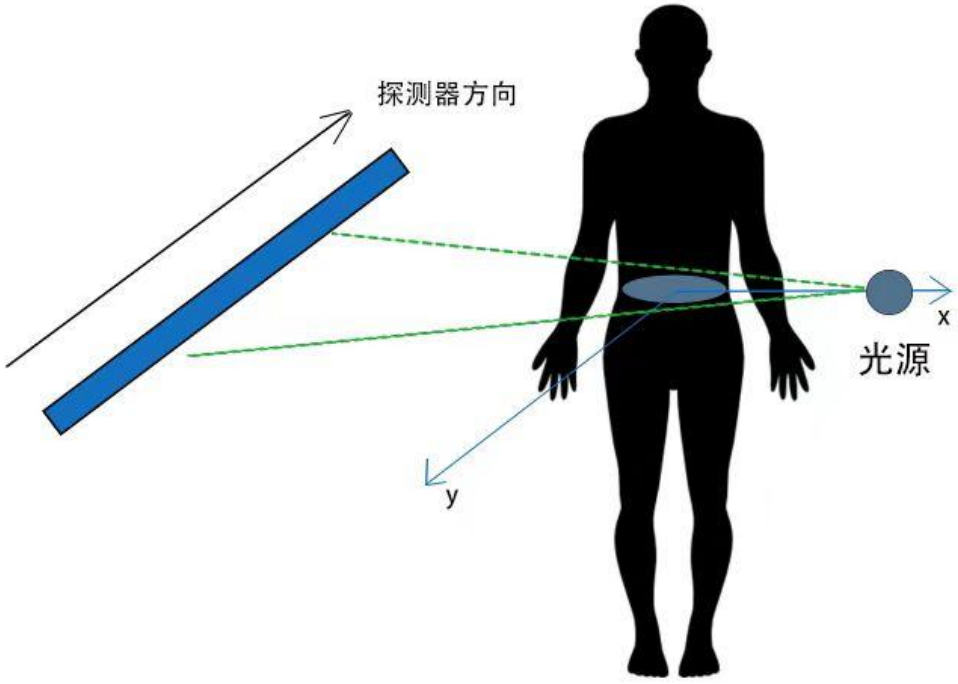
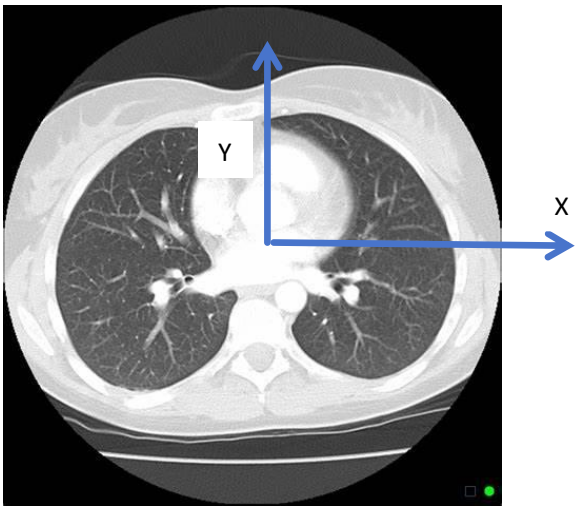
实际系统中正投影数据的获取：

对于图像重建(二维)中正投影数据的获取，我们的实际系统通常如图一摆放，其中虚线代表光源发出的光束，想象这些光束可以构成一个从外到内的横截面，从而照射图中圆所在的平面打到矩形模样的探测器上以得到人体在这个横截面上的一个角度的正投影数据。通过旋转光源与探测器(或旋转物体)即可得到所需数量角度的这个横截面的正投影数据。



图一

我们要重建的是图中横截圆面所截的人体内部心肺图，它大概如图二所示:



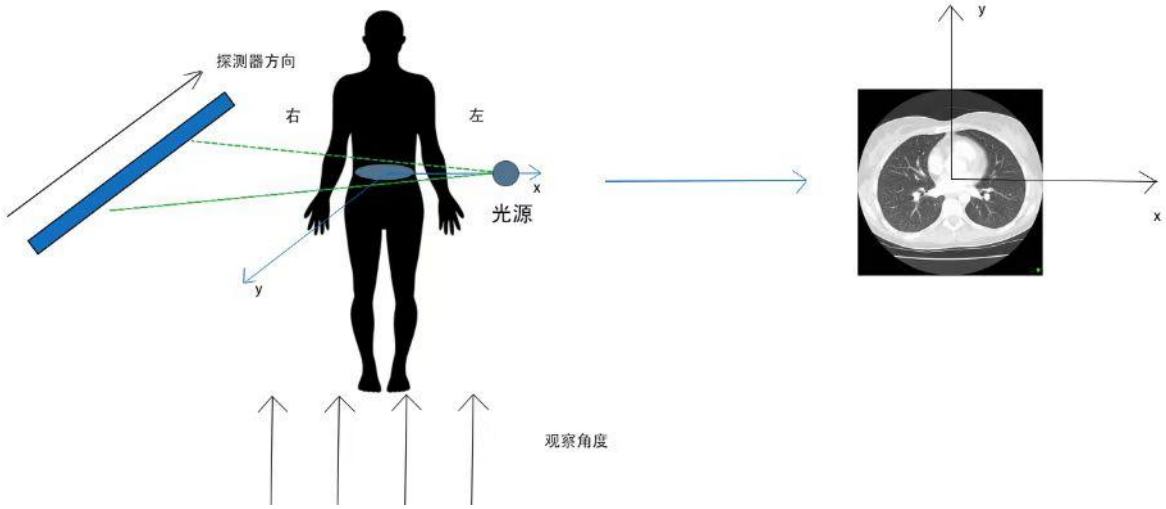
右

左

图二

图二是从人体**下方向上方看去看到的那个横截面图像**，我们标注了坐标系，这是为了方便后续反投影计算。所以坐标系标注的原则就是从人体下方向上方看去，

看到的横截面中心处作原点，水平向右为 X 轴，竖直向上为 Y 轴。体现在图一上可以按图三理解：



图三

此外，按照如上所述，所重建的图片的**右侧对应的是实际人体的左侧**，可通过仔细对比图二与图三去理解图片左右侧与实际人体左右侧的对应关系。

注：人体 CT 图一般左右是相反的。

常见的实际系统：

这是一个通过旋转物体从而得到各个角度正投影数据的系统，图中 PCD 标识的黑色板子上的白色矩形框就是一个探测器。左侧机器产生光束穿过人头部打在右侧探测器上即可得到一个角度上的脑部正投影数据。通过下方的 Rotary stage 旋转人头部即可得到各个角度的正投影数据。可将此图中人头对比图一的人体，光源与探测器也分别对应到图一来体会。可尝试在此图中标出坐标系来加深理解。

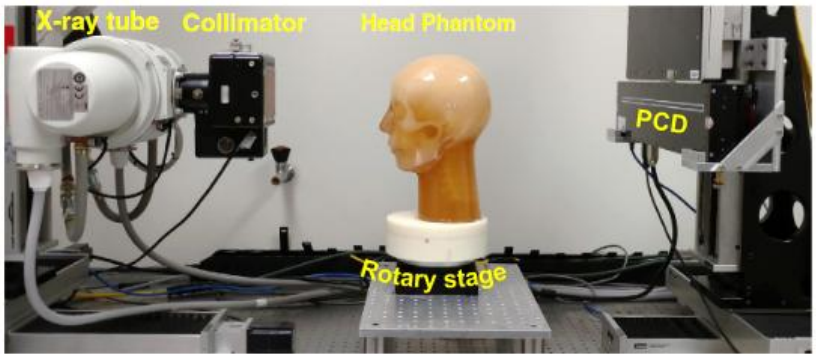
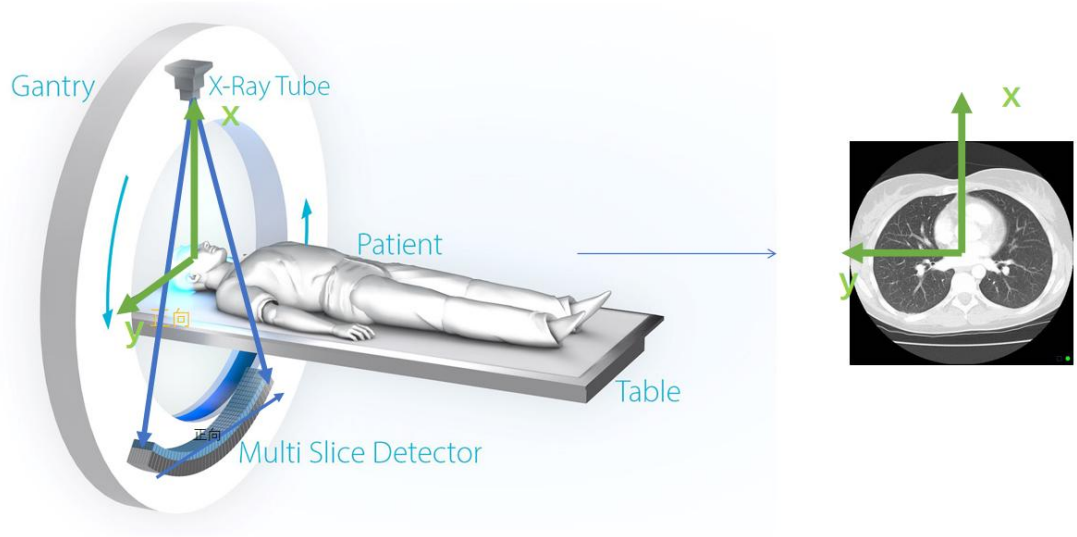


Figure 1. The PCCT benchtop system used in this work.

图四

图四为医院中常见的 CT 系统，光源与探测器内嵌于器械内部，位置如图所示。



图五

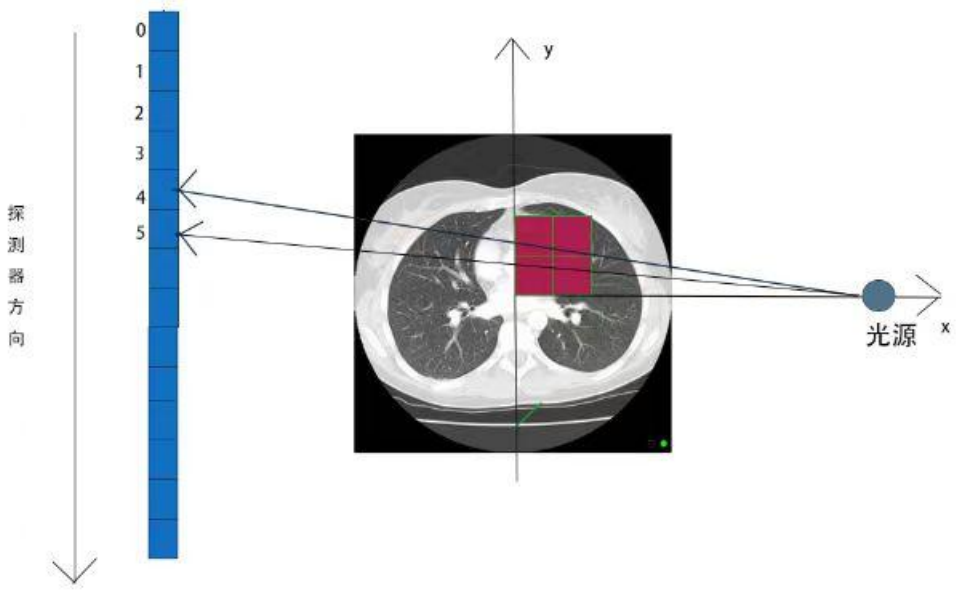
此系统（如图五所示）是通过旋转光源与探测器来获得各个角度的正投影数据的(旋转人体不现实，会把人转晕掉)。同样，可以与图一对应去加深理解。

反投影过程中图片像素与探测器像素的对应问题：

下面我们将详细介绍反投影的过程。根据我们之前学到的知识，反投影实际上就是将各个角度的探测器上的正投影数据对我们的图片进行反投影最终得到我们的重建图。下面将介绍反投影过程的细节。

首先，反投影过程在实际应用中有两种，一种叫做基于光线的反投影，一种叫做基于像素的反投影。

在之前的学习中我们对反投影的理解一直都是基于光线的反投影，它的过程可以用以下的图六进行讲解。

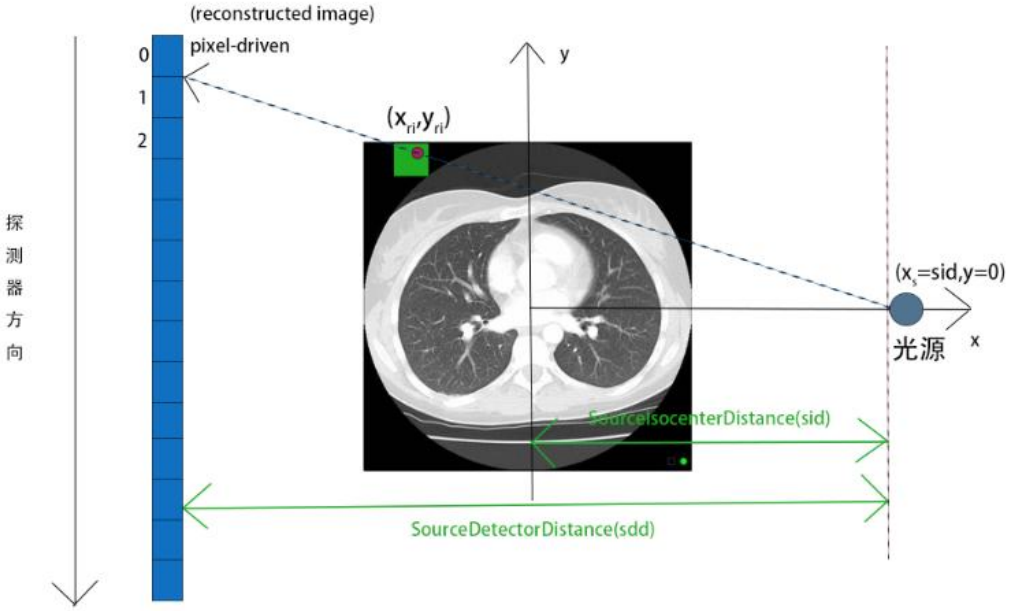


图六 基于光线的反投影

我们首先按照正投影时光源、探测器、图片在坐标系中的位置将其摆好。以一个角度下的反投影过程为例，在基于光线的反投影中，光束经过某一个像素的中心打到探测器上对应像素处，则此像素就用这个探测器像素进行反投影。从图中我们也可以发现，经过一个像素的光束并不一定经过像素的中心，且一个像素可能经过多个对应探测器坐标不同的光束，在这种情况下，我们难以确定该像素到底应该用探测器上的哪一个坐标位置进行反投影，需要进行插值操作。所以这种方式在实际实现时会非常困难。

在实际进行反投影的过程中，我们更常使用第二种方式，即基于像素的反投影。

在这种方式中，光源、探测器与图片的摆放不变。如下图所示，我们从像素的角度出发，默认每个像素(以图中橙色像素为例)的中心处都有一个光束经过，此光束打到探测器上的对应坐标位置，则使用这个探测器像素的值去反投影图像的像素。至此，**我们需要做的事就是计算图像的每个像素应当用探测器上哪一个坐标的像素反投影即可**。结合图七，我们的计算过程将在下方详细解释。



图七 基于像素的反投影

在进行计算之前，我们首先要介绍一些变量，它们有的在程序中有直接对应，有的是为了便于我们进行计算推理所使用的中间变量。对于程序中有对应的，下方会特意指出。

以图示中的坐标系来标识坐标，有如下变量(由黑色到黄色为一组，可分组理解)：

sid：光源到原点的物理距离，在图中已标出，程序中由配置文件给出。

sdd：光源到探测器的物理距离，在图中已标出，程序中由配置文件给出。

(x_s,y):光源的物理坐标，如图所示，其为(sid,0)。这是我们计算探测器坐标的关键物理坐标。

u_{idx}:探测器像素的下标，其标注方式图中已给出。

du:探测器像素的大小，单位为 mm。

u[i]:下标为 i 的探测器像素在竖直方向上距离原点的一维物理坐标。但其方向与 Y 轴方向相反，即原点上方的探测器像素 i 的 u[i]为负，原点下方 j 的 u[j]为正。简单理解的话就是各个探测器像素到 X 轴的距离，但 X 轴上方的要取负，X 轴下方的仍为正。至于为什么，可以结合程序进行理解。

从 mgfbp.cpp 文件开始按照如下顺序阅读程序：

```
fbp.InitParam();
```

↓

```
InitializeU_Agent(u, config.sgmWidth, config.detEltSize, offcenter_array_cpu[0]);
```

↓

```
void InitializeU_Agent(float* &u, const int N, const float du, const float offcenter)
{
    if (u != nullptr)
        cudaFree(u);

    cudaMalloc((void**)&u, N * sizeof(float));
    InitU << <(N + 511) / 512, 512 >> > (u, N, du, offcenter);
}
```

↓

```
__global__ void InitU(float* u, const int N, const float du, const float offcenter)
{
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    if (tid < N)
    {
        u[tid] = (tid - (N - 1) / 2.0f) * du + offcenter;
    }
}
```

在函数 InitU 中，我们进行了探测器各个像素 u[i]的初始化。其中，tid 代表探测器像素的下标，N 为探测器像素的数量，这就代表 tid 范围为 0~N-1，探测器中心像素的下标为(N-1)/2。offcenter 为配置文件中给出的探测器的中心像素的 u[(N-1)/2],一般为 0，有时会有较小误差。不会超过一个像素的大小。

故可以看到 0~(N-1)/2-1 下标范围内的 u 均为负值，比中心坐标大的那些为正，所以它们的坐标与 Y 轴方向是相反的。搞明白这件事才能理解后续关键物理坐标的计算。

(X_d,Y_d):探测器上某一探测器像素的物理坐标，是我们计算探测器坐标的关键物

理坐标。

类比**(x_s,y)**，可知：

$$X_d = -(sdd - sid)$$

$$Y_d = -((u_{idx} - 0) * (du) + u[0])$$

由前面理解了 u[0]是一个负值，就可以理解Y_d的计算了。

$(x_{ri\_idx}，y_{ri\_idx})$:图像最左上角像素设为(0,0)时，图像上任一像素点的坐标，此时图像中心即坐标原点处的坐标设为**((M-1)/2,(M-1)/2)**,M 为图像的高或宽(默认重建图像高宽一致)。再次强调这个坐标不是图中坐标系的坐标，是以图像最左上角像素为原点时的坐标。

dx:重建图像的像素尺寸，单位为 mm。

(x_c,y_c):图像中心点的物理坐标，即图像中点点到坐标原点的实际距离向量。其中x_c为 X 方向的距离矢量(有正负)，y_c为 Y 方向的距离矢量(有正负)。在程序中，我们将图像中心点设置到了坐标原点处，**(x_c,y_c)为(0,0)**。

(x_{ri},y_{ri}):图像上任一像素点的物理坐标，即某一点到坐标原点的实际距离向量。其中x_{ri}为 X 方向的距离矢量(有正负)，y_{ri}为 Y 方向的距离矢量(有正负)。

其计算方式为：

$$x_{ri} = (x_{ri\_idx} - (M-1)/2) * dx + x_c$$

$$y_{ri} = (y_{ri\_idx} - (M-1)/2) * (-dx) + y_c$$

此组数据在程序中均有对应，可从 mgfbp.cpp 文件开始按照如下顺序阅读程序：

```
fbp.BackprojectPixelDrivenAndSave((outDir /
mg::FbpClass::config.outputFiles[i]).string().c_str());
```

↓

```
BackprojectPixelDriven_Agent(sinogram_filter, image,sdd_array, sid_array, offcenter_array,
pmatrix_array, u, v, beta, config, z_idx);
```

↓

```
BackprojectPixelDriven_device << <grid, block >> > (sgm_flt, img, u, v, beta, config.shortScan,
config.sgmWidth, config.views, config.sliceCount, config.coneBeam, config.imgDim,
config.imgSliceCount, sdd_array, sid_array, offcenter_array, config.pixelSize,
config.imgSliceThickness, config.xCenter, config.yCenter, config.zCenter, z_idx);
```

↓

```
__global__ void BackprojectPixelDriven_device(float* sgm, float* img, float* u, float* v, float*
beta, bool shortScan, const int N, const int V, const int S, bool coneBeam, const int M, const int
imgS, float* sdd_array, float* sid_array, float* offcenter_array, const float dx, const float dz,
```

```
const float xc, const float yc, const float zc, int imgS_idx)
{

    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int row = threadIdx.y + blockDim.y * blockIdx.y;

    float du = u[1] - u[0];
    float dv = v[1] - v[0];


    if (col < M && row < M && imgS_idx <= imgS)
    {

        float x = (col - (M - 1) / 2.0f)*dx + xc;
        float y = ((M - 1) / 2.0f - row)*dx + yc;
...
}
```

此函数中 col 与 row 分别对应x_{ri_idx}与y_{ri_idx}。x 与 y 分别对应我们计算的x_{ri}与y_{ri}，变量 v 在此处可以先不考虑，此变量用于三维重建的情况。其余变量名称与上面完全对应，可对应理解。

理解了所有变量后，接下来我们便可以通过(x_s,y)，(X_d,Y_d)，(x_{ri},y_{ri})三点位于一条直线上这一性质通过(x_s,y)，(x_{ri},y_{ri})来计算出(X_d,Y_d)，进而计算出u_{idx}，来确定此像素应当用哪个下标的探测器像素上的数据来涂抹。公式的推导过程如下：

由三点在一条直线上可得

$$\frac{Y_d - y}{y_{ri} - y} = \frac{X_d - x_s}{x_{ri} - x_s}$$

代入(x_s,y),X_d有

$$\frac{Y_d - 0}{y_{ri} - 0} = \frac{-(sdd - sid) - sid}{x_{ri} - sid}$$

可解得

$$Y_d = \frac{sdd}{sid - x_{ri}} * y_{ri}$$

解得Y_d后，由公式

$$Y_d = -\{(u_{idx} - 0) * (du) + u[0]\}$$

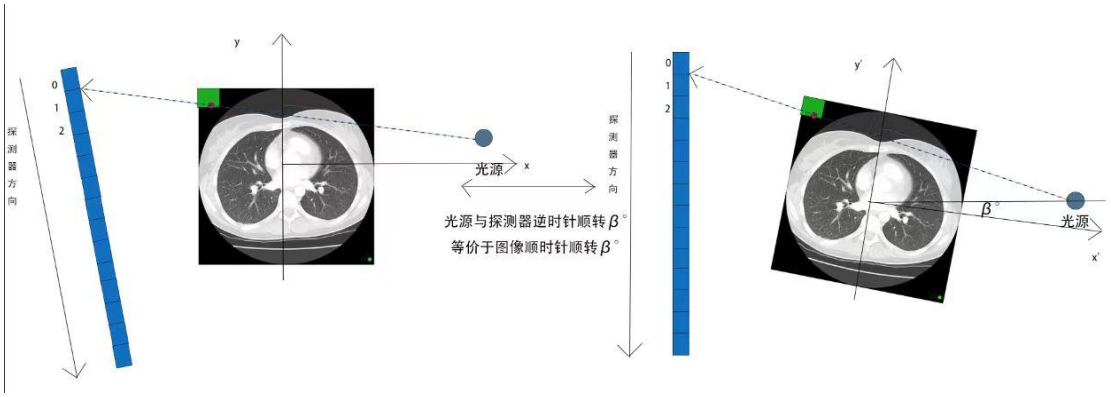
解得我们的u_{idx}

$$u_{idx} = (-Y_d - u[0])/(du)$$

至此，我们的重建图像上任一像素应当用探测器上哪个下标的数据来涂抹的问题就解决了。但要注意，这只是一个角度的正投影数据(滤波后)的涂抹问题，

一张正弦图是包含多个角度下的正投影数据的(滤波后)。接下来我们会进行光源与探测器旋转任意角度 β 后u_{idx}的计算推理并得出更普遍的u_{idx}的计算方法，便于我们编程实现。

二维几何与反投影：



图八

由图八可知，当光源与探测器所在坐标系逆时针旋转 β 角度时，可等价地看做重建图像沿坐标轴中心顺时针旋转 β 角度。

根据上述几何关系可得初始坐标系 x 轴与 y 轴与旋转后坐标系 x'轴和 y'轴有如下几何关系（其中 e _x' 表示 x' 轴的基向量，e _y' 表示 y' 轴的基向量）:

$$e\_x' = (\cos\beta, -\sin\beta)$$
$$e\_y' = (\sin\beta, \cos\beta)$$

将 x, y 用 x' , y'表示如下:

$$(x,y) = x'e\_x' + y'e\_y'$$
$$= x' * (\cos\beta, -\sin\beta) + y' * (\sin\beta, \cos\beta)$$

可得

$$x = x'\cos\beta + y'\sin\beta$$
$$y = -x'\sin\beta + y'\cos\beta$$

根据之前所得几何关系有:

$$Y_d = \frac{sdd}{sid - x_{ri}} * y_{ri}$$

将,x_{ri}, y_{ri}代入上式可得:

$$Y_d = \frac{sdd}{sid - (x'\cos\beta + y'\sin\beta)} * (-x'\sin\beta + y'\cos\beta)$$

由之前所得表达式可求解出 u_{idx}:

$$Y_d = -\{(u_{idx} - 0) * (du) + u[0]\}$$
$$=> \ u_{idx} = (-Y_d - u[0])/(du)$$

由Y_d及 u0 的如下表达式:

$$Y_d = \frac{sdd}{sid - (x'\cos\beta + y'\sin\beta)} * (-x'\sin\beta + y'\cos\beta)$$
$$u0 = -Y_d$$

可得:

$$Y_d = -\{(u_{idx} - 0) * (du) + u[0]\}$$

$$=> u_{idx} = (u0 - u[0])/(du)$$

注释（代码中变量与上述公式字符对应关系如下）：

U：$sid - (x'\cos\beta + y'\sin\beta)$

mag_factor：$Y_d = \frac{sdd}{sid - (x'\cos\beta + y'\sin\beta)}$

u0：$\frac{sdd}{sid - (x'\cos\beta + y'\sin\beta)} * (-x'\sin\beta + y'\cos\beta)$

K：$u_{idx} = \frac{u0 - u[0]}{du}$

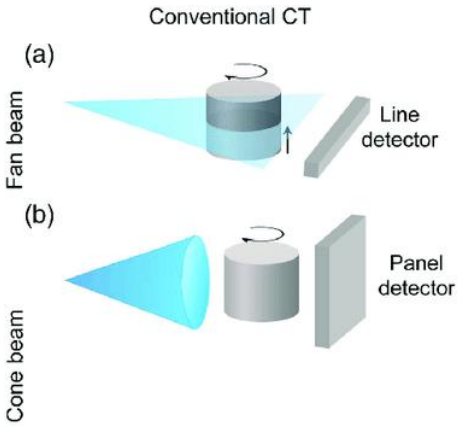
以下代码在函数 BackprojectPixelDriven_device 中如下:

```
U = sid - x * cosf(beta[view]) - y * sinf(beta[view]);

//calculate the magnification
mag_factor = sdd / U;

// find u0
u0 = mag_factor * (x*sinf(beta[view]) - y * cosf(beta[view]));


k = floorf((u0 - (u[0] + offcenter_bias)) / du);
if (k<0 || k + 1>N - 1)
{
    img_local = 0;
    break;
}

w= (u0 - (u[k] + offcenter_bias)) / du;
```

三维反投影中的 Fan beam（扇形束）与 Cone beam（锥形束）：

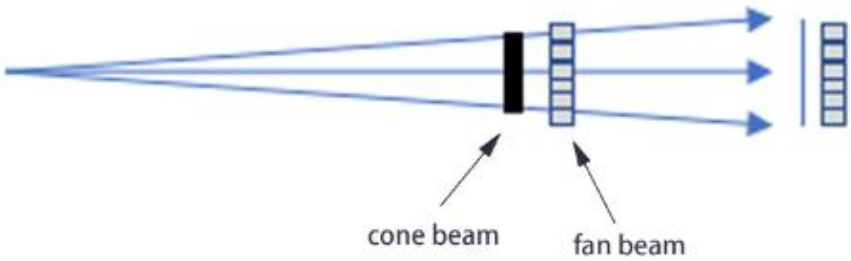
在学习过二维几何与反投影后，接下来我们将学习三维几何与反投影。从我们之前做过的实际数据去伪影的作业中也可以了解到，对于一个三维物体的重建，我们仍采取单一光源但其一个角度上正投影数据的获取并不是采取单独一个条状探测器，而是许多条状探测器进行堆叠从而一次性获得物体多个平面上的正投影数据。每个条状探测器可以通过获得多个角度上的正投影数据来拼出对应平面上的一张正弦图。

在正式学习前，我们首先要了解三维重建过程中正投影数据获取里的两种光束的概念，它们分别称为 Fan beam(扇形束)与 Cone beam(锥形束)。具体我们将结合图九进行解释。



图九

结合图九，我们可以先大致将 Fan beam 认为是仅取光源在一个平面内穿过物体打到探测器上的散射光束，在图九中呈现一个扇形状。而 Cone beam 则是取光源按照真实散射情况穿过物体打到探测器上的散射光束，呈现一个锥状。而事实上我们知道，即使是只使用一个条状探测器，探测器上的像素也是有高度的，所以严格意义上 Fan beam 是并不存在的，我们只是将 Cone angel(锥角)非常小的 Cone beam 近似看作 Fan beam。如果要对 Cone angel 做一个量化的话，那大概有如图十的标准，即 Cone angel 于 1 度的 Cone beam 我们称为 Fan beam，反之则称其为 Cone beam。



图十

这里要强调,虽然在图九中 Fan beam 的例子只用了一个条状探测器，这并不代表 Fan beam 只能去做二维重建，Fan beam 也可以去做三维重建即打到多个堆叠的条状探测器上。**我们要重点掌握使用 Fan beam 做三维重建与使用 Cone beam 做三维重建的区别。**我们仍按图十去进行理解，使用 Fan beam 去做三维重建时，我们可以近似认为每一道光束是平直穿过物体像素打到探测器像素上的，所以在高度方向上我们可以认为物体像素与探测器像素是一一对应的。使用 Cone beam 去做三维重建时，我们就不能忽略光束在高度方向上的散射，此时光束是从原点按照某一个角度倾斜射出穿过物体打到探测器上，在高度上物体与探测器便不再存在像素一一对应的关系，通常物体的物理高度要对应更大的探测器物理高度。可结合图十进行理解，将黑色的矩形看作是物体高度方向上一个切面，右侧的堆叠像素看作是堆叠的条状探测器的一个高度方向上的切面。

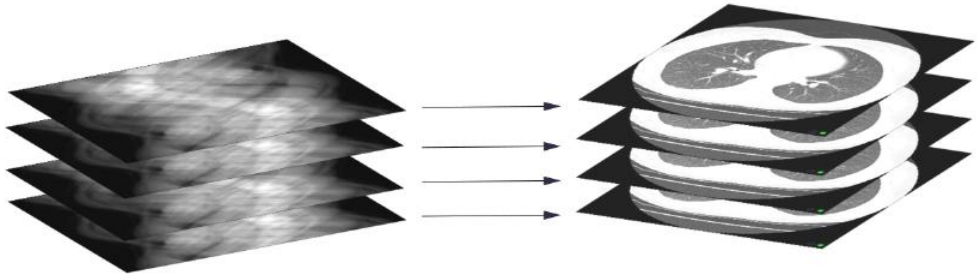
在程序的配置文件中可通过如下选项去进行 Fan beam 与 Cone beam 的选择。

```
    "ConeBeam": false,//不进行 Cone beam 重建算法进行重建
```

多层 Fan beam 重建：

由上一节的内容可知，当我们使用 Fan beam 去进行三维重建时，高度方向上物体像素与探测器像素一一对应，这也就代表我们在进行三维重建时所使用的堆叠正弦图的每一张与我们重建图像的每一个 slice 是一一对应的，可结合图十一进行理解。所以，当我们使用 Fan beam 去重建图像时，我们并不需要去指定输出图像应该有多少个 slice。对应到配置文件中就是下面这个参数在使用 Fan beam 重建时指定多少都是没有意义的，因为它已经由正弦图的数量决定。

```
"ImageSliceCount": 2,//进行 Fan beam 时指定多少都无所谓
```

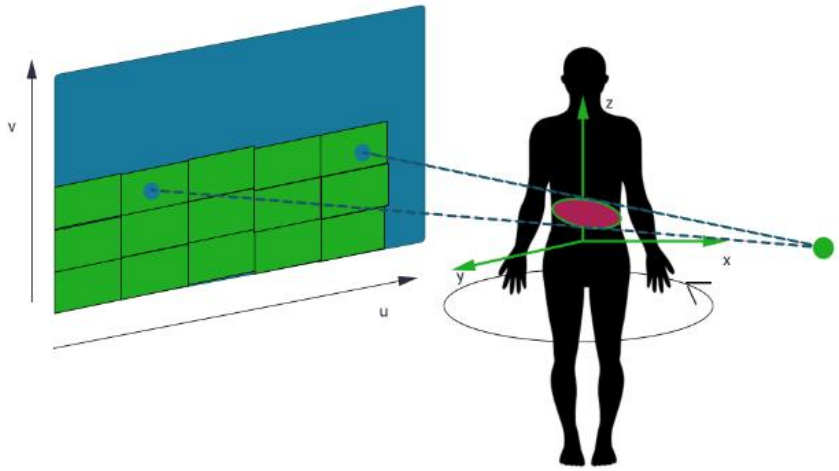


图十一

综上所述，对于 Fan beam 重建，我们可以把他想成是多个二维重建，一张正弦图对应一张重建图。这个重建里的反投影过程中图片像素与探测器像素的对应关系，与二维重建完全相同，只是进行了多张二维图片的反投影而已。此处的原理不再展开，在确定好正弦图与重建图的对应关系后，按照本文档中反投影过程中图片像素与探测器像素的对应问题这一节中所述便可明确其反投影的过程。

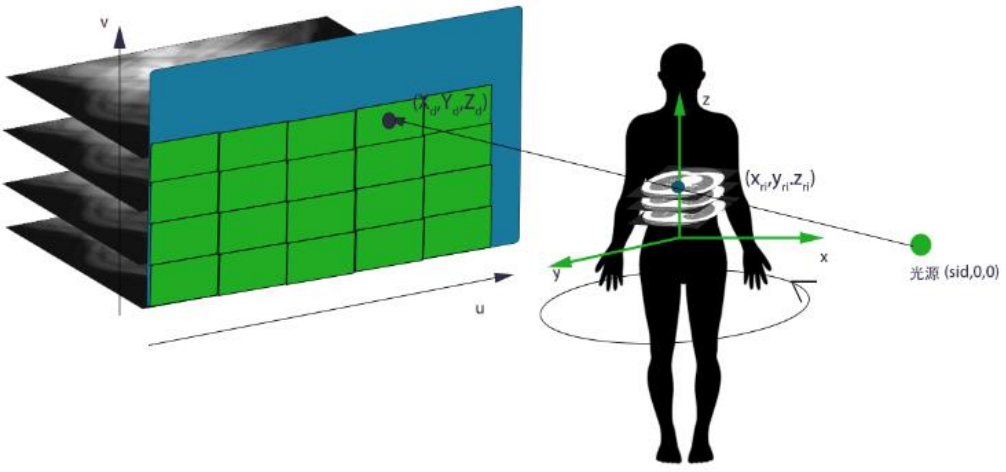
使用 Cone beam 的三维图像重建:

下面我们将详细学习使用 Cone beam 的三维图像重建，对于重建过程中正投影数据的获取与二维重建中类似。不同的是我们这次采用了多个条状探测器堆叠而成的矩形状探测器来获取正投影数据，**通过使光源与探测器在一个平面(即图十二中 X 轴与 Y 轴构成的平面)内转动不同的角度来获得不同角度的正投影数据**。可参考图十二来理解正投影数据的获取。



图十二

从图中我们可以看到，与二维重建过程不同的是，图像的坐标系增加了 Z 坐标轴。之所以这样做可以这样解释，因为在使用 Cone beam 去进行三维重建的过程中，重建图像有许多张，由于光束是倾斜着穿过人体打到探测器上的，**可以想象，打到一个条状探测器上的光束对应的正投影数据应当是人体的一个倾斜的横截面(图十二中红色圆所在的横截面)的正投影数据，而一个条状探测器在不同角度上测得的正投影数据会构成我们的一张正弦图，所以这张正弦图对应的是那个倾斜的人体横截面(红色圆面)重建图**，而我们的重建图是一张张与 X 轴和 Y 轴所构成的平面平行的人体横截面重建图(可见图十三)，所以我们的正弦图与我们的重建图并不存在一一对应的关系，当我们重建一张图像时，这张图像上的不同像素在反投影过程中会对应到不同的正弦图像素上，即会对应到不同的条状探测器上。**经过上述解释我们可以总结出，增加 Z 坐标轴有两个作用，一是在反投影过程中确定我们的重建图像素位于哪个 slice 上，二是确定我们的这个重建像素对应到哪个条状探测器上(为此我们的探测器也增加了一个 v 下标来区分不同条状探测器上的探测器像素)。**



图十三

懂得了重建图像的 slice(数量)与正弦图数量(条状探测器的数量)无关后，我们应该就可以理解为什么重建图像的 slice 可人为指定了。指定重建图像数量的

参数在配置文件中的对应上面已经给出，再次强调这个参数只有在使用 Cone beam 时有效。值得一提的是在实际工程中，对于正弦图的输入顺序我们习惯由底部向顶部输入，重建图我们也习惯按照从底部的 slice 向顶部的 slice 的顺序重建，即底部的 slice 优先重建。

我们要解决的问题仍是反投影过程重建图像像素坐标与探测器像素下标的对应关系，只不过这次重建图像像素坐标变为了三维，探测器像素增加了一个 v 来标识其所属的条状探测器下标。下面我们将类比二维重建中反投影过程里重建图像素与探测器像素的对应来解释三维重建(使用 Cone beam)中的坐标对应关系。以下会有二维反投影中重复的一些变量，可与前面对比理解。

sid：光源到原点的物理距离，在图中已标出，程序中由配置文件给出。

sdd：光源到探测器的物理距离，在图中已标出，程序中由配置文件给出。

(x_s,y,z):光源的物理坐标，如图所示，其为(sid,0,0)。这是我们计算探测器坐标的关键物理坐标。

u_{idx}:探测器像素的下标，其标注方式图中已给出。

v_{idx}:条状探测器的下标，其标注方式图中已给出，由底部到顶部顺序标注，最底部为 0。

du:探测器像素的大小，单位为 mm。

dv:条状探测器的宽度，也可以叫作正弦图的厚度,单位为 mm。其在配置文件中对应如下：

```
"SliceThickness": 6,
```

u[i]:下标为 i 的探测器像素在竖直方向上距离原点的一维物理坐标。但其方向与 Y 轴方向相反，即原点上方的探测器像素 i 的 u[i]为负，原点下方 j 的 u[j]为正。简单理解的话就是各个探测器像素到 X 轴的距离，但 X 轴上方的要取负，X 轴下方的仍为正。至于为什么，可以结合程序进行理解。

从 mgfbp.cpp 文件开始按照如下顺序阅读程序：

```
fbp.InitParam();
```

↓

```
InitializeU_Agent(u, config.sgmWidth, config.detEltSize, offcenter_array_cpu[0]);
```

↓

```
void InitializeU_Agent(float* &u, const int N, const float du, const float offcenter)
{
    if (u != nullptr)
        cudaFree(u);

    cudaMalloc((void**)&u, N * sizeof(float));
```

```
    InitU << <(N + 511) / 512, 512 >> > (u, N, du, offcenter);
}
```

```
__global__ void InitU(float* u, const int N, const float du, const float offcenter)
{
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    if (tid < N)
    {
        u[tid] = (tid - (N - 1) / 2.0f) * du + offcenter;
    }
}
```

在函数 InitU 中，我们进行了探测器各个像素 u[i]的初始化。其中，tid 代表探测器像素的下标，N 为探测器像素的数量，这就代表 tid 范围为 0~N-1，探测器中心像素的下标为(N-1)/2。offcenter 为配置文件中给出的探测器的中心像素的 u[(N-1)/2],一般为 0，有时会有较小误差。不会超过一个像素的大小。

故可以看到 0~(N-1)/2-1 下标范围内的 u 均为负值，比中心坐标大的那些为正，所以它们的坐标与 Y 轴方向是相反的。搞明白这件事才能理解后续关键物理坐标的计算。

v[i]:下标为 i 的条状探测器在 Z 轴方向距原点的一维物理坐标，也可理解为某一个探测器像素的 z 物理坐标，等价与下面的Z_d。其在程序中有对应(与 u[i]类似)。

从 mgfbp.cpp 文件开始按照如下顺序阅读程序：

```
fbp.InitParam();
```

```
InitializeU_Agent(v, config.sliceCount,config.sliceThickness,config.sliceOffcenter);
```

```
void InitializeU_Agent(float* &u, const int N, const float du, const float offcenter)
{
    if (u != nullptr)
        cudaFree(u);

    cudaMalloc((void**)&u, N * sizeof(float));
    InitU << <(N + 511) / 512, 512 >> > (u, N, du, offcenter);
}
```

```
__global__ void InitU(float* u, const int N, const float du, const float offcenter)
{
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    if (tid < N)
```

```
    {
        u[tid] = (tid - (N - 1) / 2.0f) * du + offcenter;
    }
}
```

v[i]初始化与 u[i]类似，注意此处的 InitU 中 du 对应的是 sliceThickness 即之前提到的 dv，offcenter 对应的是 sliceOffcenter，它的含义是堆叠的那些条状探测器(假设条状探测器共 N 个)中心的那个条状探测器的 v[(N-1)/2],一般为 0，有时会有较小误差。不会超过一个 sliceThickness 的大小。

(X_d,Y_d,Z_d):探测器上某一探测器像素的物理坐标，是我们计算探测器坐标的关键物理坐标。

类比**(x_s,y,z)，**可知：

$$X_d = -(sdd - sid)$$
$$Y_d = -((u_{idx} - 0) * (du) + u[0])$$
$$Z_d = ((v_{idx} - 0) * (dv) + v[0])$$

由前面理解了 u[0]是一个负值，就可以理解Y_d的计算了。

$(x_{ri\_idx}, y_{ri\_idx}, z_{ri\_idx})$:重建图像中最底部的那一张的最左上角像素设为(0,0,0)时，重建图像上任一像素点的坐标，此时坐标原点处图像像素的坐标设为**((M-1)/2,(M-1)/2,(imgSliceCount-1)/2,)**,M 为图像的高或宽(默认重建图像高宽一致),imgSliceCount 为我们指定的重建图像的 slice 数。再次强调这个坐标不是图中坐标系的坐标，是以最底部图像最左上角像素为原点时的坐标。

dx:重建图像的像素尺寸,单位为 mm。

dz:重建图像每一个 slice 的厚度,单位为 mm。在配置文件中有对应，如下：

```
"ImageSliceThickness": 2,
```

(x_c,y_c,z_c):位于中央的重建图像中心点的物理坐标。其中x_c为 X 方向的距离矢量(有正负)，y_c为 Y 方向的距离矢量(有正负)，z_c为 Z 方向的距离矢量(有正负)。在程序中，我们将中央图像中心点设置到了坐标原点处，**(x_c,y_c,z_c)为(0,0,0)**。

(x_{ri},y_{ri},z_{ri}):图像上任一像素点的物理坐标，即某一点到坐标原点的实际距离向量。其中x_{ri}为 X 方向的距离矢量(有正负)，y_{ri}为 Y 方向的距离矢量(有正负)，z_{ri}为 Z 方向的距离矢量(有正负)。

其计算方式为：

$$x_{ri} = (x_{ri\_idx} - (M-1)/2) * dx + x_c$$
$$y_{ri} = (y_{ri\_idx} - (M-1)/2) * (-dx) + y_c$$
$$z_{ri} = (z_{ri\_idx} - (imgSlicCount-1)/2) * dz + z_c$$

此组数据在程序中均有对应，可从 mgfbp.cpp 文件开始按照如下顺序阅读程序：

```
fbp.BackprojectPixelDrivenAndSave((outDir /
mg::FbpClass::config.outputFiles[i]).string().c_str());
```

```
BackprojectPixelDriven_Agent(sinogram_filter, image,sdd_array, sid_array, offcenter_array,
pmatrix_array, u, v, beta, config, z_idx);
```

```
BackprojectPixelDriven_device << <grid, block >> > (sgm_flt, img, u, v, beta, config.shortScan,
config.sgmWidth, config.views, config.sliceCount, config.coneBeam, config.imgDim,
config.imgSliceCount, sdd_array, sid_array, offcenter_array, config.pixelSize,
config.imgSliceThickness, config.xCenter, config.yCenter, config.zCenter, z_idx);
```

```
__global__ void BackprojectPixelDriven_device(float* sgm, float* img, float* u, float* v, float*
beta, bool shortScan, const int N, const int V, const int S, bool coneBeam, const int M, const int
imgS, float* sdd_array, float* sid_array, float* offcenter_array, const float dx, const float dz,
const float xc, const float yc, const float zc, int imgS_idx)
{

    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int row = threadIdx.y + blockDim.y * blockIdx.y;

    float du = u[1] - u[0];
    float dv = v[1] - v[0];


    if (col < M && row < M && imgS_idx <= imgS)
    {

        float x = (col - (M - 1) / 2.0f)*dx + xc;
        float y = ((M - 1) / 2.0f - row)*dx + yc;
...

        for (int slice = imgS_idx; slice < imgS_idx + 1; slice++)
        {
            z = (slice - (float(imgS) - 1.0f) / 2.0f) * dz + zc;
            ...
         }
...
}
```

此函数中 col 与 row 分别对应x_{ri_idx}与y_{ri_idx}。x 与 y 分别对应我们计算的x_{ri}

与y_{ri}。slice 对应z_{ri_idx}，z 对应z_{ri}，imgS 对应 imgSliceCount，其余变量名称与上面完全对应，可对应理解。

理解了所有变量后，接下来我们便可以通过(x_s,y,z)，(X_d,Y_d,Z_d)，(x_{ri},y_{ri},z_{ri})三点位于一条直线上这一性质通过(x_s,y,z)，(x_{ri},y_{ri},z_{ri})来计算出(X_d,Y_d,Z_d)，进而计算出u_{idx}与v_{idx}，来确定此像素应当用哪个下标的探测器像素上的数据来涂抹。公式的推导过程如下：
由三点在一条直线上可得

$$\frac{Y_d - y}{y_{ri} - y} = \frac{X_d - x_s}{x_{ri} - x_s} = \frac{Z_d - z}{z_{ri} - z}$$

代入$(x_s,y,z),X_d$有

$$\frac{Y_d - 0}{y_{ri} - 0} = \frac{-(sdd - sid) - sid}{x_{ri} - sid} = \frac{Z_d - 0}{z_{ri} - 0}$$

可解得

$$Y_d = \frac{sdd}{sid - x_{ri}} * y_{ri}$$

$$Z_d = \frac{sdd}{sid - x_{ri}} * z_{ri}$$

解得Y_d后，由公式

$$Y_d = -\{(u_{idx} - 0) * (du) + u[0]\}$$
$$Z_d = \{(v_{idx} - 0) * (dv) + v[0]\}$$

解得我们的u_{idx},v_{idx}

$$u_{idx} = (-Y_d - u[0])/(du)$$
$$v_{idx} = (Z_d - u[0])/(dv)$$

接下来我们会进行光源与探测器旋转任意角度β后u_{idx}，v_{idx}的计算推理并得出更普遍的u_{idx}，v_{idx}的计算方法，便于我们编程实现。

旋转β角度后使用 Cone beam 的三维图像重建:

在三维 cone beam 反投影时，可知当光源与探测器沿 xoy 坐标轴逆时针旋转β角度时，此时可等价地看做重建图像沿 xoy 所在坐标轴中心顺时针旋转β角度，而 z 轴在旋转过程中旋转角度保持不变。

根据二维几何关系所得结果可知初始坐标系 x 轴与 y 轴与旋转后坐标系 x'轴和 y'轴有如下几何关系（其中 e _x' 表示 x' 轴的基向量，e _y' 表示 y' 轴的基向量）：

$$e\_x' = (\cos\beta, -\sin\beta)$$
$$e\_y' = (\sin\beta, \cos\beta)$$

将 x, y 用 x' , y'表示如下:

$$(x,y) = x'e\_x' + y'e\_y'$$
$$= x' * (\cos\beta, -\sin\beta) + y' * (\sin\beta, \cos\beta)$$

可得：

$$x = x'\cos\beta + y'\sin\beta$$

$$y = -x'\sin\beta + y'\cos\beta$$

根据之前所得几何关系有:

$$Y_d = \frac{sdd}{sid - x_{ri}} * Y_{ri}$$

$$Z_d = \frac{sdd}{sid - x_{ri}} * Z_{ri}$$

将,x_{ri}, y_{ri}代入上式可得:

$$Y_d = \frac{sdd}{sid - (x'\cos\beta + y'\sin\beta)} * (-x'\sin\beta + y'\cos\beta)$$

$$Z_d = \frac{sdd}{sid - (x'\cos\beta + y'\sin\beta)} * Z_{ri}$$

由之前所得Y_d、Z_d及 u0 表达式：

$$Y_d = -\{(u_{idx} - 0) * (du) + u[0]\}$$

$$Z_d = \{(v_{idx} - 0) * (dv) + v[0]\}$$

$$u0 = -Y_d$$

$$v0 = Z_d$$

可得：

$$u_{idx} = (u0 - u[0])/(du)$$

$$v_{idx} = (v0 - v[0])/(dv)$$

注释（代码中变量与上述公式字符对应关系如下）：

z：Z_{ri}

mag_factor：$\frac{sdd}{sid - (x'\cos\beta + y'\sin\beta)}$

v0：$Z_d = \frac{sdd}{sid - (x'\cos\beta + y'\sin\beta)} * Z_{ri}$

k_z：$v_{idx} = \frac{v0 - v[0]}{dv}$

以下代码在函数 BackprojectPixelDriven_device 中如下:

```
// for cone beam ct, we also need to find v0
if (coneBeam && abs(dv) > 0.00001f)
{
    v0 = mag_factor * z;
    // weight for cbct recon
    k_z = floorf((v0 - v[0]) / dv);
```