

CT重建软件 设计报告

注：使用pdf阅读器打开本文档时，可在左侧查看标签。

第一章 软件概述

软件大小为42.6M，无需安装，为独立的exe文件。

可以通过与用户交互，实现图像的平行束滤波反投影重建的展示。

注：开发过程中，也实现了扇形束投影，但是扇形束重建由于时间原因尚未完全实现，因此没有集成到软件中。后文也会对扇形束部分的探究进行介绍。

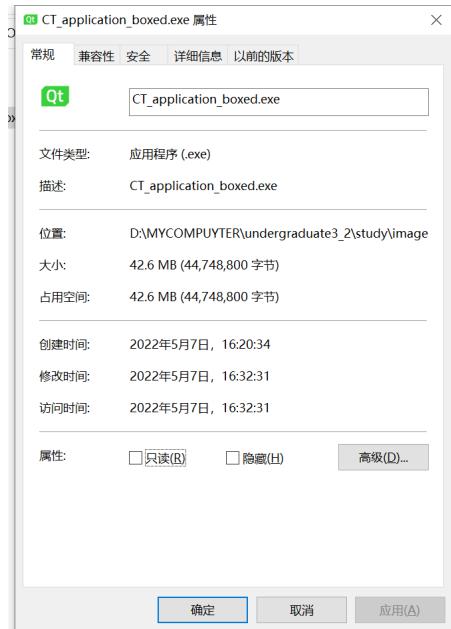


图1.1 软件参数展示

1、软件界面

软件界面展示如下。

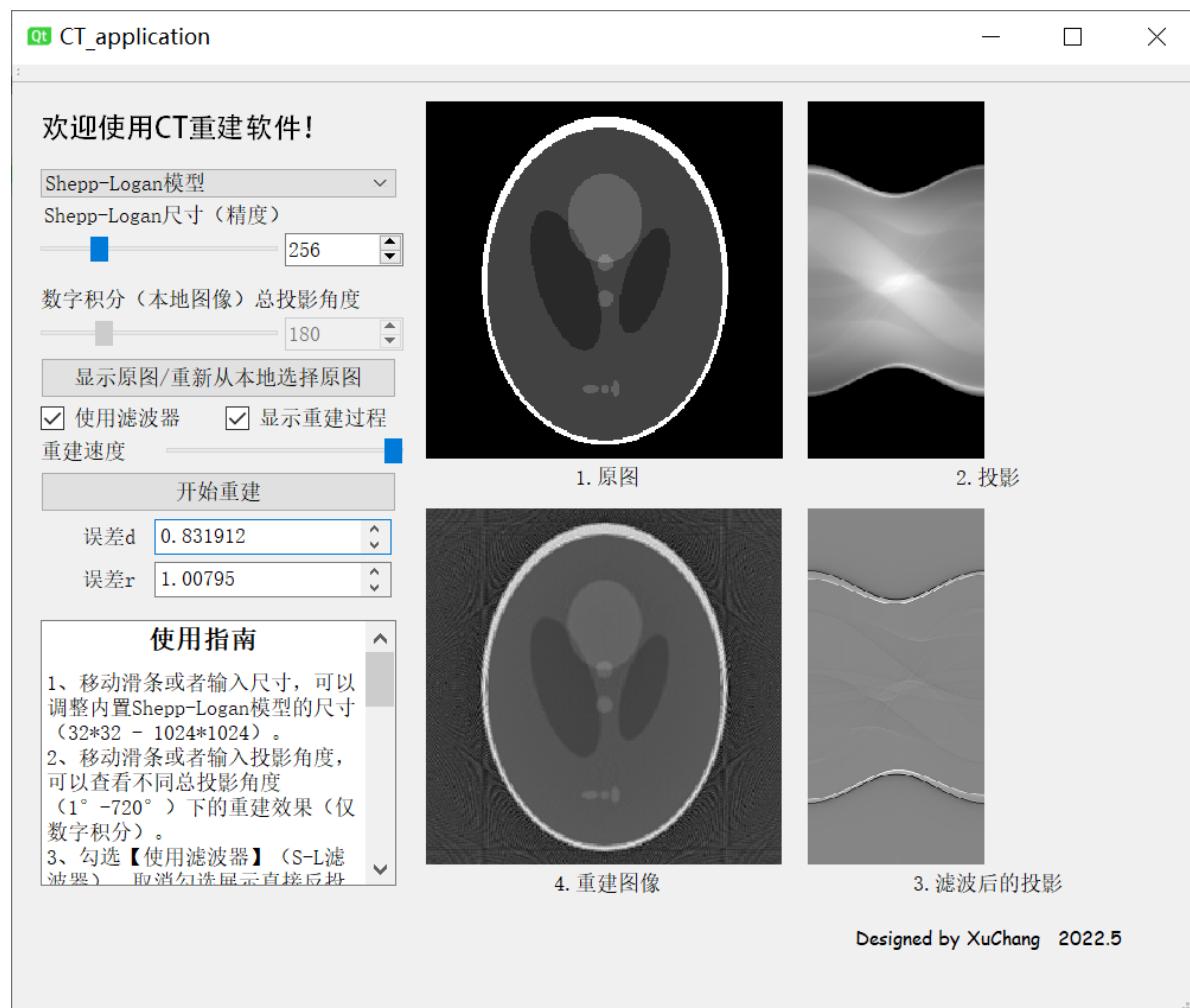


图1.2 软件界面展示 (内置Shepp-Logan模型作为输入)



图1.3 软件界面展示（彩色、长方形本地图片输入）

2、软件功能

通过与用户的交互，可以实现图像的平行束滤波反投影重建的展示。

软件功能包括：

- **可调Shepp-Logan模型的尺寸**
 - 移动滑条或者输入尺寸，可以调整内置Shepp-Logan模型的尺寸（32*32 - 1024*1024）。
- **可调总投影角度**
 - 移动滑条或者输入投影角度，可以查看不同总投影角度（1°-720°）下的重建效果（仅数字积分）。
- **滤波器勾选功能**
 - 勾选【使用滤波器】（S-L滤波器），取消勾选则展示直接反投影结果。
- **可慢速显示重建过程**
 - 勾选【显示重建过程】，可以慢速显示投影过程和反投影过程。
 - 其中投影过程中，Shepp-Logan模型的投影为椭圆的依次叠加，本地图片为各角度投影结果。
- **可调重建速度**
 - 【重建速度】为“显示重建过程”时的展示速度，滑块越往右，展示的重建速度越快。
- **支持多图片类型输入**
 - 本地图片支持彩色输入（输出为灰度图），支持长方形图片输入，图像类型支持*.png, *.jpg, *.bmp。
- **误差透明化**
 - 原图和重建图像之间的误差用误差d（归一化均方距离判据）和误差r（归一化平均绝对距离判据）表示，在重建图像后会自动计算显示。注：图像之间进行的比较时，已归一化到0-255。
- **用户友好**
 - 鼠标在按键或者图像上停留，可以查看相应的说明。

3、软件特色概述

软件的特色包括：

- 用户交互性强，支持多个参数的修改。
- 在各位置鼠标停留，会弹出使用说明。
- 本地图片支持彩色输入（输出为灰度图），支持长方形图片输入，图像类型支持*.png, *.jpg, *.bmp。
- 提供重建评价指标，便于重建效果评估。
- 代码清晰，注释完整。
- 接口简洁，便于后续进一步开发。

注：软件特色在后文还会进行详细介绍。

第二章 软件开发平台简介

1、开发环境及版本

- Visual Studio 2019 (C++)：用于编程和项目管理；
- openCV 4.4.0：图像处理的工具；
- Qt 5.12.2：界面设计的工具。

2、其他工具

- [Enigma Virtual Box](#)：打包生成独立的exe。

3、开发界面展示

开发界面展示如下：

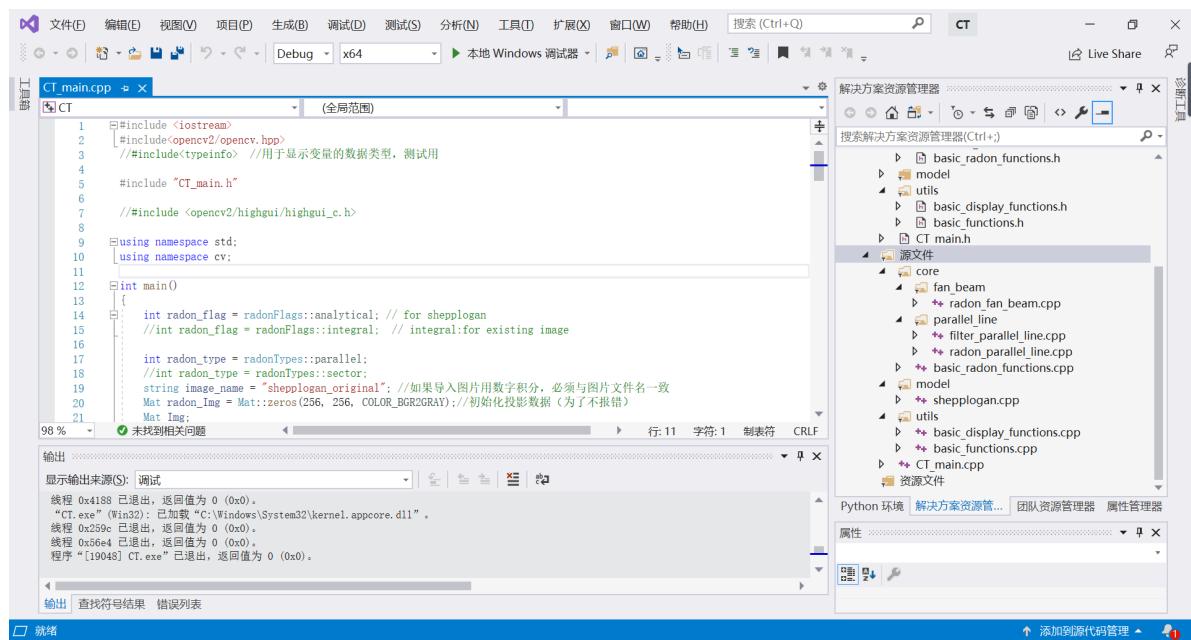


图2.1 Visual Studio环境

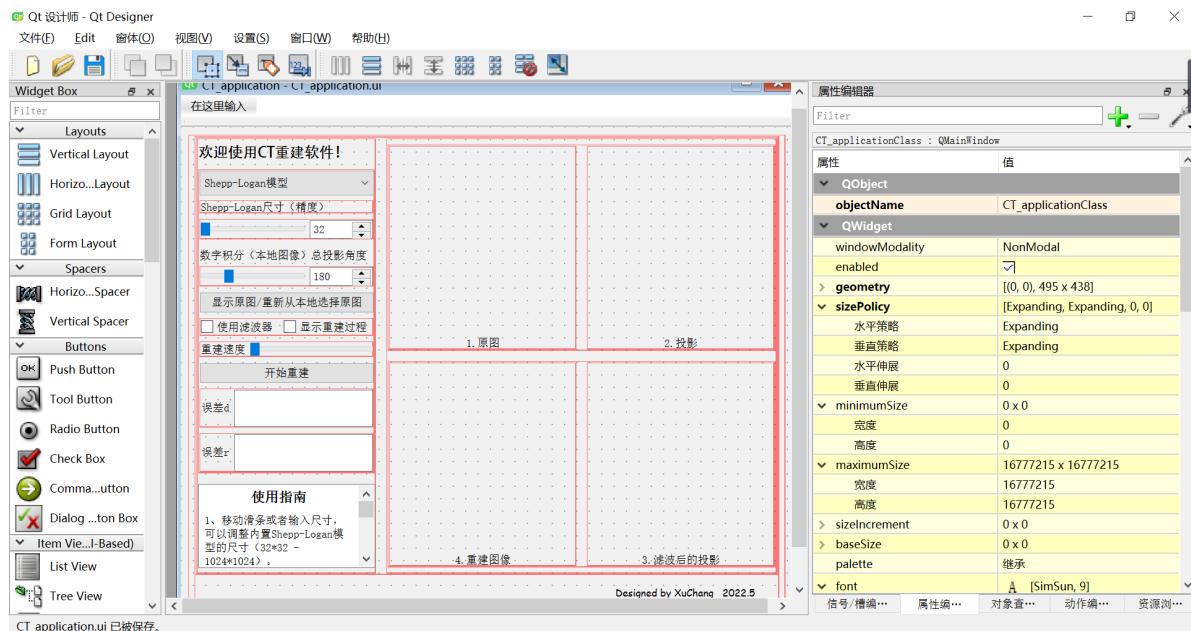


图2.2 Qt Designer界面设计

第三章 代码整体架构

1、逻辑架构

代码的整体逻辑架构整理如下：

```
└── core/
    ├── parallel_line
    │   ├── radon_parallel_line.cpp // 核心投影、反投影、滤波的实现
    │   └── filter_parallel_line.cpp // 实现平行束滤波反投影
    ├── fan_beam
    │   └── radon_fan_beam.cpp // 投影、反投影的实现
    └── basic_radon_functions.cpp // 对投影函数进行滤波
        // 尝试扇形束滤波反投影（仅实现了投影部分）
        // 实现扇形束的投影
        // 投影的底层基础函数（仅由上方的文件@内部调用@）

└── model/
    └── shepplogan.cpp // 模型
        // 解析法构建shepplogan模型及其投影结果

└── utils/
    ├── basic_functions.cpp // 基本函数（如计算最大值、点到直线距离等）
    └── basic_display_functions.cpp // 显示图像、输出图像所需函数

└── GUI/
    ├── CT_application.h // 显示界面相关函数
    ├── CT_application.cpp // GUI界面的Qt类定义
    └── CT_application.ui // GUI界面设计（按钮位置、组件位置等）

└── CT_main.cpp // 主函数（测试代码，实际exe运行不依赖于该main函数）
```

2、实际代码框架

不包含GUI的实际代码框架如下：

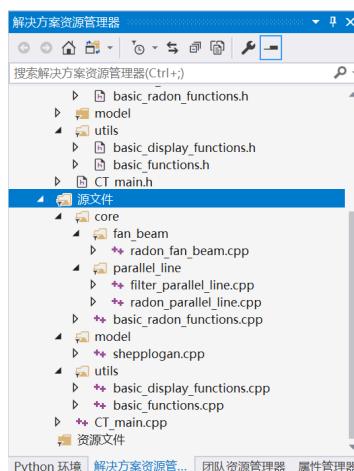


图3.1 不含GUI的实际代码框架

添加GUI后的实际代码架构框架如下：

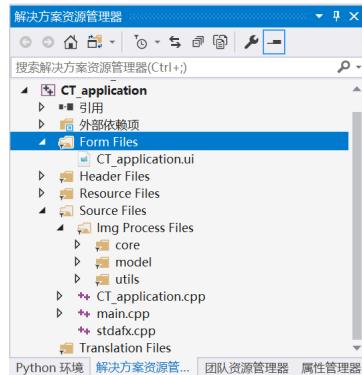


图3.2 添加GUI后的实际代码框架

重要声明：

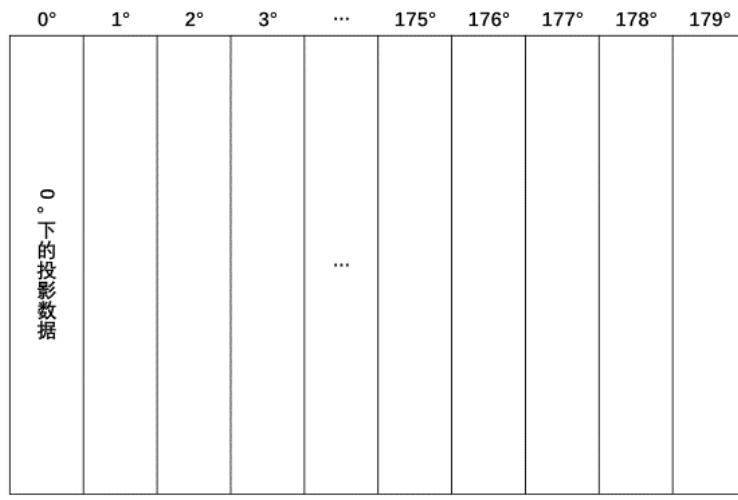
本算法框架中涉及到的所有文件、代码和函数均从零自定义实现。

OpenCV中仅使用了Mat数据类型的元素操作的功能`Img.at(i, j)`，基本的图像输入输出流、图像拷贝等功能，以及傅里叶变化相关的`dft`和`idft`的计算函数。

第四章 平行束滤波反投影的实现

1、投影

- 投影是实现滤波反投影重建的第一步，也是极为关键的一步。
- 投影的思路和部分代码，对于反投影部分也存在启发作用，甚至投影的部分代码可以复用到反投影部分。
- 本代码中的投影图像排列如下（注意！本代码中定义投影角度为：射线与x轴正方向的夹角。即 0° 投影为水平向右的投影）注：后来查阅资料，发现大多数代码定义 0° 为竖直向上投影，与本代码中的定义不同，不过计算原理上不会有本质上的变化



投影图像中数据的排列规则

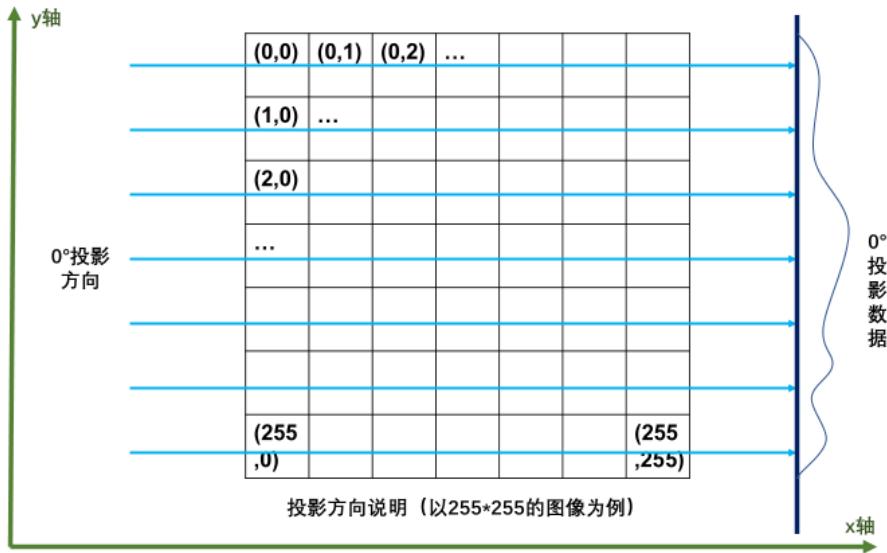


图4.1.1 投影数据说明

1.1 数字积分法投影的算法实现

数字积分法投影的思路主要参考文献：Siddon, Robert L. "Fast calculation of the exact radiological path for a three-dimensional CT array." Medical physics 12.2 (1985): 252-255.

1.1.1 数字积分法投影的思路

数字积分法的总思路为：

- 把二维图像看作水平线和竖直线组成的网格（而非从像素格点的角度来看待二维图像）。
- 在确定好发射器和接收器后，发射器与接收器的连线和网格会形成一系列交点。
- 定义 $\alpha = \frac{\text{发射器到交点的距离}}{\text{发射器到接收器的距离}}$ 。其中若交点是与垂直于x轴的直线形成的交点，把对应的 α 值放入集合 $\{\alpha_x\}$ 中；若交点是与垂直于y轴的直线形成的交点，把对应的 α 值放入集合 $\{\alpha_y\}$ 中。
- 根据网格和发射器、接收器的相对位置关系，求解集合 $\{\alpha_x\}$ 和集合 $\{\alpha_y\}$ 内的所有元素。
- 将集合 $\{\alpha_x\}$ 和集合 $\{\alpha_y\}$ 融合成一个集合 $\{\alpha\}$ ，并对集合内的元素进行升序排列。
 - 注意！每一个 α 值（不管是 α_x ，还是 α_y ）都对应了射线与某一根直线的交点（其中的 x, y 只是区分了与哪个方向的直线，但是在计算长度上——比例*总长度，是完全一致的）。而两个相邻的 α 值对应了射线穿过的某个像素格的两个边，所以可以对应计算到穿过这个像素的路径长度。
- 根据相邻的 α 值，计算穿过某个像素格的路径长度；根据相邻的 α 值的平均值，确定像素格的位置，由此求得像素格对应的灰度值。
- 射线路径 = $\sum(\text{各体素中的路径长度} * \text{密度值})$ ，即求得了所需的投影值。

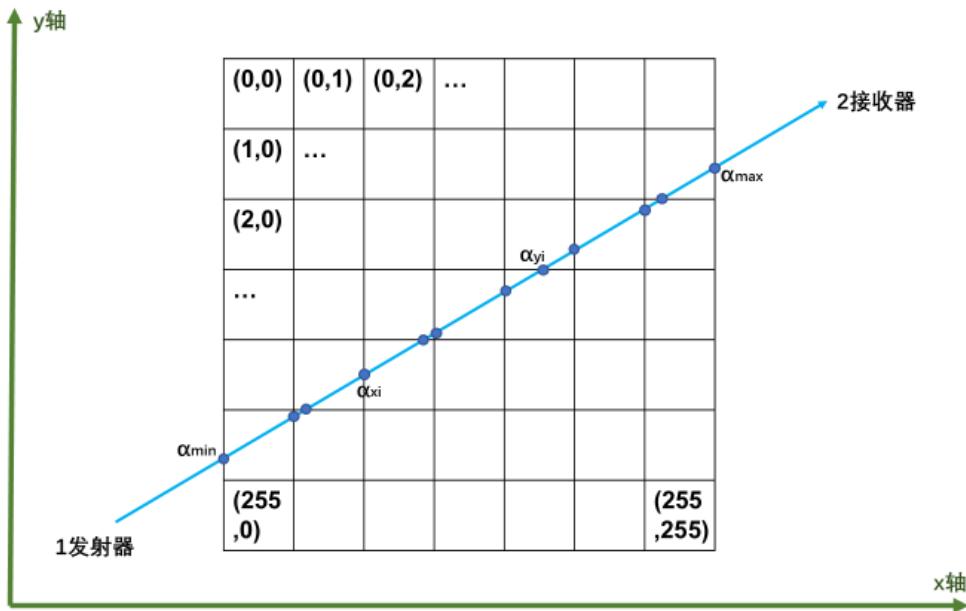


图4.1.2 数字积分投影算法 变量说明

1.1.2 数字积分法的整体算法框架

根据上述思路，本文从零开始，自定义了一个函数（radon_parallel_line.cpp中），提供了数字积分投影的接口，函数原型如下：

```
cv::Mat Radon_parallel_line(cv::Mat image, int num_degrees=180);
```

用latex写了函数的伪代码如下：

Algorithm 1: *Radon_parallel_line(image, num_degrees)*

Input: *image*: 输入的图像; *num_degrees*: 总投影角度

```

1 for 遍历所有的投影角度 do
2   设置发射器和探测器的位置;
3   初始化投影 projection1d;
4   for 遍历每一对发射器和探测器 do
5     计算“相交比例” $\alpha$  的范围;
6     计算索引  $i, j$  的范围;
7     if 投影角度不是特殊角度 (水平或垂直) then
8       计算所有与直线  $x = a$  相交的  $\{\alpha_x\}$  的值;
9       计算所有与直线  $y = a$  相交的  $\{\alpha_y\}$  的值;
10      把  $\{\alpha_x\}$  和  $\{\alpha_y\}$  融合为  $\{\alpha\}$ , 并升序排列;
11    else

```

```

12   |   if 射线平行于 x 轴 then
13   |       只需计算与直线  $x = a$  相交的  $\{\alpha_x\}$  的值，作为  $\{\alpha\}$ ;
14   |   else
15   |       只需计算与直线  $y = a$  相交的  $\{\alpha_y\}$  的值，作为  $\{\alpha\}$ ;
16   |
17   |   根据已经升序排列的  $\{\alpha\}$ ，依次计算各个相邻 alpha 之间的
     |   距离，及其对应的像素点的灰度值，由此得到当前接收器
     |   接收到的投影灰度值 =  $\sum$ (各体素中的路径长度 * 灰度值),
     |   放入 projection1d 的对应位置;
18   |
19   |   将当前角度的 projection1d (一维数组)，放入二维投影图像的
     |   对应列;
20
Output: radon_Img: 输出的投影图像，行数等于输入图像最长
          边的根号 2 倍，列数等于总投影角度

```

图4.1.2 数字积分投影整体思路 伪代码

1.1.3 发射器和接收器位置的确定

本算法自定义了函数设定平行束投影中发射器和接收器位置 (`radon_parallel_line.cpp` 中)，函数的原型如下：

```

static void Set_Transmitters_and_Sensors_parallel_line(const int theta_degree,
const int sensor_length,
Point2d* Trm, Point2d* Rcv)

```

发射器和接收器位置确定的原理如下（可参考下图）：

- 为了使发射器的长度能够覆盖整张图片，考虑 45° （和 135° ）投影时，投影数据的长度达到最大值。为了兼容长方形图像的情况，设置：发射器的长度 = 接收器的长度 = $sensor_length = \text{ceil}(\sqrt{2} * \max(length, width)) + 2$ ，即约等于图像最长边的根号2倍。而计算结果加2的原因是为了防止离散化取整带来的一些问题，保证在任何角度下，探测器和接收器之间的射线都能够覆盖整张图片。
- 设 $Transmitter(Trm)$ 为发射源， $receiver(Rcv)$ 为接收器， Trm, Rcv 均为大小为 $sensor_length$ 的数组。定义角度 θ 为射线与 x 轴正方向的夹角，例如 0° 表示从左向右的射线， 90° 表示从下向上的射线。
- 以图像中心为原点建立直角坐标系，则发射源和接收器的中点永远在圆 $x^2 + y^2 = \frac{1}{4} sensor_length^2$ 上，即图中的紫色圆上。发射器和接收器始终与紫色圆相切。
- 先确定发射源和接收器的中点位置，则可通过探测器（或接收器）的长度和角度关系确定 $Trm[0]$ 和 $Rcv[0]$ 的坐标位置，再根据角度关系，遍历求得每一个 $Trm[i]$ 和 $Rcv[i]$ 的坐标位置。

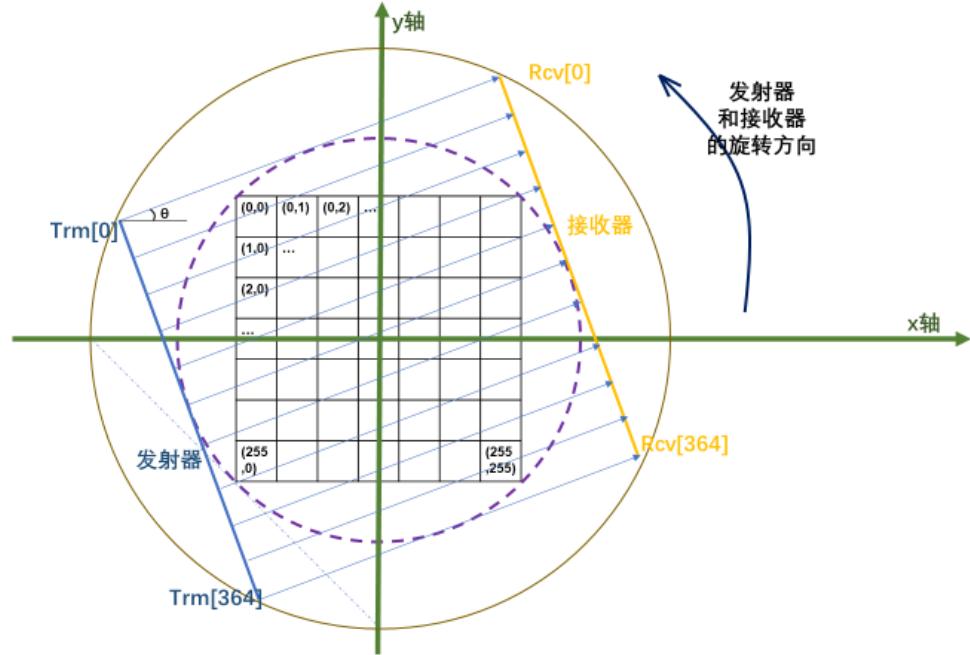


图4.1.3 数字积分投影 发射器和接收器的位置确定

部分核心代码（以发射器位置的确定为例，接收器同理）如下：

```

Point2d* Trm;
Trm = new Point2d[sensor_length];

//发射器中点
Point2d mid_Trm;
mid_Trm.x = 0.5 * (double(sensor_length) - 2) * cos(theta + pi);
mid_Trm.y = 0.5 * (double(sensor_length) - 2) * sin(theta + pi);

//发射器Trm[0]的位置确定
Trm[0].x = mid_Trm.x + 0.5 * sensor_length * cos(0.5 * pi + theta);
Trm[0].y = mid_Trm.y + 0.5 * sensor_length * sin(0.5 * pi + theta);

//所有发射器位置的确定，Trm[0]~Trm[sensor_length-1]共sensor_length个
for (int i = 1; i < sensor_length; ++i)
{
    Trm[i].x = Trm[i - 1].x - 1 * cos(0.5 * pi + theta);
    Trm[i].y = Trm[i - 1].y - 1 * sin(0.5 * pi + theta);
}

```

1.1.4 “相交比例” α 的范围和索引*i, j*范围的确定

确定“相交比例” α 的范围和索引*i, j*范围的自定义函数原型如下：

（在basic_radon_functions.cpp中定义，由于basic_radon_functions.cpp中均为自定义的投影相关的底层函数，只供radon_parallel_line.cpp, radon_fan_beam.cpp等文件使用，而在主函数中不被直接include，因此没有声明静态函数）。

```

//计算alpha的范围
void Calculate_range_of_alpha(const int i, const Point2d* Trm, const Point2d* Rcv, const Mat image,
    bool& flag_intersection, double& min_alpha, double& max_alpha);

//计算索引i, j的范围
void Calculate_range_of_index_to_image(const int i, const Point2d* Trm,
    const Point2d* Rcv, const Mat image, const double min_alpha, const double max_alpha,
    int& intersection_n_min, int& intersection_n_max, int& intersection_m_min,
    int& intersection_m_max);

```

确定“相交比例” α 的范围的原理如下：

- 把x-y直角坐标系的原点放在图像中心（参考图4.1.3），则图像的范围为：
 $x \in (-0.5 * image.cols, 0.5 * image.cols), y \in (-0.5 * image.rows, 0.5 * image.rows)$
- $\alpha_{min} = \max\{0, \min[\alpha_x(x_{min}), \alpha_x(x_{max})], \min[\alpha_y(y_{min}), \alpha_y(y_{max})]\}$
- $\alpha_{max} = \min\{0, \max[\alpha_x(x_{min}), \alpha_x(x_{max})], \max[\alpha_y(y_{min}), \alpha_y(y_{max})]\}$

确定“相交比例” α 的核心代码如下：

```

//min_alpha_x, max_alpha_x的计算 (y方向计算类似, 此处未展示)
if ((Rcv[i].x - Trm[i].x) != 0)
{
    min_alpha_x = min((img_x_min - Trm[i].x) / (Rcv[i].x - Trm[i].x),
        (img_x_max - Trm[i].x) / (Rcv[i].x - Trm[i].x));
    max_alpha_x = max((img_x_min - Trm[i].x) / (Rcv[i].x - Trm[i].x),
        (img_x_max - Trm[i].x) / (Rcv[i].x - Trm[i].x));
}

min_alpha = max_3(min_alpha_x, min_alpha_y, 0.0);
max_alpha = min_3(max_alpha_x, max_alpha_y, 1.0);

```

确定索引*i, j*范围的原理（以*i*为例）如下：

- $i_{min} = N_x - \frac{X_{plane}(N_x) - \alpha_{min}(X_2 - X_1) - X_1}{d_x}, i_{max} = 1 + \frac{X_1 + \alpha_{max}(X_2 - X_1) - X_{plane}(1)}{d_x}, \text{for } (X_2 - X_1) \geq 0$
- $i_{min} = N_x - \frac{X_{plane}(N_x) - \alpha_{max}(X_2 - X_1) - X_1}{d_x}, i_{max} = 1 + \frac{X_1 + \alpha_{min}(X_2 - X_1) - X_{plane}(1)}{d_x}, \text{for } (X_2 - X_1) \leq 0$

确定索引*i, j*范围的代码（代码中命名为m, n）如下：

（由于方向的关系，这里的公式略作了调整，与文献中的方向存在差别。一开始这个位置造成了错误，耗费了一些时间。）

```

if (Trm[i].x < Rcv[i].x)
{
    intersection_n_min = (image.cols) - (img_x_max - min_alpha * (Rcv[i].x - Trm[i].x) - Trm[i].x);
    intersection_n_max = -1 + (Trm[i].x + max_alpha * (Rcv[i].x - Trm[i].x) - img_x_min);
}

```

```

else
{
    intersection_n_min = (image.cols) - (img_x_max - max_alpha * (Rcv[i].x - Trm[i].x) - Trm[i].x);
    intersection_n_max = -1 + (Trm[i].x + min_alpha * (Rcv[i].x - Trm[i].x) - img_x_min);
}
//m范围(m为图像的行，在坐标系中对应y)
//由于随x增大，n增大；而随y增大，m减小，所以公式需要做调整(与文献中的方向有差别)。
if (Trm[i].y < Rcv[i].y)
{
    intersection_m_min = (image.rows) - (Trm[i].y + max_alpha * (Rcv[i].y - Trm[i].y) - img_y_min);
    intersection_m_max = img_y_max - min_alpha * (Rcv[i].y - Trm[i].y) - Trm[i].y - 1;
}
else
{
    intersection_m_min = (image.rows) - (Trm[i].y + min_alpha * (Rcv[i].y - Trm[i].y) - img_y_min);
    intersection_m_max = img_y_max - max_alpha * (Rcv[i].y - Trm[i].y) - Trm[i].y - 1;
}

```

1.1.5 集合 $\{\alpha\}$ 中各元素的确定

确定集合 $\{\alpha\}$ 中各元素的函数原型如下：

```

//计算alpha_x集合内所有的元素
void calculate_alpha_x(const int i, const Point2d* Trm, const Point2d* Rcv,
                      const Mat image, const int intersection_n_min, const int sizeof_alpha_x,
                      double* alpha_x);

//计算alpha_y集合内所有的元素
void calculate_alpha_y(const int i, const Point2d* Trm, const Point2d* Rcv,
                      const Mat image, const int intersection_m_min, const int sizeof_alpha_y,
                      double* alpha_y);

//将alpha_x和alpha_y融合为集合alpha
void calculate_alpha(const int sizeof_alpha_x, const int sizeof_alpha,
                     const double* alpha_x, const double* alpha_y, double* alpha);

```

计算原理：

- 由于已知了索引的范围，因此对符合索引范围内的整数进行遍历，就可以求得对应方向内所有的 α 值。
- 把两个方向求得的所有 α 值都放入一个数组排序，就可以得到最终的集合 $\{\alpha\}$ 。

(由于这部分比较简单，对这部分代码感兴趣的可以到源代码中查看)

1.1.6 特定角度特定接收器得到的投影值计算

确定特定角度特定接收器得到的投影值的自定义函数原型如下：

```
double calculate_projection_onevalue(const int i, const Point2d* Trm,
    const Point2d* Rcv, const Mat image, const int sizeof_alpha, const double*
alpha);
```

计算原理：

- 取集合 $\{\alpha\}$ 内相邻元素相减，乘上探测器和接收器之间的总长度，即可计算这一小段路径的长度；
- 对应的像素标号 $[i(m), j(m)]$ 根据中间位置的 $\alpha_{mid} = \frac{1}{2}[\alpha(m) + \alpha(m - 1)]$ 计算，就可得到对应的灰度值范围；
- 投影值 = \sum (各像素中的路径长度 * 灰度值)。

计算投影值的核心代码如下：

```
double total_TR_length = distance_2d(Trm[i], Rcv[i]);
double projection = 0.0;

//遍历集合alpha内的所有元素
for (int k = 1; k < sizeof_alpha; ++k)
{
    //计算某个像素中的路径长度
    double length_of_little_route = (alpha[k] - alpha[k - 1]) * total_TR_length;
    double mid_alpha = 0.5 * (alpha[k] + alpha[k - 1]);

    //计算像素点的中点坐标
    Point2d M_pixel;
    M_pixel.x = Trm[i].x + mid_alpha * (Rcv[i].x - Trm[i].x);
    M_pixel.y = Trm[i].y + mid_alpha * (Rcv[i].y - Trm[i].y);

    //计算对应像素的索引，获取对应像素的灰度值
    int m_pixel, n_pixel;//Caution!!n为列，对应x；m为行，对应y！！！（就是这里弄反了，重建图像旋转了90度还有镜像）
    n_pixel = floor(M_pixel.x + 0.5 * image.cols);
    n_pixel = max(0, n_pixel);
    n_pixel = min(n_pixel, image.rows - 1);

    m_pixel = floor(0.5 * image.rows - M_pixel.y);
    m_pixel = max(0, m_pixel);
    m_pixel = min(m_pixel, image.cols - 1);

    //计算这一小段路径贡献的投影值，并叠加到投影结果上
    double grayscale = image.at<uchar>(m_pixel, n_pixel);
    projection += length_of_little_route * grayscale;
}
```

注意：

- 这里的 m , n 与 x , y 的对应关系也容易弄反，如果把 m , n 弄反，最终的重建图像会旋转90度并发生镜像
- 此处仍需检查索引越界的问题。

1.1.7 易错点

1.1.7.1 索引和坐标系轴的方向

由于图像的索引(i,j)中，i表示行，j表示列；而x-y直角坐标系中，x表示横向位置（对应第二个索引，列j），y表示纵向位置（对应第一个索引，行i）。而且假如遍历图像从左向右的像素，j和x都为递增趋势，方向一致；而假如遍历图像从下往上的像素，i为递减趋势，y为递增趋势，方向相反。

因此，最初编写过程中，由于没有充分认识到这一点，常常把方向弄错，给代码实现带来了一些困难（确定索引范围Calculate_range_of_index_to_image时绕了一些弯路）。而项目完成后反思这块内容，猜测这些索引的不一致性，可能也是大部分已有算法选择以竖直向上为 0° 投影方向的原因（可能建模过程中，选择了从下往上为x轴正方向，则投影角度仍表示射线与x轴正方向的夹角，但是建系的方向与本算法中的方向相比，逆时针旋转了 90° ）。

1.1.7.2 垂直或平行于坐标轴的射线的特殊情况处理

一开始，我对于垂直或平行于坐标轴的判据使用了如下判据：

```
if ((Rcv[i].x - Trm[i].x) != 0)      //不平行于y轴的判据  
if ((Rcv[i].y - Trm[i].y) != 0)      //不平行于x轴的判据
```

但是在实际代码运行过程中，由于发射器和接收器的坐标位置都是小数，存在计算 \cos 或者 \sin 之后的近似问题，导致平行于y轴或者x轴时，**两者的横坐标（或者纵坐标）非常相近，无法完全相等**，因此无法对这些情况进行处理。

因此，最终使用的判据还是需要在角度方面做一些判断，使用以下判据：

```
if ((theta_degree%90) != 0)      //不平行于任何坐标轴的判据  
if ((theta_degree % 180) == 0)    //平行于x轴的判据  
//else, 即为平行于y轴的情况
```

1.1.7.3 与图像完全没有相交的射线的特殊情况处理

在计算alpha的时候，有一个无交点的情况容易忽略。这里面还涉及到函数之间参数传递的一些问题，有兴趣的话可以查看Calculate_range_of_alpha函数以及Radon_parallel_line函数对应部分的源代码，这里不再赘述。

```
if (min_alpha >= max_alpha)//没有交点，或者仅有一个点的交点（而不是线段），投影为0  
{  
    flag_intersection = 0;  
}
```

1.2 解析法投影的算法实现

解析法投影的公式（椭圆投影公式）以及Shepp-Logan模型的参数主要参考：Avinash C. Kak, and Malcolm Slaney, "Principles of Computerized Tomographic Imaging", IEEE Press, 1999. <https://engineering.purdue.edu/~malcolm/pct/>, Chapter 3. Algorithms for Reconstruction with Nondiffracting Sources.

相关代码全部在“shepplogan.cpp”文件中，自定义函数原型如下：

```
cv::Mat Create_shepplogan(int rows = 256);
cv::Mat Calc_shepplogan_radon(int rows = 256);
```

1.2.1 Shepp-Logan模型的生成展示

其中Shepp-Logan模型以各个椭圆叠加的方式生成，椭圆叠加利用自定义函数：

```
static Mat Add_ellipse(Mat inputImg, Point2d center_point, double major_axis,
double minor_axis,
int rotation_degree, double refractive_index)
```

将椭圆的中心坐标center_point，长轴长度major_axis，短轴长度minor_axis，逆时针旋转角度rotation_degree以及叠加的灰度值refractive_index输入自定义函数，就可以把叠加该椭圆的inputImg返回出来。

任意位置的椭圆解析式（长轴a，短轴b，中心坐标（m，n），旋转角度 α ）如下：

$$\frac{((x-m)\cos\alpha+(y-n)\sin\alpha)^2}{a^2} + \frac{((m-x)\sin\alpha+(y-n)\cos\alpha)^2}{b^2} = 1$$

根据上述公式，很容易写出Add_ellipse函数的代码。

1.2.2 Shepp-Logan模型的参数

使用的参数参考了上述书籍，为了增强图像的对比度，对部分灰度值进行了修改。

表4.1.1 本代码使用的Shepp-Logan模型的参数

中心坐标	长轴a	短轴b	旋转角度 α （度）	灰度值	备注
(0, 0)	0.92	0.69	90	2.0	主白圈
(0, -0.0184)	0.874	0.6624	90	-1.48	主灰圈
(0.22, 0)	0.31	0.11	72	-0.2	右大圈
(-0.22, 0)	0.41	0.16	108	-0.2	左大圈
(0, 0.35)	0.25	0.21	90	0.25	中上大白圈
(0, 0.1)	0.046	0.046	0	0.25	中间上面的小圈
(0, -0.1)	0.046	0.046	0	0.25	中间下面的小圈
(-0.08, -0.605)	0.046	0.023	0	0.25	下面左边的小圈
(0, -0.605)	0.023	0.023	0	0.25	下面中间的小圈
(0.06, -0.605)	0.046	0.023	90	0.25	下面右边的小圈

解析法生成的尺寸为256*256的Shepp-Logan模型如下图所示。



图4.1.4 解析法生成的Shepp-Logan模型

1.2.3 Shepp-Logan解析法计算投影

根据上述参考书中给出的公式，可以容易地算出椭圆各角度的解析投影值。公式如下：

对于标准椭圆 $\frac{x^2}{A^2} + \frac{y^2}{B^2} = 1$, 椭圆内的灰度值为 ρ , 则

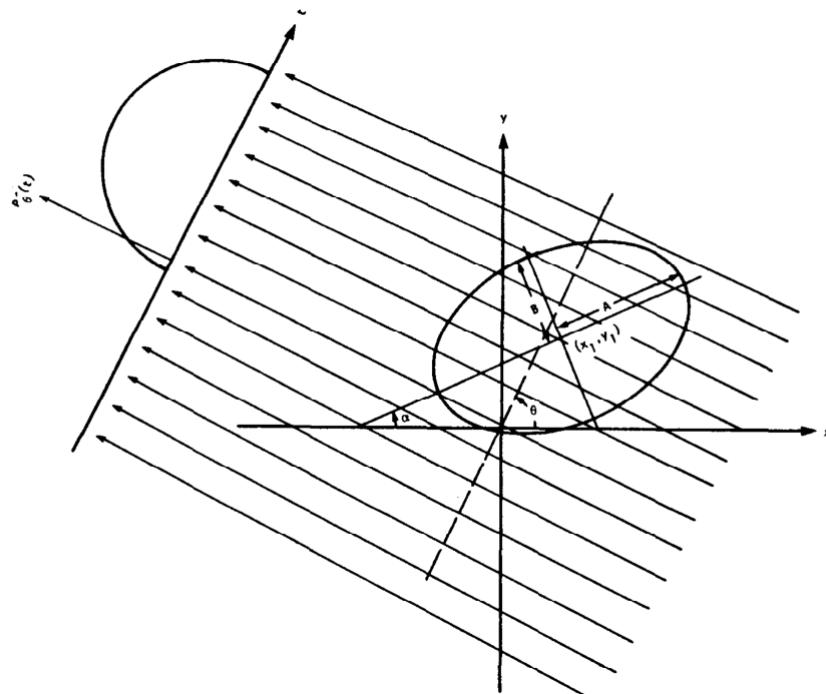


图4.1.5 解析法计算椭圆投影 图示

图片来源：“Principles of Computerized Tomographic Imaging”第三章

It is easy to show that the projections of such a function are given by

$$P_\theta(t) = \begin{cases} \frac{2\rho AB}{a^2(\theta)} \sqrt{a^2(\theta) - t^2} & \text{for } |t| \leq a(\theta) \\ 0 & |t| > a(\theta) \end{cases} \quad (5)$$

where $a^2(\theta) = A^2 \cos^2 \theta + B^2 \sin^2 \theta$. Note that $a(\theta)$ is equal to the projection half-width as shown in Fig. 3.5(a).

Now consider the ellipse described above centered at (x_1, y_1) and rotated by an angle α as shown in Fig. 3.5(b). Let $P'(\theta, t)$ be the resulting projections. They are related to $P_\theta(t)$ in (5) by

$$P_\theta(t) = P_{\theta-\alpha}(t - s \cos(\gamma - \theta)) \quad (6)$$

where $s = \sqrt{x_1^2 + y_1^2}$ and $\gamma = \tan^{-1}(y_1/x_1)$.

图4.1.6 解析法计算椭圆投影 计算公式

图片来源：“Principles of Computerized Tomographic Imaging”第三章

因此，计算某个角度某个椭圆的投影的自定义函数核心代码如下：

(需要注意的是，书中公式里的m, n只能影响gamma的值，而对于一三和二四象限内部无法区分，因此，对于各象限的数据在实际编程时需要分类讨论！)

```
double m = center_point.x * (rows / 2); //平移参数m
double n = center_point.y * (rows / 2); //平移参数n
double a = major_axis * (rows / 2); //椭圆参数a
double b = minor_axis * (rows / 2); //椭圆参数b
double alpha = rotation_degree * pi / 180; //椭圆旋转角度alpha
double theta = (theta_degree - double(90)) * pi / 180; //投影角度theta (公式中以向上为正方向，程序中所有建模以射线向右为正方向，相差了90°)
double a_theta_2 = pow(a, 2) * pow(cos(theta - alpha), 2) + pow(b, 2) *
pow(sin(theta - alpha), 2);
double s = sqrt(m * m + n * n);
double gamma = 0;

if (n == 0)
{
    if (m >= 0)      gamma = 0;
    else            gamma = pi;
}
else
{   //各象限需要区分！！，否则，m, n只能影响gamma，一三和二四象限内部无法区分，书中没有体现这个细节...
    if ((m < 0))  gamma = atan(float(n / m)) + pi; //第二、三象限
    else            gamma = atan(float(n / m)); //第一、四象限
}

for (int i = 0; i < radon_rows; ++i)
{
    double t0 = i - 0.5 * radon_rows;
    double t = t0 - s * cos(gamma - theta);
    if (t * t <= a_theta_2)
    {
```

```
    radonImg.at<double>(i, theta_degree) += ((2 * double(255) *  
refractive_index * a * b) / a_theta_2)  
        * sqrt(a_theta_2 - pow(t, 2));  
}  
}
```

解析法生成的投影数据如下图所示：

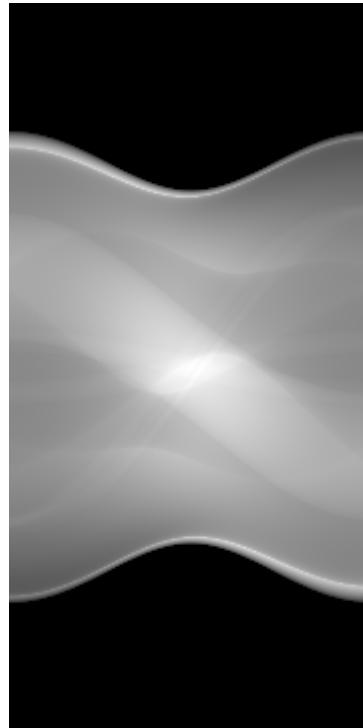


图4.1.7 解析法直接生成的Shepp-Logan模型的投影数据（以256*256为例）

2、反投影

2.1 像素驱动的反投影方法

使用像素驱动法的自定义的反投影算法的函数原型如下：

```
Mat iRadon_parallel_line(Mat radon_Img)
```

像素驱动的反投影算法的伪代码如下：

Algorithm 2: iRadon_parallel_line(radon_Img)

Input: radon_Img: 投影图像 (行数为探测器个数, 列数为投影角度个数)

- 1 记录总投影角度 = 投影图像的列数;
- 2 **for** 遍历所有的投影角度 **do**
- 3 设置发射器和探测器的位置;
- 4 **for** 遍历所有的像素格点 **do**
- 5 确定像素中心在直角坐标系中的位置;
- 6 求像素的中心【点】到 Trm[0] 和 Rcv[0] 的连线【直线】之间的距离;
- 7 根据点到直线距离, 确定对应的投影值 (一次插值), 并反投影到该像素位置;
- 8 对所有像素值除以反投影的次数;

Output: output_Img: 输出的反投影图像

图4.2.1 反投影算法 伪代码

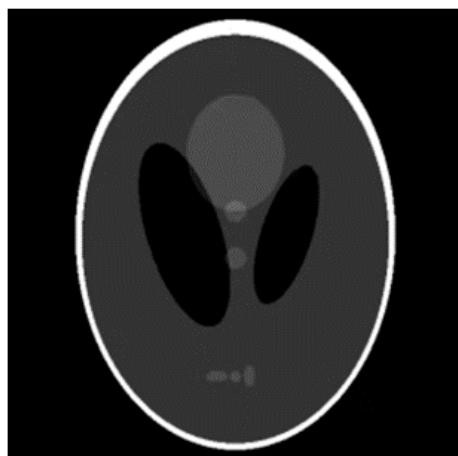
其中, 为计算点到直线的距离, 自定义了函数, 原型如下 (声明了静态函数, 仅供 radon_parallel_line.cpp 的内部函数使用) :

```
static double cal_distance_point_to_line(Point2d Point, Point2d A1, Point2d A2)
```

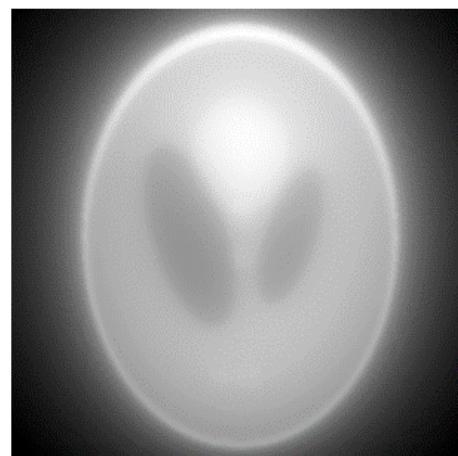
为了计算一次线性插值的投影值, 自定义了函数, 原型如下 (也为静态函数) :

```
static double Get_grayscale_iradon(double distance, const Mat radon_Img,
                                     const int sensor_length, const int theta_degree)
```

对上一步完成的数字积分的投影图像进行直接反投影, 得到的测试结果如下:



Shepp-Logan模型原图



直接反投影结果

图4.2.2 直接反投影结果

2.2 软件测试效率的提高

为了每次测试程序的时候不需要都运行投影的步骤，测试程序的时候把投影数据用txt格式数据导出。这样在测试反投影的部分代码时，可以专注于反投影的代码运行过程，提高代码的测试效率（主要是运行时间的缩短）。

名称	修改日期	类型
circle_radon.txt	2022/4/19 15:09	文本文档
rec_radon.txt	2022/4/19 15:09	文本文档
shepplogan_radon.txt	2022/4/19 15:06	文本文档

图4.2.3 输出的txt文件

写入和读出txt使用的函数原型为（工具性函数，参考了网络代码）：

```
int WriteData(string fileName, cv::Mat& matData);
int LoadData(string fileName, cv::Mat& matData, int matRows, int matCols, int matChns)
```

利用这两个函数，在测试反投影代码时，可以直接读入数据，如：

```
Mat radon_Img;
if (LoadData("./txt/" + image_name + "_radon.txt", radon_Img, 365, 180, 1) == 0)
{
    cout << "load txt to mat SUCCESS!" << endl;
}
radon_Img.convertTo(radon_Img, COLOR_BGR2GRAY);
```

3、滤波

自定义的滤波相关的函数全部在"filter_parallel_line.cpp"中

自定义的平行束投影滤波函数的原型如下：

```

cv::Mat Filter_radon_parallel_line(cv::Mat inputImg, int filter_flag =
0); //filter_flag可选

enum filterFlags {
    /** choose the filter for the radon image */
    non_filter = 0,
    RL_filter = 1, //添加滤波|w| (矩形窗)
    SL_filter = 2 //添加滤波|w|*sinc(w/2B)
};

```

平行束滤波的原理：

$$g(t, \theta) = \int_{-\infty}^{+\infty} |w| P(\omega, \theta) e^{j2\pi\omega t} d\omega = F^{-1}(H(\omega) \cdot P(\omega, \theta)), \text{ 其中 } H(\omega) = |\omega|$$

$$f(x, y) = \int_0^{\pi} g(x \cos \theta + y \sin \theta) d\theta.$$

其中 $P(\omega, \theta)$ 为投影函数的傅里叶变换， $f(x, y)$ 为重建后的空间域图像。

又由于 $H(\omega) = |\omega|$ 为无法实现的理想滤波器，故衍生出了添加矩形窗的R-L滤波器，以及添加矩形窗和 sinc 函数的S-L滤波器等等。

本算法中提供了R-L滤波器和S-L滤波器的接口，但是软件中仅提供了S-L滤波器选项。

滤波的核心代码如下（其中dft和idft使用cv库的函数，并且参考了cv的dft函数的官方测试代码；滤波操作自定义完成）：

(此处以S-L滤波器为例，若感兴趣可查看源代码中的R-L滤波器)

```

//...省略dft过程
//对某一列做过dft的投影图像进行滤波 (R-L滤波函数 or S-L滤波函数)

//S-L滤波函数: H_{SL} = |w| * sinc(w/2B) (when |w|<B)
if (filter_flag == SL_filter)
{
    int DFT_length = complexI.rows;
    double Band_width = 1 * DFT_length;//设置带宽，减小Gibbs effect
    for (int w = 0; w < floor(0.5 * DFT_length); ++w)
    {
        double filterFactor = double(w);
        double tmp = pi * ((w - floor(0.5 * DFT_length)) / (2 * Band_width));
        if (tmp != 0) filterFactor *= sin(tmp) / tmp; //再乘上sinc函数
        else filterFactor *= 1;
        complexI(cv::Range(w, w + 1), cv::Range::all()) *= filterFactor; //第w个
        complexI(cv::Range(DFT_length - w - 1, DFT_length - w),
        cv::Range::all()) *= filterFactor; //对称性
    }
}
//省略idft过程...

```

4、评价指标

参考教材《医学成像的基本原理》第五章：X射线计算机断层成像（编著：黄力宇），本算法中使用两个评价指标对重建的性能进行评估：

- 归一化均方距离判据 $d = (\sum_{u=1}^N \sum_{v=1}^N (t_{u,v} - r_{u,v})^2 / \sum_{u=1}^N \sum_{v=1}^N (t_{u,v} - \bar{t})^2)^{\frac{1}{2}}$
- 归一化平均绝对距离判据 $r = \sum_{u=1}^N \sum_{v=1}^N |t_{u,v} - r_{u,v}| / \sum_{u=1}^N \sum_{v=1}^N |t_{u,v}|$

其中d能较敏感地反映某几点产生较大误差的情况，r则较敏感地反映许多点均有一些小的误差的情况。以上评价指标的值越大，表示误差越大。

函数实现：

定义了两个函数，提供计算评价指标的接口，自定义的函数原型如下（可在 basic_display_functions.cpp 中找到函数的定义）。

```
double calculate_error_sqrt(cv::Mat image1, cv::Mat image2);
double calculate_error_abs(cv::Mat image1, cv::Mat image2);
```

5、效果展示

5.1 测试图像的重建效果展示

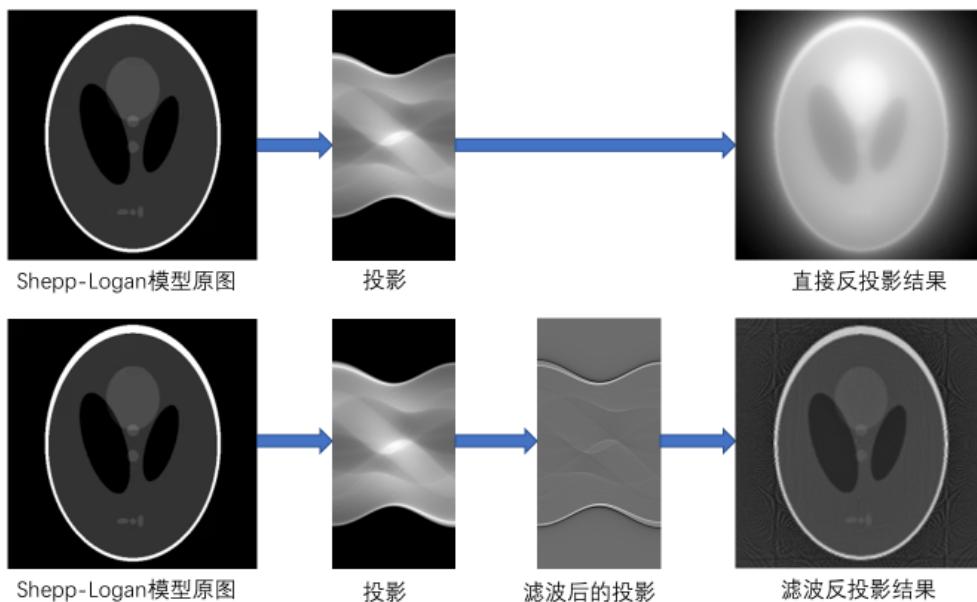


图4.5.1 测试图像的重建效果展示

5.2 解析法和数字积分法的效果对比

用解析法生成了原图和投影进行重建，并且用解析法生成的原图作为本地图片输入、用数字积分法投影、再重建的结果进行比较。



图4.5.2 解析法直接生成的Shepp-Logan模型原图 (以256*256为例)

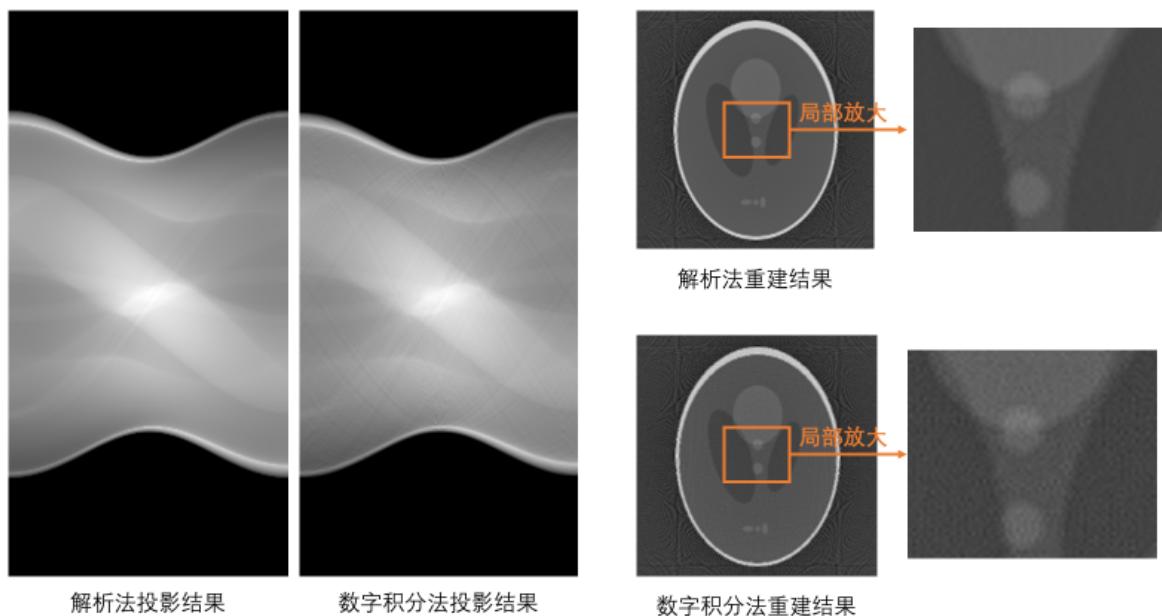


图4.5.3 解析法和数字积分法对比 (以256*256为例)

从上面的对比图中，我们可以清晰地看到：

- 对于相同的原图，解析法生成的投影图像非常的“干净”，而数字积分法由于离散化处理后的一些近似，导致生成的投影图像会出现一些伪影（细丝）。
- 从重建图像的局部放大中也可以看到，解析法重建的图像相对比较清晰，而数字积分法重建出的图像放大后，存在网格化的伪影。

从评价指标上也可以看出差别：

	解析法	数字积分法
归一化均方距离判据d	0.831912	0.86527
归一化平均绝对距离判据r	1.00795	1.01885

从定量的指标中也可以看出，解析法图像的各个误差指标，都小于数字积分法。

5.3 彩色图像的重建展示

本算法对彩色图像（或者说三通道图像）兼容，读入彩色图像，可以输出灰度化后的重建结果，如下图所示。



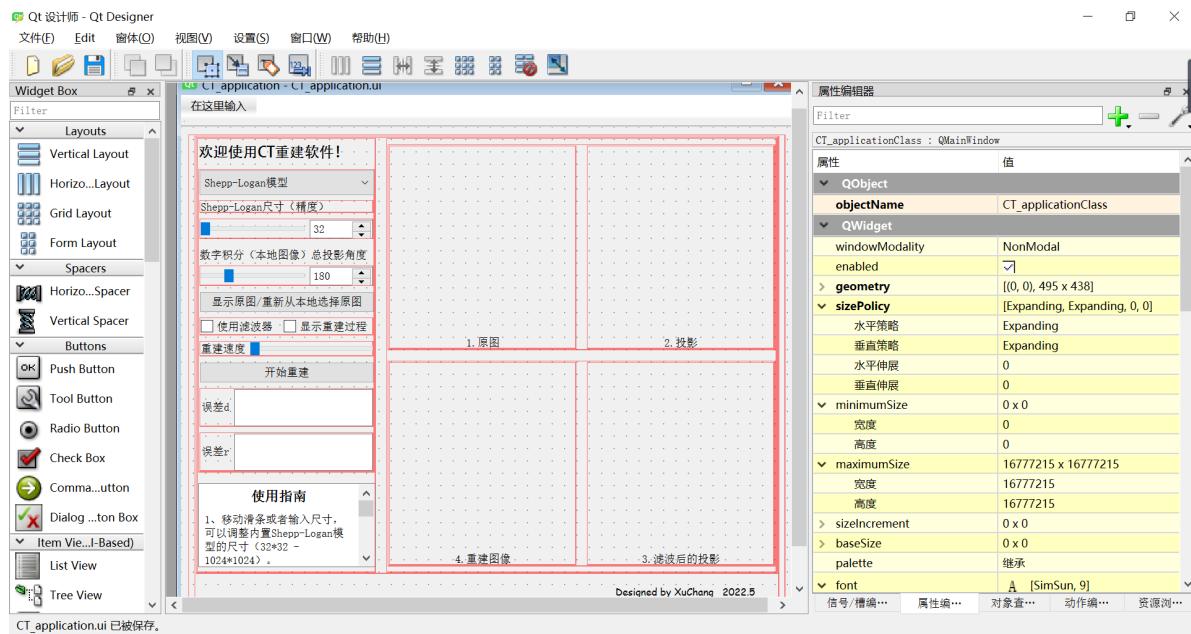
图4.5.4 彩色图像的重建

第五章 界面设计的实现

Qt Designer进行界面设计，Visual Studio进行回调函数的编写。

1、界面设计

界面使用Qt Designer进行设计。



为了使窗体放大缩小时，界面不被打乱，采用了尺寸限制和布局的方式进行排版。

2、回调函数的实现

回调函数通过在Visual Studio的头函数内编写继承的QMainWindow类，在源文件中编写回调函数，并在Qt Designer内命名控件、建立链接关系的方式，实现GUI界面中的各功能。

3、外部代码的导入

将之前编写好的各个头文件和对应的cpp文件都复制到Qt的工程模板根目录下，并且添加到工程中，头文件清单如下：

```
#include "basic_display_functions.h"
#include "basic_functions.h"
#include "shepplogan.h"
#include "radon_parallel_line.h"
#include "filter_parallel_line.h"
```

对命名空间冲突的问题进行处理，例如头文件取消“using namespace cv”的声明，并在头文件所需位置添加cv::，然后在cpp文件中添加“using namespace cv”的声明等。从而让回调函数中能够调用所有之前自定义的外部接口（函数）。

4、软件独立exe的打包

- 使用VS的Release模式，生成exe；
- 使用windeployqt生成动态链接文件（需要手动添加opencv_world440.dll）；
- 使用Enigma virtual box工具，把exe和dll文件全部打包成独立exe。

5、不同电脑上运行结果的展示

在不同电脑上运行软件，界面略有不同，但总体兼容。运行结果展示如下。

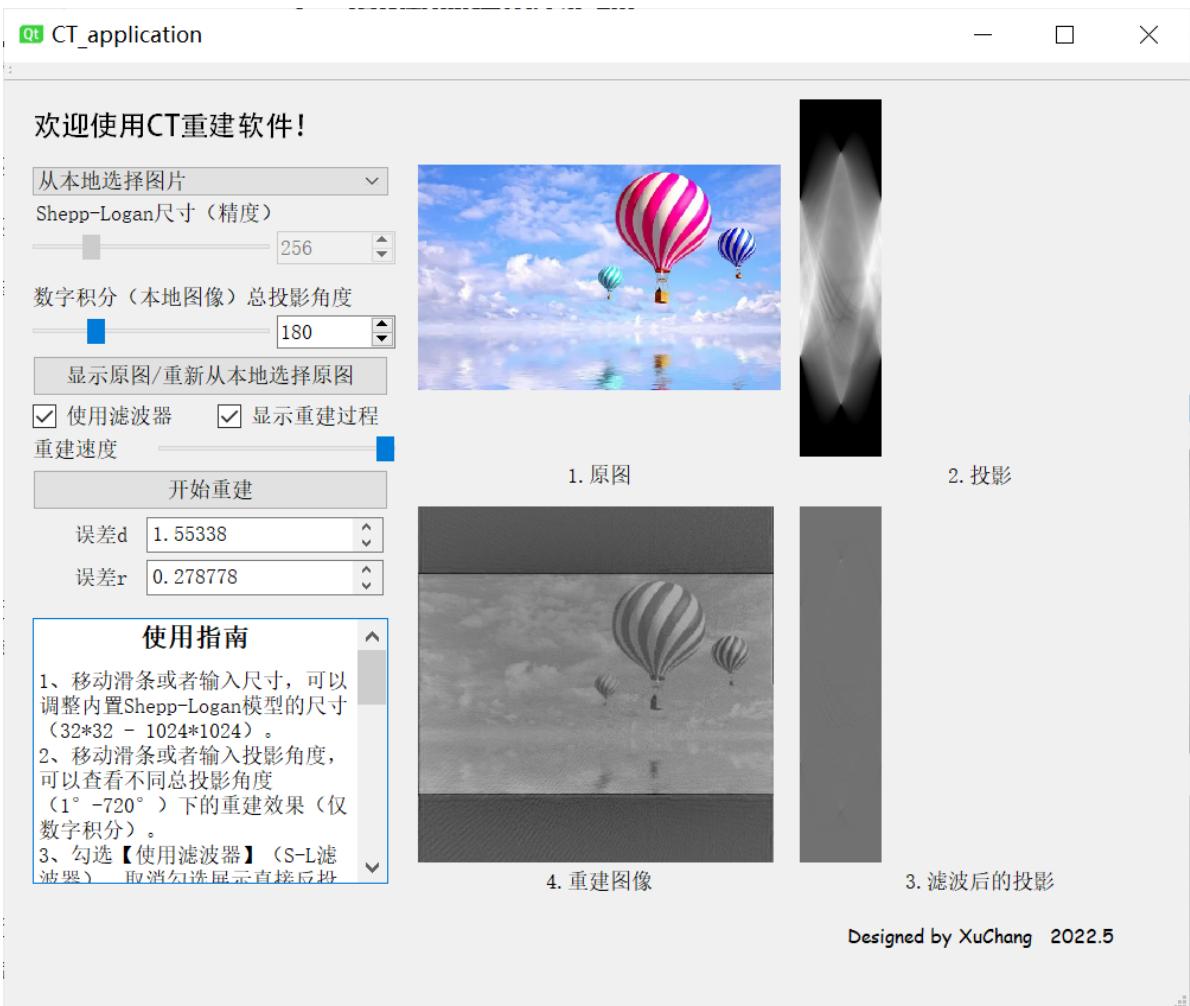


图5.5.1 本人电脑上的软件界面



图5.5.2 另一台电脑上的软件界面

第六章 扇形束滤波反投影的探究

1、扇形束投影的算法实现

扇形束投影的原理如下：

- 发射器和接收器始终在圆 $x^2 + y^2 = \frac{1}{2}a^2$, 其中 a 为图像的边长。
- 设发射器与接收器连线的中心的连线与 x 轴正方向的夹角为 θ , 扇形所对圆心角为 α 。

根据上述两条规则, 即可确定发射器和接收器的位置。

为了和与平行束投影的计算完全兼容, 尽管发射器在某一个角度下只有一个位置, 但是和平行束时的存储统一, 利用长度为接收器个数的数组, 存储相同的坐标位置。

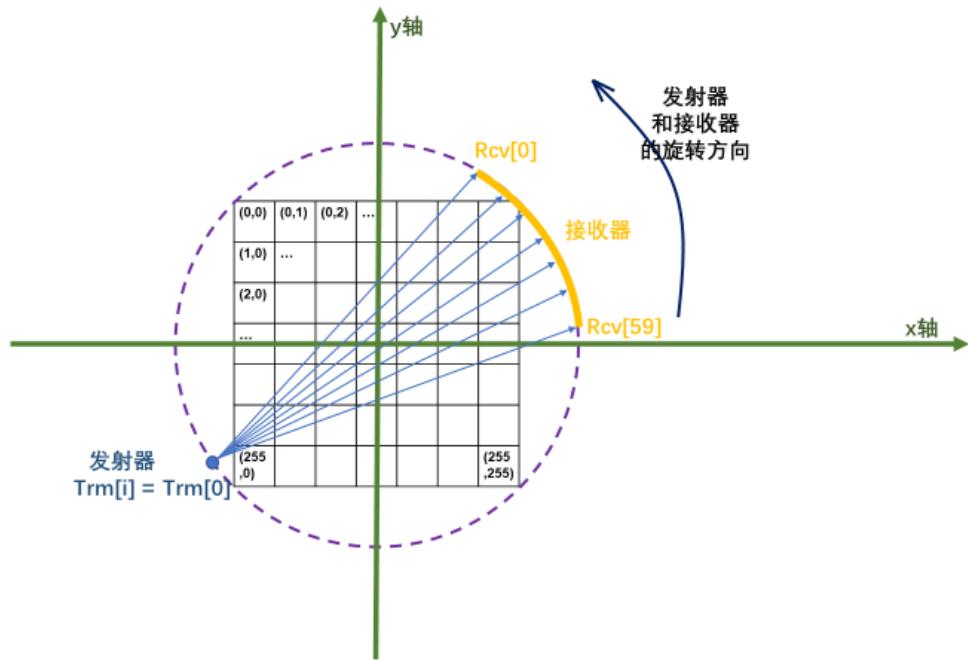


图6.1.1 扇形束重建 发射器和接收器建模

代码直接可见radon_fan_beam.cpp中，原型为：

```
Mat Radon_fan_beam(Mat image, int num_rotate_degree = 360, int
num_detector_degree = 60);
```

并且仅添加了自定义函数对扇形束的探测器和接收器的位置进行设置：

```
static void Set_Transmitters_and_Sensors_fan_beam(const int
current_rotate_degree, const int num_detector_degree, const double R_circle,
Point2d* Trm, Point2d* Rcv);
```

其他的代码直接可以复用“basic_radon_functions.cpp”中自定义的投影基本的功能函数。

极大地减少了不必要的重复代码，提升了代码的复用率。

Shepp-Logan的扇形束投影结果如下：

参数为探测器旋转360度，扇形所对圆心角为90度。

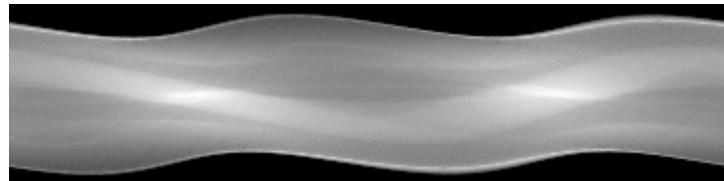


图6.1.2 扇形束投影图像（旋转360度，扇形90度）

2、扇形束滤波反投影的思路

由于时间原因尚未完成扇形束滤波反投影的完整重建，总体来说扇形束重建可以有直接扇形束重建，和扇形束重排平行束两种思路。与平行束的重建之间，存在举一反三的关系，但增加了小角度近似等步骤。

参考资料：以下书本的其中两节内容

Avinash C. Kak, and Malcolm Slaney, "Principles of Computerized Tomographic Imaging", IEEE Press, 1999. <https://engineering.purdue.edu/~malcolm/pct/>, Chapter 3. Algorithms for Reconstruction with Nondiffracting Sources.

- 3.4 Reconstruction from Fan Projections
- 3.5 Fan Beam Reconstruction from a Limited Number of Views

第七章 软件特色及完善方向

1、软件亮点和特色

1.1 用户友好

- 可调节参数多，适合初学者通过这款软件，更加深入地理解滤波反投影。
 - 可以通过调节Shepp-Logan模型的尺寸大小，深入理解图像尺寸对于重建效果的影响；
 - 可以去除滤波器，观察直接反投影和滤波反投影的效果差异；
 - 可以调节总投影角度，观察总投影角度不够多，或者超出理论最小的总投影角度时，不同的重建效果；
 - 可以慢速显示重建过程，并且对于Shepp-Logan模型，可以直观地观察到不同椭圆对于投影图像的不同贡献（在Shepp-Logan模型中的投影为椭圆的依次叠加，叠加顺序与本文档中的Shepp-Logan参数的列举顺序一致）；
- 左侧在重建后显示误差评价指标，在主观图像重建效果观察的基础上，也可以对不同图像的重建效果进行定量比较；
- 软件中提供使用指南，并且鼠标在大多数位置停留都可以看到ToolTip提示，软件使用方便。

1.2 支持图像类型丰富

对长方形的图像进行补零处理，对彩色图像进行灰度化操作，从而实现了对丰富图像类型的图像读入。

支持支持彩色输入（输出为灰度图），支持长方形图片输入，图像类型支持*.png, *.jpg, *.bmp。

1.3 注释完整

核心函数都提供了极其详细的函数说明注释，以及行内注释。

函数说明的注释格式参考了OpenCV的函数注释，格式标准。

以数字积分投影的函数为例，函数说明注释如下：

```
/** @brief Parallel line radon

@param image Input 8-bit 1-channel image.
@return radon_Img Output 64-bit 1-channel image.
    Size:rows = number of receivers = sqrt(2) * (maximum side of input
image)
    cols = number of degrees( default = 180)
```

The function references the article:

Siddon, Robert L. "Fast calculation of the exact radiological path for a three-

dimensional CT array."

Medical physics 12.2 (1985): 252-255.

```

@note
- the output image(radon_Img) has a narrow effective grayscale with a large
grayscale range.

Therefore, simple process of radon_Img before showing and storing the
result is needed.@ref function-Convert_to_show_radon_parallel_line
*/
Mat Radon_parallel_line(Mat image, int num_degrees)
{
    //此处省略函数体...
}

```

1.4 提供简洁的代码接口（利于后续开发）

尽管有很多自定义的函数代码，但是通过文件层次整理和函数封装，为使用提供了及其简洁的接口，便于后续进一步开发，代码复用以及代码维护。

以下是部分核心的的测试代码：

```

//=====Part1. 测试代码的参数设定部分
=====
int radon_flag = radonFlags::analytical; // for shepplogan
//int radon_flag = radonFlags::integral; // integral:for existing image

int radon_type = radonTypes::parallel;
//int radon_type = radonTypes::sector;

string image_name = "women"; //如果导入图片用数字积分，必须与图片文件名一致
Mat radon_Img = Mat::zeros(256, 256, COLOR_BGR2GRAY); //初始化投影数据
Mat Img;

//=====平行束测试代码=====
if (radon_type == radonTypes::parallel)
{
    // 投影-----
    if (radon_flag == radonFlags::analytical)
    {
        image_name = "shepplogan_analytical";
        int rows_of_shepplogan = 256;
        Img = Create_shepplogan(rows_of_shepplogan);
        radon_Img = calc_shepplogan_radon(rows_of_shepplogan);
    }
    if (radon_flag == radonFlags::integral)
    {
        //...此处省略读入图片
        radon_Img = Radon_parallel_line(Img); //自己写的平行束投影函数
    }
    // 滤波 -----
    Mat filtered_radon_Img = Filter_radon_parallel_line(radon_Img, SL_filter);
    // 反投影 -----
    Mat filtered_iradon_Img = iRadon_parallel_line(filtered_radon_Img);
    write_and_show_Img(filtered_iradon_Img, image_name, "filtered_iradon_SL");
}

```

```

else//=====扇形束投影测试代码
=====
{
    if (radon_flag == radonFlags::integral)
    {
        //...此处省略读入图片
        radon_Img = Radon_fan_beam(image, 360, 90); //自己写的扇形束投影函数
        write_and_show_Img(radon_Img, image_name, "radon_fanbeam");
    }
}

```

2、软件完善方向

2.1 重建效果的提升

滤波反投影算法在理想、连续的情况下，在数学上可以实现完美重建。但是实际情况中由于图像的离散化、理想滤波器无法实现，不可避免地会因为插值、近似等操作，产生误差。若要提升滤波反投影的效果，可以从以下方面入手：

滤波器的选择：本算法中采用了S-L滤波器。但在实际CT的重建算法中，还可能对特定频率成分进行抑制或者增强，达到图像增强的效果。而滤波器可能也会在常规的sinc窗函数、hanning窗函数上进行变形。

反投影的插值：本算法中采用的插值算法是最简单的一次线性插值，为了提升插值的效果，可以进一步采用二次、三次的插值算法。

2.2 空间、时间效率的提高

本算法对于高精度图片的处理速度较慢，占用的资源也较多。

如对3000*2000的高清图片进行重建时，将会占用较多的内存，如下图所示。



图7.2.1 处理高清图像时占用的内存

一方面，算法还有很大的优化空间，例如可以减少遍历的次数、减少图像拷贝的次数等等，以达到更高的空间和时间效率。

另一方面，**时间、空间效率，与代码的可读性、易维护性和复用性之间，存在矛盾**。例如，如果想要让扇形束的投影算法能够在兼容平行束投影算法的基础上进行设计，则需要为相同位置接收器开辟重复多个位置的存储空间；如果想要让代码的可读性、易维护性上升，则不可避免地需要对函数进行包装，也会增加函数之间相互调用、参数传递等带来的时间、空间成本。

2.3 扇形束的实现

后续可以在原有基础上，进一步扩展扇形束、锥形束的功能。

第八章 项目收获

1、从idea到代码的实现

本次大作业项目，让我完整地体验到了什么叫从一个idea到代码实现的过程。从滤波反投影的数学推导，到投影算法的数学推导的文献，都是非常理论性的东西，我需要先理解这些文献中前人提供的思路，然后从创建数组开始从零写起。

从理论到代码实现，其中最大的障碍就是连续到离散的过程，数学推导假设的数据都是连续的数据，而计算机中实际存储的像素都是离散的像素格，真是离散数据的特点，导致概念上非常简单的投影的过程，涉及到很多复杂的内容。我通过文献中"Fast calculation of the exact radiological path for a three-dimensional CT array."提供的思路，手绘数学建模，开辟存储空间，自己实现每一行代码和每一个函数，这个看着数学公式变成代码的过程是奇妙的，感觉像赋予了数学公式新的生命一样，使存在于纸张上的公式逐渐变成了可以运行、展示结果的算法。

代码实现和数学推导的区别，还有就是细节上的特殊情况的处理。数学公式中，也许只需要在公式最后附上理想的适用条件即可，例如“当 $X_1 \neq X_2$ 时”，而代码实现中，这些特殊情况还需要更多细节上的考虑，就以旋转角度为 90° 和 180° 时的投影需要特殊处理为例，单纯地以探测器和接收器的横坐标相等作为比较标准，不足以保证两者连线垂直于 x 轴的情况就可以被筛选出来，原因是计算探测器和接收器的坐标时为小数运算，存在近似，两者的计算结果可以非常相近（0.00001 级别），但是可能并没有完全相等，在这样的情况下，一种思路就是需要对原始的角度进行判断，还有一种思路是将两者之差与一个很小的值进行比较，无论是哪种思路，而不是简单的判断 $if(x_1! = x_2)$ 能够解决的。

最后一点想要强调的，是代码整理的重要性。代码整理的意识是本科阶段逐渐培养的，大一的时候很少有代码整理的意识，或者说难以培养整理代码的意识，是因为那时写的代码都比较简单，代码量也比较少，面对一目了然的东西，当然不会下决心把代码整理地更加整齐。微机原理的课程中，由于刚接触汇编语言时不容易理解，而且循环体结构都需要通过 JMP, JE 等语句进行跳转实现，代码不再那么一目了然，自然而然地就会自发地用过程 PROC 对代码进行封装，便于编程中的维护，也就是函数包装的意识。在图像 1 的大作业中，也对这种函数的封装进行了实践，函数封装确实可以让繁杂的主函数变得非常清晰。在嵌入式的课程中，接触到了更多工程文件，需要建立文件和文件之间的关系，把项目的代码逻辑分块分文件实现，也就养成了为不同的功能模块新建不同的.cpp 文件的习惯，这样的整理方式，可以让较为大型的代码项目的逻辑架构更加清晰，也更利于后续代码的复用、修改、维护，也增强了代码的可读性。而这次项目，将这些经验应用进来，可以明显感觉在编程的时候需要修改一个细节的时候，能够更加快速地定位需要修改的位置，在编写扇形束时，能够很方便地将平行束部分代码进行复用（不是复制，是直接调用之前写的相关基础函数），在制作 GUI 的时候，也能更加方便地开发更多的交互功能。

2、注重过程记录

对于一个时间跨度比较长的项目，我个人觉得适当的过程记录还是很重要的，便于隔了一段时间之后接着做项目的时候能够快速上手。也便于查找一些之前设置的参数，以免忘记。在写软件设计报告的时候，也有记录可以参考。

另外，一些环境配置、打包 exe 等技能性的工作也需要自学，及时记录可以再今后需要再次操作（如换电脑等）时，可以快速找到自己记录的“教程”。

以下是该项目过程中的一些记录的截图。



图8.2.1 项目的过程记录

代码的备份也很重要，避免某一次修改不小心造成的bug，使得整个项目的代码崩溃无法运行。以下是我阶段性进行的一些代码备份，一般都是在某些功能有一些小突破的时候，会进行一次备份。

名称	修改日期	类型
CT_codes_radon	2022/5/5 0:09	文件夹
CT_codes_radon - 1shepplogan投影ok 带有测试cout	2022/4/6 12:18	文件夹
CT_codes_radon - 2shepplogan投影ok radon封装	2022/4/6 19:56	文件夹
CT_codes_radon - 3shepplogan投影ok radon封装 0度90度bug解...	2022/4/6 23:54	文件夹
CT_codes_radon - 4radon搞定	2022/4/19 15:17	文件夹
CT_codes_radon - 5直接反投影ok, 但是转了90度? ?	2022/4/20 0:22	文件夹
CT_codes_radon - 6直接反投影ok!	2022/4/20 11:56	文件夹
CT_codes_radon - 7代码打包封装ok (准备滤波反投影)	2022/4/20 13:15	文件夹
CT_codes_radon - 8滤波反投影 (RL滤波) ok	2022/4/21 0:18	文件夹
CT_codes_radon - 9平行束滤波反投影ok	2022/4/23 16:26	文件夹
CT_codes_radon - 10准备解析法计算shepplogan投影	2022/4/23 18:12	文件夹
CT_codes_radon - 11解析法ok (考虑直方图均衡化后显示, 以及函...	2022/4/24 22:42	文件夹
CT_codes_radon - 12平行束滤波反投影ok (待代码整理)	2022/4/25 0:19	文件夹
CT_codes_radon - 13代码层级结构整理ok	2022/4/25 15:40	文件夹
CT_codes_radon - 14文件夹整理ok (准备GUI)	2022/4/25 15:55	文件夹
CT_codes_radon - 15平行束ok, 准备扇形束	2022/5/3 23:30	文件夹
CT_codes_radon - 16平行束+误差计算ok, 准备扇形束	2022/5/4 15:17	文件夹
CT_codes_radon - 17扇形束投影ok	2022/5/4 16:45	文件夹

图8.2.1 项目代码的备份

3、本软件的开发思路介绍

软件的开发过程并不是线性的先写投影，再写滤波，再写反投影的过程。而是以测试方便的形式，先写出便于分辨代码整体大方向是否正确的代码。

以下提供一个从零开始编写平行束滤波反投影软件的思路，仅供参考。

- 先测试图片的输入输出流，使用OpenCV，正常读入图片，并且把这个图片输出（如写到同目录下的png图片）；
- 完成投影步骤，并且把投影结果的图片输出。对基本图形（如各向一致的圆，90度旋转对称的正方形等）进行投影，并查看各行投影数据是否基本合理。对Shepp-Logan模型进行投影，并与网上其他人的投影结果图像进行比对，是否基本一致；

- 编写反投影步骤，（为了提高代码测试的效率，把投影结果以txt的形式保存下来，图片格式为了显示已经归一化到0-255，可能存在信息损失），读入txt的投影数据，进行反投影，得到直接反投影结果；观察shepplogan模型直接反投影的各椭圆大致位置是否正确；
- 最后编写滤波步骤（因为滤波较简短，而且如果先写滤波，无法从滤波图像结果上判断滤波的正确性。如果结合反投影之后，图像重建结果不对，也难以判断是滤波函数的问题，还是反投影函数的问题。因此在调试完反投影函数之后，才写滤波函数），对不同的滤波器进行尝试；
- 编写解析法，把投影数据修改成解析法生成的shepplogan投影，进行测试；
- 编写GUI；
- 封装exe。

第九章 参考资料

- 教材《医学成像的基本原理》第五章：X射线计算机断层成像（编著：黄力宇）
- Shepp-Logan的参数以及解析法投影公式：
 - 参考Avinash C. Kak, and Malcolm Slaney, "Principles of Computerized Tomographic Imaging", IEEE Press, 1999. <https://engineering.purdue.edu/~malcolm/pct/>, Chapter 3. Algorithms for Reconstruction with Nondiffracting Sources.
- 一般图像的投影（数字积分法）：
 - 参考Siddon, Robert L. "Fast calculation of the exact radiological path for a three-dimensional CT array." Medical physics 12.2 (1985): 252-255.
- 其他主要参考资料
 - [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\(Matlab代码的实现\)](#)
 - [opencv学习（十五）之图像傅里叶变换dft](#)
 - [滤波反投影重建算法（FBP）实现及应用（matlab）](#)
 - [椭圆变换方程（平移+旋转等任意变换）](#)
 - [C++图形化GUI开发框架推荐](#)
 - [如何用C++从零编写GUI？](#)
 - [Visual Studio 2019如何安装Qt插件](#)
 - [Qt: 03---Visual Studio安装Qt与使用](#)
 - [Qt官方文档](#)
 - [QWidget_QDialog_QMainWindow的异同点](#)
 - [VS打开Qt的ui界面几秒后闪退（ui无法打开文件）的解决办法](#)
 - [VS+QT快速入门教程](#)
 - [QT读取文件夹图片](#)
 - [QT读取图像显示](#)
 - [Qt之QIMAGE与MAT\(OPENCV\)](#)
 - [3-QtDesigner设计界面布局](#)
 - [Qt程序打包成一个单独exe的方法](#)
- 感谢赵老师的教导，以及其他未能全部列出的各大网友的博客、文章等提供的资料，在debug中提供了很大的帮助。

附录

如需查看具体代码，请见codes文件夹下的“3_核心代码（无测试代码）”中的代码，代码已经按照代码层次逻辑整理好。

附录仅提供各头文件中的函数原型，便于读者快速地查找感兴趣的函数。

代码的逻辑架构：

```
|-- core/                                // 核心投影、反投影、滤波的实现
|   |-- parallel_line                     // 实现平行束滤波反投影
|   |   |-- radon_parallel_line.cpp      // 投影、反投影的实现
|   |   |-- filter_parallel_line.cpp    // 对投影函数进行滤波
|   |-- fan_beam                          // 尝试扇形束滤波反投影（仅实现了投影部分）
|   |   |-- radon_fan_beam.cpp          // 实现扇形束的投影
|   |-- basic_radon_functions.cpp        // 投影的底层基础函数（仅由上方的文件@内部调用@）

|-- model/                                // 模型
|   |-- shepplogan.cpp                  // 解析法构建shepplogan模型及其投影结果

|-- utils/                                 // 基本函数（如计算最大值、点到直线距离等）
|   |-- basic_functions.cpp            // 显示图像、输出图像所需函数
|   |-- basic_display_functions.cpp

|-- GUI/                                   // 显示界面相关函数
|   |-- CT_application.h              // GUI界面的Qt类定义
|   |-- CT_application.cpp           // GUI界面的回调函数的实现
|   |-- CT_application.ui            // GUI界面设计（按钮位置、组件位置等）
|   |-- CT_main.cpp                  // 主函数（测试代码，实际exe运行不依赖于该main函数）
```

主函数需要调用的头文件：

```
#include<iostream>
#include<opencv2/opencv.hpp>

#include "basic_display_functions.h"
#include "basic_functions.h"
#include "shepplogan.h"
#include "radon_parallel_line.h"
#include "filter_parallel_line.h"
```

1、basic_display_functions.h

```
cv::Mat Convert_to_show_image(cv::Mat Img);
cv::Mat Convert_to_show_normalize(cv::Mat Img);
cv::Mat Convert_to_show_equalizeHist(cv::Mat inputImg);
cv::Mat Convert_to_show_LogTrans(cv::Mat inputImg, double gamma = 1.5);

void showImg(string windowname, cv::Mat img);
void write_and_show_Img(cv::Mat image, string image_name, string type_name);
int WriteData(string fileName, cv::Mat& matData);
int LoadData(string fileName, cv::Mat& matData, int matRows, int matCols, int matChns);

double calculate_error_sqrt(cv::Mat image1, cv::Mat image2);
double calculate_error_abs(cv::Mat image1, cv::Mat image2);
```

2、basic_functions.h

```
float sumMat(cv::Mat& inputImg);
cv::Mat Resize_img_to_Square(cv::Mat img);
int max(int a, int b);
int min(int a, int b);
double max(double a, double b);
double min(double a, double b);
double max_3(double a, double b, double c);
double min_3(double a, double b, double c);
double distance_2d(cv::Point2d A, cv::Point2d B);
```

3、shepplogan.h

```
cv::Mat Create_shepplogan(int rows = 256);
cv::Mat Calc_shepplogan_radon(int rows = 256);

void Calc_shepplogan_radon_show(int rows, int index, cv::Mat& radon_Img); //仅供
GUI【展示重建过程】时使用
```

4、radon_parallel_line.h

```
static int Set_sensors_length_parallel_line(const cv::Mat inputImg);
static void Set_Transmitters_and_Sensors_parallel_line(const int theta_degree,
const int sensor_length,
cv::Point2d* Trm, cv::Point2d* Rcv);

static double Cal_distance_point_to_line(cv::Point2d Point, cv::Point2d A1,
cv::Point2d A2);
static double Get_grayscale_iradon(double distance, const cv::Mat radon_Img,
const int sensor_length, const int theta_degree);

cv::Mat Radon_parallel_line(cv::Mat image, int num_degrees=180);
cv::Mat iRadon_parallel_line(cv::Mat radon_Img);

void Radon_parallel_line_show(cv::Mat image, int sensor_length, int
current_degree, cv::Mat& radon_Img); //仅供GUI【展示重建过程】时使用
void iRadon_parallel_line_show(cv::Mat radon_Img, int current_degree, cv::Mat&
output_Img); //仅供GUI【展示重建过程】时使用
```

5、filter_parallel_line.h

```
cv::Mat Filter_radon_parallel_line(cv::Mat inputImg, int filter_flag = 0);
```

6、basic_radon_functions.h (仅供内部函数调用, 非外部接口)

```
void calculate_range_of_alpha(const int i, const Point2d* Trm, const Point2d*
Rcv, const Mat image,
bool& flag_intersection, double& min_alpha, double& max_alpha);
```

```

void calculate_range_of_index_to_image(const int i, const Point2d* Trm,
    const Point2d* Rcv, const Mat image, const double min_alpha, const double
max_alpha,
    int& intersection_n_min, int& intersection_n_max, int& intersection_m_min,
int& intersection_m_max);
void calculate_alpha_x(const int i, const Point2d* Trm, const Point2d* Rcv,
    const Mat image, const int intersection_n_min, const int sizeof_alpha_x,
double* alpha_x);
void calculate_alpha_y(const int i, const Point2d* Trm, const Point2d* Rcv,
    const Mat image, const int intersection_m_min, const int sizeof_alpha_y,
double* alpha_y);
void calculate_alpha(const int sizeof_alpha_x, const int sizeof_alpha,
    const double* alpha_x, const double* alpha_y, double* alpha);
double calculate_projection_onevalue(const int i, const Point2d* Trm,
    const Point2d* Rcv, const Mat image, const int sizeof_alpha, const double*
alpha);
void Project_copyto_2Darray(const int theta_degree, const double* projection_1d,
Mat& radon_Img);

```

7、CT_application.h (GUI回调的声明)

```

#pragma once
#ifndef CT_APPLICATION_H
#define CT_APPLICATION_H

#include <QtWidgets/QMainWindow>
#include "ui_CT_application.h"

#include<iostream>
#include<opencv2/opencv.hpp>

#include "basic_display_functions.h"
#include "basic_functions.h"
#include "shepplogan.h"
#include "radon_parallel_line.h"
#include "filter_parallel_line.h"

class CT_application : public QMainWindow
{
    Q_OBJECT

public:
    CT_application(QWidget *parent = Q_NULLPTR);

private:
    Ui::CT_applicationClass ui;
    cv::Mat original_image; //存储原始图像（作为最后比对的标准）
    cv::Mat image; //初始化图像
    int radon_flag = radonFlags::analytical; // 投影模式（解析法or数字积分，可选择）默认shepplogan解析法
    //int radon_flag = radonFlags::integral;
    int radon_num_degree = 180;//数字积分（本地图片）的投影角度
    int size_of_shepplogan = 256;//解析模型的图像尺寸（默认参数256，可输入）

```

```
bool use_filter = 1; //使用滤波器（默认使用，即滤波反投影），若不使用，则为直接反投影
bool show_reconstruction_process = 1; //延时显示重建过程（默认不显示，即快速显示结果）
int reconstruction_speed = 5; //重建速度（1-5）
bool first_time_flag_text = 1; //是否是第一次进入“改变内置的解析法的shepplogan图像尺寸（输入框）”的函数，避免软件开启的显示问题
bool first_time_flag_slider = 1;

private:
    void displayMat_color(QLabel* label, cv::Mat image);
    void displayMat_gray(QLabel* label, cv::Mat image);

private slots:
    void display_img_click(void); //点击【显示图片】按钮
    void size_of_shepplogan_changed(int); //改变内置的解析法的shepplogan图像尺寸【输入框】
    void size_of_shepplogan_changed_slider(int value); //改变内置的解析法的shepplogan图像尺寸【滑动条】
    void num_degrees_changed(int value); //改变数字积分法的投影总角度【输入框】
    void num_degrees_changed_slider(int value); //改变改变数字积分法的投影总角度【滑动条】
    void speed_changed_slider(int value); //改变重建速度的【滑动条】
    void show_reconstruct_result_click(void); //点击【开始重建】按钮
    void use_filter_check(bool value); //修改【使用滤波器】的勾选状态（不勾选等同于“直接反投影”，勾选后使用S-L滤波）
    void show_process_check(bool value); //修改【显示重建过程】的勾选状态
    void comboBox_imgtype_changed(int value); //修改投影类型（解析法or数字积分）的【复选框】
};

#endif /* CT_APPLICATION_H */
```