医学图像分割与配准

（①ITK 初步分册）

周振环　王安明　王京阳　赵　明　著

电子科技大学出版社

图书在版编目（CIP）数据

医学图像分割与配准 / 周振环等著. —成都：电子科技大学出版社，2007.6

ISBN 978–7–81114–571–7

Ⅰ. 医… Ⅱ. 周… Ⅲ. 医学图像—图像数字化处理 Ⅳ. R445–39

中国版本图书馆 CIP 数据核字（2007）第 087411 号

医学图像分割与配准

（①ITK 初步分册）

周振环　王安明　王京阳　赵　明　著

出　　版： 电子科技大学出版社（成都市一环路东一段 159 号电子信息产业大厦　邮编：610051）

策划编辑： 朱　丹

责任编辑： 张　鹏

主　　页： www.uestcp.com.cn

电子邮件： uestcp@uestcp.com.cn

发　　行： 新华书店经销

印　　刷： 成都金龙印务有限责任公司

成品尺寸： 185mm×260mm　　**印张** 33.5　　**字数** 820 千字

版　　次： 2007 年 6 月第一版

印　　次： 2007 年 6 月第一次印刷

书　　号： ISBN 978-7-81114-571-7

定　　价： 98.00 元（共两册）

前　言

在 1999 年，由美国国家卫生院（NIH）下属的国力医学图书馆（NLM）发起了一个投标活动，出资赞助开发一个开放源码的分割与配准的算法研究平台。ITK 的 NIH/NLM 的工程负责人 Terry Yoo 博士带领 6 家单位合作开发，这 6 家单位包括 GE Corporate R&D、Kitware、Inc.和 MathSoft （现在公司改名为 Insightful）三个商业公司和 University of North Carolina（UNC）、University of Tennessee（UT）（Ross Whitaker 随后迁往 University of Utah）和 University of Pennsylvania （UPenn）三所大学。

2002 年 ITK 官方首次发行 ITK 版本。这本书适用于 ITK2.4 更新版本。ITK 是一个开放源码、面向对象的软件系统，提供一个医学图像处理、图像分割与配准的算法平台。虽然 ITK 结构庞大复杂，但是一旦你了解它的面向对象和执行基本方法，就可以灵活应用。这本软件指南的目的正是帮你了解这些方法以及这个平台中的主要算法和数据表达。书中已经提供了一些实例的使用资料，你在阅读本书时便可以编译运行。

鉴于 ITK 是一个庞大的系统，因此本书不可能完全介绍所有的 ITK 对象和方法。本书将尽最大能力指导你了解重要的系统概念，并尽快尽好地指导你学习。ITK 是一个开放源码的软件系统，这就意味着 ITK 用户和开发团体可以方便地对软件进行软件的开发和改进。

这本软件指南分为两部分，每部分又包括几个章节。第一部分是 ITK 的基本情况介绍。第一章和接下来的两章介绍如何在你的计算机上安装 ITK，包括安装预编译库和运行以及从源代码编译软件。第一部分同样也介绍了一些基本的系统概念，如：系统结构概述、如何使用 C++、Tcl 和 Python 编程语言建立应用程序。第二部分从用户角度来介绍软件，提供了大量实例描述系统的主要特征。

ITK 用户可以明显地分为两类。第一类人是使用 C++创建新类的开发者，另一类人是用已有的 C++类进行应用的使用者。类开发者必须非常精通 C++。如果他们要对 ITK 进行扩展和改进，就必须非常熟悉 ITK 的内部结构和设计。作为 ITK 的使用者，你必须了解 ITK 类和外部界面接口以及它们之间的关系。

学会使用 ITK 的关键就是熟悉各个对象的调色板和它们的方式。如果你是一位 ITK 的初学者，那么就从安装软件开始学起吧。如果你是一位类开发者，就需要安装源代码并编译。用户仅需预编译和执行程序。我们推荐你通过学习实例来了解系统。类开发者可学习源代码。先学习第三章提供的系统主要概念的综述，然后回顾第二部分的实例。你也可以编译和运行这些实例。这些实例的源代码也可以在目录 Insight/Examples 中找到（参见文件 Insight/Examples/README.txt 中包含的各种子目录里对这些实例的介绍）。在目录 Insight/Testing/Code 中的源代码分布里也可以找到许多测试，这些测试大部分是测试代码。然而它们非常有助于了解 ITK 中类的使用方式，尤其是它们尽可能地表达出了类的功能。

ITK 可以免费从以下网站下载 http://www.itk.org/HTML/Download.php。你可以得到一个稳定的版本或通过 CVS 得到比较新的版本。发行的版本比较稳定可靠但可能缺少研发平坦最新的特性功能。CVS 版本将含有最新的功能但有一定的不稳定性。

首先阅读 GettingStarted.txt 了解下载和安装进程。然后选择一个适合你系统的版本。有.zip 和.tgz 两个类型的文件供选择。第一种更适合于 MS-Windows 系统，而第二种是 UNIX 系统的最佳版本。一旦你解压缩文件包，就将在你的电脑上生成一个称为 insight 的目录，

你就可以按照书中描述的那样开始安装配置过程。

强烈建议你加入用户 mailing list。这是获得指导和使用帮助的主要来源。你可以在以下网站订阅拥护列表：http://www.itk.org/HTML/MailingLists.htm。用户 mailing list 同样也是表达你自己关于研发平台的观点和开发者了解有用的、期望的甚至是不必要的特征的最主要的机制。利用反馈的信息可以使 ITK 的开发者创立一个开放源码的 ITK 社区。

为了开始你的 ITK 之旅，首先你需要了解 ITK 的软件和目录结构。即使你安装了预编译库，这也将对你通过基本编码寻找实例、程序编码及文件有很大的帮助。

ITK 分为几个不同的模块或 CVS 校验区。官方或光盘版本有三大主要模块：Insight, InsightDocuments 和 InsightApplications 模块。Insight 模块包括有源代码、实例及应用；InsightDocuments 模块中包括有文件、指南和 ITK 的设计行销的有关材料；InsightApplications 模块中包含有 ITK（与其他如 VTK、Qt 和 FLTK 系统）的综合复杂应用。通常只需工作在 Insight 模块，开发者、培训课程者和许多具体细节的设计文件除外。只有当 Insight 模块的程序可以正常运行时才能下载和编译 InsightApplications 模块。

Insight 模块包含以下子目录：

- Insight/Auxiliary—工具包到 ITK 的界面代码。
- Insight/Code—软件的核心。主要源代码的位置。
- Insight/Documentation—用户开始 ITK 获得文献的简洁的子集。
- Insight/Examples—一系列样例、这本指南使用的和阐述重要 ITK 概念的文献样例。
- Insight/Testing—用来测试 ITK 的大量小程序。
- Insight/Code/Common—核心类、大量的定义、声明和 ITK 中其他重要的软件结构。
- Insight/Code/Numerics—数学库和支持类。
- Insight/Code/BasicFilters—基本的图像处理滤波器。
- Insight/Code/IO—支持读取和写数据的类。
- Insight/Code/Algorithms—大部分分割和配准算法的位置。
- Insight/Code/SpatialObject—使用空间关系表达和组织数据的类。
- Insight/Code/Patented—这里提供申请专利的算法，使用这些算法进行应用需要一个专利号。
- Insight/Code/Local—开发者使用的空目录，用户实验的新代码。

InsightDocuments 模块包含以下子目录：

- InsightDocuments/CourseWare—与 ITK 教学相关的材料。
- InsightDocuments/Developer—覆盖 ITK 设计和创建的历史文献，包括进程报告和设计文献。
- InsightDocuments/Web—http://www.itk.org 上找到的网站工具包的 HTML 源和其他材料。

InsightApplications 模块包含有大量相关的复杂的 ITK 使用实例。网页 http://www.itk.org/HTML/Applications.htm 上有相关的描述。

ITK 支持可视化人体工程（VHP）及它的相关数据。这些数据可以在国立医学图书馆的网站 http://www.nlm.nih.gov/research/visible/visible_human.html 上找到。

基金项目：2005 年粤港关键领域重点突破项目（2005A11304003）、深圳市科技计划项目（05KJcd004）联合资助。

目　　录

第 1 部分

第 1 部分

医学图像分割与配准——ITK 初步

第一章　欢　　迎

欢迎阅读医学影像分割与配准算法的研发平台（ITK）软件指南。这本书适用于 ITK 2.4 更新版本。

ITK 是一个开放源码、面向对象的软件系统，提供一个医学图像处理、图像分割与配准的算法平台。虽然 ITK 结构庞大、复杂，一旦你了解它的面向对象和执行基本方法，就可以灵活应用。这本软件指南的目的正是帮你了解这些方法以及这个平台中的主要算法和数据表达。书中已经提供了一些实例的使用资料，你在阅读本书时便可以编译运行。

鉴于 ITK 是一个庞大的系统，因此本书不可能完全介绍所有的 ITK 对象和方法。本书将尽最大能力指导你了解重要的系统概念，并尽快、尽好地指导你学习。我们建议你在精通了基本知识后，就可以充分利用尽可能多的资源，包括 Doxygen 文献网页（http://www.itk.org/HTML/Documentation.htm）和 ITK 用户团体（参考 1.5 节）。

ITK 是一个开放源码的软件系统，这就意味着 ITK 用户和开发团体就可以方便地对软件进行开发和改进。用户和开发者可通过程序错误报告、调试、测试、新类以及其他反馈来对 ITK 进行应用开发。请贡献你的想法（首选方法是通过 ITK 用户邮件发送列表 mailing list，开发者邮件发送列表亦可）。

1.1　团体机构

这本软件指南分为三部分，每部分又包括几个章节。第一部分是 ITK 的基本情况介绍。本章和接下来的两章将介绍如何在你的计算机上安装 ITK，包括预编译库的安装和运行以及从源代码编译软件。第一部分同样也介绍了一些基本的系统概念，如：系统结构概述、如何使用 C++、Tcl 和 Python 编程语言建立应用程序。第二部分从用户角度来介绍软件，提供了大量实例来描述系统的主要特征。第三部分主要针对 ITK 开发者。第三部分介绍了如何创建新类、扩展系统和各种 Windows GUI 系统界面接口。

1.2　如何学习 ITK

ITK 用户可以明显地分为两类。第一类用户是使用 C++创建新类的开发者，另一类用户是使用已有的 C++类进行应用的使用者。类开发者必须非常精通 C++。如果他们要对 ITK 进行扩展和改进，就必须非常熟悉 ITK 的内部结构和设计（内容在第三部分）。由于编译中包含了转译 Tcl 和 Python 语言的 C++类库，因此用户没必要必须使用 C++语言。作为 ITK 的使用者，你必须了解 ITK 类和外部界面接口以及它们之间的关系。

学会使用 ITK 的关键就是熟悉各个对象的调色板和它们的方式。如果你是一位 ITK 的初学者，那么就从安装软件开始学起吧。如果你是一位类开发者，就需要安装源代码并编译。用户仅需预编译和执行程序。我们推荐你通过学习实例来了解系统。类开发者可学习源代码。先学习第三章提供的系统主要概念的综述，然后回顾第二部分的实例。你也可以编译和运行

这些实例。这些实例的源代码也可以在目录 Insight/Examples 中找到(参见文件 Insight/Examples/README.txt 中包含的各种子目录里对这些实例的介绍)。在目录 Insight/Testing/Code 中的源代码分布里也可以找到许多测试，这些测试大部分是测试代码。然而它们非常有助于了解 ITK 中类的使用方式，尤其是它们尽可能地表达出了类的功能。

1.3 软件组织

接下来的章节介绍目录内容、各个目录中的软件功能的总结以及文献和数据的地址。

1.3.1 获取软件

有三种不同的途径可以得到 ITK 源代码（参见 1.4 章节）。

(1)ITK 网站上定期公布官方版本；

(2)由光盘 CD-ROM 获取；

(3)直接由 CVS 代码库获取。

官方每年都会在 ITK 网站上和邮件列表中提供几次软件版本供下载，但并非都是最新最好的版本。总的来说，在网站上下载的和由光盘获得的都是一样的，只不过光盘会额外多一些源代码数据。CVS 提供的是及时的最新的版本，但与官方提供的版本相比稳定性不高，经常出现堵塞无法编译，甚至产生错误结论的现象。

本书将以官方发行的 2.4 版为准（可在 ITK 网站上下载）。对初学者我们建议使用官方发行的版本，它比较稳定可靠，比从 CVS 得到的源代码能更好地进行测试。对老用户，在对 ITK 应用有一些经验以后，可以从 CVS 开始工作。ITK 主要使用开放源码的 DART regression 测试系统来进行测试(http://public.kitware.com/dashboard.php)。在更新你的 CVS 库之前，如果 dashboard 是绿色的，表示代码稳定；如果不是绿色的那么你的软件更新将很可能不稳定（参考 14.2 节可以得到更多关于 ITK 中 dashboard 的信息）。

1.4 ITK 下载

ITK 可以免费从以下网站下载：http://www.itk.org/HTML/Download.php 。

为了更好地使用 ITK 的各项应用，在下载软件之前先填写一个表格，这个表格提供了一些有助于开发者得到一些灵感的信息和使用者的使用技巧，它还有助于研究部门指定研究要求。

填完这个表格就会出现一个网页，网页上有得到这个软件的两个选项（完成这一页以后你就可以到达下载页面）。你可以选择得到一个发行的稳定的版本或通过 CVS 得到一个比较新的版本。发行的版本比较稳定可靠，但可能缺少研发平台最新的特性功能。CVS 版本含有最新的功能但不稳定。接下来的章节我们将详细讨论这两种方式。

1.4.1 下载发行的版本

首先阅读 GettingStarted.txt 了解下载和安装进程。然后选择一个适合你系统的版本。有.zip 和.tgz 两个类型的文件供选择。第一种更适合于 MS-Windows 系统，而第二种是 UNIX 系统的最佳版本。

一旦你解压缩文件包，就将在你的电脑上生成一个称为 insight 的目录，你可以按照第 2.1.1 小节描述的那样开始安装配置过程。

1.4.2 从 CVS 下载

CVS 是 Concurrent Version System(并行版本系统)的缩写，是一个控制软件版本的工具。原则上只有开发者才用得到 CVS。因此这里我们假设你了解 CVS 并知道如何使用它。了解更多 CVS 的信息请参见 14.1 节（注意：请确保只有在 ITK Quality Dashboard 表达的代码是稳定的时候才从 CVS 访问。参看 14.2 节可以了解更多关于 Quality Dashboard 的信息）。

使用以下命令经过 CVS 来访问 ITK(在 UNIX 和 Cygwin 系统下)：

```
cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight login
(respond with password "insight")
cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight co Insight
```

这将触发软件下载进入一个 Insight 目录。任何时候你需要更新你的版本，都可以打入以下命令来改变 Insight 目录：

```
cvs update -d -P
```

一旦获得软件你就可以进行设置和编译(见 2.1.1 小节)。不过我们建议你首先加入 mailing list 并阅读以下关于软件机构描述的章节。

1.4.3 加入 Mailing List

强烈建议你加入用户 Mailing List，这是获得指导和使用帮助的主要来源。你可以在以下网站订阅用户列表：

http://www.itk.org/HTML/MailingLists.htm

用户 Mlist List 同样也是表达你自己关于研发平台的观点和开发者了解有用的、期望的甚至是不必要的特征的最主要的机制。利用用户反馈的信息可以使 ITK 的开发者创立一个开放源码的 ITK 社区。

1.4.4 目录结构

为了开始你的 ITK 之旅，首先你需要了解 ITK 的软件和目录结构。即使你安装了预编译库，这也将对你通过基本编码寻找实例、程序编码及文件有很大的帮助。

ITK 分为几个不同的模块或 CVS 校验区。官方或光盘版本有三大主要模块：Insight、InsightDocuments 和 InsightApplications 模块。Insight 模块包括有源代码、实例及应用；InsightDocuments 模块中包括文件、指南、ITK 设计、市场的相关材料；InsightApplications

模块中包含有 ITK（与其他如 VTK、Qt 和 FLTK 系统）的综合复杂应用。通常只需工作在 Insight 模块，开发者、培训课程者和许多设计文件的细节查找者除外。只有当 Insight 模块的程序可以正常运行才能下载和编译 InsightApplications 模块。

Insight 模块包含以下子目录：

- Insight/Auxiliary——工具包到 ITK 的界面代码。
- Insight/Code——软件的核心。主要源代码的位置。
- Insight/Documentation——用户开始 ITK 获得文献的简洁的子集。
- Insight/Examples——一系列样例、这本指南使用的和阐述重要 ITK 概念的文献样例。
- Insight/Testing——用来测试 ITK 的大量小程序。这些例子是最低限度的文献，但可以帮助示例许多系统概念。DART 使用这些测试来生成 ITK Quality Dashboard（见第 14.2 节）。
- Insight/Utilities——ITK 源代码的支持软件。例如 DART 和 Doxygen 支持, 如 png 和 zlib 库一样。
- Insight/Validation——一系列包含用来产生结果的源代码的 validation 案例学习。
- Insight/Wrapping——支持 CABLE 封装工具。ITK 使用 CABLE 来创建 C++库和许多编译语言（支持当前的 Tcl 和 Python 语言）之间互译的界面。

在 Insight/Code 中可以找到的源代码目录结构对了解隐藏在 Insight/Code 中的其他目录结构（如 Testing 和 Wrapping 目录）是非常重要的。

- Insight/Code/Common——核心类、大量的定义、声明和 ITK 中其他重要的软件结构。
- Insight/Code/Numerics——数学库和支持类（注意：ITK 的数学库是基于 VXL/VNL 的软件包 http://vxl.sourceforge.net）。
- Insight/Code/BasicFilters——基本的图像处理滤波器。
- Insight/Code/IO——支持读写数据的类。
- Insight/Code/Algorithms——大部分分割和配准算法的位置。
- Insight/Code/SpatialObject——使用空间关系表达和组织数据的类（例如：脊椎骨和腿骨的组织结构等）。
- Insight/Code/Patented——这里提供申请专利的算法，使用这些算法进行应用需要一个专利号。
- Insight/Code/Local——开发者使用的空目录，用户实验的新代码。

InsightDocuments 模块包含以下子目录：

- InsightDocuments/CourseWare——与 ITK 教学相关的材料。
- InsightDocuments/Developer——覆盖 ITK 设计和创建的历史文献，包括进程报告和设计文献。
- InsightDocuments/Latex——像其他文献一样产生这个指南的 LATEX 风格。
- InsightDocuments/Marketing——用于简单描述 ITK 的市场材料和文献。
- InsightDocuments/Papers——与 ITK 中使用的大量的算法、数据表达和软件工具相关的论文。
- InsightDocuments/SoftwareGuide—— 用来创建这个指南的 LATEX 文件(注意在 Insight/Examples 中找到的文件用来和 LATEX 文件关联)。
- InsightDocuments/Validation——使用 ITK 的 validation 案例学习。

• InsightDocuments/Web ——http://www.itk.org 上找到的网站工具包的 HTML 源和其他材料。

与 Insight 模块相类似，也可以使用命令经 CVS 来访问 InsightDocuments 模块（在 UNIX 和 Cygwin 系统下），命令如下：

cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight co InsightDocuments

InsightApplications 模块包含有大量相关的、复杂的 ITK 使用实例。网页 http://www.itk.org/HTML/Applications.htm 上有相关的描述。其中有些应用需要使用 Qt 和 FLTK 之类的 GUI 研发平台或者如 VTK 之类的研发平台（VTK 网站 http://www.vtk.org）。只有当 Insight 模块创建成功后才能进行模块的编译和创建。

与 Insight 模块和 InsightDocuments 模块相类似，也可以使用命令经 CVS 来访问 InsightApplications 模块（在 UNIX 和 Cygwin 系统下），命令如下：

cvs -d:pserver:anonymous@www.itk.org:/cvsroot/Insight \co InsightApplications

1.4.5 文献

除了这个文本之外，还有其他一些文件需要掌握。

Doxygen 文献：Doxygen 文献是 ITK 工作必需的资源。它全面详细地描述了系统中的各个类和方法。这个文件同时包括了遗传和协作图表、重要的创新列表和数据成员，和其他类和源代码都息息相关。这个文献有光盘版，也可以在网站 http://www.itk.org 上找到。使用时务必确认你的版本是源代码所对应的版本。

头文件：每个 ITK 类都是使用一个.h 和.cxx/.txx 文件来执行的（txx 文件用来模板化类）。所有的方法都可以在.h 的文件中找到，同时还提供了一个找到文件的特殊快捷方式（事实上，Doxygen 使用头文件来产生输出）。

1.4.6 数据

ITK 支持可视化人体工程（VHP）及其相关数据。这些数据可以在国立医学图书馆的网站 http://www.nlm.nih.gov/research/visible/visible_human.html 上找到。

其他数据可以在 ITK 的以下任一网站上获得：

http://www.itk.org/HTML/Data.htm

ftp://public.kitware.com/pub/itk/Data/

1.5 ITK 团体和服务

ITK 最初是作为一个协作团体而创建的。现在正在发掘它在研究、教育和商业方面的应用。可以有很多种途径参与到团体中。

• 用户可以对 API 系统的错误和瑕疵进行报告并递交申请。无疑最好的方法是通过用户 Mailing List。

• 开发者可以开发新类和提高已有类的功能。开发者可以通过申请加入 ITK 开发者 Mailing List。申请时发送邮件到 will.schroeder@kitware.com。作为开发者，你必须同时具备

一定的信誉度和能力。你可以从递交邮件到 ITK 用户 Mailing List 开始做起。

• 鼓励与软件社团成员进行合作研究。NIH 和 NLM 在今后几年中只能提供有限的发现，你可以使用 ITK 进行协作工作。

• 对于 ITK 在商业应用的开发和发展，可在 http://www.kitware.com 的软件工具箱中找到客服和参考资料。这个软件工具箱同时也提供一些小的 ITK 课件。

• 教育工作者可以使用 ITK 作为课程传授。有许多这方面的材料正在被发行，例如日常讲座、会议文件和长学期的毕业课程。在网站上和 InsightDocuments/CourseWare 目录中可以得到更多的信息。

1.6 ITK 的主要历史

1999 年，由美国国家卫生院（NIH）下属的国立医学图书馆（NLM）发起了一个投标活动，出资赞助开发一个开放源码的分割与配准的算法研究平台。ITK 的 NIH/NLM 的工程负责人 Terry Yoo 博士带领六家单位合作开发，这六家单位包括 GE Corporate R&D、Kitware、Inc. and MathSoft（现在公司改名为 Insightful）三个商业公司和 University of North Carolina（UNC）、University of Tennessee（UT）（Ross Whitaker 随后迁往 University of Utah）和 University of Pennsylvania（Upenn）三所大学。参与开发的主要成员有：GE CRD 的 Bill Lorensen、Kitware 的 Will Schroeder、Insightful 的 Vikram Chalana、UNC 的 Stephen Aylward 和 Luis Ibanez（Luis 现在在 Kitware）、在 UT 的 Ross Whitaker 和 Josh Cates（现在两个人都在 Utah）以及 UPenn 的 Dimitri Metaxas（现在在 Rutgers）。另外还有几位转承包商社团包括：Brigham&Women's Hospital 的 Peter Raitu、Columbia University 的 Celina Imielinska 和 Pat Molholt、UPenn's Grasp Lab 的 Jim Gee 以及 University of Pittsburgh 的 George Stetten。

2002 年 ITK 官方首次发行 ITK 版本。另外，国立医学图书馆授予几个机构共计 13 份合同来扩展 ITK 的功能。NLM 通过在 2003 年对 ITK 的开发发展，有望今后开发出新的应用和维护支持。若你有意进行潜力开发，建议你咨询国立医学图书馆的 Terry Yoo 博士得到更多相关信息。

第二章　安　　装

本章将介绍 ITK 的安装过程。ITK 本身只是一个算法研究平台，所以安装以后并不能运行任何应用。但你可以使用 ITK 进行自己的开发应用。ITK 除了提供给你一个平台以外，同时也提供了大量的文件和实例来帮助你了解 ITK 中的概念和如何在你的工作中使用 ITK。

有些实例需要应用到第三方数据库，你必须下载这些数据库。对 ITK 初始安装使用者，你可能只安装 ITK 本身而忽略这些额外的数据库。在过去，ITK 用户 Mailing List 阻塞的大部分原因是由于第三方数据库的编译和安装造成的而不是由于 ITK 本身的创建问题。

ITK 已经在许多不同的操作系统下通过了开发和测试。硬件平台包括 MS-Windows、Intel 编译硬件环境下的 Linux、Solaris、IRIX、Mac OSX 和 Cygwin。它可以在以下编译器条件下工作：

- Visual Studio 6, .NET 2002, .NET 2003
- GCC 2.95.x, 2.96, 3.x
- SGI MIPSpro 7.3x
- Borland 5.5

由于在平台中使用了 C++特征的改进用法，所以有一些编译器在处理代码时会出现问题。这也正是你换掉旧编译器升级软件的机会。

2.1　ITK 设置

使用 CMake 可以使 ITK 跨平台工作。CMake 是一个跨平台，开放源码的安装（编译）工具。可以使用简单的语言来描述所有平台的安装（编译）过程。CMake 的应用是非常广泛的，它支持要求系统设置、编译器特征测试和代码继承的复杂环境。

CMake 可以输出 UNIX 和 Cygwin 系统下的 Makefiles 和 Windows 系统下的 Visual Studio 工作区（支持像 Borland 之类的其他编译器安装文件）。CMake 的信息由 CMakeList.txt 文件提供。这些文件在 ITK 目录中可以找到。这些文件为用户提供了 CMake 的安装时间、系统的公用路径和软件选项的选择等信息。

2.1.1　CMake 准备

CMake 可以在网站 http://www.cmake.org 上免费下载。

ITK 需要 CMake 2.0 版及其以上版本。将版本下载到普通平台包括 Windows、Solaris、IRIX、HP、Mar 和 Linux，也可以选择下载源代码并安装到 CMake 中。按照网站上的下载和安装软件步骤进行操作。

运行 CMake 需要提供两个信息：源代码输入目录地址（ITK SOURCE DIR）和对象代码输出目录地址（ITK BINARY DIR）。这两个目录是不同的，但是设置为同一个目录 ITK 也会照常运行。

在 Unix 系统下，由用户创建目标目录而由 CMake 来给源代码目录设定路径。例如：

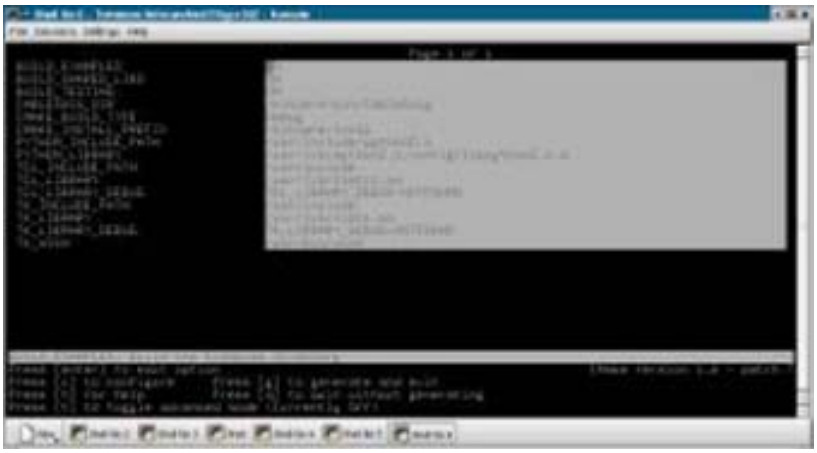
mkdir Insight-binary

cd Insight-binary

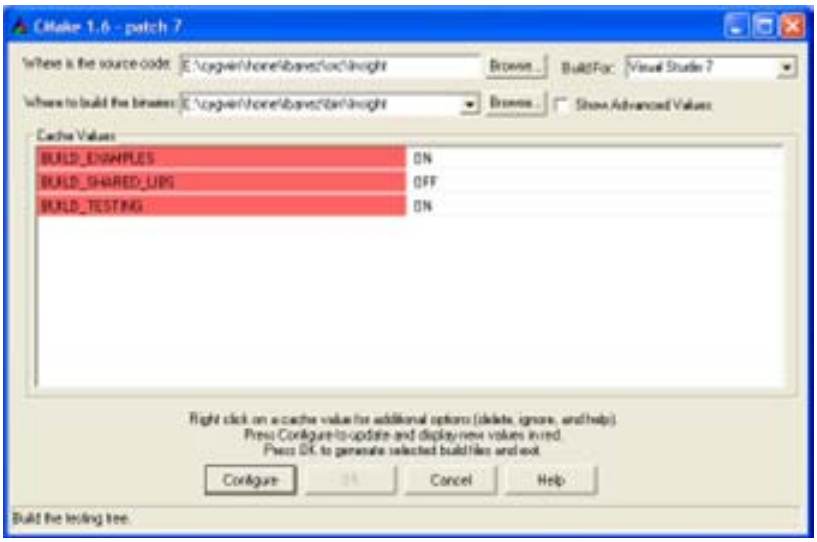
ccmake ../Insight

在 Windows 系统下，用 CMake GUI 来指定源代码和创建目录（如图 2-1 所示）。

CMake 运行在一种迭代模式中、并不断地对选项进行相应配置的对话方式下。迭代过程直到选择完所有的对话框为止。下一步就是根据你的设置创建相应的文件。



(a) UNIX 系统基于 Courses 的 ccmake



(b) MS-Windows 系统下基于 MFC 的 CmakeSetup

图 2-1　CMake 界面

你可以将这个迭代设置的过程理解为一个有很多分叉的路口。你所做的每一个选择都有可能产生新的独立的选项，这些新的选项将出现在 CMake 界面上。直到没有新的对话框出现才说明你做完了所有必要的设定。根据你的设定就会创建出相应的文件。

2.1.2 ITK 设置

图 2-1 展示了 UNIX 和 MS-Windows 系统的 CMake 界面。为了加快安装进度，可以将选项 BUILD TESTING 和 BUILD EXAMPLES 设置为 OFF，这样就不用安装测试和实例。这些实例只是为了帮助用户学习如何使用 ITK 成员而并不是研发平台本身的使用方法。测试部分包括有大量检测 ITK 类的功能的小程序，安装这些测试程序无疑将增加安装时间。对于初次安装平台的人来说，这些都是不必要的。

在 InsightApplications 模块中有一些附加代码。这些代码包括一系列多重应用的混合 GUIs 和不同级别的视窗。由于这个模块中的应用较广且需要用到第三方数据库，所以只有熟悉了平台的基本结构和创建过程以后再创建应用模块。

初次运行 CMake 时，在 Unix 系统下使用 CMake，而在 Windows 系统下使用 CMakeSetup。在 Unix 系统下从目标目录开始运行 CMake。在 Windows 系统下，先给定 GUI 中的源代码和目标目录，然后开始设定必要的 GUI 变量。大部分变量都含有默认值。在 CMake 中，每改变一次变量的设定，都会产生另外一个设置步骤。在 Windows 系统中只要点击“Configure”就可以完成。在 UNIX 系统下是通过在一个 curses 库的界面上敲击“C”键来完成的。

当 CMake 不再出现新的选择时，就可以输出各种格式的 Makefile 或 Visual Studio 工程文件（或根据你的编译器创建相应的文件）。这些文件在 Windows 系统中是通过点击“OK”键创建的，而在 UNIX 系统中，在目标目录做简单的操作即可开始创建过程。在 Windows 系统中，启动运行 CMake GUI 中目标目录里的 ITK.dsw（若使用 MSDEV）或 ITK.sln（若使用.NET 编译器）。

安装编译过程一般需要 15~30 分钟，安装正常的使用程序大概需要编译 600 个小测试程序。这些程序将校验 ITK 基本成员是否正常。

2.2 开始使用 ITK

使用 ITK 创建一个新的工程，最简单的方法就是在你的电脑中建立一个新的文件夹，在文件夹中建立两个文件：一个是 CMakeLists.txt，CMake 使用该文件来创建一个 Makefile（若你使用 UNIX）或一个 Visual Studio 工作区（若你使用 MS-Windows）；另一个文件是真正的调用 ITK 中的类的 C++程序文件。接下来将对这些文件做详细介绍。

一旦建立了这两个文件，你就可以运行 CMake 来设定你的工程。在 UNIX 系统中，你可以命名文件名为“ccmake.”存储在你刚建的文件夹中。注意命令行中的“.”表示 CMakeLists.txt 文件已经存在于当前目录中。这个 curses 界面需要你提供 ITK 创建目录的地址，这个地址和你在设定 ITK 中的 ITK_BINARY_DIR 的目录是一样的。在 Windows 系统下，运行 CMakeSetup 并提供源代码目录和目标目录给你的新工程。CMake 需要你提供一个创建 ITK 的目标目录路径。ITK 目标目录中包括有 ITK 安装过程中设定创建的 ITKConfig.cmake 文件。CMake 将根据这个文件恢复你创建的新工程的所有信息。

Hello World

这里是你在新工程中要写的两个文件目录。这两个文件可以在 Insight/Examples/Installation 目录中找到。CMakeLists.txt 文件包括以下内容：

```
PROJECT(HelloWorld)
FIND_PACKAGE(ITK)
IF(ITK_FOUND)
INCLUDE(${ITK_USE_FILE})
ELSE(ITK_FOUND)
MESSAGE(FATAL_ERROR
"ITK not found. Please set ITK_DIR.")
ENDIF(ITK_FOUND)
ADD_EXECUTABLE(HelloWorld HelloWorld.cxx )
TARGET_LINK_LIBRARIES(HelloWorld ITKCommon)
```

第一行定义了在 Visual Studio 中出现的工程名字(这在 UNIX 系统下毫无影响)。第二行载入一个带有寻找 ITK 的预定义策略的 CMake 文件。如果寻找 ITK 的策略失败，CMake 将会要求你提供 ITK 的安装位置。你可以在 ITK_BINARY_DIR 变量中填入正确的位置信息。命令行 INCLUDE(${ITK_USE_FILE})载入 UseITK.cmake 文件来设定所有的 ITK 配置信息。命令行 ADD_EXECUTABLE 定义了一个此工程产生运行结果文件的名字。ADD_EXECUTABLE 中的保留变量是源代码编译和链接的名字。最后一行命令 TARGET_LINK_LIBRARIES 指出哪些 ITK 类库将链接到这个工程中来。

这一节的源代码可以在 Examples/Installation/HelloWorld.cxx 中找到。

接下来的代码是一段小的应用编程执行代码，用来测试包括头文件和类库的链接：

```
#include "itkImage.h"
#include <iostream>
int main( )
{
typedef itk::Image< unsigned short, 3 > ImageType;
ImageType::Pointer image = ImageType::New( );
std::cout << "ITK Hello World !" << std::endl;
return 0;
}
```

这段程序代码是将一个三维图像中的像素用无符号短字符类型数字代替，然后构造这个图像并指向一个 itk::SmartPointer。后面将详细介绍智能指针，现在你可以把它看做是对一个对象的实例的操作（详见 3.2.4 小节）。类 itk::Image 将在 4.1 节中详细介绍。

到此你已经成功地安装、运行了 ITK，并创建了你的第一个简单程序。如果你有任何问题都可以使用用户 mailing lis（详见 1.4.3 小节）提出问题来寻求帮助。

第三章　系 统 概 述

本章主要介绍研发平台的总体概述。希望通过本章可以使你对 ITK 的使用广度和领域有一定的了解。

3.1　系统组织

平台由许多子系统组成，下面将对一些主要的子系统进行介绍。本章的后半部分（在后续章节中）将对这些概念进行详细介绍（注意：前面章节已经介绍了其他两个 InsightDocuments 和 InsightApplications）。

系统基本概念：像其他软件系统一样，ITK 也是围绕一些核心设计概念创建的。重要的概念包括有：范型编程、内存管理智能指针、可修改对象实例的对象工厂、使用 command/observer 图表的事件管理和多线程支持。

数字化：ITK 使用 VXL 的 VNL 数字类型库。通过 Netlib Fortran 数字化分析路径（http://www.netlib.org）就可以使用 C++来编程。

数据表达和访问：有两个基本的类来表示数据：itk::Image 和 itk::Mesh。另外，有许多类型的迭代器和容器用来保持和转移数据。其他一些重要但不常用的类也可以用来表示数据，例如 histograms 和 BLOX 图像。

数据处理管道：表示数据的类（数据对象）经过滤波器操作被组织进入数据流管道。这些管道保持静态并只在必要时才会运行。它们同样支持多线程和流动功能（例如：能将数据最小化到内存区域中）。

IO 框架：同数据处理管道相关联的是 sources 和 mappers，sources 是初始化管道的滤波器，mappers 是终止管道的滤波器。sources 和 mappers 的标准样例分别是 readers 和 writters。readers 输入数据(通常从一个文件)，而 writters 从管道输出数据。

空间对象：ITK 中使用空间对象层来表示几何图形。这些类支持解剖结构模式。使用一个普通的基本界面，空间对象就可以用不同的方式来表示空间区域。例如：网状结构、图像模块和用来作为潜在表达方案的暗含等式方程。空间对象是一个自然数据结构，它可以与图像分割结果相关联和对分割和配准方法事先做解剖介绍。

配准框架：一个灵活的配准框架支持四种不同的配准类型：图像配准、多方式配准、基于 PDE 的配准和 FEM（有限成员方法 Finite Element Method）配准。

FEM 框架：ITK 包含有一个处理基本 FEM 问题的子系统，尤其是非严格配准。FEM 工具包包含有网格定义（结点和成员）、载入和边界条件。

水平集框架：水平集框架是一个创建滤波器的类群，用来增加有限变换反复迭代的方法求解关于图像的偏微分方程。这个水平集框架由有限的几个不同的 solvers 组成，包括一个稀疏的水平集 solvers、一个通用水平集分割滤波器和一些特定的子类，这些子类包括基于阈值、Canny 和拉普拉斯的方法。

封装：ITK 使用一个独特的、强有力的系统来生成一个界面（例如：封装），该界面用来解释像 Tcl 和 Python 之类的程序语言。GCC_XML 工具用来产生一个对任何复杂 C++代码的 XML 描述；然后使用 CSWIG 工具包将 XML 描述转化成封装。

辅助功能：在系统中有许多辅助子系统可以为其他类做支持。例如：计算机类产生特定的算子来为滤波器提供支持（例如：均值计算机计算一个样本的均值）。其他功能还包括有：一个偏 DICOM 剖析器、MetaIO 文件支持、png、zlib、FLTK/Qt 图像视窗和 VTK 系统界面。

3.2 系统基本概念

本节将介绍 ITK 中的一些核心概念和实际应用特征。

3.2.1 范型编程

范型编程是一种由一般的或可再度使用的软件成员来组成组织类库的编程技术。其目的是以一种有效的、可适应的方式来使得软件具有“plugging together”的能力。范型编程的基本思想是：保持数据的容器、储存数据的迭代器以及范型算法使用容器和迭代器来创建有效的基本算法，如 sorting。范型编程是在 C++中使用模板编程机制和 STL 标准模板类库来实现的。

C++模板是一个编程技术，它允许用户以一种或多种不知类型的 T 形式来编写软件。为了产生可运行代码，软件的用户必须指定所有的类型 T（模板实例）并可以成功地在编译器上处理代码。这个 T 可以是一个已知的类型，如浮点型和整型，也可以是用户定义的类型（如类）。在编译时，编译器要确保模板类型和实例代码是一致兼容的，并且是由必要的方法和操作符支持的。

ITK 在它的实现中使用范型编程的技术。这种方法的优点是通过定义相近的模板类型可以支持几乎无限的数据类型。例如，在 ITK 中可以创建由几乎任何像素类型组成的图像。另外，由于这个类型判决是在编译时执行的，因此编译器可以最优化代码来实现最高的性能。范型编程的缺点是有很多编译器仍然不支持这些更新的概念，而且不能在 ITK 中编译，即使它们支持这些概念，也很有可能产生完全不可辨认的错误信息，哪怕是由于最简单的语法错误造成的。

3.2.2 包含文件和类定义

在 ITK 中类最大限度地定义在两个文件中：一个是.h 后缀的头文件；另一个是执行文件，非模板类时为.cxx 后缀，模板类为.txx 后缀。头文件包括类的声明和格式化注释。格式化注释用来在 Doxygen 文献系统下自动生成 HTML 网页。

除了类的头文件外，下面介绍一些其他重要的头文件。

itkMacro.h：在目录 Code/Common 中定义了标准的系统 macros（例如 Set/Get、常数和其他参数）。

itkNumericTraits.h：在目录 Code/Common 中定义了已知类型的数字化参数，例如它的最大值和最小值。

itkWin32Header.h：在目录 Code/Common 中用来定义控制编译过程的操作系统参数。

3.2.3 对象工厂

ITK 中大部分类是通过对象工厂机制来实例化的。与标准的 C++类使用构造和析构函数不同的是，ITK 中类的实例化是用静态类 New( )方式创建的。事实上，构造和析构是受保护的，因此它不可能用来构造一个 ITK 示例（注意：这种情况适合于由 itk::LightObject 分离出来的类，在一些情况下由于需要增加或减少内存需求量而并非从 LightObject 分离出来类，在这种情况下实例可能是从堆栈创建的，例如 itk::EventObject 类）。

对象工厂使得用户可以通过使用 itk::ObjectFactoryBase 登录一个或多个工厂来控制类的实例的运行时间。这些登录工厂支持 CreateInstance（类名）方式，该方式通过输入一个类名来进行创建。工厂可以通过选择来创建基于包括计算机系统配置和环境变量的一系列因子的类。例如，在一个 ITK 的特定应用中，用户需要使用一个特定的图像处理硬件来实现自己的类的配置。通过使用对象工厂机制，就可以用一个普通类来代替特定的 ITK 滤波器（当然，这个类必须与它所代替的类有相同的 API）。用户就可以创建自己的类（使用同样的编译器、创建选项等），并且在共用的库或 DLL 中插入对象代码。这个库然后被安置在由 ITK_AUTOLOAD_PATH 环境变量确定的目录）。在实例化时，对象工厂将载入该库，使它创建一个特定名字的类并用工厂来创建实例（注意：如果 CreatInstance( )方式找不到创建类名的工厂，类的实例化将失败，并转到普通构造）。

实际上，对象工厂主要应用在 ITK 的 put/output(IO)类中。用户大多使用 New( )方式来创建类。一般地，New( )方式通过在 Common/itkMacro.h 中的 macro itkNewMacro( )方式来声明和实现。

3.2.4 智能指针和内存管理

系统面向对象的、天然的特性通过各种各样的对象类型或类来描述和操作数据。当对一个特定类实例化来产生这个类的实例时，系统会分配内存来存储数据属性值和方法指针（例如纯虚函数指针表）。然后其他类或数据结构就可以用普通程序操作来引用这个对象。一般地，在程序执行过程中，当实例被删除或内存恢复时，应用将会消失。然而，确定实例何时删除是很困难的。删除实例过快会造成程序崩溃；而删除太慢会出现内存泄露（或过多消耗内存）。这种内存分配和释放的过程就是内存管理。

在 ITK 中，内存管理是通过引用计数来实现的。这和另一种普通的包括 JAVA 等其他系统使用的垃圾回收方法形成鲜明对比。在引用计数中，每个实例的应用次数将被记忆。当对象的引用计数为零时进行释放。在垃圾回收中，一个后台处理系统将自动识别并释放不再被系统引用的对象实例。与垃圾回收相关的问题是内存删除的确切时间。当对象数据很大（如以 G 为单位的三维图像）时是不可接受的。引用计数将迅速删除内存（一旦对象的所有引用消失）。

引用计数通过一个 Register( )/Delete( )成员函数界面来实现。一个 ITK 对象的所有实例都是由引用它们的对象使用一个 Register( )方式来调用的。Register( )方式增加实例的引用计数。当实例引用消失时，调用一个 Delete ( )方式来减少实例的引用计数——这和 UnRegister( )

方式是相同的。当引用计数为零时，实例将消失。

使用一个称为 itk::SmartPointer 的类来大大简化这一方案。智能指针的作用与常规指针一样(例如支持操作符->和*)：当引用一个实例时执行 Register( )而当不再引用实例时用 UnRegister( )。与 ITK 中大部分实例不同，智能指针可以指向程序块，当智能指针生成的范围关闭时会自动删除。因此，在 ITK 中就应该尽量少用 Register( )和 Delete( )。例如：

```
MyRegistrationFunction( )
{ <----- Start of scope
// here an interpolator is created and associated to the
// SmartPointer "interp".
InterpolatorType::Pointer interp = InterpolatorType::New( );
} <------ End of scope
```

在这个例子中，有一个引用计数为 1 的引用计数对象（使用 new( )方式）。赋值给对象指针的 interp 并未改变引用计数。在程序最后，interp 被释放，当前对象 interpolator 的引用计数减少，当减少到零时，对象 interpolator 也被释放。

注意：ITK 智能指针通常指向由 itk::LightObject 分离出来的类。方法创新和函数调用经常返回一个指向实例的真指针，但它们立即指向智能指针。当需要增加或内存命令需要一个更小、更快的类时，裸指针指向一个非 LightObject 类。

3.2.5 错误处理和异常处理

一般地，在程序运行中 ITK 使用异常处理来处理错误。异常处理是 C++语言中的一个标准部分，通常按以下形式来阐明：

```
try
{
//...try executing some code here...
}
catch ( itk::ExceptionObject exp )
{
//...if an exception is thrown catch it here
}
```

正如以下程序所论证的那样，这里一个特定的类将抛出一个异常（这是 itk::ByteSwapper 中的一段代码）：

```
switch( sizeof(T))
{
//non-error cases go here followed by error case
default:
ByteSwapperError e (__FILE__, __LINE__);
e.SetLocation("SwapBE");
e.SetDescription("Cannot swap number of bytes requested");
```

throw e;

}

注意：itk::ByteSwapperError 是 itk::ExceptionObject 的一个子类（事实上，ITK 中所有的异常都是从 ExceptionObject 演绎出来的）。在这个例子中，一个特定的结构和 C++预处理程序变量 FILE 和 LINE 用来实例化异常对象并为用户提供附加信息。你可以选择抓住一个特定的异常并因此得到一个特定的 ITK 错误，或者你可以通过抓住异常对象来捕获 ITK 异常。

3.2.6 事件处理

ITK 中的事件处理使用 Subject/Observer 设计模式来执行（有时涉及 Command/Observer 设计模式）。在这一方式中，对象通过注册它们正在观察的实例，来关注这个特定实例所调用的一个特定的事件。例如：ITK 中的滤波器周期性地调用 itk::ProgressEvent。当事件发生时对象已经记录下在这一事件中通报的信息。在记录过程中，通过调用一个命令来发出通告是特定的（例如：函数回收、方法调用等）（注意：ITK 中的事件是 EventObject 的子类；查看 itkEventObject.h 可以得到确定的可能事件）。

例如：ITK 中的许多对象在运行时将从 ProcessObject 调用特定事件:

this->InvokeEvent(ProgressEvent( ));

为了观察这一事件，注册需要结合事件 Object::AddObserver( ) 方法中的一个命令（例如：回收函数）：

unsigned long progressTag =

filter->AddObserver(ProgressEvent( ), itk::Command*);

当事件发生时，所有注册观察者经过调用相应的 Command::Execute( )方式来通告。注意到命令中的许多子类能支持如 C 类型函数的常数和非常数成员函数（查看 Common/Command.h 得到 Command 的预定义子类。如果找不到合适的，可以选择 derivation）。

3.2.7 多线程

ITK 中的多线程是通过一个高层次的设计抽象来处理的。这种方法提供了一个轻便的多线程，并隐藏了不同线程在 ITK 支持的许多系统中实现的复杂度。例如，itk::MultiThreader 类提供支持多线程在一个 SGI 中使用 sproc( )来运行，或者在任何支持 POSIX 线程平台上使用 pthread_creat.

多线程通常是在它的运行阶段通过一个算法来使用的。多线程用来在多个线程中用单一的方法运行或经一个线程指定的方法运行。例如，在 itk::ImageSource 类（对大多数图像处理滤波器的一个超类）中，GenerateData( )方式使用以下方法：

multiThreader->SetNumberOfThreads(int);

multiThreader->SetSingleMethod(ThreadFunctionType, void* data);

multiThreader->SingleMethodExecute( );

在这个例子中，每个线程调用同一方法。多线程滤波器将图像分为许多部分，这些部分

不能对写操作交叠。

在 ITK 基本原理中，线程安全是对一个类的不同实例(和它的方法)进行存储，它是一个安全线程操作。应避免在不同的线程中调用同一个实例方法。

3.3 数字化

ITK 使用 VNL 数字化库为数字化编程提供源代码，综合了像 Mathematica 和 Matlab 工具包的简单使用、C 的速度和 C++的高雅。它提供了一个高质量的 C++语言转换界面，使得数字化分析研究者可以在公共领域进行研究。ITK 通过恰当地包含 VNL 和 ITK 之间的接口类来扩展 VNL 的功能。

VNL 数字化类库中包含的类有：

矩阵和向量：支持标准的矩阵和向量以及对它们类型的操作运算。

标准矩阵和向量类：许多特定的矩阵和向量都有特定的数字化属性。vnl_diagonal_matrix 类提供了一个方便快捷的对角矩阵。固定大小的矩阵和向量允许“fast-as-C”计算（参见 vnl_matrix_fixed<T,n,m>和子类样例 vnl_double_3x3 和 vnl_double_3）。

矩阵分解：vnl_svd<T>、vnl_symmetric_eigensystem<T>和 vnl_generalized_eigensystem 类。

实多项式：类 vnl_real_polynomial 储存了实多项式的系数，并提出了多项式对任意 x 的估计方法。而类 vnl_rpoly_roots 提供了多项式的根。

优化：类 vnl_levenberg_marquardt、vnl_amoeba、vnl_conjugate_gradiedt、vnl_lbfgs 允许用户提供的功能方法在无论有无用户提供的衍生状态下都可以进行优化。

标准函数和常量：类 vnl_math 定义常量（pi、e、eps 等）和简单函数（sqr、abs、rnd 等）。类 numeric_limits 是 ISO 标准文件的一种形式，并提供了一种存储一个类型基本极限的方法。例如，numeric_limits<short>::max( )返回一个短字符型数据的最大值。

大多数 VNL 程序环绕高质量的公式语言转换 Fortran 程序来执行。在过去 40 年中数字化分析机构对 Fortran 程序进行了发展并公布于众。这些程序的中心库是“netlib”服务器 http://www.netlib.org/。国际标准技术委员会在 http://gams.nist.gov 上的 Guide to Available Mathematical Software (GAMS)中为这个库提供了一个完美的搜索界面，同时包括有数搜索和文本搜索。

ITK 也提供了一个额外的数字化功能。一套优化方法可以在 hood 下使用 VNL，并可结合注册框架使用。Insight/Numerics/Statistics 目录中同样也提供了大量的统计学函数（并非由 VNL 而来）。另外，一个完整的有限成员包主要用来支持 ITK 中的可形变注册。

3.4 数据表达

ITK 中有两种基本的数据表示类型：图像和网格。这个功能是在类 Image 和 Mesh 中实现的，这两个类都是 itk::DataObject 的子类。在 ITK 中，数据对象是那些分送系统和参与数据流管道的类（参见 3.5 节）。

itk::Image 表示一个 n 维、规则的样品数据。采样方向平行于任一个坐标轴，采样起点、像素间隔和每个方向上的样品数量（如图像维数）是特定的。这个样品或像素在 ITK 中的类型是任意的，一个模板参数 TPixel 指定了模板实例的类型（当对图像类实例化时图像的维也必须指定）。如果要在所有情况下编译代码（例如使用这些操作通过特定的滤波器来进行处理），关键就是像素的类型必须支持这些操作（如加法和减法）。在实际中 ITK 用户通常使用 C++中的简单类型（如整型、浮点型）或预定义像素类型而很少创建新的像素类型。

ITK 中关于图像的一个重要概念是区域 regions，它是一个矩形的、连续的图像块。区域用来指定图像中处理的部分，例如多线程或保留在内存中的部分。在 ITK 中有三种常用的区域类型：

（1）LargestPossibleRegion——全部图像。

（2）BufferedRegion——保留在内存中的图像部分。

（3）RequestedRegion——对图像操作时滤波器或其他类需要的部分。

网格类表示一个 n 维无结构的格子。网格类的拓扑布局是由一套单元（cells）来表示的，单元（cells）是由一个类型和连通性列表来定义的；这个连通性列表依次涉及点（points）。网格的几何结构是通过 n 维的点（points）组合相关的单元插补函数来定义的。网格设计为一个自适应表达结构，按照在其上实行的操作来进行改变。表达一个网格最起码是需要点和单元的；但是也可以增加额外的拓扑信息。例如，通过增加每个点的信息来把点链接到单元；这提供了在假定暗含的拓扑结构正是所需要的结构情况下的邻域信息。它同样也可以明确地指定边界单元、从暗含的邻域关系来表示不同的连通性和储存单元边界的信息。

网格以三个模板参数的形式来定义：（1）和点、单元和单元边界相关联的一个像素类型；（2）点的维数(依次限制了单元的最大维)；（3）一个网格特征模板参数，指定了用来储存点、单元、边界的容器和迭代器的类型。通过谨慎使用网格特征，可以在允许表达复杂性、存储器和速度之间的一个平衡的条件下，创建更适合于编辑或者更适合于只读操作的网格。

网格是 itk::pointSet 的一个子类。点集类可以用来表示云点和随意分布的特征点等。点集类和拓扑结构无关。

3.5 数据处理管道

数据对象（例如图像和网格）是用来代表数据的，过程对象就是用来操作数据对象并产生新的数据对象的类。像 sourse、filter object 和 mappers 都是过程对象。Sourse（例如 readers）生成数据，filter object 载入数据并经过处理产生新的数据，而 mappers 接收数据并输出到一个文件或其他系统中。有时 filter 可以在广义上表示所有三种类型。

数据处理管道连接了数据对象（如图像和网格）和过程对象。管道支持一个自动更新机制。该机制在当且仅当它的输入或内在状态改变时才会唤起一个滤波器来运行。另外，数据管道还支持 streaming，可以自动将数据分成许多小部分，对这些小部分进行逐一处理，并重新集合处理后的数据产生最终结果。

通常使用 SetInput( )和 GetOutput( )方式连接数据对象和过程对象，方法代码如下：

```
typedef itk::Image<float,2> FloatImage2DType;
```

```
itk::RandomImageSource<FloatImage2DType>::Pointer random;
random = itk::RandomImageSource<FloatImage2DType>::New( );
random->SetMin(0.0);
random->SetMax(1.0);
itk::ShrinkImageFilter<FloatImage2DType,FloatImage2DType>::Pointer shrink;
shrink = itk::ShrinkImageFilter<FloatImage2DType,FloatImage2DType>::New( );
shrink->SetInput(random->GetOutput( ));
shrink->SetShrinkFactors(2);
itk::ImageFileWriter::Pointer<FloatImage2DType> writer;
writer = itk::ImageFileWriter::Pointer<FloatImage2DType>::New( );
writer->SetInput (shrink->GetOutput( ));
writer->SetFileName( "test.raw" );
writer->Update( );
```

在这个例子中，源代码对象 itk::RandomImageSource 同 itk::ShrinkImageFilter 相连接，shrink 滤波器和 mapper itk::ImageFileWriter 相连。当调用 Update( )方式时，数据处理管道就会按顺序使用这些滤波器，并将最终结果储存到文件中。

3.6 空间对象

ITK 空间对象框架支持对象处理的任务就是图像分割和配准的理论。图像只是表示物体的一个媒介，许多处理和数据算法应该能够针对物体本身而并不是基于表示物体的媒介。

ITK 空间对象提供了一个界面，用来储存对象的物理位置、几何性质以及对象之间的关系，储存是在独立于用来表示这些对象的形式的情况下进行的。也就是说，由一个空间对象保持的内在表达可能是一个对象内在的列表、对象的表面网格、对象的内在点或表面的一个连续的或参数的表达等等。

空间对象框架提供的功能支持对象分割、配准、表面/立体视图以及其他显示和分析功能。空间对象框架扩张了对计算机视图工具包很普遍的一个“scene graph”概念，因此也支持这些新函数功能。使用空间对象可以做到以下几点：

（1）指定空间对象的父对象和子对象。在这种方式下，一个 liver 可以包含许多 vessels，而这些 vessels 可以用一个树型结构来组织。

（2）查询一个物理点是否在一个对象或它的子对象中。

（3）求出由一个对象或其子对象所指定的一个相关强度函数上某一物理点的值或派生值。

（4）指定由一个父对象坐标系转换到一子对象坐标系的转换公式。

（5）计算一个空间对象及其子对象的 bounding box。

（6）查询对象最初计算的 resolution。例如，你可以查询一个用来生成 itk::BlobSpatialObject 的一个特定实例的图像的 resolution（例如 voxel 空间）。

目前已经实现的空间对象类型包括：Blob、Ellipse、Group、Image、Line、Surface 和

Tube。对象 itk::Scene 用来控制一个个依次含有子对象的空间对象列表。每个空间对象都以一个色彩特征标记，每个空间对象都有自己的功能。例如，itk::TubeSpatialObjects 表达了与它们相连的父对象。

ITK 中的空间对象和它们的方法有限，但它们的数量正在增长且潜力很大。使用这些名义上的空间对象能力方法如 martching cubes 或 mutual information registration，可以应用于对象而不用考虑它们的内在表示。通过拥有一个共有的 API，同一方法就可以用来将一个心脏的一个参数表示和一个单独的 CT 数据进行配准或将一个肝脏的两个侧面的分割进行配准。

3.7 封装

尽管 ITK 的核心是用 C++来实现的，但也可以自动转换 Tcl 和 Python 语言，使用这些编程语言就可以生成 ITK 程序。这种功能仅仅适用于高级用户而且正在大力发展。若对此工具有兴趣的话，这个概述将给你提供一些有用的意见并告诉你应该看的地方。

由于使用范型编程（例如 C++模板的扩展使用），所以 ITK 中的封装过程是很复杂的。像 VTK 系统是使用自己的封装工具，是非模板化的、是在系统中由编码技术定制的。由于一般的 C++很难被解析，所以即使像 SWIG 这样专为通用封装生成器设计的系统也很难处理 ITK 编码。因此，ITK 封装生成器使用一个综合产生语言的绑定工具。

(1)gccxml 是 GNU 编译器的一个改进版本，可以产生一个输入 C++程序的 XML 描述。

(2)CABLE 处理从 gccxml 而来的 XML 信息并生成下一个工具的额外输入（例如：包含即将封装的信息的 CSWIG）。

(3)CSWIG 是 SWIG 的一个改进版本，用 XML 解析器来代替通常的 SWIG 解析器（从 CABLE 和 gccxml 生成 XML）。CSWIG 生成一个合适的语言绑定（不论是 Tcl 还是 Python）。（注意：由于 SWIG 具有对包括 Java 和 Perl 等 11 种不同语言产生语言绑定的能力，所以大家希望在将来对这些语言以更大的支持力度）。

阅读 Wrapping/CSwig/README 中的文件可以得到更多关于封装处理的信息。在 Wrapping/CSwig/Tests 中也有一些简单的测试文件，在目录 Testing/Code/*/中有大量的测试和样例。

封装处理的结果是产生一系列可供解释语言使用的共享库，依照每种语言不同的句法要求甚至可以直接转换到 C++。例如：在文件夹 Testing/Code/Algorithms 中，测试 itkCurvatureFlowTestTcl2.tcl 中有一段代码如下：

```
set reader [itkImageFileReaderF2_New]
$reader SetFileName "${ITK_TEST_INPUT}/cthead1.png"
set cf [itkCurvatureFlowImageFilterF2F2_New]
$cf SetInput [$reader GetOutput]
$cf SetTimeStep 0.25
$cf SetNumberOfIterations 10
```

在 C++中有同样的代码如下：

```
itk::ImageFileReader<ImageType>::Pointer reader =
```

```
itk::ImageFileReader<ImageType>::New( );
reader->SetFileName("cthead1.png");
itk::CurvatureFlowImageFilter<ImageType,ImageType>::Pointer cf =
itk::CurvatureFlowImageFilter<ImageType,ImageType>::New( );
cf->SetInput(reader->GetOutput( ));
cf->SetTimeStep(0.25);
cf->SetNumberOfIterations(10);
```

这个例子论证了 C++和如 Tcl 等的封装语言的一个重要的差别。模板类在封装之前必须进行实例化。也就是说，模板参数必须指定为封装过程的一部分。在上面的例子中，CurvatureFlowImageFilterF2F2 表示滤波器已经被一个输入图像和二维浮点型数据类型的输出图像（例如 F2）进行了实例化。通常仅仅选择几个普通的类型进行封装处理来避免类型溢出，然后封装到一个类库大小。增加一个新类型就需要再运行封装来生成新库。

通译语言的优点就是不用像 C++语言那样需要一个很长的编译/链接周期。另外，它们通常都会附带一套生成有用功能的工具包。例如，Tk 工具包（例如 Tcl/Tk 和 Python/Tk）提供生成功能广泛的用户界面的工具。将来 ITK 支持的各种通译语言必将能实现更多的应用和测试。

第四章　数 据 表 达

本章将介绍 ITK 中表示数据的基类。典型的类有：itk::Image，itk::Mesh 和 itk::PointSet。

4.1　图像

itk::Image 是遵循范型编程思想的类，其类型是由算法行为的类演化而来。ITK 支持任何像素类型和空间维的图像。

4.1.1　创建图像

这部分源代码在 Examples/DataRepresentation/Image/Image.cxx 中。

这个例子阐述了如何人为地创建一个 itk::Image 类，下面是对图像类进行实例化、声明和创建的最简单程序代码。

首先，必须包含图像类的头文件：

#include “itkImage.h”

然后必须决定像素的类型和图像的维，用这两个参数对图像类进行实例化。下面是创建一个三维、像素是无符号短字符数据类型的图像的程序：

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

然后就可以调用 New( )操作创建图像并将结果分配到 itk::SmartPointer。

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

在 ITK 中，图像以一个或多个区域组合的形式存在。一个区域是图像的一个子集，并有可能是系统中其他类所占有图像的一部分。一个比较普遍的区域是 LargePossibleRegion 是一个完全定义的图像。其他重要的区域还有：BufferedRegion 是内存中图像的一部分；RequestedRegion 是在对图像进行操作时被滤波器或其他类要求的一部分。

如上所述，在 ITK 中人为地创建图像需要对图像进行实例化，并将描述图像的区域与图像结合起来。

一个区域是由两个类来定义的：itk::Index 和 itk::Size 类。与图像结合的图像中的原始区域是由 Index 来定义的。区域的延伸或大小是由 Size 来定义的。Index 是由一个 n 维数列来表示的，在拓扑图像结构中这些表示图像最初的像素的数列的元素都是整数。当人为创建图像时，用户就需要定义图像的大小和图像的起始位置。有了这两个参数，就可以选择处理的区域。

图像的起始点是由一个 Index 类定义的，这个类中存放了一个 n 维数列，数列中的元素都是整数，表示图像中各维上最初的像素值。

ImageType::IndexType start;

start[0] = 0; // first index on X

start[1] = 0; // first index on Y

start[2] = 0; // first index on Z

区域大小是用一个相同大小的图像数列来表示的(使用 Size 类)，数列中的元素是无符号整数，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

定义了起始地址和图像大小这两个参数就可以创建一个 ImageRegion 对象，这个区域是用图像的起始地址和大小来初始化的。

ImageType::RegionType region;

region.SetSize(size);

region.SetIndex(start);

最后，这个区域传递给图像对象来定义其延伸和初始地址。SetRegion 方法同时设定了 LargePossibleRegion、BufferedRegion 和 RequestedRegion。注意：并未执行任何操作来给图像像素数据分配内存，调用 Allocate( )来进行分配内存，直到给区域分配了足够的内存来存放信息分配都不需要任何指令。

image->SetRegions(region);

image->Allocate( );

实际上，很少直接给图像分配内存和对图像进行初始化，图像通常都是从一个源文件直接读取的，比如从一个文件或从硬件获取数据。下面的例子阐述如何从一个文件中读取图像。

4.1.2 从文件读取图像

本小节的源代码在文件 Examples/DataRepresentation/Image/Image2.cxx 中。

从一个文件中读取图像的首要任务是包含 itk::ImageFileReader 类的头文件。

#include "itkImageFileReader.h"

然后，通过指定表示图像的像素和维数的值来定义图像的类型。

typedef unsigned char PixelType;

const unsigned int Dimension = 3;

typedef itk::Image< PixelType, Dimension > ImageType;

使用图像类型就可以把图像 Reader 类实例化。把图像类型作为模板参数来定义读入内存中的数据代表的意义。这里的类型不需要和文件中储存的类型完全一样。由于使用了一个基于 C 格式类型转换器的转换，以至于选做代表硬盘上数据的类型的特点必须被充分地精确表示出来。除了将文件中的像素类型转换为 ImageFileReader 中的像素类型外，不需要再对像素数据做任何转换。下面的程序阐述了 ImageFileReader 类型的一个典型实例。

typedef itk::ImageFileReader< ImageType > ReaderType;

现在就可以用 Reader 类型来创建 Reader 对象。一个由::Pointer 定义的 itk::SmartPointer 指针用来指向新创建的 Reader。调用 New( )方法来创建图像 Reader 的实例。

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

Reader 的最低需求是内存中调用图像的文件名，可以通过 SetFileName( )方法来实现。这里的文件格式可由文件名推断得知。用户也必须使用 itk::ImageIO（具体内容见 7.1 章节）明确地指出数据类型。

const char * filename = argv[1];

reader->SetFileName(filename);

Reader 对象被作为管道源对象，它们对管道更新要求作出回应并发动管道中的数据流。管道更新机制确保 Reader 仅在 Reader 得到数据请求并没有读取任何数据的情况下运行。由于 Reader 的输出并未连接到其他的滤镜，所以在目前的例子中我们必须精确地调用 Updata( )方法。在一般应用中 Reader 的输出是和图像滤镜的输入和滤镜上对 Reader 的更新方式相连接的。下面这行程序是对 Reader 调用更新的阐述：

reader->Update( );

通过对 Reader 使用 GetOutput( )方法来访问新读取的图像。这个方法也可以在对 Reader 更新之前使用，直到 Reader 被运行，即使图像是空的图像 reference 都是有效的。

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

在 Reader 运行之前任何访问图像数据的企图都将产生一个无像素数据的图像，正如未对图像初始化必将导致程序崩溃一样。

4.1.3 访问像素数据

本小节的源代码在文件 Examples/DataRepresentation/Image/Image3.cxx 中。

这个例子阐述了 SetPixel( )和 GetPixel( )方法的用法。这两个方法可以直接访问图像中包含的像素数据。考虑到这两种方法相对缓慢，在高性能的访问需求中不宜使用。图像迭代器是有效访问图像像素数据的合适机制(有关图像迭代器的信息参见第十一章)。

图像中每个像素的位置是由一个特定的 index 来区分的。一个 index 是一个整数数列，它定义了像素在图像中位置的坐标值。index 类型可以由图像自动定义并可以被像 itk::Index 这样的操作访问。数列的长度必须与对应图像的大小相匹配。

下面的代码是对一个 index 变量的声明并对其成员值进行分配。注意：index 并不是用智能指针进行访问的，这是因为 index 是不对任何对象共享的轻度对象。这些小对象产生出多样的版本，比使用智能指针机制分享它们更加有效。

下面的代码是对 index 类型实例的声明并进行初始化，以便与其在图像中的像素位置进行联系：

ImageType::IndexType pixelIndex;

pixelIndex[0] = 27; // x position

pixelIndex[1] = 29; // y position

pixelIndex[2] = 37; // z position

用 index 定义了像素位置后，就可以访问图像中像素的内容。GetPixel( )方法将可得到像素值。

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

SetPixel( )方法可设定像素值。

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

请注意 GetPixel( )使用拷贝来返回像素值，所以该方法不能用来更改图像数据的值。

记住 SetPixel( )和 GetPixel( )这两种方法都是低效率的，只能用来调试或支持交互，例如点击鼠标查询像素值。

4.1.4 定义原点和间距

本小节的源代码在文件 Examples/DataRepresentation/Image/Image4.cxx 中。

尽管 ITK 可以执行一般图像处理的任务，但是这个平台的基本目的是处理医学图像数据，因此必须强制增加图像额外的信息。特别地，在一些坐标系中同像素和图像之间的物理空间位置相关联的信息极其重要。

图像原点和间距是很多应用的基础。例如配准就是在物理坐标中执行的。在这样的处理中没有定义合适的原点和间距必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、图像分析、特征提取、放射辅助治疗和图像指导手术。换句话说，缺乏空间信息的医学图像不仅仅是无用的，而且是很危险的。

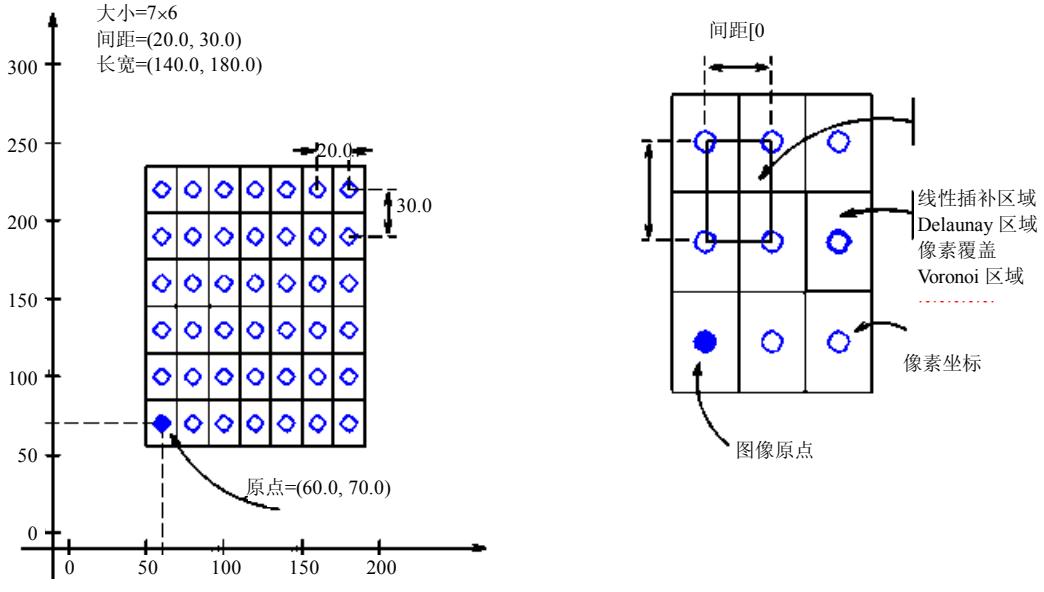


图 4-1 与 ITK 图像相关的几何概念

图 4-1 阐述了与 itk::image 相关的几个重要几何概念。在这个图表中，圆卷表示像素的中心。像素值假定为像素中心的单位脉冲函数。像素间距是像素中心之间的距离，在各个坐标方向上可以是不同的。图像原点是图像中第一个像素的坐标。一个像素就是含有数据值的像素中心周围的矩形区域，如图 4-1 中右图所示，它可以被认为是图像网格中的一个单元。图像值的线性内插法就是在以这些像素中心为拐点的 Delaunay 区域中执行的。

图像间距是在一个大小与图像坐标相匹配的 FixedArray 中表示的。为了人为地设定图像的间距，必须创建一个和图像数据类型相一致的数列。然后将这个数列的元素用相邻像素

中心的间距来进行初始化。下面的程序代码阐述了图像类中处理原点和间距的方法：

```
ImageType::SpacingType spacing;
// Note: measurement 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
```

使用 SetSpacing( )方法来指向数列。

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

使用GetSpacing( )方法可以从图像中得到间距信息。这种方法返回一个FixedArray的值。返回对象可以用来读取数列的内容。注：关键字 const 表示数列是不可修改的。

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

图像原点的处理方式和间距是相似的。必须首先分配好合适的尺寸。任何一个组成部分都可以指定为原点坐标。这些坐标和图像中第一个像素的位置相一致，图像可以用物理空间中的任意参考系统，用户必须确保同一个应用下的所有图像都是使用相同的参考系统，这在图像配准应用中极其重要。

下面的代码阐述了初始化图像原点的变量的创建和分配：

```
ImageType::PointType origin;
origin[0] = 0.0; // coordinates of the first pixel in N-D
origin[1] = 0.0;
origin[2] = 0.0;
image->SetOrigin( origin );
```

使用 GetOrigin( )方法也可以从图像中得到原点。这将产生一个返回值，这个返回值可以用来读取数列中的内容。再次提醒注意：关键字 const 表示数列是不可修改的。

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

图像原点和间距一经初始化，就会以物理空间坐标来正确映射到图像像素。下面的代码阐述了如何将物理空间映射到读取最近像素内容的图像 index 中。

首先，必须声明一个 itk::Point 类型。这个 Point 类型在用来表示坐标的类型和空间大小之上模块化。在这种特殊的情况下，Point 的大小必须和图像的大小相匹配。

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

像itk::Index一样，Point类是一个相对较小、较简单的对象。由于这个原因，它不用像ITK中大的数据对象那样引用计数，从而它也不能同itk::SmartPointers一起操作使用。Point对象像其他C++类实例一样简单进行声明。Point一经声明，就可以使用传统的数列符号来访问它的成员。特别地，允许使用[ ]操作。出于效率原因，在使用index访问一个特殊的Point成员时并不用执行任何校验限制。用户必须确认index在有效范围{0，−1}内。

```
PointType point;
```

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

图像将使用当前的原点和间距把 point 映射到 index。必须提供一个 index 对象来接收映射结果。可以用图像类型中定义的 IndexType 来对 index 对象进行实例化：

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

图像类的 TransformPhysicalPointToIndex( )方法可以计算出与提供的 point 最接近的像素 index。这个方法将核对这个 index 是否被包含在一个当前的缓冲像素数据里。这个方法返回一个布尔类型数据，表示这个 index 结果是否在这个缓冲区间内。当返回值是 false 时，表示输出的 index 无用。

下面的代码阐述了 Point 到 index 的映射和访问图像像素数据的像素 index 的用法：

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

GetPixel( )和 SetPixel( )方法都是非常低效的访问像素数据的方法，当需要大量访问像素数据时应该使用图像迭代器。

4.1.5 RGB 图像

RGB(红、绿、蓝)是数字图像中普遍使用的一种彩色模型。RGB 表示的是使用三基色来分析人类肉眼所能分辨的可见光的代表彩色模型。人体视网膜的敏感细胞可以处理不同类型的光。人眼的锥状体对颜色的灵敏度很高，可以分为三种主要的感觉类别，它们对应红、绿和蓝。另一种杆状体没有彩色感觉，在低照明度下对图像较敏感。第五种感受器——ganglion 细胞，是以能敏感区分白天和夜晚的光照条件而闻名的生理感受器。这些感受器作为一种生物学机制同时间一起进化发展。有节奏的细胞控制存在于生物体的每个细胞中，而且十分精确。

RGB 空间被构造成人眼中三种不同类型的锥状体对光的生理反应的表示形式。RGB 不是一个向量空间。例如在色彩空间中负数是不合适的，因为这将成为在人眼上“负激励”的等价物。在比色法中，负色彩被用来作为一个色彩对比的人为构造：

$$\text{色彩 A}=\text{色彩 B}-\text{色彩 C} \tag{4-1}$$

换种说法就是，我们可以将色彩 A 和色彩 C 混合得到色彩 B。然而，我们必须意识到去除光是不可能的(至少对发射光)。所以当我们提到公式(4.1)时通常我们指：

$$\text{色彩 B}=\text{色彩 A}+\text{色彩 C} \tag{4-2}$$

另一方面，当我们处理打印色彩和画时，与计算机屏幕的发射光相反，光的自然特性允许色彩的减少。严格地说，我们之所以能看到红色的物体，是因为光谱中红色以外的部分被

物体吸收了。

必须小心解释色彩中的加法和减法的概念。事实上，在我们谈及与人眼的三个色彩感觉器相关的途径、或大部分计算机中的三种光管、或油画使用的颜料时，所定义的 RGB 都是不同的。色彩空间通常不是线性的，甚至不仅仅只有一种组合。例如，RGB 空间并不能表示所有的可见光。

ITK 使用 itk::RGBPixel 类型作为一个 RGB 色彩空间的表示。此处的 RGBPixel 类与空间中 itk::Vector 所包含的 RGBPixel 类有着不同的含义。出于这个原因，RGBPixel 缺乏许多期望能对它做的操作。特别地，并无定义减法和加法的操作。

当你对一个 RGB 类型进行求“均值”操作时假定在色彩空间中提供了两种颜色的中间颜色，可以用代表它们的数字进行线性变换得到。不幸的是这在色彩空间中是不存在的，事实上它们是基于人的一种生理反应。

当你简单地用三基色来解释 RGB 图像时，你最好使用 itk::Vector 类型作为像素类型。这种方式下，你必须访问定义在向量空间的操作。在 ITK 中 RGBPixel 的当前执行是假定 RGB 色彩图像是使用在期望色彩正常解释的应用中，因此 RGBPixel 类只允许在色彩空间中可行的操作。

下面的例子阐述了在 ITK 中如何表示 RGB 图像。

本小节的源代码在文件 Examples/DataRepresentation/Image/RGBImage.cxx 中。

多亏了基于 ITK 的范型编程提供的机动性，才使得任意像素类型的图像实例化成为可能。为了使用 itk::RGBPixel 类，必须包含头文件。

```
#include "itkRGBPixel.h"
```

RGBPixeld 类的使用是基于用来代表红、绿和蓝的像素成分的类型之上的。下面是这种模板类的一个典型实例：

```
typedef itk::RGBPixel< unsigned char > PixelType;
```

然后将这个类型作为图像中的像素模板参数：

```
typedef itk::Image< PixelType, 3 > ImageType;
```

这个图像类型可以用来对其他滤镜进行实例化，例如，一个 itk::ImageFileReader 对象从文件中读取图像：

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

现在可以使用 RGBPixel 类提供的方法来执行对像素色彩成分的访问：

```
PixelType onePixel = image->GetPixel( pixelIndex );
PixelType::ValueType red = onePixel.GetRed( );
PixelType::ValueType green = onePixel.GetGreen( );
PixelType::ValueType blue = onePixel.GetBlue( );
```

由于 itk::RGBPixel 从 itk::FixedArray 类继承了[ ]操作，所以也可以使用 subindex 符号：

```
red = onePixel[0]; // extract Red component
green = onePixel[1]; // extract Green component
blue = onePixel[2]; // extract Blue component
std::cout << "Pixel values:" << std::endl;
std::cout << "Red = "
```

```
<< itk::NumericTraits<PixelType::ValueType>::PrintType(red)
<< std::endl;
std::cout << "Green = "
<< itk::NumericTraits<PixelType::ValueType>::PrintType(green)
<< std::endl;
std::cout << "Blue = "
<< itk::NumericTraits<PixelType::ValueType>::PrintType(blue)
<< std::endl;
```

4.1.6 向量图像

本小节的源代码在文件 Examples/DataRepresentation/Image/VectorImage.cxx 中。

许多图像处理任务要求非标量像素类型，一个常见的例子就是向量图像。这是一种要求有表示标量图像梯度的图像类型。下面的代码阐述了如何实例化和使用向量类型像素的图像。

为了方便起见，我们用 itk::Vector 类来定义像素类型。向量类用来表示空间中的集合向量。这同 STL 中把 std::vector 作为数列容器的用法是不同的。如果你对容器感兴趣，itk::VectorContainer 类将提供给你所需要的功能。

第一步就是包含向量类的头文件。

```
#include "itkVector.h"
```

向量类的使用是在基于空间中代表坐标和维数的类型之上进行模板化的。在此例中，向量的长度和图像长度相匹配，但是并不是完全相同。我们可以用一个三维的向量作为像素来定义一个四维的图像。

```
typedef itk::Vector< float, 3 > PixelType;
typedef itk::Image< PixelType, 3 > ImageType;
```

由于向量类从 itk::FixedArray 类继承了[ ]操作，所以也可以使用 index 符号来访问向量成员。

```
ImageType::PixelType pixelValue;
pixelValue[0] = 1.345; // x component
pixelValue[1] = 6.841; // y component
pixelValue[2] = 3.295; // x component
```

现在可以通过定义一个标识并调用 SetPixel( )方法来将这个向量储存到一个图像像素中。

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

4.1.7 从缓冲器中输入图像数据

本小节的源代码在文件 Examples/DataRepresentation/Image/Image5.cxx 中。

这个例子阐述了如何将数据输入到 itk::Image 类中。这在同其他软件系统相连时更加有用。许多系统都使用内存的一个邻近内存块作为图像像素数据的缓冲器。当前样例就是假定

这种情况并在缓冲器中插入一个 itk::ImportImageFilter，从而产生一个图像作为输出。

我们调用内存中心块创建一个同步的图像并将这个内存块传给 ImportImageFilter。这个例子是基于运行上而设定的，用户必须提供一个输出文件名作为一个命令行变量。

首先，必须包含 ImportImageFilter 类的头文件。

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

然后我们选择数据类型来表示图像像素。我们假设内存的外部内存块使用同样的数据类型来表示像素。

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

下面一行是 ImportImageFilter 类型的实例化：

```
typedef itk::ImportImageFilter< PixelType, Dimension > ImportFilterType;
```

使用 New( )方法创建一个滤镜对象然后指向一个智能指针。

```
ImportFilterType::Pointer importFilter = ImportFilterType::New( );
```

滤镜要求用户指定图像的大小来作为输出，使用 SetRgion( )方法即可做到。图像大小必须和当前调用的缓冲器的像素变量的数字相匹配：

```
ImportFilterType::SizeType size;
size[0] = 200; // size along X
size[1] = 200; // size along Y
size[2] = 200; // size along Z
ImportFilterType::IndexType start;
start.Fill( 0 );
ImportFilterType::RegionType region;
region.SetIndex( start );
region.SetSize( size );
importFilter->SetRegion( region );
```

使用 SetOrigin( )方法来指定输出图像的原点：

```
double origin[ Dimension ];
origin[0] = 0.0; // X coordinate
origin[1] = 0.0; // Y coordinate
origin[2] = 0.0; // Z coordinate
importFilter->SetOrigin( origin );
```

使用 SetSpacing( )方法来传递输出图像的间距：

```
double spacing[ Dimension ];
spacing[0] = 1.0; // along X direction
spacing[1] = 1.0; // along Y direction
spacing[2] = 1.0; // along Z direction
importFilter->SetSpacing( spacing );
```

现在我们分配包含像素数据的内存块传递信息到 ImportImageFilter。注意：我们使用与 SetRegion( )方法指定的大小完全相同的尺寸。在实际应用中，你可以使用一个代表图像的不同的数据结构从一些其他的类库中得到这个缓冲器。

```
const unsigned int numberOfPixels = size[0] * size[1] * size[2];
PixelType * localBuffer = new PixelType[ numberOfPixels ];
```

这里可以用一个 binary sphere 来填充这个缓冲器。这里我们像 C 或 FOTTRAN 编程语言一样使用简单的 for( )循环。注意：ITK 在其访问像素的内部编码中不能使用 for( )循环。使用支持处理 n 维图像的 itk::ImageIterators 来代替执行所以的像素访问任务。

```
const double radius2 = radius * radius;
PixelType * it = localBuffer;
for(unsigned int z=0; z < size[2]; z++)
{
const double dz = static_cast<double>( z ) - static_cast<double>(size[2])/2.0;
for(unsigned int y=0; y < size[1]; y++)
{
const double dy = static_cast<double>( y ) - static_cast<double>(size[1])/2.0;
for(unsigned int x=0; x < size[0]; x++)
{
const double dx = static_cast<double>( x ) - static_cast<double>(size[0])/2.0;
const double d2 = dx*dx + dy*dy + dz*dz;
*it++ = ( d2 < radius2 ) ? 255 : 0;
}
}
}
```

缓冲器在 SetImportPointer( )作用下传递到 ImportImageFilter。注意这种方法的最后一个问题是当内存不再使用时指定谁来释放内存。当返回值为假时，表示当调用析构时 ImportImageFilter 并没有释放缓冲器；另一方面，当返回值是真时，表示允许释放析构的输入滤镜上的内存块。

由于 ImportImageFilter 释放了适当的内存块，C++ new( )操作就可以调用这些内存。用其他分配内存机制分配的内存，比如 C 中的 malloc 和 calloc，将不会由 ImportImageFilter 来释放适当的内存。换句话说，编程应用者就需要确保仅仅给 ImportImageFilter 命令来释放 C++新操作分配内存。

```
const bool importImageFilterWillOwnTheBuffer = true;
importFilter->SetImportPointer( localBuffer, numberOfPixels,
importImageFilterWillOwnTheBuffer );
```

最后，我们将这个滤镜的输出连到一个管道上。为简便起见，我们在这里只使用一个 writer，当然其他任何滤镜都可以：

```
writer->SetInput(importFilter->GetOutput( ) );
```

注意：我们传递 true 作为 SetImportPointer( )的最后问题就不需要对缓冲器调用释放操

作。现在缓冲器归 ImportImageFilter 所有。

4.2 点集

4.2.1 创建一个点集

本小节的源代码在文件 Examples/DataRepresentation/Mesh/PointSet1.cxx 中。

Itk::PointSet 是一种在 n 维空间中以点集的形式来表示几何图形的基类。它是为 itk::Mesh 提供操作点集的必要方法的基类。点具有和它们相关的值，这些值的类型是由 itk::PointSet 类（例如 TpixelType）的模板参数来定义的。在 ITK 中存在两种基本格式的 PointSets，分别是静态的和动态的。使用静态格式的条件是：当预先知道设置中点的数目并且在设置后执行操作以后期望结果不被改变的情况。另一方面，使用动态格式的条件是：支持在有效的方式下对点进行插值和移动的情况。区分这两种格式的区别，在进行性能优化和内存管理时就能方便地转变点集 PointSet 的行为。

为了使用点集 PointSet 类，必须包含它的头文件：

```
#include "itkPointSet.h"
```

然后我们必须确定与点相关的数据类型，为了跟 itk::Image 的术语保持一致，通常称这个数据类型为 PixelType。点集 PointSet 也在空间的维上进行模板化，点将在空间中被表达。下面的声明阐述了点集 PointSet 类的一个典型实例。

```
typedef itk::PointSet< unsigned short, 3 > PointSetType;
```

点集 PointSet 对象是通过在它的类型上调用 New( )方法来创建的。对象的结果必须指向一个智能指针。然后将点集 PointSet 引用记数并分享给多个对象。当引用对象的数目减少为零时，分配给点集 PointSet 的内存就会被释放。这就意味着对这种类不需要再调用 Delete( )方法。事实上，对 ITK 中所有引用计数的类都不需要直接调用 Delete( )方法。

```
PointSetType::Pointer pointsSet = PointSetType::New( );
```

在范型编程的基本原理下，点集 PointSet 有一系列相关的定义类型来确保交互对象可以被兼容类型声明。这种类型定义以一系列特征而闻名。例如经过它们可以找到 PointType 类型。这是一种被点设置为在空间中点的表示的类型。下面的声明是以点集 PointSet 特征来定义的点的类型，并重命名使得它在地理空间中可以被方便地使用。

```
typedef PointSetType::PointType PointType;
```

现在可以用 PointType 来声明点对象插入到点集 PointSet 中。点是相对较小的对象，因此可以用引用计数和智能指针来方便地进行处理。它们同典型的 C++类一样可以被简单地实例化，点类从 itk::Array 继承了[ ]操作，这就使得使用 index 符号来对它的成员进行访问成为可能。为了考虑效率，在 index 访问时并不使用校验限制。用户需要做的就是确保 index 的使用范围在{0，−1}内。点中的每个成员都和空间坐标相关。下面的代码阐述了如何对一个点进行实例化并对它的成员进行初始化：

```
PointType p0;
p0[0] = −1.0; // x coordinate
```

```
p0[1] = −1.0; // y coordinate
p0[2] = 0.0; // z coordinate
```

使用 SetPoint( )方法来将点插入点集 PointSet 中。这种方法需要用户为点提供一个独特的标识符。这个标识符通常是一个无符号整数，在将它们插值时列举出所有点。下面的代码展示了如何将三个点插入到点集 PointSet 中：

```
pointsSet->SetPoint( 0, p0 );
pointsSet->SetPoint( 1, p1 );
pointsSet->SetPoint( 2, p2 );
```

可以通过查询点集 PointSet 来确定已经插入点集 PointSet 中点的数目。使用 GetNumberOfPoints( ) 方法的代码阐述如下：

```
const unsigned int numberOfPoints = pointsSet->GetNumberOfPoints( );
std::cout << numberOfPoints << std::endl;
```

可以通过使用 GetPoint( )方法和整形标识符从点集 PointSet 中读取点。点储存在用户提供的一个容器中。如果提供的标识符和存在的点不匹配，这个方法将返回 false，点中的内容就是错误的。下面的代码阐述了使用防御性编程的点访问：

```
PointType pp;
bool pointExists = pointsSet->GetPoint( 1, & pp );
if( pointExists )
{
std::cout << "Point is = " << pp << std::endl;
}
```

GetPoint( )和 SetPoint( )并不是访问点集 PointSet 中的点的最有效方式。直接访问由特征定义在点容器内的点和使用迭代器访问点的列表将更直接、更有效。

4.2.2 得到存储的点

本小节的源代码在文件 Examples/DataRepresentation/Mesh/PointSet2.cxx 中。

Itk::PointSet 类使用一个内部容器来储存 itk::Points。一般来说，使用访问的方法处理由点容器直接提供的点将更加有效。下面的例子阐述了如何与点容器相合和如何使用点迭代器。

这个类型是由点集 PointSet 类的特征来定义的。下面这行程序便捷地从点集 PointSet 特征得到 PointsContatainer 类型并在全局命名空间对它进行声明：

```
typedef PointSetType::PointsContainer PointsContainer;
```

PointsContainer 类型是由所使用的点集 PointSet 的格式来决定的。动态的点集 PointSet 使用 itk::MapContainer，而静态的点集 PointSet 使用 itk::VectorContainer。向量和 map 容器是围绕 STL 类 sed::map 和 std::vector 封装的基本 ITKwrapper。点集 PointSet 的默认使用静态格式，点容器的默认格式是向量容器。map 和向量容器都是在它们包含的成员的类型基础上进行模板化的，当容器作为引用计数对象时它们是在 PointType 上模板化的，然后用 New( ) 方法创建它们并在创建后指向一个 itk::SmartPointer。下面一行创建了一个同点集 PointSet

类型一致的点容器。

PointsContainer::Pointer points = PointsContainer::New( );

现在可以使用 PointType 从点集 PointSet 特征来定义点。

typedef PointSetType::PointType PointType;

PointType p0;

PointType p1;

p0[0] = −1.0; p0[1] = 0.0; p0[2] = 0.0; // Point 0 = {−1,0,0 }

p1[0] = 1.0; p1[1] = 0.0; p1[2] = 0.0; // Point 1 = { 1,0,0 }

使用方法 InsertElement( )将创建的点插入到 PointsContainer 中，这种方法需要给每个点提供一个标识符：

unsigned int pointId = 0;

points->InsertElement(pointId++ , p0);

points->InsertElement(pointId++ , p1);

最后将 PointsContainer 指向点集 PointSet。这将会替代在点集 PointSet 上前面存在的 PointsContainer。这种分派是使用 SetPoint( )方法来完成的。

PointSet->SetPoints(points);

使用 SetPoint( )方法可以从点集 PointSet 来得到 PointsContainer 对象。这种方式返回一个有点集 PointSet 所拥有的指向当前容器的指针，然后指向一个智能指针。

PointsContainer::Pointer points2 = pointSet->GetPoints( );

访问点最有效的方法是使用 PointsContainer 提供的迭代器。迭代器类型属于 PointsContainer 类的特征。它的行为与 STL 迭代器十分相似。点迭代器不是一个引用计数类，所以它可以直接从特征创建而不必使用智能指针。

typedef PointsContainer::Iterator PointsIterator;

迭代器的其他用途可以从 STL 迭代器中了解到。指向的第一个点的迭代器是用 Begin( )方式从容器得到的，并指向另一个迭代器。

PointsIterator pointIterator = points->Begin( );

迭代器中的++操作符可以被用来从一个点指向下一个点。迭代器指向点的当前值可以使用 Value( )方式得到。通过比较当前迭代器和用 End( )方法返回的迭代器就可以控制遍历 PointsContainer 中所有点的循环。下面的代码阐述了遍历点的典型循环：

```
PointsIterator end = points->End( );
while( pointIterator != end )
{
PointType p = pointIterator.Value( ); // access the point
std::cout << p << std::endl; // print the point
++pointIterator; // advance to next point
}
```

注意：在 STL 中使用 End( )方法返回的迭代器是一个无意义的迭代器。这将激发一个遍历的迭代器用来表明它是在访问了容器的最后一个成员后得到的改进值。

使用 Size( )方式可以查询储存在容器中的成员的数目。在点集 PointSet 的情况下，下面

两行代码是等价的，它们都返回点集 PointSet 中点的数目：

```
std::cout << pointSet->GetNumberOfPoints( ) << std::endl;
std::cout << pointSet->GetPoints( )->Size( ) << std::endl;
```

4.2.3 得到点中的存储数据

本小节的源代码在文件 Examples/DataRepresentation/Mesh/PointSet3.cxx 中。

Itk::PointSet 类是与图像类相互关联的。出于这个原因，点集中的点可以很方便地保持它们从图像计算得到的值。与点相关的值被指定为像素类型，以便使它跟图像术语一致。由于范型编程提供的机动性和平台中使用的很接近，所以用户可以按自己的意愿来定义类型。像素类型是点集 PointSet 的第一个模板参数。

下面的代码为像素类型定义了一个特殊的类型并用它实例化一个点集 PointSet：

```
typedef unsigned short PixelType;
typedef itk::PointSet< PixelType, 3 > PointSetType;
```

使用 SetPointData( )方式可以对点集 PointSet 插入数据。这种方式需要用户提供一个标识符。有问题的数据将会保持和点相关的一个相同的标识符。用户需要校验插入的点和插入的数据之间是否能够相互匹配。下面一行程序阐述了 SetPointData( )方式的用法：

```
unsigned int dataId = 0;
PixelType value = 79;
pointSet->SetPointData( dataId++, value );
```

使用 GetPointData( )方式可以从点集 PointSet 读入与点相关的数据。这种方式需要用户提供一个点的标识符和一个指向能安全读入像素数据的位置的指针。当这个标识符和点集 PointSet 中的标识符不相匹配时，这种方式将返回 false，返回的像素值也是无用的。用户在试图使用它以前就需要检验返回的布尔符号。

```
const bool found = pointSet->GetPointData( dataId, & value );
if( found )
{
std::cout << "Pixel value = " << value << std::endl;
}
```

SetPointData( )和 GetPointData( )并不是得到点中存储数据的最有效方法。使用 PointDataContainer 提供的迭代器来得到数据会更加有效得多。

与点相关的点储存在 PointDataContainers 内部。与点使用的方式相同，当前使用的容器的类型取决于点集 PointSet 的类型是静态的还是动态的。静态点集使用 itk::VectorContainer，而动态点集使用 itk::MapContainer。数据容器的类型是作为点集 PointSet 的一个特征来定义的。下面的声明阐述了这个类型如何提取特征并如何用来声明全局命名空间上的一个相似类型：

```
typedef PointSetType::PointDataContainer PointDataContainer;
```

现在使用这个类型就可以创建一个数据容器的实例。这是一个标准的引用计数对象，然后使用 New( )方式创建一个对象并将刚创建的对象指向一个智能指针。

PointDataContainer::Pointer pointData = PointDataContainer::New( );

使用 InsertElement( )方式将像素数据插入到容器中。这种方式需要为每个点数据提供一个标识符。

```
unsigned int pointId = 0;
PixelType value0 = 34;
PixelType value1 = 67;
pointData->InsertElement( pointId++ , value0 );
pointData->InsertElement( pointId++ , value1 );
```

最后将 PointDataContainer 指向点集 PointSet。这将替代点集 PointSet 上以前存在的任何 PointDataContainer。可以使用 SetPointData( ) 方式来完成这个指派。

```
pointSet->SetPointData( pointData );
```

使用 GetPointData( )方式可以从点集 PointSet 来得到 PointDataContainer。这种方式返回一个由点集 PointSet 所拥有指向的当前容器的指针（指向一个智能指针）。

```
PointDataContainer::Pointer pointData2 = pointSet->GetPointData( );
```

访问与点相关的数据最有效的方法是使用 PointDataContainer 提供的迭代器。迭代器类型属于 PointsContainer 类的特征。迭代器不是一个引用计数类，所以它可以直接从特征创建而不必使用智能指针。

```
typedef PointDataContainer::Iterator PointDataIterator;
```

迭代器的其他用途可以从 STL 迭代器中了解到。指向的第一个点的迭代器是用 Begin( ) 方式从容器得到的，并指向另一个迭代器。

```
PointDataIterator pointDataIterator = pointData2->Begin( );
```

迭代器中的++操作符可以被用来从一个点指向下一个点。迭代器指向点的当前值可以使用 Value( )方式得到。通过比较当前迭代器和用 End( )方法返回的迭代器就可以控制遍历 PointsContainer 中所有点的循环。下面的代码阐述了遍历点数据的典型循环。

```
PointDataIterator end = pointData2->End( );
while( pointDataIterator != end )
{
PixelType p = pointDataIterator.Value( ); // access the pixel data
std::cout << p << std::endl; // print the pixel data
++pointDataIterator; // advance to next pixel/point
}
```

注意：在 STL 中使用 End( )方法返回的迭代器不是一个有用的迭代器。这将激发一个遍历的迭代器用来表明它是在访问了容器的最后一个成员后得到的改进值。

4.2.4 RGB 作为像素类型

本小节的源代码在文件 Examples/DataRepresentation/Mesh/RGBPointSet.cxx 中。

下面的例子阐述了如何将一个点集参数化来处理一个特殊的像素类型。在这种情况下就要使用 RGB 类型的像素。第一步就是包含 itk::RGBPixel 和 itk::PointSet 类的头文件：

```
#include "itkRGBPixel.h"
#include "itkPointSet.h"
```

然后通过选择表达 RGB 每个成员的类型来定义像素类型。

```
typedef itk::RGBPixel< float > PixelType;
```

现在使用刚创建的像素类型对点集 PointSet 类型进行实例化，进而创建一个点集对象。

```
typedef itk::PointSet< PixelType, 3 > PointSetType;
PointSetType::Pointer pointSet = PointSetType::New( );
```

下面的代码生成了一个块并将 RGB 值指向点。在这个例子中计算得到的成员的 RGB 值表示点的位置：

```
PointSetType::PixelType pixel;
PointSetType::PointType point;
unsigned int pointId = 0;
const double radius = 3.0;
for(unsigned int i=0; i<360; i++)
{
const double angle = i * atan(1.0) / 45.0;
point[0] = radius * sin( angle );
point[1] = radius * cos( angle );
point[2] = 1.0;
pixel.SetRed( point[0] * 2.0 );
pixel.SetGreen( point[1] * 2.0 );
pixel.SetBlue( point[2] * 2.0 );
pointSet->SetPoint( pointId, point );
pointSet->SetPointData( pointId, pixel );
pointId++;
}
```

点集 PointSet 中的所有点都是使用以下代码来进行访问的：

```
typedef PointSetType::PointsContainer::ConstIterator PointIterator;
PointIterator pointIterator = pointSet->GetPoints( )->Begin( );
PointIterator pointEnd = pointSet->GetPoints( )->End( );
while( pointIterator != pointEnd )
{
PointSetType::PointType point = pointIterator.Value( );
std::cout << point << std::endl;
++pointIterator;
}
```

注意：这里使用 ConstIterator 来代替迭代器，所以像素值是不能被改变的。ITK 以 API 级别来支持 const-correctness.

点集 PointSet 中的所有像素值都是使用以下代码来进行访问的。

```
typedef PointSetType::PointDataContainer::ConstIterator PointDataIterator;
PointDataIterator pixelIterator = pointSet->GetPointData( )->Begin( );
PointDataIterator pixelEnd = pointSet->GetPointData( )->End( );
while( pixelIterator != pixelEnd )
{
PointSetType::PixelType pixel = pixelIterator.Value( );
std::cout << pixel << std::endl;
++pixelIterator;
}
```

再次提醒注意：此处使用常数迭代器代替了迭代器。

4.2.5 向量作为像素类型

本小节的源代码在文件 Examples/DataRepresentation/Mesh/PointSetWithVectors.cxx 中。

这个例子阐述了如何将一个点集参数化来处理一个特殊的像素类型。将向量值和点结合起来创建一个几何表示是很普遍的。下面的代码展示了如何使用向量值作为点集 PointSet 的像素类型。在这里使用 itk::Vector 类作为像素类型。这个类合适地表达了两个点之间的位置关系。然后就可以使用它来处理位移。例如：

为了使用向量类必须包含它的头文件和点集的头文件。

```
#include "itkVector.h"
#include "itkPointSet.h"
```

向量类是在表达空间坐标和空间维的类型上进行模板化的。由于像素类型是不受点类型约束的，所以我们可以对用来作为像素类型的向量的维数进行随意定义。为了创建一个生动的例子，我们使用的向量将可以代表点集 PointSet 中的点的位移。然后这些向量被选择和点集 PointSet 相同的维数。

```
const unsigned int Dimension = 3;
typedef itk::Vector< float, Dimension > PixelType;
```

然后我们使用像素类型（事实上是向量）来对点集 PointSet 类型进行实例化，进而创建一个智能指针对象。

```
typedef itk::PointSet< PixelType, Dimension > PointSetType;
PointSetType::Pointer pointSet = PointSetType::New( );
```

下面的代码生成了一个块并将向量值指向点。如图 4-2 所示，这个例子中计算得到的向量成员用来表示圆上的切线。

```
PointSetType::PixelType tangent;
PointSetType::PointType point;
unsigned int pointId = 0;
const double radius = 300.0;
for(unsigned int i=0; i<360; i++)
{
```

```
const double angle = i * atan(1.0) / 45.0;
point[0] = radius * sin( angle );
point[1] = radius * cos( angle );
point[2] = 1.0; // flat on the Z plane
tangent[0] = cos(angle);
tangent[1] = -sin(angle);
tangent[2] = 0.0; // flat on the Z plane
pointSet->SetPoint( pointId, point );
pointSet->SetPointData( pointId, tangent );
pointId++;
}
```

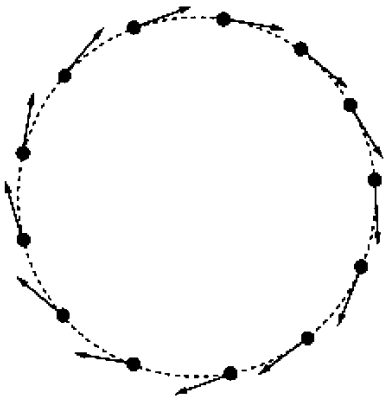


图 4-2　向量作为像素类型

现在我们可以访问所有的点并使用像素值的向量来申请一个点上的位移。这遵循一种在它的每个迭代器上都可以进行位移的可变形模式的原则。

```
typedef PointSetType::PointDataContainer::ConstIterator PointDataIterator;
PointDataIterator pixelIterator = pointSet->GetPointData( )->Begin( );
PointDataIterator pixelEnd = pointSet->GetPointData( )->End( );
typedef PointSetType::PointsContainer::Iterator PointIterator;
PointIterator pointIterator = pointSet->GetPoints( )->Begin( );
PointIterator pointEnd = pointSet->GetPoints( )->End( );
while( pixelIterator != pixelEnd && pointIterator != pointEnd )
{
pointIterator.Value( ) = pointIterator.Value( ) + pixelIterator.Value( );
++pixelIterator;
++pointIterator;
}
```

注意：这里使用常迭代器来代替一般的迭代器，所以这个像素值仅仅用来被读取而不能被改动。ITK 以 API 级别来支持 const-correctness。

Itk::Vector 类拓展了 itk::Point 中的+操作。换句话说，向量可以加到点上来产生一个新的点。在循环的中心开发出这种特性是为了使用一个单独的声明来更新点的位置。

然后我们可以访问所有的点并输出新的数值。

```
pointIterator = pointSet->GetPoints( )->Begin( );
pointEnd = pointSet->GetPoints( )->End( );
while( pointIterator != pointEnd )
{
std::cout << pointIterator.Value( ) << std::endl;
++pointIterator;
}
```

注意：itk::Vector 并不是表达表面的法线和函数梯度的合适的类，这是由于向量在仿射变换上的行为方式决定的。ITK 有一种表达法线和函数梯度的类，就是 itk::ConvariantVector 类。

4.2.6 法线作为像素类型

本小节的源代码在文件 Examples/DataRepresentation/Mesh/PointSetWithCovariantVectors.cxx 中。

使用几何对象表面的点和与这些点相关的法线来表示几何对象是很常见的。使用 itk::PointSet 类可以便捷地对这种结构进行实例化。

用来表达表面的法线和函数的梯度的常见类是 itk::CovariantVector。一个共变向量在仿射变换的行为方式上和一个向量是不同的，特别是在各项异性的缩放比例上不同。如果一个共变向量表示一个函数的梯度，那么变换后的共变向量也将表示变换函数的一个有效的梯度，这种特性是常规向量所不具备的。

下面的代码展示了如何用一个向量值作为点集 PointSet 上的像素类型。在这里使用共变向量类作为像素类型。这个例子阐述了一个可变形的模式如何在潜在函数的梯度的影响下移动。

为了使用共变向量类，必须包含它的头文件和点集 PointSet 的头文件。

```
#include "itkCovariantVector.h"
#include "itkPointSet.h"
```

共变向量类是在用来表达空间坐标和空间维数的类型上进行模板化的。由于像素类型是独立于点类型的，所以我们可以按自己的意愿来选择作为像素类型的共变向量的维数。然而，在这里我们要阐述一下可变形模式的原则，要求表示梯度的向量和空间中的点的维数相同。

```
const unsigned int Dimension = 3;
typedef itk::CovariantVector< float, Dimension > PixelType;
```

然后我们使用像素类型(事实上是共变向量)来对点集 PointSet 类型进行实例化，进而创建一个点集 PointSet 对象。

```
typedef itk::PointSet< PixelType, Dimension > PointSetType;
PointSetType::Pointer pointSet = PointSetType::New( );
```

下面的代码生成了一个块并将向量值指向点的梯度值。这个例子计算得到的共变向量成员用来表示圆上的法线：

```
PointSetType::PixelType gradient;
PointSetType::PointType point;
unsigned int pointId = 0;
const double radius = 300.0;
for(unsigned int i=0; i<360; i++)
{
const double angle = i * atan(1.0) / 45.0;
point[0] = radius * sin( angle );
point[1] = radius * cos( angle );
point[2] = 1.0; // flat on the Z plane
gradient[0] = sin(angle);
gradient[1] = cos(angle);
gradient[2] = 0.0; // flat on the Z plane
pointSet->SetPoint( pointId, point );
pointSet->SetPointData( pointId, gradient );
pointId++;
}
```

现在我们可以访问所有的点，并使用像素值的向量依照函数的梯度来申请一个点上的变形。这遵循一种在它的每个迭代器上都可以进行变形的可变形模式的原则。为了更加整齐匀称，我们应该使用函数梯度作为动力，通过调用胁强张量并增加胁强张量以便得到局部变形。最终的变形结果可以被用来申请点上的位移。为了缩减例子，我们在这里忽略这个复杂的物理因素。

```
typedef PointSetType::PointDataContainer::ConstIterator PointDataIterator;
PointDataIterator pixelIterator = pointSet->GetPointData( )->Begin( );
PointDataIterator pixelEnd = pointSet->GetPointData( )->End( );
typedef PointSetType::PointsContainer::Iterator PointIterator;
PointIterator pointIterator = pointSet->GetPoints( )->Begin( );
PointIterator pointEnd = pointSet->GetPoints( )->End( );
while( pixelIterator != pixelEnd && pointIterator != pointEnd )
{
PointSetType::PointType point = pointIterator.Value( );
PointSetType::PixelType gradient = pixelIterator.Value( );
for(unsigned int i=0; i<Dimension; i++)
{
point[i] += gradient[i];
}
pointIterator.Value( ) = point;
```

```
++pixelIterator;
++pointIterator;
}
```

共变向量类并没有拓展“itk::Point”中的“+”操作。换句话说，共变向量不能被加到点上来产生一个新的点。更进一步，由于我们忽视了例子中的物理因素，所以我们也被迫人为地在梯度和点的坐标上进行非法的加法。

注意：ITK 几何类中缺乏一些基本操作完全是为了得到阻止对它们所表示的数学概念进行不正确使用的目的。

4.3 网格

4.3.1 创建网格

本小节的源代码在文件 Examples/DataRepresentation/Mesh/Mesh1.cxx 中。

itk::Mesh 类用来表示空间中的形状。它源自 itk::PointSet 类，因此继承了与点相关的所有功能，而且接近与点相关的像素数据。网格类是 n 维的，用法中有很强的适应性。

实际上一个网格类可以被认为是一个增加了许多不同维和形状的单元的点集。网格单元使用它们的点标识符是以存在的点的形式来定义的。

和点集的方式相同，ITK 中有两种基本类型的网格，它们分别是静态的和动态的。使用静态格式的条件是：当预先知道设置的点的数目，并且在设置后执行操作以后期望结果不被改变的情况；使用动态格式的条件是：支持在有效的方式下对点进行插值和移动的情况。区分这两种格式的区别，在进行性能优化和内存管理时就能方便地转变它的行为。在网格情况下，动态/静态方面扩展到单元管理。

为了使用网格类，需要包含它的头文件：

```
#include "itkMesh.h"
```

然后，必须选择和点相关的类型并用此来对网格类型进行实例化：

```
typedef float PixelType;
```

网格类型广泛使用了范型编程提供的功能。特别地，网格类是通过像素类型和空间维来参数化的。像素类型是和每个点相关的值的类型同点集中的做法一样。接下来的代码阐述了网格的一个典型实例：

```
const unsigned int Dimension = 3;
typedef itk::Mesh< PixelType, Dimension > MeshType;
```

网格需要提供大量的内存，它们被认为是有价值的对象并使用智能指针来管理。接下来这行代码阐述了如何通过调用 MeshType 的 New( )方式来创建一个网格并将结果指向一个 itk::SmartPointer：

```
MeshType::Pointer mesh = MeshType::New( );
```

网格中点的管理方式和点集中的方式相同。可以通过 PointType 的特征来得到与网格相关的点的类型。接下来的代码展示了上面已经定义的网格类型兼容的点的创建并在它们的坐

标上分配值：

```
MeshType::PointType p0;
MeshType::PointType p1;
MeshType::PointType p2;
MeshType::PointType p3;
p0[0]= -1.0; p0[1]= −1.0; p0[2]= 0.0; // first point ( −1, −1, 0 )
p1[0]= 1.0; p1[1]= −1.0; p1[2]= 0.0; // second point ( 1, −1, 0 )
p2[0]= 1.0; p2[1]= 1.0; p2[2]= 0.0; // third point ( 1, 1, 0 )
p3[0]= −1.0; p3[1]= 1.0; p3[2]= 0.0; // fourth point ( −1, 1, 0 )
```

现在可以使用 SetPoint( )方式来在网格中插入点。注意：点是被拷贝到网格结构中的这就意味着可以改变局部点的实例而不影响网格的内容。

```
mesh->SetPoint( 0, p0 );
mesh->SetPoint( 1, p1 );
mesh->SetPoint( 2, p2 );
mesh->SetPoint( 3, p3 );
```

使用 GetNumberOfPoints( )方式可以查询网格中当前点的数目。

```
std::cout << "Points = " << mesh->GetNumberOfPoints( ) << std::endl;
```

现在可以使用点容器的迭代器来有效地访问点，和前面章节中对点集的做法相同。首先，从网格特征提取出点迭代器类型：

```
typedef MeshType::PointsContainer::Iterator PointsIterator;
```

使用点容器的 Begin( )方式来初始化一个点迭代器到第一个点：

```
PointsIterator pointIterator = mesh->GetPoints( )->Begin( );
```

现在使用“++”操作符来从一个点前进到下一个点。可以使用 Value( )方式来得到迭代器指向点的当前值。通过比较当前的迭代器和通过点容器的 End( )方式返回的迭代器来控制遍历所以点的循环。接下来的几行代码阐述了遍历点的典型循环：

```
PointsIterator end = mesh->GetPoints( )->End( );
while( pointIterator != end )
{
MeshType::PointType p = pointIterator.Value( );   // access the point
std::cout << p << std::endl;                      // print the point
++pointIterator;                                  // advance to next point
}
```

4.3.2 插入单元

本小节的源代码在文件 Examples/DataRepresentation/Mesh/Mesh2.cxx 中。

一个 itk::Mesh 包括许多单元类型。典型的单元有 itk::LineCell、itk::TriangleCell、itk::QuadrilateralCell 和 itk::TetrahedronCell。提供给单元管理额外的机动性是以增加点管理中的复杂性为代价的。

接下来的代码创建了一个多边形的线，以便阐述一个网格中最简单的单元管理。这里仅以使用 LineCell 单元类型为例。必须包含它的头文件：

#include "itkLineCell.h"

为了和网格一致，使用从网格特征提取的许多常见类型来设置单元类型。和网格类所包含的单元相关的特征设置为 CellType 特征。这个特征需要在它们实例化的时候传递给当前单元类型。接下来的一行代码展示了如何从网格类型提取单元特征：

typedef MeshType::CellType CellType;

现在使用从网格提取的特征来对 LineCell 类型进行实例化。

typedef itk::LineCell< CellType > LineType;

网格管理单元和点的方式的主要不同就是：点是通过拷贝到点容器上来储存的，而单元是使用指针来储存到单元容器中的。使用指针的原因是单元使用 C++的多形态。这就意味着网格使用指向一个通用信元，它是所有指定单元类型的基类。这种体系机构就使得在一个网格中可以综合使用不同的单元类型。另一方面，点是一个单一的类型，只有一个很小的内存需求量，这使得可以直接将它们拷贝到容器。

用指针来管理单元增加了网格的另一复杂程度，因为必须制定一个协议来决定由什么来应答单元内存的分配和释放。这个协议是以叫做 CellAutoPointer 的指针的一个指定类型的形式来执行的。这个指针是基于 itk::AutoPointer 的，在很多方面和智能指针不同。CellAutoPointer 有一个指向实际对象和布尔标识的内部指针，这个布尔标识表达了如果使用 CellAutoPointer 来释放已经毁灭的单元内存的条件。一个 CellAutoPointer 具有自身释放的单元。许多 CellAutoPointer 可以在任何给定时间指向相同单元，但是只有单元只具有一个 CellAutoPointer。

在网格类型中定义 CellAutoPointer 特征并可以按以下代码来提取：

typedef CellType::CellAutoPointer CellAutoPointer;

注意：CellAutoPointer 指向一个通用信元类型。它并不是单元的当前类型，例如 LineCell、TriangleCell 和 TetrahedronCell。这个将影响我们后面访问单元的方式。

我们可以创建一个网格并插入一些点：

```
MeshType::Pointer mesh = MeshType::New( );
MeshType::PointType p0;
MeshType::PointType p1;
MeshType::PointType p2;
p0[0] = −1.0; p0[1] = 0.0; p0[2] = 0.0;
p1[0] = 1.0; p1[1] = 0.0; p1[2] = 0.0;
p2[0] = 1.0; p2[1] = 1.0; p2[2] = 0.0;
mesh->SetPoint( 0, p0 );
mesh->SetPoint( 1, p1 );
mesh->SetPoint( 2, p2 );
```

接下来的代码创建了两个 CellAutoPointer，并使用刚刚创建的单元对象对它们进行初始化。这种情况下创建的当前单元类型是 LineCell。注意单元是使用普通的 C++操作符 new 来创建的。CellAutoPointer 通过使用 TakeOwnership( )方式来得到接收指针的所有权。尽管看

起来很冗长，为了使它更清楚，通过 AutoPointer 来假定内存释放的应答代码还是很有必要的。

```
CellAutoPointer line0;
CellAutoPointer line1;
line0.TakeOwnership( new LineType );
line1.TakeOwnership( new LineType );
```

现在应该把 LineCell 关联到网格中的点，当把它们插入网格时使用指向点的标识符就可以完成。每个单元类型都有一定数量的点和它相关联。例如：LineCell 需要两个点；一个 TriangleCell 需要三个点；一个 TetrahedronCell 则需要四个。单元对点的使用有一个内在的编号系统，它是在范围{0，NumberOfPoints−1}中的一个简单标识。使用 SetPointId( )方式来完成点和单元的关联，这种方式需要用户提供单元中点的内部标识和网格中相关的点标识符。单元内在的标识是 SetPointId( )的第一个参数而网格点标识符是第二个。

```
line0->SetPointId( 0, 0 ); // line between points 0 and 1
line0->SetPointId( 1, 1 );
line1->SetPointId( 0, 1 ); // line between points 1 and 2
line1->SetPointId( 1, 2 );
```

使用 SetCell( )方式来插入单元，它需要一个标识符和指向单元的自动指针，网格将把单元的所有权指向自动指针指向的单元。这个是通过使用 SetCell( )内部完成的。在这种方式中，CellAutoPointer 的析构不包括相应单元的析构。

```
mesh->SetCell( 0, line0 );
mesh->SetCell( 1, line1 );
```

在作为 SetCell( )方式的一个变量完成设置后，CellAutoPointer 将不再拥有单元的所有权。很重要的一点是，在没有对第一个单元所有权做保护之前，是不能使用相同的 CellAutoPointer 作为 SetCell( )方式的变量设置另一个单元的所有权的。

使用 GetNumberOfCells( )方式可以查询当前插入网格的单元数量。

```
std::cout << "Cells = " << mesh->GetNumberOfCells( ) << std::endl;
```

和点的方式相同，可以使用网格中的单元容器的迭代器来访问单元。可以从网格中提取出单元迭代器的特征并用来定义一个局部类型。

```
typedef MeshType::CellsContainer::Iterator CellIterator;
```

然后使用单元容器的 Begin( )和 End( )方式就可以分别得到网格中的第一个和最后一个迭代器。使用 GetCells( )方式来返回网格的单元容器。

```
CellIterator cellIterator = mesh->GetCells( )->Begin( );
CellIterator end = mesh->GetCells( )->End( );
```

最后使用一个标准的循环来遍历所有的单元。注意使用 Value( )方式从单元迭代器得到单元的实际指针，同样也要注意返回值是继承 CellType 的指针。这些指针按等级向下造型以便作为实际 LineCell 类型使用。安全向下造型等级使用 dynamic_cast 操作符来执行，当转化不能安全执行时操作将抛出一个异常。

```
while( cellIterator != end )
{
```

```
MeshType::CellType * cellptr = cellIterator.Value( );
LineType * line = dynamic_cast<LineType *>( cellptr );
std::cout << line->GetNumberOfPoints( ) << std::endl;
++cellIterator;
}
```

4.3.3 管理单元中的数据

本小节的源代码在文件 Examples/DataRepresentation/Mesh/Mesh3.cxx 中。

与将用户数据关联到网格中的点的方式相同，可以将用户数据关联到单元。和单元关联的数据的类型可以不同于点关联的数据的类型。然而，默认这两个类型是相同的。接下来的例子阐述了如何访问与单元关联的数据，方法和访问点数据的方法类似。

考虑到在包含行的网格的例子中，行的值是和每一个行相关联的，必须首先包含网格和单元的头文件：

```
#include "itkMesh.h"
#include "itkLineCell.h"
```

然后定义像素类型并使用它来对网格类型进行实例化。

```
typedef float PixelType;
typedef itk::Mesh< PixelType, 2 > MeshType;
```

现在使用从网格得到的特征来对 itk::LineCell 进行实例化。

```
typedef MeshType::CellType CellType;
typedef itk::LineCell< CellType > LineType;
```

现在我们创建一个网格并对它插入一些点。注意点的维要和网格的维相匹配。这里我们插入类似 log( )函数的一个图表的一个点序列。

```
MeshType::Pointer mesh = MeshType::New( );
typedef MeshType::PointType PointType;
PointType point;
const unsigned int numberOfPoints = 10;
for(unsigned int id=0; id<numberOfPoints; id++)
{
point[0] = static_cast<PointType::ValueType>( id ); // x
point[1] = log( static_cast<double>( id ) ); // y
mesh->SetPoint( id, point );
}
```

创建一系列行单元并使用点迭代器关联到存在的点。在这个例子中，由于行单元是以同样的方式来安排的，所以可以从单元迭代器来推导出点迭代器。

```
CellType::CellAutoPointer line;
const unsigned int numberOfCells = numberOfPoints-1;
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
```

```
{
line.TakeOwnership( new LineType );
line->SetPointId( 0, cellId ); // first point
line->SetPointId( 1, cellId+1 ); // second point
mesh->SetCell( cellId, line ); // insert the cell
}
```

使用 SetCellData( )方式将和单元关联的数据插入到 itk::Mesh 中。它需要用户提供一个标识符和被插入的值。这个标识符应该匹配一个出入单元。在这个简单的例子中，单元标识符的平方用来作为单元数据。注意：在这个指派中 static cast 对 PixelType 的使用。

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
{
mesh->SetCellData( cellId, static_cast<PixelType>( cellId * cellId ) );
}
```

使用 GetCellData( )可以从网格读取单元数据。它需要用户提供找回数据的单元的标识符。用户还应该提供一个有效的指针来指向拷贝数据的位置。

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
{
PixelType value;
mesh->GetCellData( cellId, &value );
std::cout << "Cell " << cellId << " = " << value << std::endl;
}
```

SetCellData( )和 GetCellData( )方式都不是访问单元数据的有效方法。使用创建到 CellDataContainer 中的迭代器可以得到更多访问单元数据的有效方式。

```
typedef MeshType::CellDataContainer::ConstIterator CellDataIterator;
```

注意这里使用 ConstIterator 是因为仅仅需要读取数据。这种方式和已经阐述的访问点数据的方式几乎相同。使用 CellDataContainer 的 Begin( )方式可以得到第一个单元数据的迭代器。使用 End( )方式返回最后一个迭代器。使用 GetCellData( )方式可以从网格得到单元数据容器本身。

```
CellDataIterator cellDataIterator = mesh->GetCellData( )->Begin( );
CellDataIterator end = mesh->GetCellData( )->End( );
```

最后使用一个标准的循环来遍历所有的数据入口。注意使用 Value( )方式来得到数据入口的当前值。PixelType 成员被拷贝到局部变量 cellValue 中。

```
while( cellDataIterator != end )
{
PixelType cellValue = cellDataIterator.Value( );
std::cout << cellValue << std::endl;
++cellDataIterator;
}
```

4.3.4 定制网格

本小节的源代码在文件 Examples/DataRepresentation/Mesh/MeshTraits.cxx 中。

本小节阐述了范型编程的强大能力。

这个研发平台是基于将代码的复杂性保持在一个适中的程度而提供最大的机动适应性来设计的。这在网格中是通过隐藏它的大部分参数并为它们定义合理的默认值来实现的。

一个网格的范型概念结合了很多不同的成员。理论上可以对每个这样的成员使用单独的类型。范型编程用来指定一个与概念相关的许多不同类型的机制，称为 traits。它们基本上是当前类相互影响的所有类型的列表。

itk::Mesh 是由三个参数来模板化的。到目前为止我们仅仅讨论了其中两个：PixelType 和 Dimension

```
typedef double CoordinateType;
typedef double InterpolationWeightType;
typedef itk::DefaultStaticMeshTraits<
PixelType, PointDimension, MaxTopologicalDimension,
CoordinateType, InterpolationWeightType, CellDataType > MeshTraits;
typedef itk::Mesh< PixelType, PointDimension, MeshTraits > MeshType;
```

现在使用从网格提取的特征对 itk::LineCell 类型进行实例化：

```
typedef MeshType::CellType CellType;
typedef itk::LineCell< CellType > LineType;
```

现在我们创建一个网格并插入一些值。注意点的维要和网格的维相匹配。这里我们插入类似 log( )函数的一个图表的一个点序列。

```
MeshType::Pointer mesh = MeshType::New( );
typedef MeshType::PointType PointType;
PointType point;
const unsigned int numberOfPoints = 10;
for(unsigned int id=0; id<numberOfPoints; id++)
{
point[0] = 1.565; // Initialize points here
point[1] = 3.647; // with arbitrary values
point[2] = 4.129;
mesh->SetPoint( id, point );
}
```

创建一系列行单元并使用点迭代器关联到存在的点。在这里例子中，由于行单元是以同样的方式来安排的，所以可以从单元迭代器来推导出点迭代器。注意：在上面的代码中，指向点成员的值是任意的。在一个更现实的例子中，这些值将从另一个源来计算。

```
CellType::CellAutoPointer line;
const unsigned int numberOfCells = numberOfPoints-1;
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
{
line.TakeOwnership( new LineType );
line->SetPointId( 0, cellId ); // first point
line->SetPointId( 1, cellId+1 ); // second point
mesh->SetCell( cellId, line ); // insert the cell
}
```

使用 SetCellData( )方式将和单元关联的数据插入到 itk::Mesh 中。它需要用户提供一个标识符和被插入的值。这个标识符应该匹配一个出入单元。在这个简单的例子中，单元标识符的平方用来作为单元数据。注意在这个指派中 static cast 对 PixelType 的使用。

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
{
```

CellDataType value;

mesh->SetCellData(cellId, value);

}

使用 GetCellData( )可以从网格读取单元数据。它需要用户提供找回数据的单元的标识符。用户还应该提供一个有效的指针来指向拷贝数据的位置。

for(unsigned int cellId=0; cellId<numberOfCells; cellId++)

{

CellDataType value;

mesh->GetCellData(cellId, &value);

std::cout << "Cell " << cellId << " = " << value << std::endl;

}

SetCellData( )和 GetCellData( )方式都不是访问单元数据的有效方法。使用创建到 CellDataContainer 中的迭代器可以得到更多访问单元数据的有效方式。

typedef MeshType::CellDataContainer::ConstIterator CellDataIterator;

注意这里使用 ConstIterator 是因为仅仅需要读取数据。这种方式和已经阐述的访问点数据的方式几乎相同。使用 CellDataContainer 的 Begin( )方式可以得到第一个单元数据的迭代器。使用 End( )方式返回最后一个迭代器。使用 GetCellData( )方式可以从网格得到单元数据容器本身。

CellDataIterator cellDataIterator = mesh->GetCellData( )->Begin( );

CellDataIterator end = mesh->GetCellData( )->End( );

最后使用一个标准的循环来遍历所有的数据入口。注意：使用 Value( )方式来得到数据入口的当前值。PixelType 成员被拷贝到局部变量 cellValue 中。

while(cellDataIterator != end)

{

PixelType cellValue = cellDataIterator.Value( );

std::cout << cellValue << std::endl;

++cellDataIterator;

}

4.3.5 拓扑学和 K-复合波

本小节的源代码在文件 Examples/DataRepresentation/Mesh/MeshKComplex.cxx 中。

itk::Mesh 类支持拓扑学形式的表示。特别地，在网格中可以正确的表达 K-Complex 的定义。K-复合波的一个定义信息是：K-复合波是一个拓扑结构，在这个结构中每个单元都是 N 维的，它的边界面都是 N−1 维的，同样也属于这个结构。

本小节阐述了如何使用网格来对一个 K-复合波进行实例化。这个例子的结构是由一个四面体组成，包括它的 4 个三角形表面、6 条边和 4 个顶点。

必须在网格类的头文件后包含所有涉及的单元类型的头文件：

#include "itkMesh.h"

```
#include "itkVertexCell.h"
#include "itkLineCell.h"
#include "itkTriangleCell.h"
#include "itkTetrahedronCell.h"
```

然后定义像素类型并使用它实例化网格类型，注意在这种情况下空间是三维的：

```
typedef float PixelType;
typedef itk::Mesh< PixelType, 3 > MeshType;
```

现在使用从网格得到的特征来实例化单元类型：

```
typedef MeshType::CellType CellType;
typedef itk::VertexCell< CellType > VertexType;
typedef itk::LineCell< CellType > LineType;
typedef itk::TriangleCell< CellType > TriangleType;
typedef itk::TetrahedronCell< CellType > TetrahedronType;
```

创建网格并插入与顶点相关联的点。注意：在网格中的点和 itk::VertexCell 定义之间有一个很重要的区别。VertexCell 是一个零维的单元。它和一个点的主要区别是单元可以被认为是其他单元的邻域关系。实际上，从纯粹拓扑学的观点来看，网格中的点的坐标是完全不相关的，最好把它们从网格结构中提取出来。另一方面，完全可以使用 VertexCell 来表达 K-复合波上的所有邻域关系。

可以通过从一个规则的立方体的其他结点来得到一个规则的四面体的结点的几何坐标。

```
MeshType::Pointer mesh = MeshType::New( );
MeshType::PointType point0;
MeshType::PointType point1;
MeshType::PointType point2;
MeshType::PointType point3;
point0[0] = −1; point0[1] = −1; point0[2] = −1;
point1[0] = 1; point1[1] = 1; point1[2] = −1;
point2[0] = 1; point2[1] = −1; point2[2] = 1;
point3[0] = −1; point3[1] = 1; point3[2] = 1;
mesh->SetPoint( 0, point0 );
mesh->SetPoint( 1, point1 );
mesh->SetPoint( 2, point2 );
mesh->SetPoint( 3, point3 );
```

现在我们创建单元，将它们和点相关联并将其插入到网格。从四面体开始我们写出接下来的代码：

```
CellType::CellAutoPointer cellpointer;
cellpointer.TakeOwnership( new TetrahedronType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 1 );
cellpointer->SetPointId( 2, 2 );
```

```
cellpointer->SetPointId( 3, 3 );
mesh->SetCell( 0, cellpointer );
```

现在创建 4 个三角形表面并关联到网格。第一个三角形连接到点 0、1、2：

```
cellpointer.TakeOwnership( new TriangleType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 1 );
cellpointer->SetPointId( 2, 2 );
mesh->SetCell( 1, cellpointer );
```

第二个三角形连接到点 0、2、3：

```
cellpointer.TakeOwnership( new TriangleType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 2 );
cellpointer->SetPointId( 2, 3 );
mesh->SetCell( 2, cellpointer );
```

第三个三角形连接到点 0、3、1：

```
cellpointer.TakeOwnership( new TriangleType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 3 );
cellpointer->SetPointId( 2, 1 );
mesh->SetCell( 3, cellpointer );
```

第四个三角形连接到点 3、2、1：

```
cellpointer.TakeOwnership( new TriangleType );
cellpointer->SetPointId( 0, 3 );
cellpointer->SetPointId( 1, 2 );
cellpointer->SetPointId( 2, 1 );
mesh->SetCell( 4, cellpointer );
```

注意：每次使用 CellAutoPointer 的方法。提醒：当 itk::AutoPointer 作为 SetCell( )的方式的一个变量传递时将失去单元的所有权。使用 TakeOwnership( )方式将 AutoPointer 附到一个新的单元。

现在使用四面体边缘的 6 条边的创建来继续 K-复合波的构造：

```
cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 1 );
mesh->SetCell( 5, cellpointer );
cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 1 );
cellpointer->SetPointId( 1, 2 );
mesh->SetCell( 6, cellpointer );
cellpointer.TakeOwnership( new LineType );
```

```
cellpointer->SetPointId( 0, 2 );
cellpointer->SetPointId( 1, 0 );
mesh->SetCell( 7, cellpointer );
cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 1 );
cellpointer->SetPointId( 1, 3 );
mesh->SetCell( 8, cellpointer );
cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 3 );
cellpointer->SetPointId( 1, 2 );
mesh->SetCell( 9, cellpointer );
cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 3 );
cellpointer->SetPointId( 1, 0 );
mesh->SetCell( 10, cellpointer );
```

最后创建由 itk::VertexCell 表达的零维单元并插入到网格中：

```
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 0 );
mesh->SetCell( 11, cellpointer );
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 1 );
mesh->SetCell( 12, cellpointer );
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 2 );
mesh->SetCell( 13, cellpointer );
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 3 );
mesh->SetCell( 14, cellpointer );
```

现在网格包含 4 个点和从 0~14 的 15 个单元。使用 PointContainer 迭代器可以访问这些点：

```
typedef MeshType::PointsContainer::ConstIterator PointIterator;
PointIterator pointIterator = mesh->GetPoints( )->Begin( );
PointIterator pointEnd = mesh->GetPoints( )->End( );
while( pointIterator != pointEnd )
{
std::cout << pointIterator.Value( ) << std::endl;
++pointIterator;
}
```

使用 CellsContainer 迭代器可以访问单元：

```
typedef MeshType::CellsContainer::ConstIterator CellIterator;
CellIterator cellIterator = mesh->GetCells( )->Begin( );
CellIterator cellEnd = mesh->GetCells( )->End( );
while( cellIterator != cellEnd )
{
CellType * cell = cellIterator.Value( );
std::cout << cell->GetNumberOfPoints( ) << std::endl;
++cellIterator;
}
```

注意：单元由指向一个通用信元类型的指针来储存，通用信元类型是所有单元类的基类。这就意味着我们仅仅能访问定义在 CellType 中的虚拟方法。

可以使用定义在 CellType 特征里的迭代器来访问和单元相关联的点标识符。接下来的代码阐述了 PointIdIterators 的用法。PointIdsBegin( )方式返回单元中第一个 point-idIterator 的迭代器。PointIdsEnd( )方式返回单元中最后一个 point-idIterator 的迭代器。

```
typedef CellType::PointIdIterator PointIdIterator;
PointIdIterator pointIditer = cell->PointIdsBegin( );
PointIdIterator pointIdend = cell->PointIdsEnd( );
while( pointIditer != pointIdend )
{
std::cout << *pointIditer << std::endl;
++pointIditer;
}
```

注意：可以使用更传统的*iterator 符号代替由 cell-iterators 使用的 Value( )符号来从迭代器得到 point-identifier。

到此为止，由于我们仅仅介绍了单元，所以并没有对 K-复合波的拓扑结构进行完全定义。ITK 允许用户明确地定义单元之间的邻域关系。很明确的一点是，点标识符的一个智能的探测已经允许一个用户指出邻域关系。例如，共享两个相同的标识符的两个三角形单元可能就是相邻单元。这个邻域关系探测的一些缺点是占用计算时间和一些应用不能接受同样的假设。一种情况就是外科仿真，这个典型应用仿真一个网格中的手术刀切片来表示一个器官，表面可能一个很小的切割就使得两个三角形就不再是邻域关系了。

网格中的邻域关系用符号 BoundryFeature 来表示。每个含有一个指向其他单元的 cell-iterators 的内在列表就可以认为是它的邻域。边界特征是通过维来分类的。例如，一条线将具有和它的两个顶点相关的两个零维的边界特征。一个四面体将具有和它的 4 个顶点、6 条边和 4 个三角形表面相关的零维、一维和二维边界特征。现在到了用户指定这些单元之间的连接的时候了。

我们返回到当前例子的四面体单元，关联到 cell-iterators0 并把四个顶点指向它作为零维的边界。这可以通过在网格类上调用 SetBoundaryAssignment( )方式来完成。

```
MeshType::CellIdentifier cellId = 0; // the tetrahedron
int dimension = 0; // vertices
```

```
MeshType::CellFeatureIdentifier featureId = 0;
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 11 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 12 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 13 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 14 );
```

featureId 是和指定单元有相同维的边界单元的序列相关的一个数字。例如：一个四面体的零维特征是它的 4 个顶点。这个单元的零维 feature-Ids 将在 0~3 的范围内取值。四面体的一维特征是它的 6 条边，它的一维 feature-Ids 将在 0~5 的范围内取值。四面体的二维特征是它的 4 个三角形表面。二维 feature-Ids 将在从 0~3 的范围内取值。如表格 4-1 所示。

表 4-1

维	单元类型	FeatureId 范围	单元标识符
0	VertexCell	[0:3]	{11,12,13,14}
1	LineCell	[0:5]	{5,6,7,8,9,10}
2	TriangleCell	[0:3]	{1,2,3,4}

在上面样例代码中，featureId 范围为 0~3。在这个例子中三角形单元的单元标识符是数字{1,2,3,4}，而顶点单元的单元标识符的数字是{11,12,13,14}。

现在指定四面体的一维边界特征，这些是标识符为{5,6,7,8,9,10}的线单元，注意：由于对每个维计算是独立的，所以特征标识符需要再初始化。

```
cellId = 0; // still the tetrahedron
dimension = 1; // one-dimensional features = edges
featureId = 0; // reinitialize the count
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 5 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 6 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 7 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 8 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 9 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 10 );
```

最后我们指定四面体的二维边界特征。这些是标识符为{1,2,3,4}的 4 个三角形单元。由于每个维上的 feature-Ids 是独立的，所以将 featureId 重新设置为零。

```
cellId = 0; // still the tetrahedron
dimension = 2; // two-dimensional features = triangles
featureId = 0; // reinitialize the count
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 1 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 2 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 3 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 4 );
```

现在我们就可以查询四面体单元的边界特征信息了。例如使用 GetNumberOfBoundary

Features( )方式就可以得到每个维上的边界特征数字。

```
cellId = 0; // still the tetrahedron
MeshType::CellFeatureCount n0; // number of zero-dimensional features
MeshType::CellFeatureCount n1; // number of one-dimensional features
MeshType::CellFeatureCount n2; // number of two-dimensional features
n0 = mesh->GetNumberOfCellBoundaryFeatures( 0, cellId );
n1 = mesh->GetNumberOfCellBoundaryFeatures( 1, cellId );
n2 = mesh->GetNumberOfCellBoundaryFeatures( 2, cellId );
```

使用 GetBoundaryAssigment( )方式就可以对边界分配进行恢复。例如，使用下面的代码就可以得到四面体的零维特征。

```
dimension = 0;
for(unsigned int b0=0; b0 < n0; b0++)
{
MeshType::CellIdentifier id;
bool found = mesh->GetBoundaryAssignment( dimension, cellId, b0, &id );
if( found ) std::cout << id << std::endl;
}
```

接下来的代码阐述了如何为一个三角形表面设置边缘边界：

```
cellId = 2; // one of the triangles
dimension = 1; // boundary edges
featureId = 0; // start the count of features
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 7 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 9 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 10 );
```

4.3.6 表达一个 PolyLine

本小节的源代码在文件 Examples/DataRepresentation/Mesh/MeshPolyLine.cxx 中。

本小节阐述了如何使用 itk::Mesh 来表达一个经典的 PolyLine 结构。

一个 PolyLine 仅仅包含零维和一维单元，由 itk::VertexCell 和 itk::LineCell 来表达。

```
#include "itkMesh.h"
#include "itkVertexCell.h"
#include "itkLineCell.h"
```

然后定义像素类型并使用它来实例化网格类型。注意：这种情况下空间是二维的。

```
typedef float PixelType;
typedef itk::Mesh< PixelType, 2 > MeshType;
```

现在可以使用从网格得到的特征来实例化单元类型。

```
typedef MeshType::CellType CellType;
typedef itk::VertexCell< CellType > VertexType;
typedef itk::LineCell< CellType > LineType;
```

创建网格并插入与顶点相关联的点。注意：在网格中的点和 itk::VertexCell 定义之间有一个很重要的区别，VertexCell 是一个零维的单元，它和一个点的主要区别是单元可以被认为是其他单元的邻域关系。实际上，从纯粹拓扑学的观点来看，网格中的点的坐标是完全不相关的，最好把它们从网格结构中提取出来。另一方面，很有必要使用 VertexCell 来表达 PolyLine 上的所有邻域关系。

在下面这个例子中我们创建一个通过空间 3 个表面连接到空间 4 个顶点上的 PolyLine：

```
MeshType::Pointer mesh = MeshType::New( );
MeshType::PointType point0;
MeshType::PointType point1;
MeshType::PointType point2;
MeshType::PointType point3;
point0[0] = −1; point0[1] = −1;
point1[0] = 1; point1[1] = −1;
point2[0] = 1; point2[1] = 1;
point3[0] = −1; point3[1] = 1;
mesh->SetPoint( 0, point0 );
mesh->SetPoint( 1, point1 );
mesh->SetPoint( 2, point2 );
mesh->SetPoint( 3, point3 );
```

现在我们创建单元，把它们和点关联并将它们插入网格：

```
CellType::CellAutoPointer cellpointer;
cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 1 );
mesh->SetCell( 0, cellpointer );
cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 1 );
cellpointer->SetPointId( 1, 2 );
mesh->SetCell( 1, cellpointer );
cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 2 );
cellpointer->SetPointId( 1, 0 );
mesh->SetCell( 2, cellpointer );
```

最后使用创建的 itk::VertexCell 来表达零维单元并插入到网格中：

```
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 0 );
mesh->SetCell( 3, cellpointer );
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 1 );
```

```
mesh->SetCell( 4, cellpointer );
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 2 );
mesh->SetCell( 5, cellpointer );
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 3 );
mesh->SetCell( 6, cellpointer );
```

现在网格就包括有 4 个点和 3 个单元，使用 PointContainer 迭代器就能访问点：

```
typedef MeshType::PointsContainer::ConstIterator PointIterator;
PointIterator pointIterator = mesh->GetPoints( )->Begin( );
PointIterator pointEnd = mesh->GetPoints( )->End( );
while( pointIterator != pointEnd )
{
std::cout << pointIterator.Value( ) << std::endl;
++pointIterator;
}
```

使用 CellsContainer 迭代器访问单元：

```
typedef MeshType::CellsContainer::ConstIterator CellIterator;
CellIterator cellIterator = mesh->GetCells( )->Begin( );
CellIterator cellEnd = mesh->GetCells( )->End( );
while( cellIterator != cellEnd )
{
CellType * cell = cellIterator.Value( );
std::cout << cell->GetNumberOfPoints( ) << std::endl;
++cellIterator;
}
```

注意：单元作为指针指向一个通用信元类型，通用信元类型是所有特定单元类的一个基类，这就意味着我们仅仅能访问定义在 CellType 中的虚拟单元。

可以使用定义在 CellType 特征里的迭代器来访问和单元相关联的点标识符。接下来的代码阐述了 PointIdIterators 的用法。PointIdsBegin( )方式返回单元中第一个 point-idIterator 的迭代器。PointIdsEnd( )方式返回单元中最后一个 point-idIterator 的迭代器。

```
typedef CellType::PointIdIterator PointIdIterator;
PointIdIterator pointIditer = cell->PointIdsBegin( );
PointIdIterator pointIdend = cell->PointIdsEnd( );
while( pointIditer != pointIdend )
{
std::cout << *pointIditer << std::endl;
++pointIditer;
}
```

注意：可以使用更传统的*iterator 符号代替由 cell-iterators 使用的 Value( )符号来从迭代器得到 point-identifier。

4.3.7 简化网格的创建

本小节的源代码在文件 Examples/DataRepresentation/Mesh/AutomaticMesh.cxx 中。

尽管 itk::Mesh 类是极其概括和灵活的，但是简化它还是有一定的代价的。如果简化正是你所需要的，就可以通过 itk::AutomaticTopologyMeshSource 类的方法以交换一些机动性来完成。基于你创建的单元，这个类将自动创建一个外在的 K-复合波。它明确地包含所有的边界信息，以至于可以很容易地研究作为网格的结果。它包含了所有的共享边缘、顶点和表面，因此一次就可以出现所有的特征。

本小节展示了如何使用 AutomaticTopologyMeshSource 来实例化一个表示 K-复合波的网格。首先我们产生和 4.3.5 小节相同的四面体，然后我们将增加一个洞来阐述源网格的一些额外特征。

在网格类的头文件后包含所以涉及的单元类型的头文件。

```
#include "itkMesh.h"
#include "itkVertexCell.h"
#include "itkLineCell.h"
#include "itkTriangleCell.h"
#include "itkTetrahedronCell.h"
#include "itkAutomaticTopologyMeshSource.h"
```

然后我们定义必要的类型并实例化网格源。两个新类型是 IdentifierType 和 IdentifierArrayType。网格中的每个单元都有一个标识符，是由网格特征来确定类型的。AutomaticTopologyMeshSource 需要所有的顶点和单元的标识符类型都是无符号长整型，同样这些类型也是默认值。然而，如果你创建一个新网格特征类来使用字符串标签作为标识符，网格结果就不能和 itk::AutomaticTopologyMeshSource 兼容。一个 IdentifierArrayType 是 IdentifierType 对象的一个简单 itk::Array。

```
typedef float PixelType;
typedef itk::Mesh< PixelType, 3 > MeshType;
typedef MeshType::PointType PointType;
typedef MeshType::CellType CellType;
typedef itk::AutomaticTopologyMeshSource< MeshType > MeshSourceType;
typedef MeshSourceType::IdentifierType IdentifierType;
typedef MeshSourceType::IdentifierArrayType IdentifierArrayType;
MeshSourceType::Pointer meshSource;
meshSource = MeshSourceType::New( );
```

现在我们来生成四面体。接下来的代码生成了所有的顶点、边和表面，并结合连通性信息将它们增加到网格：

```
meshSource->AddTetrahedron(meshSource->AddPoint( −1, −1, −1 ),
```

```
meshSource->AddPoint( 1, 1, −1 ),meshSource->AddPoint( 1, −1, 1 ),
meshSource->AddPoint( −1, 1, 1 ));
```

函数 AutomaticTopologyMeshSource::AddTetrahedron( )以点标识符来作为参数，这些标识符必须和已经增加的点相关。AutomaticTopologyMeshSource::AddPoint( )返回增加的点的合适标识符类型。首先它会核实点是否已经存在于网格内。如果已经存在于网格内，它返回网格中点的 ID，如果不存在于网格内，它就生成一个新的唯一的 ID，使用这个 ID 增加点，并返回这个 ID。

实际上，AddTetrahedron( )具有同样的方式。如果已经增加了四面体，它保持网格不变并返回四面体的 ID。如果不存在，它将增加四面体(以及它的所有表面、边和顶点)，并生成一个新的返回的 ID。

这样做也是可以的：首先增加所有的点，然后直接使用点的 IDs 增加一定数量的单元。这种方式适合于储存在许多文件中形成 3 维多边形模型的数据。

首先我们增加点(这种情况下是一个大的四面体的顶点)。这个例子同样也阐述了如果需要的话，AddPoint( )也可以用一个单一的 PointType 作为一个参数，而不是一个浮点型序列。另一种可能性(未阐述)是以一个 C-风格数列传递。

```
PointType p;
IdentifierArrayType idArray( 4 );
p[ 0 ] = −2;
p[ 1 ] = −2;
p[ 2 ] = −2;
idArray[ 0 ] = meshSource->AddPoint( p );
p[ 0 ] = 2;
p[ 1 ] = 2;
p[ 2 ] = −2;
idArray[ 1 ] = meshSource->AddPoint( p );
p[ 0 ] = 2;
p[ 1 ] = −2;
p[ 2 ] = 2;
idArray[ 2 ] = meshSource->AddPoint( p );
p[ 0 ] = −2;
p[ 1 ] = 2;
p[ 2 ] = 2;
idArray[ 3 ] = meshSource->AddPoint( p );
```

现在我们增加单元。这一次我们只创建一个四面体的边界，因此我们必须各自增加每个表面。

```
meshSource->AddTriangle( idArray[0], idArray[1], idArray[2] );
meshSource->AddTriangle( idArray[1], idArray[2], idArray[3] );
meshSource->AddTriangle( idArray[2], idArray[3], idArray[0] );
meshSource->AddTriangle( idArray[3], idArray[0], idArray[1] );
```

实际上，我们已经完成了这个工作。例如 AddTriangle(4, 5, 6)，由于 IDs 是按次序从零开始分配的，所以 idArray[0]包含增加的第五个点的 ID。但是只有在你知道这个 ID 的情况下你才能这样做。如果你增加同样的点两次而并未意识到，你的计算将和网格原始资源不同。

如果你在 AddEdge(1,0)之后调用了 AddEdge(0, 1)，结果它们就计算同一个边，因此只增加了一条边。顶点的顺序决定了边的方向，保留的方向是第一个指定的方向。

一旦你创建了自己想要的网格，就可以通过使用 GetOutput( )来访问网格。这里我们将它传递给 cout，cout 输出一些网格主要数据。

和典型的滤波器相比，GetOutput( )不触发一个更新过程。网格总是保持在一个可以增加单元的有效状态，并随时可以被访问。然而，通过其他方式来更改网格是不正确的，除非结合 AutomaticTopologyMeshSource 来做，因为网格源随后会有一个当前网格中点和单元的错误记录。

4.3.8 通过单元迭代

本小节的源代码在文件 Examples/DataRepresentation/Mesh/MeshCellsIteration.cxx 中。

单元储存在 itk::Mesh 中并作为指针指向一个通用信元 itk::CellInterface。这就表示只能调用在这些基本单元类上定义的虚拟方法。为了对每个单元类型使用这种方式，向下造型指向单元当前类型的指针是很必要的。通过使用 GetType( )方式可以安全做到这一点，并允许标识一个单元的当前类型。

我们从假设用一个四面体和它的所有边界表面定义一个网格开始。边界表面包括 4 个三角形、6 条边和 4 个顶点。

使用 CellsContainer 迭代器可以访问单元。迭代器 Value( )对应于 CellType 基类的一个裸指针。

```
typedef MeshType::CellsContainer::ConstIterator CellIterator;
CellIterator cellIterator = mesh->GetCells( )->Begin( );
CellIterator cellEnd = mesh->GetCells( )->End( );
while( cellIterator != cellEnd )
{
CellType * cell = cellIterator.Value( );
std::cout << cell->GetNumberOfPoints( ) << std::endl;
++cellIterator;
}
```

为了以一种安全方式执行向下造型，可以首先使用 GetType( )来查询单元类型。使用头文件 itkCellInterface.h 上的一个 enum 类型已经定义了单元类型的代码。这些代码有：

- VERTEX CELL
- LINE CELL
- TRIANGLE CELL
- QUADRILATERAL CELL
- POLYGON CELL

- TETRAHEDRON CELL
- HEXAHEDRON CELL
- QUADRATIC EDGE CELL
- QUADRATIC TRIANGLE CELL

GetType( )方式返回这些代码中的一个。然后在向下造型单元类型的指针指向当前类型之前可以测试单元类型。例如，下面的代码访问了网格中的所有单元及当前类型 LINE_CELL。只有那些向下造型到 LineType 单元和指向 LineType 的一个方法可以被调用。

```
cellIterator = mesh->GetCells( )->Begin( );
cellEnd = mesh->GetCells( )->End( );
while( cellIterator != cellEnd )
{
CellType * cell = cellIterator.Value( );
if( cell->GetType( ) == CellType::LINE_CELL )
{
LineType * line = static_cast<LineType *>( cell );
std::cout << "dimension = " << line->GetDimension( );
std::cout << " # points = " << line->GetNumberOfPoints( );
std::cout << std::endl;
}
++cellIterator;
}
```

为了执行不同单元中的不同任务，可以对每个单元类型的情况使用一个 switch 语句。接下来的代码阐述了单元中的一个迭代和对每个单元类型采用的不同方法：

```
cellIterator = mesh->GetCells( )->Begin( );
cellEnd = mesh->GetCells( )->End( );
while( cellIterator != cellEnd )
{
CellType * cell = cellIterator.Value( );
switch( cell->GetType( ) )
{
case CellType::VERTEX_CELL:
{
std::cout << "VertexCell : " << std::endl;
VertexType * line = dynamic_cast<VertexType *>( cell );
std::cout << "dimension = " << line->GetDimension( ) << std::endl;
std::cout << "# points = " << line->GetNumberOfPoints( ) << std::endl;
break;
}
case CellType::LINE_CELL:
```

```
{
std::cout << "LineCell : " << std::endl;
LineType * line = dynamic_cast<LineType *>( cell );
std::cout << "dimension = " << line->GetDimension( ) << std::endl;
std::cout << "# points = " << line->GetNumberOfPoints( ) << std::endl;
break;
}
case CellType::TRIANGLE_CELL:
{
std::cout << "TriangleCell : " << std::endl;
TriangleType * line = dynamic_cast<TriangleType *>( cell );
std::cout << "dimension = " << line->GetDimension( ) << std::endl;
std::cout << "# points = " << line->GetNumberOfPoints( ) << std::endl;
break;
}
default:
{
std::cout << "Cell with more than three points" << std::endl;
std::cout << "dimension = " << cell->GetDimension( ) << std::endl;
std::cout << "# points = " << cell->GetNumberOfPoints( ) << std::endl;
break;
}
}
++cellIterator;
}
```

4.3.9 访问单元

本小节的源代码在文件 Examples/DataRepresentation/Mesh/MeshCellVisitor.cxx 中。

为了访问特殊单元类型，在 itk::Mesh 上已经建立了一个便捷机制。这个机制是基于在参考文献[28]中介绍的 Visitor Pattern。Visitor Pattern 是基于便捷地遍历共享一个公有基类的一个异类的列表的过程而设计的。

使用 CellVisitor 机制的首要条件是包含 CellInterfaceVisitor 的头文件：

```
#include "itkCellInterfaceVisitor.h"
```

现在定义一个典型的网格类型：

```
typedef float PixelType;
typedef itk::Mesh< PixelType, 3 > MeshType;
```

```
typedef MeshType::CellType CellType;
typedef itk::VertexCell< CellType > VertexType;
typedef itk::LineCell< CellType > LineType;
typedef itk::TriangleCell< CellType > TriangleType;
typedef itk::TetrahedronCell< CellType > TetrahedronType;
```

然后，需要定义一个用户 CellVisitor 类。在这个特定的例子中，访问器类只在 TriangleType 单元有效。声明访问器类的唯一要求是它必须提供一个称为 Visit( )的方式。这个方式被认为是一个单元标识符和一个指向这个想要访问的单元类型的指针。没有任何东西可以阻止一个访问器类为几种不同的单元类型提供 Visit( )方式。多重方式将通过 C++函数自然机制来区分。接下来的代码是单元访问类的最小限度的代码：

```
class CustomTriangleVisitor
{
public:
typedef itk::TriangleCell<CellType> TriangleType;
public:
void Visit(unsigned long cellId, TriangleType * t )
{
std::cout << "Cell # " << cellId << " is a TriangleType ";
std::cout << t->GetNumberOfPoints( ) << std::endl;
}
};
```

现在使用这个新定义的类来实例化一个单元访问器。在这个特定的例子中我们创建一个 CustomTriangleVisitor 类，当网格迭代器执行到这个单元时，每调用一次这个类将找到一个三角形单元。

```
typedef itk::CellInterfaceVisitorImplementation<PixelType,MeshType::CellTraits,
TriangleType,CustomTriangleVisitor> TriangleVisitorInterfaceType;
```

注意：当前的 CellInterfaceVisitorImplementation 是基于 PixelType、CellTraits 和访问的 CellType 之上模板化的，而且访问器类是使用单元定义的。

现在可以使用通常的 New( )方式来创建一个访问器执行并将结果指向一个 itk::SmartPointer。

```
TriangleVisitorInterfaceType::Pointer triangleVisitor =TriangleVisitorInterfaceType::New( );
```

按照这种方式可以设置许多不同的访问器。所有访问器的设置都可以使用网格提供的 MultiVisitor 类来进行注册。MultiVisitor 类的一个实例将遍历单元并在遇到合适的单元类型时委派每个已经注册的访问器为代表。

```
typedef CellType::MultiVisitor CellMultiVisitorType;
CellMultiVisitorType::Pointer multiVisitor = CellMultiVisitorType::New( );
```

使用 AddVisitor( )方式和网格来注册访问器：

```
multiVisitor->AddVisitor( triangleVisitor );
```

最后通过调用 itk::Mesh 上的 Accept( )方式来触发单元的迭代。

mesh->Accept(multiVisitor);

Accept( )方式将迭代所有的单元，并逐一访问 MultiVisitor 来试图在每个单元上执行一个行为。如果没有访问器访问当前单元类型，单元将被忽略和遗漏。

MultiVisitors 使得在不为单元类型创建新方式和不创建每个 CellType 都知道的一个综合的访问器的前提下增加单元的行为成为可能。

4.3.10 访问单元的更多信息

本小节的源代码在文件 Examples/DataRepresentation/Mesh/MeshCellVisitor2.cxx 中。

接下来将阐述在 itk::Mesh 上使用单元访问器的一个真实的例子。这里定义一系列不同的访问器，每个访问器都和一个特定的单元类型相关联。所有的访问器是使用传递给网格的一个 MultiVisitor 类来注册的。

第一步是包含 CellInterfaceVisitor 的头文件：

```
#include "itkCellInterfaceVisitor.h"
```

现在声明一个典型的网格类型：

```
typedef float PixelType;
typedef itk::Mesh< PixelType, 3 > MeshType;
typedef MeshType::CellType CellType;
typedef itk::VertexCell< CellType > VertexType;
typedef itk::LineCell< CellType > LineType;
typedef itk::TriangleCell< CellType > TriangleType;
typedef itk::TetrahedronCell< CellType > TetrahedronType;
```

然后，需要声明一个用户 CellVisitor 类。声明访问器类的唯一要求是它必须提供一个称为 Visit( )的方式。这个方式被认为是一个单元标识符和一个指向这个想要访问的单元类型的指针。

接下来，顶点访问器简单输出与单元相关点的标识符。注意：单元使用 GetPointID( ) 方式而不必使用任何变量。这种方式只在 VertexCell 上定义。

```
class CustomVertexVisitor
{
public:
void Visit(unsigned long cellId, VertexType * t )
{
std::cout << "cell " << cellId << " is a Vertex " << std::endl;
std::cout << " associated with point id = ";
std::cout << t->GetPointId( ) << std::endl;
}
};
```

接下来的代码访问器计算线的长度。注意：这个访问器更加复杂，因为需要访问当前网格以便从线单元返回的点标识符得到点坐标。这可以通过指向一个网格指针来完成并在每次

访问网格时点坐标也被访问。在这种情况下使用 SetMesh( )方式来设置网格指针。

```
class CustomLineVisitor
{
public:
CustomLineVisitor( ):m_Mesh( 0 ) {}
void SetMesh( MeshType * mesh ) { m_Mesh = mesh; }
void Visit(unsigned long cellId, LineType * t )
{
std::cout << "cell " << cellId << " is a Line " << std::endl;
LineType::PointIdIterator pit = t->PointIdsBegin( );
MeshType::PointType p0;
MeshType::PointType p1;
m_Mesh->GetPoint( *pit++, &p0 );
m_Mesh->GetPoint( *pit++, &p1 );
const double length = p0.EuclideanDistanceTo( p1 );
std::cout << " length = " << length << std::endl;
}
private:
MeshType::Pointer m_Mesh;
};
```

下面代码是三角形访问器输出它的点的标识符。注意：PointIdIterator、PointIdsBegin( )和 PointIdsEnd( )方式的用法。

```
class CustomTriangleVisitor
{
public:
void Visit(unsigned long cellId, TriangleType * t )
{
std::cout << "cell " << cellId << " is a Triangle " << std::endl;
LineType::PointIdIterator pit = t->PointIdsBegin( );
LineType::PointIdIterator end = t->PointIdsEnd( );
while( pit != end )
{
std::cout << " point id = " << *pit << std::endl;
++pit;
}
}
};
```

下面代码是四面体访问器简单返回这个图的表面的数目。注意：GetNumberOfFaces( )是 3 维单元的一个扩展。

```
class CustomTetrahedronVisitor
{
public:
void Visit(unsigned long cellId, TetrahedronType * t )
{
std::cout << "cell " << cellId << " is a Tetrahedron " << std::endl;
std::cout << " number of faces = ";
std::cout << t->GetNumberOfFaces( ) << std::endl;
}
};
```

现在我们使用单元访问器来实例化 CellVisitor 的执行。上面定义的访问器作为单元访问器执行的模板参数。

```
typedef itk::CellInterfaceVisitorImplementation<
PixelType, MeshType::CellTraits, VertexType, CustomVertexVisitor
> VertexVisitorInterfaceType;
typedef itk::CellInterfaceVisitorImplementation<
PixelType, MeshType::CellTraits, LineType, CustomLineVisitor
> LineVisitorInterfaceType;
typedef itk::CellInterfaceVisitorImplementation<
PixelType, MeshType::CellTraits, TriangleType, CustomTriangleVisitor
> TriangleVisitorInterfaceType;
typedef itk::CellInterfaceVisitorImplementation<
PixelType, MeshType::CellTraits, TetrahedronType, CustomTetrahedronVisitor
> TetrahedronVisitorInterfaceType;
```

注意：当前 CellInterfaceVisitorImplementation 是基于 PixelType、CellTraits 和访问的 CellType 之上模板化的，而且访问器类是使用单元定义的。

现在可以使用通常的 New( )方式来创建一个访问器执行并将结果指向一个 itk::SmartPointer。

```
VertexVisitorInterfaceType::Pointer vertexVisitor =
VertexVisitorInterfaceType::New( );
LineVisitorInterfaceType::Pointer lineVisitor =
LineVisitorInterfaceType::New( );
TriangleVisitorInterfaceType::Pointer triangleVisitor =
TriangleVisitorInterfaceType::New( );
TetrahedronVisitorInterfaceType::Pointer tetrahedronVisitor =
TetrahedronVisitorInterfaceType::New( );
```

记住 LineVisitor 需要指向网格对象的指针，因为它需要访问当前点坐标。这可以通过调用前面定义的 SetMesh( )方式来实现。

```
lineVisitor->SetMesh( mesh );
```

仔细观察你就会发现，SetMesh( )方式是在 CustomLineVisitor 中声明的，但是我们在 LineVisitorInterfaceType 上调用它，在定义 VisitorInterfaceImplementation 的方式下是允许的。这个类与用户提供的作为第四个模板参数的访问器类型是不同的，LineVisitorInterfaceType 是源自 MultiVisitor 类的。

现在使用 MultiVisitor 类来注册访问器的设置，MultiVisitor 类将遍历单元并在遇到合适的单元类型时委派每个已经注册的访问器为代表，接下来几行创建了一个 MultiVisitor 对象。

```
typedef CellType::MultiVisitor CellMultiVisitorType;
CellMultiVisitorType::Pointer multiVisitor = CellMultiVisitorType::New( );
```

每个访问器的执行都是使用 AddVisitor( )方式和网格注册。

```
multiVisitor->AddVisitor( vertexVisitor );
multiVisitor->AddVisitor( lineVisitor );
multiVisitor->AddVisitor( triangleVisitor );
multiVisitor->AddVisitor( tetrahedronVisitor );
```

最后通过调用网格类上的 Accept( )方式来触发单元的迭代：

```
mesh->Accept( multiVisitor );
```

Accept( )方式将迭代所有的单元，并逐一访问 MultiVisitor 来试图在每个单元上执行一个行为。如果没有访问器访问当前单元类型，单元将被忽略和遗漏。

4.4 路径

创建一个 PolyLineParametricPath

本小节的源代码在文件 Examples/DataRepresentation/Path/PolyLineParametricPath1.cxx 中。

这个例子阐述了如何使用 itk::PolyLineParametricPath，这个类是典型地应用于以一种简明的方式来表达在 2 维情况下一个图像分割算法的输出。然而 PolyLineParametricPath 也可以作为一个分段线性化方式用来在 N 维情况下表示任何开或闭曲线。

首先必须包含 PolyLineParametricPath 类的头文件：

```
#include "itkPolyLineParametricPath.h"
```

路径是基于图像维来实例化的，在这种情况下是 2 维。

```
const unsigned int Dimension = 2;
typedef itk::Image< unsigned char, Dimension > ImageType;
typedef itk::PolyLineParametricPath< Dimension > PathType;
ImageType::ConstPointer image = reader->GetOutput( );
PathType::Pointer path = PathType::New( );
path->Initialize( );
typedef PathType::ContinuousIndexType ContinuousIndexType;
ContinuousIndexType cindex;
```

```
typedef ImageType::PointType ImagePointType;
ImagePointType origin = image->GetOrigin( );
ImageType::SpacingType spacing = image->GetSpacing( );
ImageType::SizeType size = image->GetBufferedRegion( ).GetSize( );
ImagePointType point;
point[0] = origin[0] + spacing[0] * size[0];
point[1] = origin[1] + spacing[1] * size[1];
image->TransformPhysicalPointToContinuousIndex( origin, cindex );
path->AddVertex( cindex );
image->TransformPhysicalPointToContinuousIndex( point, cindex );
path->AddVertex( cindex );
```

4.5 容器

本节的源代码在文件 Examples/DataRepresentation/Containers/TreeContainer.cxx 中。

这个例子展示了如何使用 itk::TreeContainer 并和 TreeIterators 关联。itk::TreeContainer 是执行树的想法并基于结点类型模板化，因此它可以真实地操作任何对象。假如每个结点只有一个父代，那么在树中是没有循环的，也不需要校验来确认一个循环树。

我们从包含相应的头文件开始：

```
#include <itkTreeContainer.h>
#include "itkTreeContainer.h"
#include "itkChildTreeIterator.h"
#include "itkLeafTreeIterator.h"
#include "itkLevelOrderTreeIterator.h"
#include "itkInOrderTreeIterator.h"
#include "itkPostOrderTreeIterator.h"
#include "itkPreOrderTreeIterator.h"
#include "itkRootTreeIterator.h"
#include "itkTreeIteratorClone.h"
```

首先，我们创建一个整数树。基于结点类型来对 TreeIterators 模板化：

```
typedef int NodeType;
typedef itk::TreeContainer<NodeType> TreeType;
TreeType::Pointer tree = TreeType::New( );
```

接下来我们使用 SetRoot( )来设置根结点的值：

```
tree->SetRoot(0);
```

然后我们使用 Add( )函数增加树的结点，第一个变量是新结点的值，第二个变量是当前结点的值。如果两个结点的值相同，就取第一个值。在这种特定情况下，最好使用一个迭代器来填充树：

```
tree->Add(1,0);
tree->Add(2,0);
tree->Add(3,0);
tree->Add(4,2);
tree->Add(5,2);
tree->Add(6,5);
tree->Add(7,1);
```

我们定义一个 itk::LevelOrderTreeIterator 来按级别顺序解析树，这个特定的迭代器有 3 个变量，第一个是当前解析的树，第二个是级别的最大值，而第三个是起始结点。GetNode( ) 函数返回一个给定值的结点。再次返回和这个值相关的第一个结点。

```
itk::LevelOrderTreeIterator<TreeType> levelIt(tree,10,tree->GetNode(2));
levelIt.GoToBegin( );
while(!levelIt.IsAtEnd( ))
{
std::cout << levelIt.Get( ) << " ("<< levelIt.GetLevel( ) << ")" << std::endl;;
++levelIt;
}
std::cout << std::endl;
```

TreeIterators 含有测试当前指向结点所有权的函数。在这些函数中，如果当前结点是一个分支，则 IsLeaf 返回为真；如果这个结点是根，则 IsRoot 返回为真；如果结点含有父代，则 HasParent 返回为真；而 CountChildren 返回指定结点的子代数目。

```
levelIt.IsLeaf( );
levelIt.IsRoot( );
levelIt.HasParent( );
levelIt.CountChildren( );
```

itk::ChildTreeIterator 提供了另外一种方式：它通过罗列结点的所有子代来迭代一个树：

```
itk::ChildTreeIterator<TreeType> childIt(tree);
childIt.GoToBegin( );
while(!childIt.IsAtEnd( ))
{
std::cout << childIt.Get( ) << std::endl;;
++childIt;
}
std::cout << std::endl;
```

GetType( )函数返回迭代器使用的类型。类型列表列举如下：PREORDER，INORDER，POSTORDER，LEVELORDER，CHILD，ROOT 和 LEAF。

```
if(childIt.GetType( ) != itk::TreeIteratorBase<TreeType>::CHILD)
{
std::cout << "[FAILURE]" << std::endl;
```

return EXIT_FAILURE;
}
TreeIterator 含有一个 Clone( )函数，返回当前迭代器的一个拷贝。注意：用户必须手动释放创建的迭代器。

```
childIt.GoToParent( );
itk::TreeIteratorBase<TreeType>* childItClone = childIt.Clone( );
delete childItClone;
```

itk::LeafTreeIterator 遍历树的所有分支：

```
itk::LeafTreeIterator<TreeType> leafIt(tree);
leafIt.GoToBegin( );
while(!leafIt.IsAtEnd( ))
{
std::cout << leafIt.Get( ) << std::endl;;
++leafIt;
}
std::cout << std::endl;
```

itk::InOrderTreeIterator 是按从左到右的顺序遍历树：

```
itk::InOrderTreeIterator<TreeType> InOrderIt(tree);
InOrderIt.GoToBegin( );
while(!InOrderIt.IsAtEnd( ))
{
std::cout << InOrderIt.Get( ) << std::endl;;
++InOrderIt;
}
std::cout << std::endl;
```

itk::PreOrderTreeIterator 是按从左到右的顺序遍历树，但是首先搜索级别：

```
itk::PreOrderTreeIterator<TreeType> PreOrderIt(tree);
PreOrderIt.GoToBegin( );
while(!PreOrderIt.IsAtEnd( ))
{
std::cout << PreOrderIt.Get( ) << std::endl;;
++PreOrderIt;
}
std::cout << std::endl;
```

itk::PostOrderTreeIterator 是按从左到右的顺序遍历树，从分支到根的顺序搜索的：

```
itk::PostOrderTreeIterator<TreeType> PostOrderIt(tree);
PostOrderIt.GoToBegin( );
while(!PostOrderIt.IsAtEnd( ))
{
```

```
std::cout << PostOrderIt.Get( ) << std::endl;;
++PostOrderIt;
}
std::cout << std::endl;
```

itk::RootTreeIterator 是从一个结点到根的搜索，第二个变量是起始结点，这里我们从分支结点(值为 6)向上搜索到根。

```
itk::RootTreeIterator<TreeType> RootIt(tree,tree->GetNode(6));
RootIt.GoToBegin( );
while(!RootIt.IsAtEnd( ))
{
std::cout << RootIt.Get( ) << std::endl;;
++RootIt;
}
std::cout << std::endl;
```

使用 Clear( )函数可以再次移动树的所有结点：

```
tree->Clear( );
```

我们展示如何使用一个 TreeIterator 通过创建结点来形成一个树，使用 Add( )函数增加一个结点并对它赋值，使用 GoToChild( )来跳过一个结点。

```
itk::PreOrderTreeIterator<TreeType> PreOrderIt2(tree);
PreOrderIt2.Add(0);
PreOrderIt2.Add(1);
PreOrderIt2.Add(2);
PreOrderIt2.Add(3);
PreOrderIt2.GoToChild(2);
PreOrderIt2.Add(4);
PreOrderIt2.Add(5);
```

可以使用 itk::TreeIteratorClone 来对一个迭代器生成一个通用拷贝：

```
typedef itk::TreeIteratorBase<TreeType> IteratorType;
typedef itk::TreeIteratorClone<IteratorType> IteratorCloneType;
itk::PreOrderTreeIterator<TreeType> anIterator(tree);
IteratorCloneType aClone = anIterator;
```

第五章　空 间 对 象

本章将介绍描述itk::SpatialObject的基类。

5.1　绪论

我们提倡这样的理论：如果把医学图像处理放在对象处理的大背景下的话，那么将更加有效地完成医学图像处理的目标。ITK的空间对象类提供了一个兼容一致的API，用来查询、操作和互相连接物理空间中的对象。经过这个API，就可以通过编码的方法将处理的对象转化为一个特定的数据结构来进行存储。通过提炼对象的表达方式来支持它们由数据结构而不是由图像来表达的方式，从而支持一个大的医学图像分析研究范围。接下来介绍几个重要的例子。

模型到图像注册：一个对象的数学实例可以通过在一个图像中局部化这个对象的实例来进行注册。使用SpatialObject，就可以将相互的信息、相互关系和边界到图像方法不加修改应用于执行空间对象到图像注册。

模型到模型注册：任何ITK变换都可以使用反复迭代接近点、关键特征点和表面距离最小化的方法，来严格地或非严格地注册图像、FEM和基于傅立叶描述的SpatialObject对象的表示。

图集结构：图像和SpatialObject集可以综合表达对象的特征及其变量的共有模型。Lables可以和一个图集的对象相关联。

用一个或多个扫描来存储分割结果：分割的结果最好存储在physical/world里，以便用来和使用不同方式从其他图像得到的分割结果进行组合和比较。无论是从提取轮廓、像素标签还是从模型到图像注册得到的分割结果都是一致对待的。

捕获对象之间的功能和逻辑关系：SpatialObject可以拥有父对象和子对象。可以结合从对象的子对象得到的响应来查询一个对象的组成(例如，确定一个点是否在对象中)。对一个父代的变换同样也可以传递给子代来应用。例如，当移动一个肝脏时，它的脉管也跟着移动。

变换图像和图像的转化：提供将任何SpatialObject(或SpatialObject集)变换成一幅图像的基本功能。

IO：SpatialObject的读写和SpatialObject类的层次结构是相互独立的。提供Meta对象IO(通过itk::MetaImageIO)方式和简单定义其他的方式。

Tubes、blobs、images、surfaces：提供一些SpatialObject数据容器和类型。可以增加新类型，但通常只能在一个起源类中定义一个或两个成员函数。

在本章下面的部分中将介绍一些例子，这些例子用来示例ITK中的许多空间对象并展示如何使用itk::SceneSpatialObject将它们组织到层次结构中。另外这些例子还阐述了如何使用SpatialObject变换来控制和计算对象在空间中的位置。

5.2 层次结构

组合空间对象可以形成一个树型的层次结构。一个SpatialObject仅仅可以有一个父对象。另外由于每次变换都存储在每个对象中，因此层次结构不能作为一个有向无环图(DAG)来描述，而只能作为一个树来描述。用户只需要保持这个树型结构，并不需要做核实来确认其是否是一个自由循环的树。

本节的源代码在文件Examples/SpatialObjects/SpatialObjectHierarchy.cxx中。

这个例子描述了如何使用itk::SpatialObject来形成一个层次结构。作为第一个例子同样也展示了如何创建和操作空间对象。

```
#include "itkSpatialObject.h"
```

首先，我们创建两个空间对象并分别以First Object和Second Object来命名。

```
typedef itk::SpatialObject<3> SpatialObjectType;
SpatialObjectType::Pointer object1 = SpatialObjectType ::New( );
object1->GetProperty( )->SetName("First Object");
SpatialObjectType::Pointer object2 = SpatialObjectType ::New( );
object2->GetProperty( )->SetName("Second Object");
```

然后我们使用AddSpatialObject( )方式来将第二个对象增加到第一个对象上，这样object2就成了object1的一个子对象：

```
object1->AddSpatialObject(object2);
```

如果一个对象有父代，就可以使用HasParent( )方式来进行查询。如果只有一个父对象，GetParent( )方式返回一个指向父对象的常指针。在本例中，如果我们查询object2的父对象的名字，我们将得到First Object：

```
if(object2->HasParent( ))
{
std::cout << "Name of the parent of the object2: ";
std::cout << object2->GetParent( )->GetProperty( )->GetName( ) << std::endl;
}
```

GetChildren( )方式返回指向子代列表的一个指针(STL)，用来存储对象的子代列表：

```
SpatialObjectType::ChildrenListType * childrenList = object1->GetChildren( );
std::cout << "object1 has " << childrenList->size( ) << " child" << std::endl;
SpatialObjectType::ChildrenListType::const_iterator it = childrenList->begin( );
while(it != childrenList->end( ))
{
std::cout << "Name of the child of the object 1: ";
std::cout << (*it)->GetProperty( )->GetName( ) << std::endl;
it++;
}
```

由于GetChildren( )函数创建了一个内在的列表，因此千万不要忘记删除子代列表：

delete childrenList;

可以使用RemoveSpatialObject( )方式来移动一个对象：

object1->RemoveSpatialObject(object2);

我们可以使用GetNumberOfChildren( )方式来查询一个对象的子对象数目：

std::cout << "Number of children for object1: ";

std::cout << object1->GetNumberOfChildren( ) << std::endl;

Clear( )方式擦除所有和对象相关的信息和数据：

object1->Clear( );

接下来是第一个例子的输出：

Name of the parent of the object2: First Object

object1 has 1 child

Name of the child of the object 1: Second Object

Number of children for object1: 0

5.3 SpatialObject 树容器

本节的源代码在文件Examples/SpatialObjects/SpatialObjectTreeContainer.cxx中。

这个例子阐述了如何使用itk::SpatialObjectTreeContainer来形成一个SpatialObjects的层次结构。首先我们包含相应的头文件：

#include "itkSpatialObjectTreeContainer.h"

接下来我们定义结点的类型和我们计划使用的树的类型，这两个类型都是基于空间的维来模板化的。我们创建一个二维树。

typedef itk::GroupSpatialObject<2> NodeType;

typedef itk::SpatialObjectTreeContainer<2> TreeType;

然后，我们创建三个结点并设置它们相应的标识数字(使用SetID)：

NodeType::Pointer object0 = NodeType::New( );

object0->SetId(0);

NodeType::Pointer object1 = NodeType::New( );

object1->SetId(1);

NodeType::Pointer object2 = NodeType::New( );

object2->SetId(2);

使用AddSpatialObject( )函数来形成层次结构：

object0->AddSpatialObject(object1);

object1->AddSpatialObject(object2);

在树实例化之后使用SetRoot( )函数来设置它的根：

TreeType::Pointer tree = TreeType::New( );

tree->SetRoot(object0.GetPointer( ));

可以使用本书前面章节中介绍的树迭代器来解析这个层次结构。例如，经过一个基于树

的类型模板化的itk::LevelOrderTreeIterator，我们可以解析SpatialObjects的层次结构。在我们的例子中，由于我们的层次结构只有2层，所以我们设置最大层为10已经足够了。

```
itk::LevelOrderTreeIterator<TreeType> levelIt(tree,10);
levelIt.GoToBegin( );
while(!levelIt.IsAtEnd( ))
{
std::cout << levelIt.Get( )->GetId( ) << " ("<< levelIt.GetLevel( )<< ")" << std::endl;;
++levelIt;
}
```

同样也可以使用树迭代器来增加空间对象的层次。这里我们展示如何使用itk::PreOrderTreeIterator来增加树的第四个对象：

```
NodeType::Pointer object4 = NodeType::New( );
itk::PreOrderTreeIterator<TreeType> preIt( tree );
preIt.Add(object4.GetPointer( ));
```

5.4 变换

本节的源代码在文件Examples/SpatialObjects/SpatialObjectTransforms.cxx中。

这个例子描述了与一个空间对象相关的几种不同变换。

图5-1显示了变换系统。

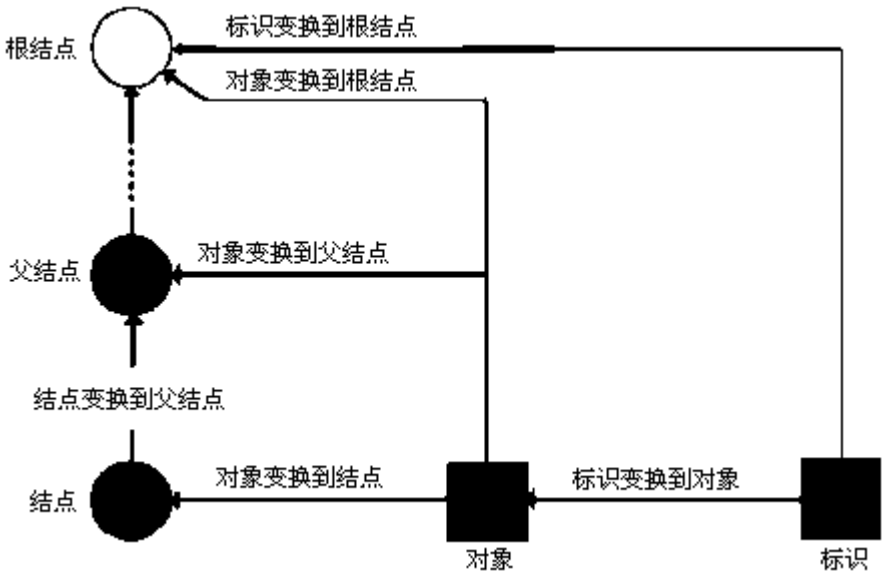


图 5-1　与一个空间对象相关的变换系统

像第一个例子一样，我们创建两个空间对象并分别以First Object和Second Object来命名：

```
SpatialObjectType::Pointer object2 = SpatialObjectType ::New( );
```

```
object2->GetProperty( )->SetName("Second Object");
object1->AddSpatialObject(object2);
```

itk::SpatialObject的实例包含了可以用来计算数据、对象位置和方位的3种内在的变换。这些变换是：一个IndexToObjectTransform、一个ObjectToParentTransform和一个ObjectToWorldTransform。为了方便用户，这个类也包含了全局变换IndexToWorldTransform和它的反变换WorldToIndexTransform。SpatialObject提供了存储和计算这些变换的方法。

两个主要的变换IndexToObjectTransform和ObjectToParentTransform是连续应用的。ObjectToParentTransform是应用于子代的。

IndexToObjectTransform将对象的点从内在数据坐标系(通常是对象定义的图像指标)变换到“物理”空间(说明了指标的间距、方位和偏移量的空间)。

ObjectToParentTransform将点从指定对象的“物理”空间变换到它的父代对象的“物理”空间。可以从图5.1看出，ObjectToParentTransform由两个变换组成：ObjectToNodeTransform和NodeToParentNodeTransform。ObjectToParentTransform不能应用于子代，而ObjectToNodeTransform可以应用于子代。因此如果设置了ObjectToParentTransform，那么就更新了ObjectToNodeTransform。

ObjectToWorldTransform将点从SpatialObject的引用系统映射到全局坐标系。这在仅仅知道对象在全局坐标体系中的位置时是很有用的。注意通过设置这个变换，再次计算了ObjectToParent变换。

这些变换使用了itk::FixedCenterOfRotationAffineTransform，它们在空间itk::SpatialObject的构造器中构造。

首先我们定义对object2的一个指数缩放比例因子为2，这可以通过设置IndexToObjectTransform的比例来完成：

```
double scale[2];
scale[0]=2;
scale[1]=2;
object2->GetIndexToObjectTransform( )->SetScale(scale);
```

接下来我们在子对象的ObjectToParentTransform上使用一个偏移量，现在我们就可以使用一个跟它的父代相关的向量[4,3]来对object2进行变换：

```
TransformType::OffsetType Object2ToObject1Offset;
Object2ToObject1Offset[0] = 4;
Object2ToObject1Offset[1] = 3;
object2->GetObjectToParentTransform( )->SetOffset(Object2ToObject1Offset);
```

为了实现前面变换中的操作，我们需要调用ComputeObjectToWorldTransform( )来验算相应的变换：

```
object2->ComputeObjectToWorldTransform( );
```

现在我们可以展示两个对象的ObjectToWorldTransform。需要注意的一点是：由于FixedCenterOfRotationAffineTransform是源自于itk::AffineTransform的，因此这个变换唯一有效的成员是一个矩阵和一个偏移量。作为一个实例，当我们调用Scale( )方式时再计算这个内在矩阵来反映这个变换。

FixedCenterOfRotationAffineTransform执行下面的计算：

$$X'=R•(S•X−C)+C+V \tag{5-1}$$

其中R是旋转矩阵，S是一个比例因子，C是旋转中心，而V是一个变换向量或偏移量。因此定义仿射矩阵M和仿射偏移量T为：

$$M= R•S \tag{5-2}$$

$$T=C+V−R•C \tag{5-3}$$

这就意味着GetScale( )和GetOffset( )像GetMatrix( )一样不能被设置为所期望的值，尤其是对与另外的变换合成得到的变换，因为这个合成是使用这个仿射变换的矩阵和偏移量来完成的。

接下来，我们展示和这两个对象相关的两个仿射变换：

```
std::cout << "object2 IndexToObject Matrix: " << std::endl;
std::cout << object2->GetIndexToObjectTransform( )->GetMatrix( ) << std::endl;
std::cout << "object2 IndexToObject Offset: ";
std::cout << object2->GetIndexToObjectTransform( )->GetOffset( ) << std::endl;
std::cout << "object2 IndexToWorld Matrix: " << std::endl;
std::cout << object2->GetIndexToWorldTransform( )->GetMatrix( ) << std::endl;
std::cout << "object2 IndexToWorld Offset: ";
std::cout << object2->GetIndexToWorldTransform( )->GetOffset( ) << std::endl;
```

然后，我们使用一个[3，3]向量对第二个对象的父对象即第一个对象进行转化。通过设置ObjectToParentTransform的偏移量可以实现。这也可以通过设置ObjectToWorldTransform来实现，因为第一个对象没有父代，因此它是附在直角坐标体系中的：

```
TransformType::OffsetType Object1ToWorldOffset;
Object1ToWorldOffset[0] = 3;
Object1ToWorldOffset[1] = 3;
object1->GetObjectToParentTransform( )->SetOffset(Object1ToWorldOffset);
```

接下来我们对改进的对象调用ComputeObjectToWorldTransform( )，这将在它所有的子代中传播这个变换：

```
object1->ComputeObjectToWorldTransform( );
```

如图5-2展示了我们的变换系统。

最后，我们展示仿射变换的结果：

```
std::cout << "object1 IndexToWorld Matrix: " << std::endl;
std::cout << object1->GetIndexToWorldTransform( )->GetMatrix( ) << std::endl;
std::cout << "object1 IndexToWorld Offset: ";
std::cout << object1->GetIndexToWorldTransform( )->GetOffset( ) << std::endl;
std::cout << "object2 IndexToWorld Matrix: " << std::endl;
std::cout << object2->GetIndexToWorldTransform( )->GetMatrix( ) << std::endl;
std::cout << "object2 IndexToWorld Offset: ";
std::cout << object2->GetIndexToWorldTransform( )->GetOffset( ) << std::endl;
```

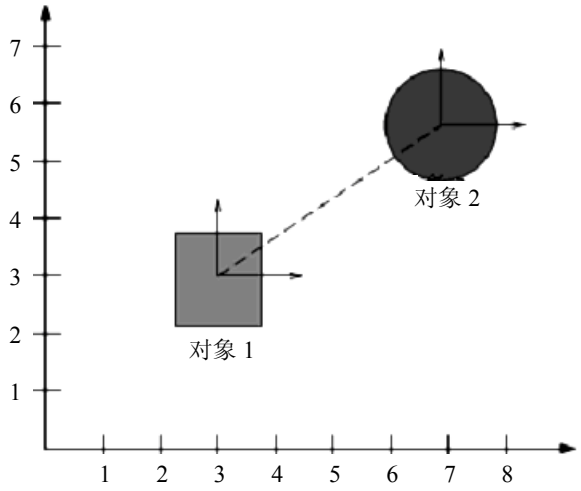


图 5-2　结构中两个对象的物理位置(形状仅为示例之用)。

接下来是第二个例子的输出：

```
object2 IndexToObject Matrix:
2 0
0 2
object2 IndexToObject Offset: 0 0
object2 IndexToWorld Matrix:
2 0
0 2
object2 IndexToWorld Offset: 4 3
object1 IndexToWorld Matrix:
1 0
0 1
object1 IndexToWorld Offset: 3 3
object2 IndexToWorld Matrix:
2 0
0 2
object2 IndexToWorld Offset: 7 6
```

5.5　空间对象类型

本节将详细介绍在ITK中实现的各种各样的空间对象。

5.5.1　ArrowSpatialObject

本小节的源代码在文件Examples/SpatialObjects/ArrowSpatialObject.cxx中。

这个例子展示了如何创建一个itk::ArrowSpatialObject，我们从包含相应的头文件开始：

#include <itkArrowSpatialObject.h>

像许多SpatialObjects一样，itk::ArrowSpatialObject是基于对象的维来进行模板化的：

typedef itk::ArrowSpatialObject<3> ArrowType;

ArrowType::Pointer myArrow = ArrowType::New( );

使用SetLength( )函数来设置坐标体系中arrow的长度，这个长度的默认值设置为1：

myArrow->SetLength(2);

可以使用SetDirection( )函数来设置arrow的方向。SetDirection( )函数更改了ObjectToParentTransform(不是IndexToObjectTransform)。这个方向的默认值是沿着X轴方向的(第一个方向)：

ArrowType::VectorType direction;

direction.Fill(0);

direction[1] = 1.0;

myArrow->SetDirection(direction);

5.5.2 BlobSpatialObject

本小节的源代码在文件Examples/SpatialObjects/BlobSpatialObject.cxx中。

itk::BlobSpatialObject定义了一个N维的blob，像其他SpatialObjects一样，这个类也是源于itk::itkSpatialObject的。一个blob是作为组成对象的一系列点来定义的。

我们从包含相应的头文件开始：

#include "itkBlobSpatialObject.h"

BlobSpatialObject是基于空间的维来模板化的，一个BlobSpatialObject包含一系列SpatialObjectPoints。基本上一个SpatialObjectPoint有一个位置和一种颜色。

#include "itkSpatialObjectPoint.h"

首先我们声明一些类型定义：

typedef itk::BlobSpatialObject<3> BlobType;

typedef BlobType::Pointer BlobPointer;

typedef itk::SpatialObjectPoint<3> BlobPointType;

然后，我们创建一系列的点并使用SetPosition( )方式来设置每个点在当前坐标系中的位置。我们设置每个点的颜色为红色。

BlobType::PointListType list;

for(unsigned int i=0; i<4; i++)

{

BlobPointType p;

p.SetPosition(i,i+1,i+2);

p.SetRed(1);

p.SetGreen(0);

p.SetBlue(0);

p.SetAlpha(1.0);

```
list.push_back(p);
}
```

接下来，我们创建blob并使用SetName( )函数来设置它的名字。我们使用SetId( )来设置它的标识符号，并且我们增加前面创建的点的序列。

```
BlobPointer blob = BlobType::New( );
blob->GetProperty( )->SetName("My Blob");
blob->SetId(1);
blob->SetPoints(list);
```

GetPoints( )方式返回对象中的点的内在序列的一个参考引用：

```
BlobType::PointListType pointList = blob->GetPoints( );
std::cout << "The blob contains " << pointList.size( );
std::cout << " points" << std::endl;
```

然后我们可以使用标准的STL迭代器存储这些点，GetPosition( )和GetColor( )函数分别返回点的位置和颜色。

```
BlobType::PointListType::const_iterator it = blob->GetPoints( ).begin( );
while(it != blob->GetPoints( ).end( ))
{
std::cout << "Position = " << (*it).GetPosition( ) << std::endl;
std::cout << "Color = " << (*it).GetColor( ) << std::endl;
it++;
}
```

5.5.3 CylinderSpatialObject

本小节的源代码在文件Examples/SpatialObjects/CylinderSpatialObject中。

这个例子展示了如何创建一个itk::CylinderSpatialObject，我们从包含相应的头文件开始：

```
#include "itkCylinderSpatialObject.h"
```

一个itk::CylinderSpatialObject只在3维中存在，因此它不用进行模板化：

```
typedef itk::CylinderSpatialObject CylinderType;
```

我们使用标准的智能指针来创建一个圆柱体cylinder：

```
CylinderType::Pointer myCylinder = CylinderType::New( );
```

使用SetRadius( )函数来设置圆柱体cylinder的半径，半径的默认值是1：

```
double radius = 3.0;
myCylinder->SetRadius(radius);
```

使用SetHeight( )函数来设置圆柱体cylinder的高，圆柱体cylinder是默认沿着X轴方向定义的(第一个维)：

```
double height = 12.0;
myCylinder->SetHeight(height);
```

像任何itk::SpatialObjects一样，可以使用IsInside( )函数来查询一个点是在圆柱体cylinder

内部还是在它外部：

```
itk::Point<double,3> insidePoint;
insidePoint[0]=1;
insidePoint[1]=2;
insidePoint[2]=0;
std::cout << "Is my point "<< insidePoint << " inside the cylinder? : "
<< myCylinder->IsInside(insidePoint) << std::endl;
```

我们可以使用Print( )函数来输出圆柱体cylinder：

```
myCylinder->Print(std::cout);
```

5.5.4 EllipseSpatialObject

本小节的源代码在文件Examples/SpatialObjects/EllipseSpatialObject.cxx中。

itk::EllipseSpatialObject定义了一个n维的椭圆ellipse。像其他空间对象一样，这个类源自于itk::SpatialObject。我们从包含相应的头文件开始：

```
#include "itkEllipseSpatialObject.h"
```

像大多数SpatialObjects一样，itk::EllipseSpatialObject是基于空间的维来模板化的。在这个例子中我们创建一个三维椭圆ellipse：

```
typedef itk::EllipseSpatialObject<3> EllipseType;
EllipseType::Pointer myEllipse = EllipseType::New( );
```

然后我们设置每个维的半径，半径设置的默认值为1：

```
EllipseType::ArrayType radius;
for(unsigned int i = 0; i<3; i++)
{
radius[i] = i;
}
myEllipse->SetRadius(radius);
```

如果我们在每个维上有相同的半径，我们可以使用：

```
myEllipse->SetRadius(2.0);
```

然后我们可以使用GetRadius( )函数来显示半径：

```
EllipseType::ArrayType myCurrentRadius = myEllipse->GetRadius( );
std::cout << "Current radius is " << myCurrentRadius << std::endl;
```

像其他SpatialObjects一样，我们可以使用IsInside(itk::Point)函数来查询一个在对象里面的点。这个函数要求点在世界坐标中。

```
itk::Point<double,3> insidePoint;
insidePoint.Fill(1.0);
if(myEllipse->IsInside(insidePoint))
{
std::cout << "The point " << insidePoint;
```

```
std::cout << " is really inside the ellipse" << std::endl;
}
itk::Point<double,3> outsidePoint;
outsidePoint.Fill(3.0);
if(!myEllipse->IsInside(outsidePoint))
{
std::cout << "The point " << outsidePoint;
std::cout << " is really outside the ellipse" << std::endl;
}
```

可以查询所有空间对象在一个点上的值。IsEvaluableAt( )函数返回一个布尔类型符来判断对象在一个特定的点上是否是可估值的：

```
if(myEllipse->IsEvaluableAt(insidePoint))
{
std::cout << "The point " << insidePoint;
std::cout << " is evaluable at the point " << insidePoint << std::endl;
}
```

如果对象在这个特定点上是可估值的，ValueAt( )函数返回在那个点上的当前值。大多数对象返回一个布尔类型符，当点在对象中时为true；当在对象外部时返回false。但是对一些对象，更有意思的是返回一个表达值，例如，距对象中心的距离或到边界的距离。

```
double value;
myEllipse->ValueAt(insidePoint,value);
std::cout << "The value inside the ellipse is: " << value << std::endl;
```

像其他空间对象一样，我们可以使用GetBoundingBox( )来查询对象的边框。边框的结果是在局部框架中表达的。

```
myEllipse->ComputeBoundingBox( );
EllipseType::BoundingBoxType * boundingBox = myEllipse->GetBoundingBox( );
std::cout << "Bounding Box: " << boundingBox->GetBounds( ) << std::endl;
```

5.5.5 GaussianSpatialObject

本小节的源代码在文件Examples/SpatialObjects/GaussianSpatialObject.cxx中。

这个例子展示了如何创建一个itk::GaussianSpatialObject，用来在N维空间中定义一个Gaussian。这个对象非常适合查询物理空间中一个点的值。我们从包含相应的头文件开始：

```
#include "itkGaussianSpatialObject.h"
```

itk::GaussianSpatialObject是基于对象的维来模板化的：

```
typedef itk::GaussianSpatialObject<3> GaussianType;
GaussianType::Pointer myGaussian = GaussianType::New( );
```

使用SetMaximum( )函数来设置Gaussian的最大值：

```
myGaussian->SetMaximum(2);
```

使用SetRadius( )方式来定义Gaussian的半径，这个半径的默认值设置为1.0：

```
myGaussian->SetRadius(3);
```

使用标准ValueAt( )函数确定Gaussian在物理空间中的一个特定点上的值：

```
itk::Point<double,3> pt;
pt[0]=1;
pt[1]=2;
pt[2]=1;
double value;
myGaussian->ValueAt(pt, value);
std::cout << "ValueAt(" << pt << ") = " << value << std::endl;
```

5.5.6 GroupSpatialObject

本小节的源代码在文件Examples/SpatialObjects/GroupSpatialObject.cxx中。

并没有和itk::GroupSpatialObject相关的数据。可以用它来聚合对象或对一个当前对象增加变换。在这个例子中我们展示了如何使用一个GroupSpatialObject。

首先我们包含相应的头文件：

```
#include <itkGroupSpatialObject.h>
```

itk::GroupSpatialObject是基于对象的维来进行模板化的：

```
typedef itk::GroupSpatialObject<3> GroupType;
GroupType::Pointer myGroup = GroupType::New( );
```

接下来，我们创建一个itk::EllipseSpatialObject并将其增加到group：

```
typedef itk::EllipseSpatialObject<3> EllipseType;
EllipseType::Pointer myEllipse = EllipseType::New( );
myEllipse->SetRadius(2);
myGroup->AddSpatialObject(myEllipse);
```

然后我们在每个方向上使用10mm来对group进行变换。因此在物理空间中的椭圆同时也被变换。

```
GroupType::VectorType offset;
offset.Fill(10);
myGroup->GetObjectToParentTransform( )->SetOffset(offset);
myGroup->ComputeObjectToWorldTransform( );
```

然后我们可以使用IsInside( )函数来查询在对象中的点。在这个例子中我们想查看所有的层次结构，因此我们设置深度为2。

```
GroupType::PointType point;
point.Fill(10);
std::cout << "Is my point " << point << " inside?: "
<< myGroup->IsInside(point,2) << std::endl;
```

像其他SpatialObjects一样，我们可以使用RemoveSpatialObject( )方式来从group中移动椭

圆ellipse：

```
myGroup->RemoveSpatialObject(myEllipse);
```

5.5.7 ImageSpatialObject

本小节的源代码在文件Examples/SpatialObjects/ImageSpatialObject.cxx中。

一个itk::ImageSpatialObject包含一个itk::Image，但是增加了空间变换和父子层次结构的概念。我们从包含响应的头文件来开始下一个例子：

```
#include "itkImageSpatialObject.h"
```

首先我们创建一个简单的二维图像image，大小为10×10像素：

```
typedef itk::Image<short,2> Image;
Image::Pointer image = Image::New( );
Image::SizeType size = {{ 10, 10 }};
Image::RegionType region;
region.SetSize(size);
image->SetRegions(region);
image->Allocate( );
```

接下来我们使用渐增的值来填充图像：

```
typedef itk::ImageRegionIterator<Image> Iterator;
Iterator it(image,region);
short pixelValue =0;
it.GoToBegin( );
for(; !it.IsAtEnd( ); ++it, ++pixelValue)
{
it.Set(pixelValue);
}
```

现在我们定义基于图像的维和像素类型模板化的ImageSpatialObject：

```
typedef itk::ImageSpatialObject<2,short> ImageSpatialObject;
ImageSpatialObject::Pointer imageSO = ImageSpatialObject::New( );
```

然后我们使用SetImage( )函数来设置itkImage到ImageSpatialObject：

```
imageSO->SetImage(image);
```

我们可以使用从SpatialObjects继承的IsInside( )、ValueAt( )和DerivativeAt( )函数。当处理注册时，IsInside( )值是非常有用的。

```
typedef itk::Point<double,2> Point;
Point insidePoint;
insidePoint.Fill(9);
if( imageSO->IsInside(insidePoint) )
{
std::cout << insidePoint << " is inside the image." << std::endl;
```

}

ValueAt( )返回最接近的像素值，不加修改地给出一个物理点。

```
double returnedValue;
imageSO->ValueAt(insidePoint,returnedValue);
std::cout << "ValueAt(" << insidePoint << ") = " << returnedValue << std::endl;
```

使用DerivativeAt( )函数可以计算空间中一个指定位置的派生值。第一个变量是我们计算派生值在物理坐标中的点，第二个变量是起源的次序，而第三个变量是以一个itk::Vector表示的结果。派生值是使用有限的方差反复迭代计算的，像ValueAt( )一样，不用任何修改。

```
ImageSpatialObject::OutputVectorType returnedDerivative;
imageSO->DerivativeAt(insidePoint,1,returnedDerivative);
std::cout << "First derivative at " << insidePoint;
std::cout << " = " << returnedDerivative << std::endl;
```

5.5.8 ImageMaskSpatialObject

本小节的源代码在文件Examples/SpatialObjects/ImageMaskSpatialObject.cxx中。

一个itk::ImageMaskSpatialObject是源自于itk::ImageSpatialObject，而且与它非常相似。主要的差别是：如果图像中的像素的亮度为非零的话，则IsInside( )返回true。

支持的像素类型并不包括itk::RGBPixel、itk::RGBAPixeletc等。它只能处理像无符号短字符类型、无符号整型或itk::Vector等简单类型的图像。我们从包含相应的头文件开始：

```
#include "itkImageMaskSpatialObject.h"
```

ImageMaskSpatialObject是基于维来进行模板化的：

```
typedef itk::ImageMaskSpatialObject<3> ImageMaskSpatialObject;
```

接下来我们创建一个大小为50×50×50的itk::Image，除了中心的一个亮的正方形区域以外，其他区域都用零来定义这个模块。

```
typedef ImageMaskSpatialObject::PixelType PixelType;
typedef ImageMaskSpatialObject::ImageType ImageType;
typedef itk::ImageRegionIterator<ImageType> Iterator;
ImageType::Pointer image = ImageType::New( );
ImageType::SizeType size = {{ 50, 50, 50 }};
ImageType::IndexType index = {{ 0, 0, 0 }};
ImageType::RegionType region;
region.SetSize(size);
region.SetIndex(index);
image->SetRegions( region );
image->Allocate( );
PixelType p = itk::NumericTraits< PixelType >::Zero;
image->FillBuffer( p );
```

```
ImageType::RegionType insideRegion;
ImageType::SizeType insideSize = {{ 30, 30, 30 }};
ImageType::IndexType insideIndex = {{ 10, 10, 10 }};
insideRegion.SetSize( insideSize );
insideRegion.SetIndex( insideIndex );
Iterator it( image, insideRegion );
it.GoToBegin( );
while( !it.IsAtEnd( ) )
{
it.Set( itk::NumericTraits< PixelType >::max( ) );
++it;
}
```

然后我们创建一个ImageMaskSpatialObject：

```
ImageMaskSpatialObject::Pointer maskSO = ImageMaskSpatialObject::New( );
```

接着我们将一个相应的指针传递给图像：

```
maskSO->SetImage(image);
```

然后我们就可以测试一个itk::Point是在模块图像里面还是外面。在当图像中仅有一个部分用来计算metric时在注册过程中显得非常有用。

```
ImageMaskSpatialObject::PointType inside;
inside.Fill(20);
std::cout << "Is my point " << inside << " inside my mask? "
<< maskSO->IsInside(inside) << std::endl;
ImageMaskSpatialObject::PointType outside;
outside.Fill(45);
std::cout << "Is my point " << outside << " outside my mask? "
<< !maskSO->IsInside(outside) << std::endl;
```

5.5.9 LandmarkSpatialObject

本小节的源代码在文件Examples/SpatialObjects/LandmarkSpatialObject.cxx中。

itk::LandmarkSpatialObject包含一系列具有位置和颜色的itk::SpatialObjectPoints，我们从包含相应的头文件来开始这个例子：

```
#include "itkLandmarkSpatialObject.h"
```

LandmarkSpatialObject是基于空间的维来模板化的：

这里我们创建一个三维的landmark：

```
typedef itk::LandmarkSpatialObject<3> LandmarkType;
typedef LandmarkType::Pointer LandmarkPointer;
typedef itk::SpatialObjectPoint<3> LandmarkPointType;
LandmarkPointer landmark = LandmarkType::New( );
```

接下来我们设置对象的一些特征，包括它的名字和它的标识号：

```
landmark->GetProperty( )->SetName("Landmark1");
landmark->SetId(1);
```

现在我们对landmark增加点。首先我们创建一系列SpatialObjectPoint并对每个点设置相应的位置和颜色：

```
LandmarkType::PointListType list;
for( unsigned int i=0; i<5; i++)
{
LandmarkPointType p;
p.SetPosition(i,i+1,i+2);
p.SetColor(1,0,0,1);
list.push_back(p);
}
```

然后我们使用SetPoints( )方式对对象增加成员列表：

```
landmark->SetPoints(list);
```

可以使用GetPoints( )方式来访问当前点列表，这种方式返回一个(STL)列表的引用参考：

```
unsigned int nPoints = landmark->GetPoints( ).size( );
std::cout << "Number of Points in the landmark: " << nPoints << std::endl;
LandmarkType::PointListType::const_iterator it = landmark->GetPoints( ).begin( );
while(it != landmark->GetPoints( ).end( ))
{
std::cout << "Position: " << (*it).GetPosition( ) << std::endl;
std::cout << "Color: " << (*it).GetColor( ) << std::endl;
it++;
}
```

5.5.10 LineSpatialObject

本小节的源代码在文件Examples/SpatialObjects/LineSpatialObject.cxx中。

itk::LineSpatialObject定义了n维空间中的一条线(line)。一条线(line)是使用组成线(line)的一系列点来定义的，例如polyline等。我们从包含相应的头文件来开始我们的例子：

```
#include "itkLineSpatialObject.h"
#include "itkLineSpatialObjectPoint.h"
```

LineSpatialObject是基于空间的维来进行模板化的。一个LineSpatialObject包含一系列LineSpatialObjectPoints。一个LineSpatialObjectPoint含有一个位置、n－1条法线和一个颜色。每条法线是用一个大小为N的itk::CovariantVector来表示的。

首先我们定义一些类型，并创建我们的线(line)：

```
typedef itk::LineSpatialObject<3> LineType;
typedef LineType::Pointer LinePointer;
```

```
typedef itk::LineSpatialObjectPoint<3> LinePointType;
typedef itk::CovariantVector<double,3> VectorType;
LinePointer Line = LineType::New( );
```

我们创建一个点序列并使用SetPosition( )方式来设置每个点在当前坐标系中的位置。同样我们也设置每个点的颜色为红色。

使用SetNormal( )函数来设置两条法线。第一个变量是法线本身，而第二个变量是法线的索引。

```
LineType::PointListType list;
for(unsigned int i=0; i<3; i++)
{
LinePointType p;
p.SetPosition(i,i+1,i+2);
p.SetColor(1,0,0,1);
VectorType normal1;
VectorType normal2;
for(unsigned int j=0;j<3;j++)
{
normal1[j]=j;
normal2[j]=j*2;
}
p.SetNormal(normal1,0);
p.SetNormal(normal2,1);
list.push_back(p);
}
```

接下来我们使用SetName( )来设置对象的名字。同样地我们也使用SetId( )来设置它的标识号并设置前面创建的点的列表：

```
Line->GetProperty( )->SetName("Line1");
Line->SetId(1);
Line->SetPoints(list);
```

GetPoints( )方式返回对象中点的内在列表的一个引用：

```
LineType::PointListType pointList = Line->GetPoints( );
std::cout << "Number of points representing the line: ";
std::cout << pointList.size( ) << std::endl;
```

然后我们可以使用STL的迭代器来访问这些点。GetPosition( )和GetColor( )函数分别返回点的位置和颜色。使用GetNormal(unsigned int)函数我们可以访问每条法线。

```
LineType::PointListType::const_iterator it = Line->GetPoints( ).begin( );
while(it != Line->GetPoints( ).end( ))
{
std::cout << "Position = " << (*it).GetPosition( ) << std::endl;
```

```
std::cout << "Color = " << (*it).GetColor( ) << std::endl;
std::cout << "First normal = " << (*it).GetNormal(0) << std::endl;
std::cout << "Second normal = " << (*it).GetNormal(1) << std::endl;
std::cout << std::endl;
it++;
}
```

5.5.11 MeshSpatialObject

本小节的源代码在文件Examples/SpatialObjects/MeshSpatialObject.cxx中。

虽然一个itk::MeshSpatialObject只包含itk::Mesh的一个指针，但是增加了空间变换和父子层次结构的概念。这个例子展示了如何创建一个itk::MeshSpatialObject，并使用它来形成一个二值图像和如何将网格写在磁盘上。

我们从包含相应的头文件开始：

```
#include <itkMeshSpatialObject.h>
#include <itkSpatialObjectReader.h>
#include <itkSpatialObjectWriter.h>
#include <itkSpatialObjectToImageFilter.h>
```

MeshSpatialObject隐含了一个itk::Mesh，因此我们首先创建一个网格：

```
typedef itk::DefaultDynamicMeshTraits< float , 3, 3 > MeshTrait;
typedef itk::Mesh<float,3,MeshTrait> MeshType;
typedef MeshType::CellTraits CellTraits;
typedef itk::CellInterface< float, CellTraits > CellInterfaceType;
typedef itk::TetrahedronCell<CellInterfaceType> TetraCellType;
typedef MeshType::PointType PointType;
typedef MeshType::CellType CellType;
typedef CellType::CellAutoPointer CellAutoPointer;
MeshType::Pointer myMesh = MeshType::New( );
MeshType::CoordRepType testPointCoords[4][3]
= { {0,0,0}, {9,0,0}, {9,9,0}, {0,0,9} };
unsigned long tetraPoints[4] = {0,1,2,4};
int i;
for(i=0; i < 4 ; ++i)
{
myMesh->SetPoint(i, PointType(testPointCoords[i]));
}
myMesh->SetCellsAllocationMethod(
MeshType::CellsAllocatedDynamicallyCellByCell );
CellAutoPointer testCell1;
```

```
testCell1.TakeOwnership( new TetraCellType );
testCell1->SetPointIds(tetraPoints);
myMesh->SetCell(0, testCell1 );
```

然后我们创建一个MeshSpatialObject，它是基于前面定义的网格类型模板化的。

```
typedef itk::MeshSpatialObject<MeshType> MeshSpatialObjectType;
MeshSpatialObjectType::Pointer myMeshSpatialObject =
MeshSpatialObjectType::New( );
```

而且将网格指针指向MeshSpatialObject：

```
myMeshSpatialObject->SetMesh(myMesh);
```

使用GetMesh( )函数可以得到指向网格的当前指针：

```
myMeshSpatialObject->GetMesh( );
```

像其他任何SpatialObjects一样，可以使用GetBoundingBox( )、ValueAt( )、IsInside( )函数得到重要的信息。

```
std::cout << "Mesh bounds : " <<
myMeshSpatialObject->GetBoundingBox( )->GetBounds( ) << std::endl;
MeshSpatialObjectType::PointType myPhysicalPoint;
myPhysicalPoint.Fill(1);
std::cout << "Is my physical point inside? : " <<
myMeshSpatialObject->IsInside(myPhysicalPoint) << std::endl;
```

现在我们已经定义了MeshSpatialObject，我们可以使用itk::SpatialObjectWriter来保存当前网格。为了做到这一点，我们需要指定我们写的网格类型：

```
typedef itk::SpatialObjectWriter<3,float,MeshTrait> WriterType;
WriterType::Pointer writer = WriterType::New( );
```

然后我们设置网格空间对象和文件的名字并调用Update( )函数：

```
writer->SetInput(myMeshSpatialObject);
writer->SetFileName("myMesh.meta");
writer->Update( );
```

使用itk::SpatialObjectReader来读取保存的网格。我们需要再次指定想要读取的网格的类型。

```
typedef itk::SpatialObjectReader<3,float,MeshTrait> ReaderType;
ReaderType::Pointer reader = ReaderType::New( );
```

下面设置我们想要读取的文件的名字并调用update：

```
reader->SetFileName("myMesh.meta");
reader->Update( );
```

接下来我们展示如何使用itk::SpatialObjectToImageFilter来创建MeshSpatialObject的一个二值图像。结果图像中网格里面应该为1，而外面应该为0。首先我们定义并实例化SpatialObjectToImageFilter：

```
typedef itk::Image<unsigned char,3> ImageType;
typedef itk::GroupSpatialObject<3> GroupType;
```

typedef itk::SpatialObjectToImageFilter< GroupType, ImageType >

SpatialObjectToImageFilterType;

SpatialObjectToImageFilterType::Pointer imageFilter =

SpatialObjectToImageFilterType::New( );

然后我们传递reader的输出(例如MeshSpatialObject)给这个滤波器：

imageFilter->SetInput(reader->GetGroup( ));

最后我们通过调用Update( )方式来触发滤波器的执行。注意：计算时间是依赖于网格的大小而变化的。

imageFilter->Update( );

然后我们可以使用GetOutput( )函数来得到二值图像结果。

ImageType::Pointer myBinaryMeshImage = imageFilter->GetOutput( );

5.5.12 SurfaceSpatialObject

本小节的源代码在文件Examples/SpatialObjects/SurfaceSpatialObject.cxx中。

itk::SurfaceSpatialObject定义了n维空间中的一个表面(surface)。通过表面(surface)上的一系列点来定义一个SurfaceSpatialObject。每个点都具有一个位置和一条独特的法线。下面通过包含相应的头文件来开始这个例子：

#include "itkSurfaceSpatialObject.h"

#include "itkSurfaceSpatialObjectPoint.h"

SurfaceSpatialObject是基于空间的维模板化的。一个SurfaceSpatialObject包含一系列SurfaceSpatialObjectPoints。一个SurfaceSpatialObjectPoint具有一个位置、一条法线和一个颜色。

首先我们定义一些类型定义：

typedef itk::SurfaceSpatialObject<3> SurfaceType;

typedef SurfaceType::Pointer SurfacePointer;

typedef itk::SurfaceSpatialObjectPoint<3> SurfacePointType;

typedef itk::CovariantVector<double,3> VectorType;

SurfacePointer Surface = SurfaceType::New( );

我们创建一个点列表并使用SetPosition( )方式设置每个点在当前坐标系中的位置。同样，我们将每个点的颜色也设置为红色。

SurfaceType::PointListType list;

for(unsigned int i=0; i<3; i++)

{

SurfacePointType p;

p.SetPosition(i,i+1,i+2);

p.SetColor(1,0,0,1);

VectorType normal;

for(unsigned int j=0;j<3;j++)

```
{
normal[j]=j;
}
p.SetNormal(normal);
list.push_back(p);
}
```

接下来我们创建表面(surface)并使用SetName( )来设置它的名字。我们也使用SetId( )来设置它的标识号并增加前面创建的点列表：

```
Surface->GetProperty( )->SetName("Surface1");
Surface->SetId(1);
Surface->SetPoints(list);
```

GetPoints( )方式返回对象中点的内在列表的一个引用：

```
SurfaceType::PointListType pointList = Surface->GetPoints( );
std::cout << "Number of points representing the surface: ";
std::cout << pointList.size( ) << std::endl;
```

然后我们使用STL迭代器来访问这些点。GetPosition( )和GetColor( )函数分别返回点的位置和颜色。GetNormal( )返回一条itk::CovariantVector形式的法线。

```
SurfaceType::PointListType::const_iterator it = Surface->GetPoints( ).begin( );
while(it != Surface->GetPoints( ).end( ))
{
std::cout << "Position = " << (*it).GetPosition( ) << std::endl;
std::cout << "Normal = " << (*it).GetNormal( ) << std::endl;
std::cout << "Color = " << (*it).GetColor( ) << std::endl;
std::cout << std::endl;
it++;
}
```

5.5.13 TubeSpatialObject

itk::TubeSpatialObject 代表了使用 SpatialObjects 表示的管状结构的一个基类。itk::VesselTubeSpatialObject 和 itk::DTITubeSpatialObject 类都源自于这个基类。VesselTubeSpatialObject表示从一幅图像中提取的血管，而DTITubeSpatialObject表示从磁共振弥散张量成像得到的神经纤维束。

本小节的源代码在文件Examples/SpatialObjects/TubeSpatialObject.cxx中。

itk::TubeSpatialObject定义了一个n维的管(tube)。一个管(tube)是以一系列中心线的点来定义的，每个点具有一个位置、一个半径、几条法线和其他特征。我们从包含相应的头文件开始：

```
#include "itkTubeSpatialObject.h"
#include "itkTubeSpatialObjectPoint.h"
```

TubeSpatialObject是基于空间的维来模板化的。一个TubeSpatialObject包含了一系列的TubeSpatialObjectPoints。

首先我们定义一些类型并创建tube：

```
typedef itk::TubeSpatialObject<3> TubeType;
typedef TubeType::Pointer TubePointer;
typedef itk::TubeSpatialObjectPoint<3> TubePointType;
typedef TubePointType::CovariantVectorType VectorType;
TubePointer tube = TubeType::New( );
```

我们创建一个点列表并做以下设置：

(1)使用SetPosition( )方式设置每个点在当前坐标系中的位置。

(2)使用SetRadius( )来设置tube在这个位置上的半径。

(3)使用SetNormal1( )和SetNormal2( )来设置两条法线。

(4)在下面的例子中我们将点的颜色设置为红色。

```
TubeType::PointListType list;
for( i=0; i<5; i++)
{
TubePointType p;
p.SetPosition(i,i+1,i+2);
p.SetRadius(1);
VectorType normal1;
VectorType normal2;
for(unsigned int j=0;j<3;j++)
{
normal1[j]=j;
normal2[j]=j*2;
}
p.SetNormal1(normal1);
p.SetNormal2(normal2);
p.SetColor(1,0,0,1);
list.push_back(p);
}
```

接下来我们创建管(tube)并使用SetName( )设置它的名字。我们也使用SetId( )来设置它的标识号并在后面增加前面创建的点列表：

```
tube->GetProperty( )->SetName("Tube1");
tube->SetId(1);
tube->SetPoints(list);
```

GetPoints( )方式返回对象中点的内在列表的一个引用：

```
TubeType::PointListType pointList = tube->GetPoints( );
std::cout << "Number of points representing the tube: ";
```

```
std::cout << pointList.size( ) << std::endl;
```

ComputeTangentAndNormals( )函数使用微分来计算每个点的法线和切线：

```
tube->ComputeTangentAndNormals( );
```

然后我们可以使用STL迭代器来访问点。GetPosition( )和GetColor( )函数分别返回点的位置和颜色。GetRadius( )返回在那个点的半径，GetNormal1( )函数返回一个itk::CovariantVector而GetTangent( )返回一个itk::Vector。

```
TubeType::PointListType::const_iterator it = tube->GetPoints( ).begin( );
i=0;
while(it != tube->GetPoints( ).end( ))
{
std::cout << std::endl;
std::cout << "Point #" << i << std::endl;
std::cout << "Position: " << (*it).GetPosition( ) << std::endl;
std::cout << "Radius: " << (*it).GetRadius( ) << std::endl;
std::cout << "Tangent: " << (*it).GetTangent( ) << std::endl;
std::cout << "First Normal: " << (*it).GetNormal1( ) << std::endl;
std::cout << "Second Normal: " << (*it).GetNormal2( ) << std::endl;
std::cout << "Color = " << (*it).GetColor( ) << std::endl;
it++;
i++;
}
```

5.5.14 VesselTubeSpatialObject

这部分的源代码在文件Examples/SpatialObjects/VesselTubeSpatialObject.cxx中。

itk::VesselTubeSpatialObject源自于itk::TubeSpatialObject。它表示从一幅图像分割而得到的一个血管。一个VesselTubeSpatialObject是以一系列点来描述的，每个点都具有一个位置、一个半径和法线。

我们从包含相应的头文件开始：

```
#include "itkVesselTubeSpatialObject.h"
#include "itkVesselTubeSpatialObjectPoint.h"
```

VesselTubeSpatialObject是基于空间的维来模板化的。一个VesselTubeSpatialObject包含一系列VesselTubeSpatialObjectPoints。

首先我们定义一些类型并创建管(tube)：

```
typedef itk::VesselTubeSpatialObject<3> VesselTubeType;
typedef itk::VesselTubeSpatialObjectPoint<3> VesselTubePointType;
VesselTubeType::Pointer VesselTube = VesselTubeType::New( );
```

我们创建一个点列表并做以下设置：

(1)使用SetPosition( )方式设置每个点在当前坐标系中的位置。

(2)使用SetRadius( )设置tube在这个位置上的半径。

(3)使用SetMedialness( )设置的值描述点是如何分布在脉管中间的。

(4)使用SetRidgeness( )设置的值描述点是如何分布在山脊上的。

(5)使用SetBranchness( )设置的值来描述一个支脉点。

(6)使用SetAlpha1( )、SetAlpha2( )和SetAlpha3( )来设置与Hessian的特征值相关的3个α值。

(7)使用SetMark( )设置标记(mark)值。

(8)在这个例子中点的颜色设置为红色，且不透明性为1。

```
VesselTubeType::PointListType list;
for( i=0; i<5; i++)
{
VesselTubePointType p;
p.SetPosition(i,i+1,i+2);
p.SetRadius(1);
p.SetAlpha1(i);
p.SetAlpha2(i+1);
p.SetAlpha3(i+2);
p.SetMedialness(i);
p.SetRidgeness(i);
p.SetBranchness(i);
p.SetMark(true);
p.SetColor(1,0,0,1);
list.push_back(p);
}
```

接下来我们创建管(tube)并使用SetName( )设置它的名字。我们也使用SetId( )来设置它的标识号，并在后面增加前面创建的点列表。

```
VesselTube->GetProperty( )->SetName("VesselTube");
VesselTube->SetId(1);
VesselTube->SetPoints(list);
```

GetPoints( )方式返回对象中点内在列表的一个引用：

```
VesselTubeType::PointListType pointList = VesselTube->GetPoints( );
std::cout << "Number of points representing the blood vessel: ";
std::cout << pointList.size( ) << std::endl;
```

然后我们可以使用STL迭代器来访问点。GetPosition( )和GetColor( )函数分别返回点的位置和颜色。

```
VesselTubeType::PointListType::const_iterator
it = VesselTube->GetPoints( ).begin( );
i=0;
while(it != VesselTube->GetPoints( ).end( ))
```

```
{
std::cout << std::endl;
std::cout << "Point #" << i << std::endl;
std::cout << "Position: " << (*it).GetPosition( ) << std::endl;
std::cout << "Radius: " << (*it).GetRadius( ) << std::endl;
std::cout << "Medialness: " << (*it).GetMedialness( ) << std::endl;
std::cout << "Ridgeness: " << (*it).GetRidgeness( ) << std::endl;
std::cout << "Branchness: " << (*it).GetBranchness( ) << std::endl;
std::cout << "Mark: " << (*it).GetMark( ) << std::endl;
std::cout << "Alpha1: " << (*it).GetAlpha1( ) << std::endl;
std::cout << "Alpha2: " << (*it).GetAlpha2( ) << std::endl;
std::cout << "Alpha3: " << (*it).GetAlpha3( ) << std::endl;
std::cout << "Color = " << (*it).GetColor( ) << std::endl;
it++;
i++;
}
```

5.5.15 DTITubeSpatialObject

这部分的源代码在文件Examples/SpatialObjects/DTITubeSpatialObject.cxx中。

itk::DTITubeSpatialObject源自于itk::TubeSpatialObject，它表示从磁共振弥散张量成像得到的神经纤维束。一个DTITubeSpatialObject是由中心线上的一系列点来描述的，每个点都具有一个位置、一个半径、法线、各向异性分数(FA)值、ADC(表观扩散系数)值、各向异性测量(GA)值、特征值和类似全张量矩阵的向量。

我们从包含相应的头文件开始：

```
#include "itkDTITubeSpatialObject.h"
#include "itkDTITubeSpatialObjectPoint.h"
```

DTITubeSpatialObject是基于空间的维模板化的。一个DTITubeSpatialObject包含一系列的DTITubeSpatialObjectPoints。

首先我们定义一些类型定义并创建管(tube)：

```
typedef itk::DTITubeSpatialObject<3> DTITubeType;
typedef itk::DTITubeSpatialObjectPoint<3> DTITubePointType;
DTITubeType::Pointer dtiTube = DTITubeType::New( );
```

我们创建一个点列表并做以下设置：

(1)使用SetPosition( )方式设置每个点在当前坐标系中的位置。

(2)使用SetRadius( )设置tube在这个位置上的半径。

(3)使用AddField(DTITubePointType::FA)设置FA值。

(4)使用AddField(DTITubePointType::ADC)设置ADC值。

(5)使用AddField(DTITubePointType::GA)设置GA值。

(6)使用SetTensorMatrix( )设置全张量矩阵，假设是一个确定的正值。

(7)在这个例子中设置点的颜色为红色。

```
DTITubeType::PointListType list;
for( i=0; i<5; i++)
{
DTITubePointType p;
p.SetPosition(i,i+1,i+2);
p.SetRadius(1);
p.AddField(DTITubePointType::FA,i);
p.AddField(DTITubePointType::ADC,2*i);
p.AddField(DTITubePointType::GA,3*i);
p.AddField("Lambda1",4*i);
p.AddField("Lambda2",5*i);
p.AddField("Lambda3",6*i);
float* v = new float[6];
for(unsigned int k=0;k<6;k++)
{
v[k] = k;
}
p.SetTensorMatrix(v);
delete v;
p.SetColor(1,0,0,1);
list.push_back(p);
}
```

接下来我们创建管(tube)，并使用SetName( )设置它的名字。我们也使用SetId( )来设置它的标识号，并在后面增加前面创建的点列表：

```
dtiTube->GetProperty( )->SetName("DTITube");
dtiTube->SetId(1);
dtiTube->SetPoints(list);
```

GetPoints( )方式返回对象中点的内在列表的一个引用：

```
DTITubeType::PointListType pointList = dtiTube->GetPoints( );
std::cout << "Number of points representing the fiber tract: ";
std::cout << pointList.size( ) << std::endl;
```

然后我们可以使用STL迭代器来访问点。GetPosition( )和GetColor( )函数分别返回点的位置和颜色。

```
DTITubeType::PointListType::const_iterator it = dtiTube->GetPoints( ).begin( );
i=0;
while(it != dtiTube->GetPoints( ).end( ))
{
```

```
std::cout << std::endl;
std::cout << "Point #" << i << std::endl;
std::cout << "Position: " << (*it).GetPosition( ) << std::endl;
std::cout << "Radius: " << (*it).GetRadius( ) << std::endl;
std::cout << "FA: " << (*it).GetField(DTITubePointType::FA) << std::endl;
std::cout << "ADC: " << (*it).GetField(DTITubePointType::ADC) << std::endl;
std::cout << "GA: " << (*it).GetField(DTITubePointType::GA) << std::endl;
std::cout << "Lambda1: " << (*it).GetField("Lambda1") << std::endl;
std::cout << "Lambda2: " << (*it).GetField("Lambda2") << std::endl;
std::cout << "Lambda3: " << (*it).GetField("Lambda3") << std::endl;
std::cout << "TensorMatrix: " << (*it).GetTensorMatrix( )[0] << " : ";
std::cout << (*it).GetTensorMatrix( )[1] << " : ";
std::cout << (*it).GetTensorMatrix( )[2] << " : ";
std::cout << (*it).GetTensorMatrix( )[3] << " : ";
std::cout << (*it).GetTensorMatrix( )[4] << " : ";
std::cout << (*it).GetTensorMatrix( )[5] << std::endl;
std::cout << "Color = " << (*it).GetColor( ) << std::endl;
it++;
i++;
}
```

5.6 SceneSpatialObject

本节的源代码在文件Examples/SpatialObjects/SceneSpatialObject.cxx中。

这个例子描述了如何使用itk::SceneSpatialObject。一个SceneSpatialObject包含一个SpatialObjects集。下面从包含相应的头文件开始这个例子：

```
#include "itkSceneSpatialObject.h"
```

一个SceneSpatialObject是基于空间的维模板化的，需要由SceneSpatialObject引用的对象都具有相同的维。

首先我们定义一些类型，并创建SceneSpatialObject：

```
typedef itk::SceneSpatialObject<3> SceneSpatialObjectType;
SceneSpatialObjectType::Pointer scene = SceneSpatialObjectType::New( );
```

然后我们创建两个itk::EllipseSpatialObjects：

```
typedef itk::EllipseSpatialObject<3> EllipseType;
EllipseType::Pointer ellipse1 = EllipseType::New( );
ellipse1->SetRadius(1);
ellipse1->SetId(1);
EllipseType::Pointer ellipse2 = EllipseType::New( );
```

ellipse2->SetId(2);

ellipse2->SetRadius(2);

然后我们在SceneSpatialObject中增加两个椭圆ellipses：

scene->AddSpatialObject(ellipse1);

scene->AddSpatialObject(ellipse2);

我们可以使用GetNumberOfObjects( )来查询在SceneSpatialObject中的对象的数量。这个函数有两个可选择的变量：计算对象数量的层数（默认设置为无穷大）和计算对象的名字（默认设置为NULL）。例如，这允许用户仅仅计算椭圆ellipses。

std::cout << "Number of objects in the SceneSpatialObject = ";

std::cout << scene->GetNumberOfObjects( ) << std::endl;

GetObjectById( )返回在SceneSpatialObject中的第一个对象，具有指定的标识号。

std::cout << "Object in the SceneSpatialObject with an ID == 2: " << std::endl;

scene->GetObjectById(2)->Print(std::cout);

使用RemoveSpatialObject( )函数也可以从SceneSpatialObject中移出对象：

scene->RemoveSpatialObject(ellipse1);

使用GetObjects( )方式可以得到在SceneSpatialObject中的当前对象。像GetNumberOfObjects( )方式一样，GetObjects( )有两个变量：搜索深度和一个相关名字。

SceneSpatialObjectType::ObjectListType * myObjectList = scene->GetObjects( );

std::cout << "Number of objects in the SceneSpatialObject = ";

std::cout << myObjectList->size( ) << std::endl;

在一些案例中使用ParentId( )定义层次结构和定义当前的标识号是很有用的。这将导致在SceneSpatialObject中有一系列单调的SpatialObjects。因此，SceneSpatialObject提供了FixHierarchy( )方式基于标识号来识别父子层次关系：

scene->FixHierarchy( );

可以使用Clear( )函数来清除屏幕：

scene->Clear( );

5.7 读/写 SpatialObjects

本节的源代码在文件Examples/SpatialObjects/ReadWriteSpatialObject.cxx中。

读和写SpatialObjects是一个相对简单的任务。分别使用itk::SpatialObjectReader和itk::SpatialObjectWriter来读和写这些对象（注意：这些类使用MetaIO来辅助常规的I/O，因此含有一个.meta的后缀文件）。

我们从包含相应的头文件来开始这个例子：

#include "itkSpatialObjectWriter.h"

#include "itkSpatialObjectReader.h"

接下来我们创建一个SpatialObjectWriter，基于我们想要写的对象的维来进行模板化：

typedef itk::SpatialObjectWriter<3> WriterType;

WriterType::Pointer writer = WriterType::New( );

对于这个例子，我们创建一个itk::EllipseSpatialObject：

typedef itk::EllipseSpatialObject<3> EllipseType;

EllipseType::Pointer ellipse = EllipseType::New( );

ellipse->SetRadius(3);

最后，我们使用SetInput( )方式将对象放置到writer，并使用SetFileName( )来设置文件的名字，使用Update( )方式来写入信息：

writer->SetInput(ellipse);

writer->SetFileName("ellipse.meta");

writer->Update( );

现在我们准备打开刚创建的对象。首先我们创建一个SpatialObjectReader，也是基于文件中对象的维来进行模板化的。这就意味着文件必须仅仅包含相同维的对象。

typedef itk::SpatialObjectReader<3> ReaderType;

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

接下来我们使用SetFileName( )方式来设置读取的文件名，并调用Update( )方式来读取文件：

reader->SetFileName("ellipse.meta");

reader->Update( );

可以调用GetScene( )或GetGroup( )方式来得到文件中的对象。GetScene( )返回一个指向itk::SceneSpatialObject的指针：

ReaderType::SceneType * scene = reader->GetScene( );

std::cout << "Number of objects in the scene: ";

std::cout << scene->GetNumberOfObjects( ) << std::endl;

ReaderType::GroupType * group = reader->GetGroup( );

std::cout << "Number of objects in the group: ";

std::cout << group->GetNumberOfChildren( ) << std::endl;

5.8 通过 SpatialObjects 进行统计计算

本节的源代码在文件Examples/SpatialObjects/SpatialObjectToImageStatisticsCalculator.cxx中。

这个例子描述了如何使用itk::SpatialObjectToImageStatisticsCalculator来计算定义在一个给定的itk::SpatialObject里面的一个区域中的一个itk::Image的统计。

#include "itkSpatialObjectToImageStatisticsCalculator.h"

首先我们使用itk::RandomImageSource来创建一个测试图像：

typedef itk::Image<unsigned char,2> ImageType;

typedef itk::RandomImageSource<ImageType> RandomImageSourceType;

RandomImageSourceType::Pointer randomImageSource = RandomImageSourceType::

New( );

unsigned long size[2];

size[0] = 10;

size[1] = 10;

randomImageSource->SetSize(size);

randomImageSource->Update( );

ImageType::Pointer image = randomImageSource->GetOutput( );

接下来我们创建一个半径为2的itk::EllipseSpatialObject。我们也通过增加IndexToObjectTransform的偏移量来将一个椭圆ellipse移动到图像的中心：

typedef itk::EllipseSpatialObject<2> EllipseType;

EllipseType::Pointer ellipse = EllipseType::New( );

ellipse->SetRadius(2);

EllipseType::VectorType offset;

offset.Fill(5);

ellipse->GetIndexToObjectTransform( )->SetOffset(offset);

ellipse->ComputeObjectToParentTransform( );

然后我们创建itk::SpatialObjectToImageStatisticsCalculator。

typedef itk::SpatialObjectToImageStatisticsCalculator<

ImageType, EllipseType > CalculatorType;

CalculatorType::Pointer calculator = CalculatorType::New( );

我们传递一个指向指针的指针给计数器calculator：

calculator->SetImage(image);

我们也传递了SpatialObject。统计将在SpatialObject里面进行计算（在计数器calculator内部使用IsInside( )方式）：

calculator->SetSpatialObject(ellipse);

最后我们通过调用Update( )函数来触发计算，并且我们可以使用GetMean( )和GetCovarianceMatrix( )分别得到矩阵的均值和方差：

calculator->Update( );

std::cout << "Sample mean = " << calculator->GetMean( ) << std::endl ;

std::cout << "Sample covariance = " << calculator->GetCovarianceMatrix( );

第六章　滤　　波

本章将介绍研发平台中最常用的滤波器。这些滤波器大部分是用来处理图像的。它们可以有一个或多个输入，并将产生一个或多个输出。ITK 是基于一个数据管道体系结构的，其中一个滤波器的输出可以作为输入传递给另一个滤波器（参考 3.5 节可以得到更多信息）

6.1　门限处理

门限处理(Thresholding)操作是基于指定一个或多个值（称为阈值，Threshold）来改变或确定像素值的。接下来的内容将描述如何使用 ITK 来实现门限处理（Thresholding）操作。

6.1.1　二值门限处理

本小节的源代码在文件 Examples/Filtering/BinaryThresholdImageFilter.cxx 中。

这个例子阐述了二值阈值图像滤波器的用法。这个滤波器通过在如图 6-1 所示的规则下改变像素值来将一个图像转化成一个二值图像。用户需要指定两个门限：上门限和下门限，也就是两个亮度值：内部和外部。对于输入图像中的每个像素，使用上、下门限来和像素值进行比较。如果像素值在由[下门限，上门限]定义的范围内，则输出像素指定为 InsideValue；否则输出像素指定为 OutsideValue。门限处理通常作为一个分割途径的最后一个操作。

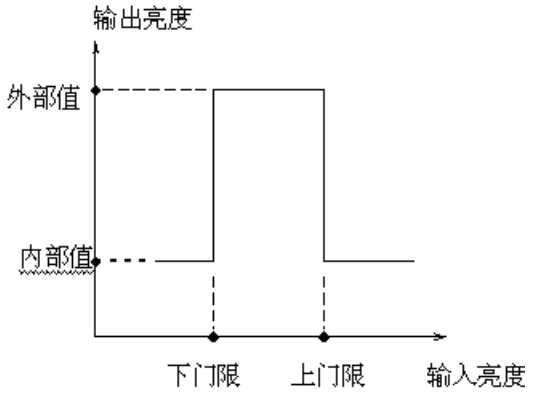


图 6-1　BinaryThresholdImageFilter 的转换函数

使用 itk::BinaryThresholdImageFilter 需要的第一步是包含它的头文件：

```
#include "itkBinaryThresholdImageFilter.h"
```

下一步是确定输入和输出图像的像素类型：

```
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
```

现在分别使用像素类型和维来定义输入和输出图像的类型：

typedef itk::Image< InputPixelType, 2 > InputImageType;

typedef itk::Image< OutputPixelType, 2 > OutputImageType;

使用前面定义的输入和输出图像的类型来对滤波器的类型进行实例化：

typedef itk::BinaryThresholdImageFilter<

InputImageType, OutputImageType > FilterType;

同样也实例化一个 itk::ImageFileReader 类来从一个文件中读取图像（参考第七章可以得到更多关于读写数据的信息）：

typedef itk::ImageFileReader< InputImageType > ReaderType;

实例化一个 itk::ImageFileWriter 以便将输出图像写到一个文件中：

typedef itk::ImageFileWriter< InputImageType > WriterType;

滤波器和 reader 都是通过调用它们的 New( )方式来创建的，并将结果指向 itk::SmartPointers：

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

FilterType::Pointer filter = FilterType::New( );

由 reader 得到的图像作为输入传递给 BinaryThresholdImageFilter：

filter->SetInput(reader->GetOutput( ));

SetOutsideValue( )方式定义了指向那些亮度值在上、下门限范围之外的像素的亮度值。SetInsideValue( )方式定义了指向那些亮度值在上、下门限范围之内的像素的亮度值：

inside the threshold range.

filter->SetOutsideValue(outsideValue);

filter->SetInsideValue(insideValue);

SetLowerThreshold( )和 SetUpperThreshold( )方式定义了将转化为 InsideValue 的输入图像亮度范围。注意上、下门限值的类型是输入图像像素类型，而 InsideValue 和 OutsideValue 的类型是输出图像的像素类型。

filter->SetLowerThreshold(lowerThreshold);

filter->SetUpperThreshold(upperThreshold);

通过调用 Update( )方式来触发滤波器的运行。如果滤波器的输出作为输入传递给后面的滤波器，Update( )将调用传递途径中任何较后的滤波器直接触发这个滤波器的更新。

filter->Update( );

图 6-2 举例说明了这个滤波器在一个脑部 MRI 质子浓度图像作用的效果。这个图表展示了这个滤波器实现分割时自身的局限性。这些局限性在噪声图像和类似于 MRI 情况由于场偏磁造成的缺少空间均匀性的图像中显得尤为明显。

下面的类提供了相似的功能：

- itk::ThresholdImageFilter

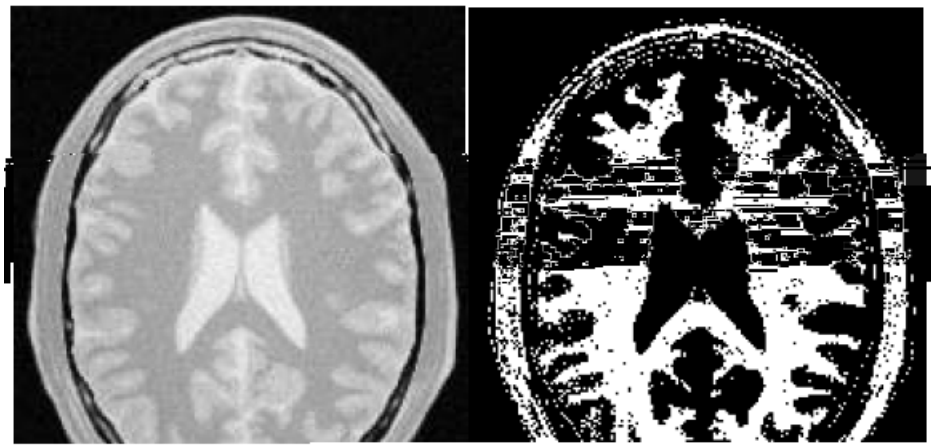


图 6-2　BinaryThresholdImageFilter 对一个脑部 MRI 质子浓度图像作用的效果

6.1.2　门限处理概要

本小节的源代码在文件 Examples/Filtering/ThresholdImageFilter.cxx 中。

这个例子阐述了 itk::ThresholdImageFilter 的用法。这个滤波器可以使用三种不同的方式来转化一个图像的亮度级。

第一种：用户定义一个单一的阈值。任何像素值在这个阈值以下的像素将由一个用户定义的值来代替，这里成为 OutsideValue。像素值在这个阈值以上的像素将保持不变。如图 6-3 所示，举例说明了这种类型的门限处理。

第二种：用户定义一个特定的阈值，所有像素值在这个阈值以上的像素将由 OutsideValue 来代替。像素值在这个阈值以下的像素将保持不变。如图 6-4 所示。

第三种：用户定义两个门限。所有像素值在这两个门限定义的范围之内的像素将保持不变，像素值在这个范围以外的像素将指定为 OutsideValue，如图 6-5 所示。

接下来是从滤波器的这三种操作模式中选择一种方式：

(1)ThresholdBelow( )

(2)ThresholdAbove( )

(3)ThresholdOutside( )

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkThresholdImageFilter.h"
```

然后我们必须确定使用图像的像素类型。这个滤波器是基于一个单一的图像类型来模板化的，因为这个算法仅仅确定了在指定范围之外的像素值，且保持不变来传递剩余的值。

```
typedef unsigned char PixelType;
```

使用像素类型和维来定义图像：

```
typedef itk::Image< PixelType, 2 > ImageType;
```

使用前面定义的图像类型来对滤波器进行实例化：

```
typedef itk::ThresholdImageFilter< ImageType > FilterType;
```

同样也实例化一个 itk::ImageFileReader 类来从一个文件中读取图像：

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

实例化一个 itk::ImageFileWriter 以便将输出图像写到一个文件中：

```
typedef itk::ImageFileWriter< ImageType > WriterType;
```

滤波器和 reader 都是通过调用它们的 New( )方式来创建的，并将结果指向 itk::SmartPointers：

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

由 reader 得到的图像作为输入传递给 itk::ThresholdImageFilter：

```
filter->SetInput( reader->GetOutput( ) );
```

SetOutsideValue( )方式定义了指向那些亮度值在上、下门限范围之外的像素的亮度值：

```
filter->SetOutsideValue( 0 );
```

ThresholdBelow( )方式定义了输入图像中要改变为 OutsideValue 的像素的亮度值：

```
filter->ThresholdBelow( 180 );
```

通过调用 Update( )方式来触发滤波器的运行。如果滤波器的输出作为输入传递给后面的滤波器，Update( )将调用传递途径中任何较后的滤波器直接触发这个滤波器的更新。

```
filter->Update( );
```

这个例子的输出如图 6-3 所示。第二种操作模式可以通过调用 ThresholdAbove( )方式来实现。

```
filter->ThresholdAbove( 180 );
filter->Update( );
```

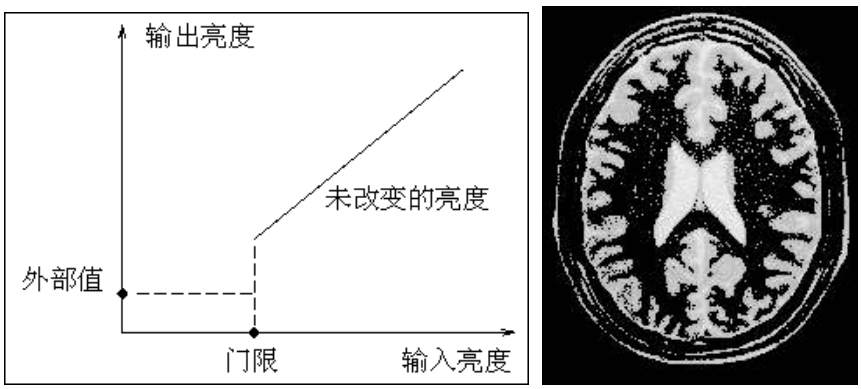


图 6-3　ThresholdImageFilter 使用改变低于门限的亮度值的模式

使用新的设置更新这个滤波器将产生如图 6-4 所示的输出。第三种操作模式可以通过调用 ThresholdOutside( )来实现。

```
filter->ThresholdOutside( 170,190 );
filter->Update( );
```

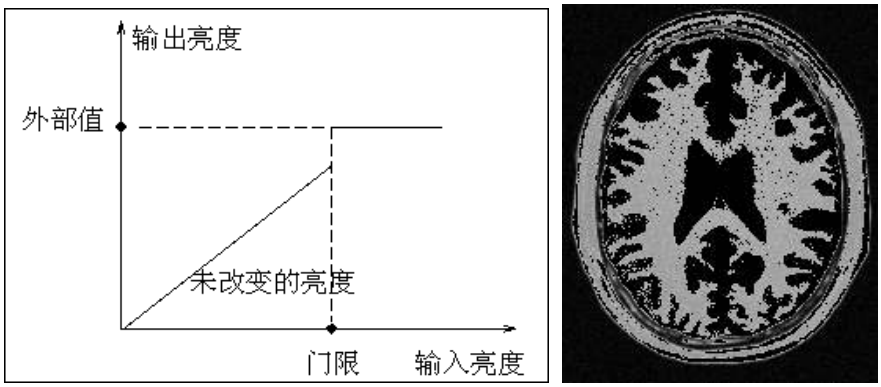


图 6-4　ThresholdImageFilter 使用改变高于门限的亮度值的模式

第三种模式“带通”门限处理模式的输出如图 6-5 所示。

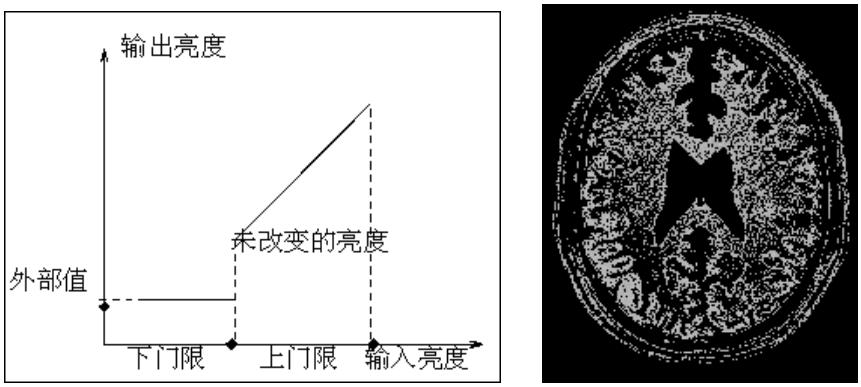


图 6-5　ThresholdImageFilter 使用改变门限外亮度值的模式

本小节的例子也说明了这个滤波器实现分割时自身的局限性。这些局限性在噪声图像和类似于 MRI 情况由于场偏磁造成的缺少空间均匀性的图像中显得尤为明显。

下面的类提供了相似的功能：

- itk::BinaryThresholdImageFilter

6.2　边缘检测

Canny 边缘检测

本小节的源代码在文件 Examples/Filtering/CannyEdgeDetectionImageFilter.cxx 中。

这个例子介绍了 itk::CannyEdgeDetectionImageFilter 的用法。这个滤波器由于它灵敏度高、定位精确和抗噪声能力强可以得到最佳的解决方案而广泛应用在边缘检测中。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkCannyEdgeDetectionImageFilter.h"
```

这个滤波器对浮点型数据类型的像素类型图像进行操作，然后必须转换通常是整型的图

像类型。这里使用 itk::CastImageFilter 来达到这一目的。定义它的图像模板参数用来把输入图像类型转换为用于处理的浮点型。

typedef itk::CastImageFilter< CharImageType, RealImageType> CastToRealFilterType;

使用浮点型图像类型来对 itk::CannyEdgeDetectionImageFilter 进行实例化。

6.3 投射和亮度映射

本节介绍的滤波器实现 pixel-wise 亮度映射。投射用来将一种像素类型转换成另一种，而亮度映射也用来计算不同像素类型的亮度范围。

6.3.1 线性映射

本小节的源代码在文件 Examples/Filtering/CastingImageFilters.cxx 中。

由于在研发平台中使用范型编程，所以大多数类型都由编译时间决定，很少有关于类型变换的决定留到运行时间解决的。由用户将像素类型转换为数据流水线所需要的像素类型。在医学图像应用中，通常不会使用一种通用的像素类型，因为这样将导致有价值信息的流失。

本小节介绍流经流水线的图像的外在投射的机制。下面将处理接下来的四个滤波器：itk::CastImageFilter 、 itk::RescaleIntensityImageFilter 、 itk::ShiftScaleImageFilter 和 itk::NormalizeImageFilter。这些滤波器除了它们都可以修改像素值之外，互相之间并不直接相关。这里将它们放在一起是为了比较它们之间各自的特征。

CastImageFilter 是一个非常简单的滤波器，它对输入图像进行 pixel-wise，将每个像素投射到输出图像类型。注意这个滤波器并不对亮度执行任何算法操作。应用 CastImageFilter 等价于对每个像素执行一个 C 类型的投射。

outputPixel = static cast<OutputPixelType>(inputPixel)

RescaleIntensityImageFilter 以一种将输入的最大值、最小值映射到用户提供的最大值、最小值的方式来线性地度量像素值。这是强制将图像的动态范围转换为适合于图像显示常见的一个特定范围内的一个普遍过程。这个滤波器应用的变换可以表达为：

$$\text{输出像素} = (\text{输入像素} - \text{输入最小值}) \times \frac{(\text{输出最大值} - \text{输出最小值})}{(\text{输入最大值} - \text{输入最小值})} + \text{输出最小值}$$

ShiftScaleImageFilter 同样应用一个输入图像亮度的线性变换，不过这个变换是由用户以增加一个值并乘一个乘数因子的形式来指定的。可以表达为：

$$\text{输出像素} = (\text{输入像素} + \text{偏移量}) \times \text{缩放比例因子}$$

由 NormalizeImageFilter 应用的线性变换的参数是在中心计算的，因此输出图像灰度级的统计分布的均值为 0，方差为 1。这个亮度修正在作为相互信息度量的估计的一个预处理步骤的注册应用中特别有用。NormalizeImageFilter 的线性变换可以表达为：

$$\text{输出像素} = \frac{(\text{输入像素} - \text{均值})}{\sqrt{\text{方差}}}$$

和通常一样，使用这些滤波器的第一步需要包含它们的头文件：

#include "itkCastImageFilter.h"

```
#include "itkRescaleIntensityImageFilter.h"
#include "itkShiftScaleImageFilter.h"
#include "itkNormalizeImageFilter.h"
```

我们定义输入和输出图像的像素类型：

```
typedef unsigned char InputPixelType;
typedef float OutputPixelType;
```

然后，定义输入和输出图像类型：

```
typedef itk::Image< InputPixelType, 3 > InputImageType;
typedef itk::Image< OutputPixelType, 3 > OutputImageType;
```

使用定义的图像类型来对滤波器进行实例化：

```
typedef itk::CastImageFilter<
InputImageType, OutputImageType > CastFilterType;
typedef itk::RescaleIntensityImageFilter<
InputImageType, OutputImageType > RescaleFilterType;
typedef itk::ShiftScaleImageFilter<
InputImageType, OutputImageType > ShiftScaleFilterType;
typedef itk::NormalizeImageFilter<
InputImageType, OutputImageType > NormalizeFilterType;
```

通过调用 New( )操作来创建对象滤波器并将结果指向 itk::SmartPointers：

```
CastFilterType::Pointer castFilter = CastFilterType::New( );
RescaleFilterType::Pointer rescaleFilter = RescaleFilterType::New( );
ShiftScaleFilterType::Pointer shiftFilter = ShiftScaleFilterType::New( );
NormalizeFilterType::Pointer normalizeFilter = NormalizeFilterType::New( );
```

现在将一个 reader 滤波器(这里并未展示它的创建)的输出作为输入连接到投射滤波器：

```
castFilter->SetInput( reader->GetOutput( ) );
shiftFilter->SetInput( reader->GetOutput( ) );
rescaleFilter->SetInput( reader->GetOutput( ) );
normalizeFilter->SetInput( reader->GetOutput( ) );
```

接下来我们设置每个滤波器需要的参数。CastImageFilter 和 NormalizeImageFilter 不需要任何参数。另一方面，RescaleIntensityImageFilter 需要用户提供想得到的输出图像像素值的最大最小值，这可以通过使用 SetOutputMinimum( )和 SetOutputMaximum( )方式来实现，如下所示：

```
rescaleFilter->SetOutputMinimum( 10 );
rescaleFilter->SetOutputMaximum( 250 );
```

ShiftScaleImageFilter 需要一个乘数因子（比例）（一个传递比例的附加值（移位）。可以分别使用 SetScale( )和 SetShift( )方式来设置这些值：

```
shiftFilter->SetScale( 1.2 );
shiftFilter->SetShift( 25 );
```

最后，通过调用 Update( )方式来运行滤波器：

```
castFilter->Update( );
shiftFilter->Update( );
rescaleFilter->Update( );
normalizeFilter->Update( );
```

6.3.2 非线性映射

接下来的滤波器可以被看成是投射滤波器的一个变量。它的主要特性是使用一个平滑、连续的非线性形式的变换函数。

本小节的源代码在文件 Examples/Filtering/SigmoidImageFilter.cxx 中。

itk::SigmoidImageFilter 通常作为一个亮度变换使用。它通过在亮度值的一个特定范围的边界的一个非常平滑连续的转变将这个范围映射到一个新的亮度范围，是广泛使用 Sigmoids 来作为关注值的一个设置并逐渐削弱范围之外的值的一个机制。为了扩展 Sigmoids 滤波器的机动性，使用四个参数通过选择它的输入、输出亮度范围来调节它在 ITK 中的执行。接下来的方程表达了 Sigmoids 亮度变换：

$$I' = (\mathrm{Max} - \mathrm{Min}) \cdot \frac{1}{(1 + e^{-(\frac{1-\beta}{\partial})})} + \mathrm{Min} \tag{6-1}$$

在上面的方程中，I 是输入像素的亮度，I′是输出像素的亮度，Min、Max 是输出图像的最小值和最大值，α 定义了输入亮度范围的宽度，β 定义了围绕在范围中心的亮度。如图 6-6 所示阐述了每个参数的意义。

这个滤波器可以对任何维的图像使用并在必要时可以使用多处理。

首先必须包含与这个滤波器相关的头文件：

```
#include "itkSigmoidImageFilter.h"
```

然后必须定义这个滤波器的输入、输出像素和图像类型：

```
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

我们使用图像类型来实例化滤波器类型并创建滤波器对象。

```
typedef itk::SigmoidImageFilter<InputImageType, OutputImageType > SigmoidFilterType;
SigmoidFilterType::Pointer sigmoidFilter = SigmoidFilterType::New( );
```

输出中的最小值和最大值分别使用 SetOutputMinimum( )和 SetOutputMaximum( )方式来定义。

```
sigmoidFilter->SetOutputMinimum( outputMinimum );
sigmoidFilter->SetOutputMaximum( outputMaximum );
```

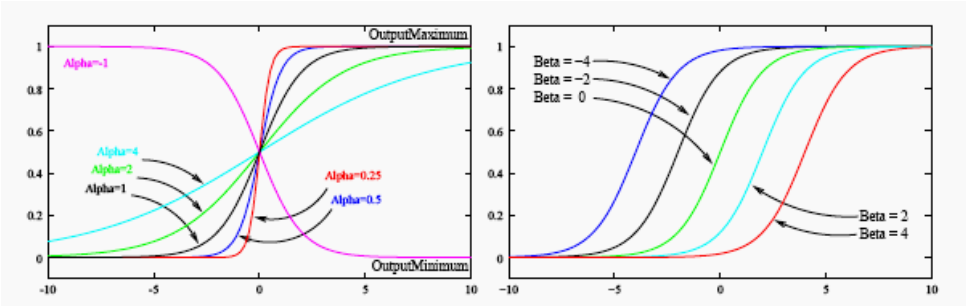


图 6-6　SigmoidImageFilter 使用不同参数的效果。参数 α 定义了窗口的宽度；参数 β 定义了窗口的亮度

使用 SetAlpha( )和 SetBeta( )来设置系数 α 和 β。注意 α 是和输入亮度窗口成比例的。按照惯例我们可以说这个窗口是间距[−3α，3α]。亮度窗口的边界并不明显。如图 6-6 所示，α 平稳地接近它的极值。当你想通过在围绕人口均值周围定义一个间距[−3σ，3σ]来设置一个人口测量的范围时，你可以同样的形式来进行考虑。

```
sigmoidFilter->SetAlpha( alpha );
sigmoidFilter->SetBeta( beta );
```

可以从其他滤波器得到 SigmoidImageFilter 的输入，例如一个图像文件 reader。输出可以像一个图像文件 writer 一样传递给其他滤波器流水线。任何下游的滤波器调用 update 都可以触发 Sigmoids 滤波器的运行。

```
sigmoidFilter->SetInput( reader->GetOutput( ) );
writer->SetInput( sigmoidFilter->GetOutput( ) );
writer->Update( );
```

如图 6-7 所示阐述了这个滤波器使用下面参数对一个 MRI 脑部图像切片作用的效果：

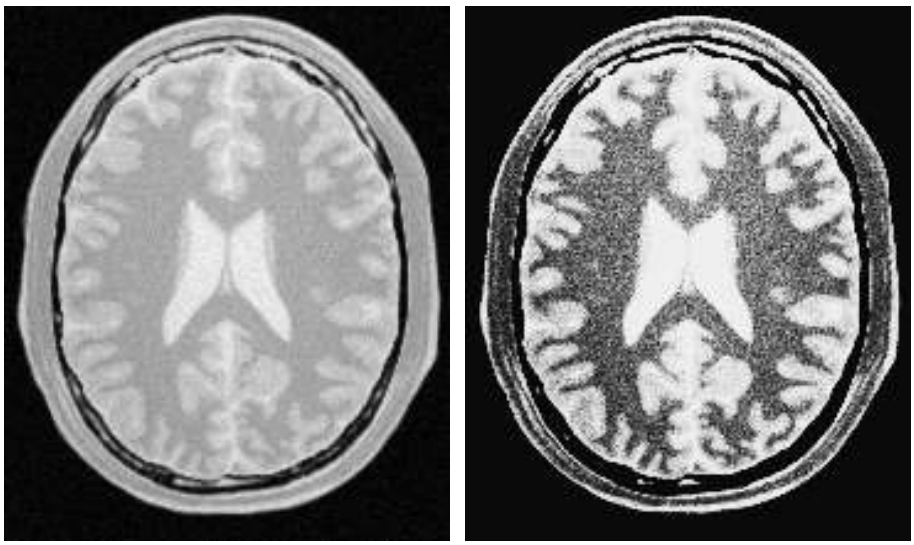


图 6-7　Sigmoid 滤波器对一个 MRI 脑部图像切片作用的效果

- 最小值=10
- 最大值=240

- α = 10
- β = 170

从图片中我们可以看到，白质部分的亮度扩展到它们的动态范围，而亮度值在 β−3α 之下和在 β+3α 之上的值分别映射到输出值的最小值和最大值。这是一个 Sigmoids 用来执行平滑亮度窗口时使用的方式。

注意：α 和 β 都是可正可负的。一个负的 α 将对图像有相反的效果。这可以在图 6-6 中左边的图像中看到。在 9.3.1 小节中表达了一个使用 Sigmoids 滤波器作为分割的预处理的应用实例。

在现实中 Sigmoids 曲线是很常见的。它们表达了图对一个刺激的灵敏性。它们也是高斯积分曲线，因此自然地可以作为高斯分布信号的一个响应。

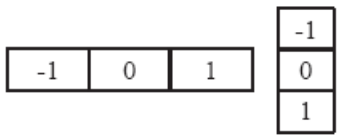
6.4 梯度

梯度计算是图像处理中的一个常见操作。梯度在一些背景下表示梯度向量而在其他情况下可以表示梯度向量的大小。ITK 滤波器在涉及这个概念时通过强度这个概念来区别该不明确概念。ITK 提供可以计算图像梯度向量和图像强度大小的滤波器。

6.4.1 梯度强度

本小节的源代码在文件 Examples/Filtering/GradientMagnitudeImageFilter.cxx 中。

图像梯度的强度广泛地应用在图像分析中，主要用来帮助检测对象轮廓和分离均匀区域。itk::GradientMagnitudeImageFilter 使用一个简单的有限差分方式来计算图像中每个像素位置的梯度强度。例如：在二维情况下计算等同于将图像使用模块类型，如下所示：



然后计算它们的平方和并计算和的平方根。

由于在内部使用了 itk::NeighborhoodIterator 和 itk::NeighborhoodOperator，所以这个滤波器可以对任何维的图像进行操作。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkGradientMagnitudeImageFilter.h"
```

必须选择输入和输出图像的像素类型：

```
typedef float InputPixelType;
typedef float OutputPixelType;
```

使用像素类型可以定义输入、输出图像类型：

```
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

通过定义输入输出图像类型来定义梯度强度的类型：

typedef itk::GradientMagnitudeImageFilter<

InputImageType, OutputImageType > FilterType;

通过调用 New( )方式来创建一个滤波器对象并将结果指向一个 itk::SmartPointer：

FilterType::Pointer filter = FilterType::New( );

输入图像可以从另一个滤波器的输出得到。这里，源于一个图像 reader 来得到：

filter->SetInput(reader->GetOutput( ));

最后，通过调用 Update( )方式来执行滤波器：

filter->Update( );

如果这个滤波器的输出已经连接到流水线中的其他滤波器，更新任何下游的滤波器将同样触发这个滤波器的一个更新。例如，梯度强度滤波器可能连接到一个图像 writer：

rescaler->SetInput(filter->GetOutput( ));

writer->SetInput(rescaler->GetOutput( ));

writer->Update( );

如 6-8 所示阐述了梯度强度滤波器对一个 MRI 脑部图像切片作用的效果。这个图片展示了这个滤波器对噪声数据的灵敏性。

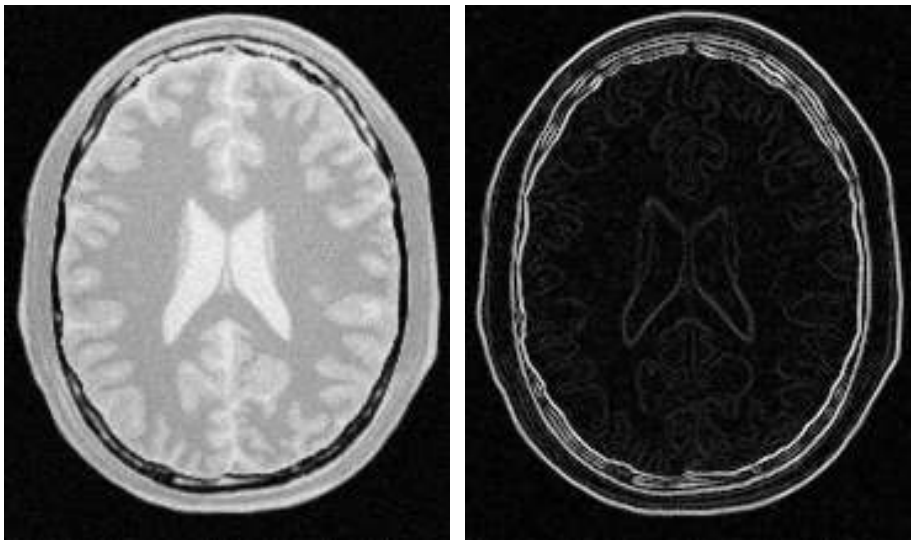


图 6-8　GradientMagnitudeImageFilter 对一个 MRI 脑部图像切片作用的效果

由于梯度强度图像的动态范围往往比输入图像的动态范围要小，所以必须注意选择用来表达输出图像的图像类型。通常，这个规则会产生异常，例如，合成图像包含高对比对象。

这个滤波器在计算梯度前不会对图像进行任何滤波。因此这个结果对噪声非常敏感，而且并不一定是尺度空间分析的最佳选择。

6.4.1　带滤波的梯度强度

本小节的源代码在文件 Examples/Filtering/GradientMagnitudeRecursiveGaussianImageFilter.cxx 中。

微分是对一个数字数据的不规则操作。实际中可以方便地定义一个执行微分的比例。在

执行这样的滤波时使用一个高斯核被认为是最便捷的选择。通过选择一个特定的高斯标准差(σ)，就可以选择一个相应的比例来去除通常被认为是噪声的高频部分。

itk::GradientMagnitudeRecursiveGaussianImageFilter 计算在每个像素的图像梯度。这个计算过程等同于首先通过将图像和一个高斯核卷积来平滑图像，然后应用一个差分操作。s 的值是由用户选择的。

这些是通过使用一个 IIR（Infinite Impulse Response 无限脉冲响应）和高斯核的派生卷积来实现的。传统的卷积将产生一个更精确的结果，但是 IIR 方式更加迅速，尤其是使用大的σ 值。

GradientMagnitudeRecursiveGaussianImageFilter 通过使用高斯核和它的导函数的自然分离来对任何维的图像进行操作。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkGradientMagnitudeRecursiveGaussianImageFilter.h"
```

类型必须基于输入和输出图像的像素进行实例化：

```
typedef float InputPixelType;
typedef float OutputPixelType;
```

使用它们就可以对输入、输出图像进行实例化。

```
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在同时使用输入和输出图像类型来实例化滤波器类型：

```
typedef itk::GradientMagnitudeRecursiveGaussianImageFilter<
InputImageType, OutputImageType > FilterType;
```

通过调用 New( )方式来创建一个滤波器对象并将结果指向一个 itk::SmartPointer。

```
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从另一个滤波器的输出得到。这里，源自于一个图像 reader 来得到。

```
filter->SetInput( reader->GetOutput( ) );
```

现在设置高斯滤波核的标准差。

```
filter->SetSigma( sigma );
```

最后，通过调用 Update( )方式来执行滤波器：

```
filter->Update( );
```

如果这个滤波器的输出已经连接到流水线中的其他滤波器，更新任何下游的滤波器将同样触发这个滤波器的一个更新。例如，梯度强度滤波器可能连接到一个图像 writer：

```
rescaler->SetInput( filter->GetOutput( ) );
writer->SetInput( rescaler->GetOutput( ) );
writer->Update( );
```

如图 6-9 所示阐述了这个滤波器对一个 MRI 脑部图像切片作用的效果，使用的 σ 值分别为 3（左图）、5（右图）。这个图片展示了可以通过选择一个合　　打

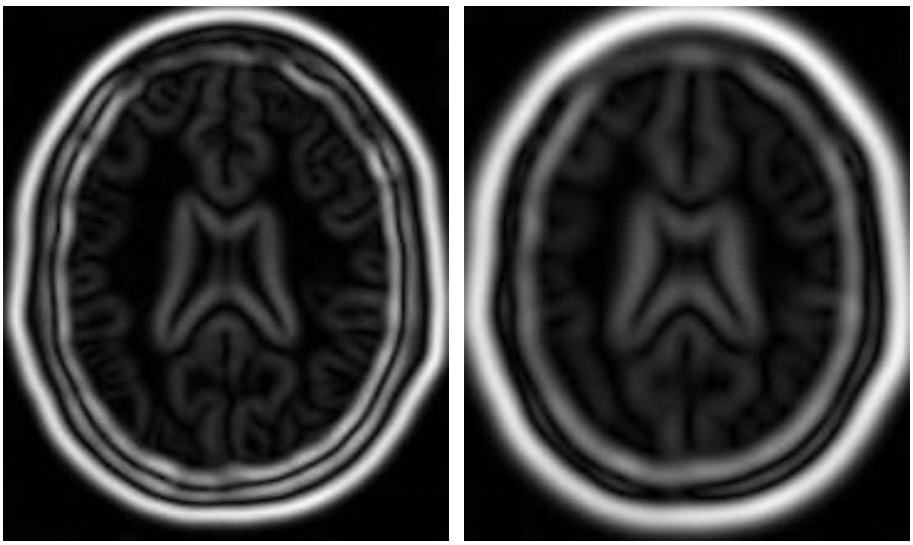


图 6-9　GradientMagnitudeRecursiveGaussianImageFilter 对一个 MRI 脑部图像切片作用的效果

由于梯度强度图像的动态范围往往比输入图像的动态范围要小，所以必须注意选择用来表达输出图像的图像类型。

6.4.2　不带滤波的导函数

本小节的源代码在文件 Examples/Filtering/DerivativeImageFilter.cxx 中。

使用 itk::DerivativeImageFilter 来计算一幅图像的偏微分——图像沿一个特定的坐标轴方向上的微分。

首先必须包含这个滤波器相应的头文件：

```
#include "itkDerivativeImageFilter.h"
```

接下来，必须定义输入、输出图像的像素类型，并使用它们实例化图像类型。注意：由于导数值是可正可负的，所以选择的图像应具有符号类型是很重要的。

```
typedef float InputPixelType;
typedef float OutputPixelType;
const unsigned int Dimension = 2;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

现在就可以使用图像类型来定义滤波器类型并创建滤波器对象：

```
typedef itk::DerivativeImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

使用 SetOrder( )方式来选择微分的阶数。使用 SetDirection( )方式来选择计算微分的坐标方向：

```
filter->SetOrder( atoi( argv[4] ) );
filter->SetDirection( atoi( argv[5] ) );
```

可以从任何其他滤波器得到这个滤波器的输入，例如一个 reader。输出可以像一个 writer 一样传递给其他滤波器流水线。任何下游的滤波器调用 update 都可以触发微分滤波器的运行。

```
filter->SetInput( reader->GetOutput( ) );
writer->SetInput( filter->GetOutput( ) );
writer->Update( );
```

图 6-10 所示阐述了 DerivativeImageFilter 对一个 MRI 脑部图像作用的效果。微分是沿着 *x* 轴方向来计算的。从这个结果可以看出对图像中的噪声的敏感度是很明显的。

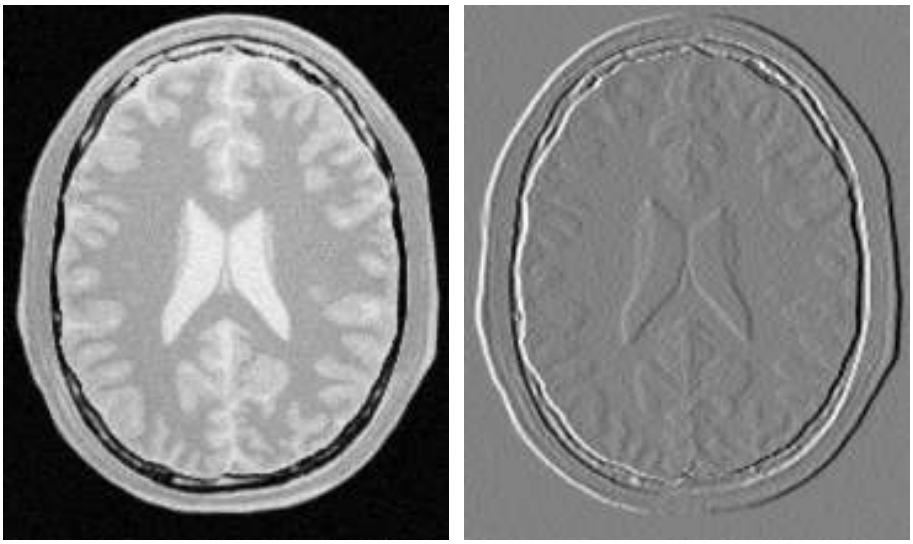


图 6-10　DerivativeFilter 对一个 MRI 脑部图像作用的效果

6.5　二阶微分

6.5.1　二阶高斯递归

本小节的源代码在文件 Examples/Filtering/SecondDerivativeRecursiveGaussianImageFilter. cxx 中。

这个例子阐述了如何使用 itk::RecursiveGaussianImageFilter 来计算一个三维图像的二阶微分。

在这个例子中，所有的二阶微分都是用相同的方式独自计算的，就像使用它们来创建图像的海塞矩阵一样：

```
#include "itkRecursiveGaussianImageFilter.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkImageDuplicator.h"
#include "itkImage.h"
```

```
#include <string>
int main(int argc, char * argv [ ] )
{
if( argc < 3 )
{
std::cerr << "Usage: " << std::endl;
std::cerr << "SecondDerivativeRecursiveGaussianImageFilter inputImage outputPrefix
[sigma] " << std::return EXIT_FAILURE;
}
typedef float PixelType;
typedef float OutputPixelType;
const unsigned int Dimension = 3;
typedef itk::Image< PixelType, Dimension > ImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
typedef itk::ImageFileReader< ImageType > ReaderType;
typedef itk::ImageFileWriter< OutputImageType > WriterType;
typedef itk::ImageDuplicator< OutputImageType > DuplicatorType;
typedef itk::RecursiveGaussianImageFilter<
ImageType,
ImageType > FilterType;
ReaderType::Pointer reader = ReaderType::New( );
WriterType::Pointer writer = WriterType::New( );
DuplicatorType::Pointer duplicator = DuplicatorType::New( );
reader->SetFileName( argv[1] );
std::string outputPrefix = argv[2];
std::string outputFileName;
try
{
reader->Update( );
}
catch( itk::ExceptionObject & excp )
{
std::cerr << "Problem reading the input file" << std::endl;
std::cerr << excp << std::endl;
return EXIT_FAILURE;
}
FilterType::Pointer ga = FilterType::New( );
FilterType::Pointer gb = FilterType::New( );
FilterType::Pointer gc = FilterType::New( );
```

```
ga->SetDirection( 0 );
gb->SetDirection( 1 );
gc->SetDirection( 2 );
if( argc > 3 )
{
const float sigma = atof( argv[3] );
ga->SetSigma( sigma );
gb->SetSigma( sigma );
gc->SetSigma( sigma );
}
ga->SetZeroOrder( );
gb->SetZeroOrder( );
gc->SetSecondOrder( );
ImageType::Pointer inputImage = reader->GetOutput( );
ga->SetInput( inputImage );
gb->SetInput( ga->GetOutput( ) );
gc->SetInput( gb->GetOutput( ) );
duplicator->SetInputImage( gc->GetOutput( ) );
gc->Update( );
duplicator->Update( );
ImageType::Pointer Izz = duplicator->GetOutput( );
writer->SetInput( Izz );
outputFileName = outputPrefix + "-Izz.mhd";
writer->SetFileName( outputFileName.c_str( ) );
writer->Update( );
gc->SetDirection( 1 ); // gc now works along Y
gb->SetDirection( 2 ); // gb now works along Z
gc->Update( );
duplicator->Update( );
ImageType::Pointer Iyy = duplicator->GetOutput( );
writer->SetInput( Iyy );
outputFileName = outputPrefix + "-Iyy.mhd";
writer->SetFileName( outputFileName.c_str( ) );
writer->Update( );
gc->SetDirection( 0 ); // gc now works along X
ga->SetDirection( 1 ); // ga now works along Y
gc->Update( );
duplicator->Update( );
ImageType::Pointer Ixx = duplicator->GetOutput( );
```

```
writer->SetInput( Ixx );
outputFileName = outputPrefix + "-Ixx.mhd";
writer->SetFileName( outputFileName.c_str( ) );
writer->Update( );
ga->SetDirection( 0 );
gb->SetDirection( 1 );
gc->SetDirection( 2 );
ga->SetZeroOrder( );
gb->SetFirstOrder( );
gc->SetFirstOrder( );
gc->Update( );
duplicator->Update( );
ImageType::Pointer Iyz = duplicator->GetOutput( );
writer->SetInput( Iyz );
outputFileName = outputPrefix + "-Iyz.mhd";
writer->SetFileName( outputFileName.c_str( ) );
writer->Update( );
ga->SetDirection( 1 );
gb->SetDirection( 0 );
gc->SetDirection( 2 );
ga->SetZeroOrder( );
gb->SetFirstOrder( );
gc->SetFirstOrder( );
gc->Update( );
duplicator->Update( );
ImageType::Pointer Ixz = duplicator->GetOutput( );
writer->SetInput( Ixz );
outputFileName = outputPrefix + "-Ixz.mhd";
writer->SetFileName( outputFileName.c_str( ) );
writer->Update( );
ga->SetDirection( 2 );
gb->SetDirection( 0 );
gc->SetDirection( 1 );
ga->SetZeroOrder( );
gb->SetFirstOrder( );
gc->SetFirstOrder( );
gc->Update( );
duplicator->Update( );
ImageType::Pointer Ixy = duplicator->GetOutput( );
```

```
writer->SetInput( Ixy );
outputFileName = outputPrefix + "-Ixy.mhd";
writer->SetFileName( outputFileName.c_str( ) );
writer->Update( );
```

6.5.2 拉普拉斯滤波器

本小节的源代码在文件 Examples/Filtering/LaplacianRecursiveGaussianImageFilter1.cxx 中。

这个例子阐述了如何使用 itk::RecursiveGaussianImageFilter 来计算一个二维图像的拉普拉斯算子。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkRecursiveGaussianImageFilter.h"
```

需要选择输入、输出像素类型：

```
typedef float InputPixelType;
typedef float OutputPixelType;
```

使用像素类型来实例化输入、输出图像类型：

```
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在同时使用输入、输出图像类型实例化滤波器类型：

```
typedef itk::RecursiveGaussianImageFilter<
InputImageType, OutputImageType > FilterType;
```

这个滤波器应用在一个单一方向上的近似卷积。因此连接几个这种滤波器来在所有方向上生成滤波是很有必要的。在这个例子中，我们创建一对滤波器以便于处理一个 2 维图像。通过调用 New( )方式来创建滤波器并将结果指向一个 itk::SmartPointer。

我们需要两个滤波器来计算拉普拉斯算子的 X 分量并用另两个滤波器来计算 Y 分量。

```
FilterType::Pointer filterX1 = FilterType::New( );
FilterType::Pointer filterY1 = FilterType::New( );
FilterType::Pointer filterX2 = FilterType::New( );
FilterType::Pointer filterY2 = FilterType::New( );
```

由于新创建的滤波器中每个都有可能沿任何维来执行滤波的潜力，所以我们将每个滤波器限定在一个特定的方向上。可以通过 SetDirection( )方式来完成：

```
filterX1->SetDirection( 0 ); // 0 --> X direction
filterY1->SetDirection( 1 ); // 1 --> Y direction
filterX2->SetDirection( 0 ); // 0 --> X direction
filterY2->SetDirection( 1 ); // 1 --> Y direction
```

itk::RecursiveGaussianImageFilter 可以逼近于它和高斯函数或和它的一阶、二阶导数的卷积。我们通过使用 SetOrder( )方式来选择这些选项。注意这个变量是一个可以为零阶、一阶和二阶值的枚举。例如：为了计算 x 的偏导数，我们选择 x 为一阶而 y 为零阶。这里我们

仅要对 x 和 y 进行平滑，所以我们选择两个方向上都为零阶：

```
filterX1->SetOrder( FilterType::ZeroOrder );
filterY1->SetOrder( FilterType::SecondOrder );
filterX2->SetOrder( FilterType::SecondOrder );
filterY2->SetOrder( FilterType::ZeroOrder );
```

有两种常见的标准化高斯函数的方式，都取决于它们的应用。对于尺度空间分析，使用一个标准化是用来阻止输入的最大值的。这个标准化可以用下面的式子表示：

$$\frac{1}{\sigma\sqrt{2\pi}} \tag{6-2}$$

在应用中使用高斯函数作为扩散方程的一个解，使用标准化函数是为了阻止信号的积分。这个最后的方式可以看成是一个质量守恒原理。这可以用下面的式子来表示：

$$\frac{1}{\sigma^2\sqrt{2\pi}} \tag{6-3}$$

itk::RecursiveGaussianImageFilter 有一个布尔标识，允许用户在这两个标准化之间进行选择。这个选择是使用 SetNormalizeAcrossScale( )方式来完成的。可以使用这个标识来分析一个跨越尺度空间的图像。在当前的例子中，这个设置并没有影响，因为我们实际上对 reader 输出的动态范围进行了再标准化。因此我们可以忽略这个标识。

```
const bool normalizeAcrossScale = false;
filterX1->SetNormalizeAcrossScale( normalizeAcrossScale );
filterY1->SetNormalizeAcrossScale( normalizeAcrossScale );
filterX2->SetNormalizeAcrossScale( normalizeAcrossScale );
filterY2->SetNormalizeAcrossScale( normalizeAcrossScale );
```

输入图像可以从另一个滤波器的输出得到。这里，我们用一个图像 reader 来作为获取图像源。将这个图像传递给 x 滤波器，然后给 y 滤波器。分别保留这两个滤波器是因为在尺度空间应用中通常不仅要计算滤波，而且还要计算不同阶的导数和滤波的组合。当使用滤波器来产生中间结果时可能要做一些因式分解。但是在这里这个功能是不太重要的，因此我们仅仅需要在所有方向上平滑图像。

```
filterX1->SetInput( reader->GetOutput( ) );
filterY1->SetInput( filterX1->GetOutput( ) );
filterY2->SetInput( reader->GetOutput( ) );
filterX2->SetInput( filterY2->GetOutput( ) );
```

现在到了选择高斯函数中用来平滑图像的 S 值的时候。注意：S 必须传递给两个滤波器，并且 S 是以毫计量的。也就是说，在应用于滤波的过程中时，滤波器将计算在图像中定义的空间值。

```
filterX1->SetSigma( sigma );
filterY1->SetSigma( sigma );
filterX2->SetSigma( sigma );
filterY2->SetSigma( sigma );
```

最后将两个拉普拉斯成员组合在一起。使用 itk::AddImageFilter 来完成：

```
typedef itk::AddImageFilter<
OutputImageType,
OutputImageType,
OutputImageType > AddFilterType;
AddFilterType::Pointer addFilter = AddFilterType::New( );
addFilter->SetInput1( filterY1->GetOutput( ) );
addFilter->SetInput2( filterX2->GetOutput( ) );
```

通过在流水线末端增加的滤波器上调用 Update( )来触发滤波器：

```
try
{
addFilter->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cout << "ExceptionObject caught !" << std::endl;
std::cout << err << std::endl;
return EXIT_FAILURE;
}
```

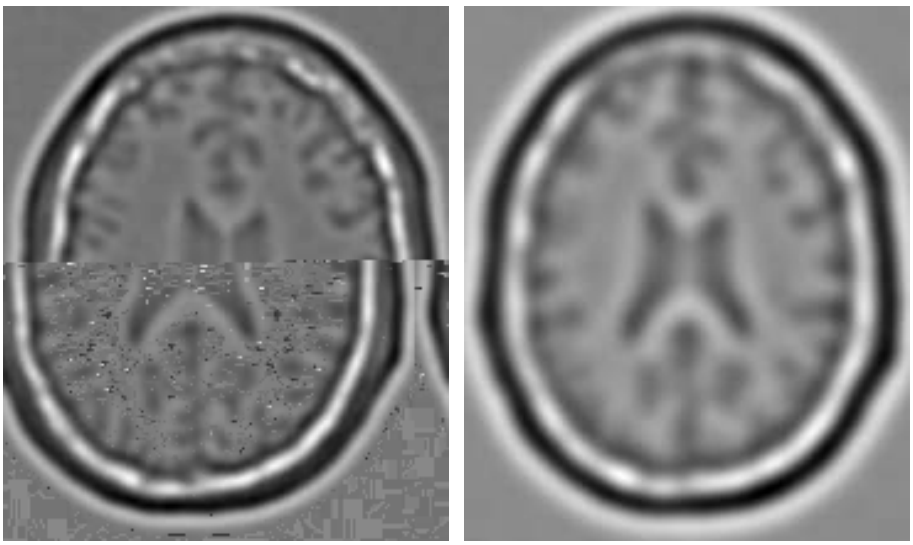


图 6-11　LaplacianRecursiveGaussianImageFilter 对一个 MRI 脑部切片亮度图像作用的效果

使用 itk::ImageFileWriter 类可以将结果图像保存到一个文件：

```
typedef float WritePixelType;
typedef itk::Image< WritePixelType, 2 > WriteImageType;
typedef itk::ImageFileWriter< WriteImageType > WriterType;
WriterType::Pointer writer = WriterType::New( );
writer->SetInput( addFilter->GetOutput( ) );
```

writer->SetFileName(argv[2]);

writer->Update( );

如图 6-11 所示阐述了这个滤波器对一个 MRI 脑部切片亮度图像作用的效果，使用的 S 值分别为 3（左图）和 5（右图）。这个图片展示了可以通过选择一个适当的标准差来调节对噪声的衰减。这种可调比例类型的滤波器适合于执行尺度空间分析。

本小节的源代码在文件 Examples/Filtering/LaplacianRecursiveGaussianImageFilter2.cxx 中。

前面的例子展示了如何使用 itk::RecursiveGaussianImageFilter 来计算一个使用高斯滤波后的图像的拉普拉斯等价物。在这以前的例子中使用的成员已经包含在 itk::LaplacianRecursiveGaussianImageFilter 中以便于简单化它的使用。在当前的例子中展示如何使用这个便捷的滤波器来得到和前面例子中相同的结果。

使用这个滤波器的第一步是包含它的头文件：

#include "itkLaplacianRecursiveGaussianImageFilter.h"

基于期望的输入和输出像素类型来选择类型：

typedef float InputPixelType;

typedef float OutputPixelType;

使用像素类型来对输入和输出图像类型进行实例化：

typedef itk::Image< InputPixelType, 2 > InputImageType;

typedef itk::Image< OutputPixelType, 2 > OutputImageType;

现在同时使用输入图像和输出图像类型来实例化滤波器类型：

typedef itk::LaplacianRecursiveGaussianImageFilter<

InputImageType, OutputImageType > FilterType;

这个滤波器的工具包已经在前面的例子中介绍过了。通过调用 New( )方式来创建这个滤波器并将结果指向一个 itk::SmartPointer：

FilterType::Pointer laplacian = FilterType::New( );

在这里滤波器中也可以选择标准化尺度空间的选项：

laplacian->SetNormalizeAcrossScale(false);

输入图像可以从另一个滤波器的输出得到。这里，我们用一个图像 reader 来作为获取图像源：

laplacian->SetInput(reader->GetOutput( ));

现在到了选择高斯中用来平滑图像的σ值的时候。注意：σ必须传递给两个滤波器，并且σ是以毫米来计量的。也就是说，在应用于滤波的过程中时，滤波器将计算在图像中定义的空间值。

laplacian->SetSigma(sigma);

最后通过调用 Update( )方式来运行流水线。

try

{

laplacian->Update( );

}

```
catch( itk::ExceptionObject & err )
{
std::cout << "ExceptionObject caught !" << std::endl;
std::cout << err << std::endl;
return EXIT_FAILURE;
}
```

如图 6-12 所示阐述了这个滤波器对一个 MRI 脑部切片亮度图像作用的效果，使用的 S 值分别为 3（左图）和 5（右图）。这个图片展示了可以通过选择一个适当的标准差来调节对噪声的衰减。这种可调比例类型的滤波器适合于执行尺度空间分析。

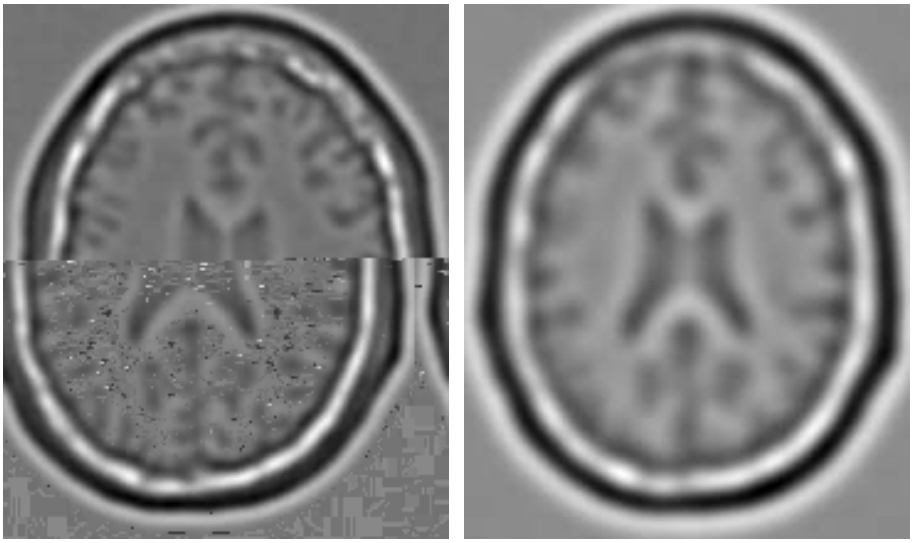


图 6-12　LaplacianRecursiveGaussianImageFilter 对一个 MRI 脑部切片亮度图像作用的效果

6.6　邻域滤波器

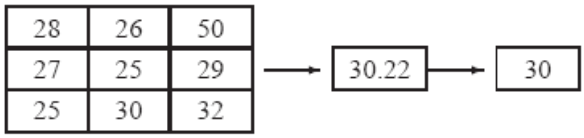
在一种使用输入像素的一个很小的邻域的信息来计算每个输出像素的滤波器形式的图像处理进程中会涉及位置这个概念。这些滤波器的一个典型形式是二维图像中的一个 3×3 滤波器。基于这些滤波器的卷积模块可以执行从消除噪声到微分操作、数学形态学范围内的多种不同的任务。

ITK 平台是以一种基于邻域图像滤波的优雅的方式来实现的。输入图像是使用一个称为 itk::NeighborhoodIterator 的特定的迭代器来处理的。这个迭代器具有移动图像中所有像素的能力，并对每个像素可以在当前邻域中标记出像素地址。定义一个操作符，对输入像素应用一个算法操作来产生一个输出像素值。接下来的章节介绍了应用这个构造的滤波器的更常见的用法（参考第 11 章可以得到更多关于迭代器的信息）。

6.6.1　均值滤波器

本小节的源代码在文件 Examples/Filtering/MeanImageFilter.cxx 中。

通常使用 itk::MeanImageFilter 来消除噪声。这个滤波器通过寻找输入像素相应邻域的统计均值来计算每个输出像素值。下面的图表阐述了在二维情况下 MeanImageFilter 的局部效果。左边的邻域的统计均值作为邻域中间像素的输出值。



注意：这个算法对邻域内呈现的奇异值很敏感。由于这个滤波器内在使用了 itk::SmartNeighborhoodIterator 和 itk::NeighborhoodOperator，因此可以对任何维的图像进行处理。用来计算均值的邻域大小可以由用户来设置。

首先必须包含与这个滤波器相关的头文件：

```
#include "itkMeanImageFilter.h"
```

然后必须定义输入和输出图像的像素类型并使用它们来对图像类型进行实例化：

```
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在可以使用图像类型来对滤波器类型进行实例化并创建滤波器对象：

```
typedef itk::MeanImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

通过对相应的值传递一个 SizeType 对象来定义沿每个维的邻域大小。每个维的值作为一个矩形框的不完全尺寸来使用。例如：在二维情况下一个大小为 1×2 的 SizeType 将产生一个 3×5 的邻域。

```
InputImageType::SizeType indexRadius;
indexRadius[0] = 1; // radius along x
indexRadius[1] = 1; // radius along y
filter->SetRadius( indexRadius );
```

这个滤波器的输入可以从其他任何滤波器得到，例如一个 reader。输出可以沿流水线传递给其他滤波器，例如一个 writer。下游的任何一个滤波器调用 update 将触发均值滤波器的运行：

```
filter->SetInput( reader->GetOutput( ) );
writer->SetInput( filter->GetOutput( ) );
writer->Update( );
```

如图 6-13 所示阐述了这个滤波器对一个 MRI 脑部切片图像作用的效果，使用的邻域半径为 1×1，对应一个 3×3 的典型邻域。从图像中可以看出在邻域亮度值的扩散下边缘迅速退化。

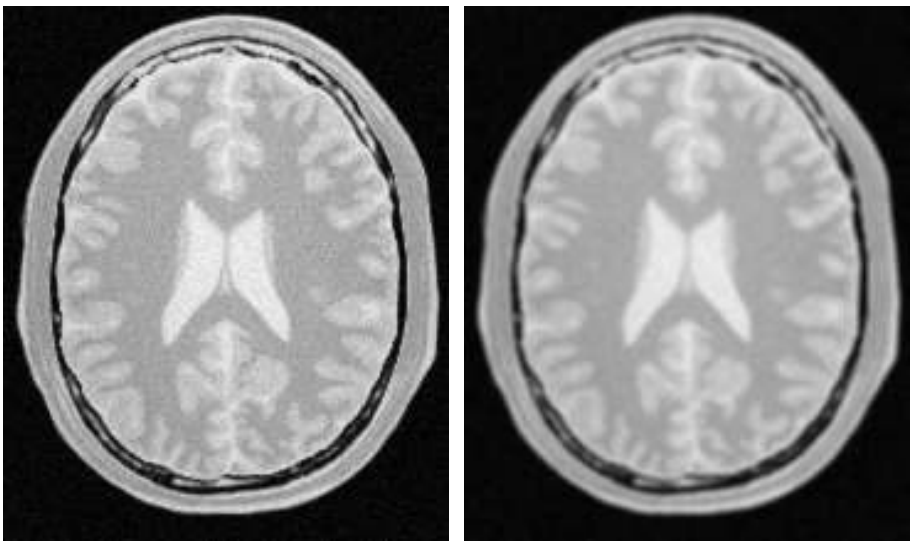


图 6-13　MeanImageFilter 对一个 MRI 脑部切片图像作用的效果

6.6.2　中值滤波器

本小节的源代码在文件 Examples/Filtering/MedianImageFilter.cxx 中。

itk::MedianImageFilter 通常用来作为消除噪声的一种稳健方法。这个滤波器对消除椒盐噪声更加有效。MedianImageFilter 计算每个输入像素周围相关邻域值的中值作为每个输出像素值。下面的图表阐述了在二维情况下这个滤波器的局部效果。左边的邻域的统计中值作为邻域中间像素的输出值。

由于这个滤波器内在使用了 itk::NeighborhoodIterator 和 itk::NeighborhoodOperator，因此可以对任何维的图像进行处理。用来计算中值的邻域大小可以由用户来设置。

首先必须包含与这个滤波器相关的头文件：

```
#include "itkMedianImageFilter.h"
```

然后必须定义输入和输出图像的像素类型，并使用它们来对图像类型进行实例化：

```
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在可以使用图像类型来对滤波器类型进行实例化并创建滤波器对象。

```
typedef itk::MedianImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

通过对相应的值传递一个 SizeType 对象来定义沿每个维的邻域大小。每个维的值作为一个矩形框的不完全尺寸来使用。例如：在二维情况下一个大小为 1×2 的 SizeType 将产生一个 3×5 的邻域。

```
InputImageType::SizeType indexRadius;
indexRadius[0] = 1; // radius along x
indexRadius[1] = 1; // radius along y
filter->SetRadius( indexRadius );
```

这个滤波器的输入可以从其他任何滤波器得到，例如一个 reader。输出可以沿流水线传递给其他滤波器，例如一个 writer。下游的任何一个滤波器调用 update 将触发中值滤波器的运行。

```
filter->SetInput( reader->GetOutput( ) );
writer->SetInput( filter->GetOutput( ) );
writer->Update( );
```

如图 6-14 所示阐述了这个滤波器对一个 MRI 脑部切片图像作用的效果，使用的邻域半径为 1×1，对应于一个 3×3 的典型邻域。这个滤波的图像展示了用中值滤波器的适中趋向来保护边缘。

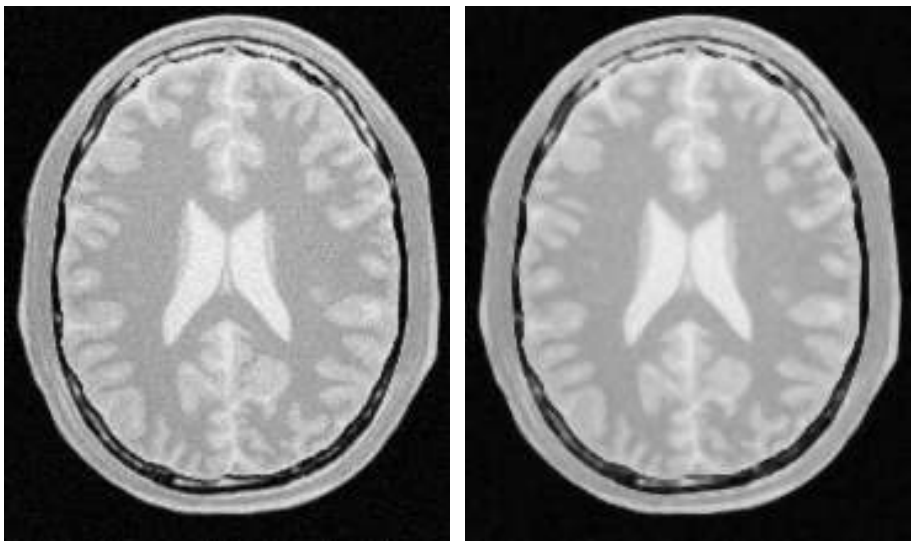


图 6-14　MedianImageFilter 对一个 MRI 脑部切片图像作用的效果

6.6.3　数学形态学

数学形态学已经被证明是图像处理和分析的一个强有力的工具。在 ITK 中使用 NeighborhoodIterators 和 itk::NeighborhoodOperators 来实现数学形态学滤波器。研发平台中包含有两种图像形态学算法：对二值图像操作的滤波器和对灰度尺图像操作的滤波器。

1．二值滤波器

本小节的源代码在文件 Examples/Filtering/MathematicalMorphologyBinaryFilters.cxx 中。

接下来的章节介绍了对二值图像执行基本数学形态学操作的滤波器的用法。这里介绍了

itk::BinaryErodeImageFilter 和 itk::BinaryDilateImageFilter。滤波器的名字已经明确地指定了它们操作的图像类型。下面包含了构造一个简单地使用数学形态学滤波器需要的头文件：

```
#include "itkBinaryErodeImageFilter.h"
#include "itkBinaryDilateImageFilter.h"
#include "itkBinaryBallStructuringElement.h"
```

接下来的代码定义了输入、输出像素类型以及和它们相关的图像类型：

```
const unsigned int Dimension = 2;
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

数学形态学操作通过对每个输入像素的邻域应用一个操作来实现。规则和邻域的结合被称为构造成员。尽管一些规则已经成为图像处理的实际标准，但是使用什么类型的算法规则应用于邻域还是有很大的选择自由的。在 ITK 中是在最小化腐蚀和最大化膨胀的典型规则下实现的。

构造成员是作为一个 NeighborhoodOperator 来实现的。事实上，默认的构造成员是 itk::BinaryBallStructuringElement 类。这个类是使用像素类型和输入图像维来进行实例化的：

```
typedef itk::BinaryBallStructuringElement<
InputPixelType,
Dimension > StructuringElementType;
```

然后使用构造成员类型和输入、输出图像类型一起来对滤波器类型进行实例化：

```
typedef itk::BinaryErodeImageFilter<
InputImageType,
OutputImageType,
StructuringElementType > ErodeFilterType;
typedef itk::BinaryDilateImageFilter<
InputImageType,
OutputImageType,
StructuringElementType > DilateFilterType;
```

现在通过调用 New( )方式来创建这个滤波器并将结果指向一个 itk::SmartPointer：

```
ErodeFilterType::Pointer binaryErode = ErodeFilterType::New( );
DilateFilterType::Pointer binaryDilate = DilateFilterType::New( );
```

构造成员不是一个引用记数类。因此它是作为一个 C++堆栈而不是使用 New( )和 SmartPointers 来创建的。使用 SetRadius( )方式来定义和构造成员相关的邻域半径并调用 CreateStructuringElement( )方式以便于初始化操作符。如下面所阐述的那样，使用 SetKernel( )方式将构造成员的结果传递给数学形态学滤波器。

```
StructuringElementType structuringElement;
structuringElement.SetRadius( 1 ); // 3x3 structuring element
structuringElement.CreateStructuringElement( );
```

```
binaryErode->SetKernel( structuringElement );
binaryDilate->SetKernel( structuringElement );
```

提供一个二值图像作为滤波器的输入。例如：这个图像可以是一个二值阈值图像滤波器的输出。

```
thresholder->SetInput( reader->GetOutput( ) );
InputPixelType background = 0;
InputPixelType foreground = 255;
thresholder->SetOutsideValue( background );
thresholder->SetInsideValue( foreground );
thresholder->SetLowerThreshold( lowerThreshold );
thresholder->SetUpperThreshold( upperThreshold );
binaryErode->SetInput( thresholder->GetOutput( ) );
binaryDilate->SetInput( thresholder->GetOutput( ) );
```

和二值图像中的对象相关的值是用 SetErodeValue( )和 SetDilateValue( )方式来指定的。传递到这些方法的值将考虑到应用于腐蚀和膨胀的值。

```
binaryErode->SetErodeValue( foreground );
binaryDilate->SetDilateValue( foreground );
```

这个滤波器是通过调用它的 Update( )方式来运行的，也可以通过更新任何下游的滤波器比如一个图像 writer 来运行。

```
writerDilation->SetInput( binaryDilate->GetOutput( ) );
writerDilation->Update( );
```

如图 6-15 所示阐述了这个滤波器对一个从 MRI 脑部切片图像得来的二值图像进行腐蚀和膨胀的效果。这个图像展示了如何使用这些操作来从分割图像中删除假的细节。

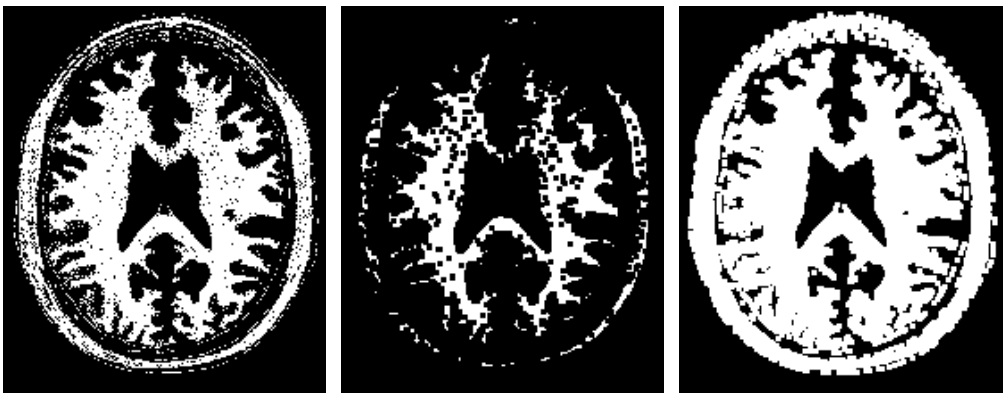


图 6-15　对一个二值图像进行腐蚀和膨胀的效果

2．灰度尺滤波器

本小节的源代码在文件 Examples/Filtering/MathematicalMorphologyGrayscaleFilters.cxx 中。

接下来的章节介绍了对灰度尺图像执行基本数学形态学操作的滤波器的用法。在这个例

子中用到了 itk::BinaryErodeImageFilter 和 itk::BinaryDilateImageFilter。滤波器的名字已经明确地指定了它们操作的图像类型。下面包含了构造一个简单的使用灰度尺数学形态学滤波器需要的头文件：

```
#include "itkGrayscaleErodeImageFilter.h"
#include "itkGrayscaleDilateImageFilter.h"
#include "itkBinaryBallStructuringElement.h"
```

接下来的代码定义了输入输出像素类型以及和它们相关的图像类型：

```
const unsigned int Dimension = 2;
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

数学形态学是基于对每个输入像素的邻域应用一个操作来操作的。规则和邻域的结合被称为构造成员。尽管一些规则已经成为图像处理的实际标准，但是使用什么类型的算法规则应用于邻域还是有很大的选择自由的。在 ITK 中是在最小化腐蚀和最大化膨胀的典型规则下实现的。

构造成员是作为一个 itk::NeighborhoodOperator 来实现的。事实上，默认的构造成员是 itk::BinaryBallStructuringElement 类。这个类是使用像素类型和输入图像维来进行实例化的。

```
typedef itk::BinaryBallStructuringElement<
InputPixelType,
Dimension > StructuringElementType;
```

然后使用构造成员类型和输入、输出图像类型一起来对滤波器类型进行实例化：

```
typedef itk::GrayscaleErodeImageFilter<
InputImageType,
OutputImageType,
StructuringElementType > ErodeFilterType;
typedef itk::GrayscaleDilateImageFilter<
InputImageType,
OutputImageType,
StructuringElementType > DilateFilterType;
```

现在通过调用 New( )方式来创建这个滤波器并将结果指向一个 itk::SmartPointer：

```
ErodeFilterType::Pointer grayscaleErode = ErodeFilterType::New( );
DilateFilterType::Pointer grayscaleDilate = DilateFilterType::New( );
```

构造成员不是一个引用记数类。因此它是作为一个 C++堆栈而不是使用 New( )和 SmartPointers 来创建的。使用 SetRadius( )方式来定义和构造成员相关的邻域半径并调用 CreateStructuringElement( )方式以便于初始化操作符。如下面所阐述的那样，使用 SetKernel( )方式将构造成员的结果传递给数学形态学滤波器：

```
StructuringElementType structuringElement;
structuringElement.SetRadius( 1 ); // 3x3 structuring element
```

```
structuringElement.CreateStructuringElement( );
grayscaleErode->SetKernel( structuringElement );
grayscaleDilate->SetKernel( structuringElement );
```

提供一个灰度尺图像作为滤波器的输入。例如，这个图像可以是一个 reader 的输出：

```
grayscaleErode->SetInput( reader->GetOutput( ) );
grayscaleDilate->SetInput( reader->GetOutput( ) );
```

这个滤波器是通过调用它的 Update( )方式来运行的，也可以通过更新任何下游的滤波器比如一个图像 writer 来运行。

如图 6-16 所示阐述了这个滤波器对一个从 MRI 脑部切片图像得来的二值图像上腐蚀和膨胀的效果。这个图像展示了如何使用这些操作来从分割图像中删除假的细节。

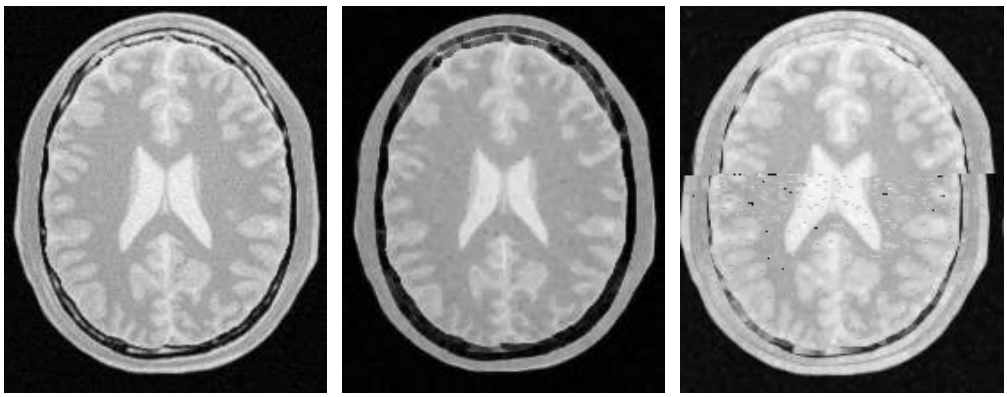


图 6-16　对一个灰度图像进行腐蚀和膨胀的效果

6.6.4　Voting 滤波器

Voting 滤波器是滤波器的一个通用字形家族。实际上，从数学形态学来的膨胀和腐蚀滤波器都是 Voting 滤波器广大家族的特定情况。在一个 Voting 滤波器中，一个像素的输出是通过计算在这个像素邻域中的像素数目并对计算结果应用一个规则来决定的。例如以一个 Voting 滤波器形式运行的一个典型例子是：如果背景邻域的数量大于等于 1，那么前景像素将成为背景。在本文中，你可以想象腐蚀计算中需要至少改变 3 个前景。

1．二值中值滤波器

Voting 滤波器的一个特定情况是 BinaryMedianImageFilter。这个滤波器等同于在一个二值图像上应用一个中值滤波器。由于使用一个二值图像作为输入可以优化滤波器的运行，所以不需要按照像素在邻域中出现的频率来对像素进行分类。

本小节的源代码在文件 Examples/Filtering/BinaryMedianImageFilter.cxx 中。

itk::BinaryMedianImageFilter 通常用来作为消除噪声的一种稳健方法。BinaryMedian ImageFilter 计算每个输入像素周围相关邻域值的中值作为每个输出像素值。当输入图像是二值图像时，可以通过简单计算当前像素周围 ON/OFF 像素的数量来优化运行。

由于这个滤波器内在使用了 itk::NeighborhoodIterator 和 itk::NeighborhoodOperator，因此可以对任何维的图像进行处理。用来计算中值的邻域大小可以由用户来设置。

首先必须包含与这个滤波器相关的头文件：

```
#include "itkBinaryMedianImageFilter.h"
```

然后必须定义输入和输出图像的像素和图像类型：

```
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在可以使用图像类型来定义滤波器的类型并创建滤波器对象：

```
typedef itk::BinaryMedianImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

通过对相应的值传递一个 SizeType 对象来定义沿每个维的邻域大小。每个维的值作为一个矩形框的不完全尺寸来使用。例如：在二维情况下一个大小为 1×2 的 SizeType 将产生一个 3×5 的邻域。

```
InputImageType::SizeType indexRadius;
indexRadius[0] = radiusX; // radius along x
indexRadius[1] = radiusY; // radius along y
filter->SetRadius( indexRadius );
```

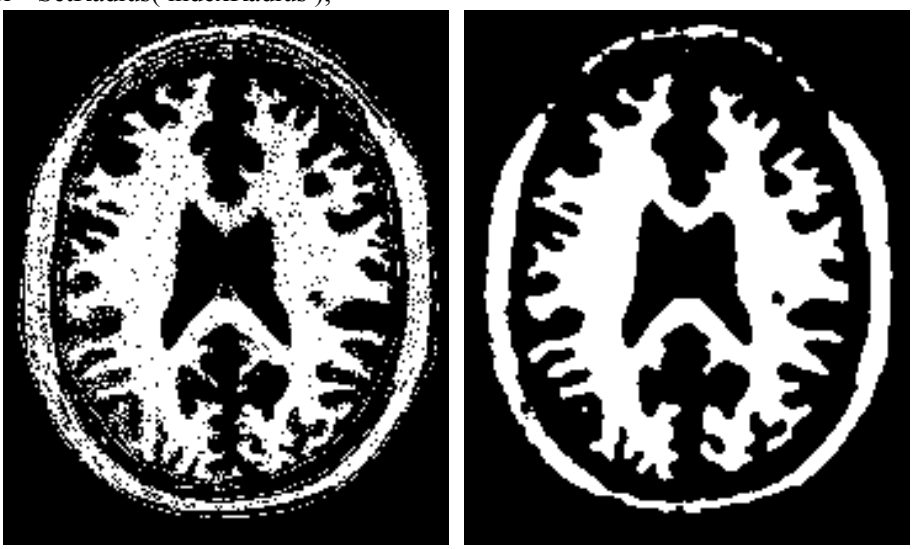


图 6-17　BinaryMedianImageFilter 对一个 MRI 脑部切片图像作用的效果

这个滤波器的输入可以从其他任何滤波器得到，例如一个 reader。输出可以沿流水线传递给其他滤波器，例如一个 writer。下游的任何一个滤波器调用 Update 将触发中值滤波器的运行。

```
filter->SetInput( reader->GetOutput( ) );
writer->SetInput( filter->GetOutput( ) );
writer->Update( );
```

如图 6-17 所示阐述了 BinaryMedianImageFilter 滤波器对一个 MRI 脑部切片图像作用的效果，使用的邻域半径为 2×2，对应于一个 5×5 的典型邻域。这个滤波的图像展示了这个滤波器同时在前景和背景中消除噪声的能力，类似于平滑区域的边界。

中值滤波对一个噪声数字图像过滤的典型效果是动态消除脉冲噪声突出。这个滤波器同样也有保留不同于信号阶的亮度的趋向，并产生减少区域边界模糊的结果。这个滤波器也有保留图像中边界位置的趋向。

图 6-18　BinaryMedianImageFilter 使用一个典型的 3×3 窗口分别运行 1、10 和 50 次的效果

如图 6-18 所示展示了使用一个典型的 3×3 窗口分别运行 1、10 和 50 次的效果。随着窗口大小的增加，在消除噪声和锐化图像之间有一个权衡。

2．洞穴充填滤波器

Voting 滤波器的另一个变量是 Hold Filling 滤波器。这个滤波器仅仅在前景像素的数量在邻域中占大多数时将背景像素转变为前景。通过选择大多数的尺度，这个滤波器可以填充

不同尺度的洞。为了更加精确，这个滤波器的效果和当前像素的边缘曲率密切相关。

本小节的源代码在文件 Examples/Filtering/VotingBinaryHoleFillingImageFilter.cxx 中。

itk::VotingBinaryHoleFillingImageFilter 应用一个 Voting 操作以便于填充洞穴。这也可以用来在二值图像中平滑轮廓和填充洞穴。

首先必须包含与这个滤波器相关的头文件：

```
#include "itkVotingBinaryHoleFillingImageFilter.h"
```

然后必须定义输入和输出图像的像素和图像类型：

```
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在可以使用图像类型来定义滤波器的类型并创建滤波器对象：

```
typedef itk::VotingBinaryHoleFillingImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

通过对相应的值传递一个 SizeType 对象来定义沿每个维的邻域大小。每个维的值作为一个矩形框的不完全尺寸来使用。例如：在二维情况下一个大小为 1×2 的 SizeType 将产生一个 3×5 的邻域。

```
InputImageType::SizeType indexRadius;
indexRadius[0] = radiusX; // radius along x
indexRadius[1] = radiusY; // radius along y
filter->SetRadius( indexRadius );
```

由于这个滤波器需要一个二值图像作为输入，所以我们必须指定用来作为背景和前景的灰度级。可以通过使用 SetForegroundValue( )和 SetBackgroundValue( )方式来设置：

```
filter->SetBackgroundValue( 0 );
filter->SetForegroundValue( 255 );
```

我们也必须指定用来作为判别将背景像素转变为前景像素的判断标准的大多数阈值。转变的规则是：通过这个大多数阈值来判断，如果邻域中前景的数量大于背景的数量，就将背景像素转变为前景像素。例如：在一个二维图像中，选择邻域半径为 1，则邻域大小为 3×3。

如果我们设置大多数阈值为 2，则我们需要邻域中前景的数量至少为(3×3-1)/2 并加上大多数值。这可以使用 SetMajorityThreshold( )方式来完成：

```
filter->SetMajorityThreshold( 2 );
```

这个滤波器的输入可以从其他任何滤波器得到，例如一个 reader。输出可以沿流水线传递给其他滤波器，例如一个 writer。下游的任何一个滤波器调用 update 将触发中值滤波器的运行。

```
filter->SetInput( reader->GetOutput( ) );
writer->SetInput( filter->GetOutput( ) );
writer->Update( );
```

如图 6-19 所示阐述了 VotingBinaryHoleFillingImageFilter 滤波器在一个 MRI 脑部图像的

阈值分割切片上作用的效果，使用的邻域半径分别为 1×1、2×2 和 3×3，对应的邻域尺度分别为 3×3、5×5 和 7×7。这个滤波的图像展示了这个滤波器同时在前景和背景中消除噪声的能力，类似于平滑区域的边界。

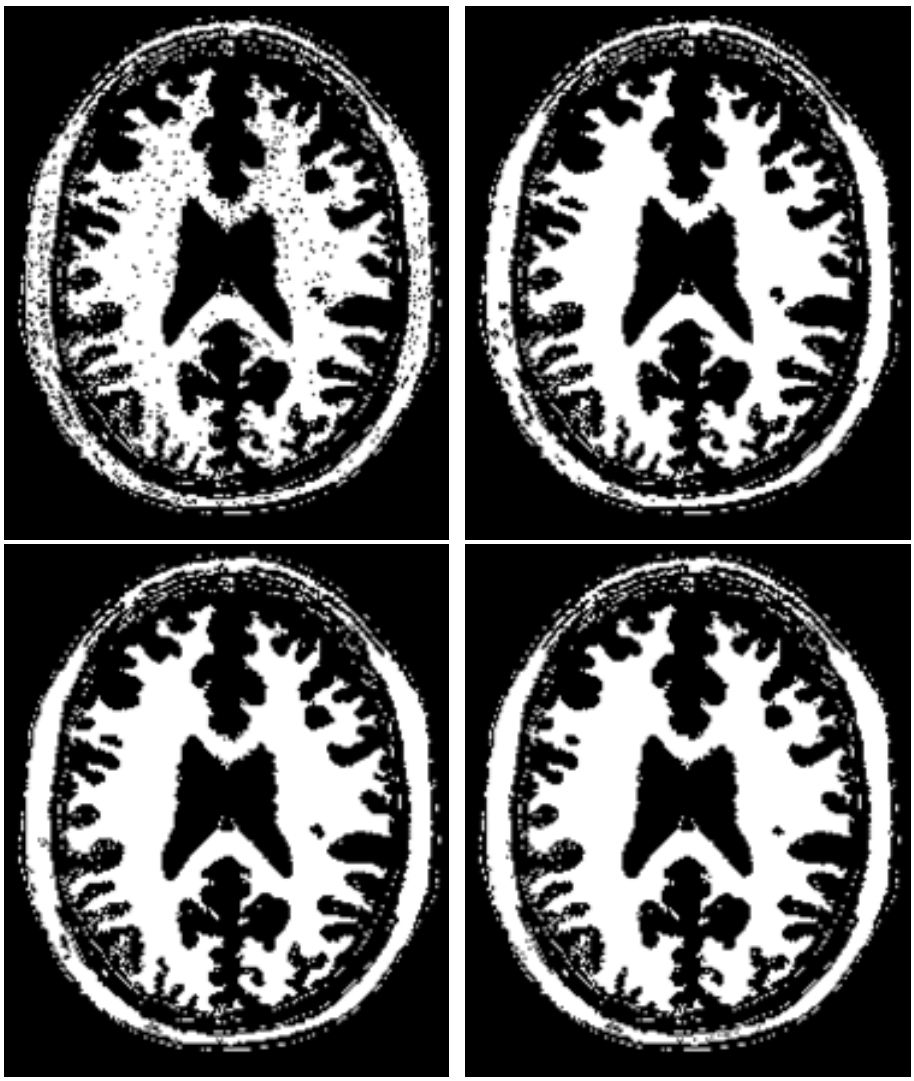


图 6-19　Effect of the VotingBinaryHoleFillingImageFilter 滤波器对一个 MRI 脑部图像的阈值分割切片作用的效果，预先对图像进行了阈值处理以生成二值图像。输出图像的半径分别为 1、2 和 3

3．迭代洞穴充填滤波器

Hold Filling 滤波器可以以迭代的方式来使用，通过反复地应用滤波器直到没有像素变化为止。本文中，滤波器可以被看成是一个水平集滤波器的二值变量。

本小节的源代码在文件 Examples/Filtering/VotingBinaryIterativeHoleFillingImageFilter.cxx 中。

itk::VotingBinaryIterativeHoleFillingImageFilter 应用一个 Voting 操作以便于填充洞穴。这也可以用来在二值图像中平滑轮廓和填充洞穴。这个滤波器内部反复运行 itk::VotingBinary

HoleFillingImageFilter 直到没有像素变化或到达了大多数迭代器。

首先必须包含与这个滤波器相关的头文件：

```
#include "itkVotingBinaryIterativeHoleFillingImageFilter.h"
```

然后必须定义像素类型和图像类型。注意这个滤波器需要输入和输出图像的类型相同，因此需要一个单一的图像类型来进行模板实例化：

```
typedef unsigned char PixelType;
typedef itk::Image< PixelType, 2 > ImageType;
```

现在可以使用图像类型来定义滤波器的类型并创建滤波器对象：

```
typedef itk::VotingBinaryIterativeHoleFillingImageFilter<
ImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

通过对相应的值传递一个 SizeType 对象来定义沿每个维的邻域大小。每个维的值作为一个矩形框的不完全尺寸来使用。例如：在二维情况下一个大小为 1×2 的 SizeType 将产生一个 3×5 的邻域：

```
ImageType::SizeType indexRadius;
indexRadius[0] = radiusX; // radius along x
indexRadius[1] = radiusY; // radius along y
filter->SetRadius( indexRadius );
```

由于这个滤波器需要一个二值图像作为输入，所以我们必须指定用来作为背景和前景的灰度级。可以通过使用 SetForegroundValue( )和 SetBackgroundValue( )方式来设置：

```
filter->SetBackgroundValue( 0 );
filter->SetForegroundValue( 255 );
```

我们也必须指定用来作为判别将背景像素转变为前景像素的判断标准的大多数阈值。转变的规则是：通过对这个大多数阈值进行来判断，如果邻域中前景的数量大于背景的数量就将背景像素转变为前景像素。例如：在一个二维图像中，选择邻域半径为 1，则邻域大小为 3×3。如果我们设置大多数阈值为 2，则我们需要邻域中前景的数量至少为(3×3-1)/2 并加上大多数值。这可以使用 SetMajorityThreshold( )方式来完成：

```
filter->SetMajorityThreshold( 2 );
```

最后我们指定这个滤波器需要运行的迭代的最大次数。迭代的数量将决定这个滤波器能填充的洞穴的最大尺度。迭代的次数越多，填充的洞穴就越大。

```
filter->SetMaximumNumberOfIterations( numberOfIterations );
```

这个滤波器的输入可以从其他任何滤波器得到，例如一个 reader。输出可以沿流水线传递给其他滤波器，例如一个 writer。下游的任何一个滤波器调用 Update 将触发中值滤波器的运行。

```
filter->SetInput( reader->GetOutput( ) );
writer->SetInput( filter->GetOutput( ) );
writer->Update( );
```

如图 6-20 所示阐述了 VotingBinaryIterativeHoleFillingImageFilter 滤波器对一个 MRI 脑部图像的阈值分割切片作用的效果，使用的邻域半径分别为 1×1、2×2 和 3×3，对应的邻域

尺度分别为 3×3、5×5 和 7×7。这个滤波的图像展示了这个滤波器同时在前景和背景中消除噪声的能力，类似于平滑区域的边界。

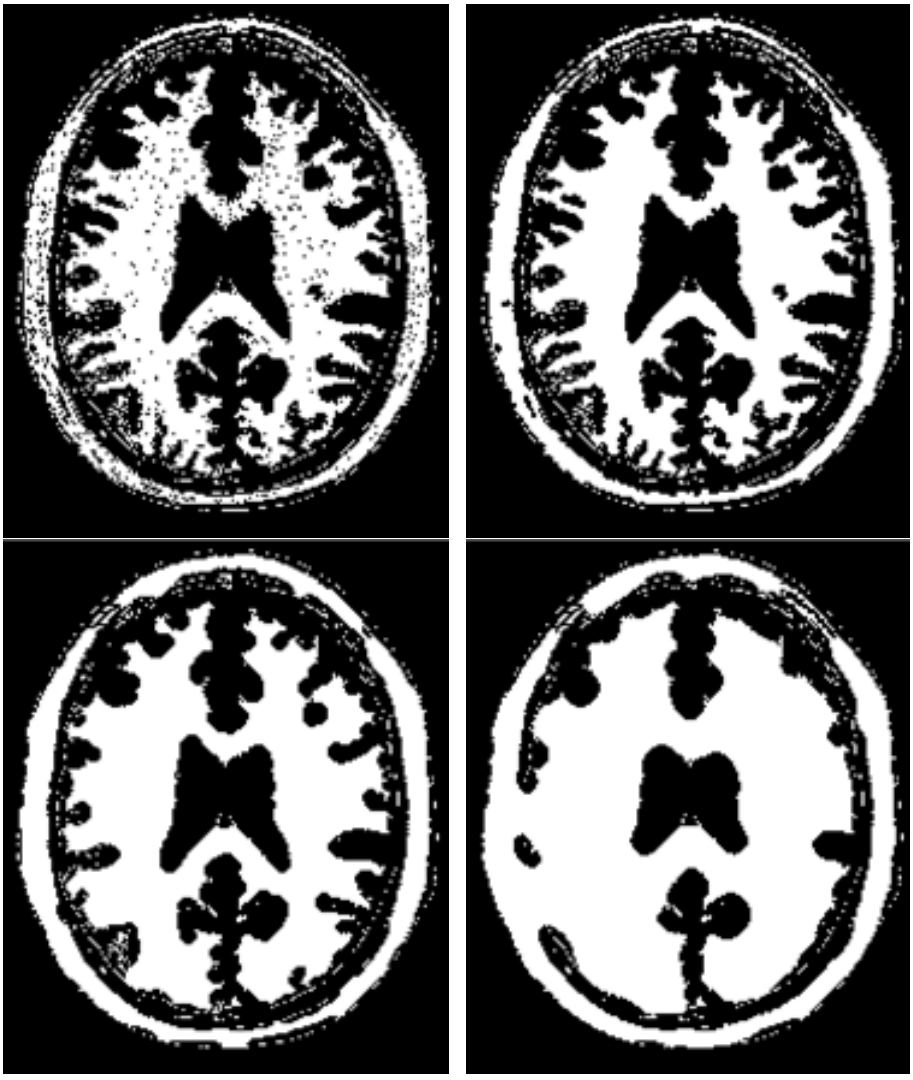


图 6-20　VotingBinaryIterativeHoleFillingImageFilter 滤波器对一个 MRI 脑部图像的阈值分割切片作用的效果，预先对图像进行了阈值处理以生成二值图像。输出图像的半径分别为 1、2 和 3

6.7　平滑滤波器

真正的图像数据有一系列的不确定性，表现在指向像素的多种度量标准中。这种不确定解释为噪声并被认为是图像数据中不需要的成分。本节将介绍可以用于消除图像中噪声的几种方法。

6.7.1 模糊

模糊是从图像中消除噪声的传统方法。它通常以和一个核进行卷积来实现。图像模糊的效果是削弱图像频谱的高频部分。不同的核以不同的方式来削弱频率。使用的一种典型的核是高斯。在 ITK 平台中实现了两种高斯平滑滤波。第一种是基于传统的卷积，而第二种是基于应用 IIR 滤波器来近似于和高斯卷积。

1．离散高斯

本小节的源代码在文件 Examples/Filtering/DiscreteGaussianImageFilter.cxx 中。

itk::DiscreteGaussianImageFilter 计算输入图像和一个高斯核的卷积。这是通过利用高斯核的可分离性在 ND 中实现的。一个一维的高斯函数在一个卷积核上进行离散化。扩展核的尺度直到在高斯函数中有足够的离散点来确保用户提供的最大错误值不会溢出。由于核的尺度是预先未知的，所以有必要对它的生长加以限制。用户可以提供一个核的尺度的最大允许值。离散化错误是作为离散高斯曲线的区域和连续高斯曲线区域的差异来定义的。如图 6-21 所示。

在 ITK 中高斯核是根据 Tony Lindeberg 理论来构造的，因此平滑和微分是在离散化前后交换使用的。换句话说，对一个通过和高斯函数卷积平滑得到的图像 I 进行有限差异微分等同于通过和一个高斯函数的微分进行卷积得到的 I 上进行有限差异计算。

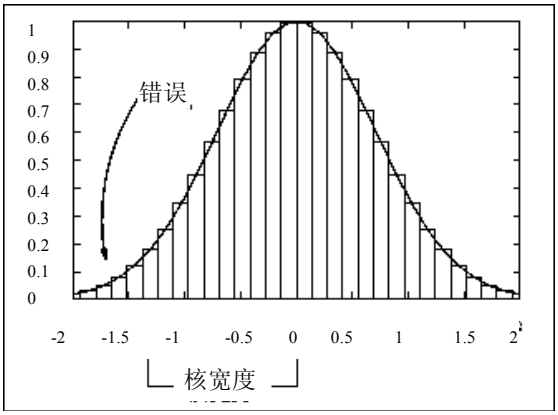


图 6-21　离散高斯函数

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkDiscreteGaussianImageFilter.h"
```

必须选择输入和输出图像的像素类型。使用像素类型和维来实例化图像类型：

```
typedef float InputPixelType;
typedef float OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

使用输入、输出图像类型来实例化离散高斯滤波器。创建一个相应的滤波器对象：

```
typedef itk::DiscreteGaussianImageFilter<
InputImageType, OutputImageType > FilterType;
```

FilterType::Pointer filter = FilterType::New( );

输入图像可以从另一个滤波器的输出得到。这里，使用一个图像 reader 作为它的输入：

filter->SetInput(reader->GetOutput( ));

滤波器需要用户提供一个和高斯核相关的变量值。使用 SetVariance( )方式来实现。离散高斯作为一个卷积核来构造。用户可以设置核的最大尺度。注意变量和核的尺度值的结合将可能导致一个调节过的高斯核。

filter->SetVariance(gaussianVariance);

filter->SetMaximumKernelWidth(maxKernelWidth);

最后，通过调用 Update( )方式来触发滤波器的执行：

filter->Update( );

如果这个滤波器的输出已经连接到流水线下游的其他滤波器，更新任何下游的滤波器将触发这个滤波器的执行。例如在这个滤波器后面使用的一个 writer：

rescaler->SetInput(filter->GetOutput( ));

writer->SetInput(rescaler->GetOutput( ));

writer->Update( );

如图 6-22 所示阐述了这个滤波器对一个脑部 MRI 切片亮度图像作用的效果。

注意：大的高斯变量将产生大的高斯核，从而相对地减缓计算时间。除非需要一个很大的精确度，否则近似使用 itk::RecursiveGaussianImageFilter 和大的变量。

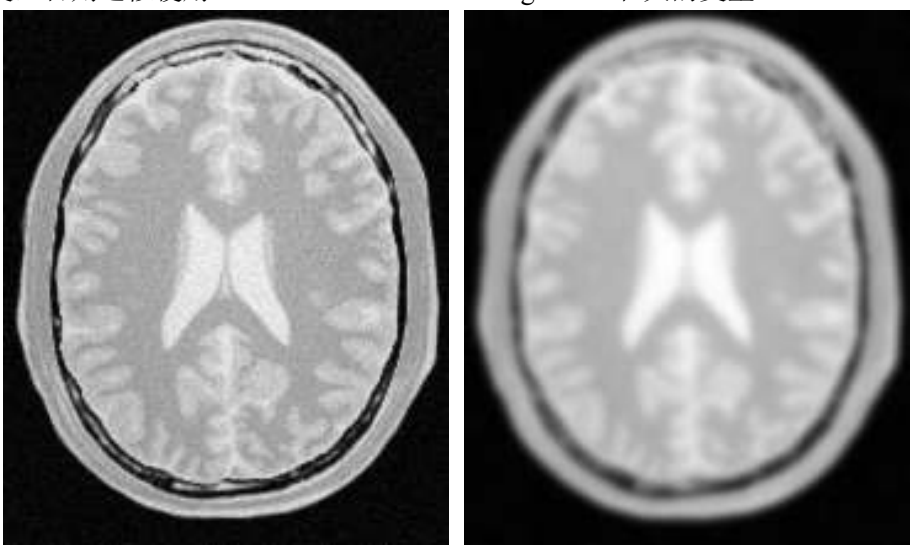


图 6-22　DiscreteGaussianImageFilter 对一个脑部 MRI 切片亮度图像作用的效果

2．二项式模糊

本小节的源代码在文件 Examples/Filtering/BinomialBlurImageFilter.cxx 中。

itk::BinomialBlurImageFilter 沿每个维来计算一个最接近的邻域平均。这个过程将按用户指定的次数重复进行。理论上，经过一定次数的迭代，结果将非常接近于和一个高斯卷积的结果。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkBinomialBlurImageFilter.h"
```

必须选择输入和输出图像的像素类型。使用像素类型和维来实例化图像类型：

```
typedef float InputPixelType;
typedef float OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

使用输入输出图像类型来实例化滤波器类型，然后创建一个相应的滤波器对象：

```
typedef itk::BinomialBlurImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从其他滤波器的输出得到。这里使用一个图像 reader 作为图像源。使用 SetRepetitions( )方式来设置重复的次数。随着选择的重复次数增多将线性地增加计算的时间。最后通过调用 Update( )方式来触发滤波器的运行：

```
filter->SetInput( reader->GetOutput( ) );
filter->SetRepetitions( repetitions );
filter->Update( );
```

如图 6-23 所示阐述了这个滤波器对一个脑部 MRI 切片亮度图像作用的效果。

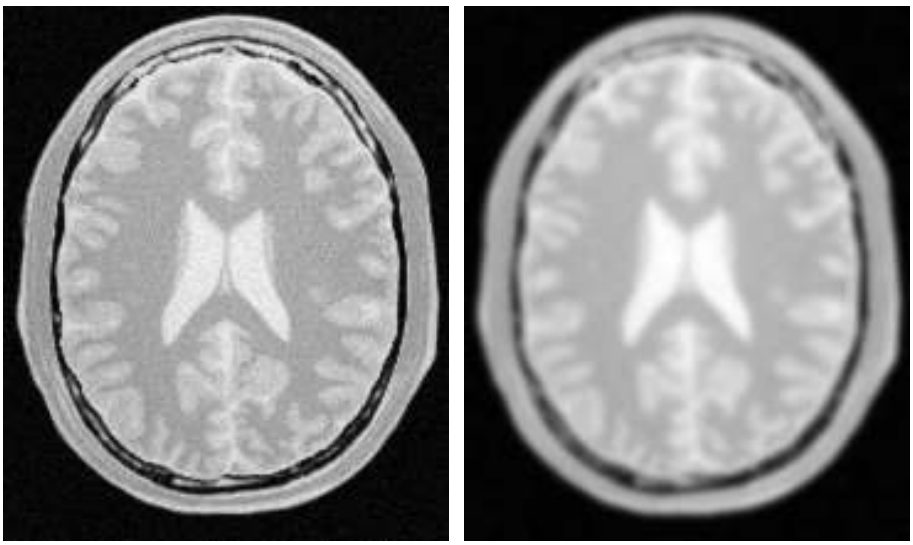


图 6-23　BinomialBlurImageFilter 对一个脑部 MRI 切片亮度图像作用的效果

注意：与高斯等价的标准差 σ 是固定的。在空间频谱中，这个滤波器的每个迭代器的效果就如同增加一个 sinus cardinal 函数。

3．IIR 高斯递归

本小节的源代码在文件 Examples/Filtering/SmoothingRecursiveGaussianImageFilter.cxx 中。

通过卷积一个高斯核的古典平滑图像方法有一个缺点：当高斯的标准差 σ 较大时将减缓速度。这是由于核的尺度过大会导致每个像素更大的计算数量造成的。

itk::RecursiveGaussianImageFilter 使用 IIR 滤波器来实现近似接近高斯函数和它的微分的卷积。实际中这个滤波器需要一定数量的操作来近似接近卷积，而不用考虑 σ 值。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkRecursiveGaussianImageFilter.h"
```

选择输入、输出像素类型：

```
typedef float InputPixelType;
typedef float OutputPixelType;
```

使用像素类型来对输入、输出图像进行实例化：

```
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在使用输入、输出图像类型对滤波器类型进行实例化：

```
typedef itk::RecursiveGaussianImageFilter<
InputImageType, OutputImageType > FilterType;
```

这个滤波器应用在一个单一方向上的近似卷积。因此连接几个这种滤波器在所有方向上生成滤波是很有必要的。在这个例子中，我们创建一对滤波器以便于处理一个二维图像。通过调用 New( )方式来创建滤波器并将结果指向一个 itk::SmartPointer：

```
FilterType::Pointer filterX = FilterType::New( );
FilterType::Pointer filterY = FilterType::New( );
```

由于新创建的滤波器中每个滤波器都有可能沿任何维来执行滤波的潜力，所以我们将每个滤波器限定在一个特定的方向上。可以通过 SetDirection( )方式来完成：

```
filterX->SetDirection( 0 ); // 0 --> X direction
filterY->SetDirection( 1 ); // 1 --> Y direction
```

itk::RecursiveGaussianImageFilter 可以近似接近和高斯函数或它的一阶和二阶导数的卷积。我们通过使用 SetOrder( )方式来选择这些选项。注意：这个变量是一个可以为零阶、一阶和二阶值的枚举。例如：为了计算 X 的偏导数，我们选择 X 为一阶而 Y 为零阶。这里我们仅要对 X 和 Y 进行平滑，所以我们选择两个方向上都为零阶：

```
filterX->SetOrder( FilterType::ZeroOrder );
filterY->SetOrder( FilterType::ZeroOrder );
```

有两种常见的标准化高斯函数的方式，都取决于它们的应用。对于尺度空间分析，使用一个标准化是用来阻止输入的最大值的。这个标准化可以用下面的式子表示：

$$\frac{1}{\sigma\sqrt{2\pi}} \tag{6-4}$$

在应用中使用高斯函数作为扩散方程的一个解，使用标准化函数是为了阻止信号的积分。这个最后的方式可以看成是一个质量守恒原理。这可以用下面的式子来表示：

$$\frac{1}{\sigma^2\sqrt{2\pi}} \tag{6-5}$$

itk::RecursiveGaussianImageFilter 有一个布尔标识，允许用户在这两个标准化之间进行选择。这个选择是使用 SetNormalizeAcrossScale( )方式来完成的。可以使用这个标识来分析一个跨越尺度空间的图像。在当前的例子中，这个设置并没有影响，因为我们实际上对 reader

输出的动态范围进行了再标准化。因此我们可以忽略这个标识。

```
filterX->SetNormalizeAcrossScale( false );
filterY->SetNormalizeAcrossScale( false );
```

输入图像可以从另一个滤波器的输出得到。这里，我们用一个图像 reader 来作为获取图像源。将这个图像传递给 x 滤波器，然后给 y 滤波器。分别保留这两个滤波器是因为在尺度空间应用中通常不仅要计算滤波，而且还要计算不同阶的导数和滤波的组合。当使用滤波器来产生中间结果时可能要做一些因式分解。但是在这里这个功能是不太重要的，因此我们仅仅需要在所有方向上来平滑图像。

```
filterX->SetInput( reader->GetOutput( ) );
filterY->SetInput( filterX->GetOutput( ) );
```

现在到了选择高斯函数中用来平滑图像的 σ 值的时候。注意：s 必须传递给两个滤波器，并且 σ 是以毫米来计量的。也就是说，在应用于滤波的过程中时，滤波器将计算在图像中定义的空间值。

```
filterX->SetSigma( sigma );
filterY->SetSigma( sigma );
```

最后通过调用 Update( )方式来运行流水线：

```
filterY->Update( );
```

如图 6-24 所示阐述了这个滤波器对一个 MRI 脑部切片亮度图像作用的效果，使用的 σ 值分别为 3（左图）和 5（右图）。这个图片展示了如何通过选择一个适当的标准差来调节对噪声的衰减。这种可调比例类型的滤波器适合于执行尺度空间分析。

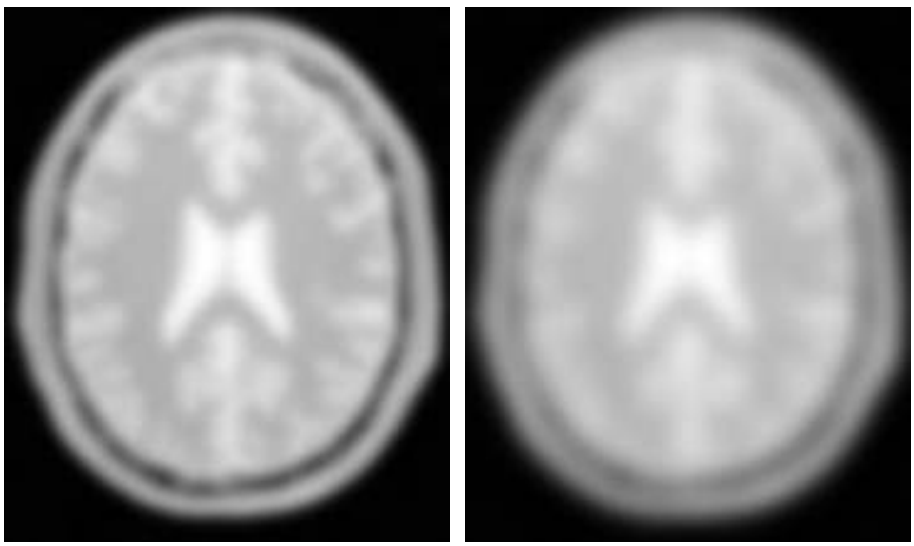


图 6-24　SmoothingRecursiveGaussianImageFilter 对一个 MRI 脑部切片亮度图像作用的效果

RecursiveGaussianFilters 也可以应用于多成员图像。例如上面的滤波器可以应用于以 RGBPixel 作为像素类型的图像。然后对每个成员分别进行滤波。由于 RescaleIntensityImageFilter 并不具有重新调节多成员图像的计算意义，所以它并不能应用于 RGBPixel。

6.7.2 局部模糊

在一些情况下，期望滤波图像中的有限区域或局部使用不同的参数进行计算。接下来的章节将介绍图像中的局部滤波。

高斯模糊图像函数

本小节的源代码在文件 Examples/Filtering/GaussianBlurImageFunction.cxx 中。

6.7.3 保留边缘平滑滤波

1．各向异性扩散绪论

图像去噪（平滑）的缺点是模糊了用来辨别大尺度解剖结构特征的明显的边界（在大多数应用中也同样限制了平滑滤波核的大小）。即使在平滑不删除边界的情况下，它也歪曲了图像的精练的结构并改变了解剖形状的敏感部分。

Perona and Malik 介绍了一种他们称为各向异性扩散 anisotropic diffusion 的线性滤波。各向异性扩散与 Grossberg 早期的工作密切相关，他使用相似的非线性扩散处理人类视觉模型。各向异性扩散（也称为不均匀或可变电导扩散）的动机是：一个高斯平滑图像是以原始图像作为初始条件的热传导方程解的任一个时间层面。从而方程

$$\frac{\partial g(x,y,t)}{\partial t} = \nabla \cdot \nabla g(x,y,t) \tag{6-6}$$

的解为g(x,y,t)=G(√2t)⊗f (x,y)，其中G(σ)是标准差为σ的高斯。方程中g(x,y,0)=f(x,y)是输入图像。

各向异性扩散包括取决于图像微分结构的一个可变电导系数。从而可变电导可以明确表达平滑在图像边缘的限制，像由高梯度强度测量一样，例如：

$$g_t = \nabla \cdot c(|\nabla g|)\nabla g \tag{6-7}$$

其中对于符号变量，我们舍弃g的独立参数而使用和那些表示偏微分的参数相关的下标。函数$c(|\nabla g|)$是减少大的区域$|\nabla g|$的电导的模糊中止，并可以作为众多函数之一。文献中表明

$$c(|\nabla g|) = e^{-\frac{|\nabla g|^2}{2k^2}} \tag{6-8}$$

是非常有效的。注意：电导系数引入了一个自由参数 k，控制边缘对比处理的灵敏性。从而各向异性扩散有两个自由参数：电导系数 k 和时间参数 t，类似于在使用高斯核时 σ 的效果。

等式(6-7)是一个可以使用有限前向差分来解决一个离散网格的偏微分方程。从而平滑的图像只能通过一个迭代处理来得到，而不是通过卷积或不固定线性滤波器。通常实际结果需要的迭代次数是很少的，并且通过在现代、通用目的和单一处理器的计算机上运行代码经过数十秒钟就可以处理大的二维图像。这个技术可以容易地有效地应用于三维图像，但是需要更多的处理时间。

在 20 世纪 90 年代初期，几个研究机构论证了各向异性扩散在医学图像应用的有效性。在这个课题的一系列论文中，Whitaker 介绍了细节解析和实验分析，介绍了一个使得处理更加稳健的电导参数，发明了一些实际消除初始算法视觉失真的方案并将各向异性扩散应用于

向量值图像——一种可以用于向量值医学数据（如人体可视化工程中的彩色 cryosection 数据）的图像处理技术。

对于一个向量值输入 $\vec{F}:U \to \Re^m$ 处理形式为：

$$\vec{F} = \nabla g(D\vec{F})\vec{F} \tag{6-9}$$

其中$\mathcal{D}\vec{F}$是$\vec{F}$的一个差异测量，是向量值图像的梯度强度的一般化，可以合并范围$\vec{F}$上的线性和非线性坐标变换。在这种方式下，多样图像的平滑通过电导和向量值数据相关联，融合不同图像中的信息。从而向量值非线性扩散可以通过一个向量值图像的所有频谱结合低灰度级图像特征（如边缘），以便于对所有的图像频谱保留或增强那些特征。

向量值各向异性扩散对从产生多样值设备而得到的降噪数据非常有效，如 MRI 和彩色摄影。当对一个彩色图像执行非线性扩散时，彩色频谱是分别扩散的，通过电导系数来连接。向量值扩散对处理从不同设备得到的注册数据和从标量值图像得到的降低高阶噪声几何特征和统计学特征都非常有效。

各向异性扩散的输出是降低了噪声和纹理但保留甚至增强边缘的图像或图像集。这些图像对许多包括统计学分类、可视化和几何特征提取的处理是非常有效的。前面的工作已经表明了各向异性扩散在一个很大范围的电导参数上提供了超越医学图像边缘检测线性滤波的许多优势。

由于非线性扩散是首次论证，所以在文献中出现了许多方式[79]。包括可选择的差异测量、方向（如张量）电导系数和水平集解释。

2．梯度各向异性扩散

本小节的源代码在文件 Examples/Filtering/GradientAnisotropicDiffusionImageFilter.cxx 中。

itk::GradientAnisotropicDiffusionImageFilter 执行标量值图像的经典 Perona-Malik 各向异性方程的一个 N 维情况。

这种实现方式中，电导系数作为图像在每个点上的一个梯度强度函数来选择，减少边缘像素的扩散强度。

$$C(x) = e^{-(\frac{|\nabla U(x)|}{K})^2} \tag{6-10}$$

这个方程的数字化实现同 Perona-Malik 论文中描述的非常类似，只是对梯度强度估计采用了一个更加稳健的技术并一般化到 N 维情况。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkGradientAnisotropicDiffusionImageFilter.h"
```

必须基于输入、输出图像需要的像素类型来选择类型。使用像素类型和维来定义图像类型：

```
typedef float InputPixelType;
typedef float OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
```

```
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在使用输入、输出图像类型来实例化滤波器类型。通过 New( )方式来创建滤波器对象：

```
typedef itk::GradientAnisotropicDiffusionImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从另一个滤波器的输出得到。这里使用一个图像 reader 作为图像源：

```
filter->SetInput( reader->GetOutput( ) );
```

这个滤波器需要三个参数，执行迭代的次数、time step 和用在水平集活动计算中的电导系数。分别使用 SetNumberOfIterations( )、SetTimeStep( )和 SetConductanceParameter( )方式来设置这些参数。可以通过调用 Update( )来运行滤波器：

```
filter->SetNumberOfIterations( numberOfIterations );
filter->SetTimeStep( timeStep );
filter->SetConductanceParameter( conductance );
filter->Update( );
```

通常在二维图像中使用的 time step 为 0.25 而三维图像中为 0.125。迭代的次数通常设置为 5，太多的迭代次数将导致过度平滑并相对地增加计算时间。

如图 6-25 所示阐述了这个滤波器对一个脑部 MRI 质子密度图像作用的效果。在这个例子中运行滤波器的 time step 为 0.25 并使用 5 次迭代。图片展示了均匀区域的平滑和边缘的保留。

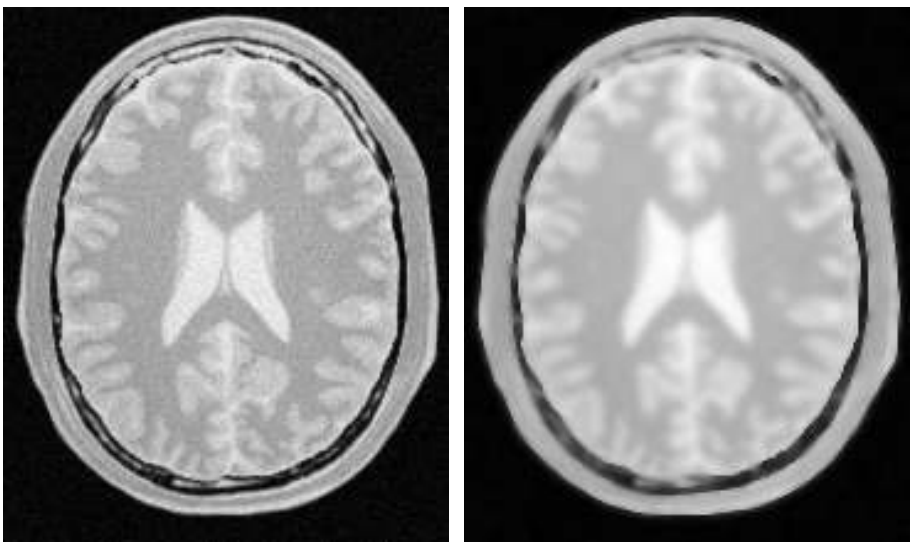


图 6-25　GradientAnisotropicDiffusionImageFilter 对一个脑部 MRI 质子密度图像作用的效果

下面的类提供了相似的功能：

- itk::BilateralImageFilter
- itk::CurvatureAnisotropicDiffusionImageFilter
- itk::CurvatureFlowImageFilter

3．曲率各向异性扩散

本小节的源代码在文件 Examples/Filtering/CurvatureAnisotropicDiffusionImageFilter.cxx 中。

itk::CurvatureAnisotropicDiffusionImageFilter 使用一个改进的曲率扩散方程(MCDE)对图像执行各向异性扩散。

MCDE 并不具备经典各向异性扩散的边缘增强特征，经典的各向异性扩散在一定的条件下进行一个“负”扩散并增强边缘差异。MCDE 形式的方程进行正扩散，仅仅使用不同扩散强度的电导。

和其他非线性扩散技术相比，MCDE 在质量上更好。与经典的 Perona-Malik 扩散方式相比具有更低的敏感度，并更好地保留了图像中的结构细节。在 itkGradientNDAnisotropic DiffusionFunction 中使用这个函数将具有更快的交换速度。这种方式每次迭代将使用大约与通常相比两倍的时间，然而可能使用更少的迭代就可以得到一个可接受的结果。

MCDE 方程为：

$$f_t = |\nabla f| \nabla \cdot c(|\nabla f|) \frac{\nabla f}{|\nabla f|} \tag{6-11}$$

其中电导改进曲率系数为：

$$\nabla \cdot \frac{\nabla f}{|\nabla f|} \tag{6-12}$$

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkCurvatureAnisotropicDiffusionImageFilter.h"
```

必须基于输入、输出图像需要的像素类型来选择类型。使用像素类型和维来定义图像类型：

```
typedef float InputPixelType;
typedef float OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在使用输入、输出图像类型来实例化滤波器类型。通过 New( )方式来创建滤波器对象：

```
typedef itk::CurvatureAnisotropicDiffusionImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从另一个滤波器的输出得到。这里使用一个图像 reader 作为图像源：

```
filter->SetInput( reader->GetOutput( ) );
```

这个滤波器需要三个参数，执行迭代的次数、time step 和用在水平集活动计算中的电导系数。分别使用 SetNumberOfIterations( )、SetTimeStep( )和 SetConductanceParameter( )方式来设置这些参数。可以通过调用 Update( )来运行滤波器：

```
filter->SetNumberOfIterations( numberOfIterations );
filter->SetTimeStep( timeStep );
filter->SetConductanceParameter( conductance );
```

```
if (useImageSpacing)
{
filter->UseImageSpacingOn( );
}
filter->Update( );
```

通常在二维图像中使用的 time step 为 0.25 而三维图像中为 0.125。迭代的次数通常设置为 5，太多的迭代次数将导致过度平滑并相对地增加计算时间。通常使用的电导系数是 3.0 左右。

如图 6-26 所示阐述了这个滤波器对一个脑部 MRI 质子密度图像作用的效果。在这个例子中运行滤波器的 time step 为 0.25 并使用 5 次迭代。图片展示了均匀区域的平滑和边缘的保留。

下面的类提供了相似的功能：

- itk::BilateralImageFilter
- itk::CurvatureFlowImageFilter
- itk::GradientAnisotropicDiffusionImageFilter

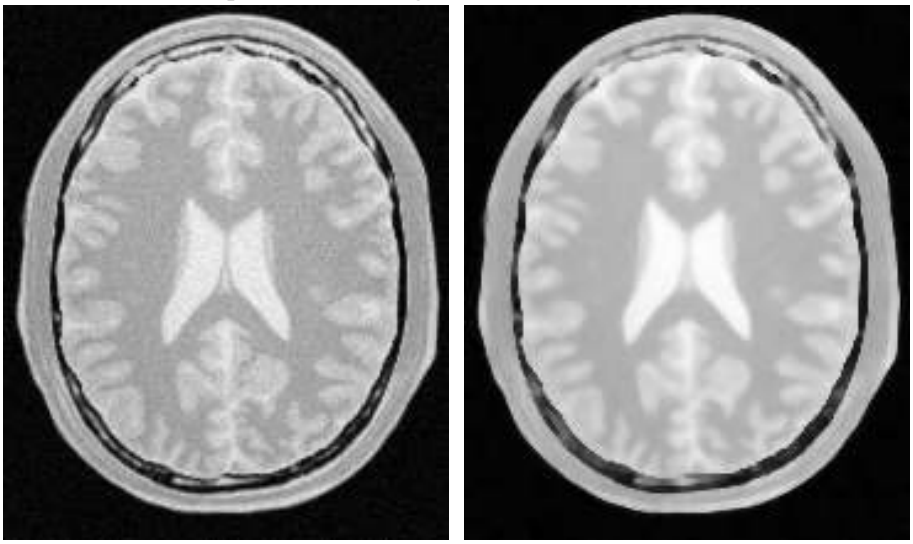


图 6-26　CurvatureAnisotropicDiffusionImageFilter 对一个脑部 MRI 质子密度图像作用的效果

4．曲线流

本小节的源代码在文件 Examples/Filtering/CurvatureFlowImageFilter.cxx 中。

itk::CurvatureFlowImageFilter 以一种和经典各向异性扩散方式类似的方式来执行保留边缘平滑滤波。这个滤波器使用图像中的一系列等亮度轮廓表达的水平集，水平集是由一个特定亮度像素组成的一个集合。然后用一个扩散方程来控制水平集函数，其中速度和轮廓曲线是成比例的：

$$I_t = \kappa |\nabla I| \tag{6-13}$$

其中 k 是曲率。

高曲率的区域比低曲率区域扩散得更快。小的锯齿状失真将迅速消失，而大范围的截面

将活动减慢，从而保留对象的尖锐边界。需要注意的是：尽管边界的活动迟缓，但是仍然会发生一些扩散。因此，连续使用曲线流方案将导致每个轮廓收缩为一个点或者消失。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkCurvatureFlowImageFilter.h"
```

必须基于输入、输出图像需要的像素类型来选择类型：

```
typedef float InputPixelType;
typedef float OutputPixelType;
```

使用它们来对输入、输出图像进行实例化：

```
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在同时使用输入、输出图像类型实例化 CurvatureFlow 滤波器：

```
typedef itk::CurvatureFlowImageFilter<
InputImageType, OutputImageType > FilterType;
```

通过调用 New( )方式来创建一个滤波器对象并将结果指向一个 itk::SmartPointer：

```
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从另一个滤波器的输出得到。这里使用一个图像 reader 作为图像源：

```
filter->SetInput( reader->GetOutput( ) );
```

CurvatureFlow 滤波器需要两个参数：执行迭代的次数和用在水平集活动计算中的 time step。分别使用 SetNumberOfIterations( )和 SetTimeStep( )方式来设置这些参数。然后可以通过调用 Update( )来运行滤波器：

```
filter->SetNumberOfIterations( numberOfIterations );
filter->SetTimeStep( timeStep );
filter->Update( );
```

通常在二维图像中使用的 time step 为 0.125 而三维图像中为 0.0625。迭代的次数通常设置为 10 左右，太多的迭代次数将导致过度平滑并相对地增加计算时间。这个滤波器不用保留边缘行为，在边缘将发生一些退化并随着迭代次数的增加而增大。

如果这个滤波器的输出已经连接到流水线下游的其他滤波器，更新任何下游的滤波器将触发这个滤波器的运行。例如在曲线流滤波器后可能使用的一个 writer 滤波器：

```
rescaler->SetInput( filter->GetOutput( ) );
writer->SetInput( rescaler->GetOutput( ) );
writer->Update( );
```

如图 6-27 所示阐述了这个滤波器对一个脑部 MRI 质子密度图像作用的效果。在这个例子中运行滤波器的 time step 为 0.25 并使用 10 次迭代。图片展示了均匀区域的平滑和边缘的保留。

下面的类提供了相似的功能：

- itk::GradientAnisotropicDiffusionImageFilter
- itk::CurvatureAnisotropicDiffusionImageFilter
- itk::BilateralImageFilter

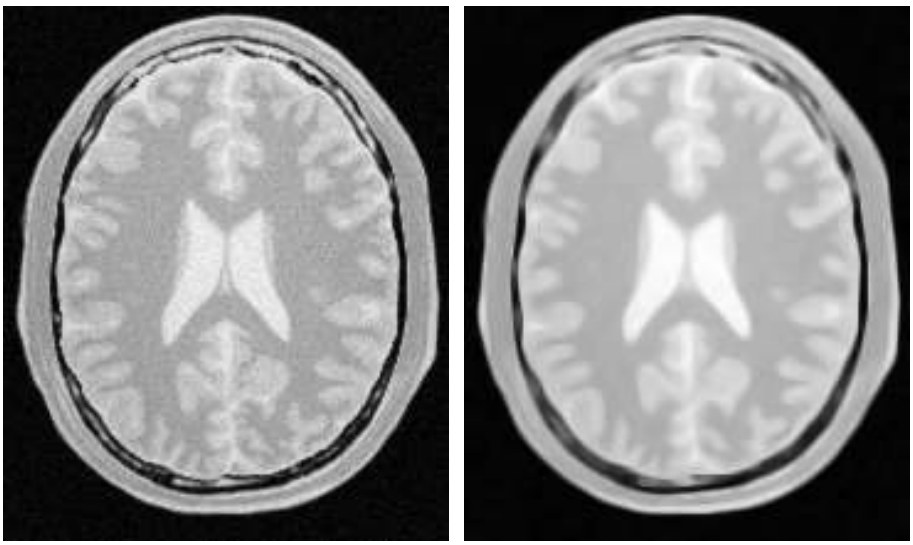


图 6-27　CurvatureFlowImageFilter 对一个脑部 MRI 质子密度图像作用的效果

5．最大最小曲线流

本小节的源代码在文件 Examples/Filtering/MinMaxCurvatureFlowImageFilter.cxx 中。

MinMax 曲线流滤波器应用曲线流算法的一个变量，其中扩散的开启和关闭取决于去除噪声的范围大小。活动速率在 min(*k*,0)和 max(*k*,0)之间转变，如：

$$I_t = F|\nabla I| \tag{6-14}$$

其中 F 的定义为：

$$F = \begin{cases} \max(\kappa,0) : \text{average} < \text{阈值} \\ \min(\kappa,0) : \text{average} \geq \text{阈值} \end{cases} \tag{6-15}$$

Average 是一个用户指定半径的像素邻域计算出来的亮度均值。半径的选择应覆盖要去除的噪声。Threshold 是作为沿和局部邻域的极值的梯度垂直的方向上的亮度均值来计算的。

F= max(*k*,0)的速率将收缩在主要亮区域中的小块暗区域；相反的，F= min(*k*,0)的速率将收缩在主要暗区域中的亮区域。对比邻域均值和阈值来选择正确的速率函数。这个转换阻止了简单曲线流方法中不必要的扩散。

如图 6-28 所示显示了计算中的主要成员。这个正方形像素集表示了用于计算亮度均值的邻域。灰度像素是那些接近于梯度垂直方向上的像素。和邻域范围相交的像素用来计算上面方程中的阈值。由用户来选择邻域的整数半径。

使用 itk::MinMaxCurvatureFlowImageFilter 的第一步是包含它的头文件：

```
#include "itkMinMaxCurvatureFlowImageFilter.h"
```

必须基于输入输出图像需要的像素类型来选择类型。实例化输入、输出图像类型：

```
typedef float InputPixelType;
typedef float OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

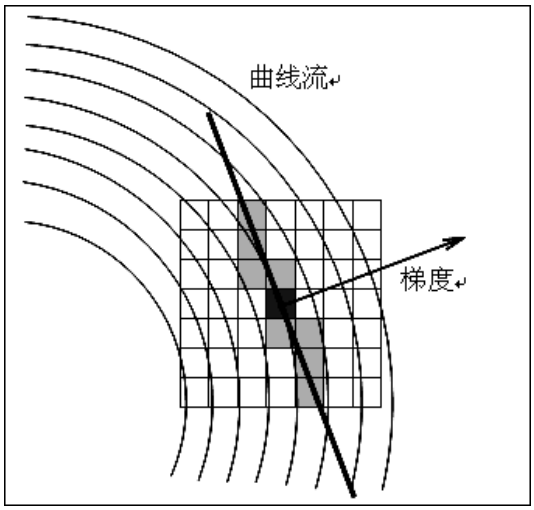


图 6-28　最大最小曲线流计算中的相关成员

现在使用输入、输出图像类型来对 itk::MinMaxCurvatureFlowImageFilter 类型进行实例化。然后使用 New( )方式来创建滤波器：

```
typedef itk::MinMaxCurvatureFlowImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从另一个滤波器的输出得到。这里使用一个图像 reader 作为图像源：

```
filter->SetInput( reader->GetOutput( ) );
```

itk::MinMaxCurvatureFlowImageFilter 需要 CurvatureFlow 图像的两个常规参数：执行迭代的次数和用在水平集活动计算中的 time step。在它们之上还需要邻域的半径。最后一个参数是使用 SetStencilRadius( )方式来传递的。注意：由于半径对应于邻域中的像素数量，所以必须提供整数作为半径。然后可以通过调用 Update( )来运行滤波器：

```
filter->SetTimeStep( timeStep );
filter->SetNumberOfIterations( numberOfIterations );
filter->SetStencilRadius( radius );
filter->Update( );
```

通常在二维图像中使用的 time step 为 0.125 而三维图像中为 0.0625。迭代的次数通常设置为 10 左右，太多的迭代次数将导致过度平滑并相对地增加计算时间。模板的半径通常为 1。这个滤波器不用保留边缘行为，在边缘将发生一些退化并随着迭代次数的增加而增大。

如果这个滤波器的输出已经连接到流水线下游的其他滤波器，更新任何下游的滤波器将触发这个滤波器的运行。例如在曲线流滤波器后可能使用的一个 writer 滤波器：

```
rescaler->SetInput( filter->GetOutput( ) );
writer->SetInput( rescaler->GetOutput( ) );
writer->Update( );
```

如图 6-29 所示阐述了这个滤波器对一个脑部 MRI 质子密度图像作用的效果。在这个例

子中运行滤波器的 time step 为 0.25，并使用 10 次迭代和邻域半径为 1。图片展示了均匀区域的平滑和边缘的保留。同样也值得注意的是：图片中的结果比如图 6-27 所示使用简单曲线流的同样例子得到的结果边缘更尖锐。

下面的类提供了相似的功能：

- itk::CurvatureFlowImageFilter

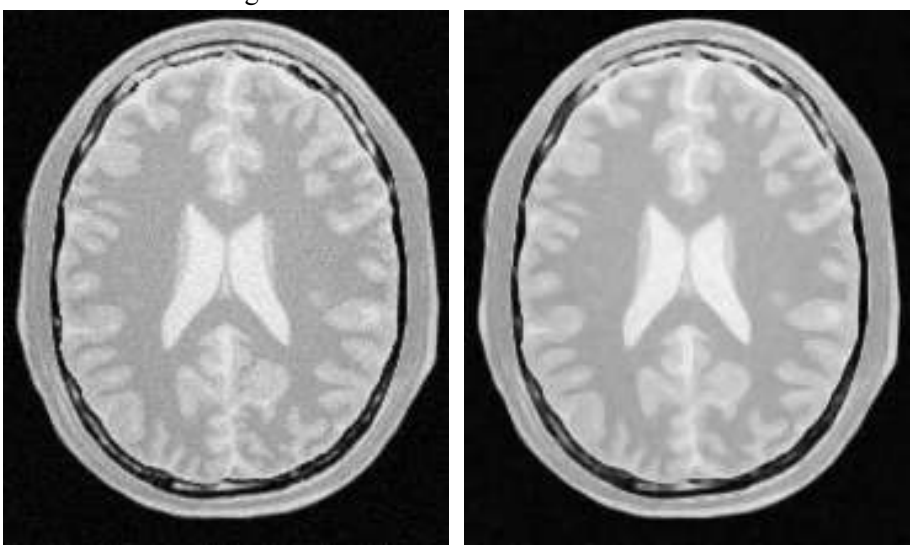


图 6-29　MinMaxCurvatureFlowImageFilter 对一个脑部 MRI 质子密度图像作用的效果

6．双边滤波器

本小节的源代码在文件 Examples/Filtering/BilateralImageFilter.cxx 中。

itk::BilateralImageFilter 通过同时使用邻域的 domain 和 range 来执行平滑滤波。使用接近于图像 domain 中的一个像素和图像 range 中的一个类似像素来计算滤波值。使用两个高斯核(一个是图像 domain，另一个是图像 range)来平滑图像，结果是一个仍保留边缘而平滑了均匀区域的图像。结果和各向异性扩散很相似，但不需要进行迭代。双边滤波的另一个优点是可以对核使用任何以米为单位的距离来平滑图像 range。双边滤波在保持边缘时具有通过梯度的一个次序来降低图像中的噪声的能力。这里使用的双边操作是由 Tomasi andManduchi 所描述的(Bilateral Filtering for Gray and Color Images，IEEE ICCV.，1998)。

这个滤波操作可以由下面的等式来描述：

$$h(x) = k(x)^{-1} \int_w f(w) c(x,w) s(f(x), f(w)) \mathrm{d}w \tag{6-16}$$

其中 x 控制一个 ND 点的坐标，f(x)是输入图像，而 h(x)为输出图像。卷积核 c( )和 s( )分别和空间和亮度 domain 相关联。ND 是基于 w 计算的，w 是定位在 x 点的一个像素邻域。标准化因子 k(x)是按下式来计算的：

$$k(x) = \int_w c(x,w) s(f(x), f(w)) \mathrm{d}w \tag{6-17}$$

这个滤波器的默认实现是使用 c( )和 s( )的高斯核。C 核可以使用下式来描述：

$$c(x,w) = \mathrm{e}^{(\frac{\|x-w\|^2}{\sigma_c^2})} \tag{6-18}$$

其中 σ_C 是由用户来提供的并定义了接近像素邻域的程度以便于计算输出值。s 核使用下式来描述：

$$s(f(x), f(w)) = e^{(\frac{(f(x)-f(w))^2}{\sigma_s^2})} \tag{6-19}$$

其中 σ_S 是由用户来提供的并定义了接近邻域亮度的程度以便于计算输出值。

使用这个滤波器的第一步需要包含它的滤波器：

```
#include "itkBilateralImageFilter.h"
```

使用像素类型和维来实例化图像类型：

```
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

现在使用输入、输出图像类型来实例化双边滤波器类型并创建滤波器对象：

```
typedef itk::BilateralImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从另一个滤波器的输出得到。这里使用一个图像 reader 作为图像源：

```
filter->SetInput( reader->GetOutput( ) );
```

双边滤波器需要两个参数。第一个是在图像亮度上应用于高斯核的 σ。第二个是空间域中沿每个维上使用的 σ_S 集。第二个参数是以一个浮点型或双精度数据值序列来提供的。序列维和图像维相匹配。这种机制可以增强沿某些方向的一致性。例如：可以在沿 X 方向上比沿 Y 方向做更多的平滑滤波。

在接下来的例子代码中，从命令行得到 σ 值。注意使用 ImageType::ImageDimension 在编译时间访问图像维。

```
const unsigned int Dimension = InputImageType::ImageDimension;
double domainSigmas[ Dimension ];
for(unsigned int i=0; i<Dimension; i++)
{
domainSigmas[i] = atof( argv[3] );
}
const double rangeSigma = atof( argv[4] );
```

使用 SetRangeSigma( )和 SetDomainSigma( )方式来设置滤波器参数：

```
filter->SetDomainSigma( domainSigmas );
filter->SetRangeSigma( rangeSigma );
```

这里将滤波器的输出连接到一个亮度调节滤波器，然后连接到一个 writer。在 writer 上调用 Update( )来触发这两个滤波器的运行：

```
rescaler->SetInput( filter->GetOutput( ) );
writer->SetInput( rescaler->GetOutput( ) );
writer->Update( );
```

如图 6-30 所示阐述了这个滤波器对一个脑部 MRI 质子密度图像作用的效果。在这个例子中运行滤波器的 sigma 为 5.0 并使用一个 domain's 值为 6.0。图片展示了均匀区域的平滑和边缘的保留。

下面的类提供了相似的功能：

- itk::GradientAnisotropicDiffusionImageFilter
- itk::CurvatureAnisotropicDiffusionImageFilter
- itk::CurvatureFlowImageFilter

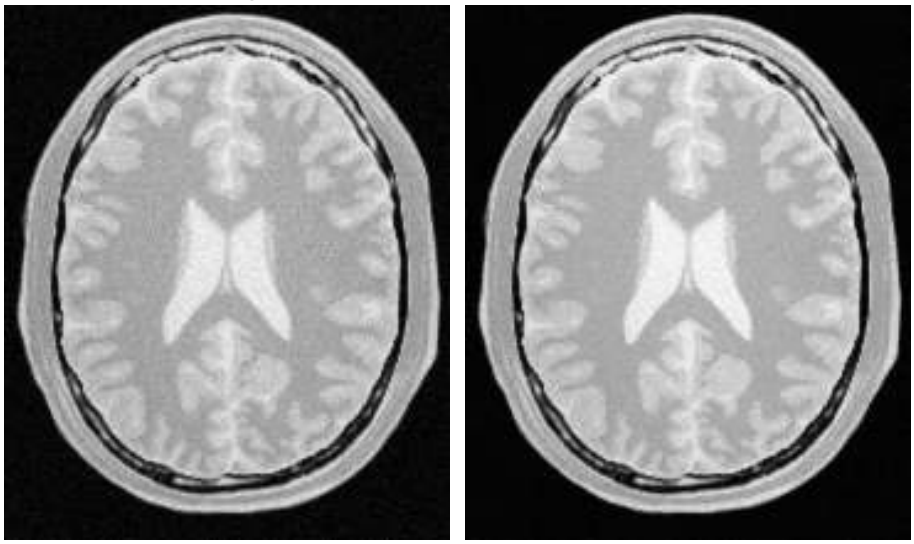


图 6-30　BilateralImageFilter 对一个脑部 MRI 质子密度图像作用的效果

6.7.4　向量图像中的保留边缘平滑滤波

各向异性扩散也可以应用于向量像素的图像。在这种情况下扩散是在每个向量成员上独立计算的。接下来都是各向异性扩散对向量图像实现的类。

1．向量梯度各向异性扩散

本小节的源代码在文件 Examples/Filtering/VectorGradientAnisotropicDiffusionImageFilter.cxx 中。

itk::VectorGradientAnisotropicDiffusionImageFilter 执行向量值图像的经典 Perona-Malik 各向异性方程的一个 N 维情况。通常在向量值扩散中，使用连接到向量成员的一个电导逐个对向量成员进行扩散。在 6.7.3 小节中阐述了扩散方程。

这个滤波器是按处理 itk::Vector 类型的图像来设计的。代码是依照各种类型定义和向量中定义的操作符来编写的。然而对其他图像应用这个滤波器，由用户定义的相关类型和操作类型都是很有效的。作为一个通用的规则，下面向量的例子是基于用户定义的数据类型的。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkVectorGradientAnisotropicDiffusionImageFilter.h"
```

必须基于输入、输出图像需要的像素类型来选择类型。使用像素类型和维来定义图像类型：

```
typedef float InputPixelType;
typedef itk::CovariantVector<float,2> VectorPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< VectorPixelType, 2 > VectorImageType;
```

现在使用输入、输出图像类型来实例化滤波器类型。通过 New( )方式来创建滤波器对象：

```
typedef itk::VectorGradientAnisotropicDiffusionImageFilter<
VectorImageType, VectorImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从另一个滤波器的输出得到。这里使用一个图像 reader 作为图像源并通过一个梯度滤波器传递它的数据以便生成一个向量图像。

```
gradient->SetInput( reader->GetOutput( ) );
filter->SetInput( gradient->GetOutput( ) );
```

这个滤波器需要两个参数，执行迭代的次数和用在水平集活动计算中的 time step。分别使用 SetNumberOfIterations( )和 SetTimeStep( )方式来设置这些参数。可以通过调用 Update( )来运行滤波器：

```
filter->SetNumberOfIterations( numberOfIterations );
filter->SetTimeStep( timeStep );
filter->SetConductanceParameter(1.0);
filter->Update( );
```

通常在二维图像中使用的 time step 为 0.125 而三维图像中为 0.0625。迭代的次数通常设置为 5 左右，太多的迭代次数将导致过度平滑并相对地增加计算时间。

如图 6-31 所示阐述了这个滤波器对一个脑部 MRI 质子密度图像作用的效果。图像展示了应用这个滤波器前（左图）后（右图）的梯度的 X 分量。在这个例子中运行滤波器的 time step 为 0.25 并使用 5 次迭代。

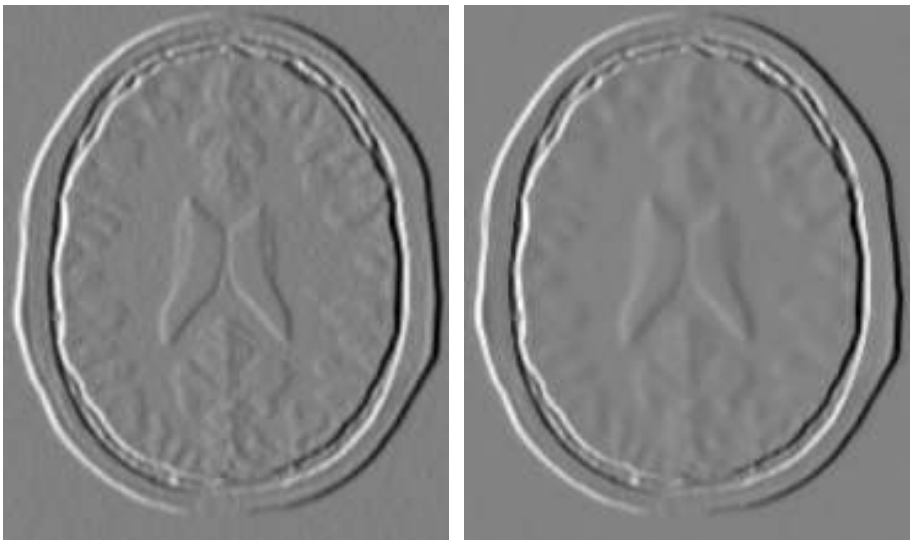


图 6-31　VectorGradientAnisotropicDiffusionImageFilter 对一个脑部 MRI 质子密度图像作用的效果

2．向量曲线各向异性扩散

本小节的源代码在文件 Examples/Filtering/VectorCurvatureAnisotropicDiffusionImageFilter.cxx 中。

itk::VectorCurvatureAnisotropicDiffusionImageFilter 使用一个改进的曲线扩散方程(MCDE)对一个向量图像执行各向异性扩散。这里的 MCDE 和 6.7.3 小节中描述的相同。

通常向量值扩散中，向量成员使用一个和它的成员相关联的电导系数来逐个进行扩散。

这个滤波器是按处理 itk::Vector 类型的图像来设计的。代码是依照各种类型定义和向量中定义的操作符来编写的。然而对其他图像应用这个滤波器，由用户定义的相关类型定义和操作的类型都是很有效的。作为一个通用的规则，下面向量的例子是基于你定义的数据类型的。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkVectorCurvatureAnisotropicDiffusionImageFilter.h"
```

必须基于输入、输出图像需要的像素类型来选择类型。使用像素类型和维来定义图像类型：

```
typedef float InputPixelType;
typedef itk::CovariantVector<float,2> VectorPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< VectorPixelType, 2 > VectorImageType;
```

现在使用输入、输出图像类型来实例化滤波器类型。通过 New( )方式来创建滤波器对象：

```
typedef itk::VectorCurvatureAnisotropicDiffusionImageFilter<
VectorImageType, VectorImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从另一个滤波器的输出得到。这里使用一个图像 reader 作为图像源并通过一个梯度滤波器传递它的数据以便生成一个向量图像。

```
gradient->SetInput( reader->GetOutput( ) );
filter->SetInput( gradient->GetOutput( ) );
```

这个滤波器需要两个参数，执行迭代的次数和用在水平集活动计算中的 time step。分别使用 SetNumberOfIterations( )和 SetTimeStep( )方式来设置这些参数。可以通过调用 Update( )来运行滤波器：

```
filter->SetNumberOfIterations( numberOfIterations );
filter->SetTimeStep( timeStep );
filter->SetConductanceParameter(1.0);
filter->Update( );
```

通常在二维图像中使用的 time step 为 0.125 而三 3 维图像中为 0.0625。迭代的次数通常设置为 5 左右，太多的迭代次数将导致过度平滑并相对地增加计算时间。

如图 6-32 所示阐述了这个滤波器对一个脑部 MRI 质子密度图像作用的效果。图像展示了应用这个滤波器前（左图）后（右图）的梯度的 X 分量。在这个例子中运行滤波器的 time step 为 0.25 并使用 5 次迭代。

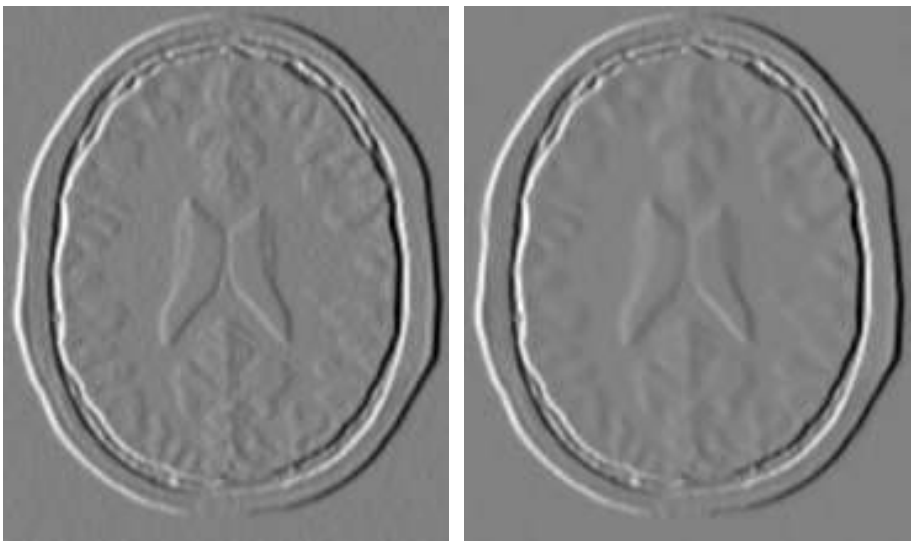


图 6-32　VectorCurvatureAnisotropicDiffusionImageFilter 对一个脑部 MRI 质子密度图像作用的效果

6.7.5　彩色图像中的保留边缘平滑滤波

1．梯度各向异性扩散

本小节的源代码在文件Examples/Filtering/RGBGradientAnisotropicDiffusionImageFilter.cxx 中。

向量各向异性扩散方法同样可以应用于彩色图像。在向量情况中，每个 RGB 分量是独自扩散的。下面的例子阐述了向量曲线各向异性扩散滤波器在基于 itk::RGBPixel 类型的图像上的用法。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkVectorGradientAnisotropicDiffusionImageFilter.h"
```

同样也需要包含图像和 RGBPixel 类型的头文件：

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

我们期望计算的 RGB 图像是使用浮点型数据类型的。然而，输入、输出图像通常使用的是无符号字符型 RGB 分量。在将彩色分量沿流水线写入一个文件之前投射为彩色分量类型是很有必要的。使用 itk::VectorCastImageFilter 来达到这一目的：

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkVectorCastImageFilter.h"
```

使用像素类型和维来定义图像类型：

```
typedef itk::RGBPixel< float > InputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
```

现在实例化滤波器类型并通过 New( )方式来创建一个滤波器对象：

```
typedef itk::VectorGradientAnisotropicDiffusionImageFilter<
```

```
InputImageType, InputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从另一个滤波器的输出得到。这里使用一个图像 reader 作为图像源：

```
typedef itk::ImageFileReader< InputImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New( );
reader->SetFileName( argv[1] );
filter->SetInput( reader->GetOutput( ) );
```

这个滤波器需要两个参数，执行迭代的次数和用在水平集活动计算中的 time step。分别使用 SetNumberOfIterations( )和 SetTimeStep( )方式来设置这些参数。可以通过调用 Update( )来运行滤波器：

```
filter->SetNumberOfIterations( numberOfIterations );
filter->SetTimeStep( timeStep );
filter->SetConductanceParameter(1.0);
filter->Update( );
```

现在通过使用 itk::VectorCastImageFilter 将滤波器输出投射到无符号字符型 RGB 分量：

```
typedef itk::RGBPixel< unsigned char > WritePixelType;
typedef itk::Image< WritePixelType, 2 > WriteImageType;
typedef itk::VectorCastImageFilter<
InputImageType, WriteImageType > CasterType;
CasterType::Pointer caster = CasterType::New( );
```

最后可以实例化 writer 类型。创建一个 writer 并连接到投射滤波器的输出：

```
typedef itk::ImageFileWriter< WriteImageType > WriterType;
WriterType::Pointer writer = WriterType::New( );
caster->SetInput( filter->GetOutput( ) );
writer->SetInput( caster->GetOutput( ) );
writer->SetFileName( argv[2] );
writer->Update( );
```

如图 6-33 所示阐述了这个滤波器对一个从人体可视化数据集低温学部分而来的 RGB 图像作用的效果。在这个例子中运行滤波器的 time step 为 0.125 并使用 20 次迭代。这个图像为 570×670 像素，使用 Pentium4 2GHz 处理需要 4 分钟。

2．曲线各向异性扩散

本小节的源代码在文件 Examples/Filtering/RGBCurvatureAnisotropicDiffusionImageFilter.cxx 中。

向量各向异性扩散方法同样可以应用于彩色图像。在向量情况中，每个 RGB 分量是独自扩散的。下面的例子阐述了 itk::VectorCurvatureAnisotropicDiffusionImageFilter 在一个基于 itk::RGBPixel 类型的图像上的用法。

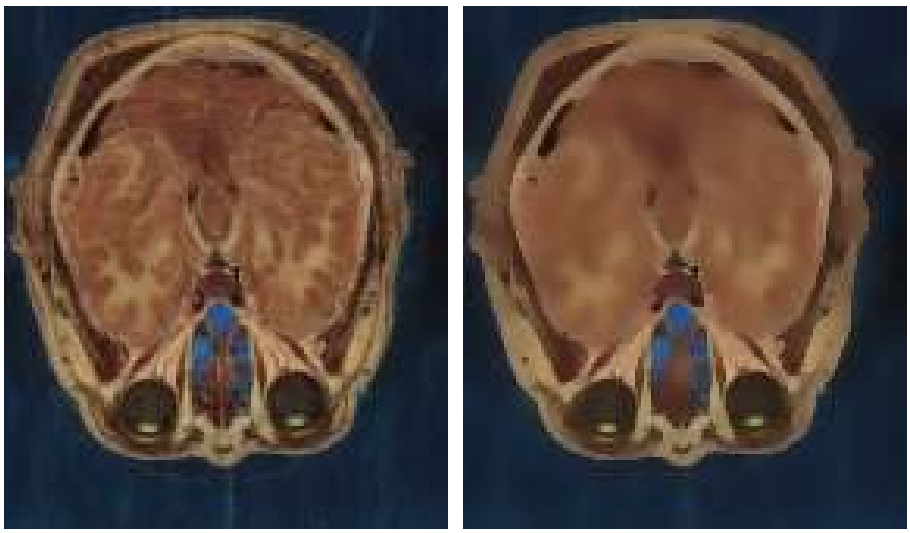


图 6-33　VectorGradientAnisotropicDiffusionImageFilter 对一个从人体可视化数据集低温学部分而来的 RGB 图像作用的效果

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkVectorCurvatureAnisotropicDiffusionImageFilter.h"
```

同样也需要包含图像和 RGBPixel 类型的头文件：

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

我们期望计算的 RGB 图像是使用浮点型数据类型的。然而，输入输出图像通常使用的是无符号字符型 RGB 分量。在将彩色分量沿流水线写入一个文件之前投射为彩色分量类型是很有必要的。使用 itk::VectorCastImageFilter 来达到这一目的。

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkVectorCastImageFilter.h"
```

使用像素类型和维来定义图像类型：

```
typedef itk::RGBPixel< float > InputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
```

现在实例化滤波器类型并通过 New( )方式来创建一个滤波器对象：

```
typedef itk::VectorCurvatureAnisotropicDiffusionImageFilter<
InputImageType, InputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

输入图像可以从另一个滤波器的输出得到。这里使用一个图像 reader 作为图像源：

```
typedef itk::ImageFileReader< InputImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New( );
reader->SetFileName( argv[1] );
filter->SetInput( reader->GetOutput( ) );
```

这个滤波器需要两个参数，执行迭代的次数和用在水平集活动计算中的 time step。分别使用 SetNumberOfIterations( )和 SetTimeStep( )方式来设置这些参数。可以通过调用 Update( )来运行滤波器：

```
filter->SetNumberOfIterations( numberOfIterations );
filter->SetTimeStep( timeStep );
filter->SetConductanceParameter(1.0);
filter->Update( );
```

现在通过使用 VectorCastImageFilter 将滤波器输出投射到无符号字符型 RGB 分量：

```
typedef itk::RGBPixel< unsigned char > WritePixelType;
typedef itk::Image< WritePixelType, 2 > WriteImageType;
typedef itk::VectorCastImageFilter<
InputImageType, WriteImageType > CasterType;
CasterType::Pointer caster = CasterType::New( );
```

最后可以实例化 writer 类型。创建一个 writer 并连接到投射滤波器的输出：

```
typedef itk::ImageFileWriter< WriteImageType > WriterType;
WriterType::Pointer writer = WriterType::New( );
caster->SetInput( filter->GetOutput( ) );
writer->SetInput( caster->GetOutput( ) );
writer->SetFileName( argv[2] );
writer->Update( );
```

如图 6-34 所示阐述了这个滤波器对一个从人体可视化数据集低温学部分而来的 RGB 图像作用的效果。在这个例子中运行滤波器的 time step 为 0.125 并使用 20 次迭代。这个图像为 570×670 像素，使用 Pentium4 2GHz 处理需要 4 分钟。

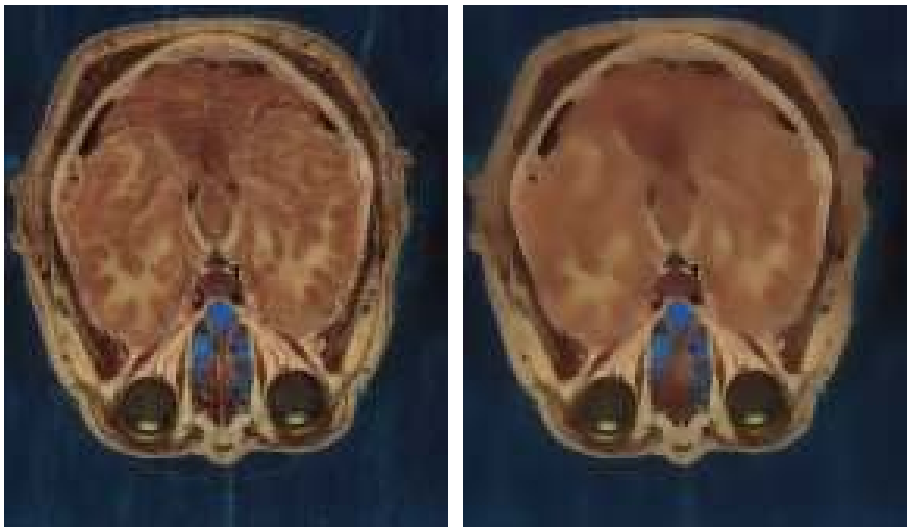


图 6-34　VectorCurvatureAnisotropicDiffusionImageFilter 对一个从人体可视化数据集低温学部分而来的 RGB 图像作用的效果

如图 6-35 所示对比了梯度和曲线各向异性扩散滤波器在和图 6.34 中使用的低温学切片相同的图像的一个小区域上作用的效果。在这个图片中使用的区域仅仅是 127×162 像素的，在同样的平台上计算需要 14 秒的时间。

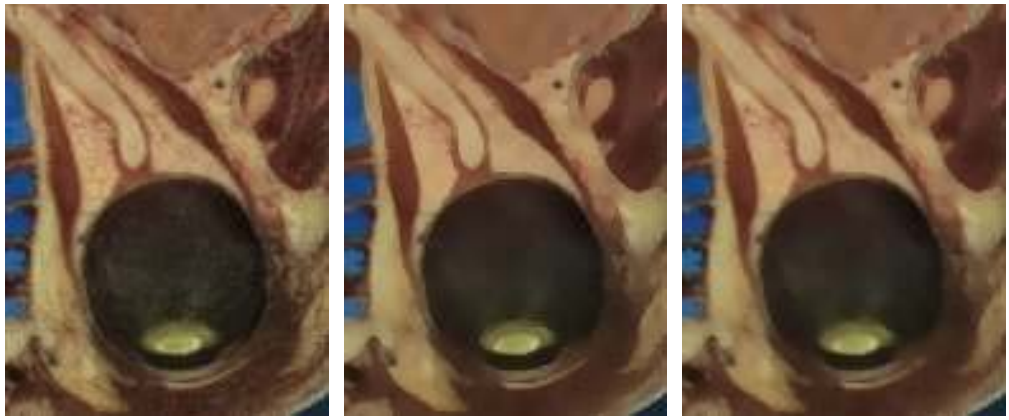


图 6-35　梯度(中图)和曲线(右图)各向异性扩散滤波器作用效果对比；左图是原始图像

6.8　距离映射

本节的源代码在文件 Examples/Filtering/DanielssonDistanceMapImageFilter.cxx 中。

这个例子阐述了 itk::DanielssonDistanceMapImageFilter 的用法。这个滤波器使用由 Danielssion 发展的算法从输入图像生成一个距离映射。作为二级输出，产生了输入成员的一个 Voronoi 分割，也即产生一个由最近点的距离向量组成的向量图像。假设这个映射的输入是输入图像上的一系列点。尽管每个点或像素都分享同样的灰度级值，但是认为每个点或像素是一个独立的实体。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkDanielssonDistanceMapImageFilter.h"
```

然后我们必须决定输入、输出图像使用的像素类型。由于输出将包含以像素衡量的距离，所以像素类型应该至少可以表示图像的宽度或在 N-D 系统中沿每个维的最大范围。现在使用输入、输出图像各自的像素类型和维来定义它们的图像类型：

```
typedef unsigned char InputPixelType;
typedef unsigned short OutputPixelType;
typedef itk::Image< InputPixelType, 2 > InputImageType;
typedef itk::Image< OutputPixelType, 2 > OutputImageType;
```

使用上面定义的输入、输出图像实例化滤波器对象，并使用 New( )方式来创建一个滤波器对象：

```
typedef itk::DanielssonDistanceMapImageFilter<
InputImageType, OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

从一个 reader 来得到这个滤波器的输入并将它的输出传递给一个 itk::RescaleIntensity

ImageFilter，然后传递给一个 writer：

```
filter->SetInput( reader->GetOutput( ) );
scaler->SetInput( filter->GetOutput( ) );
writer->SetInput( scaler->GetOutput( ) );
```

必须指定输入图像的类型。在这种情况下，选择一个二值图像：

```
filter->InputIsBinaryOn( );
```

如图 6-36 所示阐述了这个滤波器在一个具有一系列点集的二值图像上作用的效果。左图是输入图像，中图是距离映射而右图是 Voronoi 分割。这个滤波器在 N 维情况下计算距离映射，因此具有生成 Voronoi 分割的能力。

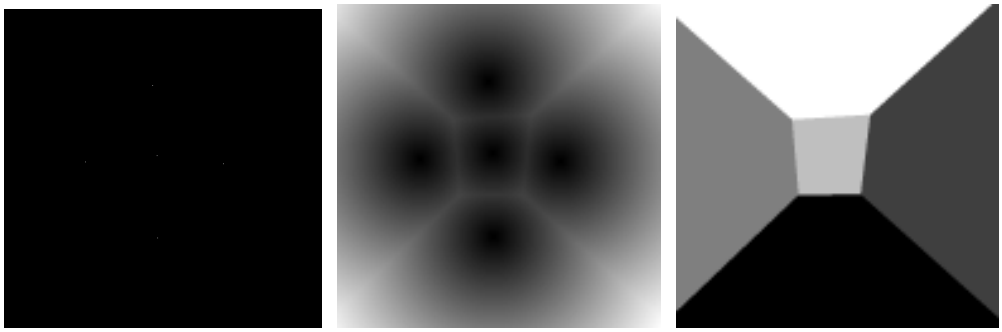


图 6-36　DanielssonDistanceMapImageFilter 输出；像素集、距离映射和 Voronoi 分割

Voronoi 映射是通过 GetVoronoiMap( )方式来得到的。在下面的代码中我们将这个输出连接到亮度调节并将结果保存到一个文件中。

```
scaler->SetInput( filter->GetVoronoiMap( ) );
writer->SetFileName( voronoiMapFileName );
writer->Update( );
```

距离滤波器也产生一个 itk::Offset 像素的图像来表示在这种情况下到最近的对象的向量距离。这个输出图像的类型是通过滤波器类型的 VectorImageType 特征来定义的。

```
typedef FilterType::VectorImageType OffsetImageType;
```

在下面的几行代码中我们可以使用这个类型来实例化一个 itk::ImageFileWriter 类型并创建了这个类的一个对象：

```
typedef itk::ImageFileWriter< OffsetImageType > WriterOffsetType;
WriterOffsetType::Pointer offsetWriter = WriterOffsetType::New( );
```

距离滤波器的输出可以作为输入连接到 writer：

```
offsetWriter->SetInput( filter->GetVectorDistanceMap( ) );
```

通过调用 Update( )方式来触发 writer 的执行。由于这种方式可以抛出异常，所以必须将它放在一个 try/catch 模块中：

```
try
{
offsetWriter->Update( );
}
```

```
catch( itk::ExceptionObject exp )
{
std::cerr << "Exception caught !" << std::endl;
std::cerr << exp << std::endl;
}
```

注意：仅仅只有 itk::MetaImageIO 类支持读写 itk::Offset 像素类型的图像。.

本小节的源代码在文件 Examples/Filtering/SignedDanielssonDistanceMapImageFilter.cxx 中。

这个例子阐述了 itk::SignedDanielssonDistanceMapImageFilter 的用法。这个滤波器通过运行两次 Danielsson 距离映射来生成一个距离映射，这两次映射，一次是对输入图像，而另一次是对翻转图像。

使用这个滤波器的第一步是包含它的头文件：

```
#include "itkSignedDanielssonDistanceMapImageFilter.h"
```

然后我们必须决定输入、输出图像使用的像素类型。由于输出将包含以像素衡量的距离，所有像素类型应该至少可以表示图像的宽度或在 N-D 系统中沿每个维的最大范围。现在使用输入、输出图像各自的像素类型和维来定义它们的图像类型：

```
typedef unsigned char InputPixelType;
typedef float OutputPixelType;
const unsigned int Dimension = 2;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

与前面例子相比唯一的变换就是将 DanielssonDistanceMapImageFilter 换成了 SignedDanielssonDistanceMapImageFilter。

```
typedef itk::SignedDanielssonDistanceMapImageFilter<
InputImageType,
OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

内部被认为是负距离，而外部是正距离。使用 InsideIsPositive(bool)函数可以改变这一规则。

如图 6-37 所示阐述了这个滤波器作用的效果，显示了输入图像和距离映射。

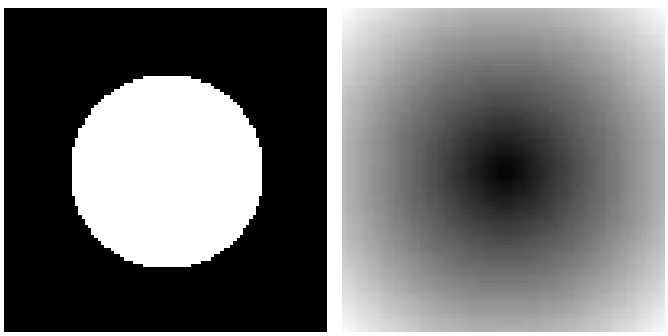


图 6-37　SignedDanielssonDistanceMapImageFilter 应用于一个圆的二值图像；为了显示，重新调节了亮度

6.9 几何变换

6.9.1 改变图像信息滤波器

这个滤波器是 ITK 研发平台中易引起惊慌、最危险的滤波器之一。除非用户完全确信自己所做的操作，否则就不要使用这个滤波器。如果用户使用这个滤波器，应该编写自己的代码，然后再次经过深思熟虑、反复询问自己是否一定需要使用这个滤波器。如果你的答案是确定，然后你应该和最信任的朋友讨论这个课题并得到他/她在编写方面的建议。

6.9.2 翻转图像滤波器

本小节的源代码在文件 Examples/Filtering/FlipImageFilter.cxx 中。

使用 itk::FlipImageFilter 来翻转任何坐标轴中的图像。这个滤波器必须和 EXTREME 警告一起使用。例如，将一个 CT 扫描的 cranio-caudal 轴翻转为 left-right 轴以便看清图像。

使用这个滤波器必须首先包含相应的头文件：

```
#include "itkFlipImageFilter.h"
```

然后必须定义输入、输出图像的像素类型，并使用它们来实例化图像类型：

```
typedef unsigned char PixelType;
typedef itk::Image< PixelType, 2 > ImageType;
```

现在可以使用图像类型来实例化滤波器类型并创建滤波器对象。

```
typedef itk::FlipImageFilter< ImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

待翻转的轴是以一个数列的形式来指定的。这里我们从命令行变量来得到它们：

```
typedef FilterType::FlipAxesArrayType FlipAxesArrayType;
FlipAxesArrayType flipArray;
flipArray[0] = atoi( argv[3] );
flipArray[1] = atoi( argv[4] );
filter->SetFlipAxes( flipArray );
```

这个滤波器的输入可以从其他任何滤波器得到，例如一个 reader。输出可以沿流水线传递给其他滤波器，例如一个 writer。下游的任何一个滤波器调用 update 将触发均值滤波器的运行。

```
filter->SetInput( reader->GetOutput( ) );
writer->SetInput( filter->GetOutput( ) );
writer->Update( );
```

如图 6-38 所示阐述了这个滤波器对一个脑部 MRI 切片图像作用的效果，使用一个[0，1]翻转排列，表示 X 轴保持不变时对 Y 轴进行翻转。

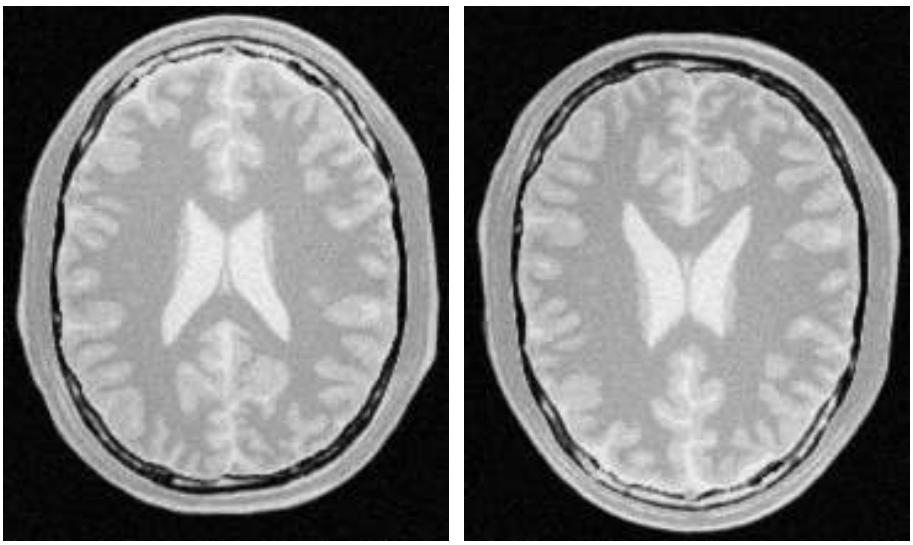


图 6-38　FlipImageFilter 对一个脑部 MRI 切片图像作用的效果

6.9.3　重采样图像滤波器

1．绪论

本小节的源代码在文件 Examples/Filtering/ResampleImageFilter.cxx 中。

图像重采样是图像分析中的一个非常重要的任务，在图像配准框架中尤为重要。itk::ResampleImageFilter 通过使用 itk::Transforms 来实现图像重采样。这个滤波器的输入应该是一个图像、一个变换和一个校对机。图像的空间坐标通过变换进行映射以便生成一个新的图像。结果图像中的范围和间距是由用户来选择的。重采样是在空间坐标中执行的，而不是像素/网格坐标。确信图像间距确实是在待处理的图像上设置的这一点非常重要。由于从一个空间到另一个空间的映射需要图像在非网格位置进行活动，所以需要使用校对机。

使用这个滤波器必须首先包含相应的头文件：

```
#include "itkResampleImageFilter.h"
```

同样也必须包含相应的变换和校对机的头文件：

```
#include "itkAffineTransform.h"
#include "itkNearestNeighborInterpolateImageFunction.h"
```

必须定义输入、输出图像的像素类型和维并使用它们实例化图像类型：

```
const unsigned int Dimension = 2;
typedef unsigned char InputPixelType;
typedef unsigned char OutputPixelType;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

现在可以使用图像和变换类型来实例化滤波器类型并创建滤波器对象：

```
typedef itk::ResampleImageFilter<InputImageType,OutputImageType> FilterType;
FilterType::Pointer filter = FilterType::New( );
```

通常使用图像维和用来表示空间坐标的类型来定义变换类型：

```
typedef itk::AffineTransform< double, Dimension > TransformType;
```

实例化变换对象的一个实例并传递给重采样滤波器。默认地，设置变换参数来表示恒等变换：

```
TransformType::Pointer transform = TransformType::New( );
filter->SetTransform( transform );
```

使用图像类型和用来表示空间坐标的类型来定义校对机类型：

```
typedef itk::NearestNeighborInterpolateImageFunction<
InputImageType, double > InterpolatorType;
```

实例化校对机的一个实例并传递给重采样滤波器：

```
InterpolatorType::Pointer interpolator = InterpolatorType::New( );
filter->SetInterpolator( interpolator );
```

由于输出图像的一些像素在输入图像范围之外将停止映射，所以有必要决定分派给它们的值。可以通过调用 SetDefaultPixelValue( )方式来实现：

```
filter->SetDefaultPixelValue( 0 );
```

结合沿每个维和源的间距来指定输出图像的重采样网格：

```
double spacing[ Dimension ];
spacing[0] = 1.0; // pixel spacing in millimeters along X
spacing[1] = 1.0; // pixel spacing in millimeters along Y
filter->SetOutputSpacing( spacing );
double origin[ Dimension ];
origin[0] = 0.0; // X space coordinate of origin
origin[1] = 0.0; // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

通过一个 SizeType 来定义输出图像上重采样的范围并使用 SetSize( )方式来进行设置：

```
InputImageType::SizeType size;
size[0] = 300; // number of pixels along X
size[1] = 300; // number of pixels along Y
filter->SetSize( size );
```

这个滤波器的输入可以从其他任何滤波器得到，例如一个 reader。输出可以沿流水线传递给其他滤波器，例如一个 writer。下游的任何一个滤波器调用 update 将触发重采样滤波器的运行。

```
filter->SetInput( reader->GetOutput( ) );
writer->SetInput( filter->GetOutput( ) );
writer->Update( );
```

如图 6-39 所示阐述了这个滤波器在一个脑部 MRI 切片图像上作用的效果，使用一个包含恒等变换的仿射变换。注意：这个滤波器的任何分析行为都必须以毫米度量在空间坐标系上实现，而不是关于以像素类型的重采样。图像展示了这个范围左下四分之一大小的结果图像。在以图像为单位进行分析似乎显得是临时的，但在空间坐标中是很清晰的。由于图像进

行了重新调节以适应本书的内容，所以图 6-39 所示容易引起人们误解。为此，如图 6-40 所示澄清了这个情况。它展示了同一坐标系范围中的两个相同图像。这里使用一个恒等变换来映射图像数据，所以变得很清晰，而且我们很简单地在图像周围添加了空白空间进行重采样。我们使用一个 181×217 像素的输入图像并需要一个 300×300 像素的图像。在这种情况下，输入、输出图像同时具有 1mm×1mm 的间距和(0.0,0.0)的原点。

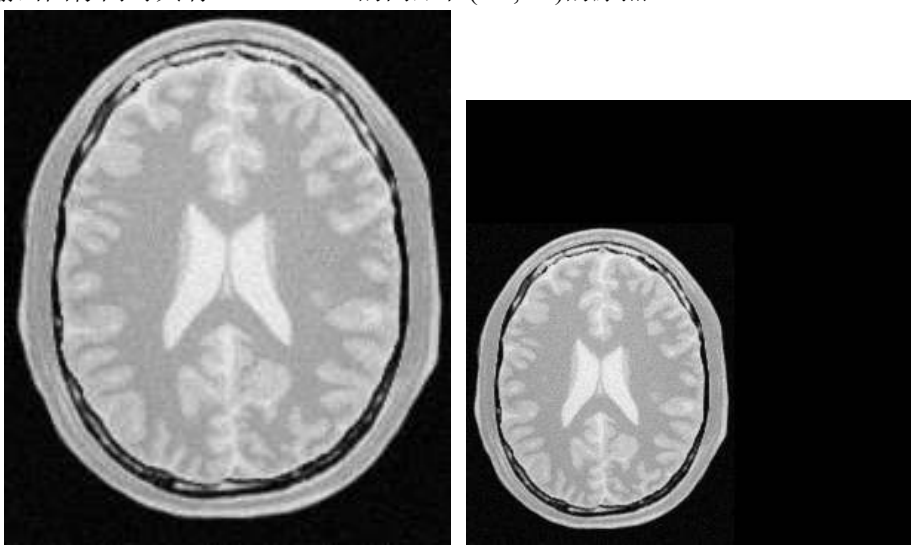


图 6-39　重采样滤波器的效果

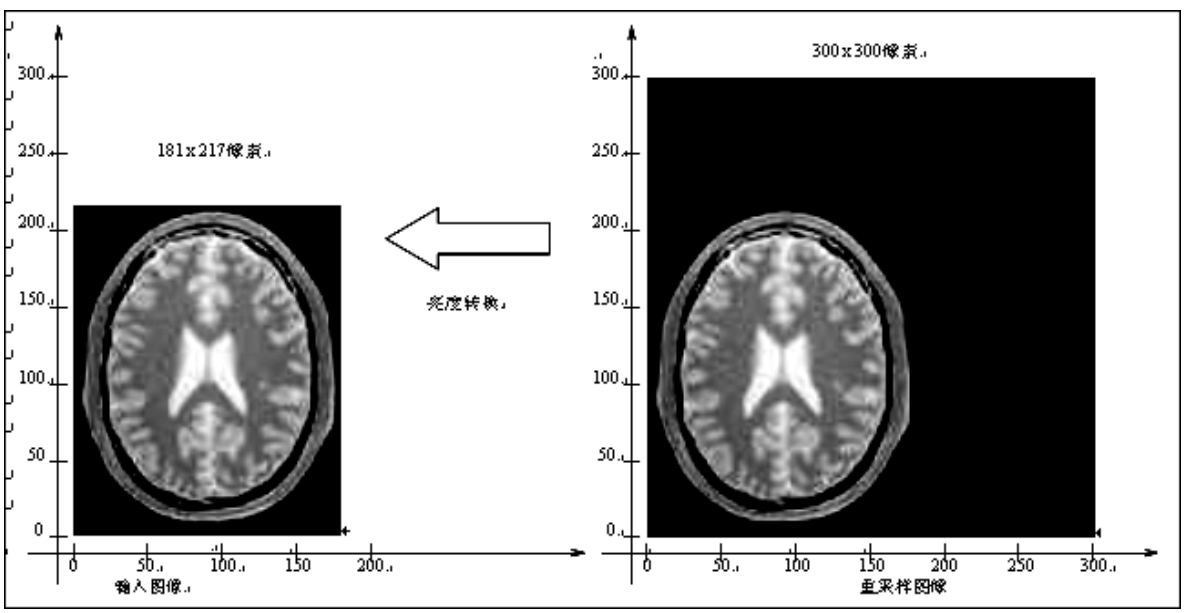


图 6-40　在同一坐标系统中做的重采样图像分析

现在让我们设置变换值。注意：供给的变换表示从输出空间到输入空间的点映射。下面的代码设置了一个变换：

TransformType::OutputVectorType translation;

translation[0] = -30; // X translation in millimeters

translation[1] = -50; // Y translation in millimeters

transform->Translate(translation);

从图 6-41 中可以看出输出图像是从变换结果得来的。如图 6-42 所示再一次更好地阐述了在同一个坐标系中的结果。

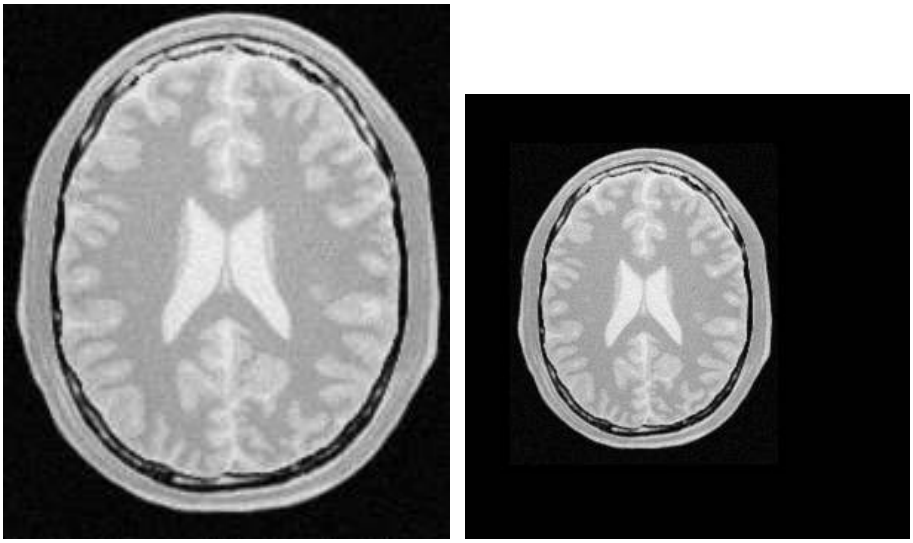


图 6-41　ResampleImageFilter 使用(–30，–50)变换的效果

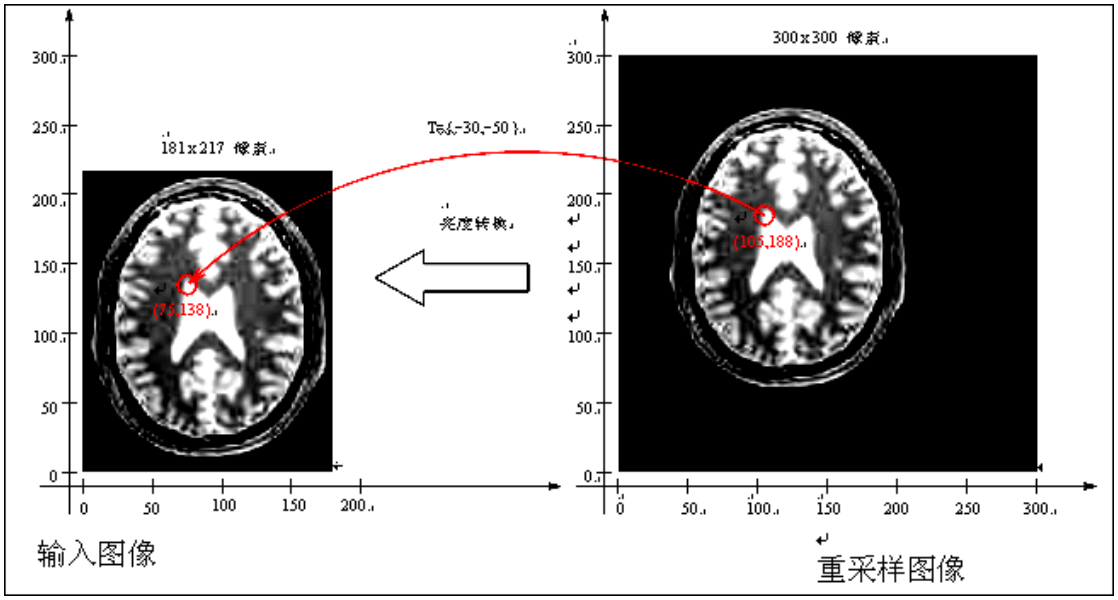


图 6-42　ResampleImageFilter 使用(–30,–50)变换的分析

当进行图像重采样时，必须记住：用来进行点映射的变换是从输出图像空间到输入图像空间的。在这种情况下，如图 6-42 所示展示了应用于输出图像的每个点的变换和用于读取输入图像的亮度的结果位置。在这种方式中，输出图像中点 P 的灰度级是从输入图像中的点

T(P)得到的。其中 T 是变换。在图 6-42 所示指定的案例中，由于使用了一个(−30，−50)的变换，所以输出图像中点(105，188)的值是从输入图像中的点(75，138)得到的。

有时有意地将默认输出值设置为一个独特的灰度值以便强调图像边界的映射是很有用的。例如：下面的代码将默认外部值设置为 100。图 6-43 所示的右图显示了这个结果。

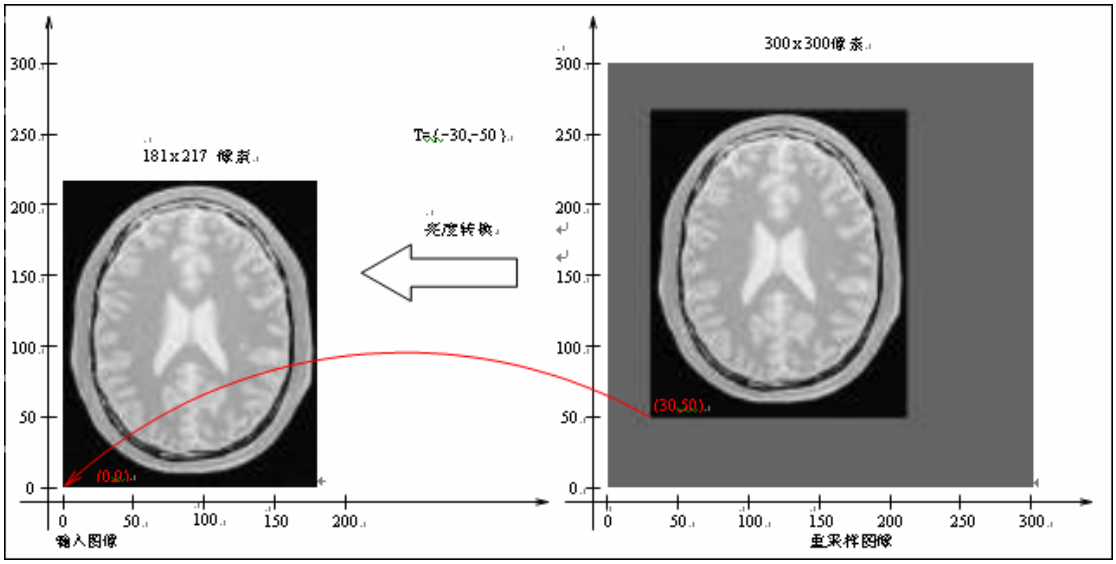


图 6-43　ResampleImageFilter 使用 SetDefaultPixelValue( )强调图像的边界

filter->SetDefaultPixelValue(100);

使用这个变换我们可以更加清晰地了解前面在图像采样中的变化。如图 6-43 所示阐述了输出图像中的点(30,50)如何从输入图像的点(0,0)来得到它的灰度值。

2．间距和原点的重要性

本小节的源代码在文件 Examples/Filtering/ResampleImageFilter2.cxx 中。

在图像重采样的计算中访问了输出区域中的所有像素。这个访问是使用整数网格空间图像中的遍历 ImageIterators 来实现的。对于每个像素，我们需要使用图像间距和原点来转变空间坐标的网格位置。

例如，一个原点为 O = (19.0,29.0)和像素间距为 S = (1.3,1.5)的图像中标记的像素 I = (20,50)的相应空间位置为：

$$P[i] = I[i] \times S[i] + O[i] \tag{6-20}$$

其中，P = (20×1.3+19.0,50×1.5+29.0)而最后 P = (45.0,104.0)。

使用变换 T 映射 P 的空间坐标，变换是支持 itk::ResampleImageFilter 的，以便映射 P 到输入图像空间点 Q=T(P)。

如图 6-44 所示阐述了整个处理过程。为了更精确地介绍 ResampleImageFilter 的处理，应该注意输入、输出图像中原点和间距的设置。

为了方便介绍变换，我们设置默认像素值为从图像背景而来的一个独特的值：

filter->SetDefaultPixelValue(50);

让我们为输出图像设置一个统一的间距：

```
double spacing[ Dimension ];
spacing[0] = 1.0; // pixel spacing in millimeters along X
spacing[1] = 1.0; // pixel spacing in millimeters along Y
filter->SetOutputSpacing( spacing );
```

另外我们将指定一个非零原点。注意：这里提供的值将是那些对标记(0,0)的像素的空间坐标。

```
double origin[ Dimension ];
origin[0] = 30.0; // X space coordinate of origin
origin[1] = 40.0; // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

我们设置变换为恒等的以便于更好地理解原点选择的效果。

```
transform->SetIdentity( );
filter->SetTransform( transform );
```

如图 6-44 所示中分析了从这些滤波器设置得到的输出结果。

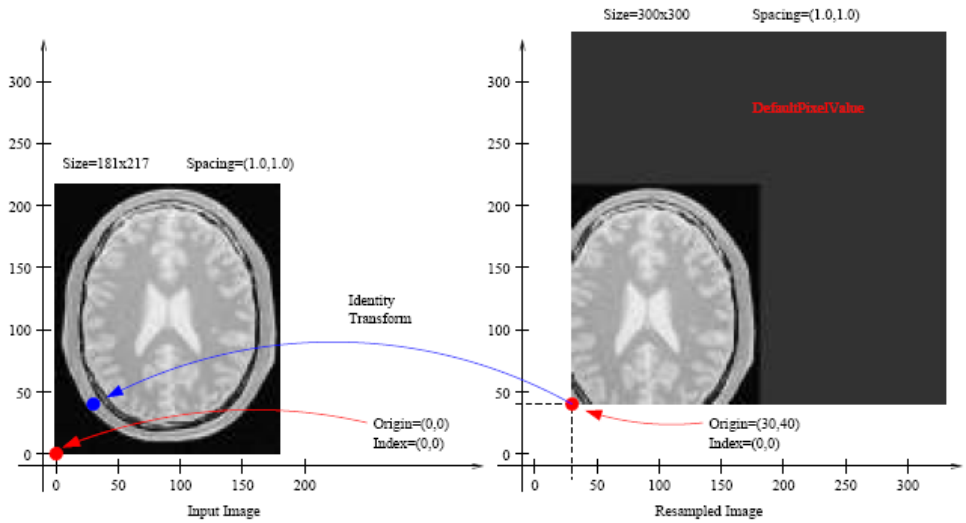


图 6-44　ResampleImageFilter 选择输出图像的原点

在图表中标记为 I = (0,0)的输出图像点的空间坐标为 P = (30,40)。恒等变换将这个点映射为输入图像空间中的 Q = (30,40)。由于在这个案例中使用的输入图像的间距为(1.0,1.0)，原点为(0.0,0.0)，所以物理点 Q = (30,40)映射为标记为 I = (30,40)的像素。

下面传输选择一个不同原点和图像尺度的代码。在图 6-45 中表达了输出结果。

```
size[0] = 150; // number of pixels along X
size[1] = 200; // number of pixels along Y
filter->SetSize( size );
origin[0] = 60.0; // X space coordinate of origin
origin[1] = 30.0; // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

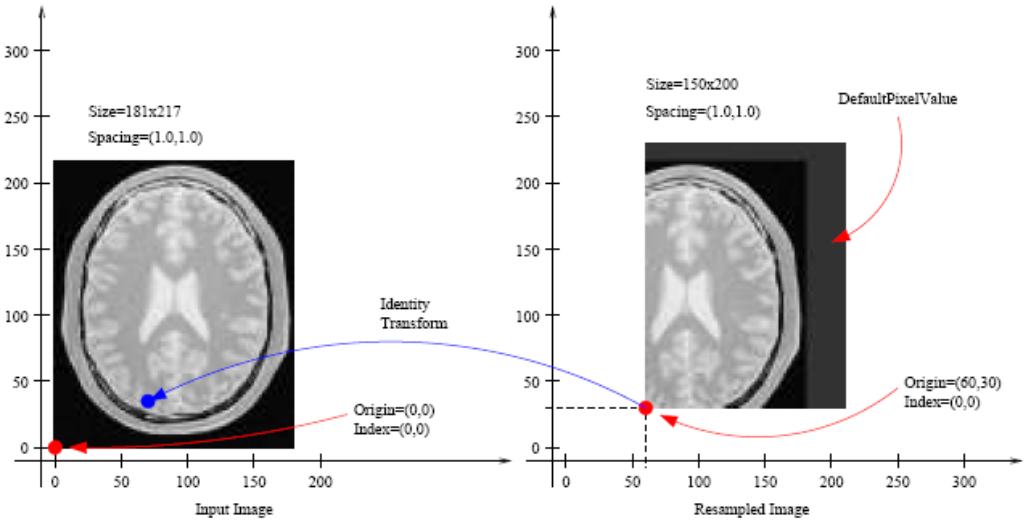


图 6-45　ResampleImageFilter 选择输出图像的原点

输出图像中标记为 I = (0,0)的点现在的空间坐标为 P = (60,30)。恒等变换将这个点映射为输入图像空间的点 Q = (60,30)。由于在这个案例中使用的输入图像的间距为(1.0,1.0)，原点为(0.0,0.0)，所以物理点 Q = (60,30)映射为标记为 I = (60,30)的像素。

现在我们分析输入图像中非零原点的影响。保留和前面例子中对输出图像同样的设置，我们仅仅更改输入图像文件头的原点值。输入图像的新原点指定为 O= (50,70)。同样使用一个恒等变换作为 ResampleImageFilter 滤波器的输入。如图 6-46 所示显示了使用这些参数运行滤波器得到的结果。

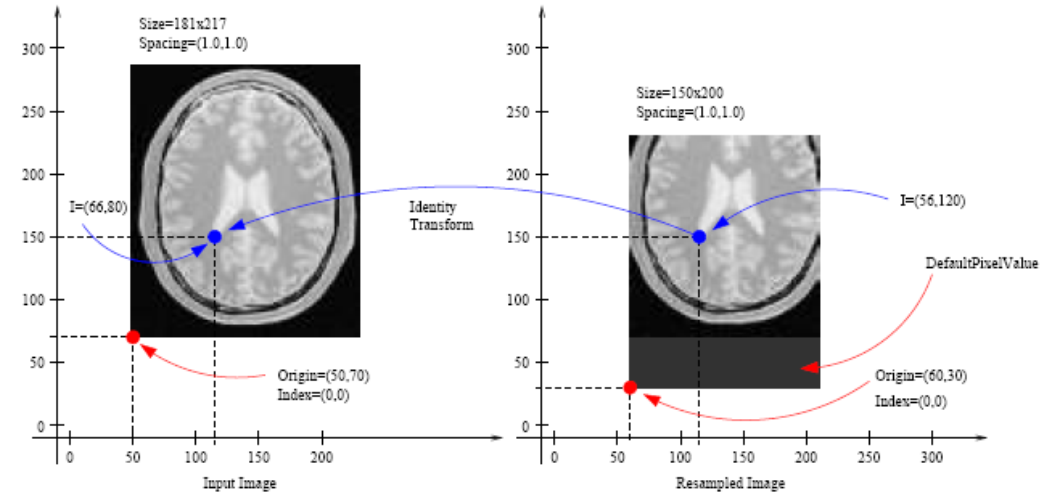


图 6-46　使用 ResampleImageFilter 选择输入图像原点的效果

输出图像中带有标记为 I = (56,120)的像素在物理空间的坐标为 P = (116,150)。恒等变换将 P 映射为输入图像空间中的点 Q = (116,150)。Q 点的坐标和输入图像上标记为 I = (66,80)的像素相关联。

现在考虑图像重采样过程中输出间距的影响。为了简化分析，我们将输入和输出图像中

的原点都设置为零。

```
origin[0] = 0.0; // X space coordinate of origin
origin[1] = 0.0; // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

然后我们为输出图像指定一个非单位间距：

```
spacing[0] = 2.0; // pixel spacing in millimeters along X
spacing[1] = 3.0; // pixel spacing in millimeters along Y
filter->SetOutputSpacing( spacing );
```

另外，由于现在新像素覆盖了一个 2.0mm×3.0mm 的大区域，所以我们减少输出图像的范围：

```
size[0] = 80; // number of pixels along X
size[1] = 50; // number of pixels along Y
filter->SetSize( size );
```

使用这些参数后，输出图像的物理范围为 160mm×150mm。

在试图分析重采样滤波器的效果之前，确定用来显示输入、输出图像的视窗接受这个间距并使用它适当地调整屏幕上图像的范围。注意：像 PNG 这样格式的图像并没有描述原点和间距的能力。ITK 平台为它们假设了默认值。如图 6-47 所示（中图）阐述了使用一个并未接受像素间距的视窗的效果。在右图中就表达了正确的显示。

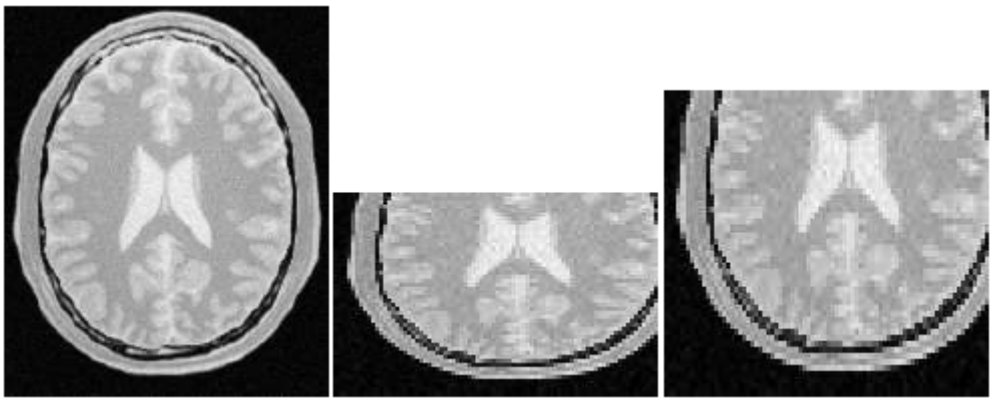


图 6-47　输入图像（左图）、在理想视窗（中图）和正确视窗（右图）中表达的不同间距的重采样效果

如图 6-48 所示阐述了输入图像和在同一个坐标系中的滤波器输出。在这个图片中，输出图像的像素 I = (33,27)在物理空间的坐标为 P= (66.0,81.0)。恒等变换将这个点映射为输入图像物理空间中的点 Q= (66.0,81.0)。由于这个图像使用的是零原点和单位间距，点 Q 将关联到输入图像中标记为 I = (66,81)的像素。

输入图像间距同样也是重采样一个图像过程中的一个重要因素。下面的例子阐述了非单位像素间距在输入图像上的影响。对一个类似于图 6-44 到图 6-48 所示使用的那些输入图像进行一个 2mm×3mm 像素间距的重采样。如图 6-49 所示使用一个理想视窗(左图)和一个正确图像视窗(右图)显示了输入图像。

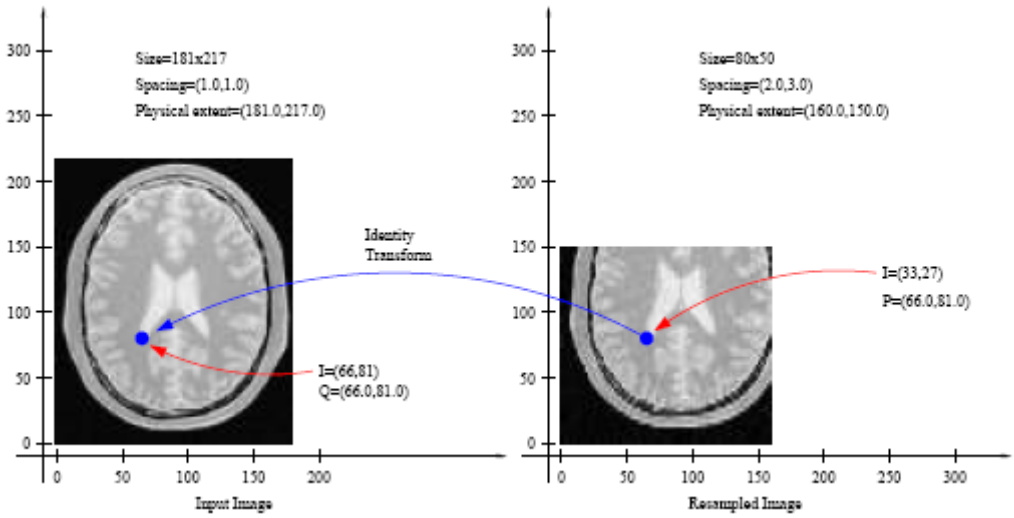


图 6-48　在输出图像中选择间距的影响效。

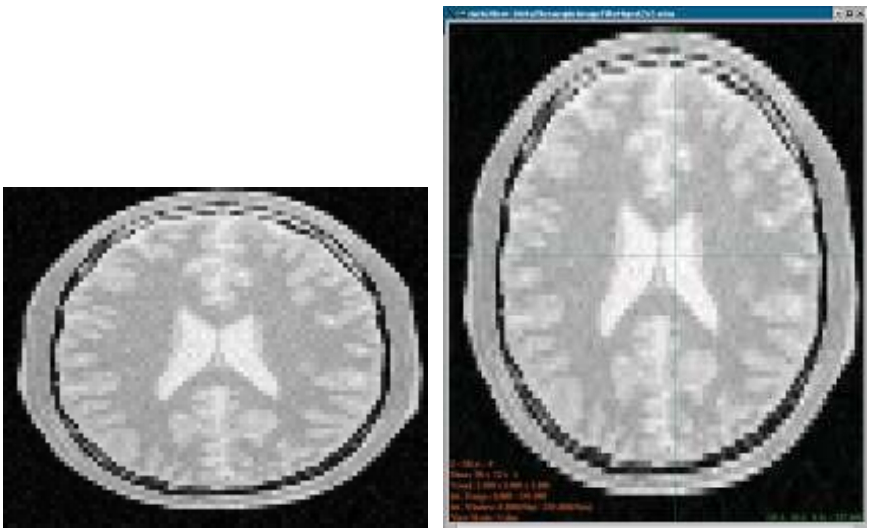


图 6-49　在理想视窗（左图）和正确图像视窗（右图）显示间距为 2mm×3mm 的输入图像

下面的代码用来将这个非单位间距变换成在一个非零原点载入的另一个非单位间距图像。如图 6-50 所示表达了在一个普通引用系统中输入和输出之间的对比。

这里我们从选择输出图像的原点开始：

```
origin[0] = 25.0; // X space coordinate of origin
origin[1] = 35.0; // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

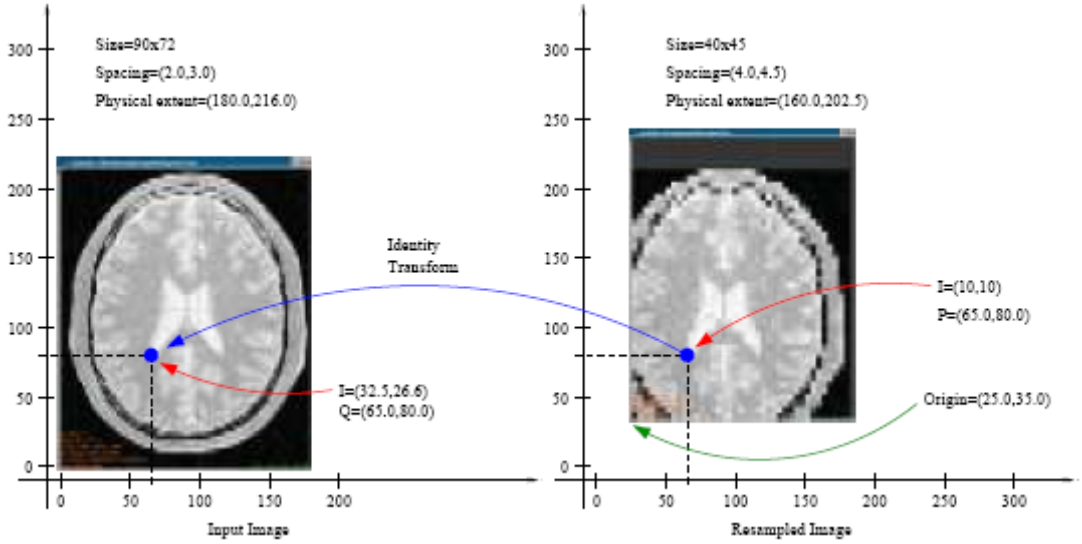


图 6-50　输入、输出图像中非单位间距的效果

然后我们选择沿每个维的像素数量：

```
size[0] = 40; // number of pixels along X
size[1] = 45; // number of pixels along Y
filter->SetSize( size );
```

最后我们设置输出像素间距：

```
spacing[0] = 4.0; // pixel spacing in millimeters along X
spacing[1] = 4.5; // pixel spacing in millimeters along Y
filter->SetOutputSpacing( spacing );
```

图 6-50 显示了在这些条件下对滤波器输出的分析。首先，注意与输出图像相关的设置为 O= (25.0,35.0)毫米，间距为(4.0,4.5)和(40,45)的像素大小。使用这些参数，输出图像中标记为 I = (10,10)的像素将与空间点坐标 P= (10×4.0+25.0,10×4.5+35.0)= (65.0,80.0)相关联。通过一个变换(在这种特定情况下为恒等变换)将这个点映射为输入图像空间中的点 Q= (65.0,80.0)。然后将点 Q 与标记为 I=((65.0−0.0)/2.0,(80.0−0.0)/3.0)= (32.5,26.6)相关联。注意：标记并不是落在网格位置，所以指向输出像素的值是通过在输入图像中围绕非整数标记 I= (32.5,26.6)插入值来计算的。

同样也注意由于选择了一个低的值（仅为 40×45 像素），所以图像的离散化比图 6-50 所示右边表示的输出可见度更高。

3．一个完整的例子

本小节的源代码在文件 Examples/Filtering/ResampleImageFilter3.cxx 中。

前面的例子描述了基于 itk::ResampleImageFilter 的基本原则。现在我们使用它来进行一些应用。如图 6-51 所示。

如图 6-52 所示阐述了重采样处理的一个通用案例。输出图像的原点和间距选择不同于输入图像中的那些值。圆圈表示像素的中心。它们标记在一个矩形区域中来表示这个像素覆

盖的区域。间距指定了沿每个维上像素中心之间的距离。

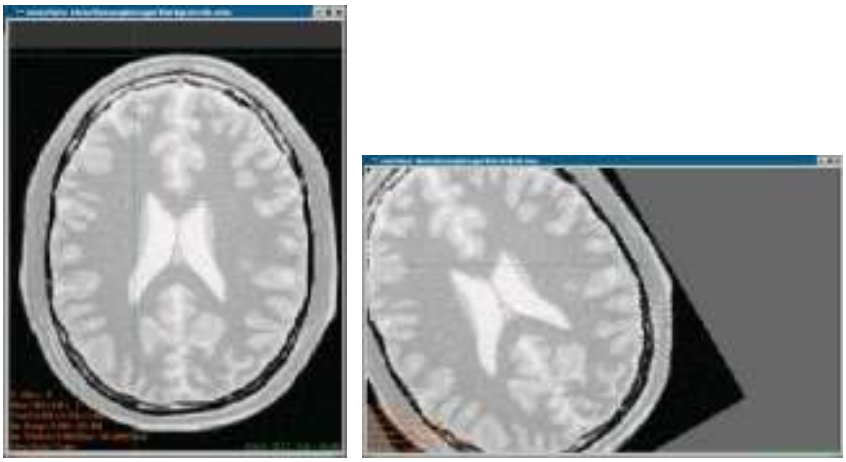


图 6-51　重采样滤波器旋转的效果。左图为输入图像，右图为输出图像

这个变换使用了一个 30 度的旋转。注意：这里这个变换使用 itk::ResampleImageFilter 做的是一个顺时针旋转是很重要的。这个变换将输出坐标系做了一个沿顺时针方向 30 度的旋转。如图 6-51 所示，如果从图像坐标系的垂直排列方向来看，图像内容有一个沿逆时针方向 30 度的旋转。如图 6-52 所示，当两个图像在同一个坐标系中时，结果就是输出图像的框架出现了一个沿顺时针 30 度的旋转。在继续阅读下面内容之前，你需要慎重考虑这个事实。

下面的代码实现了图 6-52 中阐述的条件下的执行，仅仅将输出间距减小了 40 倍而将两个维上的像素扩大了 40 倍。不使用这些改变，图像就无法辨认出很多细节。注意由于这个滤波器不能以任何方式来改变输入图像的实际内容，所以需要通过使用其他的均值来更新输入图像的间距和原点值。

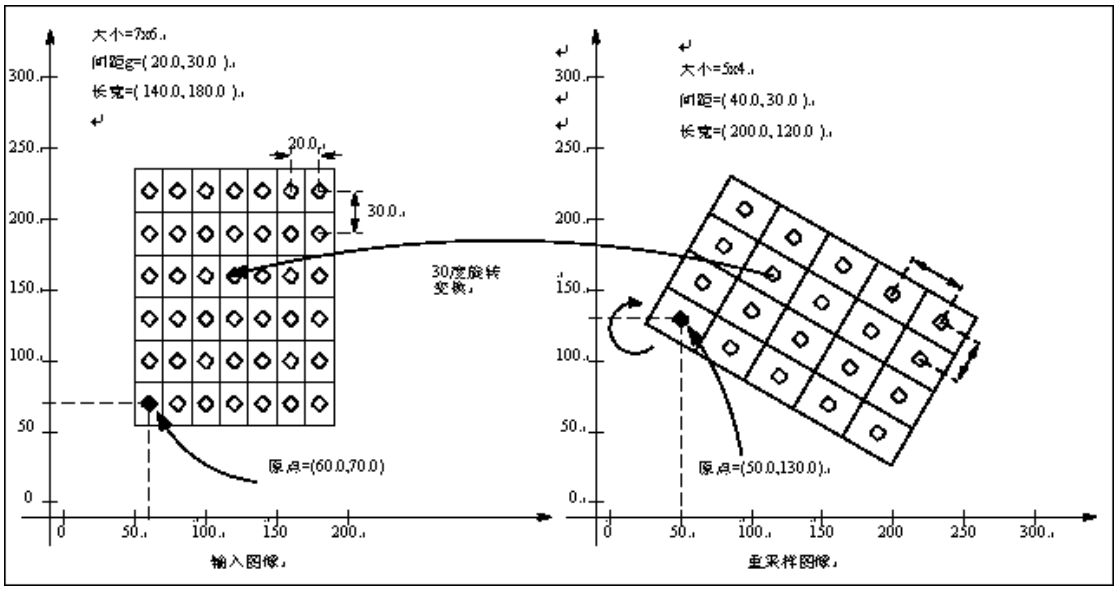


图 6-52　放在同一坐标系统中的输入、输出图像

为方便阐明变换，我们设置默认像素值为不同于图像背景的值：

```
filter->SetDefaultPixelValue( 100 );
```

这里选择一个比图 6-52 中阐述的值小 40 倍的值：

```
double spacing[ Dimension ];
spacing[0] = 40.0 / 40.0; // pixel spacing in millimeters along X
spacing[1] = 30.0 / 40.0; // pixel spacing in millimeters along Y
filter->SetOutputSpacing( spacing );
```

现在我们来设置输出图像的原点。注意：这里提供的值将是标记为(0,0)的输出图像像素的那些空间坐标。

```
double origin[ Dimension ];
origin[0] = 50.0; // X space coordinate of origin
origin[1] = 130.0; // Y space coordinate of origin
filter->SetOutputOrigin( origin );
```

将输出图像大小定义为图 6-52 阐述的那个值的 40 倍：

```
InputImageType::SizeType size;
size[0] = 5 * 40; // number of pixels along X
size[1] = 4 * 40; // number of pixels along Y
filter->SetSize( size );
```

旋转是围绕物理坐标的原点而不是图像原点也不是图像中心执行的。如图 6.52 所示来配置输出图像。过程有三步：首先，必须将图像原点移动到坐标系的原点，这可以通过使用一个与图像原点相反的值的变换来完成。

```
TransformType::OutputVectorType translation1;
translation1[0] = -origin[0];
translation1[1] = -origin[1];
transform->Translate( translation1 );
```

第二步，实现一个 30 度的旋转。在 itk::AffineTransform 中，角度是以弧度来指定的。如果变换的当前更新需要预构成或后构成当前变换内容，那么就需要使用第二个布尔变量来指定。在这个案例中，变量设置为 false 表示旋转需要应用在当前变换内容之后。

```
const double degreesToRadians = atan(1.0) / 45.0;
transform->Rotate2D( -30.0 * degreesToRadians, false );
```

第三步，将图像原点转变为它原来的位置。这是通过应用一个和原点值同值的变换来完成的。

```
TransformType::OutputVectorType translation2;
translation2[0] = origin[0];
translation2[1] = origin[1];
transform->Translate( translation2, false );
filter->SetTransform( transform );
```

如图 6-51 所示显示了这个例子的输入、输出图像，通过一个考虑间距的正确的视窗来显示。注意：在前面讨论了图 6-51 和图 6-52 中表示的逆时针和顺时针的影响。

作为最后一个案例，我们来追踪一个个别像素。记住这个变换是通过遍历输出图像的像素来初始化的。这是确定图像并未产生空洞或多余值的唯一方法。当你考虑变换时，分析相对于输入图像的输出图像是很有用的。

我们以输出图像中标记为 I= (1,2)的像素为例。这个点在输出图像引用系统中的物理坐标为 P= (1×40.0+50.0,2×30.0+130.0)= (90.0,190.0)毫米。

现在通过 itk::AffineTransform 将这个点 P 映射到输入图像空间。这个操作需要减去原点，使用一个 30 度的旋转并加回原点。接下来我们完成这些步骤：减去原点，P 将变成 P1=(40.0，60.0)；旋转将 P1 映射为 P2= (40.0×cos(30.0)+60.0×sin(30.0),40.0×sin(30.0)-60.0×cos(30.0))= (64.64,31.96)；通过考虑图像原点变换回去，这将使 P2 移动为 P3=(114.64,161.96)。

现在点 P3 在输入图像的坐标系中。和这个物理位置相关的输入图像的像素通过使用输入图像的原点和间距来计算。I=((114.64－60.0)/20.0, (161－70.0)/30.0)，结果为 I=(2.7,3.0)。注意这是非网格位置，因为这些值是非整数的。这就意味着指定给输出图像像素 I= (1,2)的灰度值必须通过输入图像的插补来计算。

在这个特定的代码中，校对机使用一个 itk::NearestNeighborInterpolateImageFunction，将指向最接近像素的值。从图 6-52 所示可以看出这是在标记为 I= (3,3)的像素结束。

4．旋转图像

本小节的源代码在文件 Examples/Filtering/ResampleImageFilter4.cxx 中。

下面的例子阐述了如何围绕一个图像的中心来旋转图像。在这个特定的案例中使用一个 itk::AffineTransform 来将输入空间映射到输出空间。

下面包含仿射变换的头文件：

```
#include "itkAffineTransform.h"
```

使用坐标表示类型和空间维来实例化变换类型。然后使用 New( )方式来构造一个变换对象并传递给一个 itk::SmartPointer：

```
typedef itk::AffineTransform< double, Dimension > TransformType;
TransformType::Pointer transform = TransformType::New( );
```

输出图像参数从输入图像得到：

```
reader->Update( );
const InputImageType::SpacingType&
spacing = reader->GetOutput( )->GetSpacing( );
const InputImageType::PointType&
origin = reader->GetOutput( )->GetOrigin( );
InputImageType::SizeType size =
reader->GetOutput( )->GetLargestPossibleRegion( ).GetSize( );
filter->SetOutputOrigin( origin );
filter->SetOutputSpacing( spacing );
filter->SetSize( size );
```

旋转是围绕物理坐标的原点而不是图像原点也不是图像中心执行的。如图 6-53 所示来配置输出图像。过程有三步：首先，必须将图像原点移动到坐标系的原点，这可以通过使用

一个与图像原点相反的值的变换来完成。

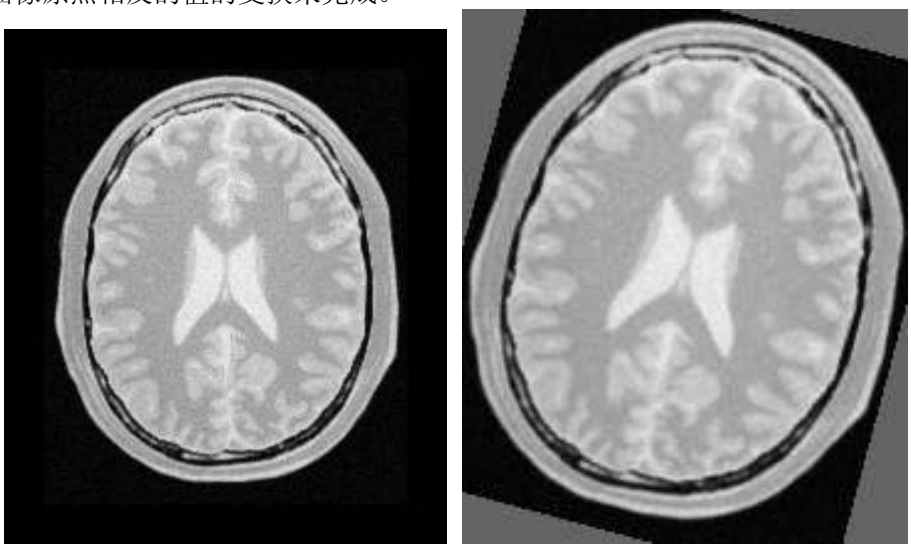


图 6-53　重采样滤波器旋转一个图像的效果

```
TransformType::OutputVectorType translation1;
const double imageCenterX = origin[0] + spacing[0] * size[0] / 2.0;
const double imageCenterY = origin[1] + spacing[1] * size[1] / 2.0;
translation1[0] = -imageCenterX;
translation1[1] = -imageCenterY;
transform->Translate( translation1 );
```

第二步，使用 Rotate2D( )方式来指定旋转。

```
const double degreesToRadians = atan(1.0) / 45.0;
const double angle = angleInDegrees * degreesToRadians;
transform->Rotate2D( −angle, false );
```

第三步，将图像原点转变为它原来的位置。这是通过应用一个和原点值同值的变换来完成的。

```
TransformType::OutputVectorType translation2;
translation2[0] = imageCenterX;
translation2[1] = imageCenterY;
transform->Translate( translation2, false );
filter->SetTransform( transform );
```

重采样滤波器的输出连接到一个 writer 并同一个 writer 更新来触发流水线的运行。

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & excep )
```

```
{
std::cerr << "Exception caught !" << std::endl;
std::cerr << excep << std::endl;
}
```

5．旋转和缩放图像

本小节的源代码在文件 Examples/Filtering/ResampleImageFilter5.cxx 中。

这个例子阐述了 itk::Similarity2DTransform 的用法。一个类似的变换包括旋转、平移和缩放。由于在一个 N 维案例中是很难进行参数化和旋转的，所以通常对一个二维案例来进行实现。

下面包含这个变换的头文件：

```
#include "itkSimilarity2DTransform.h"
```

使用坐标表示类型作为单一模板参数来实例化变换类型：

```
typedef itk::Similarity2DTransform< double > TransformType;
```

通过调用 New( )方式来构造一个变换对象并将结果指向一个 itk::SmartPointer：

```
TransformType::Pointer transform = TransformType::New( );
```

输出图像的参数是从输入图像得到的。

Similarity2DTransform 允许用户选择旋转的中心。这个中心同时用于旋转和缩放操作：

```
TransformType::InputPointType rotationCenter;
rotationCenter[0] = origin[0] + spacing[0] * size[0] / 2.0;
rotationCenter[1] = origin[1] + spacing[1] * size[1] / 2.0;
transform->SetCenter( rotationCenter );
```

使用 SetAngle( )方式来指定旋转：

```
const double degreesToRadians = atan(1.0) / 45.0;
const double angle = angleInDegrees * degreesToRadians;
transform->SetAngle( angle );
```

使用 SetScale( )方式来定义缩放变换：

```
transform->SetScale( scale );
```

在旋转和缩放之后使用的平移可以使用 SetTranslation( )方式来指定：

```
TransformType::OutputVectorType translation;
translation[0] = 13.0;
translation[1] = 17.0;
transform->SetTranslation( translation );
filter->SetTransform( transform );
```

注意：在这个变换中定义的旋转、缩放和平移顺序是不相关的。在仿射变换中这种情况是不允许的，仿射变换非常特别并允许不同的初始化组合。在 Similarity2DTransform 类中，旋转和缩放总是在平移之前使用。

如图 6-54 所示展示了这个滤波器在一个脑部 MRI 切片上旋转、平移和缩放的效果。这个图片使用的缩放比例为 1.2，并有一个 10 度的旋转角度。

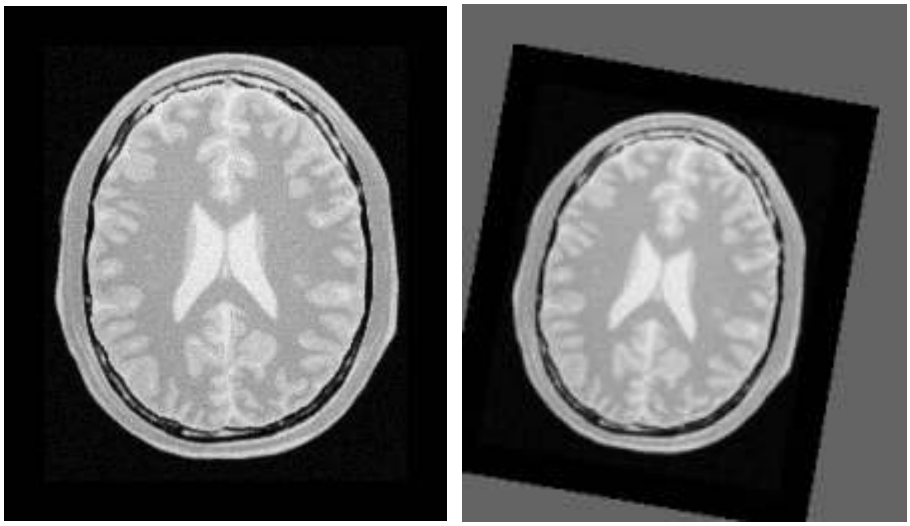


图 6-54　重采样滤波器旋转和缩放图像的效果

6．使用一个形变场进行重采样

本小节的源代码在文件 Examples/Filtering/WarpImageFilter1.cxx 中。

这个例子阐述了如何使用 WarpImageFilter 和用于重采样图像的形变场。这通常作为一个形变注册算法的最后一步。

```
#include "itkWarpImageFilter.h"
#include "itkLinearInterpolateImageFunction.h"
```

形变场作为一个向量像素类型图像来表达。向量的维和输入图像的维一样。形变场中的每个向量表示输入空间中的一个几何点和输出空间中的点的距离，可表示为：

$$Pin=Pout+distance \tag{6.21}$$

```
typedef float VectorComponentType;
typedef itk::Vector< VectorComponentType, Dimension > VectorPixelType;
typedef itk::Image< VectorPixelType, Dimension > DeformationFieldType;
typedef unsigned char PixelType;
typedef itk::Image< PixelType, Dimension > ImageType;
```

通过一个基于向量像素类型实例化的一个 reader 从一个文件来读取场区域：

```
typedef itk::ImageFileReader< DeformationFieldType > FieldReaderType;
FieldReaderType::Pointer fieldReader = FieldReaderType::New( );
fieldReader->SetFileName( argv[2] );
fieldReader->Update( );
DeformationFieldType::ConstPointer deformationField = fieldReader->GetOutput( );
```

itk::WarpImageFilter 基于输入图像类型、输出图像类型和形变场类型进行模板化：

```
typedef itk::WarpImageFilter< ImageType,
ImageType,
DeformationFieldType > FilterType;
```

FilterType::Pointer filter = FilterType::New( );

通常，位置映射并不一定和输入图像中的一个整数像素相关联。经过一个图像函数使用插补来计算非整数位置的值。这可以通过 SetInterpolator( )方式来完成：

typedef itk::LinearInterpolateImageFunction<

ImageType, double > InterpolatorType;

InterpolatorType::Pointer interpolator = InterpolatorType::New( );

filter->SetInterpolator(interpolator);

可以经过 SetOutputSpacing( )、SetOutputOrigin( )来设置输出图像的间距和原点。这可以从形变场得到：

filter->SetOutputSpacing(deformationField->GetSpacing( ));

filter->SetOutputOrigin(deformationField->GetOrigin( ));

filter->SetDeformationField(deformationField);

7．二次采样和同一空间中的图像

本小节的源代码在文件 Examples/Filtering/SubsampleVolume.cxx 中。

这个例子阐述了如何使用 ITK 类来执行一个二次采样。为了避免混淆，必须在重采样之前使用一个低通滤波器来进行处理。这里我们是用 itk::RecursiveGaussianImageFilter 作为低通滤波器。然后通过使用三个不同的因子（每个图像维上一个）来对图像进行重采样。

这里需要包含的最主要的头文件是和重采样图像、平移、校对和平滑滤波器相关的头文件：

#include "itkResampleImageFilter.h"

#include "itkIdentityTransform.h"

#include "itkLinearInterpolateImageFunction.h"

#include "itkRecursiveGaussianImageFilter.h"

我们明确地实例化像素类型和输入图像的维以及用来内在重采样计算的图像：

const unsigned int Dimension = 3;

typedef unsigned char InputPixelType;

typedef float InternalPixelType;

typedef unsigned char OutputPixelType;

typedef itk::Image< InputPixelType, Dimension > InputImageType;

typedef itk::Image< InternalPixelType, Dimension > InternalImageType;

typedef itk::Image< OutputPixelType, Dimension > OutputImageType;

在这个特定的案例中我们直接从命令行变量得到重采样的要素：

const double factorX = atof(argv[3]);

const double factorY = atof(argv[4]);

const double factorZ = atof(argv[5]);

实例化一个投射滤波器以便于将输入图像的像素类型转变为期望用于重采样计算的像素类型：

typedef itk::CastImageFilter< InputImageType,

InternalImageType > CastFilterType;

CastFilterType::Pointer caster = CastFilterType::New( );

caster->SetInput(inputImage);

选择 RecursiveGaussianImageFilter 作为平滑滤波器。我们创建三个滤波器以便沿每个维使用不同的 Sigma 值来执行平滑滤波：

typedef itk::RecursiveGaussianImageFilter<

InternalImageType,

InternalImageType > GaussianFilterType;

GaussianFilterType::Pointer smootherX = GaussianFilterType::New( );

GaussianFilterType::Pointer smootherY = GaussianFilterType::New( );

GaussianFilterType::Pointer smootherZ = GaussianFilterType::New( );

这些平滑滤波器层叠的连接在流水线中：

smootherX->SetInput(caster->GetOutput( ));

smootherY->SetInput(smootherX->GetOutput( ));

smootherZ->SetInput(smootherY->GetOutput( ));

平滑滤波器中使用的 Sigma 值是基于输入图像的像素间距和作为变量提供的要素来计算的：

const InputImageType::SpacingType& inputSpacing = inputImage->GetSpacing( );

const double sigmaX = inputSpacing[0] * factorX;

const double sigmaY = inputSpacing[1] * factorY;

const double sigmaZ = inputSpacing[2] * factorZ;

smootherX->SetSigma(sigmaX);

smootherY->SetSigma(sigmaY);

smootherZ->SetSigma(sigmaZ);

我们将这些滤波器逐一沿图像的一个特定方向，并使用标准交叉比例空间来设置它们，以便阻止伴随和高斯滤波相关的扩散过程中的亮度降低：

smootherX->SetDirection(0);

smootherY->SetDirection(1);

smootherZ->SetDirection(2);

smootherX->SetNormalizeAcrossScale(false);

smootherY->SetNormalizeAcrossScale(false);

smootherZ->SetNormalizeAcrossScale(false);

使用图像内在类型和输出图像类型来实例化重采样滤波器类型：

typedef itk::ResampleImageFilter<

InternalImageType, OutputImageType > ResampleFilterType;

ResampleFilterType::Pointer resampler = ResampleFilterType::New( );

由于重采样是在和输入图像相同的物理范围中执行的，所以重采样滤波器选择 IdentityTransform。

typedef itk::IdentityTransform< double, Dimension > TransformType;

```
TransformType::Pointer transform = TransformType::New( );
transform->SetIdentity( );
resampler->SetTransform( transform );
```

由于线性校对提供一个好的运行时间性能，所以选择使用线性校对。对于更加精确的应用，可以使用 itk::BSplineInterpolateImageFunction interpolator 或 itk::WindowedSinc InterpolateImageFunction interpolator 来代替这个校对机。

```
typedef itk::LinearInterpolateImageFunction<
InternalImageType, double > InterpolatorType;
InterpolatorType::Pointer interpolator = InterpolatorType::New( );
resampler->SetInterpolator( interpolator );
```

在重采样图像网格中使用的间距是使用输入图像间距和在命令行变量提供的因子来计算的：

```
OutputImageType::SpacingType spacing;
spacing[0] = inputSpacing[0] * factorX;
spacing[1] = inputSpacing[1] * factorY;
spacing[2] = inputSpacing[2] * factorZ;
resampler->SetOutputSpacing( spacing );
```

保持输入图像的原点并传递给输出图像：

```
resampler->SetOutputOrigin( inputImage->GetOrigin( ) );
```

重采样图像网格上沿每个方向上使用的像素数量是使用输入图像中的像素数量和采样因子来计算的：

```
InputImageType::SizeType inputSize =
inputImage->GetLargestPossibleRegion( ).GetSize( );
typedef InputImageType::SizeType::SizeValueType SizeValueType;
InputImageType::SizeType size;
size[0] = static_cast< SizeValueType >( inputSize[0] / factorX );
size[1] = static_cast< SizeValueType >( inputSize[1] / factorY );
size[2] = static_cast< SizeValueType >( inputSize[2] / factorZ );
resampler->SetSize( size );
```

最后从平滑滤波器的输出得到重采样的输入。

```
resampler->SetInput( smootherZ->GetOutput( ) );
```

现在我们可以通过调用 Update( )方式来触发重采样的运行，或我们可以将重采样滤波器的输出传递给流水线的另一个部分，例如一个图像 writer。

8．重采样一个各向异性图像使得它各向同性

本小节的源代码在文件 Examples/Filtering/ResampleVolumesToBeIsotropic.cxx 中。

不幸的是，对于要求大的插入间距的医学图像数据通常会导致各向异性形状的三维像素。在许多案例中，这些三维像素在水平面(x,y)和 Z 轴之间的比率是[1:5]甚至是[1:10]。这些数据对计算机辅助图像分析是毫无用处的。以这样的方式需求数据的趋势恰恰表现出在临

床设置和许多放射医学领域已经得到的第三维数据的缺乏。这些大的各向异性的数据的需求也带来了负面的信息：“我认为三维是毫无情报的。”而另一些人顽固地重复说：“你可以通过逐一观看单独的切片来得到你需求的信息。”然而，在这样的综述的谬论的误导下，使得在对任何平面直角进行再构造时对观察切片的简单行为显得更明显，极端的矩形像素形状变得更加显而易见。即使实现这些技术，也不能在这样的图像中很好地执行信号处理或算法。

图像分析在放射医学设置中还有很长一段路要走，为了给三维数据提供需要的各向异性大于[1:2]的信息，就不能轻视数字信号处理的最基本概念：香农采样定理。

在面对许多临床成像部门和他们坚持的这些应该可以足够好地进行图像处理图像时，一些图像分析家试图处理这些不完整的数据。这些图像分析家通常对高水平面进行二次采样并在采样基础上带着捏造数据类型的目的进行切片交互，他们应该首先收到这些数据：一个各向同性数据集。这个例子阐述了如何使用 ITK 平台允许的滤波器实现这些操作。

注意：这个例子并不是作为各向异性数据集问题的一个解答来表达的。相反的，这个案例有助于图像获得部门对失误进行简单辩解。这个萎缩的各向异性数据集问题的真正解决办法是在图像处理的基本理论上进行放射医学训练。如果你真正关心医学图像处理领域的技术，如果你确实关注自己提供的处理图像可以直接或间接地帮助病人恢复健康，那么你就有责任抵制各向异性数据集并为放射线学者耐心讲解例如为什么一个[1:5]各向异性比率使得一个数据集仅仅是一个切片组合而不是一个可信的三维数据。

在使用这一部分包含的技术之前，请与放射线学者一起来讨论在一个平面直角切片中的数据集。在视窗中缩放图像，直到看到平面直角像素的实体都不可能会有任何线性插值。让他/她知道对以[1:5]或[1:10]的比率采样的数字数据进行处理是多么荒谬。然后，让他们知道你需要做的第一件事就是丢掉所有的高分辨率并编辑整理切片之间的数据以便补偿它们的低分辨率。仅仅到此时你才可以使用这个代码。

现在我们来看代码，同时也给你带来愧疚，因为你即将使用的下面的代码表明了你必然将失去真正的三维数据处理中的一个或多个行为。

这个例子在面内分辨率上执行一个二次采样并沿交互切片分辨率之上执行采样。二次采样处理需要我们使用一个平滑滤波器预处理数据，以防止由于频域重叠而发生混淆现象。这里使用 RecursiveGaussian 滤波器，因为它提供了一个便利的运行时间性能。

为了重采样这个各向异性数据集需要做的第一件事是包含 itk::ResampleImageFilter 和高斯平滑滤波器的头文件：

```
#include "itkResampleImageFilter.h"
#include "itkRecursiveGaussianImageFilter.h"
```

重采样滤波器需要一个变换来映射点坐标，并需要一个校对来计算新的重采样图像的亮度值。在这个特定的案例中我们使用 itk::IdentityTransform，因为图像是通过阻止采样区域的物理范围来进行重采样的。线性校对机作为一个普通的权衡来使用，尽管我们必须在二次采样处理面内使用一种类型的校对机，而在交互切片之上采样时使用另一种类型，但是我们必须知道为什么在这里加入技术混合，我们要做的事情是用来掩饰从医学数据中得到的不合适的数据，并且我们仅仅试图使得它看起来似乎是正确的数据。

```
#include "itkIdentityTransform.h"
#include "itkLinearInterpolateImageFunction.h"
```

注意：作为图像预处理的一部分，在这个例子中我们也重新改变了亮度范围的尺度。这个操作作为 Intensity Windowin 来描述。在一个真正的临床应用中，这一步需要谨慎考虑包含当前临床应用中的解剖结构信息的亮度范围。实际上你可能需要移除亮度尺度改变这一步。

```
#include "itkIntensityWindowingImageFilter.h"
```

现在必须清楚我们对即将处理的输入图像的像素类型和维，同样还有我们在平滑和重采样时用于内部计算的像素类型：

```
const unsigned int Dimension = 3;
typedef unsigned short InputPixelType;
typedef float InternalPixelType;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< InternalPixelType, Dimension > InternalImageType;
```

我们对用于重采样面内分辨率数据进行预处理使用的平滑滤波器进行实例化：

```
typedef itk::RecursiveGaussianImageFilter<
InternalImageType,
InternalImageType > GaussianFilterType;
```

我们创建两个平滑滤波器实例，一个沿 X 方向平滑滤波而另一个沿 Y 方向。他们在流水线中是层叠连接的，从亮度 windowing 滤波器得到它们的输入。注意：你可以跳过亮度 windowing 范围并简单地直接从 reader 得到输入：

```
GaussianFilterType::Pointer smootherX = GaussianFilterType::New( );
GaussianFilterType::Pointer smootherY = GaussianFilterType::New( );
smootherX->SetInput( intensityWindowing->GetOutput( ) );
smootherY->SetInput( smootherX->GetOutput( ) );
```

现在我们必须提供重采样本身的设置。这是通过寻找一个用于提供在二次采样和超级采样之间的一个权衡的各向同性值来完成的。这里我们改变一个便于各向同性分辨率使用的面内和交互切片分辨率之间几何意义的猜想假设。这个假设仅仅是一个直觉。当然你可以考虑选择更先进的技术。如果你倾向于图像采样过程中先进技术的正确性，你大可不必使用这些代码，因为这些代码仅仅考虑各向异性数据的放射学技术的正确性。

我们从输入得到图像并要求像素间距值的序列：

```
InputImageType::ConstPointer inputImage = reader->GetOutput( );
const InputImageType::SpacingType& inputSpacing = inputImage->GetSpacing( );
```

并且应用我们关于使用的合适的各向异性分辨率就是面内和交互切片分辨率的几何意义的特设性假设。然后将在预处理阶段用于高斯滤波的 Sigma 值设置为这个间距。

```
const double isoSpacing = sqrt( inputSpacing[2] * inputSpacing[0] );
smootherX->SetSigma( isoSpacing );
smootherY->SetSigma( isoSpacing );
```

我们分别沿 X 和 Y 方向来构造平滑滤波器。并且作为从使用高斯滤波器继承的扩散过程的结果来定义用于消除亮度损失的设置。

```
smootherX->SetDirection( 0 );
```

```
smootherY->SetDirection( 1 );
smootherX->SetNormalizeAcrossScale( true );
smootherY->SetNormalizeAcrossScale( true );
```

现在我们来仔细考虑面内平滑滤波，我们对用于再构造一个各向同性图像的重采样滤波器进行实例化。我们通过声明用在这个滤波器输出的像素类型开始，然后实例化图像类型和重采样滤波器类型。最后我们构造这样一个滤波器的实例。

```
typedef unsigned char OutputPixelType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
typedef itk::ResampleImageFilter<
InternalImageType, OutputImageType > ResampleFilterType;
ResampleFilterType::Pointer resampler = ResampleFilterType::New( );
```

重采样滤波器需要我们提供一个变换，在这个特定的案例中可以简化为一个恒等变换：

```
typedef itk::IdentityTransform< double, Dimension > TransformType;
TransformType::Pointer transform = TransformType::New( );
transform->SetIdentity( );
resampler->SetTransform( transform );
```

这个滤波器同样也需要一个用语传递它的校对机。在这个案例中我们使用一个线性校对机：

```
typedef itk::LinearInterpolateImageFunction<
InternalImageType, double > InterpolatorType;
InterpolatorType::Pointer interpolator = InterpolatorType::New( );
resampler->SetInterpolator( interpolator );
```

重采样数据集的像素间距是在一个 SpacingType 载入的并传递给重采样滤波器：

```
OutputImageType::SpacingType spacing;
spacing[0] = isoSpacing;
spacing[1] = isoSpacing;
spacing[2] = isoSpacing;
resampler->SetOutputSpacing( spacing );
```

由于我们决定以和输入各向异性图像相同的物理范围对图像进行重采样，所以我们保留输出图像的原点：

```
resampler->SetOutputOrigin( inputImage->GetOrigin( ) );
```

重采样图像网格中沿每个维使用的像素数量是使用输入图像和输出图像的像素间距之间的比率来计算的。注意：沿 Z 方向的像素数量的计算，与确定那些不参与计算的原始各向异性数据集之外的像素的目的，是有一定区别的。

```
InputImageType::SizeType inputSize =
inputImage->GetLargestPossibleRegion( ).GetSize( );
typedef InputImageType::SizeType::SizeValueType SizeValueType;
const double dx = inputSize[0] * inputSpacing[0] / isoSpacing;
const double dy = inputSize[1] * inputSpacing[1] / isoSpacing;
const double dz = (inputSize[2] - 1 ) * inputSpacing[2] / isoSpacing;
```

最后，以一个 SizeType 储存这个值并传递给重采样滤波器。注意：这个计算是以双精度型执行的而 SizeType 是整型的，所以这个过程需要一个投射。

```
InputImageType::SizeType size;
size[0] = static_cast<SizeValueType>( dx );
size[1] = static_cast<SizeValueType>( dy );
size[2] = static_cast<SizeValueType>( dz );
resampler->SetSize( size );
```

最后，从平滑滤波器的输出得到重采样图像滤波器的输入，然后通过在重采样滤波器上调用 Update( )方式来触发流水线的运行：

```
resampler->SetInput( smootherY->GetOutput( ) );
resampler->Update( );
```

6.10 频域

6.10.1 快速傅立叶变换(FFT)计算

本小节的源代码在文件 Examples/Filtering/FFTImageFilter.cxx 中。

本小节中我们假设已经熟悉了频谱分析，特别是傅立叶变换的定义和快速傅立叶变换的数字实现。如果不熟悉这些概念，首先就需要参考和频谱分析有关的许多可行的介绍性书籍。

这个例子阐述了如何在频域中使用快速傅立叶变换(FFT)滤波器处理图像。由于 FFT 计算可以是 CPU 处理器，所以有多种实现 FFT 的硬件。IT 在很多代表变换到局部可行数据库的计算的案例中是很便捷的。那些数据库的典型例子是 fftw 和实现 FFT 的 VXL。由于这个原因，ITK 提供了一个抽象基类，这个类可以因式分解 FFT 的多样特定实现的接口。这个基类是 itk::FFTRealToComplexConjugateImageFilter，它的两个起始类是 itk::VnlFFTRealToComplexConjugateImageFilter 和 itk::FFTWRealToComplexConjugateImageFilter。

FFT 的典型应用将需要包含下面的头文件：

```
#include "itkImage.h"
#include "itkVnlFFTRealToComplexConjugateImageFilter.h"
#include "itkComplexToRealImageFilter.h"
#include "itkComplexToImaginaryImageFilter.h"
```

首先要确定的是我们计算傅立叶变换的图像的像素类型和维：

```
typedef float PixelType;
const unsigned int Dimension = 2;
typedef itk::Image< PixelType, Dimension > ImageType;
```

我们使用相同的图像类型实例化 FFT 滤波器。这个案例使用 itk::VnlFFTRealToComplexConjugateImageFilter。注意：和大多数 ITK 滤波器不同，FFT 滤波器是使用像素类型和图像维来进行实例化的。一旦实例化了滤波器类型，我们可以通过调用 New( )方式来创建对象并将结果指向一个 SmartPointer：

```
typedef itk::VnlFFTRealToComplexConjugateImageFilter<
PixelType, Dimension > FFTFilterType;
FFTFilterType::Pointer fftFilter = FFTFilterType::New( );
```

这个滤波器的输入可以从一个 reader 得到：

```
typedef itk::ImageFileReader< ImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New( );
reader->SetFileName( argv[1] );
fftFilter->SetInput( reader->GetOutput( ) );
```

可以通过调用 Update( )方式来触发滤波器的运行。由于可以抛出错误和异常，所以调用必须在一个 try/catch 模块中：

```
try
{
fftFilter->Update( );
}
catch( itk::ExceptionObject & excp )
{
std::cerr << "Error: " << std::endl;
std::cerr << excp << std::endl;
return EXIT_FAILURE;
}
```

FFT 滤波器的输出通常是一个合成的图像。我们可以将这个图像保存在一个文件中以便进行更进一步的分析。这可以通过使用从 FFT 滤波器得到的输出图像的特征简单实例化一个 itk::ImageFileWriter 来实现。我们构造 writer 的一个实例并将 FFT 滤波器的输出作为输入传递给 writer：

```
typedef FFTFilterType::OutputImageType ComplexImageType;
typedef itk::ImageFileWriter< ComplexImageType > ComplexWriterType;
ComplexWriterType::Pointer complexWriter = ComplexWriterType::New( );
complexWriter->SetFileName("complexImage.mhd");
complexWriter->SetInput( fftFilter->GetOutput( ) );
```

最后我们调用一个 try/catch 模块中的 Update( )方式：

```
try
{
complexWriter->Update( );
}
catch( itk::ExceptionObject & excp )
{
std::cerr << "Error: " << std::endl;
std::cerr << excp << std::endl;
return EXIT_FAILURE;
```

}

在把这个合成的图像保存到一个文件中后，我们也可以提取它的真实部分和虚构部分做进一步的分析。这可以使用 itk::ComplexToRealImageFilter 和 itk::ComplexToImaginaryImageFilter 来实现。

我们首先实例化 ImageFilter，这将帮助我们从组合图像中提取出真实的部分。ComplexToRealImageFilter 作为组合图像类型的第一个模板参数并作为第二个模板参数带来输出图像像素的类型。我们创建这个滤波器的一个实例并将 FFT 滤波器的输出作为它的输入和它进行连接。

```
typedef itk::ComplexToRealImageFilter<
ComplexImageType, ImageType > RealFilterType;
RealFilterType::Pointer realFilter = RealFilterType::New( );
realFilter->SetInput( fftFilter->GetOutput( ) );
```

由于在傅立叶领域中的亮度范围可能是非常集中的，所以为了显示它就要对图像进行重新调节。出于这个目的，这里我们实例化一个 itk::RescaleIntensityImageFilter，将把真正图像的亮度范围调节为一个合适的范围写到一个文件中。我们同样设置用于写图像的像素类型范围输出的最大值、最小值：

```
typedef itk::RescaleIntensityImageFilter<
ImageType,
WriteImageType > RescaleFilterType;
RescaleFilterType::Pointer intensityRescaler = RescaleFilterType::New( );
intensityRescaler->SetInput( realFilter->GetOutput( ) );
intensityRescaler->SetOutputMinimum( 0 );
intensityRescaler->SetOutputMaximum( 255 );
```

现在我们实例化 ImageFilter，这将帮助我们从组合图像中提取虚构部分。这里我们使用的滤波器是 itk::ComplexToImaginaryImageFilter。它作为组合图像类型的第一个模板参数并作为第二个模板参数带来输出图像像素的类型。创建这个滤波器的一个实例，并将 FFT 滤波器的输出作为它的输入和它进行连接：

```
typedef FFTFilterType::OutputImageType ComplexImageType;
typedef itk::ComplexToImaginaryImageFilter<
ComplexImageType, ImageType > ImaginaryFilterType;
ImaginaryFilterType::Pointer imaginaryFilter = ImaginaryFilterType::New( );
imaginaryFilter->SetInput( fftFilter->GetOutput( ) );
```

然后重新调节虚构图像并保存到一个文件中，和对真实的部分做的一样。

为了阐述一个 itk::ImageFileReader 在合成图像上的用法，这里我们实例化一个 reader 来载入我们刚刚保存的图像。注意：这个案例中没有任何特别。这个实例化和其他任何图像类型是相同的，再一次阐述了范型编程的能力。

```
typedef itk::ImageFileReader< ComplexImageType > ComplexReaderType;
ComplexReaderType::Pointer complexReader = ComplexReaderType::New( );
complexReader->SetFileName("complexImage.mhd");
```

complexReader->Update( );

6.10.2 频域平滑滤波

本小节的源代码在文件 Examples/Filtering/FFTImageFilterFourierDomainFiltering.cxx 中。

在傅立叶域中执行的一个常见的图像处理操作是频谱掩藏，它是为了消除从输入图像来的空间频率。这个操作的典型实现是载入输入图像，使用一个 FFT 滤波器计算它的傅立叶变换，在傅立叶域中使用一个 mask 来掩饰图像结果，最后得到掩藏结果并计算它的傅立叶反变换。

下面的代码阐述了这个典型处理。

我们通过包含 FFT 滤波器和 Mask 图像滤波器的头文件开始。注意：这里我们使用两种不同类型的 FFT 滤波器。第一个作为真像素类型的图像输入（组合成员中感觉为真的）并作为输出产生一个组合图像。第二个 FFT 滤波器作为一个组合图像输入表达并生成一个真图像作为输出：

```
#include "itkVnlFFTRealToComplexConjugateImageFilter.h"
#include "itkVnlFFTComplexConjugateToRealImageFilter.h"
#include "itkMaskImageFilter.h"
```

首先要确定的是我们计算傅立叶变换的图像的像素类型和维：

```
typedef float InputPixelType;
const unsigned int Dimension = 2;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
```

然后我们选择用于 mask 图像的像素类型并实例化 mask 的图像类型：

```
typedef unsigned char MaskPixelType;
typedef itk::Image< MaskPixelType, Dimension > MaskImageType;
```

输入和 mask 图像都可以从文件读取或从一个预处理流水线的输出得到。由于处理过程很标准，所以这里我们忽略读取图像的细节。

现在实例化 itk::VnlFFTRealToComplexConjugateImageFilter。注意：和大多数 ITK 滤波器不同，FFT 滤波器是使用像素类型和图像维来进行实例化的。使用这个类型我们构造滤波器的一个实例：

```
typedef itk::VnlFFTRealToComplexConjugateImageFilter<
InputPixelType, Dimension > FFTFilterType;
FFTFilterType::Pointer fftFilter = FFTFilterType::New( );
fftFilter->SetInput( inputReader->GetOutput( ) );
```

由于我们的目的是通过改变图像频域的权重来实现频域的滤波，所以这里我们需要一个可以使用二值图像来 Mask 输入图像的傅立叶变换的滤波器。注意：这里的光谱图像的类型是从 FFT 滤波器的特征得到的。

```
typedef FFTFilterType::OutputImageType SpectralImageType;
typedef itk::MaskImageFilter< SpectralImageType,
MaskImageType,
```

SpectralImageType > MaskFilterType;

MaskFilterType::Pointer maskFilter = MaskFilterType::New( );

我们通过从第一个 FFT 滤波器和从 Mask 图像 reader 的输出作为输入连接到 mask 滤波器：

maskFilter->SetInput1(fftFilter->GetOutput( ));

maskFilter->SetInput2(maskReader->GetOutput( ));

出于核实频域方面的目的，在使用 mask 滤波后，我们可以将 mask 滤波器的输出写到一个文件中：

typedef itk::ImageFileWriter< SpectralImageType > SpectralWriterType;

SpectralWriterType::Pointer spectralWriter = SpectralWriterType::New( );

spectralWriter->SetFileName("filteredSpectrum.mhd");

spectralWriter->SetInput(maskFilter->GetOutput( ));

spectralWriter->Update( );

mask 滤波器的输出将包含输入图像滤波后的频域。然后我们必须对它使用一个傅立叶反变换，来得到输入图像滤波后的情况。为了这个目的我们创建另一个 FFT 滤波器实例：

typedef itk::VnlFFTComplexConjugateToRealImageFilter<

InputPixelType, Dimension > IFFTFilterType;

IFFTFilterType::Pointer fftInverseFilter = IFFTFilterType::New( );

fftInverseFilter->SetInput(maskFilter->GetOutput( ));

可以通过在最后一个滤波器调用 Update( )方式来触发流水线的运行。由于可能抛出错误和异常，所以调用必须在一个 try/catch 模块中：

try

{

fftInverseFilter->Update( );

}

catch(itk::ExceptionObject & excp)

{

std::cerr << "Error: " << std::endl;

std::cerr << excp << std::endl;

return EXIT_FAILURE;

}

现在可以把滤波的结果保存到一个图像文件中，或传递给一个并发的处理流水线。这里我们简单地将它写到一个图像文件中：

typedef itk::ImageFileWriter< InputImageType > WriterType;

WriterType::Pointer writer = WriterType::New( );

writer->SetFileName(argv[3]);

writer->SetInput(fftInverseFilter->GetOutput( ));

注意：这个例子仅仅是在傅立叶领域中处理多样类型的一个最简单阐述。

6.11 提取表面

本小节的源代码在文件 Examples/Filtering/SurfaceExtraction.cxx 中。

由于图像分析的早期研究吸引了人们对表面提取的很大兴趣，尤其是在医学应用背景下。尽管和图像分割相关联，但是表面提取本身并不是一个分割技术，它是一个改变分割方式描述的变换。在它通常的形式中，等值面提取等同于表面提取基础上的图像阈值。

表面提取最普遍的方法是 Marching Cubes 算法。尽管它是在一系列变量基础上的，但是 Marching Cubes 已经成为医学图像处理的标志。下面的例子阐述了如何在 ITK 中使用一个类似于 Marching Cubes 的算法来实现表面提取。

在 ITK 中的未构造数据是用 itk::Mesh 来表示的。这个类允许表达各式各样的 N 维拓扑格子。从一个图像提取表面来生成一个网格作为它的输出是很自然的。

我们通过包含表面提取滤波器、图像和网格的头文件开始：

```
#include "itkBinaryMask3DMeshSource.h"
#include "itkImage.h"
#include "itkMesh.h"
```

然后我们定义我们即将提取表面的图像的像素类型和维：

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

使用同样的图像类型我们实例化 ImageFileReader 类型并带着读取输入图像的目的构造一个对象：

```
typedef itk::ImageFileReader< ImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New( );
reader->SetFileName( argv[1] );
```

通过指定和网格结点的像素值相关的类型来实例化 itk::Mesh 的类型。这个特定的像素类型正好和提取表面的目的是不相关的。

第七章　读与写图像

本章介绍支持对文件的图像读、写操作的工具箱的体系结构。ITK 不支持任何的文件格式；相反，它提供各种各样的能够被用户很容易使其变成可用的新格式的格式。

我们用一些文件 I/O 的例子开始这一章。

7.1　基本例子

这部分的源码在文件Examples/IO/ImageReadWrite.cxx中。可靠的读写图像的类位于数据处理通道的开始和结束。这些类被认为是数据源(readers)和数据槽(writers)。一般来说，它们作为滤波器被涉及了，虽然readers没有输入通道，writers也没有输出通道。

itk::ImageFileReader类管理图像的读入，而itk::ImageFileWriter管理图像的写操作。两个类对于任何文件格式都是独立的。实际上低水平的读、写特殊文件格式的任务由一组itk::ImageIO类在后台执行。

执行读、写的第一步是要包含以下头文件：

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

这时，就像前面一样，必须决定被数据通道处理的表现图像的像素类型。注意：当读、写图像时，图像的像素类型不必与存储在文件中的像素类型一样。你对像素类型的选择主要有以下两个考虑：

- 应该能够将在文件中的像素类型转换为你选择的像素类型。这需要用标准的C语言规则执行，所以必须保证变换没有导致信息丢失。
- 内存中的像素类型应该适合你打算应用于图像的处理类型。

医学图像处理一个典型的选择在下面的代码中介绍：

```
typedef unsigned short PixelType;
const unsigned int Dimension = 2;
typedef itk::Image< PixelType, Dimension > ImageType;
```

注意：在内存中图像的维数应该与文件中的图像的维数相匹配。有一对特别的状态，这种情况也许不是很严格，但一般能够确保两个维数匹配。

我们现在以reader 和writer的类型为例，这两个类通过图像类型被参数化：

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

这时，我们用New( )函数并把结果赋值给一个itk::SmartPointer来创建每个类型的对象：

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

被读或写的文件名用SetFileName( )函数传递：

```
reader->SetFileName( inputFilename );
writer->SetFileName( outputFilename );
```

我们把这些readers和writers连接到滤波器上来创建一个数据通道。例如，我们直接传递reader的输出给writer的输入来创建一个短数据通道：

```
writer->SetInput( reader->GetOutput( ) );
```

这也许是一个很没有用的程序，实际上是一个强大的文件格式转换工具。数据通道的执行被最后对象的Update( )触发。在这个情况下，最终的数据通道对象是writer。在通道被执行时，最明智的自我保护程序的做法是在一个try/catch模块中插入Update( )以抛出异常。

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cerr << "ExceptionObject caught !" << std::endl;
std::cerr << err << std::endl;
return EXIT_FAILURE;
}
```

注意：异常只能被知道用它们去做什么的代码捕捉到。Catch 模块的典型应用应该是存在于GUI代码里。Catch 模块的活动是通知用户IO操作的失败。

工具箱的IO体系结构使得避免外在地用于读、写图像的文件格式的说明是可能的。对象factory机构使ImageFileReader和ImageFileWriter能够决定(在运行时间)它将用哪种文件格式工作。典型地，文件格式基于文件扩展名选择，但是体系结构支持决定一个文件是否能被读或写的任意复杂的程序。作为选择，用户可以通过外在实例化和设置适当的itk::ImageIO子类来指定数据文件格式。

因为历史原因和方便用户，itk::ImageFileWriter也有一个Update( )，别名是Write( )。你能使用任意一个，但是推荐Update( )，因为Write( )也许在将来就会被摒弃。

结合图7-1至图7-3所示的讲解，可以更好地理解IO体系结构。

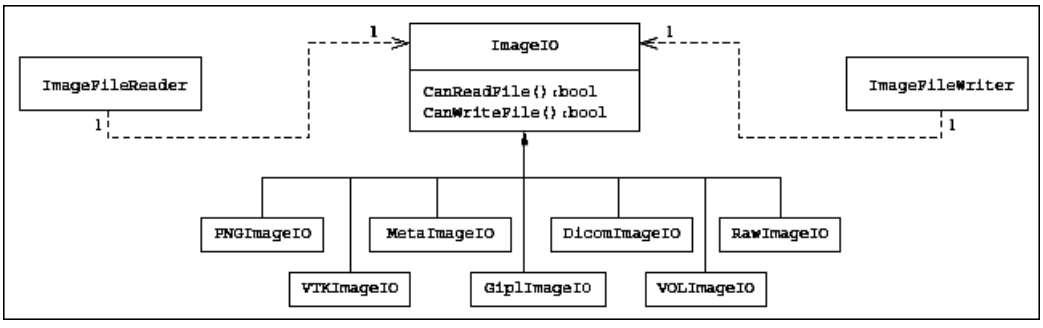


图 7-1　ImageIO 类的层次结构图

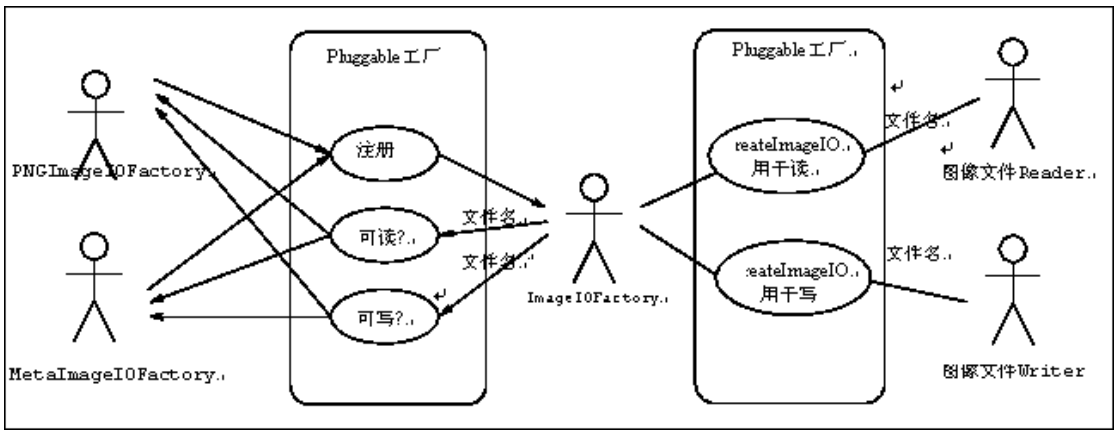


图 7-2　ImageIO 工厂的使用案例

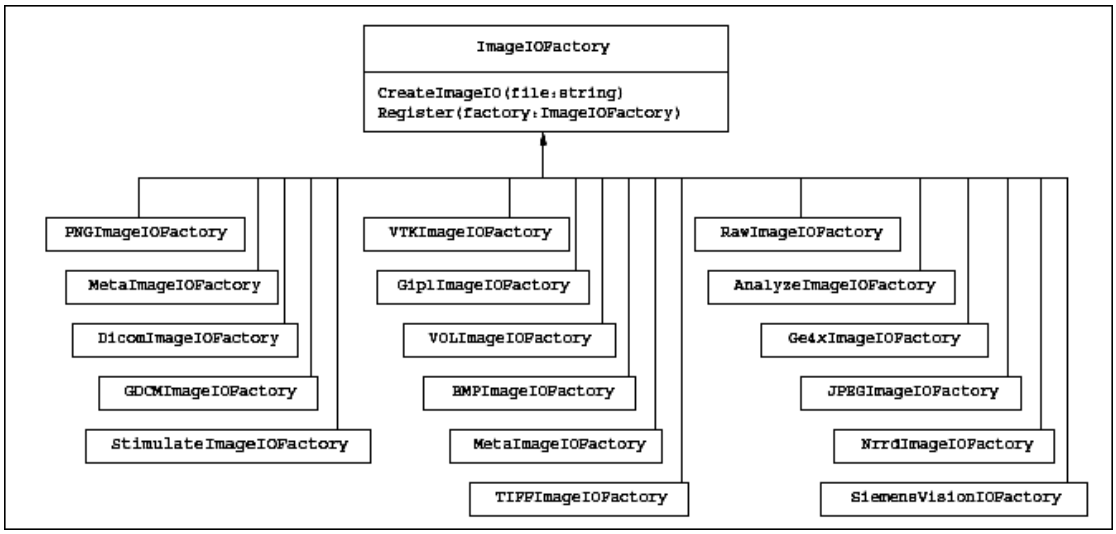


图 7-3　ImageIO 工厂的类结构表

下面介绍在工具箱中提供的IO体系结构的内部框架。

7.2　插拔式工厂

ITK 中在输入/输出结构后面的原理叫做插拔式工厂。这个概念被解释在图 7-1 所示的 UML 图表里。从用户的观点看，可靠的读、写文件的类是 itk::ImageFileReader 和 itk::ImageFileWriter。这两个类不知道读或写特殊文件格式如 PNG 或 DICOM 的细节，它们所做的就是分派用户的要求给知道文件格式细节的类。这些类是 itk::ImageIO 类。ITK 授权机构使用户能够通过添加新类给 ImageIO 来扩展被支持的文件格式的数量。

ImageFileReader 和 ImageFileWriter 的每一个例子都有一个指向 ImageIO 对象的一个指针。如果指针是空的，读或写一个图像是不可能的，图像文件 reader/writer 必须决定由哪一个 ImageIO 类去执行 IO 操作。这个通过传递文件名给中央类、itk::ImageIOFactory 和确定读

或写用户指定的文件的能力的 ImageIO 的子类来做。这个通过图 7-2 所示右边的例子进行介绍。

来源于 ImageIO 的每一个类必须提供一个达到 ImageIO 类要求的关联的工厂类。例如 PNG 文件，有一个知道如何读文件的 itk::PNGImageIO 的对象和有一个能够构建 PNGImageIO 对象并能返回指向它的一个指针的 itk::PNGImageIOFactory 类。每一次一个新的文件格式被添加时(例如一个新的 ImageIO 子类被创建)，一个工厂类必须作为在图 7-3 所示介绍的 ImageIOFactory 类的一个父类被执行。

例如，为了读 PNG 文件，一个 PNGImageIOFactory 文件被创建并被图 7.2 中左边介绍的中央 ImageIOFactory 单独类配准。当 ImageFileReader 要求 ImageIOFactory 具有作为一个读文件的 ImageIO 能力，ImageIOFactory 将会按照配准工厂的列表进行迭代，并询问它们是否知道如何去读文件。可以作出肯定地响应的工厂将用于创建特殊的 ImageIO 案例，这个案例将会被返回到 ImageFileReader 并用于执行读操作。

在大多数情况下，机构对于仅仅与 ImageFileReader 和 ImageFileWriter 结合的用户来说是透明的。然而，明确地选择 ImageIO 的对象类型也是可能的。下面的例子会有介绍。

7.3 明确地使用 ImageIO 类

这部分的代码在文件Examples/IO/ImageReadExportVTK.cxx中。

在用户知道用什么文件格式并想明确地指出的情况下，能够以一个itk::ImageIO类为例并赋给图像文件reader或writer。这要围绕着尝试找到合适的ImageIO类去执行IO操作的itk::ImageIOFactory机构。ImageIO的外在选择也允许用户调用一个特殊类的专门特征，这个类也许在ImageIO提供的普通的API中不可用。

下面的例子介绍了一个IO类的外在例子(一个VTK文件格式)，设置它的参数并将它连接到itk::ImageFileWriter。

例子的开始先要包含以下头文件：

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkVTKImageIO.h"
```

这时，我们选择像素类型和图像的维数。记住，如果文件的格式用一种特殊的类型表示像素，C风格的计算将会执行数据的变换：

```
typedef unsigned short PixelType;
const unsigned int Dimension = 2;
typedef itk::Image< PixelType, Dimension > ImageType;
```

我们现在能够以reader和writer为例。这两个类被图像类型参数化。我们也以itk::VTKImageIO类为例。注意：ImageIO对象不是模板类。

```
typedef itk::ImageFileReader< ImageType > ReaderType;
typedef itk::ImageFileWriter< ImageType > WriterType;
typedef itk::VTKImageIO ImageIOType;
```

这时，我们用New( )函数创建每一个类型的对象并把结果赋给itk::SmartPointer：

```
ReaderType::Pointer reader = ReaderType::New( );
WriterType::Pointer writer = WriterType::New( );
ImageIOType::Pointer vtkIO = ImageIOType::New( );
```

被读或写的文件名由SetFileName( )来传递：

```
reader->SetFileName( inputFilename );
writer->SetFileName( outputFilename );
```

现在我们能够在数据通道里将readers和writers连接到滤波器上。例如，我们能够直接将reader的输出数据传递给writer的输入来创建一个短数据通道：

```
writer->SetInput( reader->GetOutput( ) );
```

VTKImageIO允许用户调用特殊IO类的函数。例如，当写像素数据时，下面的代码用ASCII格式：

```
vtkIO->SetFileTypeToASCII( );
```

VTKImageIO对象这时连接到ImageFileWriter上，这将缩短ImageIOFactory机构的循环。ImageFileWriter不会寻找另外能够执行写操作的ImageIO对象。它只是简单地调用了用户提供的一个ImageIO对象。

```
writer->SetImageIO( vtkIO );
```

最后我们调用ImageFileWriter上的Updata( )并放置这个问询在一个try/catch模块中，以防在写过程中有任何错误发生：

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cerr << "ExceptionObject caught !" << std::endl;
std::cerr << err << std::endl;
return EXIT_FAILURE;
}
```

虽然这个例子仅仅介绍了如何用带ImageFileWriter的一个外在ImageIO类，但是ImageFileReader同样可以做。典型情况是用itk::RawImageIO对象读原始图像文件时。这种方法的缺点是图像的参数必须明确地写在代码里。在医学图像中直接使用原始文件根本不合适。用联合文本头文件或原始二元文件，像itk::MetaImageIO、itk::GiplImageIO和itk::VTKImageIO创建一个原始文件的头文件最好。

7.4 读、写RGB图像

这部分源码在文件Examples/IO/RGBImageReadWrite.cxx中。

RGB图像普遍用于表示从低温截面、光学显微镜和内窥镜中获得的数据。这个例子介绍了如何对于一个文件读或写RGB颜色图像。要求的头文件为：

```
#include "itkRGBPixel.h"
#include "itkImage.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

itk::RGBPixel是一个表现红、绿和蓝成分的模板类。典型的RGB图像类应用如下：

```
typedef itk::RGBPixel< unsigned char > PixelType;
typedef itk::Image< PixelType, 2 > ImageType;
```

图像类型用于一个表示reader和writer的模板参数：

```
typedef itk::ImageFileReader< ImageType > ReaderType;
typedef itk::ImageFileWriter< ImageType > WriterType;
ReaderType::Pointer reader = ReaderType::New( );
WriterType::Pointer writer = WriterType::New( );
```

输入、输出的文件的文件名必须分别提供给reader和writer：

```
reader->SetFileName( inputFilename );
writer->SetFileName( outputFilename );
```

最后，通过调用writer中的Update( )函数触发数据通道的完成：

```
writer->Update( );
```

你也许注意到从PixelType的声明里分离的表示RGB图像的代码没有什么东西，所有要求支持颜色图像的部分在itk::ImageIO对象里被内部执行。

7.5 读、重塑和写图像

这部分的代码在文件Examples/IO/ImageReadCastWrite.cxx中。

ITK基于范型编程的，大多数类型定义在编辑时间上。最重要的是预见不同图像之间的转化。下面的例子介绍了读一种像素类型的图像并写它用不同类型的普遍情况。这个过程不仅要调用重塑而且要缩放图像亮度，因为输入输出的像素类型的动态范围可能是不同的。itk::RescaleIntensityImageFilter用来线性调节图像值。

第一步是包含以下头文件：

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkRescaleIntensityImageFilter.h"
```

这时，要决定用于表示图像的像素类型。注意：读图像时，像素的类型不必与文件中存储的像素类型相同，而是与将要读入内存的类型相同。

```
typedef float InputPixelType;
typedef unsigned char OutputPixelType;
const unsigned int Dimension = 2;
```

typedef itk::Image< InputPixelType, Dimension > InputImageType;

typedef itk::Image< OutputPixelType, Dimension > OutputImageType;

注意：在内存中的图像的维数和文件中的维数要匹配。有一对特殊情况，这种状况也许不严格，但通常能更好地确保两个维数匹配。

我们现在以reader和writer的类型为例。这两个类通过图像参数化。

typedef itk::ImageFileReader< InputImageType > ReaderType;

typedef itk::ImageFileWriter< OutputImageType > WriterType;

下面我们以可以将图像亮度进行比例缩放的RescaleIntensityImageFilter类为例。

typedef itk::RescaleIntensityImageFilter<

InputImageType,

OutputImageType > FilterType;

一个滤波器被创建并且通过SetOutputMinimum( )和SetOutputMaximum( )选择输出的最小值和最大值：

FilterType::Pointer filter = FilterType::New( );

filter->SetOutputMinimum(0);

filter->SetOutputMaximum(255);

下面我们创建reader和writer并连接数据通道：

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

WriterType::Pointer writer = WriterType::New( );

filter->SetInput(reader->GetOutput( ));

writer->SetInput(filter->GetOutput( ));

用SetFileName( )来传递被读和被写的文件名：

reader->SetFileName(inputFilename);

writer->SetFileName(outputFilename);

最后，我们用writer上的Update( )来触发通道的执行。这时输出图像将被缩放比例且cast输入图像的版本。

try

{

writer->Update( );

}

catch(itk::ExceptionObject & err)

{

std::cerr << "ExceptionObject caught !" << std::endl;

std::cerr << err << std::endl;

return EXIT_FAILURE;

}

7.6 提取区域

这部分的代码在文件Examples/IO/ImageReadRegionOfInterestWrite.cxx中。

这个例子应该被放在先前的滤波章节。然而它对于典型的IO操作的有用性使得它在这里需要提及它。这个例子的目的是读图像，提取一个子区域并把这个子区域写入文件中。当我们对感兴趣的区域应用一个计算加强方法时，这是一个普遍的任务。对于ITK IO，我们首先要包含合适的头文件：

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

itk::RegionOfInterestImageFilter是一个从一个图像中提取一个区域的滤波器。它的头文件是：

```
#include "itkRegionOfInterestImageFilter.h"
```

图像类型定义如下：

```
typedef signed short InputPixelType;
typedef signed short OutputPixelType;
const unsigned int Dimension = 2;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

itk::ImageFileReader和itk::ImageFileWriter的类型用图像类型定义：

```
typedef itk::ImageFileReader< InputImageType > ReaderType;
typedef itk::ImageFileWriter< OutputImageType > WriterType;
```

RegionOfInterestImageFilter的类型以输入输出的图像为例。用New( )创建一个滤波器对象并把它赋给itk::SmartPointer：

```
typedef itk::RegionOfInterestImageFilter< InputImageType,
OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

RegionOfInterestImageFilter要求用户定义一个区域。这区域被itk::Index和itk::Size指定，itk::Index指出区域开始的像素，而itk::Size指出沿着每一维有多少像素。在这个例子里，区域的说明可以从命令行中得到(这个例子假设的是执行2D图形)。

```
OutputImageType::IndexType start;
start[0] = atoi( argv[3] );
start[1] = atoi( argv[4] );
OutputImageType::SizeType size;
size[0] = atoi( argv[5] );
size[1] = atoi( argv[6] );
```

一个itk::ImageRegion对象被创建，并用从命令行获得的起点和大小进行初始化。

```
OutputImageType::RegionType desiredRegion;
desiredRegion.SetSize( size );
desiredRegion.SetIndex( start );
```

这时，用SetRegionOfInterest( )将这个区域传递给滤波器：

```
filter->SetRegionOfInterest( desiredRegion );
```

下面，我们用New( )创建reader和writer并把结果赋给SmartPointer。

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

被读或被写的文件名用SetFileName( )传递：

```
reader->SetFileName( inputFilename );
writer->SetFileName( outputFilename );
```

下面我们连接reader、滤波器和writer去形成数据处理通道：

```
filter->SetInput( reader->GetOutput( ) );
writer->SetInput( filter->GetOutput( ) );
```

最终，我们通过调用writer上的Updata( )触发通道。将问询放到try/catch模块中以防异常出现。

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cerr << "ExceptionObject caught !" << std::endl;
std::cerr << err << std::endl;
return EXIT_FAILURE;
}
```

7.7 提取切片

这部分的代码在文件Examples/IO/ImageReadExtractWrite.cxx中。

这个例子介绍了从一个3D体中提取一个切片的普通任务。最典型的是用在显示目的和交互程序中加快用户反馈。注意：当操作从3D数据中的2D切片时应该谨慎，因为对于大多数图像处理操作，一个提取切片的滤波器的程序不和在体中首次应用的滤波器以及这时提取切片的滤波器相同。

这个例子中我们开始要包含以下头文件：

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

用于从一个图像中提取一个区域的滤波器是itk::ExtractImageFilter。它的头文件在下面。这个滤波器能够从N维图像中提取N−1维图像。

```
#include "itkExtractImageFilter.h"
```

下面定义图像类型。注意：输入图像的类型是3D的，而输出图像是2D的。

```
typedef signed short InputPixelType;
```

typedef signed short OutputPixelType;

typedef itk::Image< InputPixelType, 3 > InputImageType;

typedef itk::Image< OutputPixelType, 2 > OutputImageType;

itk::ImageFileReader和itk::ImageFileWriter的类型以图像类型为例：

typedef itk::ImageFileReader< InputImageType > ReaderType;

typedef itk::ImageFileWriter< OutputImageType > WriterType;

下面我们用New( )创建reader和writer并把结果赋予itk::SmartPointer：

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

WriterType::Pointer writer = WriterType::New( );

被读或被写的文件名用SetFileName( )传递：

reader->SetFileName(inputFilename);

writer->SetFileName(outputFilename);

ExtractImageFilter的类型以输入、输出图像的类型为例。由New( )创建一个滤波对象并把它赋予SmartPointer：

typedef itk::ExtractImageFilter< InputImageType, OutputImageType > FilterType;

FilterType::Pointer filter = FilterType::New( );

ExtractImageFilter要求用户定义一个区域。这个区域被itk::Index和itk::Size指定，itk::Index指定开始区域的像素，itk::Size指定沿着每维这个区域有多少像素。为了从一个3D数据集中提取一个2D图像，在其中一维上设置区域大小为零就可以了。这将说明ExtractImageFilter的一维的减小量已经指定。这里我们拿出输入图像中的最大可能区域。注意：Update( )首先在reader被问询，因为否则输出的将是残损数据。

reader->Update( );

InputImageType::RegionType inputRegion =

reader->GetOutput( )->GetLargestPossibleRegion( );

我们从区域中提取大小并将*Z*方向设置为零。这就意味着对于ExtractImageFilter，输出图像比输入图像少一维。

InputImageType::SizeType size = inputRegion.GetSize( );

size[2] = 0;

这里我们做的是Z切片，因此维数变为2。你也可以记住检索成分的联合是{X=0,Y=1,Z=2}。如果我们对垂直于*Y*轴做个切片也感兴趣，我们也可以设置size[1]=0。

这时，我们从该区域中的检索和设置提取我们想要的切片的数目的Z值。这个例子中，我们从命令行中获得切片数。

InputImageType::IndexType start = inputRegion.GetIndex( );

const unsigned int sliceNumber = atoi(argv[3]);

start[2] = sliceNumber;

最后，一个itk::ImageRegion对象被创建，并使用我们前面用切片信息定义的起点和大小对对象进行初始化：

InputImageType::RegionType desiredRegion;

desiredRegion.SetSize(size);

desiredRegion.SetIndex(start);

这时，这个区域由SetExtractionRegion( )传给滤波器：

filter->SetExtractionRegion(desiredRegion);

下面我们连接reader、滤波器和writer来形成数据处理通道：

filter->SetInput(reader->GetOutput( ));

writer->SetInput(filter->GetOutput( ));

最后，我们调用writer的Updata( )来触发通道的执行。在try/catch模块中放置问询以防异常情况出现：

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cerr << "ExceptionObject caught !" << std::endl;
std::cerr << err << std::endl;
return EXIT_FAILURE;
}
```

7.8 读、写向量图像

像素类型是向量、协向量、队列或者综合的图像，在图像处理中很常见。快速地描述这些图像如何被保存到文件和如何从这些文件中读取十分方便。

7.8.1 最简单的例子

这部分的代码在文件Examples/IO/VectorImageReadWrite.cxx中。

下面的例子介绍了如何读和写itk::Vector像素类型的图像。

我们要包含以下头文件：

```
#include "itkImage.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

接着我们定义用作像素类型的向量类型：

```
const unsigned int VectorDimension = 3;
typedef itk::Vector< float, VectorDimension > PixelType;
```

下面我们定义图像类型：

```
const unsigned int ImageDimension = 2;
typedef itk::Image< PixelType, ImageDimension > ImageType;
```

有了图像类型，我们能够定义reader和writer的类型，并用它们创建每个类型的对象：

```
typedef itk::ImageFileReader< ImageType > ReaderType;
typedef itk::ImageFileWriter< ImageType > WriterType;
ReaderType::Pointer reader = ReaderType::New( );
WriterType::Pointer writer = WriterType::New( );
```

文件名必须提供给reader和writer。特殊情况下，我们从命令行中得到文件名。

```
reader->SetFileName( argv[1] );
writer->SetFileName( argv[2] );
```

作为一个最小化的例子，我们创建一个简单连接reader输出和writer输入的短数据通道：

```
writer->SetInput( reader->GetOutput( ) );
```

这个短通道的执行通过调用writer的Update( )来触发。这个调用必须放置在一个try/catch模块里，因为它的执行也许会导致异常情况的出现。

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cerr << "ExceptionObject caught !" << std::endl;
std::cerr << err << std::endl;
return EXIT_FAILURE;
}
```

当然，你能够想象在reader和writer之间滤波器的加法。这些滤波器可以执行向量图像的操作。

7.8.2 生成和写入协变图像

这部分的代码在文件Examples/IO/CovariantVectorImageWrite.cxx中。

这个例子介绍了如何去写像素类型是CovariantVector的图像。为了切合实际，例子中所有的内容都应用像素类型为itk::Vector、itk::Point和itk::FixedArray的图像。这些像素类型在成分有相同代表性的类型中所有队列都是固定大小的。

为了使这个例子更加有趣，我们设置一个通道去读一个图像，计算它的梯度并把它写入一个文件中。梯度用于向量相反的itk::CovariantVector来表示。用这个方法，梯度在itk::AffineTransform下被直接变换或一般情况下任何变换有各向异性的比例。

让我们开始先包含以下头文件：

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

我们用itk::GradientRecursiveGaussianImageFilter来计算图像梯度。滤波器的输出是一幅像素是CovariantVectors的图像：

```
#include "itkGradientRecursiveGaussianImageFilter.h"
```

我们选择读一幅有符号短整型像素的图像和计算梯度以生成一幅像素类型是浮点型的CovariantVector图像：

```
typedef signed short InputPixelType;
typedef float ComponentType;
const unsigned int Dimension = 2;
typedef itk::CovariantVector< ComponentType,
Dimension > OutputPixelType;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

itk::ImageFileReader和itk::ImageFileWriter都以图像类型为例：

```
typedef itk::ImageFileReader< InputImageType > ReaderType;
typedef itk::ImageFileWriter< OutputImageType > WriterType;
```

GradientRecursiveGaussianImageFilter以输入和输出图像的类型例示。用New( )创建一个滤波器对象并把它赋给itk::SmartPointer：

```
typedef itk::GradientRecursiveGaussianImageFilter<
InputImageType,
OutputImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

我们选择一个值作为GradientRecursiveGaussianImageFilter的σ参数。注意σ的单位是毫米。

```
filter->SetSigma( 1.5 ); // Sigma in millimeters
```

下面我们用New( )创建reade和writer并把结果赋给SmartPointer：

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

被读和被写的文件名由SetFileName( )传递：

```
reader->SetFileName( inputFilename );
writer->SetFileName( outputFilename );
```

下面我们连接reader、滤波器和writer形成数据处理通道：

```
filter->SetInput( reader->GetOutput( ) );
writer->SetInput( filter->GetOutput( ) );
```

最后，我们通过调用writer的Update( )函数触发通道的执行。在try/catch模块中放置问询以防异常的发生：

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cerr << "ExceptionObject caught !" << std::endl;
```

```
std::cerr << err << std::endl;
return EXIT_FAILURE;
}
```

7.8.3 读协变式图像

现在让我们提取出我们刚创建的图像并将它读到另一个程序中。

这部分的代码在文件Examples/IO/CovariantVectorImageRead.cxx中。

本例介绍如何读一幅像素类型是CovariantVector的图像。为了切合实际，这个例子使用像素类型为itk::Vector、itk::Point和itk::FixedArray的图像。这些像素类型在固定大小的全队列中是相似的，这种固定大小也是在有相同表示类型的成分中。

这个例子中我们从一个文件(被当前的例子写的)中读一个梯度函数并用itk::GradientToMagnitudeImageFilter计算它的大小。这个滤波器不同于itk::GradientMagnitudeImageFilter。itk::GradientMagnitudeImageFilter实际上获取一个比例图像作为输入并计算它梯度的大小。GradientToMagnitudeImageFilter类获取一幅向量像素图像作为输入并计算每个向量的明确像素的数量。

开始我们要包含以下头文件：

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkGradientToMagnitudeImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"
```

我们读itk::CovariantVector像素的一幅图像并计算像素数量去生成一幅每个像素都是unsigned short的像素的图像。这里CovariantVector的成分选择float。注意：要求重正化以便映射数量值的动态范围到输出像素类型的输出范围。itk::RescaleIntensityImageFilter用于完成这个任务：

```
typedef float ComponentType;
const unsigned int Dimension = 2;
typedef itk::CovariantVector< ComponentType,
Dimension > InputPixelType;
typedef float MagnitudePixelType;
typedef unsigned short OutputPixelType;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< MagnitudePixelType, Dimension > MagnitudeImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

itk::ImageFileReader和itk::ImageFileWriter以图像类型为例：

```
typedef itk::ImageFileReader< InputImageType > ReaderType;
typedef itk::ImageFileWriter< OutputImageType > WriterType;
```

GradientToMagnitudeImageFilter以输入输出图像类型为例。由New( )创建一个滤波器对象并把它赋给itk::SmartPointer：

```
typedef itk::GradientToMagnitudeImageFilter<
InputImageType,
MagnitudeImageType > FilterType;
FilterType::Pointer filter = FilterType::New( );
```

The RescaleIntensityImageFilter class is instantiated next.

```
typedef itk::RescaleIntensityImageFilter<
MagnitudeImageType,
OutputImageType > RescaleFilterType;
RescaleFilterType::Pointer rescaler = RescaleFilterType::New( );
```

下面输出图像的最小值和最大值被指定。注意itk::NumericTraits类的用法，它允许用一个普通的方法定义很多关系型常数。显著的用法是普通编程的基本原理特征。

```
rescaler->SetOutputMinimum( itk::NumericTraits< OutputPixelType >::min( ) );
rescaler->SetOutputMaximum( itk::NumericTraits< OutputPixelType >::max( ) );
```

下面我们用New( )创建reader并把结果赋给SmartPointer：

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

SetFileName( )传递被读或被写的文件名：

```
reader->SetFileName( inputFilename );
writer->SetFileName( outputFilename );
```

下面我们连接reader、滤波器和writer形成数据处理通道：

```
filter->SetInput( reader->GetOutput( ) );
rescaler->SetInput( filter->GetOutput( ) );
writer->SetInput( rescaler->GetOutput( ) );
```

最后我们调用writer的Update( )来触发通道。放问询在try/catch模块中以防异常出现：

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cerr << "ExceptionObject caught !" << std::endl;
std::cerr << err << std::endl;
return EXIT_FAILURE;
}
```

7.9 读、写合成图像

这部分的代码在文件Examples/IO/ComplexImageReadWrite.cxx中。

下面的例子介绍了如何读和写一幅std::complex像素类型的图像。合成类型被定义为C++语言不可缺少的部分。类型的特征在“数字化图书馆”第26章，特别是26.2节有定义。

开始，我们先包含合成类、图像、reader和writer的头文件：

```
#include <complex>
#include "itkImage.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
```

图像维数和像素类型必须声明。这里我们以std::complex<>作为像素类型。我们用维数和像素类型作为图像类型的示例。

```
const unsigned int Dimension = 2;
typedef std::complex< float > PixelType;
typedef itk::Image< PixelType, Dimension > ImageType;
```

图像文件的reader和writer的类型以图像的类型为例。这时我们能创建它们的对象。

```
typedef itk::ImageFileReader< ImageType > ReaderType;
typedef itk::ImageFileWriter< ImageType > WriterType;
ReaderType::Pointer reader = ReaderType::New( );
WriterType::Pointer writer = WriterType::New( );
```

文件名应该提供给reader和writer。在该特殊例子中从命令行中获得文件名。

```
reader->SetFileName( argv[1] );
writer->SetFileName( argv[2] );
```

这里我们简单连接reader的输出和writer的输入。这个简单程序能够将合成图像从一种文件格式转换成另一种文件格式。

```
writer->SetInput( reader->GetOutput( ) );
```

通过调用writer的Update( )函数触发短通道的执行。问询必须放在一个try/catch模块中，因为它的运行可能导致异常情况发生。

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cerr << "ExceptionObject caught !" << std::endl;
std::cerr << err << std::endl;
return EXIT_FAILURE;
}
```

这些代码的用途是：你可以在reader和writer之间添加一个滤波器并执行任何合成图像的操作。这些代码的一个特别应用在6.10节的Fourier分析中有介绍。

7.10 从向量图像中提取成分

这部分的代码在文件Examples/IO/CovariantVectorImageExtractComponent.cxx中。

这个例子介绍了如何读一幅像素类型是CovariantVector的图像，提取它的一个成员来形成一个梯度图像，并最终保存图像到一个文件中。

itk::VectorIndexSelectionCastImageFilter用于从向量图像中提取一个梯度。当用这个滤波器的时候投射成分类型也是可能的。确保投射的成分不会导致任何丢失是用户的职责。

开始时，现包含以下相关头文件：

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkVectorIndexSelectionCastImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"
```

我们读itk::CovariantVector像素的一幅图像，并提取它的成分生成一个相容像素类型的梯度图像。这时我们重新调节梯度图像的亮度并写它作为一幅unsigned short像素图像。

```
typedef float ComponentType;
const unsigned int Dimension = 2;
typedef itk::CovariantVector< ComponentType,
Dimension > InputPixelType;
typedef unsigned short OutputPixelType;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
typedef itk::Image< ComponentType, Dimension > ComponentImageType;
typedef itk::Image< OutputPixelType, Dimension > OutputImageType;
```

itk::ImageFileReader和itk::ImageFileWriter以图像类型为例：

```
typedef itk::ImageFileReader< InputImageType > ReaderType;
typedef itk::ImageFileWriter< OutputImageType > WriterType;
```

VectorIndexSelectionCastImageFilter被输入、输出的图像类型例示。用New( )创建一个滤波器类型并把它赋给itk::SmartPointer：

```
typedef itk::VectorIndexSelectionCastImageFilter<
InputImageType,
ComponentImageType > FilterType;
FilterType::Pointer componentExtractor = FilterType::New( );
```

VectorIndexSelectionCastImageFilter类要求我们指定从向量图像中提取哪种向量成分。任务由SetIndex( )来执行。在这个例子中，我们从命令行中获得这个值。

```
componentExtractor->SetIndex( indexOfComponentToExtract );
```

itk::RescaleIntensityImageFilter滤波器在这里被示例。

```
typedef itk::RescaleIntensityImageFilter<
ComponentImageType,
OutputImageType > RescaleFilterType;
```

RescaleFilterType::Pointer rescaler = RescaleFilterType::New( );

对于输出图像的最小值和最大值在下面指定。注意itk::NumericTraits类的用途，它允许在普通方法里定义很多类型的关系常数。使用特征是范型编程的一个基本原则。

rescaler->SetOutputMinimum(itk::NumericTraits< OutputPixelType >::min( ));

rescaler->SetOutputMaximum(itk::NumericTraits< OutputPixelType >::max( ));

下面，我们用New( )创建reader和writer并把结果赋给SmartPointer：

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

WriterType::Pointer writer = WriterType::New( );

SetFileName( )传递被读和被写的文件名：

reader->SetFileName(inputFilename);

writer->SetFileName(outputFilename);

下面我们连接reader、滤波器和writer形成数据处理通道：

componentExtractor->SetInput(reader->GetOutput( ));

rescaler->SetInput(componentExtractor->GetOutput( ));

writer->SetInput(rescaler->GetOutput( ));

最后我们调用writer的Update( )来触发通道的执行。问询被放在try/catch模块中以防异常情况的发生。

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cerr << "ExceptionObject caught !" << std::endl;
std::cerr << err << std::endl;
return EXIT_FAILURE;
}
```

7.11 读、写序列图像

存储3D医学图像在一个文件集中仍然是很普遍的，每一个文件包含一幅体数据的单独切片。这些2D文件可以作为单独的2D图像来读，也能组成一组构成一个3D数据集。这个方法也可以扩展到更高的维数，例如，用一些3D文件的集合组成4D的数据集。这些在心脏的成像、灌注以及MRI和PET特性中是很普遍的。这部分介绍了在ITK中处理读和写文件使用的广泛性。

7.11.1 读序列图像

这部分的源码在文件Examples/IO/ImageSeriesReadWrite.cxx中。

这个例子介绍了如何从独立的文件中读取一系列的2D切片从而形成一个体。这里要用到itk::ImageSeriesReader。这个类与提供需要被读的文件的一个列表的发生器联合工作。这里我们用itk::NumericSeriesFileNames作为发生器。这个发生器是用一个字符串格式的打印风格。这里我们用一个像“file%03d.png”的格式读取名字为file001.png、file002.png、file003.png……的PNG文件。

需要下面的头文件：

```
#include "itkImage.h"
#include "itkImageSeriesReader.h"
#include "itkImageFileWriter.h"
#include "itkNumericSeriesFileNames.h"
#include "itkPNGImageIO.h"
```

我们首先定义PixelType和ImageType：

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

图像类型用于定义reader和writer的模板参数：

```
typedef itk::ImageSeriesReader< ImageType > ReaderType;
typedef itk::ImageFileWriter< ImageType > WriterType;
ReaderType::Pointer reader = ReaderType::New( );
WriterType::Pointer writer = WriterType::New( );
```

这时，我们声明一个文件名生成器的类型并创建一个例子：

```
typedef itk::NumericSeriesFileNames NameGeneratorType;
NameGeneratorType::Pointer nameGenerator = NameGeneratorType::New( );
```

文件名发生器要求我们提供文件名的原文的模式，初始值的数量，最后值和被用作产生文件名的增加值。

```
nameGenerator->SetSeriesFormat( "vwe%03d.png" );
nameGenerator->SetStartIndex( first );
nameGenerator->SetEndIndex( last );
nameGenerator->SetIncrementIndex( 1 );
```

实际执行读取任务的ImageIO对象现在被连接到ImageSeriesReader。这是确保我们用适于我们想要读取的文件类型的ImageIO对象的最安全方法。

```
reader->SetImageIO( itk::PNGImageIO::New( ) );
```

输入文件的文件名必须提供给reader，而writer在一个单独的文件中写相同的体数据：

```
reader->SetFileNames( nameGenerator->GetFileNames( ) );
writer->SetFileName( outputFilename );
```

我们将reader的输出连接到writer的输入上：

```
writer->SetInput( reader->GetOutput( ) );
```

最后，我们调用writer的Updata函数触发通道的执行。问询必须放在一个try/catch模块里以防在读或写过程中异常情况的发生。

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & err )
{
std::cerr << "ExceptionObject caught !" << std::endl;
std::cerr << err << std::endl;
return EXIT_FAILURE;
}
```

7.11.2 写序列图像

这部分的源码在文件Examples/IO/ImageReadImageSeriesWrite.cxx中。

这个例子介绍了如何用itk::ImageSeriesWriter保存一幅图像。这个类能用每个文件包含一个2D切片的文件集方法保存一个3D体。

这里声明输入图像的类型并用来声明reader的类型。它将是一个传统的3D图像reader。

```
typedef itk::Image< unsigned char, 3 > ImageType;
typedef itk::ImageFileReader< ImageType > ReaderType;
```

用New( )构建reader对象并把结构赋给SmartPointer。从命令行中读出需要被读取的3D体的文件名并用SetFileName( )传给reader。

```
ReaderType::Pointer reader = ReaderType::New( );
reader->SetFileName( argv[1] );
```

连续writer的类型必须考虑输入文件是一个3D体而输出文件是2D图像。另外，reader的输出作为输入传给writer。

```
typedef itk::Image< unsigned char, 2 > Image2DType;
typedef itk::ImageSeriesWriter< ImageType, Image2DType > WriterType;
WriterType::Pointer writer = WriterType::New( );
writer->SetInput( reader->GetOutput( ) );
```

writer要求一个产生的文件名的清单。这个清单能够通过itk::NumericSeriesFileNames类的帮助产生。

```
typedef itk::NumericSeriesFileNames NameGeneratorType;
NameGeneratorType::Pointer nameGenerator = NameGeneratorType::New( );
```

NumericSeriesFileNames类要求一个输入的字符串以便有一个生成所有输出切片的文件名的模板。这里我们用从命令行中读取出的一个前缀来排定字符串并添加PNG文件的扩展。

```
std::string format = argv[2];
format += "%03d.";
format += argv[3]; // filename extension
nameGenerator->SetSeriesFormat( format.c_str( ) );
```

输入的字符串将通过设置第一和最后切片的值生成文件名。通过从输入图像中收集信息做这些事情。注意：在从reader读取任何信息之前，它的执行必须由Update( )触发，并且这个调用可能出现异常，所以必须放在一个try/catch模块中。

```
try
{
reader->Update( );
}
catch( itk::ExceptionObject & excp )
{
std::cerr << "Exception thrown while reading the image" << std::endl;
std::cerr << excp << std::endl;
}
```

现在图像已经被读取，我们可以查询它的最大可能区域和恢复关于沿每个维度像素的数量：

```
ImageType::ConstPointer inputImage = reader->GetOutput( );
ImageType::RegionType region = inputImage->GetLargestPossibleRegion( );
ImageType::IndexType start = region.GetIndex( );
ImageType::SizeType size = region.GetSize( );
```

有了这些信息，我们能找出确定第一和最后3D数据集的切片的数值。这些数值将会传递给文件名发生器对象，它会排列存储切片的文件名。

```
const unsigned int firstSlice = start[2];
const unsigned int lastSlice = start[2] + size[2] - 1;
nameGenerator->SetStartIndex( firstSlice );
nameGenerator->SetEndIndex( lastSlice );
nameGenerator->SetIncrementIndex( 1 );
```

文件名发生器产生文件名列表并将它传给连续writer：

```
writer->SetFileNames( nameGenerator->GetFileNames( ) );
```

最后，我们用writer的Update( )触发管道的执行。这时图像的切片保存在单独的文件中，每个文件包含一个单独的切片。用于这些切片的文件名由文件名发生器生成。

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & excp )
{
std::cerr << "Exception thrown while reading the image" << std::endl;
std::cerr << excp << std::endl;
}
```

注意：通过保存数据到单独的切片中，我们丢失了一些对于医学应用有用的信息，例如

在毫米间距之内的信息。

7.11.3 读、写 RGB 序列图像

这部分的源码在文件Examples/IO/RGBImageSeriesReadWrite.cxx中。

RGB图像普遍用在从低温截面、光学显微镜和内窥镜等方法获得的数据中。这个例子介绍了如何从一个构成一个3D颜色数据的2D切片的文件集中读取RGB颜色图像。这时将它保存为一个单独的3D文件并再一次用其他文件名保存一个2D切片集。

需要下面的头文件：

```
#include "itkRGBPixel.h"
#include "itkImage.h"
#include "itkImageFileWriter.h"
#include "itkImageSeriesReader.h"
#include "itkImageSeriesWriter.h"
#include "itkNumericSeriesFileNames.h"
#include "itkPNGImageIO.h"
```

itk::RGBPixel类是一个模板，用于表示红、绿和蓝成分的类型。一个典型的RGB图像类的例子如下：

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

图像类型作为模板参数去例示连续reader和测定体积的writer：

```
typedef itk::ImageSeriesReader< ImageType > SeriesReaderType;
typedef itk::ImageFileWriter< ImageType > WriterType;
SeriesReaderType::Pointer seriesReader = SeriesReaderType::New( );
WriterType::Pointer writer = WriterType::New( );
```

我们用NumericSeriesFileNames类生成被读取的切片的文件名。接着我们在这个例子里重用这个对象以便生成被写的切片的文件名。

```
typedef itk::NumericSeriesFileNames NameGeneratorType;
NameGeneratorType::Pointer nameGenerator = NameGeneratorType::New( );
nameGenerator->SetStartIndex( first );
nameGenerator->SetEndIndex( last );
nameGenerator->SetIncrementIndex( 1 );
nameGenerator->SetSeriesFormat( "vwe%03d.png" );
```

执行读取程序的ImageIO对象现在连接到ImageSeriesReader：

```
seriesReader->SetImageIO( itk::PNGImageIO::New( ) );
```

输入切片的文件名由文件名生成器生成并传递给连续reader：

```
seriesReader->SetFileNames( nameGenerator->GetFileNames( ) );
```

测定体积的输出图像的名字传递给图像writer，我们现在连接连续reader的输出和测定体

积的writer的输入：

```
writer->SetFileName( outputFilename );
writer->SetInput( seriesReader->GetOutput( ) );
```

最后调用测定体积的writer的Updata( )触发管道的执行：

```
try
{
writer->Update( );
}
catch( itk::ExceptionObject & excp )
{
std::cerr << "Error reading the series " << std::endl;
std::cerr << excp << std::endl;
}
```

现在我们继续保存相同的测定体积的数据作为一个切片集。这个仅仅通过介绍保存一个体作为2D单独数据的连续集就可以做。连续writer的输入必须实例化，考虑输入文件是一个3D体和输出文件是2D图像。另外，连续reader的输出要连到连续writer的输入上。

```
typedef itk::Image< PixelType, 2 > Image2DType;
typedef itk::ImageSeriesWriter< ImageType, Image2DType > SeriesWriterType;
SeriesWriterType::Pointer seriesWriter = SeriesWriterType::New( );
seriesWriter->SetInput( seriesReader->GetOutput( ) );
```

我们现在重新使用文件名发生器以便产生输出的连续集的文件名列表。这种情况下，我们仅仅需要修改文件名发生器的格式。这时，我们传递输出文件名的列表给连续writer。

```
nameGenerator->SetSeriesFormat( "output%03d.png" );
seriesWriter->SetFileNames( nameGenerator->GetFileNames( ) );
```

最后，我们从一个try/catch模块触发连续writer的执行：

```
try
{
seriesWriter->Update( );
}
catch( itk::ExceptionObject & excp )
{
std::cerr << "Error reading the series " << std::endl;
std::cerr << excp << std::endl;
}
```

你也许注意到，在表示RGB图像的代码中，PixelType的声明中分离出的没有什么东西。所有要求支持颜色图像的活动在itk::ImageIO的对象中内部执行。

7.12 读、写 DICOM 图像

7.12.1 前言

20世纪70年代，在CT引入之后，数字诊断图像模式例如MRI和电脑在临床应用的增加，美国放射医学学会(ACR)和国家电子制造学会(NEMA)认为有设置一个标准的需要，以便为不同的厂商制造的装置之间传递信息和图像带来方便。

ACR和NEMA建立了一个联合委员会来制定在医学方面的数字图像和交换信息的标准(DICOM)。这个标准在发展过程中吸收另外的标准化组织例如CEN TC251，JIRA包括IEEE，HL7和ANSI USA作为观察员。

DICOM文件由一个头文件和一个图像数据体构成。头文件包括标准和自由形成域。标准化域的集合叫作公共DICOM词典，一个ITK例子在Insight/Utilities/gdcm/Dict/dicomV3.dic文件中。自由域的列表也叫作shadow dictionary。

一个简单的DICOM文件能够包含多框架，允许存储体和生理活动。图像数据可以用很多种标准压缩，包括JPEG、LZW和RLE。

DICOM标准是一个还在发展的标准， DICOM标准委员会仍在对其进行改进。增加的改进来自于DICOM委员会的成员组织。这些改进会在将来的版本中讲解。更新标准的一个要求就是要兼容以前的版本。

接下来的部分介绍了如何使用ITK提供的范函功能来读取和写入DICOM文件。这些在医学图像领域十分重要，因为大多数医学领域的图像存储和传递都用的是DICOM标准。

在ITK中，DICOM的范函性由GDCM库提供。这个开放的源码库由INSA-Lyon[26]的creatis团队发展。虽然最初这个库在一个LGPL许可下是开放式的，但是CREATIS对这个许可的限制理解得很清晰，而且同意采取更多的BSD许可，这个许可可以供ITK使用。它们许可的改变使得连同ITK描述GDCM成为可能。

GDCM仍然在原始的站点进行维护和改进。版本也用GDCM库的最主要版本更新。

7.12.2 读、写一幅 2D 图像

这部分的源码在文件Examples/IO/DicomImageReadWrite.cxx中。

这个例子介绍了如何读一个单独的DICOM切片以及把它写作另一个DICOM切片。在处理过程中要应用亮度变化。

为了读和写切片，我们这里使用itk::GDCMImageIO类，itk::GDCMImageIO类压缩了一个优先的GDCM库的连接。用这种方法我们就可以进行从ITK到GDCM提供的DICOM的范函性的存取。GDCMImageIO对象被作为itk::ImageFileWriter使用的ImageIO的对象连接。

我们首先包括以下头文件：

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkRescaleIntensityImageFilter.h"
```

```
#include "itkGDCMImageIO.h"
```

这时我们声明像素类型和图像维数，并用它们实例化被读的图像类型：

```
typedef signed short InputPixelType;
const unsigned int InputDimension = 2;
typedef itk::Image< InputPixelType, InputDimension > InputImageType;
```

用图像类型我们能够实例化reader的类型，创建它并设置被读的图像的文件名：

```
typedef itk::ImageFileReader< InputImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New( );
reader->SetFileName( argv[1] );
```

GDCMImageIO是一个读取和写入DICOM V3和ACR/NEMA图像的ImageIO类。在这里GDCMImageIO对象被创建并与ImageFileReader相连。

```
typedef itk::GDCMImageIO ImageIOType;
ImageIOType::Pointer gdcmImageIO = ImageIOType::New( );
reader->SetImageIO( gdcmImageIO );
```

这时我们调用Update( )来触发读取过程。因为这个读取过程会导致异常，我们放置一个问询在try/catch模块中。

```
try
{
reader->Update( );
}
catch (itk::ExceptionObject & e)
{
std::cerr << "exception in file reader " << std::endl;
std::cerr << e << std::endl;
return EXIT_FAILURE;
}
```

现在图像在内存中，通过GetOutput( )我们可以对它进行存取。在当前例子的维护中，焦点放在我们如何再一次将图像在新文件中保存成DICOM格式。

首先，我们必须实例化一个ImageFileWriter类型。这时，我们创建它并设置用于写的文件名，连接被写的输入图像。在这个例子中，我们用不同的方法写图像，每种情况下我们用不同的writer，我们列举writer对象的变量名及其类型。

```
typedef itk::ImageFileWriter< InputImageType > Writer1Type;
Writer1Type::Pointer writer1 = Writer1Type::New( );
writer1->SetFileName( argv[2] );
writer1->SetInput( reader->GetOutput( ) );
```

我们需要明确地设置对于writer滤波器的合适的图像IO，因为输入的DICOM是沿着写入过程被传递的。这个名称包含所有有效的DICOM文件应该包含的所有信息，像病人名字、病人ID、机构名等等。

```
writer1->SetImageIO( gdcmImageIO );
```

通过调用Update( )来触发写入程序。因为执行会导致异常情况出现，我们放Update( )问询在一个try/catch模块里。

```
try
{
writer1->Update( );
}
catch (itk::ExceptionObject & e)
{
std::cerr << "exception in file writer " << std::endl;
std::cerr << e << std::endl;
return EXIT_FAILURE;
}
```

现在我们使用重新调节亮度图像滤波器对图像进行重新调节。为了这个目的，我们使用一个更适合的像素类型：无符字符型代替有符短型。输出图像的最小值和最大值在缩放滤波器中明确定义。

```
typedef unsigned char WritePixelType;
typedef itk::Image< WritePixelType, 2 > WriteImageType;
typedef itk::RescaleIntensityImageFilter<
InputImageType, WriteImageType > RescaleFilterType;
RescaleFilterType::Pointer rescaler = RescaleFilterType::New( );
rescaler->SetOutputMinimum( 0 );
rescaler->SetOutputMaximum( 255 );
```

我们现在创建第二个writer对象，保存图像到一个文件中。这时不是DICOM格式。做这个只是为了校验图像，对照在这个例子里以后被保存成DICOM的格式。

```
typedef itk::ImageFileWriter< WriteImageType > Writer2Type;
Writer2Type::Pointer writer2 = Writer2Type::New( );
writer2->SetFileName( argv[3] );
rescaler->SetInput( reader->GetOutput( ) );
writer2->SetInput( rescaler->GetOutput( ) );
```

writer能够通过调用try/catch模块里的Update( )来执行。

我们现在保存同一个重新调节的图像到一个DICOM格式的文件中。为此我们仅仅需要设置itk::ImageFileWriter，并传递给它重新调节的图像作为输入。

```
typedef itk::ImageFileWriter< WriteImageType > Writer3Type;
Writer3Type::Pointer writer3 = Writer3Type::New( );
writer3->SetFileName( argv[4] );
writer3->SetInput( rescaler->GetOutput( ) );
```

我们现在需要明确地设置合适的图像IO(GDCMImageIO)，但是我们必须告诉ImageFileWriter不要从输入中用MetaDataDictionary而是从GDCMImageIO中用，因为IO包含DICOM的精确信息。

GDCMImageIO对象将会自动地探测像素类型，这种情况下无符字符串和GDCMImageIO对象将会更新DICOM头文件信息。

```
writer3->UseInputMetaDataDictionaryOff ( );
writer3->SetImageIO( gdcmImageIO );
```

最后我们调用一个try/catch模块的Update( )来触发DICOM writer的执行：

```
try
{
writer3->Update( );
}
catch (itk::ExceptionObject & e)
{
std::cerr << "Exception in file writer " << std::endl;
std::cerr << e << std::endl;
return EXIT_FAILURE;
}
```

7.12.3 读 2D DICOM 序列图像并写入体数据

这部分的头文件在文件Examples/IO/DicomSeriesReadImageWrite2.cxx中。

大概在临床应用中最普遍的数据表示是用DICOM切片集去组成三维空间图像。这种情况适用于CT、MRI和PET扫描仪。这对于图像分析处理测定体积的图像是很普遍的，测定体积的图像被存储在一个DICOM格式的文件集中。

接下来的例子介绍了如何使用ITK的范函性读取一个体的DICOM层级并保存这个体到另一种文件格式。

这个例子开始要包含合适的头文件。为了读取DICOM文件，我们需要创建itk::GDCMImageIO对象访问GDCM库的性能，生成文件名列表的itk::GDCMSeriesFileNames对象确定一个测定体积的数据的切片。

```
#include "itkGDCMImageIO.h"
#include "itkGDCMSeriesFileNames.h"
#include "itkImageSeriesReader.h"
#include "itkImageFileWriter.h"
```

我们定义像素类型和图像的维数。特殊情况下，图像的维度是3，我们假设一个无符号短型像素类型，因为它普遍用于X光CT扫描仪。

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

我们用图像类型实例化连续reader的类型并构建这种类型的一个对象：

```
typedef itk::ImageSeriesReader< ImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New( );
```

一个GDCMImageIO对象被创建并连接到reader上。这个对象是知道DICOM格式的内部结构的。

```
typedef itk::GDCMImageIO ImageIOType;
ImageIOType::Pointer dicomIO = ImageIOType::New( );
reader->SetImageIO( dicomIO );
```

现在我们要面对一个主要挑战——读取一个DICOM层级，也就是在一个给定的文件集目录中鉴别出属于同一测定的图像。幸运的是，GDCM提供了解决这个问题的功能，我们仅仅需要通过压缩了与GDCM类交换信息的ITK类调用这些功能。为方便起见，我们仅仅需要把DICOM切片存储的目录名传递给这个类。这些可以交给SetDirectory( )。GDCMSeriesFileNames将会搜索这个目录并生成DICOM文件名的序列。在这个例子中，我们也运行SetUseSeriesDetails(true)函数告诉GDCMSereiesFileNames对象用附加的DICOM信息去辨别目录中唯一的体。这很有用，例如，如果一个DICOM装置把一个相同的SeriesID给一个监测扫描和它的3D体；通过用附加的DICOM信息监测扫描将不会作为3D体的一部分。注意：SetUseSeriesDetails(true)必须预先运行SetDirectory( )。

```
typedef itk::GDCMSeriesFileNames NamesGeneratorType;
NamesGeneratorType::Pointer nameGenerator = NamesGeneratorType::New( );
nameGenerator->SetUseSeriesDetails( true );
nameGenerator->SetDirectory( argv[1] );
```

GDCMSeriesFileNames对象首先识别在被给目录中出现的DICOM层级的列表。我们收到一个字符串的集的参考目录的列表，这时我们可以处理类似于输出所有发生器建立的层级标识符之类的事情。因为寻找标识符可能导致异常，放这些代码在一个try/catch模块中比较明智。

```
typedef std::vector< std::string > SeriesIdContainer;
const SeriesIdContainer & seriesUID = nameGenerator->GetSeriesUIDs( );
SeriesIdContainer::const_iterator seriesItr = seriesUID.begin( );
SeriesIdContainer::const_iterator seriesEnd = seriesUID.end( );
while( seriesItr != seriesEnd )
{
std::cout << seriesItr->c_str( ) << std::endl;
seriesItr++;
}
```

在相同的目录中寻找多线成DICOM层级是很普遍的，我们必须告诉GDCM类我们想去读什么样的层级。在这个例子里，我们首先通过检查确定用户是否已经在代码中提供了一个层级标识符。如果没有层级标识符传递，我们就简单地在目录的搜索里建立层级。

```
std::string seriesIdentifier;
if( argc > 3 ) // If no optional series identifier
{
seriesIdentifier = argv[3];
}
```

```
else
{
seriesIdentifier = seriesUID.begin( )->c_str( );
}
```

我们传递层级标识符给名字发生器并询问所有与层级对应的文件名。这个列表会通过GetFileNames( )返回。

```
typedef std::vector< std::string > FileNamesContainer;
FileNamesContainer fileNames;
fileNames = nameGenerator->GetFileNames( seriesIdentifier );
```

现在可以用SetFileNames( )将文件名的列表传递给itk::ImageSeriesReader：

```
reader->SetFileNames( fileNames );
```

最后我们调用reader的Update( )函数触发读取程序。通常这个程序放在一个try/catch模块中。

```
try
{
reader->Update( );
}
catch (itk::ExceptionObject &ex)
{
std::cout << ex << std::endl;
return EXIT_FAILURE;
}
```

我们在内存中有一幅测定体积的图像，可以调用reader的GetOutput( )进行存取。

由于ImageIO工厂机构，我们现在保存测定体积的图像到另一个由用户指定的文件中，仅仅需要通过文件扩展名识别文件格式。

```
typedef itk::ImageFileWriter< ImageType > WriterType;
WriterType::Pointer writer = WriterType::New( );
writer->SetFileName( argv[2] );
writer->SetInput( reader->GetOutput( ) );
```

写入图像的程序通过调用writer的Update( )来触发：

```
writer->Update( );
```

注意：写测定体积的图像到一个文件中，我们将用它任何3D处理管道的输入。记住：DICOM是一个简单的文件格式和一个网络协议。一旦图像数据下载到内存中，它就能表示任何你从别的文件格式下载的体数据。

7.12.4 读 2D DICOM 序列图像并写 2D DICOM 序列图像

这部分的源码在文件Examples/IO/DicomSeriesReadSeriesWrite.cxx中。

这个例子介绍了如何读一个DICOM序列图像成一个体，并用相同的头文件信息把这个

体保存成另一个DICOM序列图像。它利用了GDCM库。

本例的主要目的是介绍如何沿着通道合适地传递DICOM的信息并能从输入的DICOM文件的信息正确地写回图像中。

请注意：写DICOM文件是一件非常精密的操作，因为我们正在处理重要而且大量的病人详细而精确的数据。确保你处理包含病人的私人信息的数据集时病人的隐私应被保护，这是你的责任。在美国，隐私问题通过HIPAA标准管制。

当用DICOM格式保存数据时，必须清楚数据是否已经被处理过，如果被处理过，你应该通知数据的接收者处理的目的和可能的结果。如果数据被打算用来诊断、治疗或监测病人，这是基本原则。例如，从一个16位像素到8位像素的变化，也许会发现某种病理，并且使病人暴露在危险之中，而且还有可能延误病人的治疗时间。

一定要熟悉“To Err is Human”，这种美国医学学会创建的医学过失报告。懂得医学过失发生的频率很高是减少它发生的第一步。

在这些警告之后，我们回到代码中熟悉ITK和GDCM在写DICOM层级文件中的使用。第一步是我们包含合适的头文件。我们要包含GDCM图像IO类、GDCM文件名发生器、层级文件reader和writer。

```
#include "itkGDCMImageIO.h"
#include "itkGDCMSeriesFileNames.h"
#include "itkImageSeriesReader.h"
#include "itkImageSeriesWriter.h"
```

第二步，我们在例子中用到图像类型。这需要明确选择像素类型和维度。用图像类型我们能定义序列图像reader的类型。

```
typedef signed short PixelType;
const unsigned int Dimension = 3;
typedef itk::Image< PixelType, Dimension > ImageType;
typedef itk::ImageSeriesReader< ImageType > ReaderType;
```

我们也要声明实际读、写DICOM图像的itk::GDCMImageIO对象的类型。itk::GDCMSeriesFileNames对象将生成并将构成所有体数据的切片的文件名进行排序。一旦我们有了这些类型，我们就可以创建这两个对象的例子。

```
typedef itk::GDCMImageIO ImageIOType;
typedef itk::GDCMSeriesFileNames NamesGeneratorType;
ImageIOType::Pointer gdcmIO = ImageIOType::New( );
NamesGeneratorType::Pointer namesGenerator = NamesGeneratorType::New( );
```

就像以前的例子，我们从目录中得到DICOM文件名。然而，我们这里用SetInputDirectory( )代替SetDirectory( )，这是因为现在我们要用文件名发生器生成被读的文件名和被写的文件名。这时，我们调用GetInputFileNames( )得到被读的文件名的列表。

```
namesGenerator->SetInputDirectory( argv[1] );
const ReaderType::FileNamesContainer & filenames =
namesGenerator->GetInputFileNames( );
```

我们构建一个序列图像reader对象的例子。设置DICOM图像IO对象和被读的文件名的列

表。

```
ReaderType::Pointer reader = ReaderType::New( );
reader->SetImageIO( gdcmIO );
reader->SetFileNames( filenames );
```

我们用序列图像reader的Update( )触发读取程序。放一个问询在一个try/catch模块中比较明智，因为运行中可能导致异常出现。

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

这时我们将体积数据下载到内存中，可以通过调用GetOutput( )来对它进行存取。

现在我们能比较写数据的过程。首先，我们从代码中得到输出目录的名字：

```
const char * outputDirectory = argv[2];
```

接着，我们确保输出目录存在，可以使用交叉平台工具itksys::SystemTools。如果目录还不存在的话，我们就选择创建目录。

```
itksys::SystemTools::MakeDirectory( outputDirectory );
```

我们例示被用作写的图像类型，并用图像类型实例化序列图像writer的类型。

```
typedef signed short OutputPixelType;
const unsigned int OutputDimension = 2;
typedef itk::Image< OutputPixelType, OutputDimension > Image2DType;
typedef itk::ImageSeriesWriter<
ImageType, Image2DType > SeriesWriterType;
```

我们创建一个序列图像writer并从reader的输出连接writer的输入。这时我们传递GDCM图像IO对象以便能用DICOM格式写这个图像。

```
SeriesWriterType::Pointer seriesWriter = SeriesWriterType::New( );
seriesWriter->SetInput( reader->GetOutput( ) );
seriesWriter->SetImageIO( gdcmIO );
```

现在可以用另外的输出目录设置GDCMSeriesFileNames生成新的文件名。这时传递最新的生成文件给序列图像writer。

```
namesGenerator->SetOutputDirectory( outputDirectory );
seriesWriter->SetFileNames( namesGenerator->GetOutputFileNames( ) );
```

接下来的代码对于程序正常工作极其重要。

从输入reader读取MetaDataDictionary并传递给输出writer。这一步重要的原因是MetaDataDictionary包含了所有输入DICOM头文件。

```
seriesWriter->SetMetaDataDictionaryArray(reader->GetMetaDataDictionaryArray( ) );
```

最后，我们调用序列图像的Update( )触发写程序。在一个try/catch模块中放这个问询以防异常情况的发生。

```
try
{
seriesWriter->Update( );
}
catch( itk::ExceptionObject & excp )
```

```
{
std::cerr << "Exception thrown while writing the series " << std::endl;
std::cerr << excp << std::endl;
return EXIT_FAILURE;
}
```

时刻紧记应该避免由扫描仪在DICOM文件中生成表象。要清楚这些数据是一些算法执行的结果。这将会防止你的数据被偶然的用于扫描仪的数据。

7.12.5 从一幅切片中输出 DICOM 标签

这部分的源码在文件Examples/IO/DicomImageReadPrintTags.cxx中。

从一个DICOM文件的头文件能够访问整体是很重要的。这能用于检查密度或检验我们手里的数据的正确性。这个例子介绍了如何读一个DICOM文件并输出大多数DICOM头文件的信息。例子里被主要类调用的头文件在下面，它们包括图像文件reader、GDCM图像IO对象、Meta数据词典和Meta数据对象的登录元素。

```
#include "itkImageFileReader.h"
#include "itkGDCMImageIO.h"
#include "itkMetaDataDictionary.h"
#include "itkMetaDataObject.h"
```

一旦图像被读入内存，我们实例化用于存储图像的类型：

```
typedef signed short PixelType;
const unsigned int Dimension = 2;
typedef itk::Image< PixelType, Dimension > ImageType;
```

用图像类型作为模板参数，我们例示图像文件reader的类型并构建一个例子：

```
typedef itk::ImageFileReader< ImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New( );
```

GDCM图像IO类型被声明并用于构建一个图像IO对象：

```
typedef itk::GDCMImageIO ImageIOType;
ImageIOType::Pointer dicomIO = ImageIOType::New( );
```

我们将被读的图像的文件名传递给reader并将ImageIO对象与它相连。

```
reader->SetFileName( argv[1] );
reader->SetImageIO( dicomIO );
```

通过调用Update( )触发读取过程。这个调用将放在一个try/catch模块，因为运行时可能导致异常情况发生：

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

现在图像已经被读取，用GetMetaDataDictionary( )从ImageIO对象获得元数据字典文件：

```
typedef itk::MetaDataDictionary DictionaryType;
const DictionaryType & dictionary = dicomIO->GetMetaDataDictionary( );
```

因为我们只对能用字符串表达的DICOM标签感兴趣，我们声明一个适合于管理字符串

的MetaDataObject：

typedef itk::MetaDataObject< std::string > MetaDataStringType;

我们示例贯穿于MetaDataDictionary整体的迭代：

DictionaryType::ConstIterator itr = dictionary.Begin( );

DictionaryType::ConstIterator end = dictionary.End( );

对于每一个字典的入口，我们首先检查是否它的元素可以被修改成一个字符串，这里我们用dynamic cast：

while(itr != end)

{

itk::MetaDataObjectBase::Pointer entry = itr->second;

MetaDataStringType::Pointer entryvalue =

dynamic_cast<MetaDataStringType *>(entry.GetPointer( )) ;

对于这些能被修改的入口，我们查找它们的DICOM标签并把它们传递给GDCMImageIO类的GetLabelFromTag( )。这些函数检查DICOM标签并返回与我们在标签关键变量中提供的相关联的标签。如果标签找到了，将返回一个labelId变量。如果标签关键词没有找到，函数本身返回false。例如在标签关键词中的“0010-0010”在labelId中变成病人的名字。

if(entryvalue)

{

std::string tagkey = itr->first;

std::string labelId;

bool found = itk::GDCMImageIO::GetLabelFromTag(tagkey, labelId);

GetMetaDataObjectValue( )获得字典入口的实际值作为一个字符串：

std::string tagvalue = entryvalue->GetMetaDataObjectValue( );

这时，我们能通过连接DICOM名或者标签、数字标签和它的实际值指出一个入口：

if(found)

{

std::cout << "(" << tagkey << ") " << labelId;

std::cout << " = " << tagvalue.c_str( ) << std::endl;

}

最后我们关闭这个循环：

++itr;

}

读一个特别的标签也是可能的。入口的字符串能够用于询问MetaDataDictionary。

std::string entryId = "0010|0010";

DictionaryType::ConstIterator tagItr = dictionary.Find(entryId);

如果在字典中被发现，这时我们能够用dynamic cast修改它，并变成一个字符串入口。

if(tagItr != end)

{

MetaDataStringType::ConstPointer entryvalue =

dynamic_cast<const MetaDataStringType *>(tagItr->second.GetPointer( ));

如果动态构建成功，我们能够指出标签值和实际值。

```
if( entryvalue )
{
std::string tagvalue = entryvalue->GetMetaDataObjectValue( );
std::cout << "Patient's Name (" << entryId << ") ";
std::cout << " is: " << tagvalue << std::endl;
}
```

完整的DICOM字典的描述在文件Insight/Utilities/gdcm/Dicts/dicomV3.dic中。

7.12.6 从序列图像输出 DICOM 标签

这部分的源码在文件Examples/IO/DicomSeriesReadPrintTags.cxx中

本例介绍了如何读一个序列图像成为一个体并输出DICOM的头文件信息。二元域将会跳过。

序列图像reader的头文件信息和对于图像IO和命名生成的GDCM类应该首先被包含：

```
#include "itkImageSeriesReader.h"
#include "itkGDCMImageIO.h"
#include "itkGDCMSeriesFileNames.h"
```

一旦读入内存，我们例示用于存储图像的类型：

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

我们用图像类型例示序列图像reader类型并构建这个类的一个对象：

```
typedef itk::ImageSeriesReader< ImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New( );
```

创建一个GDCMImageIO对象并赋给reader：

```
typedef itk::GDCMImageIO ImageIOType;
ImageIOType::Pointer dicomIO = ImageIOType::New( );
reader->SetImageIO( dicomIO );
```

为了生成DICOM切片的名字，我们声明一个GDCMSeriesFileNames。我们用SetInputDirectory( )指定目录，这时从代码中读取目录名。你能从一个GIU的文件对话框获得目录名。

```
typedef itk::GDCMSeriesFileNames NamesGeneratorType;
NamesGeneratorType::Pointer nameGenerator = NamesGeneratorType::New( );
nameGenerator->SetInputDirectory( argv[1] );
```

通过调用GetInputFileNames( )从名字生成器获得被读的文件名的列表，并在一个字符串库中接收结果。用SetFileNames( )将文件名的列表传递给reader。

```
typedef std::vector<std::string> FileNamesContainer;
FileNamesContainer fileNames = nameGenerator->GetInputFileNames( );
```

```
reader->SetFileNames( fileNames );
```

我们调用Update( )触发reader。查询被放在一个try/catch模块中以防异常发生：

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

ITK内部查询GDCM并从头文件中获得所有的DICOM标签。标签值存储在itk::MetaDataDictionary中。通过调用GetMetaDataDictionary( )元数据字典能够从任何ImageIO类中恢复过来。

```
typedef itk::MetaDataDictionary DictionaryType;
const DictionaryType & dictionary = dicomIO->GetMetaDataDictionary( );
```

在这个例子中，我们仅仅对能用字符串表示的DICOM标签感兴趣。我们因此声明一个itk::MetaDataObject的字符串类型以便接受这些特殊值。

```
typedef itk::MetaDataObject< std::string > MetaDataStringType;
```

元数据词典被作为一个相应迭代的库被组织，因此我们能通过首先询问它的Begin( )和End( )访问所有的入口：

```
DictionaryType::ConstIterator itr = dictionary.Begin( );
DictionaryType::ConstIterator end = dictionary.End( );
```

现在我们准备浏览DICOM标签的列表。为此我们使用刚刚声明的迭代。在每一个入口我们用基于RTTI信息的dynamic_cast修改它成为一个字符串入口。字典被组织得像std::map结构，我们因此用每个入口的第一和第二成员询问{key,value}对。

```
while( itr != end )
{
itk::MetaDataObjectBase::Pointer entry = itr->second;
MetaDataStringType::Pointer entryvalue =
dynamic_cast<MetaDataStringType *>( entry.GetPointer( ) ) ;
if( entryvalue )
{
std::string tagkey = itr->first;
std::string tagvalue = entryvalue->GetMetaDataObjectValue( );
std::cout << tagkey << " = " << tagvalue << std::endl;
}
++itr;
}
```

也能查询指定的入口代替读取它们。在这里，用户必须用标准的DICOM标识符提供标签标识。标识存储在一段字符串中，被看成是字典中的关键字。

```
std::string entryId = "0010|0010";
DictionaryType::ConstIterator tagItr = dictionary.Find( entryId );
if( tagItr == end )
{
std::cerr << "Tag " << entryId;
std::cerr << " not found in the DICOM header" << std::endl;
}
```

因为入口可能是也有可能不是字符串类型，所以我们必须再一次使用dynamic_cast，把它转化成字符串类型。如果转化成功了，我们就能够指出它的内容。

```
MetaDataStringType::ConstPointer entryvalue =
dynamic_cast<const MetaDataStringType *>( tagItr->second.GetPointer( ) );
if( entryvalue )
{
std::string tagvalue = entryvalue->GetMetaDataObjectValue( );
std::cout << "Patient's Name (" << entryId << ") ";
std::cout << " is: " << tagvalue << std::endl;
}
```

这种范函类型通过一个绘图用户界面可能更加有用。更加详细的DICOM字典的描述参照以下文件：

Insight/Utilities/gdcm/Dicts/dicomV3.dic

7.12.7 改变 DICOM 头文件

这部分的源码在文件Examples/IO/DicomImageReadChangeHeaderWrite.cxx中。

这个例子介绍了如何读取一幅单独的DICOM切片和改变一些头文件信息后作为另一幅DICOM切片写回。头文件关键词/值能在代码中指定。关键词定义在以下文件中：

Insight/Utilities/gdcm/Dicts/dicomV3.dic

记住：修改DICOM头文件的内容是一件危险的操作。头文件包含病人的基本信息因此它的秘密性必须被保护。在修改DICOM的头文件之前，必须确信有这个必要，且必须保证信息改变不会导致较低的质量。

我们必须在开始时包含相关的头文件。这里我们包含图像reader、图像writer、图像、与数据词典和元数据对象以及GDCMImageIO的接口。元数据词典保存了DICOM图像文件被读到ITK图像中时DICOM头文件的所有接口：

```
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkImage.h"
#include "itkMetaDataDictionary.h"
#include "itkMetaDataObject.h"
#include "itkGDCMImageIO.h"
```

我们通过一个特别的像素类型和图像维度声明图像类型：

```
typedef signed short InputPixelType;
const unsigned int Dimension = 2;
typedef itk::Image< InputPixelType, Dimension > InputImageType;
```

我们通过作为模板参数的图像类型来实例化reader的类型。Reader 的一个例子被创建、被读取的文件名从代码中得到：

```
typedef itk::ImageFileReader< InputImageType > ReaderType;
ReaderType::Pointer reader = ReaderType::New( );
```

```
reader->SetFileName( argv[1] );
```

为了提供读、写文件的服务，我们创建GDCMImageIO对象。新创建的图像IO类与reader相连：

```
typedef itk::GDCMImageIO ImageIOType;
ImageIOType::Pointer gdcmImageIO = ImageIOType::New( );
reader->SetImageIO( gdcmImageIO );
```

调用reader的Update( )触发图像的读取：

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

我们从reader下载到内存中的图像读取元数据词典：

```
InputImageType::Pointer inputImage = reader->GetOutput( );
typedef itk::MetaDataDictionary DictionaryType;
DictionaryType & dictionary = inputImage->GetMetaDataDictionary( );
```

现在我们访问元数据词典中的入口，并且对于一个关键值我们赋一个新的内容给入口。我们可以通过从代码中读取{关键词，值}来做。相关的函数是EncapsulateMetaData，它读取词典并通过entryId提供一个关键词，用变量值的内容取代当前值。这会在代码中不断重复。

```
for (int i = 3; i < argc; i+=2)
{
std::string entryId( argv[i] );
std::string value( argv[i+1] );
itk::EncapsulateMetaData<std::string>( dictionary, entryId, value );
}
```

现在词典已经更新，我们保存图像。输出的图像已经有与它的DICOM头文件相关的数据。

用图像类型，我们实例化一个writer类型并创建一个writer。在reader和writer之间的短通道将会被创建。从代码中读取被写的文件名。图像IO对象与writer相连。

```
typedef itk::ImageFileWriter< InputImageType > Writer1Type;
Writer1Type::Pointer writer1 = Writer1Type::New( );
writer1->SetInput( reader->GetOutput( ) );
writer1->SetFileName( argv[2] );
writer1->SetImageIO( gdcmImageIO );
```

调用Update( )函数触发writer的执行：

```
writer1->Update( );
```

再次记住，修改DICOM文件的开始入口包含一些对病人的危险，因此做的时候必须非常谨慎。