

**人工智能学院**

**智能医学影像技术实验报告**



**姓 名：**

**学 号：**

**日 期：** 2024/11/28

实验三 基于CT重建的计算机模拟实验

1. 实验目的

* 理解CT重建的基本原理，包括投影数据的获取和图像重建算法。
  + 基本原理：CT重建基于物体对 X 射线的衰减特性。从不同角度获取物体的投影数据，然后利用重建算法还原出物体内部的密度分布图像。常用的重建算法有滤波反投影算法（FBP）和迭代重建算法等。
* 通过模拟实验，对比不同重建算法的性能，如重建图像的质量、计算效率等。

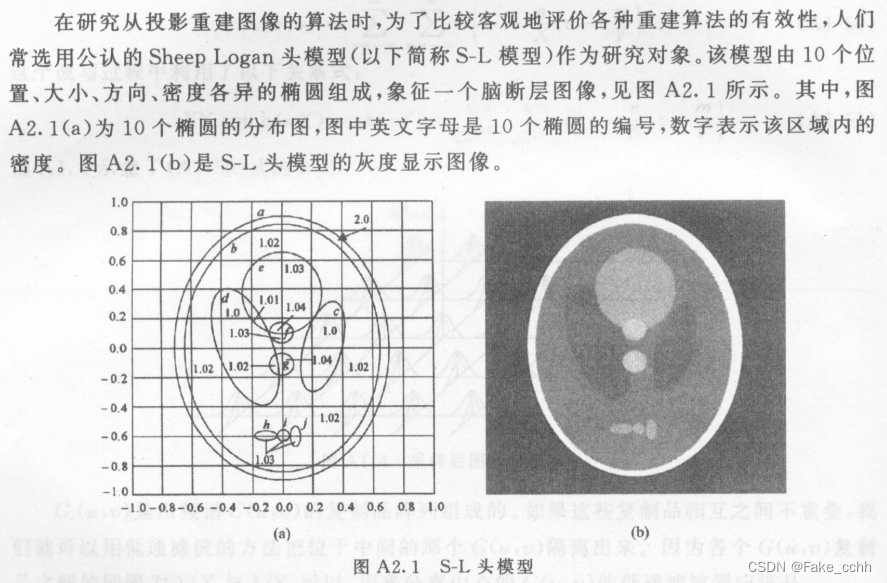
1. 实验设备与环境

* 软件工具：MATLAB、Python（NumPy, SciPy, PyTorch等）、或其他科学计算软件。
* 硬件要求：一台性能良好的计算机，足够的内存和处理能力来运行模拟和重建算法。

1. 实验内容与步骤

1.生成模拟物体模型（20分）

* 在二维平面上，创建Shepp-Logan phantom，用矩阵表示物体的密度分布，每个像素点的值代表该点的密度。



参考：

[【医学成像原理实验——CT重建】1.产生shepp-logan模型（C++(VS的MFC)和matlab）-CSDN博客](https://blog.csdn.net/m0_47979560/article/details/133812654)

[CT典型数据——shepp\_logan体模数据的生成 python版本\_phantominator-CSDN博客](https://blog.csdn.net/u013832707/article/details/112555161)

操作：

Shepp-Logan phantom 是一个常用于医学成像算法测试的标准模型，通常用于CT体模。我通过导入phantominator库（一个用于生成医学成像体模（phantoms）的库）种的shepp\_logan函数创建Shepp-Logan phantom。

最终，phantom 矩阵中的每个元素的值即代表该点的密度。代码如下：

import numpy as np

import phantominator

from phantominator import shepp\_logan

phantom = shepp\_logan(512)

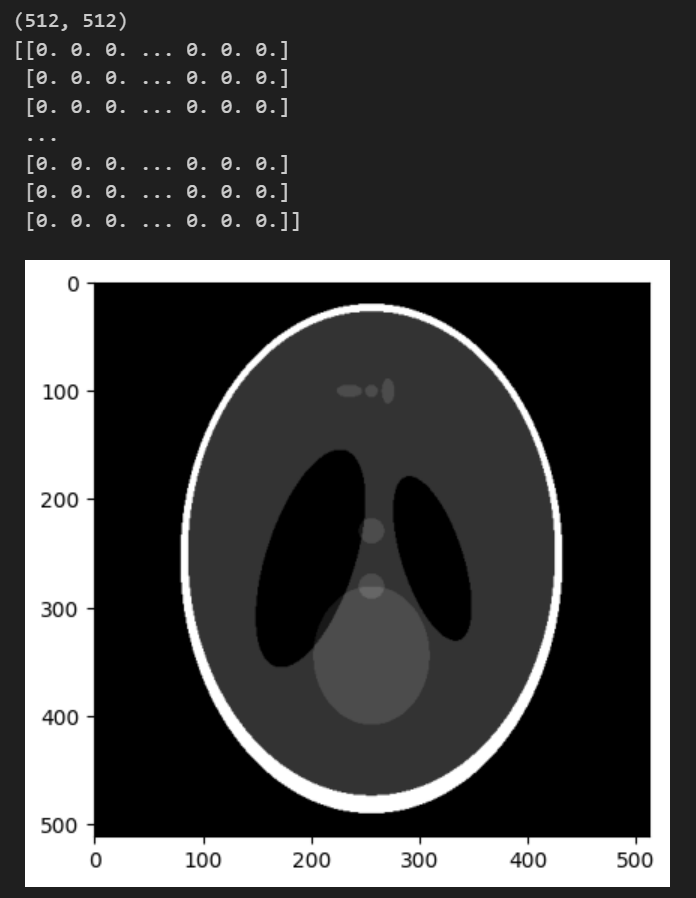
print(phantom.shape)

print(phantom)

plt.imshow(phantom, cmap='gray')

plt.show()

结果如下：



2.模拟 X 射线投影数据获取（30分）

* 选择一定数量的投影角度（如 0° - 180°，每隔 1° 或 2° 取一个角度）。
* 对于每个投影角度，计算 X 射线穿过物体模型时的衰减情况，得到投影数据。这可以通过对物体模型中沿 X 射线传播路径上的像素密度进行积分（模拟 X 射线的衰减过程）来实现。

1. 使用 Radon 变换（SciPy 库中的 radon 函数）来获取投影数据

#### 2. 模拟 X 射线投影数据

# 选择投影角度

theta = np.arange(0, 180, 1)  # 从 0° 到 179°，每隔 1° 取一个角度

# 计算 Radon 变换（X 射线投影数据）

sinogram = radon(phantom, theta=theta)

# 可视化投影数据

plt.figure(figsize=(10, 6))

plt.imshow(sinogram, cmap='gray', aspect='auto', extent=(0, 180, 0, sinogram.shape[0]))

plt.title("X-ray Projections (Radon Transform)")

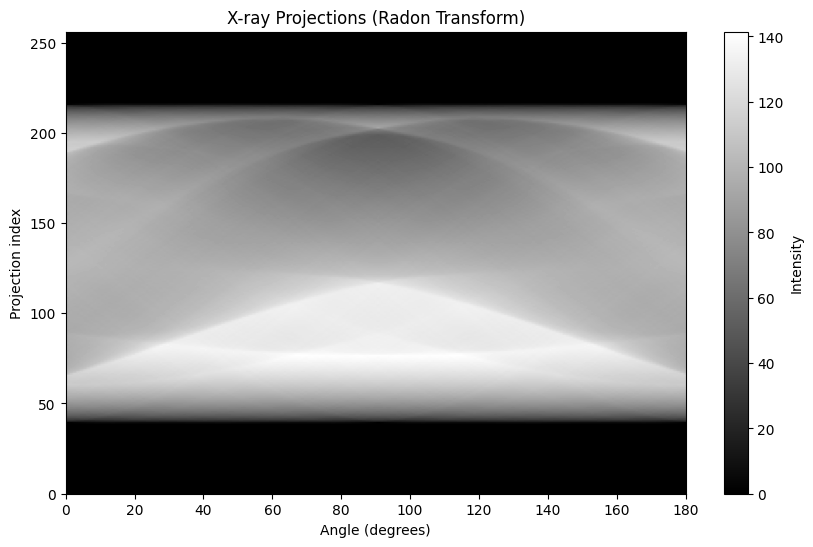
plt.xlabel("Angle (degrees)")

plt.ylabel("Projection index")

plt.colorbar(label='Intensity')

plt.show()

在实验中，我选择了选择了从 0° 到 180° 的投影角度，每隔 1° 取一个角度，并使用了SciPy库的radon 函数计算 X 射线的衰减情况，得到投影数据（sinogram）；最后使用 Matplotlib 可视化 Shepp-Logan phantom 和 X 射线投影数据，如下图。



1. 通过Shepp-Logan phantom前向投影和使用forward\_projection函数计算X 射线投影数据

theta = np.arange(0, 180, 1)

在实验中，我同样选择了从 0° 到 180° 的投影角度，每隔 1° 取一个角度。

def forward\_projection(theta\_proj, N, N\_d):

    shep = np.array( #定义 Shepp-Logan Phantom 参数

        [

            [1, 0.69, 0.92, 0, 0, 0],

            [-0.8, 0.6624, 0.8740, 0, -0.0184, 0],

            [-0.2, 0.1100, 0.3100, 0.22, 0, -18],

            [-0.2, 0.1600, 0.4100, -0.22, 0, 18],

            [0.1, 0.2100, 0.2500, 0, 0.35, 0],

            [0.1, 0.0460, 0.0460, 0, 0.1, 0],

            [0.1, 0.0460, 0.0460, 0, 0.1, 0],

            [0.1, 0.0460, 0.0230, -0.08, -0.605, 0],

            [0.1, 0.0230, 0.0230, 0, -0.606, 0],

            [0.1, 0.0230, 0.0460, 0.06, -0.605, 0],

        ]

    )

    # 初始化变量

    theta\_num = len(theta\_proj)

    P = np.zeros((int(N\_d), theta\_num))

    rho = shep[:, 0]

    ae = 0.5 \* N \* shep[:, 1]

    be = 0.5 \* N \* shep[:, 2]

    xe = 0.5 \* N \* shep[:, 3]

    ye = 0.5 \* N \* shep[:, 4]

    alpha = shep[:, 5]

    alpha = alpha \* np.pi / 180 # 转换为弧度

    theta\_proj = theta\_proj \* np.pi / 180  # 转换为弧度

    TT = np.arange(-(N\_d - 1) / 2, (N\_d - 1) / 2 + 1)

    # 计算前向投影

    for k1 in range(theta\_num): # 遍历每个投影角度k1

        P\_theta = np.zeros(int(N\_d))

        for k2 in range(len(xe)): # 遍历每个椭圆k2，计算每个椭圆对当前角度投影的贡献

            # 计算椭圆在当前角度的有效面积

            a = (ae[k2] \* np.cos(theta\_proj[k1] - alpha[k2])) \*\* 2 + (

                be[k2] \* np.sin(theta\_proj[k1] - alpha[k2])

            ) \*\* 2

            # 计算当前投影线与椭圆的交点

            temp = (

                a

                - (

                    TT

                    - xe[k2] \* np.cos(theta\_proj[k1])

                    - ye[k2] \* np.sin(theta\_proj[k1])

                )

                \*\* 2

            )

            #用于索引有效交点，计算投影值 P\_theta

            ind = temp > 0

            P\_theta[ind] += rho[k2] \* (2 \* ae[k2] \* be[k2] \* np.sqrt(temp[ind])) / a

        P[:, k1] = P\_theta

    # 归一化投影数据,确保投影值在 0 到 1 的范围内

    P\_min = np.min(P)

    P\_max = np.max(P)

    P = (P - P\_min) / (P\_max - P\_min)

    return P

我先将Shepp-Logan Phantom 的参数定义为一个数组（其中每一行代表一个椭圆的属性——密度、长轴、短轴、位置和旋转角度），后用forward\_projection 函数遍历每个投影角度，计算每个椭圆对当前投影的贡献。这一过程模拟了 X 射线穿过物体模型时的衰减情况。

在函数内部，通过使用椭圆的几何参数和密度信息计算沿 X 射线传播路径上的像素密度的积分，即对每个椭圆在当前角度的有效面积进行积分，给出投影角度下的投影数据。

之后对投影数据归一化，以确保值在 0 到 1 的范围内，方便后续可视化。

3.重建算法实现（30分）

* 基于滤波反投影的方法
  1. 对获取的投影数据进行滤波处理，常用的滤波器有 Ram - Lak 滤波器、Shepp - Logan 滤波器等。
  2. 然后将滤波后的投影数据进行反投影，将各个角度的投影信息还原到图像空间，得到重建图像。

1. 使用skimage库的iradon 函数进行反投影，默认使用Ram-Lak 滤波器处理输入的 sinogram（sinogram是使用radon函数计算的投影数据）。

def apply\_filter\_and\_backprojection(sinogram, theta):

    """应用 Ram-Lak 滤波器并进行反投影"""

    # iradon函数在执行反投影时，会默认应用Ram-Lak滤波器

    reconstructed\_image = iradon(sinogram, theta=theta, interpolation='linear')

    return reconstructed\_image

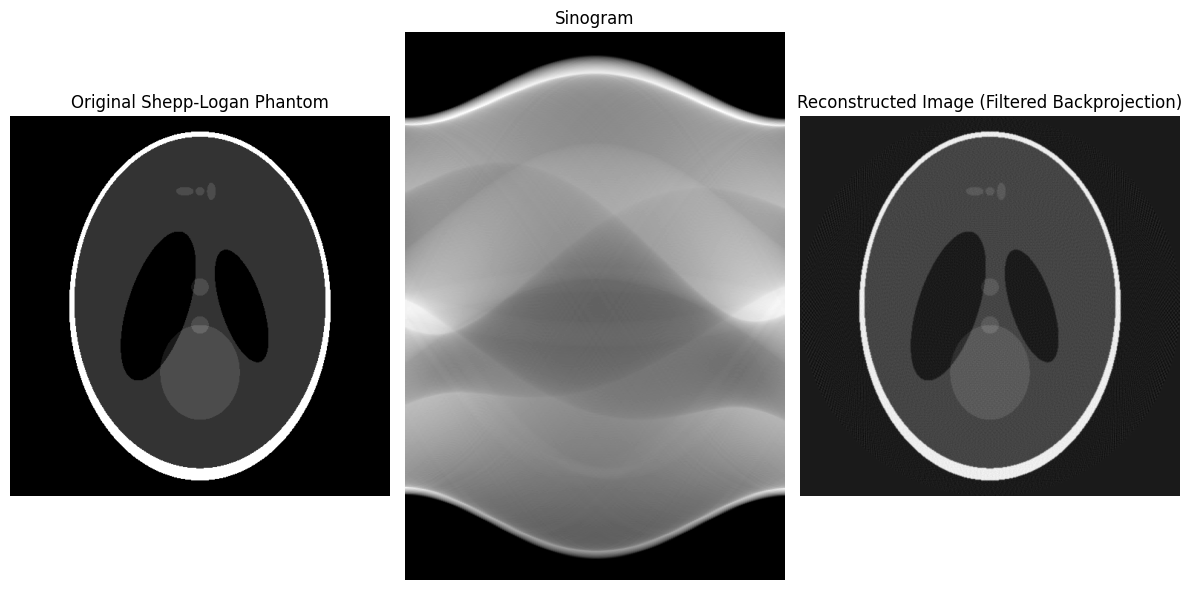
# 获取投影数据

theta = np.arange(0, 180, 1)

sinogram = radon(phantom, theta=theta)

# 使用滤波反投影方法重建图像

reconstructed\_image\_fb = apply\_filter\_and\_backprojection(sinogram, theta)



1. 手动实现滤波和反投影

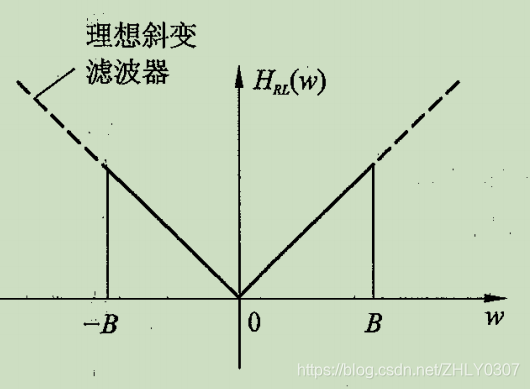
参考：

[CT图像重构方法详解——傅里叶逆变换法、直接反投影法、滤波反投影法-CSDN博客](https://blog.csdn.net/qq_33924470/article/details/106874678)

[图像重建中常用的滤波器的设计\_sl 滤波器是使用 sinc 函数对斜坡滤波器进行截断产生-CSDN博客](https://blog.csdn.net/ZHLY0307/article/details/119419167)

1. 滤波器定义：

* Ram-Lak 滤波器 (RLFilter): RL滤波器是使用窗函数对斜坡滤波器进行截断产生的，如下图。



在使用该滤波器的时候，需要将其离散化，离散化之后的滤波器的函数表达式为：

通过计算频域滤波器的值来实现，使用负的二次函数调整滤波器的形状。

def RLFilter(N, d):

    filterRL = np.zeros((N,))

    for i in range(N):

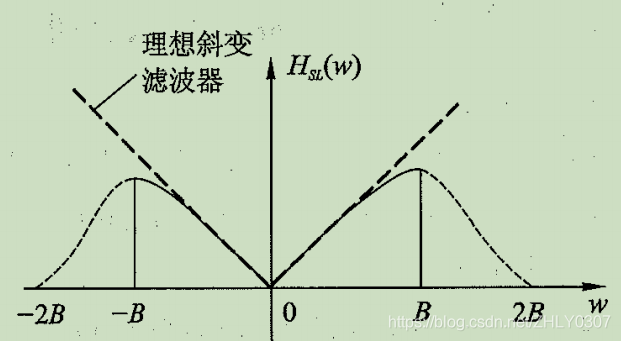
        filterRL[i] = -1.0 / np.power((i - N / 2) \* np.pi \* d, 2.0)

        if np.mod(i - N / 2, 2) == 0:

            filterRL[i] = 0

    filterRL[int(N / 2)] = 1 / (4 \* np.power(d, 2.0)) #中心频率的值被设置为1/(4 \* d^2)

    return filterRL

* Shepp-Logan 滤波器 (SLFilter): SL滤波器是使用Sinc函数对斜坡滤波器进行截断产生的，如下图。

在使用该滤波器的时候，同样需要将其离散化，离散化之后的滤波器的函数表达式为：

通过公式计算滤波器的值，强调在特定频率下的响应。

def SLFilter(N, d):

    filterSL = np.zeros((N,))

    for i in range(N):

        # filterSL[i] = - 2 / (np.power(np.pi, 2.0) \* np.power(d, 2.0) \* (np.power((4 \* (i - N / 2)), 2.0) - 1))

        filterSL[i] = -2 / (np.pi\*\*2.0 \* d\*\*2.0 \* (4 \* (i - N / 2) \*\* 2.0 - 1))

    return filterSL

1. 图像重建函数：

通过RL\_Transform和SL\_Transform这两个函数分别实现了基于 Ram-Lak 和 Shepp-Logan 滤波器的滤波反投影过程。

对每个投影角度，首先对投影数据进行滤波，然后将其展开并旋转，最后进行累加以得到重建图像。代码如下：

# 2）定义用于存储重建后的图像的数组

channels = 512

def RL\_Transform(image, steps):

    #channels = len(image[0])

    # print(channels)

    origin = np.zeros((steps, channels, channels))

    # filter = RLFilter(channels, 1)

    filter = RLFilter(channels, 1)

    for i in range(steps):

        projectionValue = image[:, i]

        projectionValueFiltered = convolve(filter, projectionValue, "same")

        projectionValueExpandDim = np.expand\_dims(projectionValueFiltered, axis=0)

        projectionValueRepeat = projectionValueExpandDim.repeat(channels, axis=0)

        origin[i] = ndimage.rotate(

            projectionValueRepeat, i \* 180 / steps, reshape=False

        ).astype(np.float64)

    iradon = np.sum(origin, axis=0)

    return iradon

def SL\_Transform(image, steps):

    # channels = len(image[0])

    origin = np.zeros((steps, channels, channels))

    # filter = RLFilter(channels, 1)

    filter = SLFilter(channels, 1)

    for i in range(steps):

        projectionValue = image[:, i]

        projectionValueFiltered = convolve(filter, projectionValue, "same")

        projectionValueExpandDim = np.expand\_dims(projectionValueFiltered, axis=0)

        projectionValueRepeat = projectionValueExpandDim.repeat(channels, axis=0)

        origin[i] = ndimage.rotate(

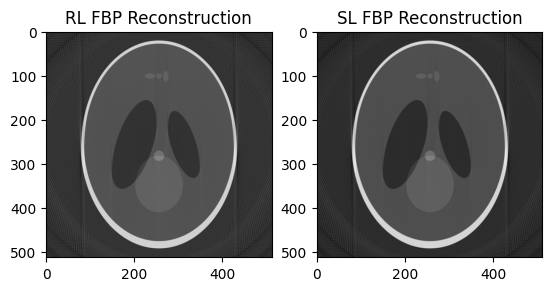
            projectionValueRepeat, i \* 180 / steps, reshape=False

        ).astype(np.float64)

    iradon = np.sum(origin, axis=0)

    return iradon

结果如图所示：



* 基于代数重建技术 ART的方法
  1. 初始化一个重建图像（通常为全零矩阵或具有一定初始猜测值的矩阵）。为所有像素赋值初始值，即在第0次迭代时图像全部为零。

# 初始化重建图像

x0 = np.zeros\_like(phantom)

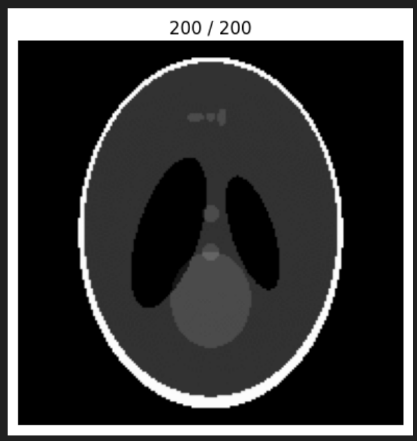
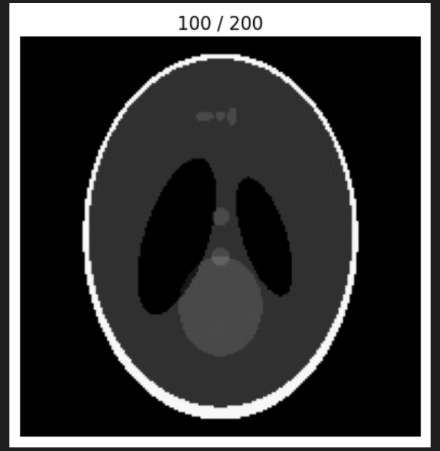
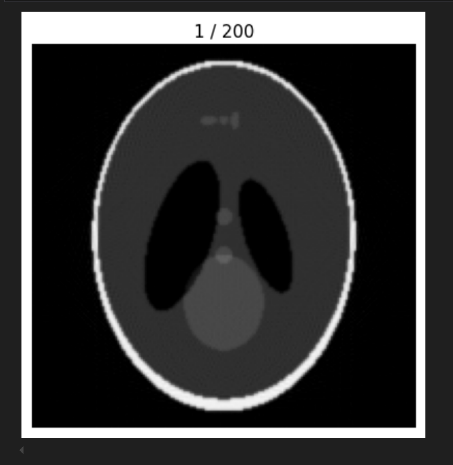
* 1. 对于每个投影角度，根据投影数据和当前重建图像计算误差，然后更新重建图像，重复多次迭代过程，直到满足收敛条件（如重建图像的变化小于一定阈值）。

    for i in range(int(niter)):

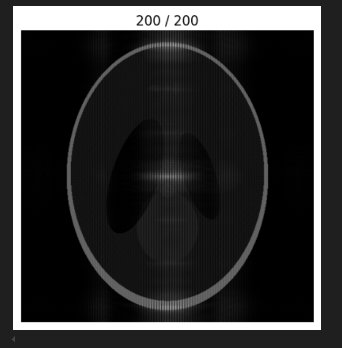
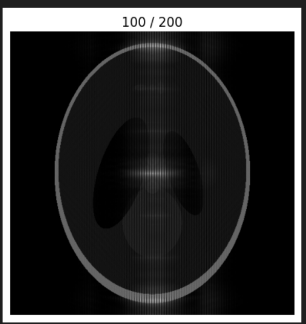
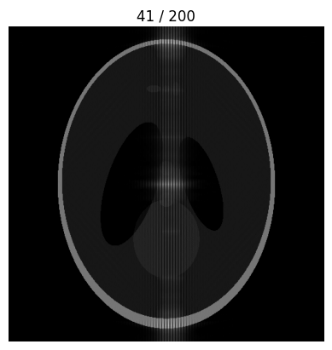
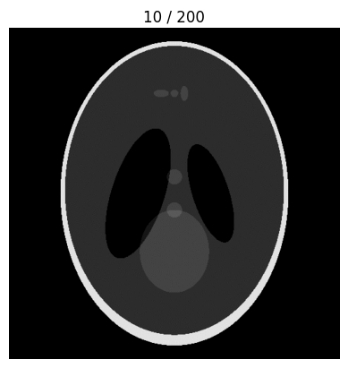
        # np.divide计算更新量

        x = x + np.divide(mu \* AT(b - A(x)), ATA)

* 计算误差：b - A(x)计算实际投影与估计投影之间的误差
* 计算修正值：
  + 修正值通过np.divide(mu \* AT(b - A(x)), ATA)函数，对反转变换 AT 和当前图像的差异计算得出。
  + 在每次迭代中，**只对射线穿过的像素进行修正**。未被射线穿过的像素在该次迭代中不会受到影响，因此修正值对于这些像素为零。
* 计算估计投影值，并更新像素值：



当图片尺寸为180\*180时，整个重建过程速度较快，约在10分钟左右，结果如上图。ART重建算法并不是很明显。



当图片尺寸为180\*180时，整个重建过程较慢，大约在30分钟左右，结果如上图。ART重建算法相对较为明显，迭代41次左右时出现振荡，迭代100次以上效果差距不大。

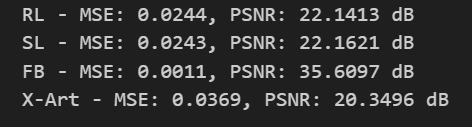
分析出现伪影的原因：ART 是一种迭代算法，其每次更新都基于当前估计的图像。如果某些像素的更新不够平滑，可能会在图像中产生明显的条纹。

4. 结果评估与比较（20分）

* 计算重建图像与原始物体模型之间的误差指标，如均方误差（MSE）、峰值信噪比（PSNR）等并绘制误差曲线。
* 通过可视化对比原始物体模型、不同算法重建的图像，观察图像的清晰度、伪影情况等。（见五、实验结果）

1. 该图为直接调用skimage.metrics中mean\_squared\_error, peak\_signal\_noise\_ratio

函数得到的结果。



1. 手动实现

# 计算均方误差 (MSE)

def mse(image1, image2):

    return np.mean((image1 - image2) \*\* 2)

# 计算峰值信噪比 (PSNR)

def psnr(image1, image2):

    mse\_value = mse(image1, image2)

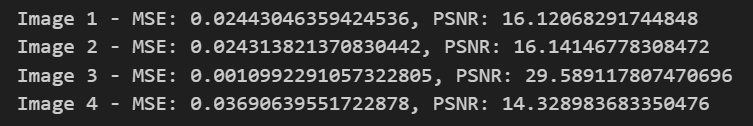
    if mse\_value == 0:

        return 100  # 如果没有误差，返回一个高值（理论上的 PSNR）

    max\_pixel = 1.0  # 假设图像像素值范围是 [0, 1]

    return 20 \* np.log10(max\_pixel / np.sqrt(mse\_value))

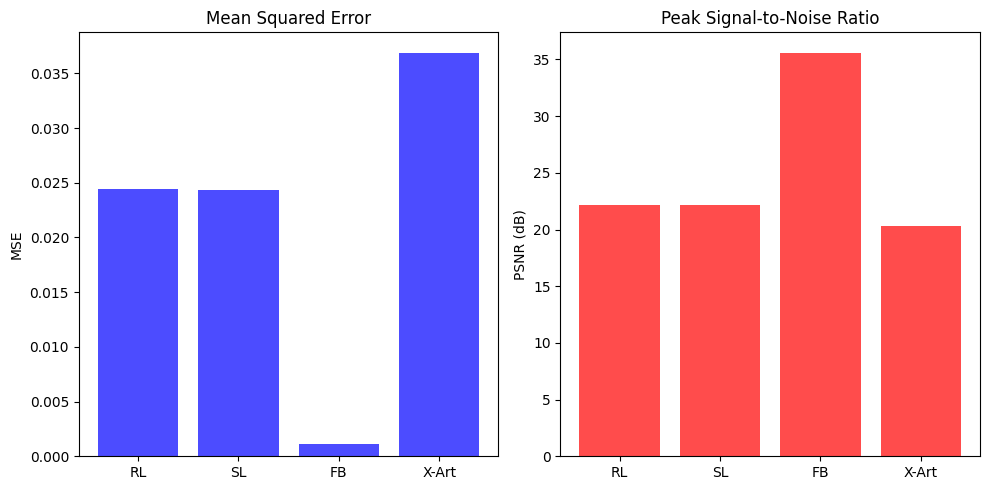
结果如下：

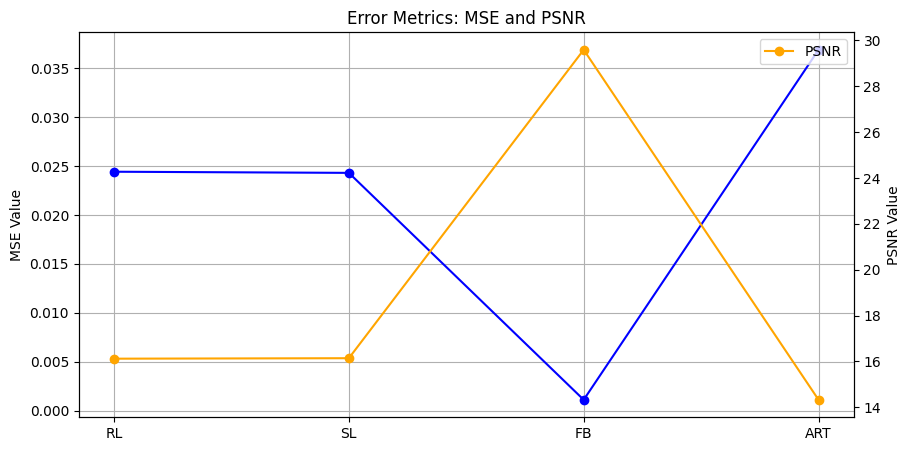
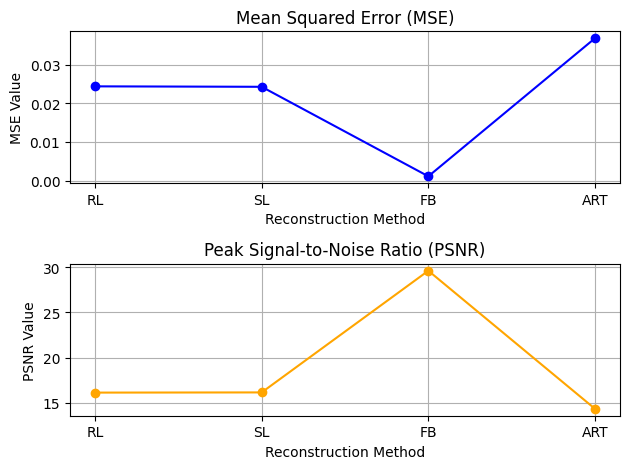


1. 源程序

见附件CT\_Reconstruction.ipynb。

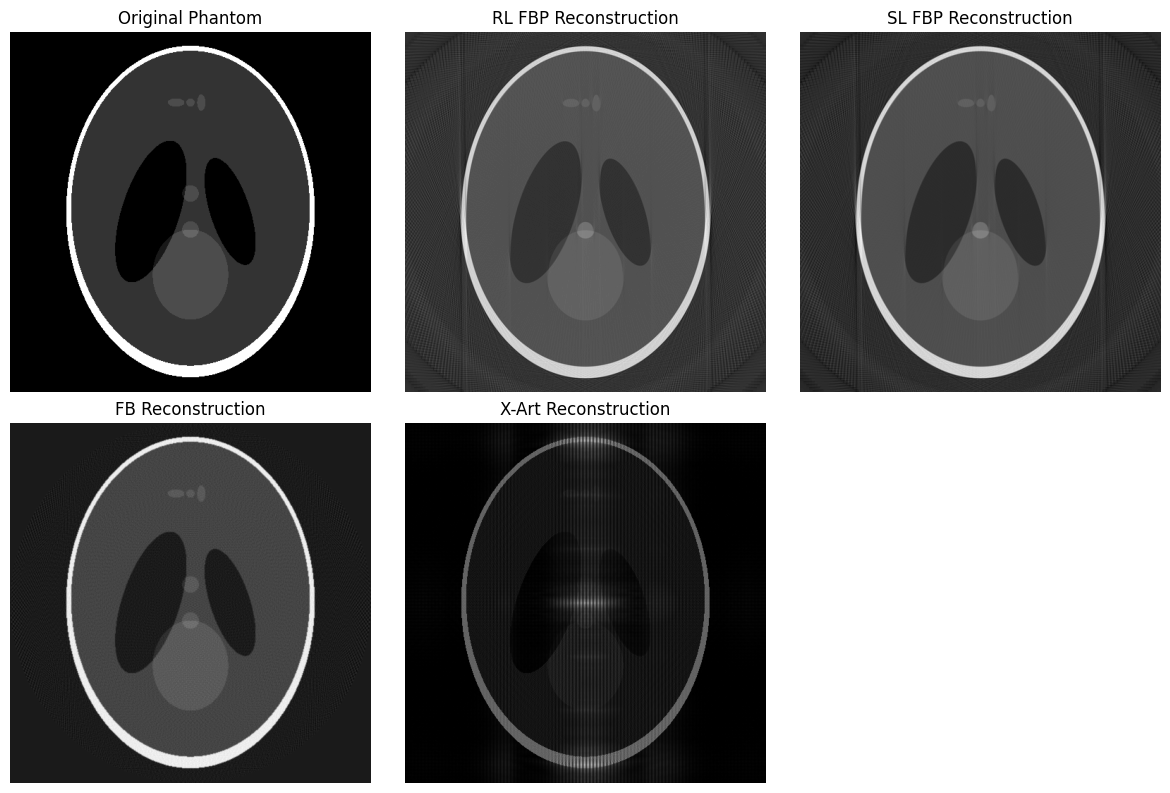
1. 实验结果



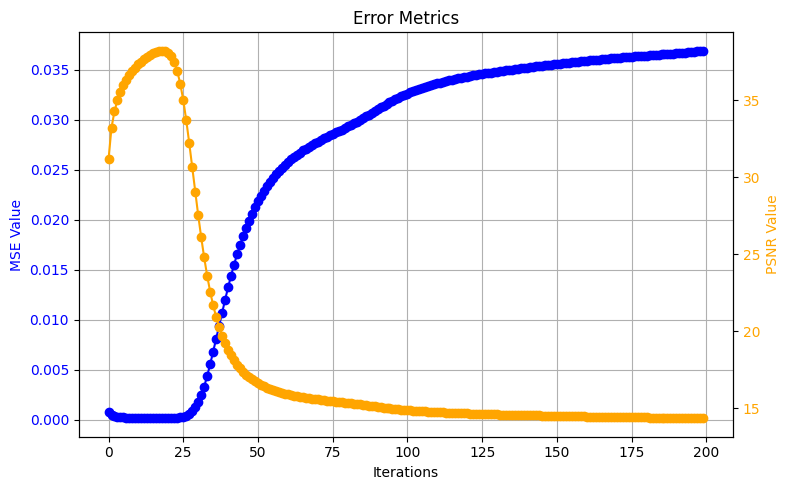


通过误差曲线图和柱状图可以得出：

1. FB 算法的 MSE 最小（0.0011），说明其重建效果最好，差异最小；X-Art 算法的 MSE 最大（0.0369），说明其重建效果相对较差。
2. FB 算法的 PSNR 值最高（35.6097 dB），表明其重建图像的质量最佳；X-Art 算法的 PSNR 值最低（20.3496 dB），表示重建质量较差。



观察五种不同算法得出的结果，发现FB算法通常能够有效减少伪影，而 X-Art算法可能会在重建图像中显现出明显的伪影。（伪影是重建过程中产生的伪影或不自然的图像特征，通常在 MSE 较高或 PSNR 较低的图像中更明显。）



在ART重建迭代过程中，误差曲线如上图。

1. 实验总结
2. **不同算法的优缺点**

滤波算法的好处在于，把两次二维傅里叶变换变成了两次一维傅里叶变换，计算速度大大提升。于是滤波反投影法的核心问题就变成了如何选择一个合适的滤波器r。R-L滤波器和S-L滤波器主要区别如下：

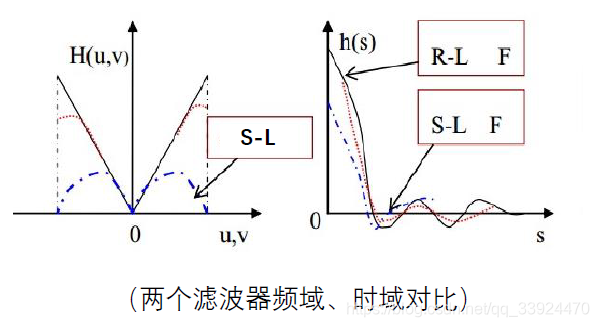
**1.1. RL（反投影法）**

* **优点**：
  + 简单易实现，计算量相对较小。
  + 适合处理低剂量投影数据。
* **缺点**：
  + 对噪声敏感，容易产生伪影。
  + 图像清晰度和细节保留能力较差。
  + R-L滤波函数是一个V字形，由于后面的直接截断，会导致吉布斯现象（重构图像存在振荡，不连续）。

**1.2. SL（Shepp-Logan 算法）**

* **优点**：
  + 在处理常见物体模型时表现良好，能够保留更多细节。
  + 比 RL 方法更有效地减少伪影。
* **缺点**：
  + 计算复杂度相对较高，尤其在大尺寸图像中。
  + 对噪声的抵抗力仍然有限。
  + S-L函数后面较为平滑，虽然减少了震荡，但是对高频的滤波效果不够理想。

可以在下图更直观地看到两个滤波器之间的区别：



**1.3. X-Art（代数重建技术）**

* **优点**：
  + 适应性强，可通过调整参数来优化重建效果。
  + 对于低剂量情况下的重建有较好的效果。
* **缺点**：
  + 迭代过程可能导致伪影，特别是在参数设置不当时。
  + 计算时间较长，尤其在迭代次数较多时。

**2. 投影数据质量对重建结果的影响**

* **完整性**：投影数据的角度覆盖范围直接影响重建效果。足够的角度分布可以有效减少伪影，提升重建图像的质量。
* **噪声**：投影数据中的噪声会被重建算法放大，导致伪影的产生。高噪声水平会降低图像的清晰度，影响诊断准确性。
* **分辨率**：投影数据的分辨率影响重建图像的细节保留能力。高分辨率投影数据可以提供更多信息，帮助算法生成更清晰的图像。

**3. 算法复杂度对重建结果的影响**

* **计算复杂度**：算法的复杂度影响重建速度和对计算资源的需求。例如，滤波反投影法和代数重建技术的计算量较大，需要更多的内存和处理时间。
* **实现难度**：一些算法（如 FB 和 X-Art）实现上相对复杂，需要更高的数学和编程技能。实现不当可能导致重建效果不佳。
* **参数调整**：算法的灵活性和参数设置会影响重建效果。适当的参数可以提高图像质量，但不合理的设置可能导致伪影或图像模糊。

1. 实验收获与心得体会

不同的重建算法在性能和适用性上各有优缺点。选择合适的算法需要综合考虑投影数据的质量、计算资源的限制以及实际应用场景的需求。通过优化投影数据和算法参数设置，可以有效提升图像重建的质量，减少伪影，增强图像的临床应用价值。在未来的工作中，结合多种算法的优点，探索新的优化策略，将是提高成像质量的重要方向。

**1. 理论知识的深化**

通过本次实验，我对图像重建算法的原理有了更深入的理解。特别是不同算法在处理投影数据时的机制和优缺点，增强了我对成像技术的认识。

**2. 实践技能的提升**

在实际操作中，我学习了如何使用 Python 和相关医学成像库进行图像重建和结果可视化。通过编写和调试代码，我提高了编程能力和问题解决能力。

**3. 数据质量的重要性**

实验过程中，我深刻体会到投影数据质量对重建结果的影响。有效的投影数据不仅需要覆盖足够的角度，还需减少噪声。数据处理和预处理的步骤在整个重建过程中至关重要。

**4. 参数调整的影响**

通过实验，我认识到重建算法中的参数（如步长、迭代次数等）对结果有显著影响。合理的参数设置可以显著提高重建图像的质量，而不当的设置则可能导致伪影和模糊。这让我明白了在实际应用中，需根据具体情况灵活调整参数。

**5. 未来的研究方向**

此次实验让我意识到，图像重建技术在医学成像领域有着广泛的应用潜力。未来，我希望能进一步研究如何结合深度学习等先进技术，改进传统的重建算法，提高图像质量和重建效率。