1.数据表示

本章介绍ITK中表示数据的基本的类。最常用的类为`itk::Image`、`itk::PointSet`和`itk::PointSet`.

1.1. Image

`itk::Image`类遵循了泛型编程的思想，类型与算法分离。ITK支持任意类型的像素和空间维度。

1.1.1. 创建一个Image

本部分的源代码为`Image1.cxx`.

例程演示了怎么样去手动创建一个`itk::Image`类。下面是实例化、声明和创建一个Image类的最小的代码.

首先，头文件必须要包含。

```
#include "itkImage.h"
```

然后我们必须觉得用什么类型表示像素，以及图像的维度。当这两个参数确定后，我们可以就可以实例化`Image`类。现在我们创建一个3D、无符号短字符数据类型的图像.

```
typedef itk::Image< unsigned short, 3 > ImageType;
```

然后就可以调用`New()`操作创建图像，将其分配给`itk::SmartPointer`.

```
ImageType::Pointer image = ImageType::New();
```

在ITK中，图像以一个或多个区域的组合形式存在。一个区域是图像的子集，有可能是系统中其他类处理的图像的一部分。最常见的区域是`LargestPossibleRegion`， 其定义了一个完整的图像。另外一种重要的区域是`BufferedRegion`,是内存中图像的一部分，以及`RequestedRegion`，是滤波器或其他类要求的一部分。

在ITK中，手动创建一幅图像需要对图像进行实例化，并将其与区域描述结合起来。

一个区域由两个类来定义:`itk::Index`和`itk::Size`. 区域的原点由`Index`来定义。区域的延伸，或者大小，由`Size`来定义。当一幅图像被创建，用户有责任定义图像的`size`和图像的起始位置的`Insex`。这两个参数使处理选择的区域变得可能。
`Index`由一个n维数组表示，每一个元素表示图像的最初像素值。

```
ImageType::IndexType start;
start[0] = 0; // first index on X
start[1] = 0; // first index on Y
start[2] = 0; // first index on Z
```

区域大小由一个相同维数的数组表示（利用`itk::Size`类）.数组中的元素是无符号整型，表示图像像素在各个方向上的延伸。

```
ImageType::SizeType size;
size[0] = 200; // size along X
size[1] = 200; // size along Y
size[2] = 200; // size along Z
```

当定义了起始`index`和图像`size`,就可以创建一个`itk::ImageRegion`对象

```
ImageType::RegionType region;
region.SetSize( size );
region.SetIndex( start );
```

最终，这个区域传递给`Image`类来定义延伸和原点。`SetRegions`方法同时设置了`LargestPossibleRegion`，`BufferedRegion`,和`RequestedRegion`.注意：没有进行任何操作给图像像素数据分配内存。调用`Allocate()`方法来分配内存。`Allocate`不需要任何变量，直到直到分配了足够的内存。

```
image->SetRegions( region );
image->Allocate();
```

实际上很少直接地去分配和初始化一幅图像。通常图像从一个源文件读取。下面的例子说明如何从一个文件读取图像。

1.1.2. 从文件读取图像

源代码为`Image2.cxx`.
首要任务是包含头`itk::ImageFileReader`类的头文件。

```
#include "itkImageFileReader.h"
```

Then, the image type should be defined by specifying the type used to represent pixels and the dimensions of the image.
然后，图像类型根据指定表示像素和维数的类型来定义图像。

```
typedef unsigned char PixelType;
const unsigned int Dimension = 3;
typedef itk::Image< PixelType, Dimension > ImageType;
```

利用图像类型，可以来实例化一个图像读取类。图像类型被用作模板参数，来表示加载到内存中的数据。这个类型不需要和文件中存储的类型完全一致。使用了一个基于C-style的一个转换器，用来表示硬盘上的数据的类型应该精确的表示出来。用户不需要对像素数据做任何转换，除了将文件的像素类型转换为`ImageFileReader`的像素类型。下面展示`ImageFileReader`类型的一个典型实例。

```
typedef itk::ImageFileReader< ImageType > ReaderType;
```

现在`reader`类型可以用来创建一个`reader`对象。一个`itk::SmartPointer`用来接收新创建的`ewader`的引用。调用`New()`方法创建一个图像`reader`的实例。

```
ReaderType::Pointer reader = ReaderType::New();
```

最小信息需求是加载到内存中的文件名，可以通过`SetFileName()`方法来实现。文件格式由文件扩展名来推断得知。用户同样需要使用`itk::ImageIOBase`类明确地指明数据格式。

```
const char * filename = argv[1];
reader->SetFileName( filename );
```

`Reader`被称为管道流对象，对应于管道更新需求，和初始化管道流。管道更新机制保证用`reader`仅在得到了一个数据请求，但是还没有读取数据时执行。在当前的例子中，我们明确地调用`Update()`方法，因为`reader`的输出没有连接到其他的滤波器。在正常的应用中，`reader`的输出被连接到一个滤波器的输入，滤波器上的更新调用引发一个`reader`的一个更新。下面的例子说明`reader`上的一个调用明确地更新。

```
reader->Update();
```

使用`GetOutput()`方法访问新读取的图像。这个方法也可以在更新需求之前被调用。直到`reader`实际执行之前，即使图像是空的，这个对图像的引用仍然是有效的。

```
ImageType::Pointer image = reader->GetOutput();
```

在`reader`执行之前，任何的访问图像将获得一个没有像素数据的图片。正如一个图像没有被合适的初始化将产生一个程序崩溃一样。

1.1.3. 访问像素数据

源代码为`Image3.cxx`.

这个例子展示了`SetPixel()`和`GetPixel()`方法的使用。这两种方法可以直接访问像素中的数据。这两种方法较为缓慢，在高性能的需求环境中不宜使用。图像迭代器是一个有效的访问图像像素数据的机制（有关图像迭代器的知识请见第6章）。

每个像素的位置由一个特定的`index`来区分。一个`index`是一个整数数组，定义了图像每一维中像素的位置。`Index`的类型由图像自动定义，可以使用作用与操作符`itk::Index`进行访问。数组的长度与图像的维数相匹配。

接下来的代码展示了`index`变量的声明，并对其成员值进行分配。并不使用`SmartPointer`访问`index`. 因为`index`是一个不与任何对象共享的轻量对象。　　与使用`SmartPointer`机制相比较，对这些小对象产生多种拷贝更加有效。

下面的代码是对`index`类型进行实例化声明并进行初始化，使其与图像的像素位置进行关联。

```
const ImageType::IndexType pixelIndex = {{27,29,37}}; // Position of {X,Y,Z}
```

用`index`定义了一个像素位置后，就可以访问图像中像素的内容。`GetPixel()`方法允许我们得到一个像素值。

```
ImageType::PixelType pixelValue = image->GetPixel( pixelIndex );
```

`SetPixel()`方法允许我们设置像素值。

```
image->SetPixel( pixelIndex, pixelValue+1 );
```

请注意`GetPixel()`方法使用拷贝值来返回数据，此方法不能改变像素值。

请记住`SetPixel()`和`GetPixel()`都是低效率的，只能用来调试或者支持交互，例如点击鼠标查询像素值。

1.1.4. 定义原点和间距

源代码`Image4.cxx`.

尽管ITK可以用于一般的图像处理任务，但是这个工具的最主要的任务是处理医学图像数据。因此，必须增加额外的信息。关于一些坐标系中同像素与图像位置之间之间的物理空间信息非常重要。

图像原点，体素方向，和间距对许多应用来说是非常重要的。例如配准，需要在物理坐标系中进行。不合适定义的间距、方向和原点将导致不一致的结果。没有空间信息的医学图像不能被用于医学诊断、特征提取、图像分析、辅助放疗治疗和图像手术导航。换句话说，缺乏空间信息的医学图像不仅仅是无用的，而且是危险的。

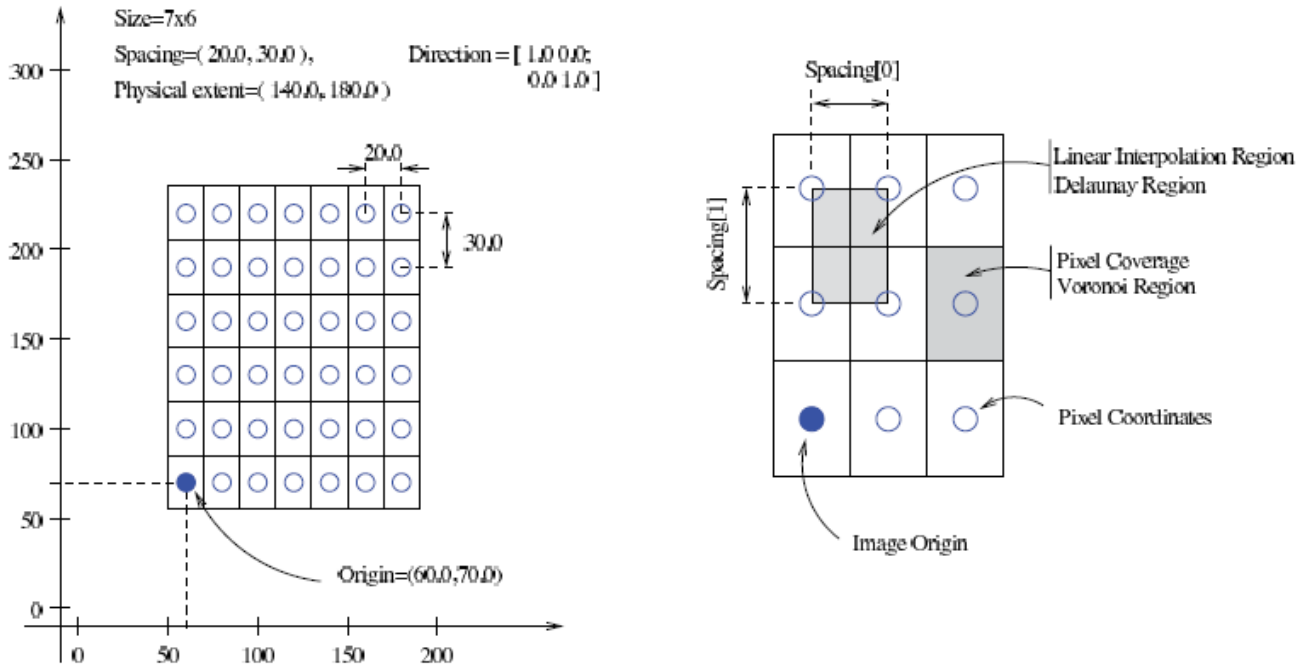

Figure 4.1: Geometrical concepts associated with the ITK image.

图4.1阐述了与`itk::Image.`相关的主要几何概念。在图中，圆圈表示像素的中心，像素值为像素中心的狄拉克 δ 函数。像素间距为像素中心之间的距离，在不同的维度上可以不同。图像原点为第一个像素的坐标。对于这个简单的例子，像素网格在物理空间中完美的分配，图像方向是一个恒等映射。如果像素网格按照物理空间进行旋转，图像方向将包括一个旋转矩阵。

一个像素是环绕在像素中心的一个矩形区域的一个有效的值。可以视为图像网格的`Voronoi`区域，如右图所示。像素值的线性内插法在`Delaunay`区域内执行，其拐点为像素中心。

图像间距在`FixedArray`中表示，其大小与图像的维度相匹配。为了可以手动设定图像的间距，必须创建一个相应的数组。数组的元素应该被相邻像素的中心的距离初始化。下面的代码展示了在`itk::Image`类中处理间距和原点的方法。

```
ImageType::SpacingType spacing;
// Units (e.g., mm, inches, etc.) are defined by the application.
spacing[0] = 0.33; // spacing along X
spacing[1] = 0.33; // spacing along Y
spacing[2] = 1.20; // spacing along Z
```

利用`SetSpacing()`方法将数组指向图像。

```
image->SetSpacing( spacing );
```

使用`GetSpacing()`方法可以从图像中得到间距信息，该方法返回一个`FixedArray`的引用。返回对象可以用于读取数组的内容。请注意：关键字`const`表示数组不可以被修改。

```
const ImageType::SpacingType& sp = image->GetSpacing();
std::cout << "Spacing = ";
std::cout << sp[0] << ", " << sp[1] << ", " << sp[2] << std::endl;
```

图像原点的处理方法与间距类似。首先分配好维度。任何一个组成部分都可以分配为原点的坐标。这些坐标对应于图像的第一个像素，可以使用任意的参考系统。用户必须保证同一个应用下的图像都是使用一致的参考系统，这对图像配准极其重要。

下面的代码阐述了初始化图像原点的变量的创建和分配。

```
// coordinates of the center of the first pixel in N-D
ImageType::PointType newOrigin;
newOrigin.Fill(0.0);
image->SetOrigin( newOrigin );
```

采用`GetOrigin()`方法可以从图像中获取原点，产生一个点的引用。这个引用可以用来读取数组的内容。

```
const ImageType::PointType & origin = image->GetOrigin();
std::cout << "Origin = ";
std::cout << origin[0] << ", "
<< origin[1] << ", "
<< origin[2] << std::endl;
```

图像方向矩阵表示图像和物理空间坐标系之前的方向关系。图像方向矩阵是一个正则化的矩阵，描述图像`index`值和旋转角度的可能的排列。图像方向是一个$N \times N$的矩阵，其中N是图像的维度。一个确定的图像方向表明1st, 2nd和3rd 的`index`元素的增加值对应于1st, 2nd and 3rd的物理空间轴的增加。

下面的代码阐述了用来初始化图像方向的变量的创建和分配。

```
// coordinates of the center of the first pixel in N-D
ImageType::DirectionType direction;
direction.SetIdentity();
image->SetDirection( direction );
```

使用方法`GetDirection()`可以从图像中获取方向，返回一个矩阵的引用。这个引用可用于读取数组的内容。

```
const ImageType::DirectionType& direct = image->GetDirection();
std::cout << "Direction = " << std::endl;
std::cout << direct << std::endl;
```

一旦间距、原点和方向被初始化，图像将精确地将像素索引映射到物理空间坐标。下面的例子展示了物理空间中的一个点怎么样被映射到一个图像`index`，为了读取最近像素的内容。

首先，一个`itk::Point`类型必须被声明。`point`类型在用来表示坐标和空间维度之上模板化。在这种特殊的情况下，`point`的维度必须匹配图像的维度。

```
typedef itk::Point< double, ImageType::ImageDimension > PointType;
```

`itk::Point`类型与`itk::Index`类型相似，是一个又小又简单的对象。这意味着不需要使用`itk::SmartPointer`。像其他C++类一样，`point`被简单实例化声明。一旦一个`point`被声明，其元素可以利用传统的数组符号进行访问。特别地，`[]`操作符变得有效。出于效率原因，在访问特定的`point`元素时不执行边界检查。用户必须保证`index`在有效范围$\{0, Dimension - 1\}$内.

```
PointType point;
point[0] = 1.45; // x coordinate
point[1] = 7.21; // y coordinate
point[2] = 9.28; // z coordinate
```

图像使用当前的原点和间距值把`point`映射到`index`. 必须提供一个`index`对象来接收映射的结果。使用图像类型中定义的`IndexType`对`index`对象进行实例化。

```
ImageType::IndexType pixelIndex;
```

`TransformPhysicalPointToIndex()`方法计算与提供的`point`的最近的像素`index`. 这个方法检查这个`index`是否包含在一个当前的缓冲像素数据中。这个方法返回一个布尔类型数据，指示结果`index`是否落在缓冲区域。输出`index`不应该被使用，当返回的数据是错误的。

下面的例子阐述了`point`到`index`的 映射，以及使用像素`index`去访问像素数据。

```
const bool isInside =
image->TransformPhysicalPointToIndex( point, pixelIndex );
if ( isInside )
{
ImageType::PixelType pixelValue = image->GetPixel( pixelIndex );
pixelValue += 5;
image->SetPixel( pixelIndex, pixelValue );
}
```

请记住`GetPixel()`和`SetPixel()`都是低效率的访问像素数据的方法。当需要访问大量像素数据时，应该使用图像迭代器。

下面的例子阐述了像素`index`位置和其相应的物理`point`表示的数学关系。

让我们想象存在一个图形用户界面，用户通过鼠标手动的选择位于左眼位置的体素`index`. 我们需要将这个`index`位置转换为物理位置，从而激光引导手术可以精确地执行。`TransformIndexToPhysicalPoint`方法可以用在这个地方。

```
const ImageType::IndexType LeftEyeIndex = GetIndexFromMouseClick();
ImageType::PointType LeftEyePoint;
image->TransformIndexToPhysicalPoint(LeftEyeIndex,LeftEyePoint);
```

对于一个给定的`index` $I_{3\times1}$, 物理位置$P_{3\times1}$计算如下：

$$P_{3\times1} = O_{3\times1} + D_{3\times3} * diag(S_{3\times1})_{3\times3} * I_{3\times1}$$

其中，D是一个标准正交的方向余弦矩阵，S是图像距离对角矩阵。

matlab语法：

```
% Non-identity Spacing and Direction
spacing=diag( [0.9375, 0.9375, 1.5] );
direction=[0.998189, 0.0569345, -0.0194113;
0.0194429, -7.38061e-08, 0.999811;
0.0569237, -0.998378, -0.00110704];
point = origin + direction * spacing * LeftEyeIndex
```

一个相应的C/C++的数学扩展为：

```
typedef itk::Matrix<double, Dimension, Dimension> MatrixType;
MatrixType SpacingMatrix;
SpacingMatrix.Fill( 0.0F );
const ImageType::SpacingType & ImageSpacing = image->GetSpacing();
SpacingMatrix( 0,0 ) = ImageSpacing[0];
SpacingMatrix( 1,1 ) = ImageSpacing[1];
SpacingMatrix( 2,2 ) = ImageSpacing[2];
const ImageType::DirectionType & ImageDirectionCosines =
image->GetDirection();
const ImageType::PointType &ImageOrigin = image->GetOrigin();
typedef itk::Vector< double, Dimension > VectorType;
VectorType LeftEyeIndexVector;
LeftEyeIndexVector[0]= LeftEyeIndex[0];
LeftEyeIndexVector[1]= LeftEyeIndex[1];
LeftEyeIndexVector[2]= LeftEyeIndex[2];
ImageType::PointType LeftEyePointByHand =
ImageOrigin + ImageDirectionCosines * SpacingMatrix * LeftEyeIndexVector;
```

1.1.5 RGB图像

The RGB space has been constructed as a representation of a physiological response to light by the three types of cones in the human eye. RGB is not a Vector space. For example, negative numbers are not appropriate in a color space because they will be the equivalent of “negative stimulation” on the human eye. In the context of colorimetry, negative color values are used as an artificial construct for color comparison in the sense that