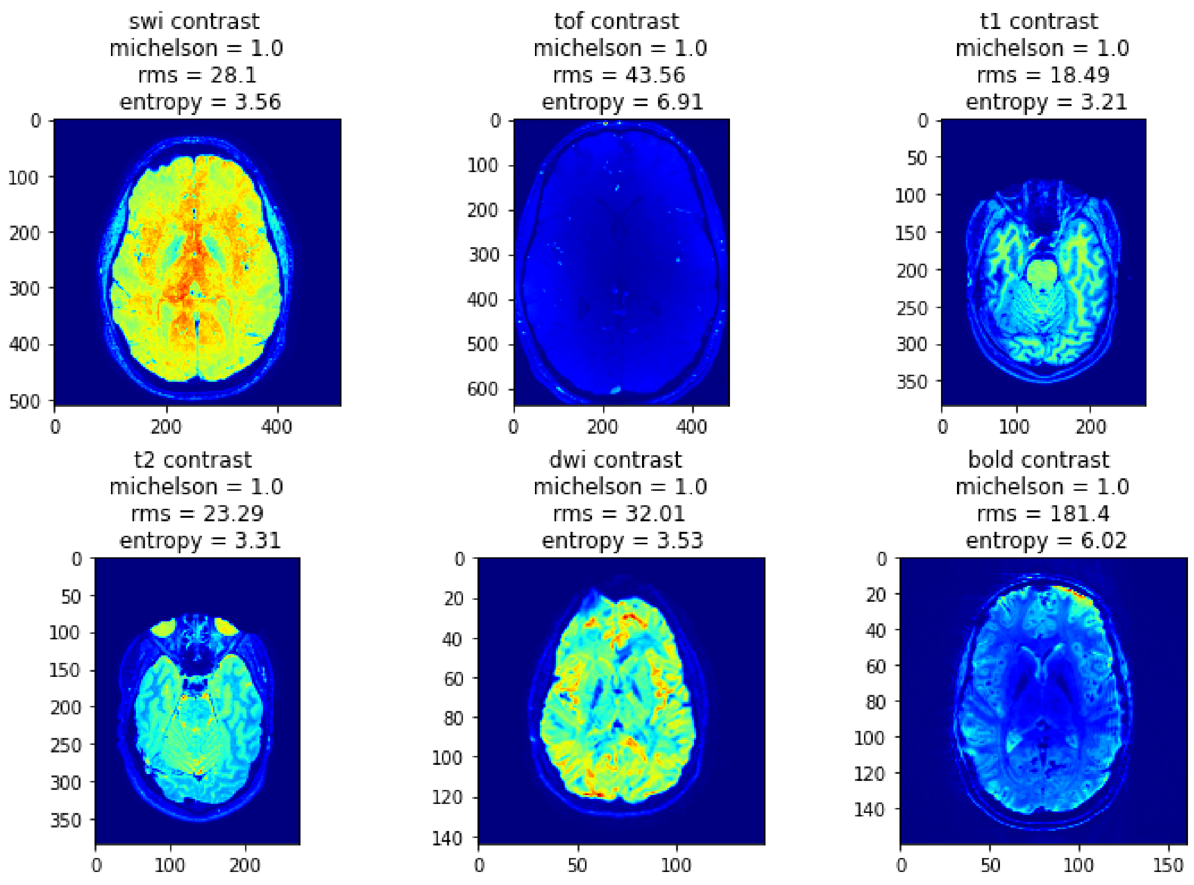
As part of contrast estimation, we have plotted each image view with contrast estimates generated using Michelson, root mean square and entropy methods. Note: Contrast measures are calculated using the entire 3D or 4D image and only for projections the slice of the image is used.

```python
import nibabel as nib
import matplotlib.pyplot as plt
import numpy as np

# Load the nii image
nii_image_swi = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-forrestgump_acq-mag_veno.nii')
nii_image_tof = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-forrestgump_angio.nii')
nii_image_t1 = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-forrestgump_T1w.nii')
```

```
nii_image_t2 = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_T2w.nii')
nii_image_dwi = nib.load('images/sub-01_ses-forrestgump_dwi_sub-01_ses-
forrestgump_dwi.nii')
nii_image_bold = nib.load('images/sub-01_ses-forrestgump_func_sub-01_ses-
forrestgump_task-forrestgump_acq-dico_run-01_bold.nii')

# Get the image data from nii_image
swi_img = nii_image_swi.get_fdata()
tof_img = nii_image_tof.get_fdata()
t1_img = nii_image_t1.get_fdata()
t2_img = nii_image_t2.get_fdata()
dwi_img = nii_image_dwi.get_fdata()
bold_img = nii_image_bold.get_fdata()


# Slicing the middle z-slice for swi image and formatting it
swi_slice_image = np.rot90(swi_img[:, :, swi_img.shape[2]//2])

# Slicing the middle z-slice for tof image and formatting it
tof_slice_image =  np.fliplr(np.rot90(np.transpose(tof_img[:, :,
tof_img.shape[2]//2]),2))

# Slicing the middle z-slice for t1 image and formatting it
t1_slice_image = np.fliplr(np.rot90(np.transpose(t1_img[:, :,
t1_img.shape[2]//2]),2))

# Slicing the middle z-slice for t2 image and formatting it
t2_slice_image = np.fliplr(np.rot90(np.transpose(t2_img[:, :,
t2_img.shape[2]//2]),2))

# Slicing the middle z-slice for dwi image and formatting it
dwi_slice_image = np.fliplr(np.rot90(np.transpose(dwi_img[:, :,
dwi_img.shape[2]//2, dwi_img.shape[3]//2]),2))

# Slicing the middle z-slice for bold image and formatting it
bold_slice_image =  np.fliplr(np.rot90(np.transpose(bold_img[:, :,
bold_img.shape[2]//2, bold_img.shape[3]//2]),2))

def calculate_michelson_contrast(img):
    return (np.max(img) - np.min(img)) / np.max(img) + np.min(img)


def calculate_rms_contrast(img):
    return round(np.std(img), 2)


def calculate_entropy_contrast(img):
    histogram, bins = np.histogram(img.flatten(), bins=256, range=(0, 255))
    normalized_histogram = histogram / np.sum(histogram)
    entropy_calculated = -np.sum(
        normalized_histogram
        * np.log2(normalized_histogram, where=(normalized_histogram != 0))
    )
    return round(entropy_calculated, 2)


# Plot the figure
fig = plt.figure(figsize=(10, 7))

# Plotting swi
fig.add_subplot(2, 3, 1)
plt.imshow(swi_slice_image, cmap="jet")
plt.title(
    f"swi contrast \n michelson = {calculate_michelson_contrast(swi_img)} \n
rms = {calculate_rms_contrast(swi_img)} \n entropy =
{calculate_entropy_contrast(swi_img)}"
)

# Plotting tof
fig.add_subplot(2, 3, 2)
plt.imshow(tof_slice_image, cmap="jet")
plt.title(
    f"tof contrast \n michelson = {calculate_michelson_contrast(tof_img)} \n
rms = {calculate_rms_contrast(tof_img)} \n entropy =
{calculate_entropy_contrast(tof_img)}"
)

# Plotting t1
fig.add_subplot(2, 3, 3)
plt.imshow(t1_slice_image, cmap="jet")
plt.title(
    f"t1 contrast \n michelson = {calculate_michelson_contrast(t1_img)} \n rms
= {calculate_rms_contrast(t1_img)} \n entropy =
{calculate_entropy_contrast(t1_img)}"
)

# Plotting t2
fig.add_subplot(2, 3, 4)
plt.imshow(t2_slice_image, cmap="jet")
plt.title(
    f"t2 contrast \n michelson = {calculate_michelson_contrast(t2_img)} \n rms
= {calculate_rms_contrast(t2_img)} \n entropy =
{calculate_entropy_contrast(t2_img)}"
)

# Plotting dwi
fig.add_subplot(2, 3, 5)
```

```
plt.imshow(dwi_slice_image, cmap="jet")
plt.title(
    f"dwi contrast \n michelson = {calculate_michelson_contrast(dwi_img)} \n
rms = {calculate_rms_contrast(dwi_img)} \n entropy =
{calculate_entropy_contrast(dwi_img)}"
)

# Plotting bold
fig.add_subplot(2, 3, 6)
plt.imshow(bold_slice_image, cmap="jet")
plt.title(
    f"bold contrast \n michelson = {calculate_michelson_contrast(bold_img)} \n
rms = {calculate_rms_contrast(bold_img)} \n entropy =
{calculate_entropy_contrast(bold_img)}"
)

plt.tight_layout()
plt.show()
```

Result:



Part 3 : SNR estimation, quantifying noise

```python
import numpy as np
import matplotlib.pyplot as plt
import nibabel as nib

# Load the nii image
nii_image_swi = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_acq-mag_veno.nii')
nii_image_tof = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_angio.nii')
nii_image_t1 = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_T1w.nii')
nii_image_t2 = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_T2w.nii')
nii_image_dwi = nib.load('images/sub-01_ses-forrestgump_dwi_sub-01_ses-
forrestgump_dwi.nii')
nii_image_bold = nib.load('images/sub-01_ses-forrestgump_func_sub-01_ses-
forrestgump_task-forrestgump_acq-dico_run-01_bold.nii')

# Get the image data from nii_image
swi_img = nii_image_swi.get_fdata()
tof_img = nii_image_tof.get_fdata()
t1_img = nii_image_t1.get_fdata()
t2_img = nii_image_t2.get_fdata()
dwi_img = nii_image_dwi.get_fdata()
bold_img = nii_image_bold.get_fdata()

image_list = {"swi_img":swi_img, "tof_img":tof_img, "t1_img":t1_img,
"t2_img":t2_img, "dwi_img":dwi_img, "bold_img":bold_img}


def plot_image(image,title):
    # Plotting the image
    plt.imshow(image)
    plt.title(title)


def plot_signal_patch(image,signal_patch,plt1,plt2, text_x, text_y):
    # Plotting the signal patch
    signal_patch_data = image[signal_patch]
    plt.imshow(signal_patch_data, cmap='viridis', alpha=0.5)
    plt.plot(plt1, plt2, 'r-', linewidth=2)  # Rectangle border
    plt.text(text_x, text_y, 'Signal', color='red', fontsize=12)
    return signal_patch_data

def plot_noise_patch(image,noise_patch,plt1,plt2, text_x, text_y):
    # Plotting the noise patch
    noise_patch_data = image[noise_patch]
    plt.imshow(noise_patch_data, cmap='cool', alpha=0.5)
    plt.plot(plt1, plt2, 'b-', linewidth=2)  # Rectangle border
    plt.text(text_x, text_y, 'Noise', color='blue', fontsize=12)
    return noise_patch_data

def calculate_SNR_formula(signal_patch_data, noise_patch_data):
    signal_mean = np.mean(signal_patch_data)
    noise_std = np.std(noise_patch_data)
    snr = signal_mean / noise_std
    return snr

def print_coordinates(image_name,signal_patch,noise_patch):
    # Printing the coordinates of signal and noise patches
    signal_coords = np.array(signal_patch)
    noise_coords = np.array(noise_patch)
    print(f'Signal Patch Coordinates for image {image_name} :')
    print('X:', signal_coords[0], 'Y:', signal_coords[1], 'Z:',
signal_coords[2])
    print('Noise Patch Coordinates:')
    print('X:', noise_coords[0], 'Y:', noise_coords[1], 'Z:', noise_coords[2])

def calculate_SNR(image_name,image):
    if image_name == "swi_img":
        # Defining the signal and noise patches
        signal_patch = np.s_[190:240, 140:220, 250]
        noise_patch = np.s_[90:110, 340:360, 250]
        print_coordinates(image_name,signal_patch,noise_patch)
        signal_patch_data = plot_signal_patch(swi_img,signal_patch, [140, 140,
219, 219, 140], [190, 239, 239, 190, 190], 140, 188)
        noise_patch_data = plot_noise_patch(swi_img, noise_patch, [340, 340,
359, 359, 340], [90, 109, 109, 90, 90], 340, 88)
        snr =
round(calculate_SNR_formula(signal_patch_data,noise_patch_data),2)
        print(f'SNR for {image_name} is {snr}')
        plot_image(swi_img[:, :, 250],f'SW1 (acq-mag_veno) Image, snr =
{snr}')

    elif image_name == "tof_img":
        # Defining the signal and noise patches
        signal_patch = np.s_[110:145, 102:110, 80]
        noise_patch = np.s_[416:424, 532:550, 80]
        print_coordinates(image_name,signal_patch,noise_patch)
        signal_patch_data = plot_signal_patch(tof_img,signal_patch, [102, 102,
109, 109, 102], [110, 144, 144, 110, 110], 102, 108)
        noise_patch_data = plot_noise_patch(tof_img, noise_patch, [532, 532,
549, 549, 532], [416, 423, 423, 416, 416], 532, 414)
        snr =
round(calculate_SNR_formula(signal_patch_data,noise_patch_data),2)
        print(f'SNR for {image_name} is {snr}')
        plot_image(tof_img[:, :, 80],f'TOF Image, snr = {snr}')

    elif image_name == "t1_img":
        # Defining the signal and noise patches
        signal_patch = np.s_[70:95, 90:115, 250]
        noise_patch = np.s_[223:228, 261:267, 250]
        print_coordinates(image_name,signal_patch,noise_patch)
        signal_patch_data = plot_signal_patch(t1_img,signal_patch, [90, 90,
115, 115, 90], [70, 95, 95, 70, 70], 90, 68)
        noise_patch_data = plot_noise_patch(t1_img, noise_patch, [261, 261,
267, 267, 261], [223, 228, 228, 223, 223], 261, 221)
        snr =
round(calculate_SNR_formula(signal_patch_data,noise_patch_data),2)
        print(f'SNR for {image_name} is {snr}')
        plot_image(t1_img[:, :, 250],f'T1-weighted Image, snr = {snr}')

    elif image_name == "t2_img":
        # Defining the signal and noise patches
        signal_patch = np.s_[85:115, 190:230, 192]
        noise_patch = np.s_[38:50, 264:278, 192]
        print_coordinates(image_name,signal_patch,noise_patch)
        signal_patch_data = plot_signal_patch(t2_img,signal_patch, [190, 190,
229, 229, 190], [85, 114, 114, 85, 85], 190, 83)
        noise_patch_data = plot_noise_patch(t2_img, noise_patch, [264, 264,
277, 277, 264], [38, 49, 49, 38, 38], 264, 36)
        snr =
round(calculate_SNR_formula(signal_patch_data,noise_patch_data),2)
        print(f'SNR for {image_name} is {snr}')
        plot_image(t1_img[:, :, 250],f'T2-weighted Image, snr = {snr}')

    elif image_name == "dwi_img":
        # Defining the signal and noise patches
        signal_patch = np.s_[48:54, 102:110, 35, 25]
        noise_patch = np.s_[103:106, 107:111, 35, 25]
        print_coordinates(image_name,signal_patch,noise_patch)
        signal_patch_data = plot_signal_patch(dwi_img,signal_patch, [102, 102,
109, 109, 102], [48, 53, 53, 48, 48], 102, 46)
        noise_patch_data = plot_noise_patch(dwi_img, noise_patch, [107, 107,
110, 110, 107], [103, 105, 105, 103, 103], 107, 101)
        snr =
round(calculate_SNR_formula(signal_patch_data,noise_patch_data),2)
```

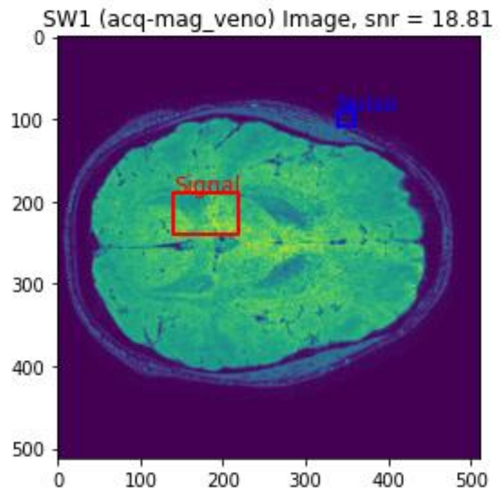
```python
        print(f'SNR for {image_name} is {snr}')
        plot_image(dwi_img[:, :, 35, 25],f'DWI Image, snr = {snr}')

    elif image_name == "bold_img":
        # Defining the signal and noise patches
        signal_patch = np.s_[70:78, 122:130, 18, 25]
        noise_patch = np.s_[111:116, 140:143, 18, 25]
        print_coordinates(image_name,signal_patch,noise_patch)
        signal_patch_data = plot_signal_patch(bold_img,signal_patch, [122,
122, 129, 129, 122], [70, 77, 77, 70, 70], 122, 68)
        noise_patch_data = plot_noise_patch(bold_img, noise_patch, [140, 140,
142, 142, 140], [111, 115, 115, 111, 111], 140, 109)
        snr =
round(calculate_SNR_formula(signal_patch_data,noise_patch_data),2)
        print(f'SNR for {image_name} is {snr}')
        plot_image(bold_img[:, :, 18, 25],f'BOLD Image, snr = {snr}')



for image_name,image in image_list.items():
    calculate_SNR(image_name,image)
    plt.tight_layout()
    plt.show()


def plot_noise_histogram(noise_patch_data,title):
    # Plot the histogram of the noise
    plt.hist(noise_patch_data.ravel(), bins=50, color='b', alpha=0.7)
    plt.title(title)
    plt.xlabel('Intensity')
    plt.ylabel('Frequency')
    plt.show()

for image_name,image in image_list.items():
    if image_name == "swi_img":
        # Define the noise patch
        noise_patch = np.s_[90:110, 340:360, 250]
        # Extract the noise data
        noise_patch_data = swi_img[noise_patch]
        plot_noise_histogram(noise_patch_data,'Histogram of Noise SW1')
    elif image_name == "tof_img":
        # Define the noise patch
        noise_patch = np.s_[416:424, 532:550, 80]
        # Extract the noise data
        noise_patch_data = tof_img[noise_patch]
        plot_noise_histogram(noise_patch_data,'Histogram of Noise TOF')
    elif image_name == "t1_img":
        # Define the noise patch
        noise_patch = np.s_[223:228, 261:267, 250]
        # Extract the noise data
        noise_patch_data = t1_img[noise_patch]
        plot_noise_histogram(noise_patch_data,'Histogram of Noise T1')
    elif image_name == "t2_img":
        # Define the noise patch
        noise_patch = np.s_[38:50, 264:278, 192]
        # Extract the noise data
        noise_patch_data = t2_img[noise_patch]
        plot_noise_histogram(noise_patch_data,'Histogram of Noise T2')
    elif image_name == "dwi_img":
        # Define the noise patch
        noise_patch = np.s_[103:106, 107:111, 35, 25]
        # Extract the noise data
        noise_patch_data = dwi_img[noise_patch]
        plot_noise_histogram(noise_patch_data,'Histogram of Noise DWI')
    elif image_name == "bold_img":
        # Define the noise patch
        noise_patch = np.s_[111:116, 140:143, 18, 25]
        # Extract the noise data
        noise_patch_data = bold_img[noise_patch]
        plot_noise_histogram(noise_patch_data,'Histogram of Noise BOLD')
```

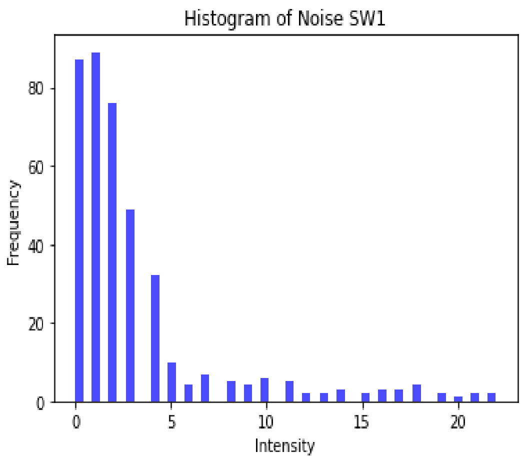
Result:

SNR for each image is calculated as per the below formula :

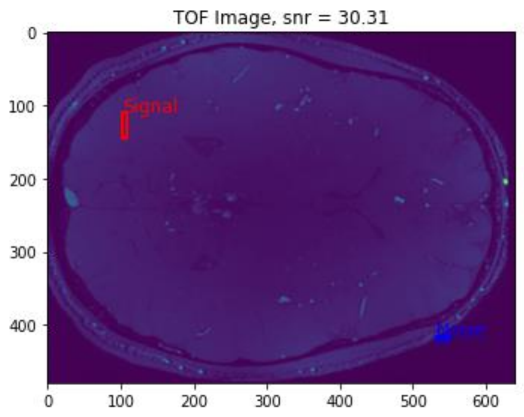
SNR = mean(signal)/std(noise)
The full image, signal patch , mean patch along with histogram for each image:

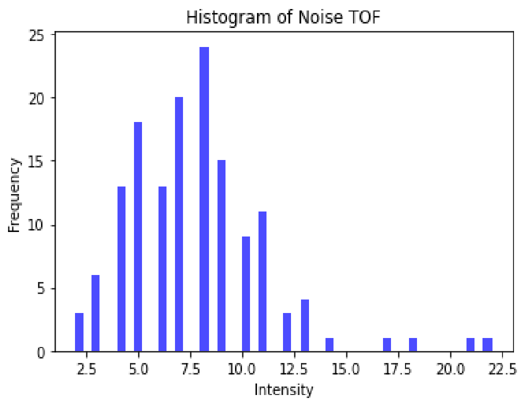
1. **SWI Image**
   Signal Patch Coordinates for image swi_img :
   X: slice(190, 240, None) Y: slice(140, 220, None) Z: 250
   Noise Patch Coordinates for image swi_img :
   X: slice(90, 110, None) Y: slice(340, 360, None) Z: 250
   SNR for swi_img is **18.81**



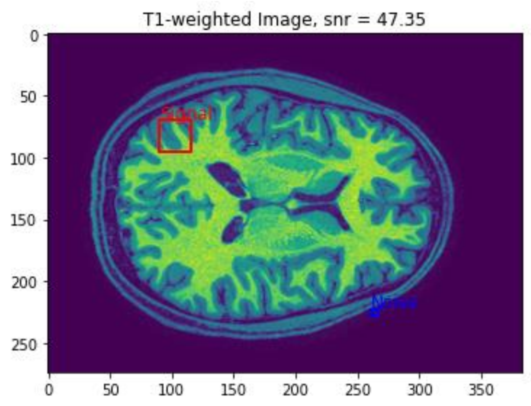


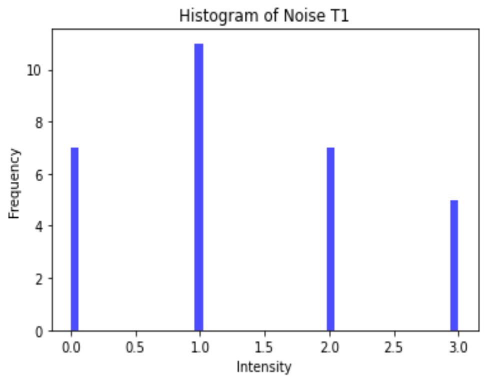
2. **TOF Image**
   Signal Patch Coordinates for image tof_img :
   X: slice(110, 145, None) Y: slice(102, 110, None) Z: 80
   Noise Patch Coordinates for image tof_img :
   X: slice(416, 424, None) Y: slice(532, 550, None) Z: 80
   SNR for tof_img is **30.31**



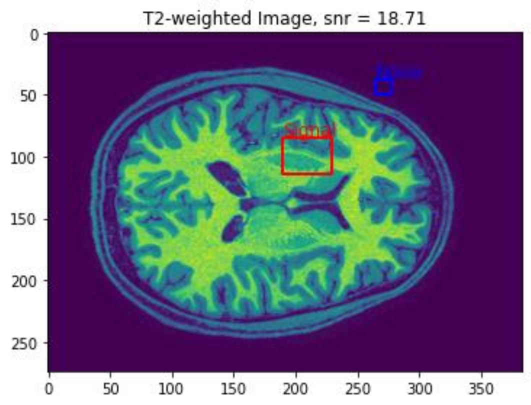


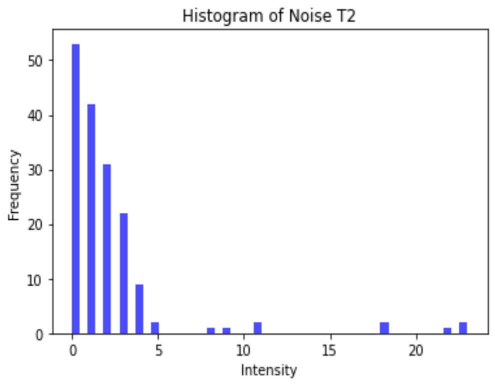
3. **T1 Image**
   Signal Patch Coordinates for image t1_img :
   X: slice(70, 95, None) Y: slice(90, 115, None) Z: 250
   Noise Patch Coordinates for image t1_img :
   X: slice(223, 228, None) Y: slice(261, 267, None) Z: 250
   SNR for t1_img is **47.35**



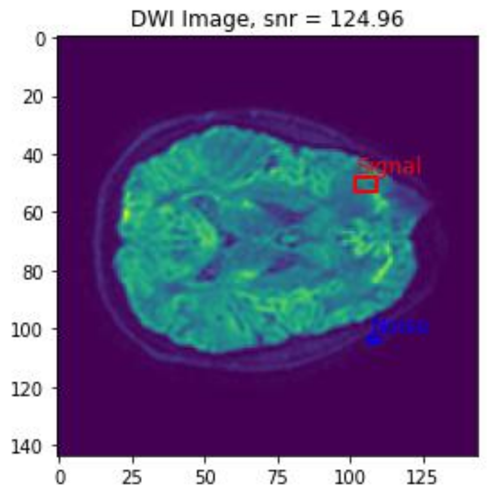


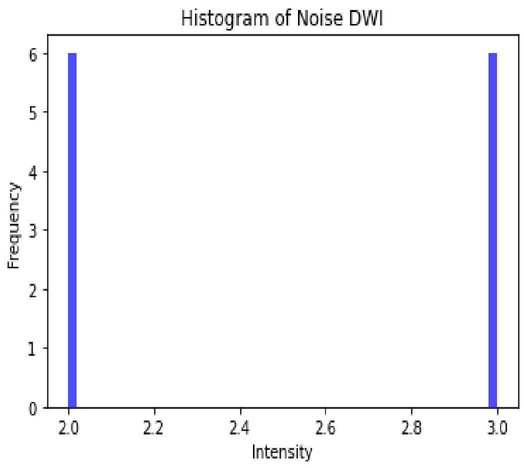
4. **T2 Image**
   Signal Patch Coordinates for image t2_img :
   X: slice(85, 115, None) Y: slice(190, 230, None) Z: 192
   Noise Patch Coordinates for image t2_img :
   X: slice(38, 50, None) Y: slice(264, 278, None) Z: 192
   SNR for t2_img is **18.71**



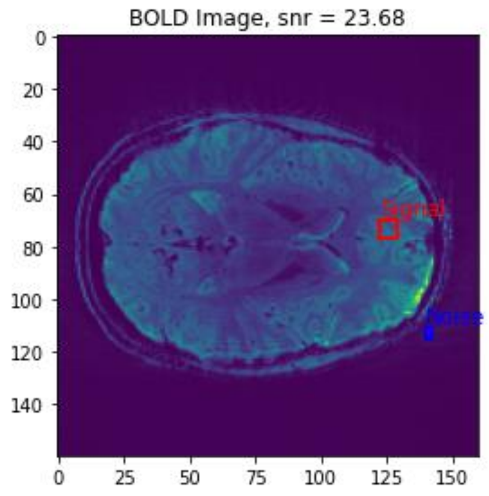


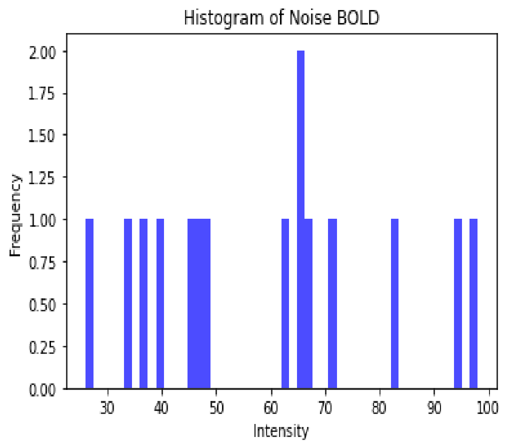
5. **DWI Image**
   Signal Patch Coordinates for image dwi_img :
   X: slice(48, 54, None) Y: slice(102, 110, None) Z: 35
   Noise Patch Coordinates for image dwi_img :
   X: slice(103, 106, None) Y: slice(107, 111, None) Z: 35
   SNR for dwi_img is **124.96**





6. **Bold Image**
   Signal Patch Coordinates for image bold_img :
   X: slice(70, 78, None) Y: slice(122, 130, None) Z: 18
   Noise Patch Coordinates for image bold_img :
   X: slice(111, 116, None) Y: slice(140, 143, None) Z: 18
   SNR for bold_img is **23.68**





<u>Observation:</u>

Highest SNR : DWI Image
Lowest SNR: T2 Image

Note: This SNR value also depends on the patches that are selected. But however for DWI image given here the Value has outperformed multiple times compared to rest of the modalities. The reasons might be the reconstruction algorithms used for DWI and acquisition parameters.

Distribution:
Unfortunately, the noise patches in the images were very small, so the plotted histograms may

not accurately represent the true distribution. However, by examining the plotted histograms, we can observe the following trends:

- T1, BOLD, and TOF images appear to follow a normal distribution. The histogram shapes resemble the typical bell-shaped curve.
- T2 and SWI images exhibit a right-skewed distribution. The histograms are skewed towardshigher intensity values, indicating a lower frequency of occurrence for brighter intensities.

The distribution of DWI (Diffusion-Weighted Imaging) is challenging to predict. This is because the frequencies for intensity values in the middle range were zero within the selected noise patch. Therefore, it is difficult to determine the overall distribution pattern for DWI. Due to the small noise patches, the plotted histograms may not precisely represent the underlying noise distribution. Further analysis or consideration of larger noise regions may be required for a more accurate assessment.

Part 4 : Denoising

a) Using the Fourier transform method and the 3d gaussian, apply linear filtering to each image for σ=2,σ=4, and σ=15. Create 3 new versions of the figure in part 1a, one figure for each sigma. Show the middle z-slice of the filtered image in all subplots.

Using fourier transform and 3d gaussian linear filtering, generated images for sigma 2, 4 and 15. Plotted the middle z-slice of each image.
Note: as discussed in mail conversation, we put fftshift on 3d fft before applying the filter and used it again after inverting the fit image. Yet we got blurry images as displayed below.

```
# -*- coding: utf-8 -*-
"""
Created on Mon May 29 20:40:33 2023

@author: scorp
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fftn, ifftn, fftshift, ifftshift
import nibabel as nib

# Load the nii image
nii_image_swi = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-forrestgump_acq-mag_veno.nii')
nii_image_t1 = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-forrestgump_T1w.nii')
nii_image_t2 = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-forrestgump_T2w.nii')
```

```
nii_image_dwi = nib.load('images/sub-01_ses-forrestgump_dwi_sub-01_ses-
forrestgump_dwi.nii')
nii_image_bold = nib.load('images/sub-01_ses-forrestgump_func_sub-01_ses-
forrestgump_task-forrestgump_acq-dico_run-01_bold.nii')


# Get the image data from nii_image
swi_data = nii_image_swi.get_fdata()
t1_data = nii_image_t1.get_fdata()
t2_data = nii_image_t2.get_fdata()
dwi_data = nii_image_dwi.get_fdata()
bold_img = nii_image_bold.get_fdata()


# Extract the first three dimensions for downsampling and filtering
bold_data = bold_img[..., 0]

# Define the desired downsampled resolution
target_resolution = (40, 40, 10)

# Compute the scale factors for each dimension
scale_factors = np.array(bold_data.shape) / np.array(target_resolution)

# Downsample the image using NumPy's indexing and interpolation
downsampled_data = bold_data[::int(scale_factors[0]), ::int(scale_factors[1]),
::int(scale_factors[2])]

# Define the 3D Gaussian kernel function
def gaussian_kernel(shape, sigma):
    kernel = np.zeros(shape)
    center_coords = [(coord - 1) / 2 for coord in shape]
    grid = np.meshgrid(*[np.arange(-coord, coord + 1) for coord in
center_coords], indexing='ij')
    kernel = np.exp(-(grid[0] ** 2 + grid[1] ** 2 + grid[2] ** 2) / (2 * sigma
** 2))
    kernel /= np.sum(kernel)
    return kernel

# Apply linear filtering for different sigma values and show the middle z-
slice of the filtered image
sigmas = [2, 4, 15]

fig, axes = plt.subplots(1, len(sigmas), figsize=(12, 4))

for i, sigma in enumerate(sigmas):
    # Generate the Gaussian kernel with the same shape as the downsampled
image
    kernel_shape = downsampled_data.shape
    kernel = gaussian_kernel(kernel_shape, sigma)
    # Apply Fourier transform to the image and the kernel (with fftshift)
    image_fft = fftshift(fftn(downsampled_data))
    kernel_fft = fftshift(fftn(kernel, kernel_shape))

    # Apply linear filtering by multiplying the Fourier transformed image with
the kernel
    filtered_image_fft = image_fft * kernel_fft

    # Perform inverse Fourier transform to obtain the filtered image (with
ifftshift)
    filtered_image = np.real(ifftn(ifftshift(filtered_image_fft)))

    # Ensure the filtered image has the same shape as the input image
    filtered_image = filtered_image[:downsampled_data.shape[0],
:downsampled_data.shape[1], :downsampled_data.shape[2]]

    # Get the middle z-slice index
    middle_slice = filtered_image.shape[2] // 2

    # Display the middle z-slice of the filtered image
    axes[i].imshow(filtered_image[..., middle_slice], cmap='gray',
vmin=np.min(filtered_image), vmax=np.max(filtered_image))
    axes[i].set_title(f'Bold Image Sigma = {sigma}')
    axes[i].axis('off')

plt.tight_layout()
plt.show()


target_resolution = (40, 40, 9)
scale_factors = np.array(t1_data.shape) / np.array(target_resolution)
downsampled_data = t1_data[::int(scale_factors[0]), ::int(scale_factors[1]),
::int(scale_factors[2])]

def gaussian_kernel(shape, sigma):
    kernel = np.zeros(shape)
    center_coords = [(coord - 1) / 2 for coord in shape]
    grid = np.meshgrid(*[np.arange(-coord, coord + 1) for coord in
center_coords], indexing='ij')
    kernel = np.exp(-(grid[0] ** 2 + grid[1] ** 2 + grid[2] ** 2) / (2 * sigma
** 2))
    kernel /= np.sum(kernel)
    return kernel

sigmas = [2, 4, 15]

fig, axes = plt.subplots(1, len(sigmas), figsize=(12, 4))
```

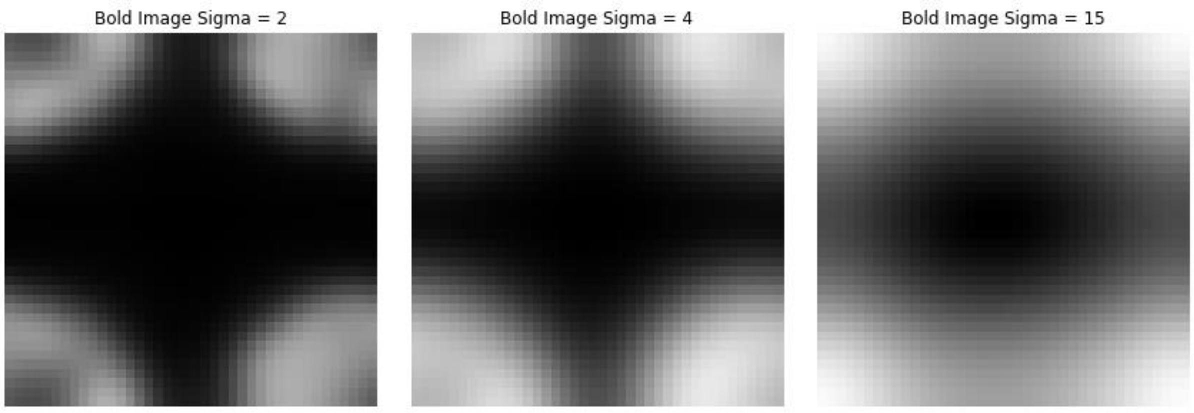
```
for i, sigma in enumerate(sigmas):
    kernel_shape = downsampled_data.shape
    kernel = gaussian_kernel(kernel_shape, sigma)

    image_fft = fftshift(fftn(downsampled_data))
    kernel_fft = fftshift(fftn(kernel, kernel_shape))

    filtered_image_fft = image_fft * kernel_fft

    filtered_image = np.real(ifftn(ifftshift(filtered_image_fft)))

    filtered_image = filtered_image[:downsampled_data.shape[0],
:downsampled_data.shape[1], :downsampled_data.shape[2]]

    middle_slice = filtered_image.shape[2] // 2

    filtered_image = np.clip(filtered_image, 0, 255)

    axes[i].imshow(filtered_image[..., middle_slice], cmap='gray')
    axes[i].set_title(f'T1 Image Sigma = {sigma}')
    axes[i].axis('off')

plt.tight_layout()
plt.show()


# Define the desired downsampled resolution
target_resolution = (40, 40,10)

# Compute the scale factors for each dimension
scale_factors = np.array(t2_data.shape) / np.array(target_resolution)

# Downsample the image using NumPy's indexing and interpolation
downsampled_data = t2_data[::int(scale_factors[0]), ::int(scale_factors[1]),
::int(scale_factors[2])]

# Define the 3D Gaussian kernel function
def gaussian_kernel(shape, sigma):
    kernel = np.zeros(shape)
    center_coords = [(coord - 1) / 2 for coord in shape]
    grid = np.meshgrid(*[np.arange(-coord, coord + 1) for coord in
center_coords], indexing='ij')
    kernel = np.exp(-(grid[0] ** 2 + grid[1] ** 2 + grid[2] ** 2) / (2 * sigma
** 2))
    kernel /= np.sum(kernel)
    return kernel

# Apply linear filtering for different sigma values and show the middle z-
slice of the filtered image
```

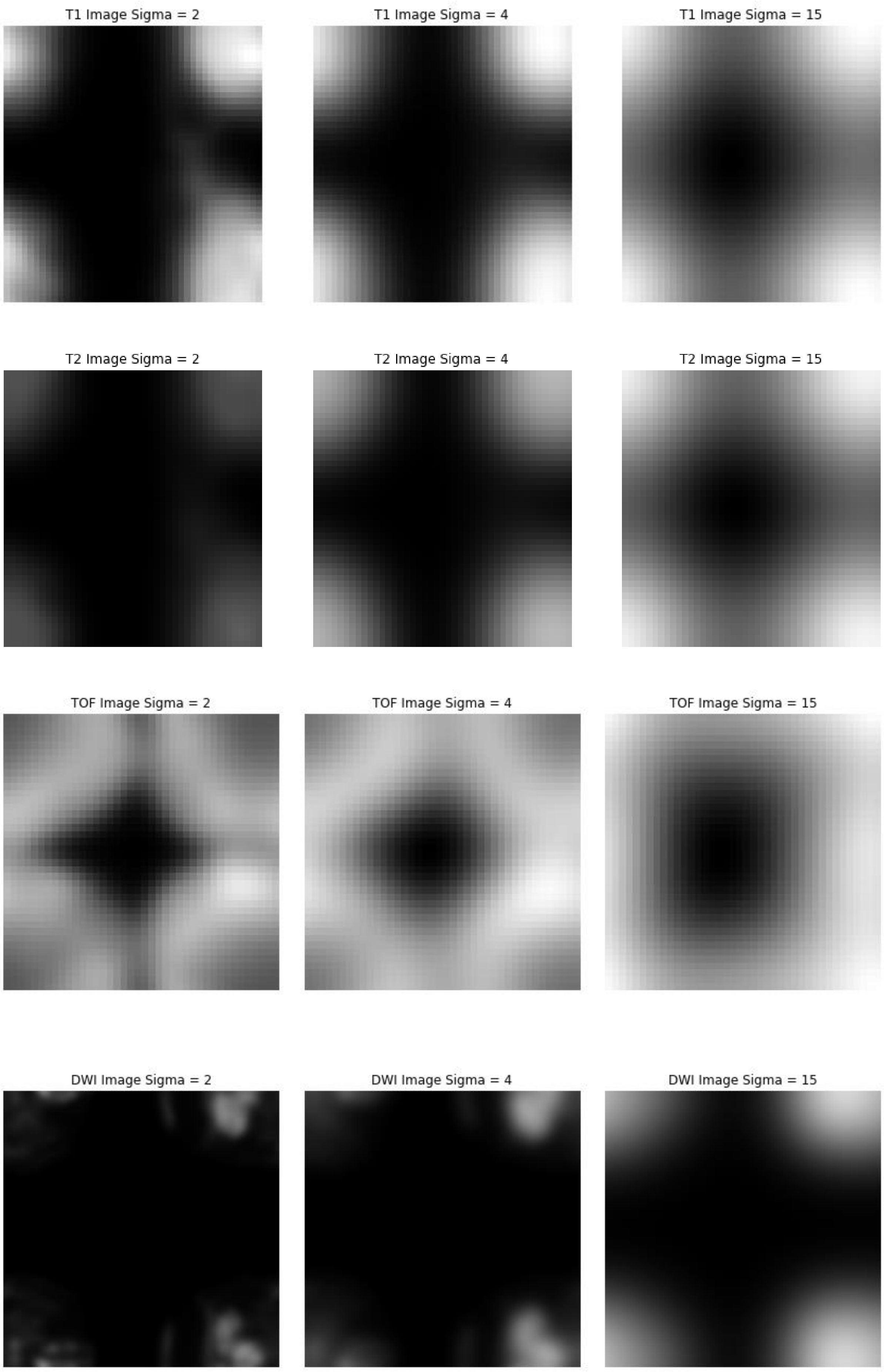
```python
sigmas = [2, 4, 15]

fig, axes = plt.subplots(1, len(sigmas), figsize=(12, 4))

for i, sigma in enumerate(sigmas):
    # Generate the Gaussian kernel
    kernel = gaussian_kernel(downsampled_data.shape, sigma)

    # Apply Fourier transform to the image and the kernel
    image_fft = fftn(downsampled_data)
    kernel_fft = fftn(kernel)

    # Apply linear filtering by multiplying the Fourier transformed image with
the kernel
    filtered_image_fft = image_fft * kernel_fft

    # Perform inverse Fourier transform to obtain the filtered image
    filtered_image = np.real(ifftn(filtered_image_fft))

    # Ensure the filtered image has the same shape as the input image
    filtered_image = filtered_image[:downsampled_data.shape[0],
:downsampled_data.shape[1], :downsampled_data.shape[2]]

    # Get the middle z-slice index
    middle_slice = filtered_image.shape[2] // 2

    # Display the middle z-slice of the filtered image
    axes[i].imshow(filtered_image[..., middle_slice], cmap='gray',
vmin=np.min(filtered_image), vmax=np.max(filtered_image))
    axes[i].set_title(f'T2 Image Sigma = {sigma}')
    axes[i].axis('off')

plt.tight_layout()
plt.show()


# Load the image
image_path = 'images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_angio.nii'
image1 = nib.load(image_path)
tof_data = image1.get_fdata()


# Ensure the data has three dimensions
if tof_data.ndim == 4:
    tof_data = tof_data[..., 0]

# Define the desired downsampled resolution
target_resolution = (40, 40, 9)
```

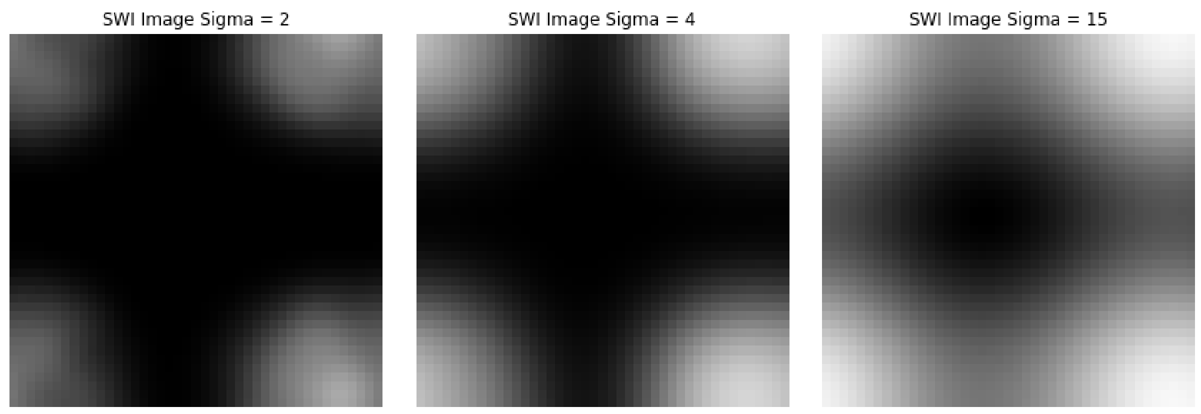
```
# Compute the scale factors for each dimension
scale_factors = np.array(tof_data.shape[:3]) / np.array(target_resolution)

# Downsample the image using numpy's indexing and interpolation
downsampled_data = tof_data[::int(scale_factors[0]), ::int(scale_factors[1]),
::int(scale_factors[2])]

# Define the 3D Gaussian kernel function
def gaussian_kernel(shape, sigma):
    kernel = np.zeros(shape)
    center_coords = [(coord - 1) / 2 for coord in shape]
    grid = np.meshgrid(*[np.arange(-coord, coord + 1) for coord in
center_coords], indexing='ij')
    kernel = np.exp(-(grid[0] ** 2 + grid[1] ** 2 + grid[2] ** 2) / (2 * sigma
** 2))
    kernel /= np.sum(kernel)
    return kernel

# Apply linear filtering for different sigma values and show the middle z-
slice of the filtered image
sigmas = [2, 4, 15]

fig, axes = plt.subplots(1, len(sigmas), figsize=(12, 4))

for i, sigma in enumerate(sigmas):
    # Generate the Gaussian kernel
    kernel = gaussian_kernel(downsampled_data.shape, sigma)

    # Apply Fourier transform to the image and the kernel
    image_fft = fftn(downsampled_data)
    kernel_fft = fftn(kernel)

    # Apply fftshift to the Fourier transformed image and kernel
    image_fft_shifted = fftshift(image_fft)
    kernel_fft_shifted = fftshift(kernel_fft)

    # Apply linear filtering by multiplying the shifted Fourier transformed
image with the shifted kernel
    filtered_image_fft_shifted = image_fft_shifted * kernel_fft_shifted

    # Perform inverse fftshift on the filtered image
    filtered_image_fft = ifftshift(filtered_image_fft_shifted)

    # Perform inverse Fourier transform to obtain the filtered image
    filtered_image = np.real(ifftn(filtered_image_fft))

    # Ensure the filtered image has the same shape as the input image
    filtered_image = filtered_image[:downsampled_data.shape[0],
:downsampled_data.shape[1], :downsampled_data.shape[2]]

    # Get the middle z-slice index
    middle_slice = downsampled_data.shape[2] // 2

    # Display the middle z-slice of the filtered image
    axes[i].imshow(filtered_image[..., middle_slice], cmap='gray',
vmin=np.min(filtered_image), vmax=np.max(filtered_image))
    axes[i].set_title(f'TOF Image Sigma = {sigma}')
    axes[i].axis('off')

plt.tight_layout()
plt.show()


# Define the desired downsampled resolution
target_resolution = (40, 40, 9)

# Compute the scale factors for each dimension
scale_factors = np.array(swi_data.shape[:3]) / np.array(target_resolution)

# Downsample the image using numpy's indexing and interpolation
downsampled_data = swi_data[::int(scale_factors[0]), ::int(scale_factors[1]),
::int(scale_factors[2])]

# Define the 3D Gaussian kernel function
def gaussian_kernel(shape, sigma):
    kernel = np.zeros(shape)
    center_coords = [(coord - 1) / 2 for coord in shape]
    grid = np.meshgrid(*[np.arange(-coord, coord + 1) for coord in
center_coords], indexing='ij')
    kernel = np.exp(-(grid[0] ** 2 + grid[1] ** 2 + grid[2] ** 2) / (2 * sigma
** 2))
    kernel /= np.sum(kernel)
    return kernel

# Apply linear filtering for different sigma values and show the middle z-
slice of the filtered image
sigmas = [2, 4, 15]

fig, axes = plt.subplots(1, len(sigmas), figsize=(12, 4))

for i, sigma in enumerate(sigmas):
    # Generate the Gaussian kernel
    kernel = gaussian_kernel(downsampled_data.shape, sigma)

    # Apply Fourier transform to the image and the kernel
    image_fft = fftn(downsampled_data)
    kernel_fft = fftn(kernel)

    # Apply linear filtering by multiplying the Fourier transformed image with
the kernel
    filtered_image_fft = image_fft * kernel_fft

    # Perform inverse Fourier transform to obtain the filtered image
    filtered_image = np.real(ifftn(filtered_image_fft))

    # Ensure the filtered image has the same shape as the input image
    filtered_image = filtered_image[:downsampled_data.shape[0],
:downsampled_data.shape[1], :downsampled_data.shape[2]]

    # Get the middle z-slice index
    middle_slice = downsampled_data.shape[2] // 2

    # Display the middle z-slice of the filtered image
    axes[i].imshow(filtered_image[..., middle_slice], cmap='gray',
vmin=np.min(filtered_image), vmax=np.max(filtered_image))
    axes[i].set_title(f'SWI Image Sigma = {sigma}')
    axes[i].axis('off')

plt.tight_layout()
plt.show()


# Define the 3D Gaussian kernel function
def gaussian_kernel(shape, sigma):
    kernel = np.zeros(shape)
    center_coords = [(coord - 1) / 2 for coord in shape]
    grid = np.meshgrid(*[np.arange(-coord, coord + 1) for coord in
center_coords], indexing='ij')
    kernel = np.exp(-(grid[0] ** 2 + grid[1] ** 2 + grid[2] ** 2) / (2 * sigma
** 2))
    kernel /= np.sum(kernel)
    return kernel

# Apply linear filtering for different sigma values and show the middle z-
slice of the filtered image
sigmas = [2, 4, 15]

fig, axes = plt.subplots(1, len(sigmas), figsize=(12, 4))

for i, sigma in enumerate(sigmas):
    # Generate the Gaussian kernel
    kernel = gaussian_kernel(dwi_data.shape, sigma)

    # Apply Fourier transform to the image and the kernel
    image_fft = fftn(dwi_data)
```

```python
    kernel_fft = fftn(kernel)

    # Apply linear filtering by multiplying the Fourier transformed image with the kernel
    filtered_image_fft = image_fft * kernel_fft

    # Perform inverse Fourier transform to obtain the filtered image
    filtered_image = np.real(ifftn(filtered_image_fft))

    # Ensure the filtered image has the same shape as the input image
    filtered_image = filtered_image[:dwi_data.shape[0], :dwi_data.shape[1], :dwi_data.shape[2], :dwi_data.shape[3]]

    # Get the middle z-slice index
    middle_slice = dwi_data.shape[2] // 2

    # Display the middle z-slice of the filtered image
    axes[i].imshow(filtered_image[..., middle_slice, 0], cmap='gray', vmin=np.min(filtered_image), vmax=np.max(filtered_image))
    axes[i].set_title(f'DWI Image Sigma = {sigma}')
    axes[i].axis('off')

plt.tight_layout()
plt.show()
```

Result:




T1 Image Sigma = 2
T1 Image Sigma = 4
T1 Image Sigma = 15
T2 Image Sigma = 2
T2 Image Sigma = 4
T2 Image Sigma = 15
TOF Image Sigma = 2
TOF Image Sigma = 4
TOF Image Sigma = 15
DWI Image Sigma = 2
DWI Image Sigma = 4
DWI Image Sigma = 15



b) Install the ‘dipy’ package and use nlmeans to denoise the images. Show, for each image, the following plots 1) noisy image 2) denoised image 3) method noise (noisy minus denoised), see below for an example on the T1.

For this task, we have plotted the original, denoised and method noise image by calculating the denoised image using nlmeans from the dipy package.

```python
import nibabel as nib
import matplotlib.pyplot as plt
import numpy as np
# !pip install dipy
from dipy.denoise.nlmeans import nlmeans
from dipy.denoise.noise_estimate import estimate_sigma

# Load the nii image
nii_image_swi = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_acq-mag_veno.nii')
nii_image_tof = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_angio.nii')
nii_image_t1 = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_T1w.nii')
nii_image_t2 = nib.load('images/sub-01_ses-forrestgump_anat_sub-01_ses-
forrestgump_T2w.nii')
nii_image_dwi = nib.load('images/sub-01_ses-forrestgump_dwi_sub-01_ses-
forrestgump_dwi.nii')
nii_image_bold = nib.load('images/sub-01_ses-forrestgump_func_sub-01_ses-
forrestgump_task-forrestgump_acq-dico_run-01_bold.nii')

# Get the image data from nii_image
swi_img = nii_image_swi.get_fdata()
tof_img = nii_image_tof.get_fdata()
t1_img = nii_image_t1.get_fdata()
t2_img = nii_image_t2.get_fdata()
dwi_img = nii_image_dwi.get_fdata()
```

```
bold_img = nii_image_bold.get_fdata()

def slice_image(image_name,image):
    if image_name == "swi_img":
        # Slicing the middle z-slice for swi image and formatting it
        return np.rot90(image[:, :, image.shape[2]//2])
    elif image_name == "tof_img":
        # Slicing the middle z-slice for tof image and formatting it
        return np.fliplr(np.rot90(np.transpose(image[:, :,
image.shape[2]//2]),2))
    elif image_name == "t1_img":
        # Slicing the middle z-slice for t1 image and formatting it
        return np.fliplr(np.rot90(np.transpose(image[:, :,
image.shape[2]//2]),2))
    elif image_name == "t2_img":
        # Slicing the middle z-slice for t2 image and formatting it
        return np.fliplr(np.rot90(np.transpose(image[:, :,
image.shape[2]//2]),2))
    elif image_name == "dwi_img":
        # Slicing the middle z-slice for dwi image and formatting it
        return np.fliplr(np.rot90(np.transpose(image[:, :, image.shape[2]//2,
image.shape[3]//2]),2))
    elif image_name == "bold_img":
        # Slicing the middle z-slice for bold image and formatting it
        return np.fliplr(np.rot90(np.transpose(image[:, :, image.shape[2]//2,
image.shape[3]//2]),2))


image_list = {"swi_img":swi_img, "tof_img":tof_img, "t1_img":t1_img,
"t2_img":t2_img, "dwi_img":dwi_img, "bold_img":bold_img}

subplot_index=1
plt.figure(figsize=(10, 7))

for image_name,image in image_list.items():
    sigma = estimate_sigma(image)
    denoised_image = nlmeans(image, sigma=sigma)
    method_noise = image - denoised_image

    plt.subplot(6, 3, subplot_index)
    plt.imshow(slice_image(image_name,image), cmap='jet')
    plt.title(f'{image_name} - Original')
    plt.axis('off')

    subplot_index = subplot_index + 1
    plt.subplot(6, 3, subplot_index)
    plt.imshow(slice_image(image_name,denoised_image), cmap='jet')
    plt.title(f'{image_name} - Denoised Image')
    plt.axis('off')
    subplot_index = subplot_index + 1
    plt.subplot(6, 3, subplot_index)
    plt.imshow(slice_image(image_name,method_noise), cmap='jet')
    plt.title(f'{image_name} - Method Noise')
    plt.axis('off')

    subplot_index = subplot_index + 1

plt.tight_layout()
plt.show()
```

Result: