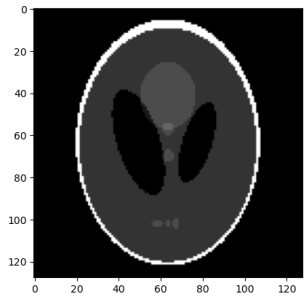
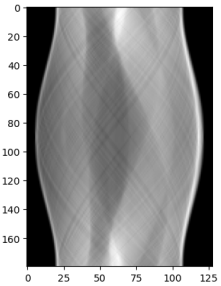
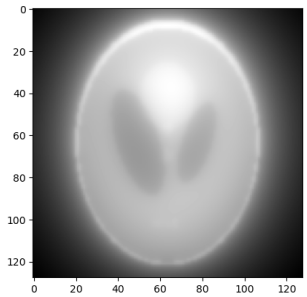
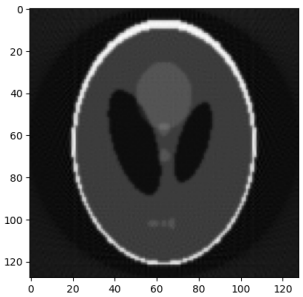
# COMP0114 Inverse Problems in Imaging. Mini Project

## Part A

### 1. Calculate the Radon transform of an image and test the back-projection method.

#### Common Operations

I use Python `astra` library to do Radon Transform and related operations.

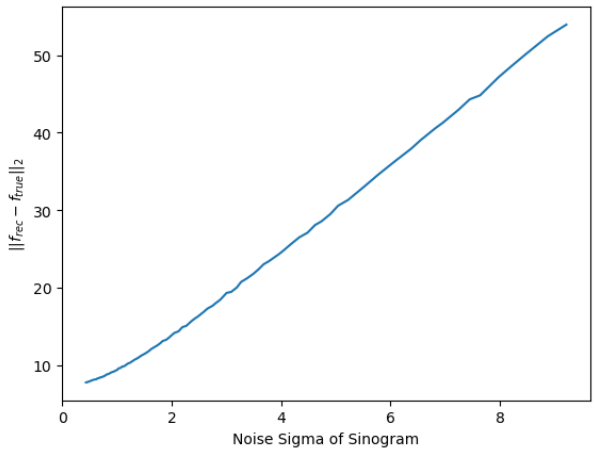
| Shepp-Logan | Radon Transform | $unfiltered$ back-projection | $filtered$ back-projection |
|---|---|---|---|
|  |  |  |  |

In the above experiments, the Radon Transformation is generated by $128$ projections with $1$ spacing and $1$-degree intervals from $0^{\circ}$ to $179^{\circ}$. So, the size of the Radon Transform image is $180 \times 128$, i.e. the number of angles times the number of projections.

The size of $unfiltered$ back-projection image should be the same as the original image, which is $128 \times 128$.

The $filtered$ back-projection is a good estimation of the inverse of the Radon transform. The $\mathrm{L}2$ distance between the original Shepp-Logan image and the $filtered$ back-projection image in this experiment is $7.36$, which is fairly small.

#### Add noise to Radon Transform



I use `astra.functions.add_noise_to_sino` function in the `astra` library to add noise to the Radon Transform, and then do the $filtered$ back-projection with the noisy sinogram. I adjust the noise level and observe the effect of back-projection.

I plot the noise variance and the back-projection error as above, and we can see that they are positively correlated.

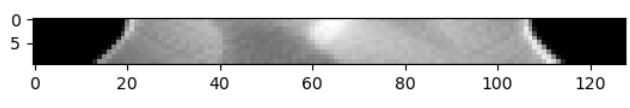
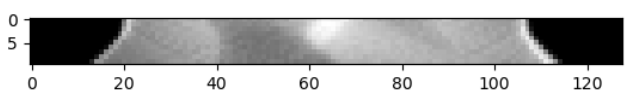
## 2. Calculate an explicit matrix form of the Radon transform and investigate its SVD.

### Construct explicit Radon Transform matrix

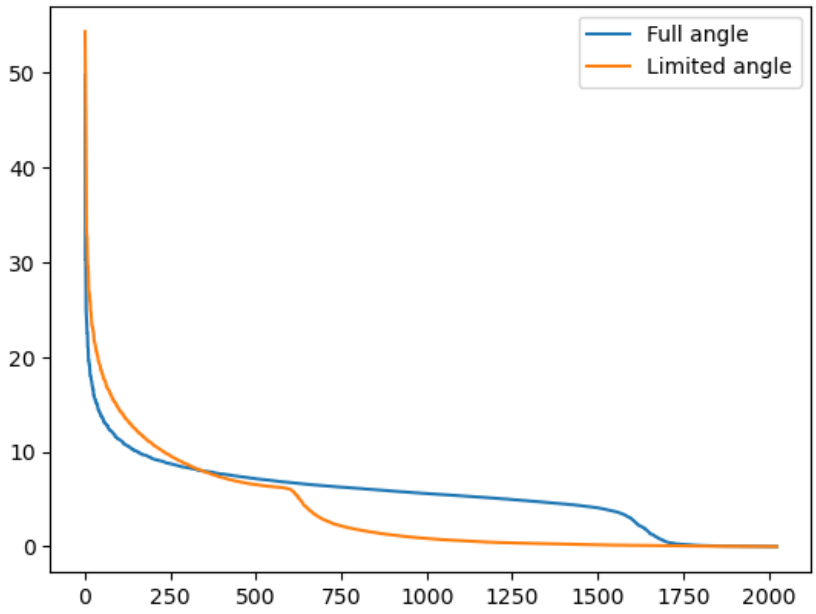
I write a function as below to calculate the explicit Matrix of Radon Transform.

```
def construct_Radon_Matrix(h, w, n_angles, max_angle, n_projs):
    # h, w: The height and width of the image applying the Randon Transform
    # The Radon Transform will do `n_projs` projection in
    # each angle in np.linspace(0, max_angle, n_angles, endpoint=False).

    f = np.zeros((h, w))
    n_pixels = h * w
    R = np.zeros((n_angles * n_projs, n_pixels))
    for i in range(n_pixels):
        r = i // w  # Calculate row index
        c = i % w   # Calculate column index
        f[r, c] = 1  # Set one pixel to 1
        # Function Randon_Transform uses astra library to do Randon Transform
        vol_geom, proj_geom, proj_id, g_id, g = Randon_Transform(
            f, det_count=n_projs, angle_count=n_angles, max_angle=max_angle)
        R[:, i] = g.reshape((n_angles * n_projs,))
        f[r, c] = 0  # Recover the pixel

    return R

def apply_Radon_Matrix(R, im, n_angles, n_projs):
    # Helper function: apply a Radon Transform Matrix to an image.
    h, w = im.shape
    return (R @ im.reshape((h * w))).reshape((n_angles, n_projs))
```

The matrix returned by the `construct_Radon_Matrix` can have the same effect as the true Radon transform. Below is applying a Radon Transform to Shepp-Logan using library function and explicit matrix, the result is the same.

| **Randon_Transform(f_true, det_count=128, angle_count=10, max_angle=np.pi/4)** | **construct_Radon_Matrix(128, 128, n_angles=10, max_angle=np.pi/4, n_projs=128)** |
| --- | --- |
|  |  |

**SVD Analysis**




As shown above, the Radon Matrix, which has limited angles, has relatively few principal components, representing that it is missing some information compared to the full-angle one.

## 3. Implement a matrix-free regularised least-squares solver for the Radon Transform.

$\mathbf{f}_{\text{true}}$ *denote the original Shepp-Logan image.*

The solvers are basically the same as what did in CW2.

Here the normal equation is:

$$(\mathbf{A}^{\mathrm{T}}\mathbf{A} + \alpha\mathbf{L})\mathbf{f}_* = \mathbf{A}^{\mathrm{T}}\mathbf{g} \tag{1}$$

, where $\mathbf{A}$ represents the Radon Transform, $\mathbf{L}$ represents the regularization matrix and $\alpha$ is its coefficient, and $\mathbf{g} = \mathbf{A}\mathbf{f}_{\text{true}}$.

In my experiments, I solve the augmented equations instead:

$$\begin{pmatrix} \mathbf{A} \\ \sqrt{\alpha}\mathbf{\Gamma} \end{pmatrix}\mathbf{f}_* = \begin{pmatrix} \mathbf{g} \\ 0 \end{pmatrix} \tag{2}$$

, where $\mathbf{\Gamma}^{\mathrm{T}}\mathbf{\Gamma} = \mathbf{L}$.

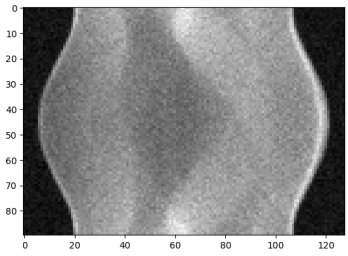
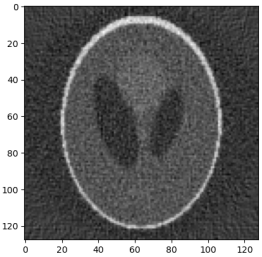
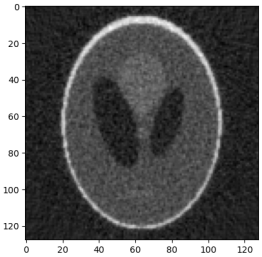
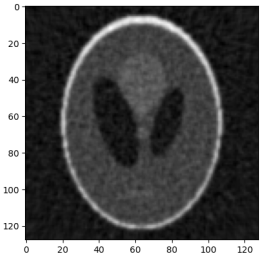
In zero-order Tikhonov regularisation (TK0) $\mathbf{\Gamma} = \mathbf{I}$, and in first-order Tikhonov regularisation (TK1) $\mathbf{\Gamma} = \begin{pmatrix} \nabla\mathbf{x} \\ \nabla\mathbf{y} \end{pmatrix}$.

In this part, I use three parameters to define a Radon Transform $\mathbf{A}$: $n_{\text{projs}}$, $n_{\text{angles}}$ and $\theta_{\max}$, which represents a Radon Transform with $n_{\text{projs}}$ projections and $n_{\text{angles}}$ angles sampled uniformly from $[0, \theta_{\max})$.

I create the Radon matrix-free operator by projection and back-projection (as adjoint) functions in `astra` library, and verify its correctness. Please see my code in order to get details of this part.

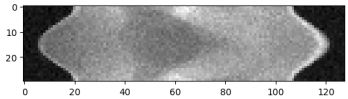
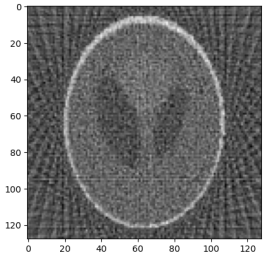
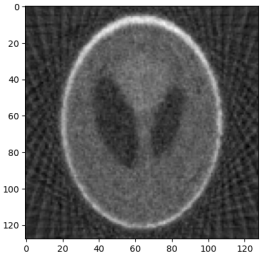
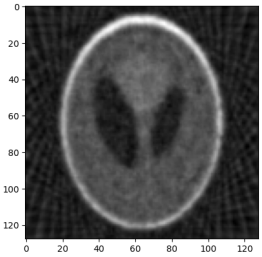
**Baseline: Just Noise**

$n_{\text{projs}} = 128$, $n_{\text{angles}} = 90$, $\theta_{\max} = \pi$

| Sinogram | FBP (Baseline) | TK0 | TK1 |
|---|---|---|---|
|  |  |  |  |
| Noise $\sigma$: $1.35$ | $\mathrm{L2}$ distance to $\mathbf{f}_{\text{true}}$: $13.68$ | $\mathrm{L2}$ distance to $\mathbf{f}_{\text{true}}$: $12.00$ | $\mathrm{L2}$ distance to $\mathbf{f}_{\text{true}}$: $11.85$ |

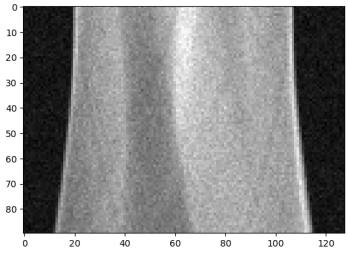
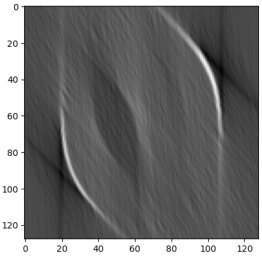
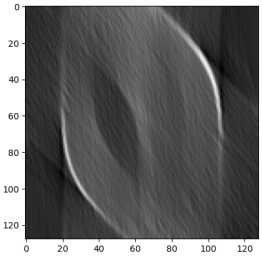
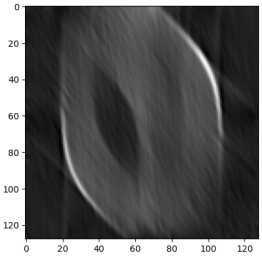
**Baseline: Full range but small number of angles**

$n_{\text{projs}} = 128$, $n_{\text{angles}} = 30$, $\theta_{\max} = \pi$

| Sinogram | FBP (Baseline) | TK0 | TK1 |
|---|---|---|---|
|  |  |  |  |
| Noise $\sigma$: $1.35$ | $\mathrm{L2}$ distance to $\mathbf{f}_{\text{true}}$: $25.10$ | $\mathrm{L2}$ distance to $\mathbf{f}_{\text{true}}$: $14.97$ | $\mathrm{L2}$ distance to $\mathbf{f}_{\text{true}}$: $14.59$ |

**Baseline: Limited angles**

$n_{\text{projs}} = 128$, $n_{\text{angles}} = 90$, $\theta_{\max} = \frac{\pi}{4}$

| Sinogram | FBP (Baseline) | TK0 | TK1 |
|---|---|---|---|
|  |  |  |  |
| Noise $\sigma$: $1.36$ | $\mathrm{L2}$ distance to $\mathbf{f}_{\text{true}}$: $41.74$ | $\mathrm{L2}$ distance to $\mathbf{f}_{\text{true}}$: $21.13$ | $\mathrm{L2}$ distance to $\mathbf{f}_{\text{true}}$: $20.29$ |

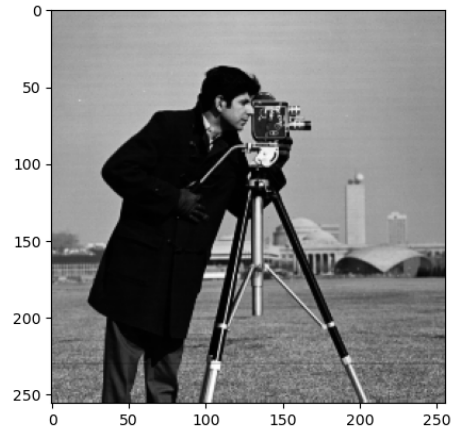
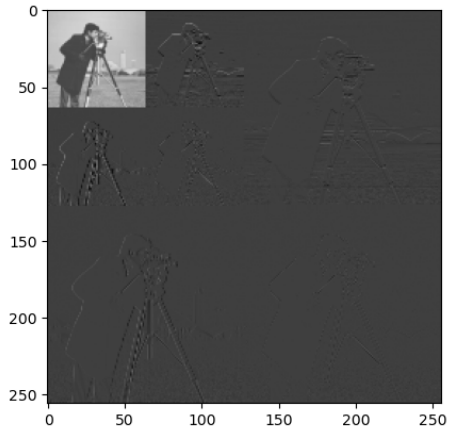
Overall, both TK0 and TK1 have a better performance than the baseline, and TK1 has a smoother effect than TK0. In the experiments with limited angles as the baseline, none of the reconstructions is very good due to the large amount of overall data missing from the sinogram.

## 4. Write a Haar wavelet denoiser.

### Haar wavelet transform

I use `pywt` library to do Haar wavelet transform.

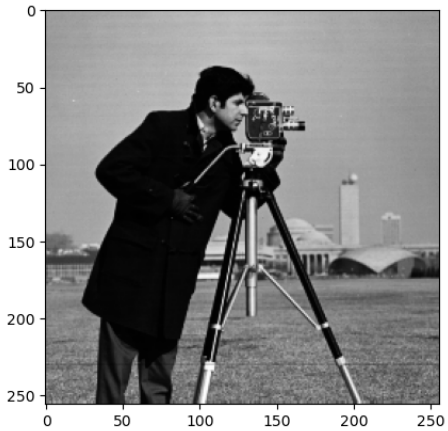
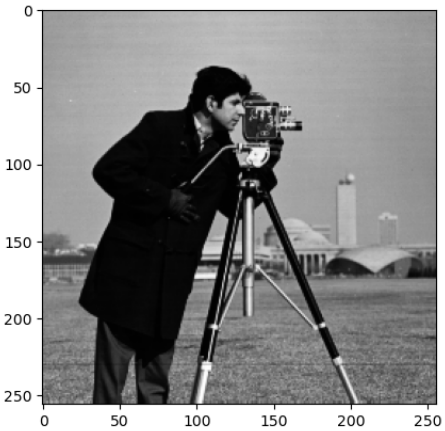
Below is the result of level $2$ Haar wavelet transform.

| Original | Haar wavelet transform coefficients |
| --- | --- |
|  |  |

In Haar wavelet transform coefficients, the image in the top leftmost corner represents the low-frequency information of the original image. The remaining images on the diagonal represent the high-frequency information of the original image along the diagonal direction. The upper triangular part of the whole image represents the high-frequency information of the original image in the vertical direction, while the lower triangular part represents the high-frequency information of the original image in the horizontal direction.

### Haar wavelet reconstruction

Below is the reconstruction result of level $2$ Haar wavelet transform.

| Original | Haar wavelet reconstruction |
| --- | --- |
|  |  |

The $L2$ distance between the original and reconstruction in this experiment is $2.18 \times 10^{-5}$, so the reconstructed image coincides with the original.

**Thresholding the Haar wavelet coefficients**

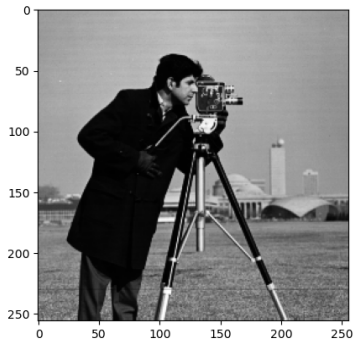
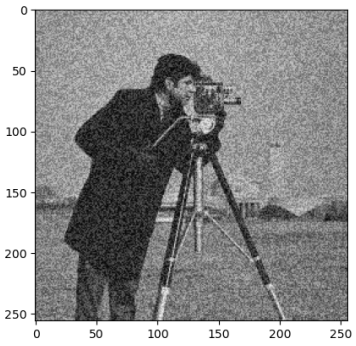
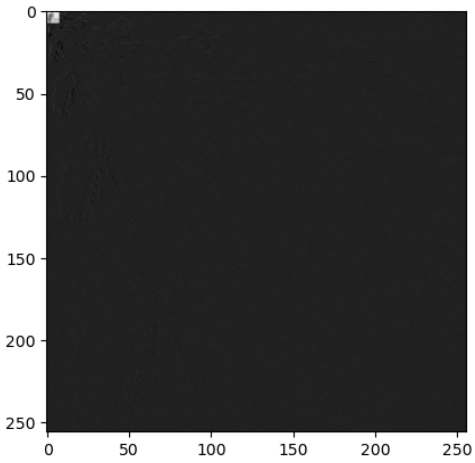
I write a series of functions to perform thresholding on the Haar wavelet coefficients.

```
def extract_coeffs(coeffs):
    # Extract all the non-zero high-frequency coefficients from coeffs
    level = len(coeffs) - 1
    all_coeffs = []
    for i in range(1, level + 1):
        for c in coeffs[i]:
            all_coeffs.extend(c.flatten().tolist())
    all_coeffs = np.array(all_coeffs)
    non0_coeffs = all_coeffs[np.where(all_coeffs!=0)]
    abs_coeffs = np.abs(non0_coeffs)
    return non0_coeffs[np.argsort(abs_coeffs)]

def determine_abs_threshold(x, coeffs):
    # Get the coefficient which is exactly on the boundary of
    # the lowest 100*x% of all the non-zero high-frequency coefficients.
    non0_coeffs = extract_coeffs(coeffs)
    cn = non0_coeffs.shape[0]
    return np.abs(non0_coeffs[int(cn * x)])

def threshold_coeffs(coeffs, threshold, level, threshold_fucntion=None):
    # thresholding a specific level in coeffs
    # by either library function or customized function
    new_coeffs = ()
    for c in coeffs[level]:
        if threshold_fucntion:
            new_coeffs += (threshold_fucntion(c, threshold),)
        else:
            new_coeffs += (pywt.threshold(c, threshold, mode='garrote', substitute=0),)
    coeffs[level] = new_coeffs

def threshold_specific_levels(coeffs, threshold, levels, threshold_fucntion=None):
    # thresholding multiple levels in coeffs
    # by either library function or customized function
    for level in levels:
        threshold_coeffs(coeffs, threshold, level, threshold_fucntion)
```

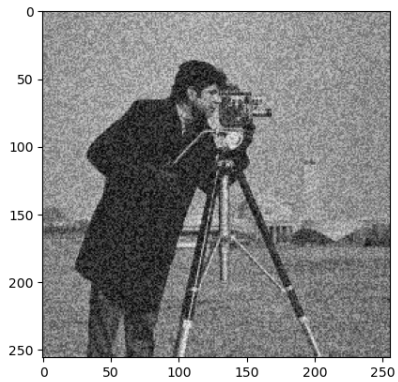
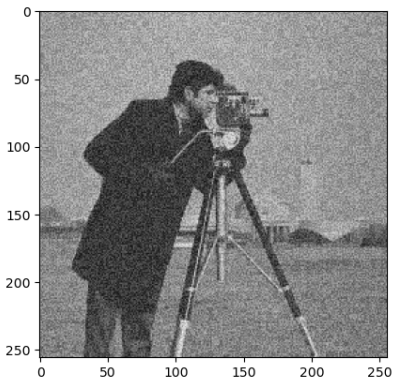
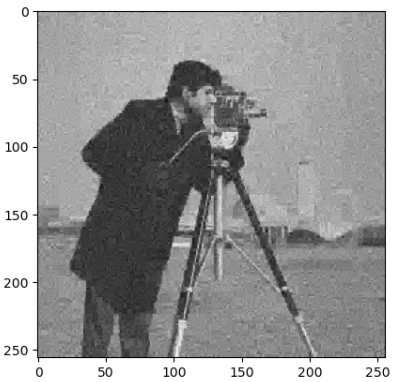
I decide the threshold by giving a percentage $x$, get the coefficient as the threshold which is exactly on the boundary of the lowest $x$% of all the non-zero high-frequency coefficients. There are also functions thresholding a specific level of coefficients or some levels of coefficients.

### Haar Denoiser

I add noise to the original image and perform a level $5$ Haar wavelet transform to the noisy image.

| Original | Noise $\sigma$: $0.14$ | Haar wavelet Transformation |
| --- | --- | --- |
|  |  |  |

There are totally $6$ components in the Haar transform coefficients. The first component represents the low-frequency information and the rest are high-frequency information. I only do thresholding on the high-frequency components ($2$~$6$ component). The $x$ for thresholding is $80$.

| Thresholding component $1$ | Thresholding component $2, 4$ | Thresholding component $2$~$6$ |
| --- | --- | --- |
|  |  |  |

We can see the more coefficients we threshold, the smoother the image is.

## 5. Iterative soft-thresholding for X-ray tomography.

### Reasoning

I use a self-defined soft-thresholding function for wavelet denoising in this part. Also, I use `db4` wavelet (level $4$) for better performance.

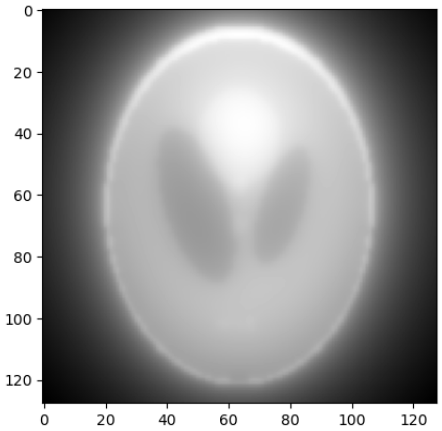
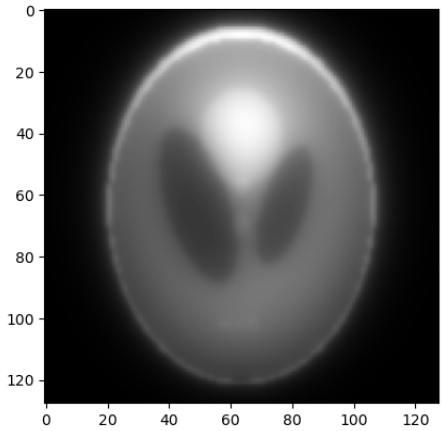
```
def soft_threshold(data, alpha):
    data_copy = data.copy()
    data_copy[data>=alpha] -= alpha
    data_copy[np.abs(data)<alpha] = 0
    data_copy[data<=-alpha] += alpha
    return data_copy
```

Also, I use prior to determine the parameter $\lambda$ and initial $\mathbf{f}_0$.

For $\lambda$, I calculate $L = \frac{1}{\max(\mathrm{SV}(\mathbf{A}))^2}$, where $\mathbf{A}$ is the Radon Transform matrix and $\mathrm{SV}$ is a function calculating all the singular values of a matrix. $\lambda$ should lie in $[0, 2L]$.

For initial $\mathbf{f}_0$, I use a filtered back-projection image of the input sinogram. Below is an example.

*Radon Transform parameters:* $n_{\text{projs}} = 128$, $n_{\text{angles}} = 180$, $\theta_{\max} = \pi$

| Pure Back-Projection | Filtered Back-Projection |
| --- | --- |
|  |  |



For the terminating of the iteration, I use two criteria to do it.

The first one is the max iteration number limitation, i.e. terminate when reaching the max iteration number $\mathrm{it}_{max}$.

The second one I calculate an error during each iteration:

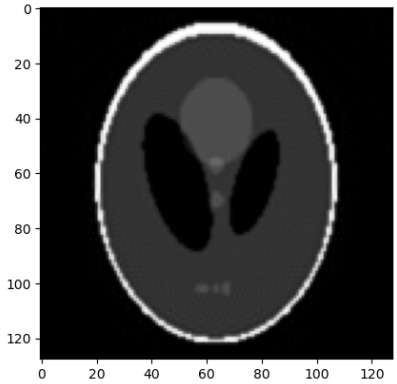
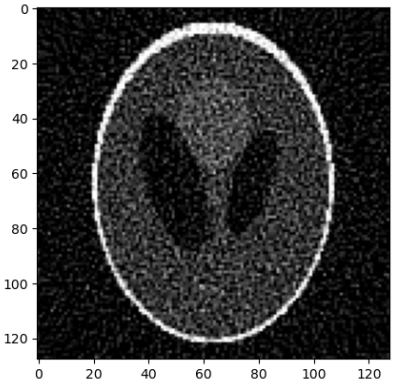
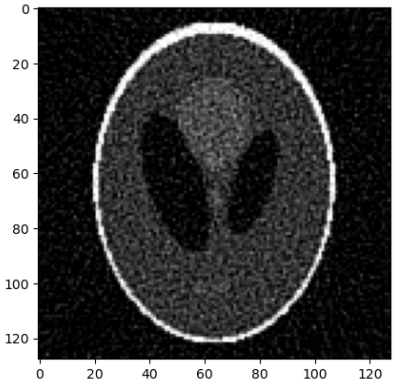
$$e_i = \frac{||\mathbf{A}\mathbf{f}_i - \mathbf{g}||_2}{|\mathbf{A}\mathbf{f}_i - \mathbf{g}|} \tag{3}$$

, given a tolerance $t$, if $|e_i - e_{i-1}| < t$, then terminate.

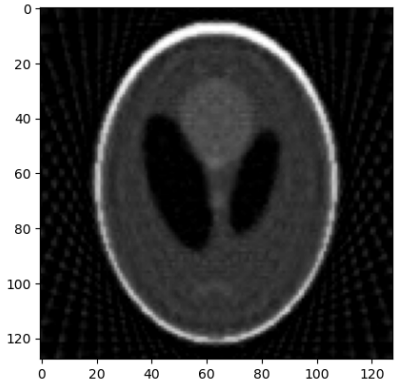
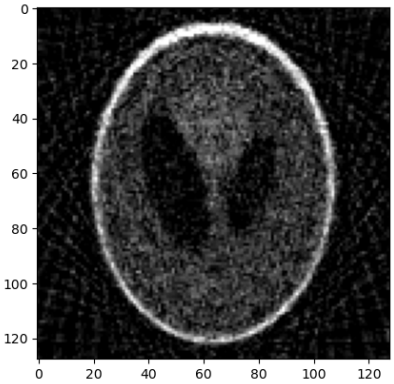
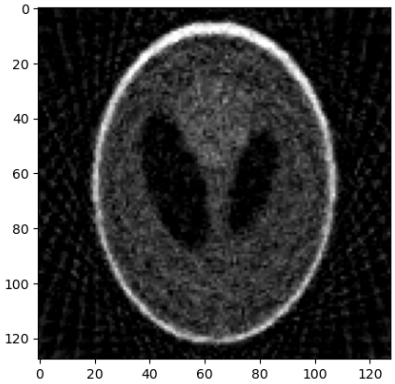
**Experiments**

*Global Parameters:* $\alpha = 0.5, \lambda = 1.5L, t = 10^{-3}, \mathrm{it}_{max} = 1000$

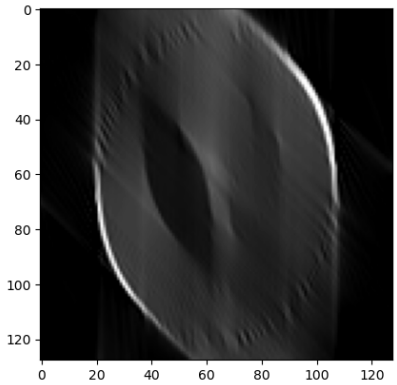
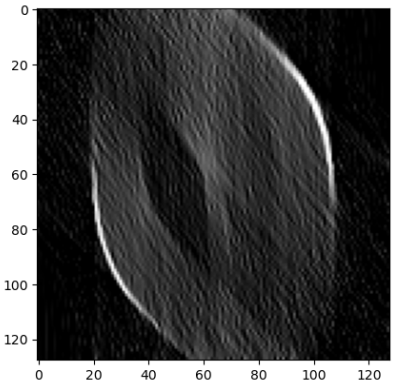
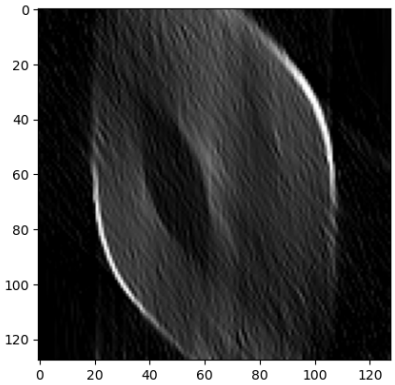
- *Radon Transform parameters:* $n_{\text{projs}} = 128$, $n_{\text{angles}} = 180$, $\theta_{\max} = \pi$

| Noise Level 0 | Noise Level 500 | Noise Level 1000 |
| --- | --- | --- |
|  |  |  |

- *Radon Transform parameters:* $n_{\text{projs}} = 128$, $n_{\text{angles}} = 30$, $\theta_{\max} = \pi$

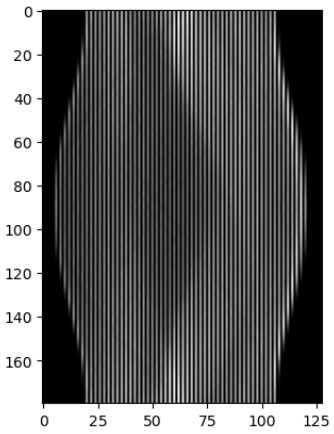
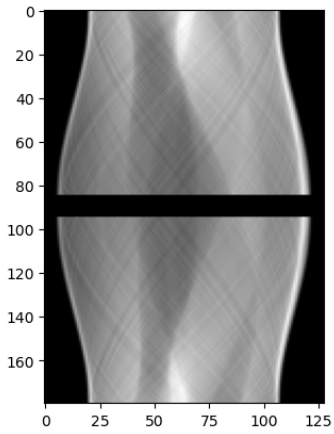
| Noise Level 0 | Noise Level 500 | Noise Level 1000 |
| --- | --- | --- |
|  |  |  |

- *Radon Transform parameters:* $n_{\text{projs}} = 128$, $n_{\text{angles}} = 90$, $\theta_{\max} = \pi/4$

| Noise Level 0 | Noise Level 500 | Noise Level 1000 |
| --- | --- | --- |
|  |  |  |

# Part B: Inpainting in Sinogram Space

## Create sinograms

| full angle with undersampled projections | limited angles |
| --- | --- |
|  |  |

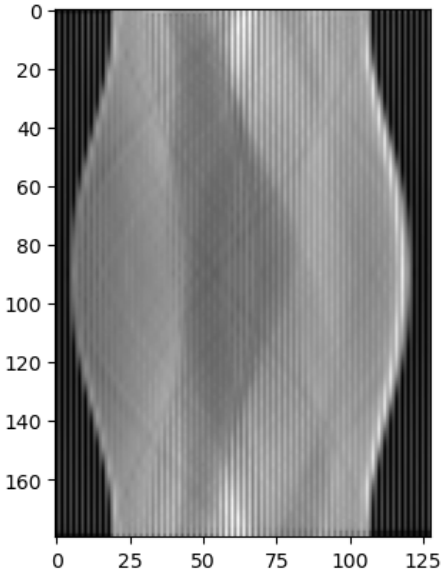
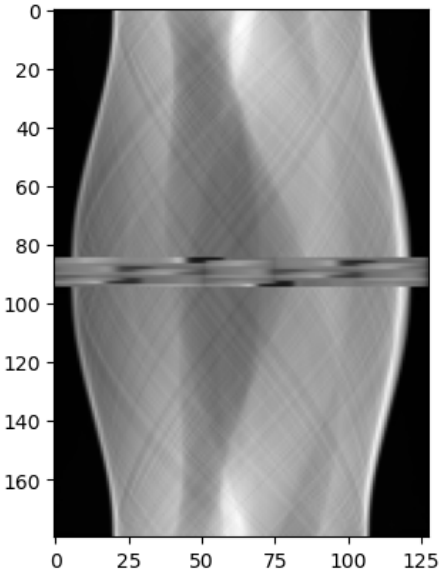
## Inpainting based on an istotropic Laplacian

Solve the augmented equation:

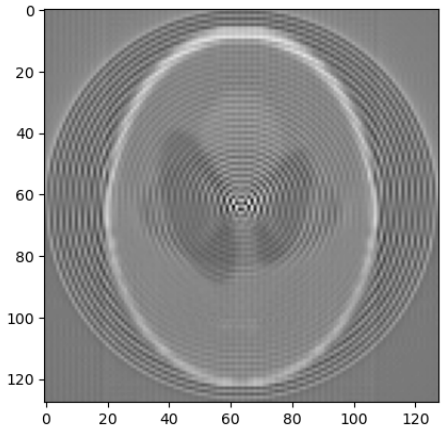
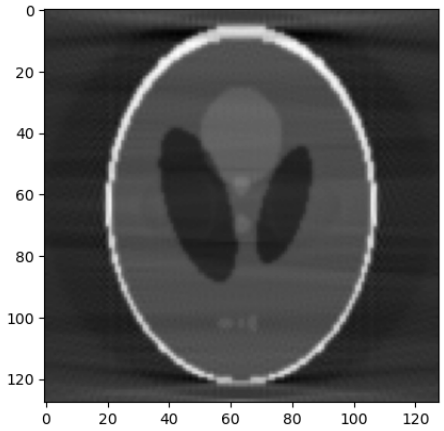
$$\begin{pmatrix} \mathbf{A} \\ \sqrt{\alpha}\nabla\mathbf{x} \\ \sqrt{\alpha}\nabla\mathbf{y} \end{pmatrix} \mathbf{f}_* = \begin{pmatrix} \mathbf{g} \\ 0 \\ 0 \end{pmatrix} \tag{4}$$

, where $\mathbf{A}$ is the masking matrix which can remove corresponding missing parts in an image and $\mathbf{g} = \mathbf{A}\mathbf{f}_{\text{true}}$.

Inpainting result:

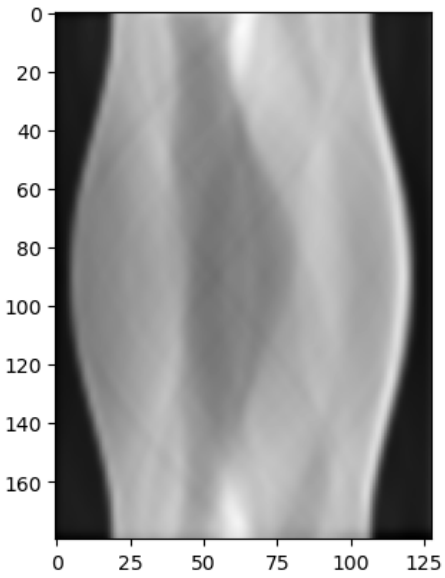
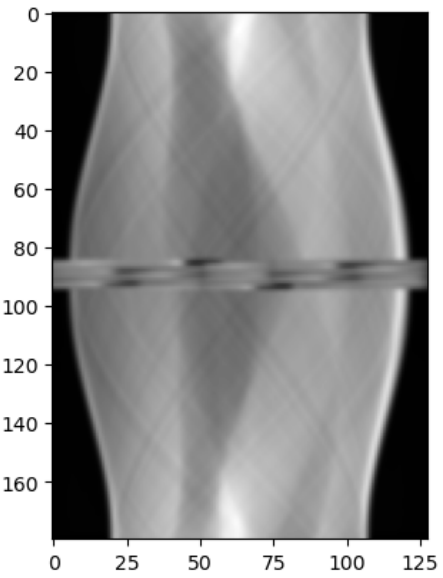
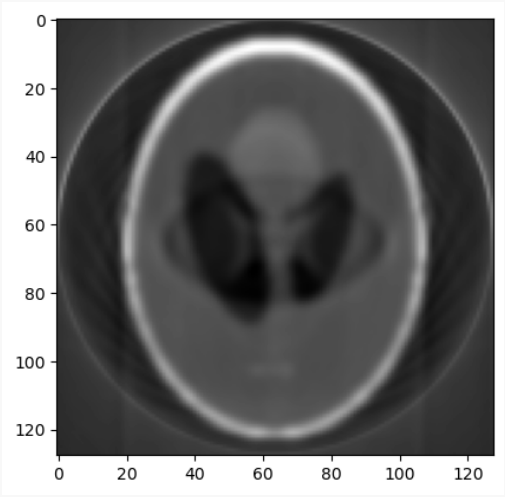
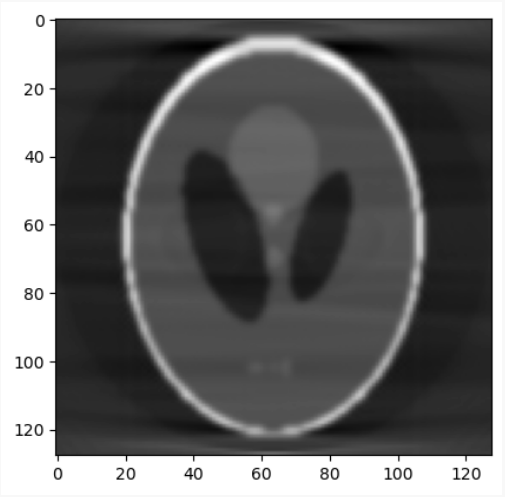
| full angle with undersampled projections | limited angles |
| --- | --- |
|  |  |

Reconstruct by corrected sinograms:

| full angle with undersampled projections | limited angles |
| --- | --- |
|  |  |

## Filtering corrected sinogram

I use gaussian blur to filter the corrected sinograms.

| | full angle with undersampled projections | limited angles |
| --- | --- | --- |
| **Filtered** |  |  |
| **Reconstruction** |  |  |

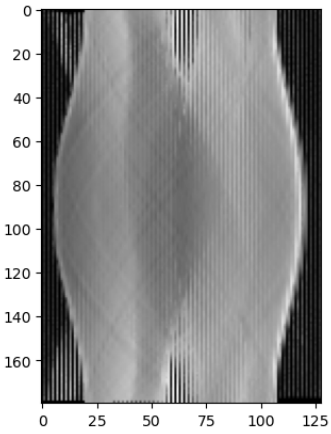
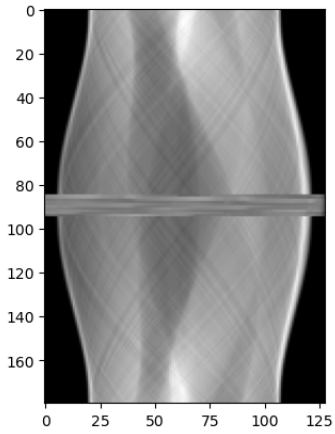
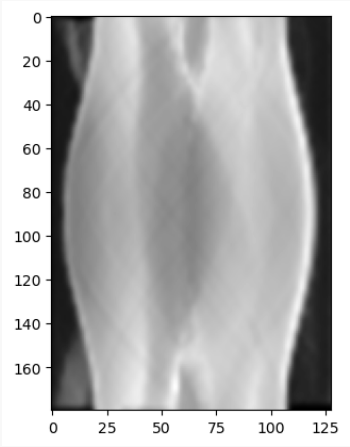
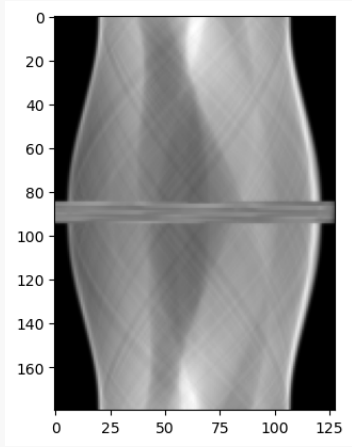
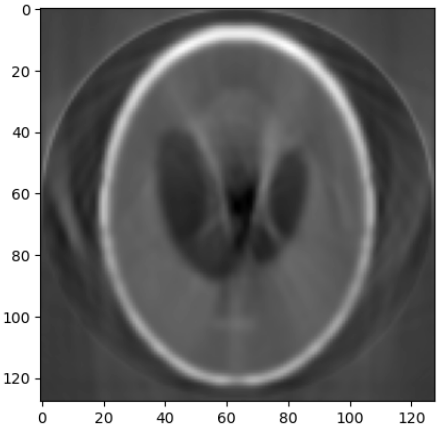
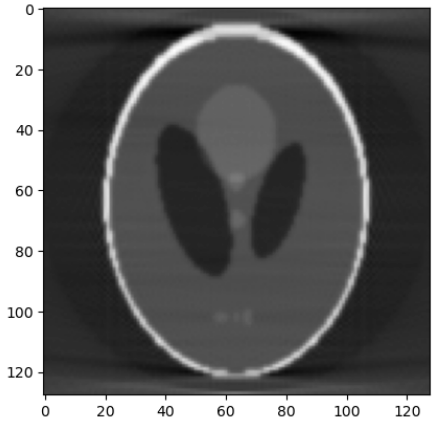
The filter has a big improvement on full angle with undersampled projections sinogram.

## Inpainting based on anisotropic regulariser

Solve the augmented equation:

$$\begin{pmatrix} \mathbf{A} \\ \sqrt{\gamma}\nabla\mathbf{x} \\ \sqrt{\gamma}\nabla\mathbf{y} \end{pmatrix} \mathbf{f}_* = \begin{pmatrix} \mathbf{g} \\ 0 \\ 0 \end{pmatrix} \tag{5}$$

, where $\gamma(\mathbf{f}) = \exp(-\sqrt{(\nabla_x\mathbf{f})^2 + (\nabla_y\mathbf{f})^2}/T)$ and $T = ||(\nabla_x\mathbf{f})^2 + (\nabla_y\mathbf{f})^2||_2/1000$.

| | full angle with undersampled projections | limited angles |
|---|---|---|
| **Corrected** |  |  |
| **Filtered** |  |  |
| **Reconstruction** |  |  |

This method improves the performance on limited angles, but it does not improve the effect of full angle with undersampled projections too much.