国外计算机科学教材系列

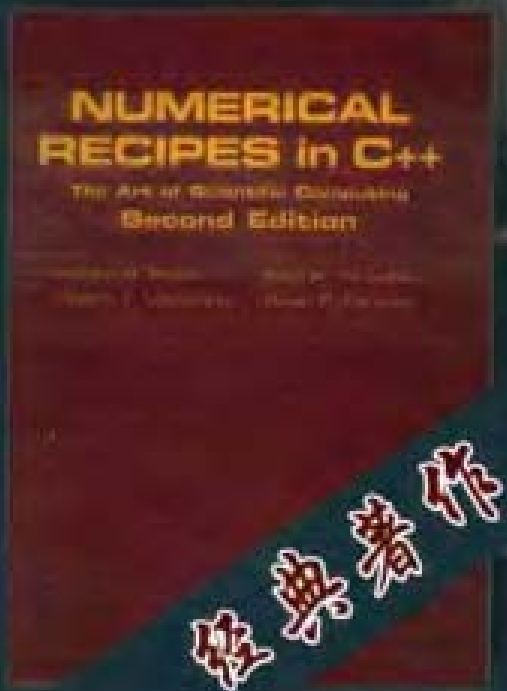
C++数值算法

（第二版）

Numerical Recipes in C++

The Art of Scientific Computing

Second Edition



经典著作

[美] William H. Press　Saul A. Teukolsky
William T. Vetterling　Brian P. Flannery

胡健伟　赵志勇　薛运华　等译

電子工業出版社
Publishing House of Electronics Industry
http://www.phei.com.cn

经典教材

C++数值算法

（第二版）

Numerical Recipes in C++

The Art of Scientific Computing, Second Edition

本书由美国洛斯阿拉莫斯国家实验室（Los Alamos National Laboratory）主任William H. Press和其他三位从事科学计算的学者合著。本书及其姊妹篇（C版、FORTRAN版以及Pascal版）已被美国哈佛大学、美国康奈尔大学、英国剑桥大学等国际著名大学选为本科生和研究生数值计算课程的教材。

本书主要特点：

- 选材内容丰富。包含了当代科学计算过程中涉及的大量内容：求特殊函数值、随机数、排序、最优化、快速傅里叶变换、谱分析、小波变换、统计描述和数据建模、偏微分方程数值解、若干编码算法和任意精度计算等。
- 科学性和实用性统一。不仅对每种算法进行了数学分析和比较，而且根据作者经验对算法给出了评论和建议，并在此基础上提供了用C++语言编写的实用程序。

ISBN 7-5053-8713-8

9 787505 387133 >

华信教育资源网 www.hsedu.com.cn

责任编辑：冯小贝
特约编辑：杜　萌
封面设计：毛惠庚

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

ISBN 7-5053-8713-8　定价：68.00元

国外计算机科学教材系列

C++ 数值算法

（第二版）

Numerical Recipes in C++

The Art of Scientific Computing

Second Edition

[美] William H. Press　Saul A. Teukolsky　William T. Vetterling　Brian P. Flannery 著

胡健伟　赵志勇　薛运华　等译

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内容简介

本书选材内容丰富，除了通常数值方法课程的内容外，还包含当代科学计算大量用到的专题，如求特殊函数值、随机数、排序、最优化、快速傅里叶变换、谱分析、小波变换、统计描述和数据建模、常微分方程和偏微分方程数值解、若干编码算法和任意精度的计算等。

本书科学性和实用性统一。每个专题中，不仅对每种算法给出了数学分析和比较，而且根据作者的经验对算法做出了评论和建议，并在此基础上给出了用C++语言编写的实用程序。读者可以很方便地直接套用这些程序，还可以结合特定的需要进行修改。本书中包含的345个程序构成了C++语言的数值计算程序库。

本书可以作为大学本科生和研究生的教材或参考书，也可以作为从事科学计算的科技工作者的工具书、计算机软件开发者的参考书。

Authorized translation from the English language edition published by The Syndicate of the Press of the University of Cambridge, England. Copyright © Cambridge University Press 2002.
All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.
This edition is licensed for distribution and sale in the People's Republic of China only excluding Hong Kong, Taiwan and Macau and may not be distributed and sold elsewhere.
Simplified Chinese language edition published by Publishing House of Electronics Industry. Copyright © 2005.

本书中文简体专有翻译出版权由Cambridge University Press授予电子工业出版社。其原文版权及中文翻译出版权受法律保护。未经许可，不得以任何形式或手段复制或抄袭本书内容。
本书中文简体字版仅限于在中华人民共和国境内（不包括香港、澳门特别行政区以及台湾地区）发行与销售，并不得在其他地区发行与销售。

版权贸易合同登记号　图字：01-2003-1231

图书在版编目（CIP）数据

C++数值算法（第二版）/（美）普雷斯（Press, W. H.）等著. 胡健伟等译.
-北京：电子工业出版社，2005.1
（国外计算机科学教材系列）
书名原文：Numerical Recipes in C++: The Art of Scientific Computing, Second Edition
ISBN 7-5053-8713-8

Ⅰ.C…　Ⅱ.①普…　②胡…　Ⅲ.C语言-程序设计-数值计算-教材　Ⅳ.TP312

中国版本图书馆CIP数据核字（2004）第118717号

责任编辑：冯小贝　　特约编辑：杜　萌
印　　刷：北京智力达印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编：100036
经　　销：各地新华书店
开　　本：787 × 1092　1/16　　印张：46.5　　字数：1190千字
印　　次：2005年1月第1次印刷
定　　价：68.00元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换；若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

出 版 说 明

21世纪初的5至10年是我国国民经济和社会发展的重要时期，也是信息产业快速发展的关键时期。在我国加入WTO后的今天，培养一支适应国际化竞争的一流IT人才队伍是我国高等教育的重要任务之一。信息科学和技术方面人才的优劣与多寡，是我国面对国际竞争时成败的关键因素。

当前，正值我国高等教育特别是信息科学领域的教育调整、变革的重大时期，为使我国教育体制与国际化接轨，有条件的高等院校正在为某些信息学科和技术课程使用国外优秀教材和优秀原版教材，以使我国在计算机教学上尽快赶上国际先进水平。

电子工业出版社秉承多年来引进国外优秀图书的经验，翻译出版了“国外计算机科学教材系列”丛书，这套教材覆盖学科范围广、领域宽、层次多，既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求，广大师生可自由选择和自由组合使用。这些教材涉及的学科方向包括网络与通信、操作系统、计算机组织与结构、算法与数据结构、数据库与信息处理、编程语言、图形图像与多媒体、软件工程等。同时，我们也适当引进了一些优秀英文原版教材，本着翻译版本和英文原版并重的原则，对重点图书既提供英文原版又提供相应的翻译版本。

在图书选题上，我们大都选择国外著名出版公司出版的高校教材，如Pearson Education培生教育出版集团、麦格劳－希尔教育出版集团、麻省理工学院出版社、剑桥大学出版社等。撰写教材的许多作者都是蜚声世界的教授、学者，如道格拉斯·科默（Douglas E. Comer）、威廉·斯托林斯（William Stallings）、哈维·戴特尔（Harvey M. Deitel）、尤利斯·布莱克（Uyless Black）等。

为确保教材的选题质量和翻译质量，我们约请了清华大学、北京大学、北京航空航天大学、复旦大学、上海交通大学、南京大学、浙江大学、哈尔滨工业大学、华中科技大学、西安交通大学、国防科学技术大学、解放军理工大学等著名高校的教授和骨干教师参与了本系列教材的选题、翻译和审校工作。他们中既有讲授同类教材的骨干教师、博士，也有积累了几十年教学经验的老教授和博士生导师。

在该系列教材的选题、翻译和编辑加工过程中，为提高教材质量，我们做了大量细致的工作，包括对所选教材进行全面论证；选择编辑时力求达到专业对口；对排版、印制质量进行严格把关。对于英文教材中出现的错误，我们通过与作者联络和网上下载勘误表等方式，逐一进行了修订。

此外，我们还将与国外著名出版公司合作，提供一些教材的教学支持资料，希望能为授课老师提供帮助。今后，我们将继续加强与各高校教师的密切联系，为广大师生引进更多的国外优秀教材和参考书，为我国计算机科学教学体系与国际教学体系的接轨做出努力。

电子工业出版社

教材出版委员会

主　任	杨芙清	北京大学教授 中国科学院院士 北京大学信息与工程学部主任 北京大学软件工程研究所所长
委　员	王　珊	中国人民大学信息学院院长、教授
	胡道元	清华大学计算机科学与技术系教授 国际信息处理联合会通信系统中国代表
	钟玉琢	清华大学计算机科学与技术系教授 中国计算机学会多媒体专业委员会主任
	谢希仁	中国人民解放军理工大学教授 全军网络技术研究中心主任、博士生导师
	尤晋元	上海交通大学计算机科学与工程系教授 上海分布计算技术中心主任
	施伯乐	上海国际数据库研究中心主任、复旦大学教授 中国计算机学会常务理事、上海市计算机学会理事长
	邹　鹏	国防科学技术大学计算机学院教授、博士生导师 教育部计算机基础课程教学指导委员会副主任委员
	张昆藏	青岛大学信息工程学院教授

译 者 序

本书由美国洛斯阿拉莫斯国家实验室(Los Alamos National Laboratory)副主任、美国哈佛大学天文学与物理学教授 William H. Press、美国康奈尔大学物理学与天体物理学教授 Saul A. Teukolsky、宝丽来有限公司(Polaroid Corporation)的 William T. Vetterling 和埃克森研究和工程公司(EXXON Research and Engineering Company)的 Brian P. Flannery 四人合著,是这四位作者合著的《Fortran 数值算法》(第二版)、《C 数值算法》(第二版)的孪生版。本系列图书的第一版还包括《Pascal 数值算法》、《BASIC 数值算法》,它们在国外广为发行,颇受读者欢迎。第二版继承了前一版的优点,并做了很大修改,增加了约 100 多种程序,使本书更为完善,内容更加丰富。

本书所涉及的内容较广,除一般数值分析课程中包含的专题外,还包括特殊函数求值、随机数、排序、最优化、FFT 和谱分析、小波变换、统计描述和建模、偏微分方程、以及格雷码和算术码等内容。对于每个专题内容,作者给出了数学分析,论述了各种算法并进行了分析和比较,并根据作者的实际工作经验,对一些算法提出了明确的判断和建议,在此基础上给出了 ANSI/ISO C++ 语言编写的实用程序。根据使用者的爱好,它们可以用于几乎所有现存的 C++ 向量/矩阵类库,以及任何与 ANSI/ISO 兼容的编译器。因此,本书中的程序不但可作为黑箱使用,而且读者将学会如何打开这些黑箱,对其进行分析和研究,然后结合读者特定的任务进行修改,以完成读者的特殊需要。本书的最大特点就是将理论与实际应用结合起来。

本书是美国哈佛大学、美国康奈尔大学、英国剑桥大学等一些著名大学高年级本科生和研究生的数值计算课程的教材。本书给出的 C++ 语言程序构成了完整的 C++ 语言数值算法程序库,所以,这也是一本不可多得的有关数值算法的 C++ 语言程序手册。

参加本书翻译的有胡健伟、赵志勇、薛运华、马岩和孙氡。胡健伟校阅了译文的全稿。考虑到本书的读者都有一定的外语基础,为便于他们查阅外文文献,故本书中部分含有外国学者人名定义的术语,我们都保留其拉丁字母拼写的名字,不译为汉字发音。程序中的注释,也只翻译最主要的部分,其他的小注释保留原文。

本书涉及内容十分广泛,由于译者水平所限,虽经校阅,仍难免有错误和不妥之处,恳请读者批评指正。

C++版序言

C++已经逐渐成为计算机程序中的主要语言，甚至在科学和工程的应用中取代了C和Fortran的地位。本书是《数值算法》第二版的完整文本，包含了用C++书写的全部程序。

对数值计算而言，C++有其特殊的问题。主要困难在于处理向量和矩阵，使之具有高效率并且使程序具有高水平的结构。至今仍然未有为实现此目的而被人们接受的通用标准库，这就使得问题更为困难。在第1章和附录中，我们将介绍我们是如何解决这一问题的。对读者而言，一种默认的选择是，使用由我们提供的非常简单的类库，只需花费几分钟安装和运行。我们也将说明如何使用其他的矩阵/向量类库。这在刚开始时会耗费一些时间，但此后会使读者清晰地访问本书的程序而且完美地发挥其效能。

借此机会，提醒我们的C版本读者，将所有的数组和矩阵换为“零基”。在本书中，我们对读者提出的正文和程序中的错误进行了修改，在此深表感谢。

2001年9月
William H. Press
Saul A. Teukolsky
William T. Vetterling
Brian P. Flannery

第二版序言

最初撰写《数值算法》[1]第一版的意图是，提供一本汇总一般性讨论、解析数学、算法论述以及实际可运行的程序等多方面内容的书籍。本书第一版取得了很大的成功，但随着时间的推移，有很多新内容值得补充，所以我们决定出版第二版。

现在掌握的素材比六年前增多了。六年前，根据当时的文献和我们自己的研究工作，我们只能从经验上估计哪一些数值计算的技巧是非常重要和最有生命力的。现在，我们已从大量读者的直接反馈中得益，我们的管理机构 Numercial Recipes Software 每年收到的信件达上千封。这些信件指出，我们忽略了某些特殊的技术，这些技术在科学与工程等特殊领域内具有众所周知的重要性。我们仔细阅读了这些信件，尤其是那些指出特定参考文献的信件。

这些新内容的加入，不可避免地增加了本书的篇幅。事实上，文字内容和程序都增加了约 50%（现在包含的程序超过 300 个）。“请勿增加篇幅”，这是几位明智的同事给我们的忠告。我们遵循了这些忠告，但还是增加了某些主要专题的原理性讨论。某些可选专题以“深入讨论”小节出现，读者阅读时忽略这些章节不会影响阅读的连贯性。

第二版中新增的主要内容有：

- 新增一章关于积分方程和反演方法的内容
- 椭圆型偏微分方程的多重网格求解法的详细讨论
- 带状对角型线性系统的程序
- 对线性代数稀疏矩阵的改进程序
- Cholesky 和 QR 分解
- 任意加权函数的正交多项式和高斯求积法
- 数值求导的计算方法
- 帕德逼近和有理切比雪夫逼近
- 贝塞尔函数、修正的贝塞尔函数、分数阶的贝塞尔函数以及其他新增的特殊函数
- 改进的随机生成程序
- 拟随机序列
- 高维空间中蒙特卡罗积分的自适应和递归程序
- 非线性方程组的全局收敛法
- 连续控制空间的模拟退火极小化
- 二维和三维实数据的快速傅里叶变换(FFT)
- 外部存储的快速傅里叶变换(FFT)
- 改进的快速余弦变换程序
- 小波变换
- 有上限、下限的傅里叶积分
- 非均匀取样数据的谱分析
- Savitzky-Golay 平滑滤波器
- 两个坐标数据都有误差的直线拟合
- 二维 Kolmogorov-Smirnoff 检验
- 统计靴带法
- 微分方程的嵌入式 Runge-Kutta-Fehlberg 法
- 刚性微分方程组的高阶法
- 新增一章关于“非典型的数值算法”的内容，包括霍夫曼码和算术编码、任意精度的计算以及几个其他专题

对于本书所涉及的基础内容，可以参阅第一版的前言或本书目录。

---

① 本书是《数值算法》系列丛书之一。——译者注

致谢

在此不可能逐个列出所有提出过有益建议的读者名单，但我们衷心地感谢他们。在正文中，我们尽量给出一些想法的原始出处，但对可能发生的遗漏表示歉意。

一些读者和同事特别慷慨地为本书提供了他们的想法、评注、建议和程序。我们特别要感谢 George Rybicki，Philip Pinto，Peter Lepage，Robert Lupton，Douglas Eardley，Ramesh Narayan，David Spergel，Alan Oppenheim，Sallie Baliunas，Scott tremaine，Glennys Farrar，Steven Block，John Peacock，Thomas Loredo，Matthew Choptuik，Gregory Cook，L. Samuel Finn，P. Deuflhard，Harold Lewis，Peter Weinberger，David Syer，Richad Ferch，Steven Ebstein，Bradley Keister 和 William Gould。Nancy Lee Snyder 帮助我们完成了复杂的 TEX 文稿。我们还要向本书的编辑 Lauren Cowles 和 Alan Harvey 、责任编辑 Russell Hahn 致谢。

我们尤其感谢本书的程序顾问 Seth Finkelstein，他编写、重写或影响了本书(以及本书的 Fortran 语言孪生版和本书伴侣版 Example Book)中的许多程序。本版之所以比其先前版本有更好的 C 类语言程序风格，应主要归功于他(当然，我们接受 Fortran 版本仍然存在错误的批评)。

我们在装有 UNIX 操作系统的 DEC 和 Sun 工作站和装有 MS－DOS 5.0 / Windows 3.0 的 486/33 PC 兼容机上运行过本书的程序(进行程序测试所用的其他计算机请参见 1.0 节所列的计算机)。我们强烈推荐以下一些软件：GUN，Emacs，TEX，Perl，Adobe Illustrator 和 PostScript。我们也曾用过 C 编译器的其他一些版本，因为太多，在此无法一一致谢(它们有时亦有太多的缺陷)。应该清醒地指出，我们的标准试验套件(运行本书中的所有程序)在许多检验过的编译中有未被克服的编译缺陷。如果可能，我们愿和开发商共同探索修复这些缺陷。

本书前两位作者感谢美国科学基金会长期资助他们研究计算方法。还感谢 D.A.R.P.A. 对本书 13.10 节中小波研究的资助。

1992 年 6 月

William H. Press

Saul A. Teukolsky

William F. Vetterling

Brian P. Flannery

第一版序言

我们称本书为《数值算法》(Numerical Recipes)有几方面的理由。从某种意义上讲,本书确实是一本数值计算方面的"食谱大全"[1]。当然,食谱和菜单有着重大区别。菜单是为让就餐者能够对餐馆提供的各道菜进行选择而设计的,每道菜中的不同配料已被混合并改变了原先的面貌。而食谱则揭示了各种配料,并讲解如何进行备料和制作。

本书取名为《数值算法》的另一目的是意味着它是展现技术、兼收并蓄的"合剂"[2]。本书是独具特色的,对所介绍的每一个专题都提供了一般性讨论、解析数学、算法论述,并(更为重要的)以可执行的计算机程序的形式提供了这些思想的实际实施。我们的任务是寻求每一个专题内容之间恰当的平衡。读者会感受到,对于某些专题我们尤为侧重于解析方面,而对其他一些数学的先决条件已普遍成立的专题,更侧重于深入地讨论其计算算法的性质,以及实施的实际问题。

本书一半内容适用于作为理工科专业本科高年级学生的数值计算教程,另一半内容适用于研究生水平的课程并作为专业研究人员的参考文献。这样安排内容的一个突出特点是,随着经验的积累,读者能够以日趋成熟的水平来使用本书。甚至对于那些缺乏经验的读者,也应该能够将本书中的大多数高级程序作为一个黑箱使用,但希望使用后能回过头来重新学习其内部的奥秘所在。

如果说本书存在一个统一的起支配作用的主旋律,那就是数值计算的实用算法应该是有效的、巧妙的,同时也是清晰的。但是另一种观点认为,有效的计算方法应该是复杂的,以至于只能作为"黑箱"的形式使用,这种观点我们是断然反对的。

因而在本书中,我们的目的是将大量的计算方法的程序黑箱打开,供读者仔细研究。我们希望教会读者怎样打开这些黑箱,然后再将它们组装起来,以便加以修改而适应读者自己的特殊需求。本书假设读者具有一定的数学基础,即具备相当于物理科学、工程科学、经济学,或一定的人文科学方面大学本科文化程度的基本数学基础。同时,假定读者知道怎样进行计算机编程,但并不假定读者在数值分析或数值算法方面具有任何预备知识。

通常,"数值算法"所涉及的范围应该是"直至偏微分方程但不包括偏微分方程的一切内容"。我们荣幸地在这方面有所突破:首先,本书有一章关于偏微分方程算法的简介(第 19 章)。其次,本书显然未能包含其他"一切"内容。数值分析课程中所有的"标准"专题都已包含在本书中:线性方程组(第 2 章)、内插法和外推法(第 3 章)、积分(第 4 章)、非线性求根(第 9 章)、特征系统(第 11 章)以及常微分方程组(第 16 章)。这些专题的大部分内容由于引进了某些高级素材而超出了它们的标准范畴,而这些素材特别重要且有用。

本书还详细叙述了某些其他专题,通常这些专题无法在标准的数值分析教材中找到,这些内容包括:函数的求值和高等数学中特定的特殊函数求值(第 5 章和第 6 章);随机数和蒙特卡罗法(第 7 章);排序(第 8 章);最优化,包括多维方法(第 10 章);傅里叶变换方法,包括 FFT 算法和其他谱分析法(第 12 章和第 13 章);统计描述和数据建模(第 14 章和第 15 章),另外还有两点边界值问题、打靶法和松弛法(第 17 章)。

本书的程序都用 ANSI 标准 C 语言编写。本书还有 FORTRAN, Pascal 和 BASIC 语言版本。在 1.1 节中还将更多地阐明关于 C 语言的编程问题,以及关于我们的程序所假定的计算环境等。

致谢

许多同事慷慨地与我们分享了他们在数值和计算方面的经验,诸如提供计算程序、对本书手稿的评注等。我们要特别感谢 George Rybicki, Douglas Eardley, Philip Marcus, Stuart Shapiro, Paul Horowitz, Bruce Musicus, Ir-

---

① Recipes 可表示食谱。——译者注

② Recipes 可表示药剂的处方。——译者注

win Shapiro, Stephen Wolfram, Henry Abarbanel, Larry Smarr, Richad Muller, John Bahcall 和 A.G.W. Cameron

我们还要感谢两位从未谋面的学者：一位是 Forman Acton，他于 1970 年出版的教科书《实用数值方法》[Numerical Methods that Work (New York: Harper and Row)]的风格深深地影响了我们；另一位是 Donald Knuth，既因为他的系列著作《计算机程序设计艺术》[The Art of Computer Programming (Reading, MA: Addison-Wesley)]，又由于他的计算机排版语言 TEX，后者极大地帮助了本书的出版。

作者在计算方法领域的研究工作部分得到了美国科学基金会的资助。

1985 年 10 月

William H. Press

Brian P. Flannery

Saul A. Teukolsky

William T. Vetterling

许可证信息

如果读者计划在自己的计算机上使用本书中的程序，应当阅读下面的内容。首先需要阅读以下的免责条款，然后把程序安装到计算机上并取得 Numerical Recipes 软件许可证（如果没有该许可证或者下文所述的免费“直接许可证”，本书只能作为教学和参考用书，仅供阅读）。

免责条款

我们无需做出任何明确的或含蓄的承诺，即本书中所包含的程序毫无差错，或者这些程序符合商业销售的特定标准，或者它们能满足读者特定应用的要求。这些程序不可用于求解以下问题：其不正确解答可能导致对某人的伤害或财产的损失。如果你以这种方式使用程序，后果自负。作者和出版者概不承担由你使用程序造成的直接或间接损失的责任。

如何在计算机上安装代码

选择以下方法之一即可进行：

- 可以从本书直接键入程序到你的计算机中。此时，只有“直接许可证”是有效的。你不可以转送或发布机器可读的副本给其他任何人。如果选择这种方式，我们不希望从用户那里得到任何关于程序缺陷的报告，因为经验表明，几乎所有此类缺陷报告皆源于程序键入错误。
- 可以从 Numerical Recipes On-Line Software Store 下载本书程序的电子版本，网址是 http://www.nr.com。所有文件（本书的程序和演示程序）被置于一个压缩文件中。需要购买下载和解压缩的许可证。单屏用户许可证可以从 On-Line Store 即时购买（对多屏用户可能有优惠），价格取决于所使用的操作系统（Windows 或 Macintosh 及 Linux 或 UNIX）以及是否属于教育机构。如果希望获得比较通用的（网站或企业）许可证，也可以先购买一个单屏许可证，单屏许可证的付费将在以后的升级许可证费用中扣除。
- 可以从剑桥大学出版社（Cambridge University Press）购买含有程序的介质。有适用于 Windows 和 Macintosh 系统的 ISO-9660 格式的 CD-ROM 版本，其中包含了完整的 C++ 软件，以及原先的 C 版本。这种 CD-ROM 提供单屏幕 Windows 和 Macintosh 许可证，订购书号为 ISBN 0-521-750377。有更加扩展的适用于 Windows，Macintosh 和 UNIX/Linux 系统的 ISO-9660 格式的 CD-ROM 版本，在单张 CD-ROM 上包含 C，C++ 和 FORTRAN 版本（以及第一版的 Pascal 和 BASIC 版本）。这些 CD-ROM 中，有 Windows 和 Macintosh 的单屏用户许可证（订购书号 ISBN 0-521-750350）或（稍贵一些的）UNIX/Linux 工作站单屏用户许可证（订购书号 ISBN 0-521-750369）。可以向剑桥大学出版社订购这些介质，电子邮件地址为 orders@cup.org（北美地区）或者 directcustserv@cambridge.org（其他地区），或者访问剑桥大学出版社的网站 http://www.cambridge.org。

许可证的类型

以下是我们提供的许可证的类型。需要指出的是，其中一些是自动取得的：可以从剑桥大学出版社直接购买介质，或者从 Numerical Recipes On-Line Software Store 购买解密密码。其他类型的许可证要求用户按照类别与 Numerical Recipes Software 联系（电子邮件地址为 orders@nr.com，传真号码为 781 863-1739）。网站 http://www.nr.com 上提供了更多的信息。

- [“直接许可证”] 如果你是个人购买本书，并且将其中的一个或多个程序键入到你的计算机中，我们同意你在该计算机上使用它们。不可以转让或者发布机器可读的副本给其他任何人，或在一台以上机器上使用这些程序，或者发布包含我们的程序的可执行程序。这是惟一的免费许可证。
- [“单屏用户许可证”] 这是最常见的低价格许可证，由单屏用户许可证[Single Screen (Shrinkwrap) License]文档上的条款所限定（完整的条款可以通过我们的网站查询）。一般来说，该许可证允许你在任何一个屏幕上（PC，工作站，X 终端等）使用本书的程序。在此许可证下，也可以将采纳了我们的程

序的预先编译过的可执行程序传送到其他无许可证的计算机上，其前提条件是(i)用于非商业用途，(ii)在有许可证的屏幕上首次开发、编译、成功地运行程序，(iii)我们的程序被捆绑到一个程序中，但它们不能作为单个程序被访问，或不能在实际中提取并用于其他程序。也就是说，在这类许可证之下，程序用户不能使用我们的程序作为程序库或工作平台的一部分。其他类型的商业或非商业分发的条件可以在网站 http://www.nr.com 上找到。

- [“多屏、服务器、网站和公司许可证”] 单屏用户的条款可以扩展到指定的机器群，它们由屏幕数目、位置或所有者定义。当估计的屏幕数目超过 40 时，单屏的价格有显著降低。请与 Numerical Recipes Software 联系以了解细节(电子邮件地址为 orders@nr.com，传真号为 781 863-1739)。
- [“课程复制权许可证”] 如果被认定的教育机构中的教师采用本书授课，并已购买一个单屏许可证，可以允许在课程中使用程序，只要将姓名、职称、地址、课程名称、编号、日期和估计注册听课人数，以及每名学生 5 美元的预付款(估算数)邮寄到 Numerical Recipes Software，地址为 P.O.Box 380243，Cambridge，MA 02238-0243(USA)。教师将收到一份邮寄的许可证，它准许教师将程序复制以供学生使用，和/或准许将程序传送到一台可供学生访问的计算机上(但仅供在授课期间使用)。

关于计算机程序的版权

和艺术或文学作品一样，计算机程序亦受版权保护。一般而言，如果将受版权保护的程序不经许可复制到你的计算机，则属于侵权行为(这也是一种不友好的行为，因为它剥夺了程序作者创造性劳动应得的报酬)。根据版权法，所有“派生工作”(修订版本或转换成其他计算机语言)也受版权保护。

版权并不保护思想，只保护那些思想的特定表现形式。对计算机程序而言，这些思想是程序的方法论和算法，包括程序员采用的一系列必要步骤。这些思想的表达形式是程序源代码(特别是任何其中所嵌入的随意或独特的风格部分)、它们导出的目标代码，以及任何其他派生工作。

如果你对程序中所包含的思想进行分析，然后用你自己的完全不同的实现方式表达这些思想，那么这种新的程序实现属于你个人。本书中的程序也是这样形成的，它们不全都是我们自己的成果。当本书中的程序称为“基于”某已发布的受版权保护的程序时，是指它们的思想是相同的。这些思想的源代码表达方式是属于我们的。我们相信，本书中的任何材料都未侵犯现有的版权。

注册商标

本书正文中出现了几个注册商标：Sun，Solaris，Ultra 和 WorkShop 是 Sun Microsystems，Inc. 的注册商标。Microsoft，Windows 和 Visual C++ 是 Microsoft Corporation 的注册商标。DEC 和 VMS 是 Compaq Computer Corporation 的注册商标。IBM，AIX 和 RS/6000 是 International Business Machines Corporation 的注册商标。Linux 是 Linus Torvals 的注册商标。Apple 和 Macintosh 是 Apple Computer，Inc. 的注册商标。Borland C++ Builder 是 Inprise Corporation 的注册商标。UNIX 是 The Open Group 的注册商标。IMSL 是 Visual Numerics，Inc. 的注册商标。NAG 是 Numerical Algorithms Group (USA) Inc. 的专属计算机软件。PostScript 和 Adobe Illustrator 是 Adobe Systems Incorporated 的注册商标。最后，毫无疑问，Numerical Recipes NR 和 nr.com 是 Numerical Recipes Software 的注册商标。

归属

从法律上说，思想“如同空气一样自由”，但这一事实不能取代以下道德规范：思想属于它们的已知创始人。对本书中那些基于已知来源的程序，无论它们受版权保护还是属于公共领域、正式出版还是手抄本，我们都尽量给出它们的正确归属。遗憾的是，许多公开流传的程序的来源并不清楚。非常感谢读者提供的关于归属方面的信息，无论是新的还是更正过的，我们都会在本书再版时尽力体现它们。

目　　录

第 1 章　绪论 …… 1
1.0　引言 …… 1
1.1　程序组织和控制结构 …… 4
1.2　科学计算的 C++ 约定 …… 11
1.3　向量和矩阵类的实施 …… 17
1.4　误差、准确性和稳定性 …… 21

第 2 章　线性代数方程组求解 …… 24
2.0　引言 …… 24
2.1　Gauss-Jordan 消去法 …… 26
2.2　具有回代过程的高斯消去法 …… 30
2.3　LU 分解法及其应用 …… 31
2.4　三对角及带状对角系统方程 …… 37
2.5　线性方程组解的迭代改进 …… 40
2.6　奇异值分解 …… 43
2.7　稀疏线性方程组 …… 51
2.8　Vandermonde 矩阵和 Toeplitz 矩阵 …… 66
2.9　深入讨论:Cholesky 分解 …… 71
2.10　深入讨论:QR 分解 …… 73
2.11　矩阵求逆是否是 N^3 阶运算 …… 76

第 3 章　内插法和外推法 …… 79
3.0　引言 …… 79
3.1　多项式内插法和外推法 …… 81
3.2　有理函数内插法和外推法 …… 83
3.3　三次样条插值 …… 85
3.4　搜索有序表的方法 …… 87
3.5　插值多项式的系数 …… 89
3.6　二维或高维插值 …… 91

第 4 章　函数积分 …… 97
4.0　引言 …… 97
4.1　坐标等距划分的经典公式 …… 98
4.2　基本算法 …… 101
4.3　龙贝格积分 …… 104
4.4　广义积分 …… 105
4.5　高斯求积法与正交多项式 …… 110
4.6　多维积分 …… 121

第 5 章　函数求值 …… 124
5.0　引言 …… 124
5.1　级数与其收敛性 …… 124
5.2　连分式求值 …… 126

5.3 多项式和有理函数 …… 129
5.4 复数运算 …… 132
5.5 递推关系及 Clenshaw 递推公式 …… 133
5.6 二次方程和三次方程 …… 136
5.7 数值求导 …… 137
5.8 切比雪夫逼近 …… 140
5.9 切比雪夫逼近函数的微分和积分 …… 143
5.10 切比雪夫系数的多项式逼近 …… 145
5.11 深入讨论:幂级数的化简 …… 146
5.12 深入讨论:帕德逼近 …… 147
5.13 深入讨论:有理切比雪夫逼近 …… 150
5.14 线积分求函数值 …… 153

第 6 章 特殊函数 …… 156
6.0 引言 …… 156
6.1 Γ 函数、B 函数、阶乘、二项式系数 …… 156
6.2 不完全 Γ 函数、误差函数、χ^2 概率函数、累积泊松函数 …… 159
6.3 指数积分 …… 163
6.4 不完全 B 函数、学生分布、F 分布、累积二项式分布 …… 166
6.5 整数阶贝塞尔函数 …… 169
6.6 修正的整数阶贝塞尔函数 …… 174
6.7 深入讨论:分数阶贝塞尔函数、艾里函数、球面贝塞尔函数 …… 178
6.8 球面调和函数 …… 188
6.9 Fresnel 积分、余弦和正弦积分 …… 190
6.10 Dawson 积分 …… 193
6.11 椭圆积分和雅可比椭圆函数 …… 194
6.12 超几何函数 …… 202

第 7 章 随机数 …… 205
7.0 引言 …… 205
7.1 一致偏离 …… 205
7.2 变换方法:指数偏离和正态偏离 …… 214
7.3 拒绝方法:伽马偏离、泊松偏离、二项偏离 …… 216
7.4 随机位的生成 …… 221
7.5 深入讨论:基于数据加密的随机序列 …… 224
7.6 简单的蒙特卡罗基分 …… 228
7.7 准随机序列 …… 231
7.8 深入讨论:自适应及递归蒙特卡罗方法 …… 236

第 8 章 排序 …… 246
8.0 引言 …… 246
8.1 直接插入法和 Shell 方法 …… 247
8.2 快速排序法 …… 248
8.3 堆排序法 …… 251
8.4 索引和分秩 …… 252
8.5 挑选第 M 大的元素 …… 255
8.6 深入讨论:等价类的确定 …… 258

第 9 章　求根与非线性方程组 …… 260
9.0　引言 …… 260
9.1　划界与二分 …… 262
9.2　弦截法、试位法和Ridders方法 …… 265
9.3　Van Wijngaarden-Dekker-Brent 方法 …… 269
9.4　利用导数的 Newton-Raphson 方法 …… 271
9.5　多项式的根 …… 275
9.6　非线性方程组的 Newton-Raphson 方法 …… 283
9.7　非线性方程组的全局收敛法 …… 285

第 10 章　函数的极值 …… 295
10.0　引言 …… 295
10.1　一维黄金分割搜索 …… 297
10.2　抛物线内插和一维 Brent 方法 …… 300
10.3　使用一阶导数的一维搜索方法 …… 303
10.4　多维下降单纯形法 …… 305
10.5　多维情况下的方向集(Powell)方法 …… 308
10.6　多维共轭梯度法 …… 313
10.7　多维变尺度法 …… 317
10.8　线性规划和单纯形法 …… 321
10.9　模拟退火法 …… 331

第 11 章　特征系统 …… 341
11.0　引言 …… 341
11.1　对称矩阵的雅可比变换 …… 345
11.2　将对称矩阵约化为三对角形式:Givens 约化和 Householder 约化 …… 350
11.3　三对角矩阵的特征值和特征向量 …… 354
11.4　埃尔米特矩阵 …… 358
11.5　将一般矩阵化为 Hessenberg 形式 …… 359
11.6　实 Hessenberg 矩阵的 QR 算法 …… 362
11.7　用逆迭代法改进特征值并求解特征向量 …… 367

第 12 章　快速傅里叶变换 …… 370
12.0　引言 …… 370
12.1　离散样本数据的傅里叶变换 …… 373
12.2　快速傅里叶变换(FFT) …… 375
12.3　实函数的 FFT、正弦变换和余弦变换 …… 380
12.4　二维或多维的 FFT …… 388
12.5　二维和三维实数据的傅里叶变换 …… 391
12.6　深入讨论:外部存储和局部内存的 FFT …… 395

第 13 章　傅里叶和谱的应用 …… 400
13.0　引言 …… 400
13.1　使用 FFT 做卷积和解卷积 …… 400
13.2　使用 FFT 做相关和自相关 …… 405
13.3　具有 FFT 的最优(维纳)滤波 …… 406
13.4　使用 FFT 做功率谱估计 …… 408
13.5　深入讨论:时域中的数字滤波 …… 414

13.6 线性预测和线性预测编码 …… 419
13.7 深入讨论:用最大熵(全极)方法做功率谱估计 …… 424
13.8 深入讨论:非均匀取样数据的谱分析 …… 427
13.9 深入讨论:使用 FFT 计算傅里叶积分 …… 434
13.10 小波变换 …… 439
13.11 深入讨论:取样定理的数值应用 …… 450

第 14 章 数据的统计描述 …… 452
14.0 引言 …… 452
14.1 分布的矩:均值、方差、偏斜度等 …… 453
14.2 两种分布是否具有相同的均值和方差 …… 456
14.3 两种分布是否不同 …… 459
14.4 两种分布的列联表分析 …… 465
14.5 线性相关 …… 470
14.6 非参数相关或秩相关 …… 472
14.7 深入讨论:二维分布 …… 477
14.8 深入讨论:Savitzky-Golay 平滑滤波器 …… 480

第 15 章 数据建模 …… 485
15.0 引言 …… 485
15.1 最大似然估计的最小二乘方法 …… 485
15.2 拟合数据成直线 …… 488
15.3 深入讨论:两个坐标数据都有误差的直线拟合 …… 491
15.4 一般的线性最小二乘方 …… 495
15.5 非线性模型 …… 503
15.6 被估模型参数的置信界限 …… 508
15.7 稳健估计 …… 515

第 16 章 常微分方程组的积分 …… 521
16.0 引言 …… 521
16.1 Runge-Kutta 方法 …… 523
16.2 Runge-Kutta 方法的自适应步长控制 …… 526
16.3 修正中点法 …… 531
16.4 Richardson 外推法和 Bulirsch-Stoer 方法 …… 533
16.5 深入讨论:二阶守恒方程组 …… 539
16.6 刚性方程组 …… 540
16.7 多步法、多值法和预测-校正法 …… 550

第 17 章 两点边值问题 …… 554
17.0 引言 …… 554
17.1 打靶法 …… 556
17.2 对拟合点打靶 …… 558
17.3 深入讨论:松弛法 …… 559
17.4 实例:球体调和函数 …… 568
17.5 深入讨论:网格点的自动分配 …… 575
17.6 深入讨论:内部边界条件或奇异点的处理 …… 577

第 18 章 积分方程和反演理论 …… 580
18.0 引言 …… 580

18.1 第二类 Fredholm 方程 …… 582
18.2 Volterra 方程 …… 584
18.3 深入讨论:具有奇异核的积分方程 …… 586
18.4 反演问题与先验信息的利用 …… 591
18.5 线性正则化方法 …… 594
18.6 Backus-Gilbert 方法 …… 599
18.7 最大熵图像恢复 …… 601

第 19 章 偏微分方程 …… 607
19.0 引言 …… 607
19.1 通量守恒的初值问题 …… 612
19.2 扩散初值问题 …… 620
19.3 多维初值问题 …… 624
19.4 边值问题的傅里叶方法和循环约简法 …… 627
19.5 边值问题的松弛法 …… 631
19.6 边值问题的多重网格法 …… 636

第 20 章 非典型的数值算法 …… 650
20.0 引言 …… 650
20.1 诊断机器的参数 …… 650
20.2 格雷码 …… 654
20.3 循环冗余度校验和其他的校验和式 …… 655
20.4 霍夫曼编码与数据压缩 …… 660
20.5 算术编码 …… 664
20.6 任意精度的运算 …… 668

附录 A 函数声明表 …… 677

附录 B 实用例程和类 …… 685

附录 C 转换为单精度 …… 700

参考文献 …… 702

程序从属表 …… 706

各章节的计算机程序 …… 719

第1章 绪 论

1.0 引言

本书和它的(用其他计算机语言的)姐妹版本一样,旨在向读者教授实用、有效且尽可能完美的数值计算方法。假定读者要进行一些特定的任务,本书将介绍如何来完成这些任务。

全书将明确地指明应该怎样做和不应该怎样做,这全部是实践经验的总结。这并不是说我们的建议绝对都正确。这样做是因为,我们反对一种倾向,即在关于数值计算的教科书和文献中,只讨论已经发现的各种可能方法,而不涉及其相对价值的实际评价。因此,我们将尽可能把实际的判断提供给读者。当读者有了一定经验后,将会对这些建议的可信度进行判断。

假定读者能够阅读 C++ 语言计算机程序,它是本书的语言。如果更乐意用其他语言的程序,可参考"Numerical Recipes in Fortran 77"、"Numerical Recipes in Fortran 90"和"Numerical Recipes in C",以及"Numerical Recipes in Pascal"和"Numerical Recipes Routines and Examples in BASIC"的早期版本,只是不包含第二版增加的内容。如果 BASIC 和 Pascal 是所选择的语言,则这些版本(的第一版)可以提供完善的服务。

列入正文中的程序都采用如下格式:

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::flmoon(const int n, const int nph, int &jd, DP &frac)
程序总以一个引导说明开始,概括程序的目的并解释它们调用的序列。本程序用于计算月相。给出一个整数 n 和所期
望的月相的代码 nph(nph = 0 为新月,nph = 1 为上弦月,nph = 2 为满月,nph = 3 为下弦月),程序返回从 1900 年 1 月起第
n 个此类月相发生时的儒略历日数 jd,并加上一天的小数部分 frac。假定采用格林尼治平均时。
{
    const DP RAD=3.141592653589793238/180.0;
    int i;
    DP am,as,c,t,t2,xtra;

    c=n+nph/4.0;                                  这是对单行进行注解的方法。
    t=c/1236.85;
    t2=t*t;
    as=359.2242+29.105356*c;                      不必理解这个算法,但它确实很有效。
    am=306.0253+385.816918*c+0.010730*t2;
    jd=2415020+28*n+7*nph;
    xtra=0.75933+1.53058868*c+((1.178e-4)-(1.55e-7)*t)*t2;
    if (nph == 0 || nph == 2)
        xtra += (0.1734-3.93e-4*t)*sin(RAD*as)-0.4068*sin(RAD*am);
    else if (nph == 1 || nph == 3)
        xtra += (0.1721-4.0e-4*t)*sin(RAD*as)-0.6280*sin(RAD*am);
    else nrerror("nph is unknown in flmoon");        此处为指示错误条件的方法。
    i=int(xtra >= 0.0 ? floor(xtra) : ceil(xtra-1.0));
    jd += i;
    frac=xtra-i;
}
```

注意,我们使用一条类似于 nrerror("some error message");的语句表示错误和异常情况。函数 nrerror()是列于附录 B 中的一个小实用程序 nrutil.h 的一部分。附录 B 还包含了其他实用程序,将在后面介绍。函数 nrerror()在用户的 stderr 设备(通常是用户的屏幕终端)上显示错误信息,然后调用函数 exit()终止运行。可以修改 nrerror(),以使程序除暂停运行外还可做其他一些事情。例如,可使程序暂停而从键盘输入数据,然后人为地中断运行。在一些应用中可能想修改 nrerror(),以进行一些更复杂的错误处理,例如通过抛出 C++ 异常向其他位置传输控制。

我们将在 1.1 节和 1.2 节中更深入地讨论 C++ 程序语言的约定和风格。

1.0.1 快速开始:使用C++数值算法程序

如果希望进行跳跃式阅读,可以参阅本节内容。我们将计算自1900年1月以来前20个满月的儒略历日数的平均值和方差。

首先,定位附录A和附录B中所列的一些重要文件nrtypes.h, nrutil.h和nr.h。其中含有程序所用的各种类型的定义、使用的向量/矩阵类、各种实用函数以及所有本书算法函数的函数声明(事实上,nrtypes.h默认地包含文件nrtypes_nr.h、nrutil.h默认地包含文件nrutil_nr.h。这种设置使用户能很容易地改变默认值,将在1.3节讨论)。

其次,建立以下主程序文件:

```
#include <iostream>
#include <iomanip>
#include "nr.h"
using namespace std;

int main(void)
{
    const int NTOT=20;
    int i,jd,nph=2;
    DP frac,ave,vrnce;
    Vec_DP data(NTOT);

    for (i=0;i<NTOT;i++) {
        NR::flmoon(i,nph,jd,frac);
        data[i]=jd;
    }
    NR::avevar(data,ave,vrnce);
    cout << "Average = " << setw(12) << ave;
    cout << " Variance = " << setw(13) << vrnce << endl;
    return 0;
}
```

下面简要解释上述程序的某些成分。

必须总要有包含函数声明的#include nr.h语句,还包含文件nrtypes.h和nrutil.h。声明Vec_DP data(NTOT)通过元素NTOT建立了一个双精度向量。类型DP(双精度,即double)和Vec_DP在nrtypes.h中定义,但是向量类在nrutil.h中定义。辖域分解前缀NR::可以访问算法函数flmoon和avevar,它们位于名字空间NR,后者在nr.h中定义。调用flmoon直到循环求得20个满月,并存储其儒略历的日数于data中。然后,调用程序avevar并显示答案。

再次,编译主程序文件及文件flmoon.cpp和avevar.cpp。连接所得到的目标文件。

最后,运行得到的可执行文件。典型的输出是

```
Average = 2.41532e+06 Variance = 30480.7
```

文件nrtypes.h、nrutil.h、nr.h及它们的概念将在1.2节和1.3节中详细讨论。

1.0.2 计算环境和程序有效性

我们的目的是,尽可能使本书程序的通用性强一些,以便能在不同型号的计算机、不同的操作系统、不同的C++编译系统上运用。C++实际上就是为了可移植性而设计的。但是我们发现,只有在多种不同的编译器上实际检查了所有的程序后,才能了解库结构和内容方面的差异,甚至会发现所允许的语法也有区别。作为多数硬件和软件配置的代表,我们已在不同的机型、操作系统和编译器的各类组合下(见表1.0.1)检验了本书的所有程序。更一般地说,各程序可以不加修改地在任何符合ANSI/ISO C++标准的编译器上运行。标准ANSI/ISO C++在Stroustrup所著的书中叙述得相当出色[1]。

表 1.0.1 经过检验的机器和编译器

O/S 和硬件	编译器版本
Microsoft Windows/Intel	Microsoft Visual C++6.0
Microsoft Windows/Intel	Borland C++5.02
Linux/Intel	GNU C++("g++")2.95.2
AIX 4.3/IBM RS/6000	KAI C++("KCC")4.0d
SunOS 5.7(Solaris 7)/Sun Ultra 5	Sun Workshop Compiler C++("CC")5.0
SunOS 5.7 (Solaris 7)/Sun Ultra 5	GNU C++("g++")2.95.2

为保证程序的有效性,我们直接将机器可读的程序作为书稿中的程序源代码,以减少印刷错误的产生。所用的驱动程序或者演示程序可从"Numerical Recipes Example Book (C++)"一书中得到,也可以从该书机器可读的版本得到。如果读者计划使用本书中较多的程序,将发现机器可读的软件版本是非常有用的,它既有算法的程序,也有演示程序的副本。

当然,程序中的错误是在所难免的。若发现了错误,请记录它并告诉我们。可以在网站 www.nr.com 查找联系信息。

1.0.3 和本书第一版的兼容性

如果已习惯于本书第一版的数值算法程序,那么请记住:第一版中的几乎所有程序都列入了第二版内,并具有相同的名字和功能,通常只是对程序代码本身做了一些改进。另外,新版中还增加了100多个程序。

本书放弃了第一版中的一小部分程序,并用我们认为更好的程序进行了替代。在表1.0.2中列出了放弃的程序以及它们的替代程序。

表 1.0.2 本书中未采用的第一版中的程序

原程序名	替代程序	注释
adi	mglin 或 mgfas	更好的方法
cosft	cosft1 或 cosft2	选择边界条件
ce1,ce12	rf,rd,rj,rc	更好的算法
des,desks	ran4 现用 psdes	原先太慢
jindexx	indexx	indexx 对 double 和 int 重载
mdian1,mdian2	select,selip	更通用
qcksrt	sort	名字改变(sort 现在是 hpsort)
rkqc	rkqs	更好的方法
smooft	从 savgol 得到系数使用 convlv	
sparse	linbcg	更通用

1.0.4 关于参考书

在本书的大多数小节后列出了参考文献,以便读者进一步研究。在正文中引用的参考文献用括号中的数字表示,如[2]。它代表的是其所在小节后面的"参考文献和深入阅读"中的序号,列在相应参考书的后面。

因为计算机的算法通常在发表之前已经流传了一段时间,因此列出"原始文献"的任务比较艰巨。本书虽然没有列出原始文献,但有些主题存在相当多的第二类文献(教材、评论中的讨论),所以本书有意识地列出了这类主题的参考文献,但仅限于少数的、非常有用的第二类文献,特别是参考价值较大的文献。当有些主题没有适当的第二类参考文献时,本书给出少数几个原始来源,它们可作为进一步阅读的起点,而不作为该领域的完整文献目录。

所列参考文献的顺序并不表示任何意义。它只反映了按引用顺序列出参考文献和大致按优先级的先后列出进一步研究的参考建议,一般来说,最有参考价值的列在最前面。

在本章的下面四节中,将复习程序设计的一些基本概念(控制结构等),讨论一套在本书中采用的针对C++的约定,并介绍一些数值分析的基本概念(舍入误差等)。

参考文献和深入阅读

Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley). [1]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [2]

1.1 程序组织和控制结构

计算机程序与赋诗和作曲之间有极其相似之处,这三者都是作为可视媒体出现在二维纸面或计算机屏幕上的符号。但在这三者中,可视的二维**静态**形式传送(或假设传送)着一些极不相同的东西,即**随时间展开**的过程不同。诗歌,供阅读;乐谱,供演奏;程序作为有顺序的计算机指令序列,供执行。

在这三者中,就传播的视觉形式而言,其目标是人。目的是最高效率地传送,进而使人最大程度地事先了解过程将如何随时间展开。就诗歌而言,其目标是读者;就乐谱而言,其目标是演奏者;而程序,其目标是程序用户。

读者可能会觉得,程序的传送目标不是人而是计算机,程序用户只是作为不相干的中介物——一个给机器供料的侍者。这也许反映了下面的情形:一个未受过专业训练的用户将 CD-ROM 插入计算机中,然后以二进制可执行形式执行一个黑箱程序。在这种情况下,计算机并不管这个程序是否是"优秀编程策略"的程序。

然而,我们假设读者处于一个完全不同的情形下。不仅需要知道一个程序在做**什么**,而且知道它是**如何**做的。这样就能够根据自己的特定用途调整和修改它。需要其他人能了解你所做的一切,进而提出建议。在这些情况下,目的是得到**可维护**或者**可再用**的代码,程序传送的目标完全是人而非机器。

要制定优秀的编程策略,关键是认识到程序、音乐和诗歌三者作为人脑的符号结构都是自然地组成层次的,并且有许多不同的嵌套层。语音(音素)形成小的有意义的单元(词素),它又组成单词;单词组合为短语,进而连成句子;句子形成段落,再构造出更高而有意义的层次。音符组成音乐短语,进而组合成主旋律、配合旋律及和声等,再形成乐章、协奏曲、交响乐等。

程序的结构同样是有层次的。恰当地说,优秀编程策略带来不同的技巧,从而产生不同的编程水平[1~3]。程序中最低层是 ascii 字符集;然后是常量、标识符、操作数、运算符,接着是像 a[j+1] = b+c/3.0;这样的程序语句。建议程序要简单明了。如果想将 j=(0,1,2) 的值循环地排列为对应的 k=(1,2,0),可能会写出如下的单行语句:

```
k=(2-j)*(1+3*j)/2;
```

但是,以后想重新理解它时可能有些困难。更好且可能是更快的程序应为

```
k=j+1;
if (k == 3) k=0;
```

某些编程专家可能会编写如下所示的程序:

```
switch (j) {
    case 0: k=1; break;
    case 1: k=2; break;
    case 2: k=0; break;
    default: {
        cerr << "unexpected value for j";
        exit(1);
    }
}
```

因为此程序很清楚,而且能够有效地预防非期望的 j 值所引起的错误。在这些程序中,我们一般倾向于中间的那个程序。

在这个简单的示例中,实际上经历了几个层次。语句频繁地以"组"或"块"的形式出现,仅当它们作为一个整体时才能表示一定的意义。例如

```
swap=a[j];
a[j]=b[j];
b[j]=swap;
```

任何程序员都会立刻领悟到,它是在交换两个变量,而

```
ans=sum=0.0;
n=1;
```

很像是在某些迭代过程前对变量进行初始赋值。程序中的这一层通常是很明显的。优秀的编程策略是,在这一层放入注释,即加"初始赋值"或"交换变量"的说明。

下一层就是**控制结构**,这主要包括如上例中的 switch 结构、for 循环等。这一层非常重要,并且和本书所列程序的层次有关。将在后面详细讨论。

更高的层次有函数、类和方法,以及待完成计算任务的"全局"结构。和音乐类比,现在是处在乐章和完整的作品这一层。在这一层,**模块化**和**封装**成为重要的编程概念,一般可理解为,各个程序单元应该通过明确的定义和严密限定的接口相互连在一起。好的模块化策略是大型复杂软件工程成功的先决条件,特别对多个编程者共同协作完成的程序更是如此。即使所遇到的不是那么庞大的编程工作,这种模块化策略也是有益的(尽管不是特别必要)。

一些计算机语言,像 C++ 和 Modula-2,能够实现高层次语言结构的模块化。例如,在 C++ 和 Modula-2 中,函数、类型定义和数据结构都能封装成"类"或者"模块",通过声明的公有接口进行通信,而它们的内部工作和程序的其余部分分离开并隐藏起来[4,5]。在 C++ 语言中,重要的概念是"类"(class),也就是用户定义的数据类型,它提供数据的隐藏、数据自动初始赋值、内存管理、动态打印和运算符的重载(即,用户可以对运算符,如 + 和 * 进行定义延伸,以使得在任何特定的类都能进行运算)[5]。如果类能正确地用来定义在不同的程序单元之间传递的数据结构,则它可使程序单元的公用接口更清晰、更确定,减少编程出错的机会,也显著减少编译时间和查找运行错误的时间。

除了模块化,还有**面向对象的程序设计**的概念,此概念依赖于模块化。这时,一些编程语言如 C++,允许模块的公有接口接受类型或活动的重定义,这些重定义在本模块这一级都能被共享(因此被称为多态性)。例如,通过重载复数数据类型和定义相应的算术运算,一个用来对实数矩阵求逆的程序能够在运行时用来对复数矩阵求逆。还有两个概念也会遇到,即**继承性**(除了自己本身的结构功能外,还能继承其他类型结构进而重定义数据类型的能力)和**对象扩展性**(在运行时,不通过源代码就直接对模块增加功能的能力)。

在本书中,没有对程序模块化、也没有对象化代码,其原因在于,我们预想读者能花一些时间,用自己选择的结构,将书中的算法模块化或对象化。目前,还没有标准的或可接受的面向科学对象计算的一些类的集合。然而,当我们不可避免地将本书的算法内容(这是本书存在的理由)和某些与类定义有关的特定集合联系起来时,我们可能正在创建这种面向科学对象计算的一个集合。

另一方面,我们不是不赞成模块化和面向对象的程序设计。我们已经尽可能使程序结构是"对象友好的"。还有,在实现各个函数时,已经特别注意了**结构化的程序设计**。

1.1.1 控制结构

执行程序随时间而展开,但却不是严格地按照程序语句的顺序线性展开的。影响语句执行顺序或影响语句是否执行的程序语句称为**控制语句**。控制语句本身不表示任何意义,它们仅在所控制的语句组或语句块中才表达一定的意义。如果假定这些语句块是含有句子的一段文章,那么最好认为控制语句是这段文章的段落空格和句子间的标点,而不是句子中的词汇。

结构化程序的设计目的是,使控制程序看起来清晰明了。可见,这个目的与计算机如何读程序没有关系。和前面所讲的一样,计算机并不理会程序员是否采用了结构化程序设计,而阅读程序的人却非常关心程序结构化。一旦发现调试结构良好的程序比结构模糊的程序容易得多的时候,自然会关心程序结构化。

可以用两种互补的方法来实现结构化程序设计的目的。第一,熟悉少数基本的、在编程中经常要用到的控制结构,这些控制结构在大多数编程语言中都有方便的表示形式。除了学习这些标准控制结构以外,应该在尽可能的范围内学会考虑自己的编程任务。在编写程序时,应该学会习惯于用一致的、惯用的方式来表示这些标准控制结构。

有人会认为这阻碍了创造性,但关键是在格式上受到适当限制更有利于进行交流。其次,尽可能避免使用很难一眼就看出控制块或控制对象的控制语句。这意味着在实际应用中,必须尽可能避免使用语句标号

和 goto 语句，goto 语句并没有太大危险(虽然它确实干扰了人们阅读程序)，问题在于语句标号。事实上，任何时候当在读程序时，碰上一个语句标号，都会直接产生以下疑问：指向这一标号的控制来自哪一分支？它可以来自程序中的任何位置。在什么情况下会导致这一分支转向标号？可能会在任何情况下。这样使确定的事情变得不确定，使对程序的理解复杂化。

下面编排了一些例子，使这些考虑更为具体(见图 1.1.1)。

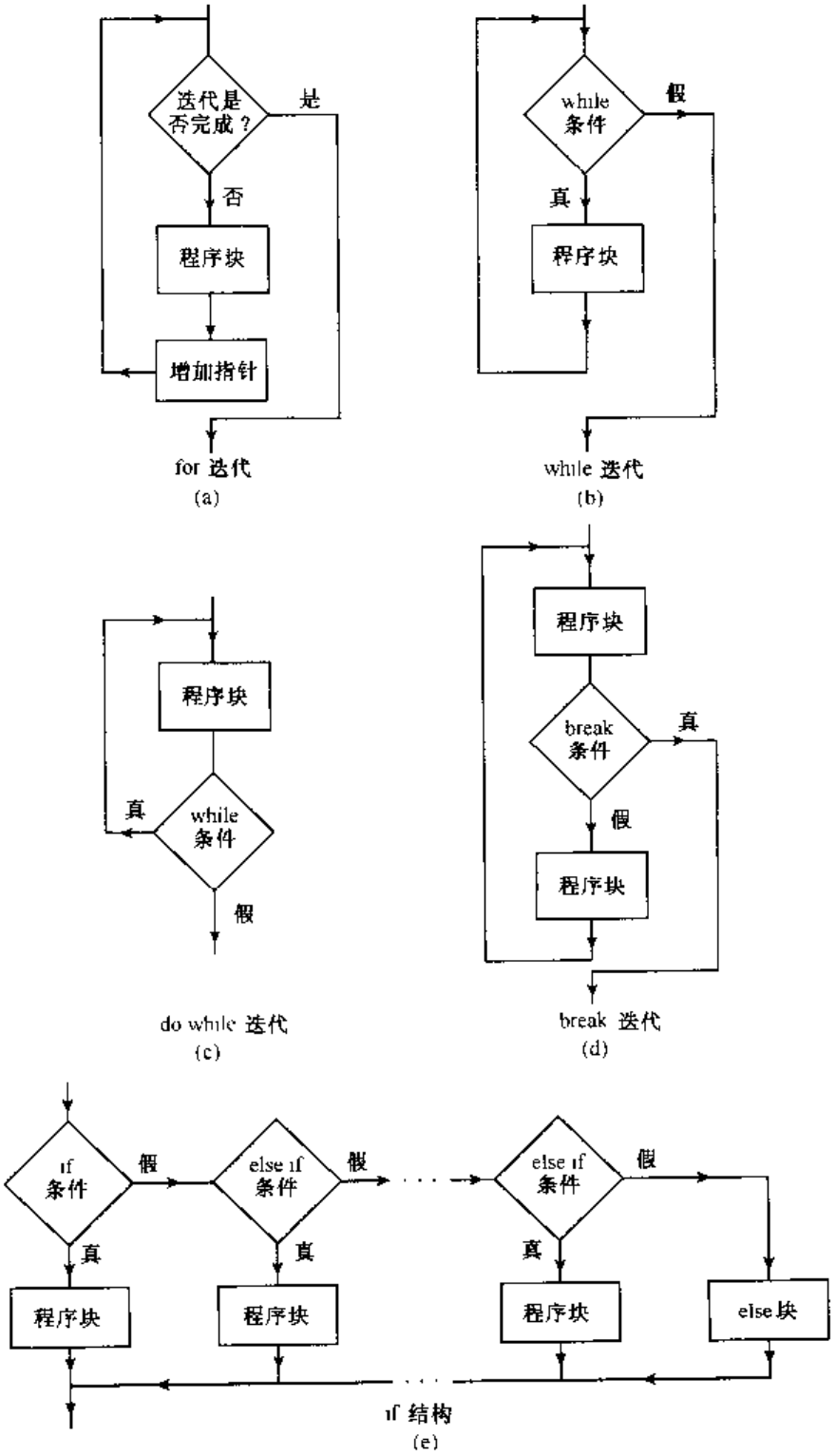
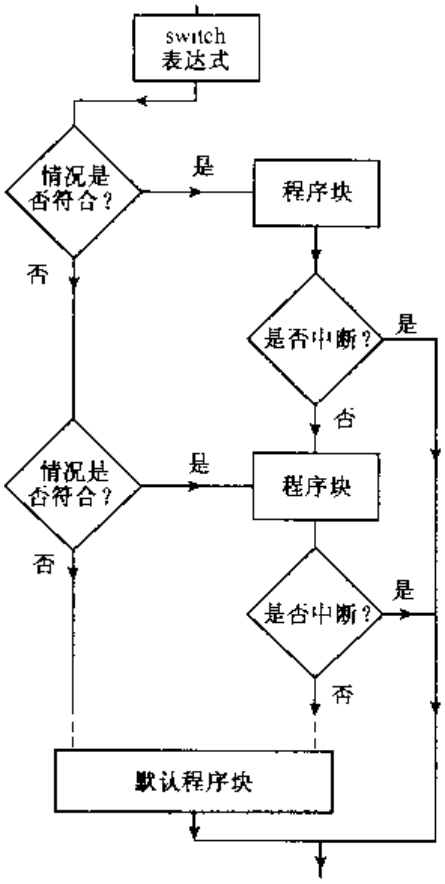


图 1.1.1 结构化程序设计中的标准控制结构：(a) for 迭代；(b) while 迭代；(c) do while 迭代；(d) break 迭代；(e) if 结构；(f) switch 结构



switch结构
(f)

图 1.1.1(续) 结构化程序设计中的标准控制结构：(a) for 迭代；(b) while 迭代；(c) do while 迭代；(d) break 迭代；(e) if 结构；(f) switch 结构

1.1.2 标准结构目录

迭代

C++语言中，简单的迭代是由 for 循环实现的，例如：

```
for (j=2;j<=1000;j++) {
    b[j]=a[j-1];
    a[j-1]=j;
}
```

注意，总是将受控制结构作用的代码块缩进几格排印，而结构本身不缩进几格排印。还应注意，一般习惯于，将起始的大括号与 for 语句放在同一行，而不是下一行，这样可节约一整行。

if 结构

C++中的这个结构与 Pascal，Algol，Fortran 以及其他语言中的结构相似，其典型形式如下所示：

```
if (...) {
    ...
}
else if (...) {
    ...
}
else {
    ...
}
```

仅当程序块中有一条以上的语句时,复合语句才需要大括号,然而,C++中的if结构可能不如上面所展示的结构明确。例如,考虑以下语句:

```
if (b > 3)
    if (a > 3) b += 1;
else b -= 1;                    /* questionable! */
```

通过第二行缩进的语句可判断,这段程序的意图是:若b大于3且a大于3,则b加1。若b不大于3,b减1。但是按照C++语言规则,这段程序的实际意义是:若b大于3,则判断a值。若a大于3,则b增加1,若a小于或等于3,则b减1。关键在于else子句与最近打开的if子句相关连,无论把它放在页面的什么位置。如果语句包含括号,可以很容易地解决这种意义混淆的问题。在一些情况下,这样做在技术上也许是多余的,但却明确了编程者的意图并且改进了程序。因此上面的程序片段应改写为

```
if (b > 3) {
    if (a > 3) b += 1;
} else {
    b -= 1;
}
```

下面是一个主要含有if控制语句的工作程序:

```
#include <cmath>
#include "nr.h"
using namespace std;

int NR::julday(const int mm, const int id, const int iyyy)
在此程序中,julday返回由mm月,id日,iyyy年确定的历书日期所对应的正午开始的儒略历日数,所有变量都是整型,
正的年份代表公元后(A.D.),负的年份代表公元前(B.C.)。要注意,公元前1年之后是公元1年。
{
    const int IGREG=15+31*(10+12*1582);        公历1582年10月15日。
    int ja,jul,jy=iyyy,jm;

    if (jy == 0) nrerror("julday: there is no year zero.");
    if (jy < 0) ++jy;
    if (mm > 2) {                              这是if结构的范例。
        jm=mm+1;
    } else {
        --jy;
        jm=mm+13;
    }
    jul = int(floor(365.25*jy)+floor(30.6001*jm)+id+1720995);
    if (id+31*(mm+12*iyyy) >= IGREG) {         测试是否转变为公历。
        ja=int(0.01*jy);
        jul += 2-ja+int(0.25*ja);
    }
    return jul;
}
```

天文学把24小时算做一个周期,起始和结束都在正午,这种整周期数就是儒略历日数(Julian Day Number)[6]。儒略历日的零(Julian Day Zero)是在很久很久以前。一个方便的参考点是儒略历日2450000起始于1995年10月9日正午。如果知道历书中某天正午起始的儒略历日数,那么将其加1再对7求模,就可知道该天是星期几。结果是零对应星期天,1对应星期一,一直类推到6对应星期六。

while迭代

除Fortran外,许多优秀的语言都提供了如下C++范例的循环结构:

```
while (n < 1000) {
    n *= 2;
    j += 1;
}
```

在每次迭代之前检测控制项(本例中是n<1000),是这一结构的典型特点。若控制项不为真,所包含的语句将不执行。特殊情况时,如果当n大于或等于1000时遇到这段程序,甚至一次都不执行。

do-while 迭代

do-while 迭代是与 while 迭代成对的控制结构，在每一迭代的**末尾**测试控制条件。在 C++ 中，其结构如下所示：

```
do {
    n *= 2;
    j += 1;
} while (n < 1000);
```

在这种情况下，所包含的语句至少要执行一次，与 n 的起始值无关。

break 迭代

在这种情况下，程序进行不定次数的循环，**直到循环中某一处**（可能不只在一个位置）的条件测试为真，在那一处，退出循环并处理后续操作。在 C++ 中，简单的 break 语句就可实现这种结构，用来中断最里层的 for，while，do 或 switch 结构的运行，进而执行下一顺序指令（在 Pascal 和标准 Fortran 77 中，这个结构需要用到语句标号，因此程序可能不够清晰）。一个 break 语句的典型应用是

```
for(;;) {
    [statements before the test]
    if (...) break;
    [statements after the test]
}
[next sequential instruction]
```

下面的程序运用了几个不同的迭代结构。假设我们要去打猎，找一个正好是星期五，同时又是 13 号且满月的日期。下列程序正是完成这个任务的程序。其中附带给出了所有其他是星期五、同时又是 13 号的日期。

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include "nr.h"
using namespace std;

int main(void)    // Program badluk

{
    const int IYBEG=2000,IYEND=2100;          要进行搜索的日期范围。
    const DP ZON=-5.0;                        时区-5是东部标准时间。
    int ic,icon,idwk,im,iyyy,jd,jday,n;
    DP timzon=ZON/24.0,frac;

    cout << endl << "Full moons on Friday the 13th from ";
    cout << setw(5) << IYBEG << " to " << setw(5) << IYEND << endl;
    for (iyyy=IYBEG;iyyy<=IYEND;iyyy++) {     每年的循环，
        for (im=1;im<=12;im++) {              每月的循环，
            jday=NR::julday(im,13,iyyy);      13日是星期五吗？
            idwk=int((jday+1) % 7);
            if (idwk == 5) {
                n=int(12.37*(iyyy-1900+(im-0.5)/12.0));
                n值是对从1990年以来有多少次满月的第一次估计。把它反馈到同相程序并反复调整它，直到所期望的13日是满月。变量icon标志调整方向。
                icon=0;
                for (;;) {
                    NR::flmoon(n,2,jd,frac);          得到满月的日子n。
                    frac=24.0*(frac+timzon);          转变到恰当时区的小时数。
                    if (frac < 0.0) {                 将从正午时开始的儒略历日转为由午夜
                        --jd;                         开始的民用日。
                        frac += 24.0;
                    }
                    if (frac > 12.0) {
                        ++jd;
```

```
                    frac -= 12.0;
                } else
                    frac += 12.0;
                if (jd == jday) {                    是否是目标日期。
                    cout << endl << setw(2) << im;
                    cout << "/13/" << setw(4) << iyyy << endl;
                    cout << fixed << setprecision(1);
                    cout << "Full moon" << setw(6) << frac;
                    cout << " hrs after midnight (EST)" << endl;
                    break;                           break结构的一部分，匹配时情况。
                } else {                             没有到达目标日期。
                    ic=(jday >= jd ? 1 : -1);
                    if (ic == (-icon)) break;        另一个break,没有匹配的情况。
                    icon=ic;
                    n += ic;
                }
            }
        }
    }
}
return 0;
}
```

既是13日(时间区GMT-5)又是星期五,同时又是满月的日期是:1903年3月13日,1905年10月13日,1919年6月13日,1922年1月13日,1970年11月13日,1987年2月13日,2000年10月13日,2019年9月13日和2049年8月13日

其他"标准"结构

建议避免使用这些结构。每一种程序语言都有一些优点。这些语言在当时看起来似乎是一些好的想法,遗憾的是经不住时间的检验,它们使用户的程序难以译成其他语言,并且可读性差(因读者不熟悉这些不常用的结构)。可以用其他方法来替换这些结构。

1.1.3 关于深入讨论

每章中都有一些关于特定主题的深入讨论部分,其中的内容是该章主要论题之外的内容,或者是要求具有更高的数学基础,或者有的是具有更深远意义的论题,或者有的是没有经过很好检验的算法。首次阅读本书时,可以依个人的需要忽略这些高级论题。

读者可能已注意到,程序badluk在进行年月循环时,避免了使用将儒略历日转换为公历日的任何算法。这种转换程序有时是有用的,示例如下:

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::caldat(const int julian, int &mm, int &id, int &iyyy)
下面给出函数julday的反函数。这里julian是儒略历日数的输入,程序输出mm,id和iyyy分别表示月、日和年。儒略历
日从此天的正午算起。
{
    const int IGREG=2299161;
    int ja,jalpha,jb,jc,jd,je;

    if (julian >= IGREG) {      跨接公历，产生此修正。
        jalpha=int((DP(julian-1867216)-0.25)/36524.25);
        ja=julian+1+jalpha-int(0.25*jalpha);
    } else if (julian < 0) {    给儒略历世纪添加整数，以便日数为正，然后在最后减去它们。
        ja=julian+36525*(1-julian/36525);
    } else
        ja=julian;
    jb=ja+1524;
    jc=int(6680.0+(DP(jb-2439870)-122.1)/365.25);
    jd=int(365*jc+(0.25*jc));
    je=int((jb-jd)/30.6001);
    id=jb-jd-int(30.6001*je);
```

```
        mm=je-1;
        if (mm > 12) mm -= 12;
        iyyy=jc-4715;
        if (mm > 2) --iyyy;
        if (iyyy <= 0) --iyyy;
        if (julian < 0) iyyy -= 100*(1-julian/36525);
    }
```

适合于各种历史性日历的其他日历算法，请参阅[7]。

参考文献和深入阅读

Harbison, S.P., and Steele, G.L., Jr. 1991, *C: A Reference Manual*, 3rd ed. (Englewood Cliffs, NJ: Prentice-Hall).

Kernighan, B.W. 1978, *The Elements of Programming Style* (New York: McGraw-Hill). [1]

Yourdon, E. 1975, *Techniques of Program Structure and Design* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Jones, R., and Stewart, I. 1987, *The Art of C Programming* (New York: Springer-Verlag). [3]

Hoare, C.A.R. 1981, *Communications of the ACM*, vol. 24, pp. 75–83.
Wirth, N. 1983, *Programming in Modula-2*, 3rd ed. (New York: Springer-Verlag). [4]

Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley). [5]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [6]

Hatcher, D.A. 1984, *Quarterly Journal of the Royal Astronomical Society*, vol. 25, pp. 53–55; see also *op. cit.* 1985, vol. 26, pp. 151–155, and 1986, vol. 27, pp. 506–507. [7]

1.2 科学计算的C++约定

C++语言来源于C语言，而后者最初是为系统编程工作而不是为科学计算而设计的。相对于其他高级编程语言，C语言在许多方面使程序员"很贴近机器"。它的运算符丰富，能对机器语言指令集的多数功能直接寻址。它有许多不同的数据类型，例如长型(long)和短型(short)、有符号型(signed)和无符号型(unsigned)整数、浮点型(floating)和双精度型(double-precision)实数、指针(pointer)类型等，并且能以简洁的句法进行(数据的)转换和间接。它定义了基于指针(地址)的算术运算，指针可以很巧妙地与数组地址相联系，并与许多计算机的变址寄存器结构高度兼容。所有这些特性在C++语言中同样具备。

可移植性一直是C语言的另一大优点，C++同样如此。C语言是UNIX和Linux操作系统的基础语言。这些语言和操作系统确确实实已在上百种不同的计算机上得到了应用。这类语言的通用性、可移植性和灵活性已经吸引了越来越多的科学家和工程师。它通常用于实验硬件的实时控制，但是作为控制目的的操作系统，标准的UNIX或Linux核还不够理想。

C或C++在高水平的科学计算中的应用，例如数据分析、建模、浮点数值计算工作等，一般来说发展较慢。一方面是因为，对1960年以前出生的所有科学家和工程师，以及以后出生的大多数科学家和工程师来说，Fortran已占有如同他们母语一样的牢固地位。另一方面，C类语言缓慢进入科学计算是因为计算机科学家很晚才认识到语言的不足之处。例如，没有好的方法计算小的整数的自乘，当用到高水平结构时难以编译出优化的代码。

过去推出本书的C语言版本就是试图通过实例推出一套合理而实用的针对C语言科学计算的约定。C++版也是基于相同的原因。

非常需要有一套用于C++的编程约定。但不是为了要克服语言本身的限制(已多次从Pascal中得到经验)，而是为了在C++中从众多的可能性中选择最好和最合乎规律的技术方法，然后将这些技术完全一致地应用于各个程序。在本节的后面将提出一些问题，并说明本书所有程序中所采用的约定。关于C++语言的详细内容，Stroustrup的著作[1]是最好的通用参考书。

1.2.1 双精度

当本书的C版本首次问世时,多数科学家进行大多数计算时默认的精度是单精度。其原因在于双精度对于运行速度和存储量都提出了很高的要求(很多情形下,C语言自动将float变量转换为double是不可取的)。

如今,对运行速度的要求基本上不存在了,甚至有些机器的单精度慢于双精度。而且人们也很少关心存储量管理问题。因此,在C++版本中,默认的精度是双精度。为了便于读者将其改变为单精度(或者为四倍精度),没有在程序中将类型double硬性编码。相反,通过定义typedef使用名字DP,

```
typedef double DP;
```

这个typedef两次出现,一次在nrtypes.h而另一次在nrutil.h。为了将默认精度改变为单精度,只需要在上述两处将定义改变为

```
typedef float DP;
```

也必须在一些程序中改变某些"精度参数"值,附录C中将对此进行深入讨论。对一些算法,需特别关心其舍入误差。在那些情况下,明确建议永远采用双精度。

1.2.2 函数声明、头文件和名字空间

在C++中,函数除非事先声明,否则不能调用。一个函数声明给出函数的返回值类型、其数值和它的自变量类型(函数说明又称为函数原型或者函数标题)。例如:

```
int g(int x, int y, double z);      函数说明。
```

一个函数必须在某处被定义(而且只定义一次)。一个函数定义由函数声明加上函数主体,即实际完成工作的代码组成。例如:

```
int g(int x, int y, double z)       函数定义。
{
    return x+int(z)/y;
}
```

如果全部代码在一个文件中,通常可以安排每一个函数定义位于调用它的函数之前。因为定义包含声明,故编译器能够检查一个给定的函数调用通过恰当的自变量类型来实施。不过,这种设置仅仅对于小的程序才是可行的。一般来说,一个C++程序由多个源文件组成,它们是被分别编译的,在缺乏某些附加的帮助时,编译器不能检查每个函数调用的一致性。在这种情况下,一种简单而安全的方法如下[1]:

- 在头(.h)文件中,每个外部函数应有一个单独的声明。
- 带有函数定义(主体)的源文件也应该包含头文件,以便编译器能够检验声明和定义是否一致。
- 调用函数的每一个源文件都应该包含相同的头文件。

在本书的程序中,nr.h是包含所有声明的头文件,列在附录A中。应该将语句#include "nr.h"放在调用本书数值算法程序的每一个源文件的顶部。

名字空间是隐藏一个程序模式之实施的工具,使得只能从用户程序访问它的界面。这使得在程序的其他部分能够控制变量和函数成为"可视的",并且提供一种途径将相关用途的对象和函数归入各自的模式。

所有本书数值算法程序的函数声明都在名字空间NR中。事实上,文件nr.h有以下结构:

```
namespace NR {
    void addint(Mat_O_DP &uf, Mat_I_DP &uc, Mat_O_DP &res);
        ...
    void zroots(Vec_I_CPLX_DP &a, Vec_O_CPLX_DP &roots, const bool &polish);
}
```

(Mat_O_DP等的类型将在后面解释。)本书数值算法程序的函数定义有以下格式:

```
#include "nr.h"

void NR::addint(Mat_O_DP &uf, Mat_I_DP &uc, Mat_O_DP &res)
    ...
```

注意代码中包含 nr.h（因为名字空间 NR 在那里定义）。另外，函数名字之前必须放置 NR::，因为 NR 是辖域，函数在其中被定义。注意，如果该函数要调用另一个数值算法的函数，不要求特别的声明。因为本书所有的算法都在名字空间 NR 中，它们都在彼此的辖域内。

当编写一个程序需要调用本书算法的函数时，程序必须包含 nr.h。有三种方式实现一个特定的算法：

- 用明显的辖域分解调用函数：

```
NR::addint(uf, uc, res);
```

- 在函数调用前放一个 using 声明：

```
using NR::addint;
    ...
addint(uf, uc, res);
```

- 在程序中包含一个 using 指令：

```
using namespace NR;
    ...
addint(uf, uc, res);
```

这种指令使得**所有**在 NR 中的函数是可视的，严格遵循规范者将觉得这样做是很草率的。

文件 nr.h 的更细致讨论在附录 A 中给出。

在此，介绍名字空间的另外一个特征，该特征在本书算法中会广泛使用。人们经常要定义一个辅助函数以便被另外一个函数应用。这种辅助函数承担的任务没有通用用途，所以希望它的定义仅局限于主要函数被定义的文件内。在 C 语言中，为达到此目的可以定义函数为 static 函数。在 C++ 中，最好保留 static 以声明函数内的对象和类，它们的存储必须是非易失的。隐藏在程序中的局部函数可以放在一个未命名的名字空间内。只有在同一个文件内的函数可以访问在一个未命名名字空间内的对象（例如，参见 10.1 节中的golden.cpp）。

1.2.3 const 校正

在关于 C++ 的讨论中，很少有论题比关键字 const 的问题更引起关注的了。我们坚信在任何可能的地方使用 const 能达到所谓的"const 校正"。如果具有在其说明后不能用 const 将其改变的限定符，则编译器会自动捕获代码错误。另外，使用 const 会使代码更易读，当一个程序的自变量之前有 const 时，就可立刻知道该程序将不修改对象。反之，如果没有 const，可以认为对象会在某处改变。

我们推荐使用 const，甚至在许多人认为它是多余的地方也插入 const：如果一个自变量**通过值**传送到一个函数，那么该函数对它进行复制。甚至这个复制被函数修改了，原来的值在函数退出之后也是不改变的。虽然可以轻松地改变这种自变量的值，后者已经通过值被传送，不过我们相信这种用法是容易出错且难于阅读的。如果通过值传送仅仅想输入一个变量，那么尽量让它清晰易懂。所以，如下声明一个函数 $f(x)$：

```
double f(const double x);
```

如果在该函数中希望用一个局部变量，它的初始值取为 x 但之后改变，需要定义一个新的量，千万不要使用 x。如果在说明中放置 const，编译器将避免造成这个错误。

有人认为，对自变量应用 const 会使函数缺乏通用性。情况正好相反，通过一个非 const 变量调用一个预设为 const 自变量的函数时，包含有一个"平凡的"转换。不过，试图传送一个 const 量给一个非 const 自变量则是错误的。

使用 const 的最后一个理由是，它允许做出一些使用者自定义的转换。如同我们将在 1.3 节中所见到的，这是一个关键点，直接关系到写的函数能够透明地用于任何矩阵/向量类库。

1.2.4 向量与矩阵

向量与矩阵是数值程序的基础建筑块。一维的 C 风格数组，例如被说明为 double b[4]，是 C++ 的一部分，偶尔用于数值任务。但是，固定尺度的二维数组，例如 double a[5][9]是绝不可取的。科学程序在实现某些力学计算时需要**可变维数数组**，后者根据其二维尺度的实时信息被传送给一个函数。

技术上，C 或 C++ 允许编译以下语句：

```
void someroutine(a,m,n)
double a[m][n];              /* ILLEGAL DECLARATION */
```

并且每次键入 someroutine() 时执行代码以计算可变维数 m 和 n(或者任何可变维数表达式)。遗憾的是,上面的片断是 C++ 语言定义所禁止的。

在 C++ 中,类是处理矩阵和向量的自然方式。一个矩阵对象不仅能包含矩阵元素的值,而且包含矩阵尺寸的信息。此外,类可以提供各种重载运算符和函数以利于高水平程序。C++ 标准已经为向量提供了这种类(valarray),但是没有提供给矩阵。许多人已经为一维和二维数组编写了他们自己的类库。多数这种库出色地提供了一套高水平结构,但是在执行中是极端无效的。

在准备编写本书时,我们的原始宗旨是"按要求做":我们将用所有必要的高水平结构去构造一个类库,使它是有效的。但这是一件极不简单的任务,尤其是因为许多编译器尚未实现 C++ 标准的特性,而后者是使代码有效的必要条件。此外,很快将意识到这是一种完全错误的方式。当读者已经投入很大精力去开发或者使用不同的类库时,就没有必要要求读者采纳我们的类库了。

因此,我们以完全相反的方式进行工作。本书中的算法函数是用向量类 NRVec 和矩阵类 NRMat 编写的。这些类在 nrutil.h 中定义,并且提供向量和矩阵运算的最小实施。可以用以下两种方式之一使用本书中的程序:

- 在你的所有程序中使用我们的类管理向量和矩阵。当包含 nr.h 使用我们的算法时,将自动包含 nrutil.h,所以我们的向量和矩阵类是可以使用的。这种方式的惟一缺点是,我们的类并不提供高水平的结构。
- 使用你喜欢的任何向量和矩阵类。在 1.3 节中,将介绍如何建立一个修正的 nrutil.h,以使你能够透明地调用我们的程序,即使程序是用数值算法的类 NRVec 和 NRMat 来声明的。

类 NRVec 和 NRMat,以及它们与其他类库一起使用的方式是相当复杂的,多数读者可能没兴趣探究,这将在 1.3 节中讨论。下面讨论如何应用算法中的向量和矩阵。

1.2.5 向量和矩阵是零基的

C 和 C++ 中的数组是"零基"或者"零偏移"的。由语句 double a[3];声明的数组的有效引用有 a[0],a[1] 和 a[2],但是没有 a[3]。然而,许多数学算法自然地从 1 到 n 而不是从 0 到 $n-1$ 进行。在本系列丛书的 C 版本中,曾经介绍如何运用 C 语言的机能去有效而简练地声明一个单位偏移数组,以及如何与零偏移数组一起使用它们。不过,一些读者并不喜欢单位偏移数组。因此,本书所有程序全部都用零基的数组。甚至在数学算法中比较自然地表示从 1 到 n 运行的指标,我们在正文中表示它们时也用从 0 到 $n-1$ 的指标。于是,代码中的数组和正文中的数组有相同的指标。

1.2.6 本书算法中向量和矩阵类型的名称:nrtypes.h

在 1.3 节中将详细地介绍向量和矩阵类如何被实施,如何交替地使用任何其他矩阵/向量库来代替它。许多读者会发现这是个很复杂的过程。幸运的是,几乎可以完全忽略矩阵/向量库的细节。实际上,如果仔细阅读本书的算法函数,会发现在任何地方都不会出现类名 NRVec 或 NRMat。反之,将看到形如 Vec_I_DP 和 Mat_O_INT 的名字声明向量和矩阵。这些是将要实际应用的标识符,文件 nrtypes.h 在一个 typedef 定义集合中封装所有这些定义。

在 nrtypes.h 中定义的典型类型的名字由三部分组成:

- Vec 或 Mat 用于向量或矩阵。
- I, O 或 IO 用于输入、输出或输入输出。这些符号对应于 Fortran 90 定名的"intent in","intent out" 和"intent inout",但这些概念是通用的。它们分别描述该数组是被传送到一个函数从而为该函数提供值,但不提供返回值(intent in);从函数返回值,但不提供任何值(intent out),或者两者都提供(intent inout)。
- 数量类型,它是默认的。例如,用 INT 表示 int,DP(双精度)表示 double,CPLX_SP(复的单精度)表示 complex<float>,等等。以下是依此表示的全部类型的列表:

```
BOOL            bool
CHR             char
UCHR            unsigned char
INT             int
UINT            unsigned int
LNG             long
ULNG            unsigned long
SP              float
DP              double
CPLX_SP         complex<float>
CPLX_DP         complex<double>
ULNG_p          unsigned long *
DP_p            double *
FSTREAM_p       fstream *
```

在 nrtypes.h 中，一个双精度向量的示例如下：

```
typedef const NRVec<DP> Vec_I_DP;
typedef NRVec<DP> Vec_DP, Vec_O_DP, Vec_IO_DP;
```

其中第一行表明，输入向量为 const 双精度。第二行表明，输出和输入输出向量不是 const。标识符 Vec_DP 是在算法函数中局部声明的向量，"in"和"out"项对它是没有意义的。

通过这样定义的类型，可以圆满地编写一个程序而不必涉及基础向量和矩阵类的实施。例如，可以用以下的语句：

```
double func(Mat_I_DP &a, Vec_IO_DP &b);
{
    const int N=10;
    Mat_DP d(N,N);

    Vec_DP *e=new Vec_DP(N);
    ...

    delete e;
}
```

这些定义的完全集合包含在文件 nrtypes.h 中，附录 B 有其副本，但可能不会经常用到。可以参考 1.0.1 节。

1.2.7 一些技巧

通常我们都希望保持程序代码紧凑，避免不必要的空格，除非这些空格能使程序更加清晰。通常不在赋值运算符(=)两旁设空格。但由于历史的原因，会看到本书中有 y= -10.0;或 y=(-10.0);以及 y= *a;或y=(*a);等形式，这是因为曾经有一些 C 编译器将(不存在的)运算符"=-"看做是等价于减法赋值运算符"-="，"=*"等同于乘法赋值运算符(*=)。圆括弧存在同样的问题。除非已记住 C++运算符的运算顺序和相关规则，否则不可能有效地编写(或阅读)C 程序。熟读表 1.2.1。

表 1.2.1 C++运算符的运算顺序及相关规则

::	辖域分辨符	从左到右
()	函数调用	从左到右
[]	数组元素引用	
.	成员引用	
->	对象指针复引用	
++	后置自增	从右到左
--	后置自减	
!	逻辑非	从右到左
~	位操作补	
-	一元减法	
++	前置自增	
--	前置自减	
&	地址	

(续表)

*	(间接引用的)内容	
new	分配(自由存储空间)	
delete	释放(自由存储空间)	
(type)	类型强制(转换)	
sizeof	字节长度	
*	乘	从左到右
/	除	
%	求余	
+	加	从左到右
-	减	
<<	位左移	从左到右
>>	位右移	
<	算术小于	从左到右
>	算术大于	
<=	算术小于或等于	
>=	算术大于或等于	
==	算术等于	从左到右
!=	算术不等于	
&	位操作与	从左到右
^	位操作异或	从左到右
|	位操作或	从左到右
&&	逻辑与	从左到右
||	逻辑或	从左到右
? :	条件表达式	**从右到左**
=	赋值操作符	**从右到左**
以及 += -= *= /= %=		
<<= >>= &= ^= |=		
,	顺序表示	从左到右

从不使用寄存器存储分类符 register 。良好的经过优化的编译器能很好地决定将什么放在寄存器中,而且有时最好的选择很不直观。

对于转换类型,用 C++ 构造函数 int(x) 而非 C 风格的 (int) x 。类似地,如果一个指向函数的指针作为自变量来传递的话,我们用 C++ 形式的 func(x) 而非 C 风格的 *func(x)来调用这个函数。

我们的某些程序需要定义一个全局向量或矩阵以在两个程序之间通信,称它们为 one() 和 two() ,不需使用函数自变量。一般而言,数据集合的尺寸需要随运行时间动态地设置。针对这样的情形,构造一个全局变化的指针指向数据,所以函数 one 看起来如下所示:

```
Vec_DP *xvec_p;                 定义全局指针。

void one(...)
{
      ...
   xvec_p=new Vec_DP(n);        分配长度为n的存储。
   Vec_DP &xvec= *xvec_p;       换名以简化后面的代码。
      ...
   delete xvec_p;               完成后收回存储。
```

定义引用变量 xvec 仅仅是使之后的代码比较容易写和读。(*xvec_p)[i] 可以用 xvec[i] 代替。

为了在函数 two 中使用全局向量,可以使用以下的方案:

```
extern Vec_DP *xvec_p;          说明全局指针，别处定义。

void two(...)
{
        ...
    Vec_DP &xvec= *xvec_p;      换名以简化后面的代码。
        ...
```

(另一种方案是全局地声明一个零尺寸的向量，然后在函数 one() 内适当地重新说明尺寸。不过，我们不希望假定在向量和矩阵库中存在一个重新说明尺寸的函数。)

我们已从侧面提到了计算数值的小整数幂，尤其是平方和立方的计算问题。C++ 中这种操作的遗漏可能会困绕科学计算程序员。所有良好的 Fortran 编译器都能识别如(a+b)**4 这样的表达式，并产生一行代码。在这种情况下，仅需一次加法和两次乘法。所识别的常整数次幂可达到 12 是件很普通的事。

在 nrutil.h 中提供了直接插入的模板函数以进行平方计算，其定义为

```
template<class T>
inline const T SQR(const T a) {return a*a;}
```

可以自己构造更高的幂次。我们还提供了一些类似的函数进行其他简单运算：SQR，MAX，MIN，SIGN 和 SWAP。它们的含义是显然的(SIGN(a,b) 返回 a 乘以 b 所得的值)。

也许在某一天，用 C++ 进行科学编程会被广泛接受，但现在还需特别注意。

参考文献和深入阅读

Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley).
[1]

1.3 向量和矩阵类的实施

在本节中介绍本书所用的向量和矩阵类的实施细节。首先介绍默认的类，然后讨论如何对本书的算法函数用其他的矩阵/向量库代替。如果你是一位很有经验的 C++ 程序员，则下面的内容将对你很有帮助。另外要记住，在实际应用中绝对不需要使用名字 NRVec 或 NRMat。反之，如上一节所述，只使用到 Vec_I_DP 之类的标识符。

1.3.1 默认的向量和矩阵类

以下是 NRVec 类的声明：

```
template <class T>
class NRVec {
    private:
        int nn;                                          数组长度，下标 0..nn-1。
        T *v;                                            指向数据数组。
    public:
        NRVec();                                         默认的构造函数。
        explicit NRVec(int n);                           构造长度为n的向量。
        NRVec(const T &a, int n);                        初始化常值a。
        NRVec(const T *a, int n);                        初始化为C类型数组a中的值。
        NRVec(const NRVec& rhs);                         复制构造函数。
        NRVec& operator=(const NRVec& rhs);              指定操作符。
        NRVec& operator=(const T& a);                    给每个元素分配a。
        inline T & operator[](const int i);              返回元素号码i。
        inline const T & operator[](const int i)const;   用const。
        inline int size() const;                         返回向量长度。
        ~NRVec();                                        析构函数。
};
```

私有变量在附录 B 中讨论。下面关注一下公共界面。各种构造函数允许向量用下例的方法来声明：

```
NRVec<double> v;            零长度数组。
NRVec<double> w(10);        w[0..9].
NRVec<int> x(a,10);         x[0..9] = a.
NRVec<int> y(b,10);         y[0..9] = b[0..9],b为C类型的数组。
NRVec<int> z=y;             z[0..9] = y[0..9],y为一个NRVec。
```

关键字 explicit 防止编译器对整数进行隐式类型转换。如果没有明确的目的,就肯定不需要这种转换,它通常会导致一些难于分析的错误。所以,使用 explicit 是一种好方法。

重载的赋值运算符允许用以下的代码(假定有上述声明):

```
y=x;                    从x向y复制值。
x=1;                    置x的所有元素为1。
```

重载的下标运算符[]在程序中反复使用,以访问数组元素。例如,x[6]是向量中的第7个元素。1.3.2节中将讨论下标运算符的两种形式。

最后,在向量中,函数 size() 返回元素的数目:

```
NRVec<double> w(10);
int n = w.size();       置n为10。
```

和后面将要讨论的一样,可以将任何你喜欢的向量类和本书中的函数一起使用,只要它提供上面的基本功能(函数名字可以不同,但它的功能必须相同)。事实上,甚至可以更方便。只要对一个长度 n 的向量提供构造函数、下标运算符以及函数 size(),多数算法仍然是可行的。大约有10个算法必须有某种被显式循环替换的代码去处理值或者数组的初始赋值(数值算法示例程序在 NRVec 类中对附加的函数做了更广泛的应用)。

矩阵类 NRMat 非常相似于 NRVec:

```
template <class T>
class NRMat {
    private:
        int nn;                                          行与列的数目,下标范围
        int mm;                                          是0 nn-1,0 mm-1。
        T **v;                                           数据存储。
    public:
        NRMat();                                         默认的构造函数。
        NRMat(int n, int m);                             构造nxm矩阵。
        NRMat(const T& a, int n, int m);                 初始化常值a。
        NRMat(const T *a, int n, int m);                 初始化为C类型数组a中的值。
        NRMat(const NRMat& rhs);                         复制构造函数。
        NRMat& operator=(const NRMat& rhs);              指定操作符。
        NRMat& operator=(const T& a);                    给每一个元素分配a。
        inline T* operator[](const int i);               下标:指向行i。
        inline const T* operator[](const int i)const;    用const。
        inline int nrows() const;                        返回行数。
        inline int ncols() const;                        返回列数。
        ~NRMat();                                        析构函数。
};
```

只有一点需要注意,就是 operator[] 的返回类型。如果 a 是类型 NRMat 的返回值,则希望 a[i] 指向矩阵的 i 行,使得 a[i][j] 是矩阵元素 (i,j)。在上述的默认类中,这意味着 a[i] 必须是类型 T *。然而,在另一个更为复杂的类库中,它可以是其他的类型。

和向量的情形一样,在下文将发现在我们的算法中如何应用其他的类,只要它提供上面提到的基本函数。如果所提供的是构造一个 $m \times n$ 矩阵的构造函数、下标运算符以及表示矩阵行和列数目的函数,则只需重写5个程序。

我们的程序中只有两个用到三维数组:12.5节中的 rlft3 和 17.3 节中的 solvde。在 nrutil.h 中的类 NRMat3d 提供了数据结构。本书没有特别说明如何用你自己的类代替我们所提供的类,不过如有必要,可以遵循下面描述的代替向量和矩阵的技巧。

NRVec 和 NRMat 类的全部实施代码在 nrutil.h 中,附录 B 有其副本。注意,很容易对向量使用标准库类 valarray 代替 NRVec,即用以下简单的方法替换全部声明和 NRVec 类的实施:

```
#define NRVec valarray
#include <valarray>
```

遗憾的是,标准库不对矩阵提供任何类似于 valarray 的类。

1.3.2 关于 const 校正的进一步讨论

需要详细说明 const 对非简单类型,诸如一个类是函数的自变量的作用。基本上它保证对象不被函数修改。换言之,数据元素不变化。但要注意:如果一个数据元素是指向某个数据的指针,而且数据自身不是一

个变元，那么尽管指针是不能变的，但该数据可以被改变。

对于一个取 NRVec 自变量 a 的函数 f，这意味着什么呢？为避免不必要的复制，总是通过引用传送数组。在声明一个函数自变量时，考虑具有和不具有 const 之间的差别：

```
void f(NRVec<double> &a)     或     void f(const NRVec<double> &a)
```

具有 const 意味着 f 不修改 a 的数据元素。但按照函数声明，语句

```
a[i] = 4;
```

是完全可以的，因为正在修改的是被指针指向的数据，而不是指针本身。

保护数据的方法是，可以声明operator[]**返回类型**是 const 。这就是为什么在 NRVec 类中存在 operator[] 两种版本的原因：

```
T & operator[](const int i);
const T & operator[](const int i) const;
```

第一种形式返回一个对可修改的向量元素的引用，而第二种则是对不可修改的向量元素的引用。在我们的例子中，这些工作的原理是：如果上面的函数 f() 的自变量 a 被说明为 const 引用，那么当 a 被间接引用时，operator[] 的第二种形式被实施（在声明中预告的 const 保证不修改被间接引用的对象，而且它必须与自变量保持 const 属性的一致）。于是，const 返回类型允许有表达式 x = a[i]，但不可以用表达式 a[i] = 4（a[i] 不是一个值）。反之，如果函数 f() 的自变量 a 被声明时不具有 const 限定符，则 operator[] 的第一种形式被实施。这时将允许两种句型的语句（a[i] 可以是一个值）。①

必须借助于引入 const 去保护数据。由于大多数矩阵/向量库遵循这种形式，许多人觉得结果是值得的。

但是，也必须认识到存在另外一种 operator[] 的实现，它忠实于 C++ 的基本原理："const 保护容器，不保护内容"。在这种情况下，应该只要一种形式的 operator[]，即

```
T & operator[](const int i) const;
```

无论向量是否通过 const 引用被传送，它将被实施。这两种情况下，向量的尺寸和指针是不变的，而元素 i 返回时有可能被修改。

虽然在我们的默认类中不使用第二种实现（因为默认类能够用 const 保护内容），但如果用后面介绍的类（这种类允许用自己喜欢的矩阵/向量库代替默认类），就必须使用它。原因在于，所有的 operator[] 调用都将强制实施对象的 const 属性，以便容许自动转换的可能性（当一个对象由引用传送给函数时，只有对象是 const 时才可能进行自动类型的转换）。虽然放弃额外的 const 检查是令人遗憾的，不过这是一条最好的途径，它允许透明地使用其他矩阵/向量库。

1.3.3 深入讨论：使用其他向量和矩阵库

假设有一个向量类 MyVec，我们想用它去代替 NRVec 。目标是当一个 MyVec 被传送到本书的算法函数时（一个 NRVec 是预期的），以及当一个 NRVec 由算法函数返回时（一个 MyVec 是预期的），有自动的类型转换。C++ 提供了必要的工具："构造函数转换"将 MyVec 转换为 NRVec ，"运算符转换"将 NRVec 转换为 MyVec 。

第一个关键思想是使 NRVec 成为一个 MyVec 的"包装类"，就是说，一个 NRVec "裹着"一个 MyVec 。第二个关键思想是该包装类既裹着 MyVec 又裹着一个对 MyVec 的引用。于是，当希望将 MyVec 转换为 NRVec 的时候，可以用构造函数初始赋值表指向存在的 MyVec 的引用。不发生任何数据的复制。

包装类必须提供早先引入的默认类的所有函数，我们将以同样的次序介绍它们。以下是包装类 NRVec 的开始：

---

① 可能有人会反对："第一种形式的 operator[] 实际上并不修改对象，只修改数据。为什么不能够在它后面放一个预告 const ？"这是因为编译器不能区分两种形式的差别。被重载的函数和运算符必须可通过自变量类型区分，而不是通过返回类型区分。

```
template<class T>
class NRVec {
    private:
        MyVec<T> myvec;
        MyVec<T> &myref;
    public:
        NRVec<T>() : myvec(), myref(myvec) {}
            ...
```

只有一个 MyVec 的变量和一个引用于它的变量是私有变量。私有变量 myvec 仅用于建立一个新的 NRVec 的时候。默认的构造函数仅调用 MyVec 的默认构造函数,然后指向相应的引用。类似地,余下的构造函数仅调用相应的 MyVec 构造函数:

```
explicit NRVec<T>(const int n) : myvec(n), myref(myvec) {}
NRVec<T>(const T &a, int n) : myvec(n,a), myref(myvec) {}
NRVec<T>(const T *a, int n) : myvec(n,a), myref(myvec) {}
```

注意,我们假定了 MyVec 类成员函数的句法和 NRVec 类是相同的,但是它容易满足不同句法的需要。例如,某些向量类要求自变量以相反方向排序:用 myvec(a,n) 代替 myvec(n,a)。

下面讨论构造函数的转换。它构造一个特定的 NRVec,使之指向 MyVec 的数据,在函数调用中后者容易满足 MyVec 当前的自变量作为 NRVec 先前的自变量被传送:

```
NRVec<T>(MyVec<T> &rhs) : myref(rhs) {}
```

因为所有在 NRVec 中的其他函数只通过 myref 访问对象,所以不需要初始赋值 myvec。

复制构造函数和赋值运算符可简单地调用 MyVec 的复制构造函数和赋值运算符:

```
NRVec(const NRVec<T> &rhs) : myvec(rhs.myref), myref(myvec) {}
inline NRVec& operator=(const NRVec &rhs) { myref=rhs.myref; return *this;}
inline NRVec& operator=(const T &rhs) { myvec=rhs; return *this;}
```

(在此假定 MyVec 函数进行"深度"复制,就是说它们复制数据,而不仅仅是复制引用。如果不符合上述情况,参看附录 B 中的 MTL 示例。)

类似地,如 size() 之类的函数通过调用对应的 MyVec 函数来实现:

```
inline int size() const {return myref.size();}
```

前面曾指出过,对于下标运算符只有保证容器的 const 属性的形式才被允许。我们希望自动的类型转换,而且仅仅对于被 const 引用传送的对象。因此,不能保证数据的 const 属性(参见 1.2.2 节中的讨论):

```
inline T & operator[](const int i) const {return myref[i];}
```

下面是从 NRVec 到 MyVec 的转换运算符,当用于 MyVec 表达式时它处理 NRVec 函数返回类型:

```
inline operator MyVec<T>() const {return myref;}
```

最后,析构函数是平凡的:MyVec 析构函数处理相应的数据。

```
~NRVec() {}
```

NRMat 的包装类恰好和 NRVec 的包装类有同一形式。只有一点需要注意:确保 operator[]的返回类型类似于单个[]解除引用的对象返回类型。于是,如 a[i][j] 之类的表达式将正确地工作。

在此不列出包装类的最终形式,将在附录 B 列出两个流行的矩阵/向量库的两个示例 nrutil.h 文件。它们是针对 Template Numerical Toolkit 或 TNT[1],以及 Matrix Template Library 或 MTL[2] 的。我们发现,TNT 特别容易与算法函数一起应用。你要做的只是对 MyVec 使用 TNT::Vector,对 MyMat 使用 TNT::Matrix。如果在使用包装类时遇到低效率的问题,我们的经验表明,对 TNT 来说,使用具有多数编译器的 TNT 比直接应用 TNT 多不到 10% 的费用。

最后需要说明:对通过引用传送自变量,文件 nrtypes.h 也需要改变为"const 保护容器,不保护内容"的约定。为便于做到这点,我们已经提供文件 nrtypes_lib 恰当地定义向量和矩阵自变量的这种 const 属性。所以,要简单地编辑 nrtypes.h 以包含这个文件,而不是默认的 nrtypes_nr.h(见附录 B)。

对 C++ 用户的建议:也可以通过将 NRVec 作为你的向量类的派生类实现到其他矩阵/向量库的界面。

不过，这并非包装类那样简练，特别是它依赖于向量类内部的实现，尽管该包装类仅仅用到公共界面和向量类的语义。

参考文献和深入阅读

Pozo, R., *Template Numerical Toolkit*, http://math.nist.gov/tnt. [1]

Lumsdaine, A., and Siek, J. 1998, *The Matrix Template Library*, http://www.lsc.nd.edu/research/mtl. [2]

1.4 误差、准确性和稳定性

虽然我们假定在正规的数值分析方面，读者没有前期的训练，但读者应对少数关键的概念有一般性的了解。下面将对这些内容进行简要的定义。

计算机不是以无限精度来存储数值的，而是以一定数量的**位**(二进制位)或**字节**(8位一组)的一种近似方式来存储数值的。几乎所有的计算机都允许程序员在几种不同的位或字节的表示或数据类型中进行选择。数据类型可以根据所用的位数(**字长**)来分类，也可从更基本的方面，即存储的数据是**定点**(int 或 long)或**浮点**(float 或 double)格式进行分类。

用整型表示的数是精确的。如果满足以下两个条件：第一，结果不超出整型数(通常指带符号整型数)所能代表的范围；第二，除法看做仅能产生整数结果，舍弃一切整余数，那么整型数据之间的算术运算也是精确的。

在浮点表示中，一个数的内部表示需要一个符号位 s(代表加号或减号)、一个精确的整指数 e 和一个精确的正整尾数 M。将它们组合起来代表的数为

$$s \times M \times B^{e-E} \tag{1.4.1}$$

其中 B 是表达式的基数(通常 $B=2$，但有时 $B=16$)，E 是指数的**偏移**(对任何给定的机器和表达式，它是一个固定的整常数)。图 1.4.1 显示了一个示例。

	符号位	8位指数	这一位能"省掉"	23位尾数	
$1/2 =$	0	1 0 0 0 0 0 0 0	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	(a)
$3 =$	0	1 0 0 0 0 0 1 0	1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	(b)
$1/4 =$	0	0 1 1 1 1 1 1 1	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	(c)
$10^{-7} =$	0	0 1 1 0 1 0 0 1	1	1 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 0 0 1 0 1 0	(d)
$=$	0	1 0 0 0 0 0 1 0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	(e)
$3+10^{-7} =$	0	1 0 0 0 0 0 1 0	1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	(f)

图 1.4.1 浮点数的典型32位(4个字节)格式。(a)数1/2(注意指数的偏移)；(b)数3；(c)数1/4；(d)数10^{-7}表示到机器精度；(e)同样的数10^{-7}，但进行了移位使得它与3有相同的指数位；数10^{-7}做了移位以后，所有的有效位都损失变成了零，移位使得两个数有相同的指数项后，这两个数才能相加；(f)两数之和$3+10^{-7}$，在机器精度范围内等于3。即使10^{-7}本身可以准确地表示，它也不能准确地添加到一个大得多的数上

几种浮点位模式都能代表相同的数。例如，如果 $B=2$，前导(高阶)位是零的尾数可以左移，即可以乘上2的幂，只要指数项减去一个补偿量就行了。"尽可能向左移位"的位模式称为**规格化**模式。大多数计算机总是产生规格化结果，这样可以不浪费尾数的任何位，从而达到较高精度的表示。由于正确的规格化尾数(当 $B=2$ 时)的高阶位**总是** 1，所以一些计算机根本不存储这一位而给出一位额外的精度。

浮点形式表示的数之间的算术运算是不精确的，即使运算的数能被精确地表示[即能用(1.4.1)式形式

表示的精确值]。例如,两个浮点数相加需要将(在数值上)较小数的尾数首先向右移(除以2),同时相应地增大它的指数,直到两个操作数有相同的指数项。通过如上的移位,较小的数将丢掉低阶(最小有效)位。如果两个运算数在数值上相差太大,那么较小的数将被零取代,因为它向右移而有效位都被忽略。

若一个最小的浮点数(在数值上)加上浮点数1.0后,产生一个不同于1.0的浮点数,则称这个最小的浮点数为**机器精度**ϵ_m。具有$B=2$和32位字长的典型计算机的ϵ_m大约是3×10^{-8}(在20.1节中还将详细讨论机器的精度问题,并且用程序来确定它们)。大概地说,机器精度ϵ_m是浮点数表示的相对精度,对应于尾数的最小精度位的变化。浮点数之间的大多数算术运算都将导致额外的至少为ϵ_m的相对误差。这种误差称为**舍入误差**

ϵ_m不是一个机器所能表示的最小浮点数,明白这一点很重要。最小浮点数取决于指数项有多少位,而ϵ_m取决于尾数有多少位。

舍入误差随着计算量的增加而积累。如果在获取运算结果的过程中,执行了N次同样的算术运算,只要舍入误差随机上下出现,**可能**仅产生$\sqrt{N}\epsilon_m$阶的总舍入误差(其中平方根产生于随机游动)。但这个估计值可能由于下面两个原因而大大偏离:

(i)经常会发生这种情况,所用的计算规则造成舍入误差优先在一个方向上积累。这时总舍入误差将是$N\epsilon_m$阶数量级。

(ii)一些特殊的不希望发生的情况下,会大大增加一次运算的舍入误差。一般来说,这种情况可出现在两个非常接近的数相减,所得结果的有效位只是两个操作数不同的(极少)几位低阶位。可能认为这样"巧合"的减法不可能发生,但事实并不总是这样。一些数学表达式大大增加了这种情况发生的可能性。例如,在熟悉的求二次方程根的公式中:

$$x=\frac{-b+\sqrt{b^2-4ac}}{2a} \tag{1.4.2}$$

当$ac\ll b^2$时,加法变得很难处理并且易于发生舍入误差(在5.6节中,将学习如何避免在这种特殊情况下产生的问题)。

舍入误差是计算机硬件的特性。还有另一种不同的误差,反映的是所用程序或算法的特性,与程序运行所依赖的硬件无关。许多数值算法都是计算"连续量"的"离散"近似值。例如,数值求积分就是通过计算在一组离散点集上的函数值,而不是计算"每个"点的值来进行的。或者,一个函数可由它的无限级数的有限项之和来近似计算,而不是无限项。在这些情况中,有一个可调整的参数,例如点数或项数,仅当参数趋于无穷时才能得到"真值"。任何实际的运算都是将参数选择为有限而又足够大的值。

真值与实际运算所得值之差称为**截断误差**。甚至在一个假设完美的、有无限精度表示的并且无舍入误差的计算机上,截断误差依然存在。作为一般的规律,程序员除了选择不会增大舍入误差的算法以外(见下面关于稳定性的讨论),对舍入误差总是无可奈何。然而,截断误差却完全在程序员的控制之中。事实上,可以略微夸张地说,灵活地使截断误差减到最小,实际上就是数值分析领域的全部内容。

大多数时候,截断误差和舍入误差彼此不会强烈地相互作用。可以将计算想像为首先具有截断误差,它在一个无限精度的计算机上运行时也会产生,然后"加上"与完成运算次数相关的舍入误差。

但有时,一个看似很吸引人的方法可能是不稳定的。这意味着,在开始阶段"加入"计算的舍入误差被连续地扩大直到淹没真值。一个不稳定的算法在一个假设完美的计算机上是有用的,但在实际应用中,我们要求算法是稳定的,或者是极其谨慎地使用那些不稳定的算法。

下面是一个简单的、带某些人为性质的不稳定算法的例子:假设要计算所谓"黄金分割"的所有整数幂,黄金分割数为

$$\phi\equiv\frac{\sqrt{5}-1}{2}\approx0.618\,033\,98 \tag{1.4.3}$$

结果表明(可以很容易地验证)幂ϕ^n满足简单的递推关系,

$$\phi^{n+1}=\phi^{n-1}-\phi^n \tag{1.4.4}$$

这样，知道了前面两个值$\phi^0 = 1$和$\phi^1 = 0.61\,803\,398$，就可以连续地使用(1.4.4)式，在每一步仅进行一次减法，而不是用ϕ进行一次较慢的乘法。

遗憾的是，递推式(1.4.4)还有另外一个解，即值$-\frac{1}{2}(\sqrt{5}+1)$。由于递推式是线性的，并且这个不期望的解在数值上比1大，因此任何由舍入误差引起的小误差都将导指数增加。在典型的32位字长计算机上，(1.4.4)式在大约$n = 16$时就会给出完全错误的结果，这时ϕ^n降到仅10^{-4}。所以递推式(1.4.4)是不稳定的，不能用于所指定的目的。

在本书后面的章节中，将遇到许多外形上更为复杂的稳定性问题。

参考文献和深入阅读

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), Chapter 1.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 2.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §1.3.

Wilkinson, J.H. 1964, *Rounding Errors in Algebraic Processes* (Englewood Cliffs, NJ: Prentice-Hall).

第 2 章　线性代数方程组求解

2.0　引言

假设有如下的线性代数方程组：

$$
\begin{aligned}
a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \cdots + a_{0,N-1}x_{N-1} &= b_0 \\
a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \cdots + a_{1,N-1}x_{N-1} &= b_1 \\
a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + \cdots + a_{2,N-1}x_{N-1} &= b_2 \\
\cdots \qquad\qquad & \cdots \\
a_{M-1,0}x_0 + a_{M-1,1}x_1 + \cdots + a_{M-1,N-1}x_{N-1} &= b_{M-1}
\end{aligned}
\tag{2.0.1}
$$

其中 N 个未知数 x_j，$j=0,1,\ldots,N-1$ 与 M 个方程相联系。系数 a_{ij} 已知，$i=0,1,\ldots,M-1$ 和 $j=0,1,\ldots,N-1$。右端项 b_i，$i=0,1,\ldots,M-1$ 也已知。

2.0.1　非奇异与奇异方程组

如果 $N=M$，方程的个数与未知数个数相同，这样就有可能求得 x_j 的惟一解集。分析可知，如果这 M 个方程中的一个或几个方程是其他方程的线性组合，这种情况被称为**行退化**；或者如果所有方程中某些变量是其他变量的同一线性组合，称之为**列退化**（对方阵来说，行退化蕴含着列退化，反之亦然），这两种情况都得不到惟一的解。退化的方程组称为"奇异"的。2.6 节中将介绍奇异矩阵的细节问题。

至少有两个附加的因素会导致数值计算上的错误：

- 尽管有些方程并非彼此精确的线性组合，但它们可能很接近线性相关，而在求解过程的某个阶段，机器的舍入误差使它们成为线性相关的。这时计算程序将失败，并会提示求解已失败。
- 求解过程中，舍入误差的积累也会使结果与真实解之间相差甚远。这个问题在 N 很大时特别容易发生，但计算进程在算法上并无错误。然而将结果直接代回原来的方程就会发现，由它计算出来的 x 解集却是错误的。由于求解过程中会不断发生相近抵消，方程组越接近奇异，这种情况越易发生。事实上，所计算的项可视为有效数字被全部舍去的特殊情况。

复杂的线性方程求解程序包中，使用了许多复杂的方法来检测和（或）改正这两种会引起错误的情况，当求解大的线性方程组时，就会需要这样的复杂处理。由于不存在所谓的典型线性问题，所以很难给出任何固定的准则。这里提供一个建议：N 为 20 至 50 之间的线性方程组，如果方程组不是接近奇异的，可以按惯例用单精度求解（32 位浮点表示），而不必借助于更复杂的方法。如果用双精度（64 位或更高），N 可以扩大到上千，在这点上，限制因素几乎总是机器时间，而不是精度。

甚至对于 N 上千或更大的线性方程组，当系数是稀疏的（即大部分为零）时也可以求解，可以采用利用了稀疏性特点的方法求解。2.7 节中将进一步讨论。

在谱问题上，由于其内在的属性经常遇到接近奇异的线性问题。在这种情况下，即便对于 $N=10$ 的情况（然而极少存在 $N=5$ 的情况），可能都需要借助很复杂的方法。奇异值分解技术（参见 2.6 节），有时能把奇异的问题变成非奇异的问题，这时额外的复杂处理就没有必要了。

2.0.2　矩阵

方程组（2.0.1）可写成下面的矩阵形式：

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{2.0.2}$$

式中黑点代表矩阵相乘，**A** 为系数矩阵，**b** 为写成列向量形式的右端项，

$$\mathbf{A}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,N-1} \\ a_{10} & a_{11} & \cdots & a_{1,N-1} \\ & \cdots & & \\ a_{M-1,0} & a_{M-1,1} & \cdots & a_{M-1,N-1} \end{bmatrix} \qquad \mathbf{b}=\begin{bmatrix} b_0 \\ b_1 \\ \cdots \\ b_{M-1} \end{bmatrix} \tag{2.0.3}$$

习惯上，元素 a_{ij} 的第一个下标表示行，第二个下标表示列。大多数实际应用中，不必知道矩阵在计算机存储器中是如何存放的，可以简单地用二维地址引用矩阵元素，例如，a_{34} = a[3][4]。在 1.2 节中已经看到这个 C++ 语言记号，事实上隐藏了一个相当精巧而又通用的物理存储方案。这方面的内容可以复习一下有关的章节。

2.0.3　线性代数数值计算

通过下面这些计算，开始进入本章要讨论的内容：

- 矩阵方程 $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$ 的求解。其中 **x** 为未知向量，**A** 为系数方阵，中间的黑点意味着矩阵相乘，**b** 为已知右端向量（参见 2.1 节～2.10 节）。
- 多个矩阵方程 $\mathbf{A}\cdot\mathbf{x}_j=\mathbf{b}_j$ 的求解问题。式中 $\mathbf{x}_j$ 为未知向量的集合，$j=0,1,\ldots$，每一个 $\mathbf{x}_j$ 对应一个不同的已知右端项向量 $\mathbf{b}_j$。这个求解计算中，关键的简化是矩阵 **A** 为常量矩阵，而右端项 **b** 是变化的（参见2.1 节～2.10 节）。
- 矩阵 $\mathbf{A}^{-1}$ 的计算。$\mathbf{A}^{-1}$ 为方阵 **A** 的逆，即 $\mathbf{A}\cdot\mathbf{A}^{-1}=\mathbf{A}^{-1}\cdot\mathbf{A}=\mathbf{1}$，其中 **1** 表示单位矩阵（除对角线上的元素是 1 外，其余元素均为零）。对一个 $N\times N$ 矩阵 **A** 来说，这个求解计算与前一个求解计算在 N 个不同 $\mathbf{b}_j$（$j=0,1,\ldots,N-1$）为单位向量（$\mathbf{b}_j$ 除第 j 个元素为 1 外均为零元素）的情况下是等价的。相应的 **x** 为 **A** 的逆矩阵的列（参见 2.1 节和 2.3 节）。
- 方阵 **A** 的行列式的计算（参见 2.3 节）。

如果 $M<N$，或者 $M=N$ 时方程组是退化的，则有效方程的个数少于未知数个数。在这种情况下，或者无解，或者有不止一个的解向量 $\mathbf{x}_i$。对后者，解空间为（不失一般性）$N-M$ 个向量（称之为矩阵 **A** 的零空间）的任意线性组合与一个特解 $\mathbf{x}_p$ 的和。寻找 **A** 的解空间包括：

- 矩阵 **A** 的奇异值分解。

这个专题在 2.6 节中讨论。

相反的情况是方程个数多于未知数数目，即 $M>N$。这种情况发生时，通常对（2.0.1）式没有解向量 **x**，这个方程组被称为**超限定的**。然而这种情况会经常出现，最好的解决方案是，寻找一个能同时近似满足所有方程的"折中"解。如果近似程度被限定在最小二乘意义上，也就是说，（2.0.1）式左端和右端之差的平方和最小，于是此超限定的线性问题简化为一个（通常是）可解的线性问题，称为

- 线性最小二乘问题。

简化后，待求解的方程组可写成 $N\times N$ 方程组的形式：

$$(\mathbf{A}^T\cdot\mathbf{A})\cdot\mathbf{x}=(\mathbf{A}^T\cdot\mathbf{b}) \tag{2.0.4}$$

式中 $\mathbf{A}^T$ 表示矩阵 **A** 的转置。方程组（2.0.4）称为线性最小二乘问题的**正规方程组**。奇异值分解与线性最小二乘问题之间有着紧密的联系，后者也将在 2.6 节中讨论。注意，直接求正规方程组（2.0.4）的解并非总是求解最小二乘问题的最好方法。

本章中其他一些论题包括：

- 解的迭代改进（2.5 节）。
- 各种特殊形式的矩阵：对称正定矩阵（2.9 节），三对角矩阵（2.4 节），带状对角矩阵（2.4 节），Toeplitz 矩阵（2.8 节），Vandermonde 矩阵（2.8 节），稀疏矩阵（2.7 节）。
- Strassen 的"快速矩阵求逆"（2.11 节）。

2.0.4 标准程序包

我们不能在本章或本书中告诉读者有关上面所述任务的全部内容。许多情况下,只能使用高级的黑箱程序包。优秀的程序包是通用的,尽管它们并非都是用C++编写的。作为大家所信赖的LINPACK的后继产品,LAPACK是由美国Argonne国家实验室开发的,并引起了特别的关注,因为它是公开发行的,并有免费使用的文档资料。ScaLAPACK可以用来进行并行计算,这些商业通用软件包在IMSL和NAG库中可以找到。

应当记住,复杂的程序包都是为超大型的线性系统求解而设计的。因而,为了减少运算次数和需要的存储空间,设计者花费了很大精力。按照输入系数矩阵的几种可能的简化形式:对称的、三角的、带状的、正定的等,为各种任务提供了不同版本的程序。如果有一个大矩阵是这几种形式之一,则自然应当利用提供的这些不同程序来提高效率,而不必使用为一般矩阵设计的形式。

有一个明显的分类方法,能把程序用的是"直接法"(即执行过程运算次数可判定)还是"迭代法"(即试图收敛到预定的解,不管需要多少步骤)区别开来。不论是由于N很大,还是由于问题是接近奇异的,当有效数字有丢失的危险时,迭代法更可取,在2.7节和第18章、第19章中将讨论关于迭代法的内容。这些方法是重要的,但超出了我们的讨论范围。然而,我们将讨论一种介于直接法和迭代法之间的技术,即解的迭代改进,它可以由直接法得到(参见2.5节)。

参考文献和深入阅读

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press)

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley).

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag). Chapter 4.

Anderson, E., et al. 2000, LAPACK User's Guide, 3rd ed. (Philadelphia: S.I.A.M.).

Coleman, T.F., and Van Loan, C. 1988, *Handbook for Matrix Computations* (Philadelphia: S.I.A.M.).

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall).

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag).

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), Chapter 9.

2.1 Gauss-Jordan消去法

求矩阵的逆时,Gauss-Jordan(高斯-若尔当)消去法与其他方法一样有效。解线性方程组时,Gauss-Jordan消去法既可求出具有一个或多个右端向量$\mathbf{b}$的方程组的解,也可求出矩阵的逆$\mathbf{A}^{-1}$。然而它的主要弱点在于:(i)需要同时存储和处理所有的右端项;(ii)不需要逆矩阵时,Gauss-Jordan消去法比求解单一线性问题的最好技术(参见2.3节)要慢1/3倍。这种方法的优点在于它跟其他直接法一样稳定,在使用全选主元法时可能更稳定(参见下面的内容)。

如果后面还要处理其他的右端向量,则求矩阵逆较好,这样可以将右端向量乘以矩阵的逆。这样做确实能求解,但却值得怀疑是否有舍入误差错误,所以不如一开始就用右端向量求得新向量。

由于这些原因,在求解线性方程组或求矩阵的逆时,通常Gauss-Jordan消去法不应成为首选方法。本章2.3节中的分解法较佳。使用Gauss-Jordan消去法的原因是,它是直接的、易懂的、非常稳定的方法。而且,当已经出错而你又认为其可能为线性方程组的解的时候,这种方法是一个极好的"心理上"的后备选择。

一些人认为这种后备不仅仅是心理上的，Gauss-Jordan 消去法是一种“独立的”数值方法。这种说法在多数情况下被证明只是一种空想。除了在选主元时相对有些差别外，Gauss-Jordan 消去法的实际操作过程与后两节中的程序操作过程是密切相关的。

为清晰起见，并避免总出现省略号(…)，下面写的方程组仅有 4 个方程和 4 个未知数，以及事先已知的 3 个不同的右端向量。用完全类似的形式，就可以写出更大一些的矩阵，并把方程组扩充成 $N\times N$ 矩阵，包含 M 个右端向量。下面实现的程序具有一般性。

2.1.1　列增广矩阵消去法

考虑线性矩阵方程

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \left[\begin{pmatrix} x_{00} \\ x_{10} \\ x_{20} \\ x_{30} \end{pmatrix} \sqcup \begin{pmatrix} x_{01} \\ x_{11} \\ x_{21} \\ x_{31} \end{pmatrix} \sqcup \begin{pmatrix} x_{02} \\ x_{12} \\ x_{22} \\ x_{32} \end{pmatrix} \sqcup \begin{pmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{pmatrix} \right]$$

$$= \left[\begin{pmatrix} b_{00} \\ b_{10} \\ b_{20} \\ b_{30} \end{pmatrix} \sqcup \begin{pmatrix} b_{01} \\ b_{11} \\ b_{21} \\ b_{31} \end{pmatrix} \sqcup \begin{pmatrix} b_{02} \\ b_{12} \\ b_{22} \\ b_{32} \end{pmatrix} \sqcup \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right] \tag{2.1.1}$$

其中圆点(·)表示矩阵相乘，而运算符⊔仅代表列增广，也就是说，去除相邻的小括号则运算符⊔的操作数构成增广矩阵。

无需花很多时间来写方程(2.1.1)，只要明白它仅仅说明 x_{ij} 是系数为 b_{ij} ($i=0,1,2,3$)的第 j 个右端项($j=0,1,2$)的向量解的第 i 个元素($i=0,1,2,3$)，而未知系数 y_{ij} 的矩阵是 a_{ij} 矩阵的逆。换句话说，矩阵

$$[\mathbf{A}]\cdot[\mathbf{x}_0 \sqcup \mathbf{x}_1 \sqcup \mathbf{x}_2 \sqcup \mathbf{Y}] = [\mathbf{b}_0 \sqcup \mathbf{b}_1 \sqcup \mathbf{b}_2 \sqcup \mathbf{1}] \tag{2.1.2}$$

其中 $\mathbf{A}$ 和 $\mathbf{Y}$ 均为方阵，$\mathbf{b}_i$ 和 $\mathbf{x}_i$ 为列向量，$\mathbf{1}$为单位阵，它同时解决了线性方程组集合

$$\mathbf{A}\cdot\mathbf{x}_0 = \mathbf{b}_0 \qquad \mathbf{A}\cdot\mathbf{x}_1 = \mathbf{b}_1 \qquad \mathbf{A}\cdot\mathbf{x}_2 = \mathbf{b}_2 \tag{2.1.3}$$

以及

$$\mathbf{A}\cdot\mathbf{Y} = \mathbf{1} \tag{2.1.4}$$

现在来证实下列关于式(2.1.1)的基本事实：

- 交换 $\mathbf{A}$ 的任意两行以及 $\mathbf{b}$ 和 $\mathbf{1}$ 的相应行，并不改变(或任何形式地打乱)$\mathbf{x}$ 及 $\mathbf{Y}$ 的解。它不过是把同一线性方程组中方程的次序调换而已。
- 同理，如果用 $\mathbf{A}$ 的任意一行和其他行的线性组合代替该行，只要对 $\mathbf{b}$ 和 $\mathbf{1}$ 的相应行进行相同的线性组合(当然变换后不再是单位阵)，解集就不会改变或受任何干扰。
- 交换 $\mathbf{A}$ 的任意两列，如果同时交换 $\mathbf{x}$ 和 $\mathbf{Y}$ 的相应行，其解不变。换言之，这种交换打乱了解的行的次序。如果是这样，需要把解的行恢复到原来的次序以得到原方程组的解。

Gauss-Jordan 消去法使用一个或多个上述操作，把矩阵 $\mathbf{A}$ 约化为单位矩阵。约化完成后，从(2.1.2)式中可以看出右端项即为解集。

2.1.2　选主元法

在“不选主元的 Gauss-Jordan 消去法”中，仅使用了上面所列的第二项操作，第零行元素被元素 a_{00} 除(可视为第零行与其他任意行的线行组合，只是对其他行用的是零系数)，然后用其他各行减去第零行乘以适当的系数，使得所有余下的 a_{i0} 为零。至此，$\mathbf{A}$ 的第零列为单位阵的形式了。现在处理第一列，将第一行除以 a_{11}，然后将第零、二、三行分别减去其适当的倍数，使得它们的第一列的元素为零。目前第一列也化简为单位阵的形式。类似处理第二列和第三列。在对 $\mathbf{A}$ 进行这些操作时，当然也要对 $\mathbf{b}$ 和 $\mathbf{1}$ 进行相应的操作(变换后的$\mathbf{1}$不是单位阵)

显然，在使用对角线上的元素作除数时，如果该对角线上的元素为零，则会遇到麻烦。附带提一下，我们

用做除数的元素称为**主元素**或**主元**。即使没有遇到零主元,事实上由于舍入误差的出现不选主元的 Gauss-Jordan 消去法(没有使用上述第一步和第二步过程)在数值上是不稳定的。尽管这一点不是一目了然的,但确实如此,请务必不要使用不选主元的 Gauss-Jordan 消去法(或是高斯消去法,原因见下面的内容)。

那么,什么是选主元法呢?其实不过是进行行交换(**部分主元法**)或是行列均交换(**完全主元法**),把一个特别需要的元素放到对角线的位置上来做主元。因为不打乱已构造好的单位矩阵部分,所以要选择的元素必须满足:(i)在要标准化的行或其下面的行中;(ii)在要消去的列或其右边的列中。部分主元法比完全主元法要简单些,因为不必记下解向量的排列顺序。部分主元法仅当主元已处于正确的列时,才是易于实现的。可以证明,在数学精度的意义上,部分主元法的效果与完全主元法一样好(进一步的论述和参考书目,参见[1])。为给出二者的区别,在本节的程序中用完全主元法,在 2.3 节中用部分主元法。

必须说明如何识别一个元素是否是特别需要的主元素。这个问题不是理论上能够完全解答的,在理论上和实践中,大家知道简单地选取最大元素(指绝对值)是个很好的方法。然而这么做的问题是,主元的选取要依赖于方程组最初的形式。如果初始方程组的第三个方程两边各乘以一百万,差不多可以保证提供第一个主元素,而方程组的实际解并不因进行过乘法而改变。因而习惯做法是,将所有方程左右两边进行缩放变换,使它们各自最大的系数标准化为单位值,这样选取的主元就应当是最大的主元素。这称为**隐式主元法**。另外需有记录保存每行所乘的比例因子(2.3 节的程序中包含了隐式主元法,但本节的程序中没有)。

最后,来考虑一下这种方法的存储要求。可以知道,在算法的每一步中,要么 **A** 的元素是 1 或是 0(如果它已经是化简成的单位矩阵的一部分),要么由单位矩阵**1**所开始的矩阵中相应元素会变成 1 或是 0(如果 **A** 中的对应元素还未被化简为单位矩阵形式)。因此,矩阵 **1** 不必单独存储;矩阵 **A** 的逆在 **A** 中逐步构造,同时破坏 **A** 原来的内容。同理,解向量 **x** 能够逐步替代右端项向量 **b**,且使用同一存储空间,因为每一列 **A** 中的元素约化后,**b** 中相应的行便不再使用。

下面的程序是完全主元法的 Gauss-Jordan 消去法:

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::gaussj(Mat_IO_DP &a, Mat_IO_DP &b)
用 Gauss-Jordan 消去法求解线性方程组(2.1.1)。a[0..n-1][0..n-1]为输入矩阵。b[0..n-1][0..m-1]为包含 m 个右
端项向量的输入矩阵。输出时,a 被其逆矩阵代替,b 被相应的解向量代替。
{
    int i,icol,irow,j,k,l,ll;
    DP big,dum,pivinv;

    int n=a.nrows();
    int m=b.ncols();
    Vec_INT indxc(n),indxr(n),ipiv(n);  这些整型数组用于记录主元。
    for (j=0;j<n;j++) ipiv[j]=0;
    for (i=0;i<n;i++) {                 约化列的主循环。
        big=0.0;
        for (j=0;j<n;j++)               用于寻找主元素的外层循环。
            if (ipiv[j] != 1)
                for (k=0;k<n;k++) {
                    if (ipiv[k] == 0) {
                        if (fabs(a[j][k]) >= big) {
                            big=fabs(a[j][k]);
                            irow=j;
                            icol=k;
                        }
                    }
                }
        ++(ipiv[icol]);
        至此我们已经求得主元素,如果需要,进行行交换把主元素放到对角线上。列并不进行实际交换,只是进行重
        新标注:indxc[i]为第 i+1 个主元素所在的列,即被约化的第 i+1 列;而 indxr[i]是主元素原来所在的行,如果 in-
        dxr[i]≠indxc[i],意味着要进行列变换。用这种形式记录,解集 b 最终的次序是正确的,而逆矩阵的列次序被
        打乱。
```

```
        if (irow != icol) {
            for (l=0;l<n;l++) SWAP(a[irow][l],a[icol][l]);
            for (l=0;l<m;l++) SWAP(b[irow][l],b[icol][l]);
        }
        indxr[i]=irow;                    用位于irow行icol列的主元素去除其所在的行。
        indxc[i]=icol;
        if (a[icol][icol] == 0.0) nrerror("gaussj: Singular Matrix");
        pivinv=1.0/a[icol][icol];
        a[icol][icol]=1.0;
        for (l=0;l<n;l++) a[icol][l] *= pivinv;
        for (l=0;l<m;l++) b[icol][l] *= pivinv;
        for (ll=0;ll<n;ll++)              下面进行行约化……
            if (ll != icol) {             ……当然主元素除外。
                    dum=a[ll][icol];
                    a[ll][icol]=0.0;
                    for (l=0;l<n;l++) a[ll][l] -= a[icol][l]*dum;
                    for (l=0;l<m;l++) b[ll][l] -= b[icol][l]*dum;
                }
        }
        这是列约化主循环的结尾。考虑已进行过列交换，为使解向量保持原来的次序，再根据其交换的相反顺序交换各
        列，以整理得到原始方程的解
        for (l=n-1;l>=0;l--) {
            if (indxr[l] != indxc[l])
                for (k=0;k<n;k++)
                    SWAP(a[k][indxr[l]],a[k][indxc[l]]);
        }                                 程序结束。
    }
```

2.1.3　深入讨论：行和列消去法策略

以上的讨论可以用下面的形式加以扩展。用某简单矩阵 **R** 前乘(即左乘)矩阵 **A** 来完成行操作。例如，矩阵 **R** 具有分量

$$R_{ij}=\begin{cases}1 & \text{如果 } i=j \text{ 和 } i\neq 2,4\\ 1 & \text{如果 } i=2,j=4\\ 1 & \text{如果 } i=4,j=2\\ 0 & \text{其他}\end{cases} \tag{2.1.5}$$

其效果是第二行和第四行交换。只进行行操作的 Gauss-Jordan 消去法(包括部分主元法的可能性)可由一系列左乘得到

$$\begin{aligned}\mathbf{A}\cdot\mathbf{x}&=\mathbf{b}\\ (\cdots\mathbf{R}_2\cdot\mathbf{R}_1\cdot\mathbf{R}_0\cdot\mathbf{A})\cdot\mathbf{x}&=\cdots\mathbf{R}_2\cdot\mathbf{R}_1\cdot\mathbf{R}_0\cdot\mathbf{b}\\ (\mathbf{1})\cdot\mathbf{x}&=\cdots\mathbf{R}_2\cdot\mathbf{R}_1\cdot\mathbf{R}_0\cdot\mathbf{b}\\ \mathbf{x}&=\cdots\mathbf{R}_2\cdot\mathbf{R}_1\cdot\mathbf{R}_0\cdot\mathbf{b}\end{aligned} \tag{2.1.6}$$

关键在于，因为矩阵 **R** 从右向左构造，所以在每一步中，等式右边是从一个向量简单地换成另一向量。

同样，列操作可以相应地用简单的称为 **C** 的矩阵后乘(或右乘)矩阵 **A** 实现，若(2.1.5)式中的矩阵右乘矩阵 **A**，则使矩阵 **A** 中第二列与第四列交换。列操作的消去法是，(概念上)在矩阵 **A** 和未知向量 **x** 之间插入列操作项及其逆操作。

$$\begin{aligned}\mathbf{A}\cdot\mathbf{x}&=\mathbf{b}\\ \mathbf{A}\cdot\mathbf{C}_0\cdot\mathbf{C}_0^{-1}\cdot\mathbf{x}&=\mathbf{b}\\ \mathbf{A}\cdot\mathbf{C}_0\cdot\mathbf{C}_1\cdot\mathbf{C}_1^{-1}\cdot\mathbf{C}_0^{-1}\cdot\mathbf{x}&=\mathbf{b}\\ (\mathbf{A}\cdot\mathbf{C}_0\cdot\mathbf{C}_1\cdot\mathbf{C}_2\cdots)\cdots\mathbf{C}_2^{-1}\cdot\mathbf{C}_1^{-1}\cdot\mathbf{C}_0^{-1}\cdot\mathbf{x}&=\mathbf{b}\\ (\mathbf{1})\cdots\mathbf{C}_2^{-1}\cdot\mathbf{C}_1^{-1}\cdot\mathbf{C}_0^{-1}\cdot\mathbf{x}&=\mathbf{b}\end{aligned} \tag{2.1.7}$$

(同时消去 $\mathbf{C}^{-1}$)得解为

$$\mathbf{x}=\mathbf{C}_0\cdot\mathbf{C}_1\cdot\mathbf{C}_2\cdots\mathbf{b} \tag{2.1.8}$$

要注意(2.1.8)式和(2.1.6)式之间的本质区别。在(2.1.8)式中，这一系列矩阵 **C** 以**倒次序**作用于向量 **b**，从而得到解向量。也就是它们必须依照这种倒次序存储。这一需求降低了列操作的实用性，例如在完全主元法的情况下，总是限制它们只进行简单的置换。

参考文献和深入阅读

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press). [1]

Carnahan, B., Luther. H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), Example 5.2, p. 282.

Bevington, P.R., and Robinson, D.K. 1992, *Data Reduction and Error Analysis for the Physical Sciences*, 2nd ed. (New York. McGraw-Hill), p. 247.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §9.3–1.

2.2 具有回代过程的高斯消去法

具有回代过程的高斯消去法的用途主要是适合于教学。它介于 Gauss-Jordan 这样的完全消去法和下章讨论的三角分解法之间。高斯消去法并不是把矩阵完全化简为单位阵,而是半单位阵,例如上三角矩阵。现在来看一下这么做的优点。

假定在进行 Gauss-Jordan 消去时,如 2.1 节所述,每一步仅有位于当前主元素下面的那些行做减法。比如,当 a_{11} 为主元素时,用它的值去除第一行(跟以前一样),但现在用主元素所在行仅把 a_{21} 和 a_{31} 消为零,而不消 a_{01},参见(2.1.1)式。再假定仅进行部分主元法消去,不进行列交换。这样未知量的顺序就不必修改。

处理完所有主元素后,得到化简后的方程如下(对单个右端向量的情况):

$$\begin{bmatrix} a'_{00} & a'_{01} & a'_{02} & a'_{03} \\ 0 & a'_{11} & a'_{12} & a'_{13} \\ 0 & 0 & a'_{22} & a'_{23} \\ 0 & 0 & 0 & a'_{33} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \end{bmatrix} \tag{2.2.1}$$

其中的撇号(′)表示 a 和 b 不再是原来的数值,而是经过所有行消去操作后得到的修改了的值。至此,**高斯消去过程**结束。

2.2.1 回代过程

下面介绍求解 x 的方法。最后一个 x(例中为 x_3)已可独立求出,即

$$x_3 = b'_3/a'_{33} \tag{2.2.2}$$

知道最后一个 x 后便可来求倒数第二个 x,

$$x_2 = \frac{1}{a'_{22}}[b'_2 - x_3 a'_{23}] \tag{2.2.3}$$

然后,继续求其前面的 x。一般步骤为

$$x_i = \frac{1}{a'_{ii}}\left[b'_i - \sum_{j=i+1}^{N-1} a'_{ij} x_j\right] \tag{2.2.4}$$

式(2.2.4)定义的过程称为**回代过程**。高斯消去法与回代过程结合起来即可求得方程组的解。

与 Gauss-Jordan 消去法相比,高斯消去法与回代过程相结合的优点在于它的运算次数要少一些。Gauss-Jordan 消去法的最内层循环每次包含一次减法和一次乘法,需要执行 N^3 次和 N^2M 次(当有 N 个方程式和 M 个未知数时)。而高斯消去法的相应循环仅执行 $\frac{1}{3}N^3$ 次及 $\frac{1}{2}N^2M$ 次(仅化简矩阵的一半,且要变成零的元素个数减少到三分之一)。每个右端项回代要执行 $\frac{1}{2}N^2$ 次类似的循环(一次乘法加一次减法)。对 $M \ll N$ 的高斯消去法(即仅有几个右端项的情况)要比 Gauss-Jordan 消去法好很多(但它不能像 Gauss-Jordan 消去法那样计算出逆矩阵)。

要计算矩阵的逆时(可视其为 $M=N$ 个右端项的情况,即单位阵的列看做 N 个单位向量),马上可以看出,高斯消去法和回代法需要$\frac{1}{3}N^3$(矩阵化简)+$\frac{1}{2}N^3$(右端项处理)+$\frac{1}{2}N^3$(N 次回代)=$\frac{4}{3}N^3$ 次循环操作,这比 Gauss-Jordan 消去法的 N^3 次要多。然而,单位向量是很特殊的,除了一个元素是 1 外,其余全是 0 元素。如果将此考虑在内,右端项处理可减为仅$\frac{1}{6}N^3$ 次循环操作,故对求矩阵的逆来说,这两种方法的效率是相同的。

高斯消去法和 Gauss-Jordan 消去法都有一个缺点,即右端项必须提前知道。下一节中要讨论的 LU 分解法就没有这个缺点。不管是对有任意个右端项的求解问题还是求矩阵的逆,这种方法的运算次数都较少。由于这个原因,就不给出高斯消去法的实现程序了。

参考文献和深入阅读

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §9.3–1.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), §2.1.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §2.2.1.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

2.3　LU 分解法及其应用

假定能够把矩阵 **A** 写成两个矩阵相乘的形式

$$\mathbf{L}\cdot\mathbf{U}=\mathbf{A} \tag{2.3.1}$$

其中 **L** 为**下三角矩阵**(仅在对角线上及其下面有元素),**U** 为**上三角矩阵**(仅在对角线上及其上面有元素)。举个例子,对 4×4 矩阵 **A** 的情况,(2.3.1)式如下:

$$\begin{bmatrix}\alpha_{00}&0&0&0\\\alpha_{10}&\alpha_{11}&0&0\\\alpha_{20}&\alpha_{21}&\alpha_{22}&0\\\alpha_{30}&\alpha_{31}&\alpha_{32}&\alpha_{33}\end{bmatrix}\cdot\begin{bmatrix}\beta_{00}&\beta_{01}&\beta_{02}&\beta_{03}\\0&\beta_{11}&\beta_{12}&\beta_{13}\\0&0&\beta_{22}&\beta_{23}\\0&0&0&\beta_{33}\end{bmatrix}=\begin{bmatrix}a_{00}&a_{01}&a_{02}&a_{03}\\a_{10}&a_{11}&a_{12}&a_{13}\\a_{20}&a_{21}&a_{22}&a_{23}\\a_{30}&a_{31}&a_{32}&a_{33}\end{bmatrix} \tag{2.3.2}$$

可以用如(2.3.1)式的分解来求解线性方程组

$$\mathbf{A}\cdot\mathbf{x}=(\mathbf{L}\cdot\mathbf{U})\cdot\mathbf{x}=\mathbf{L}\cdot(\mathbf{U}\cdot\mathbf{x})=\mathbf{b} \tag{2.3.3}$$

首先求解向量 **y** 使得

$$\mathbf{L}\cdot\mathbf{y}=\mathbf{b} \tag{2.3.4}$$

然后再来求解

$$\mathbf{U}\cdot\mathbf{x}=\mathbf{y} \tag{2.3.5}$$

把一个线性方程组拆成两个连续的方程组有什么优点呢?优点在于求解一个三角形的方程组相当容易,正如在 2.2 节的(2.2.4)式中见到的那样。这样,(2.3.4)式可用**向前替代**过程求解,如下:

$$\begin{aligned}y_0&=\frac{b_0}{\alpha_{00}}\\y_i&=\frac{1}{\alpha_{ii}}\left[b_i-\sum_{j=0}^{i-1}\alpha_{ij}y_j\right]\qquad i=1,2,\ldots,N-1\end{aligned} \tag{2.3.6}$$

而(2.3.5)式可用回代过程求解,这与(2.2.2)式~(2.2.4)式一样,

$$x_{N-1} = \frac{y_{N-1}}{\beta_{N-1,N-1}}$$
$$x_i = \frac{1}{\beta_{ii}}\left[y_i - \sum_{j=i+1}^{N-1}\beta_{ij}x_j\right] \qquad i = N-2, N-3, \ldots, 0 \tag{2.3.7}$$

(2.3.6)式和(2.3.7)式共需执行 N^2 次内层循环(对每个右端项 **b**),每个内层循环包括一次乘法和一次加法。如果有 N 个右端项,它们是单位列向量(在求矩阵逆时就是这种情况),考虑这些零元素可把(2.3.6)式的总执行次数从$\frac{1}{2}N^3$ 减少到$\frac{1}{6}N^3$,而(2.3.7)式的执行次数不变,仍为$\frac{1}{2}N^3$。

注意,一旦对 **A** 进行了 LU 分解,就可以一次求解所有要解的右端项。与 2.1 节和 2.2 节中的方法相比,这是一个显著的优点。

2.3.1 进行 LU 分解

对给定的 **A** 怎样才能求得 **L** 和 **U** 呢?首先,写出(2.3.1)式或(2.3.2)式的第 i,j 分量。它总是一个和的形式,开始部分形式如下:

$$\alpha_{i0}\beta_{0j} + \cdots = a_{ij}$$

和式中的项数依赖于 i 和 j 中较小的数。事实上有三种形式,

$$i < j: \qquad \alpha_{i0}\beta_{0j} + \alpha_{i1}\beta_{1j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{2.3.8}$$
$$i = j: \qquad \alpha_{i0}\beta_{0j} + \alpha_{i1}\beta_{1j} + \cdots + \alpha_{ii}\beta_{jj} = a_{ij} \tag{2.3.9}$$
$$i > j: \qquad \alpha_{i0}\beta_{0j} + \alpha_{i1}\beta_{1j} + \cdots + \alpha_{ij}\beta_{jj} = a_{ij} \tag{2.3.10}$$

(2.3.8)式~(2.3.10)式共有 N^2 个方程,而要求 N^2+N 个未知的α和 β(因对角线的未知元素有两套)。既然未知数的个数比方程个数多,就人为地指定 N 个未知数,然后再来求解其他的未知数。事实上,总是令

$$\alpha_{ii} \equiv 1 \qquad i = 0, \ldots, N-1 \tag{2.3.11}$$

有一个算法称为 **Crout 算法**,它仅按某种次序排列方程,就能相当容易地求出(2.3.8)式~(2.3.11)式的 N^2+N 个方程中的所有α和 β。步骤如下:

- 设$\alpha_{ii}=1, i=0,\ldots,N-1$((2.3.11)式)。
- 对每个 $j=0,1,2,\ldots,N-1$ 进行以下两步:第一步,对每个 $i=0,1,\ldots,j$ 用(2.3.8)式、(2.3.9)式和(2.3.11)式来解 β_{ij},即

$$\beta_{ij} = a_{ij} - \sum_{k=0}^{i-1}\alpha_{ik}\beta_{kj} \tag{2.3.12}$$

(当(2.3.12)式中 $i=0$ 时,求和项为零。)第二步,对每个 $i=j+1,j+2,\ldots,N-1$ 用式(2.3.10)来求解α_{ij},即

$$\alpha_{ij} = \frac{1}{\beta_{jj}}\left(a_{ij} - \sum_{k=0}^{j-1}\alpha_{ik}\beta_{kj}\right) \tag{2.3.13}$$

在求解下一个 j 之前要保证进行了以上两步。

如果按上述过程进行几次迭代后,就会发现(2.3.12)式和(2.3.13)式右端的α和 β 在需要时已经知道了。还会看到,每一个 a_{ij} 仅被使用一次便不再用。这意味着相应的α_{ij}或β_{ij}可以存储在 a 所占的存储空间,即分解是"同址"进行的。对角线上的单位元素α_{ii},见(2.3.11)式,并不存储。简而言之,Crout 算法得到的是α和 β 的混合矩阵,其排列是

$$\begin{bmatrix} \beta_{00} & \beta_{01} & \beta_{02} & \beta_{03} \\ \alpha_{10} & \beta_{11} & \beta_{12} & \beta_{13} \\ \alpha_{20} & \alpha_{21} & \beta_{22} & \beta_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \beta_{33} \end{bmatrix} \tag{2.3.14}$$

列从左向右,每一列从上到下(参见图 2.3.1)。

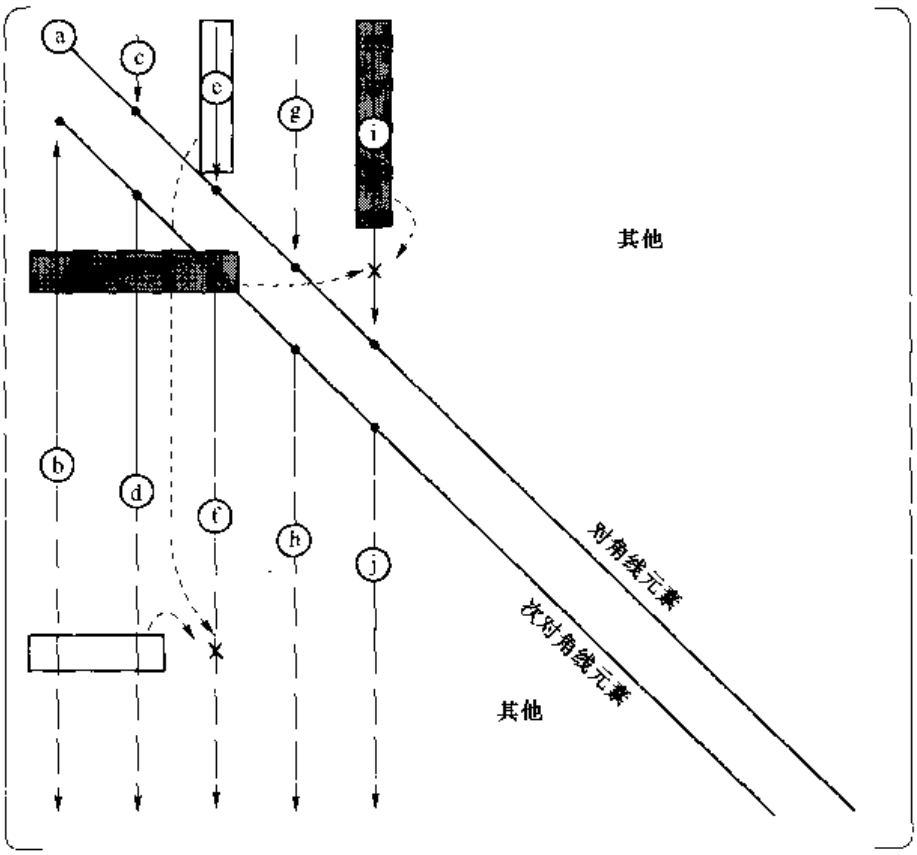


图 2.3.1　Crout 算法对矩阵进行 LU 分解。原始矩阵的元素按小写字母的顺序被修改：a, b, c, 等等。带阴影的矩形方块表示那些已被修改的元素，它们被用来修改标有"×"记号的两个典型元素

选主元会怎样呢？选主元(即选择一个合适的主元素去除(2.3.13)式的两边)，对 Crout 方法的稳定性是绝对必要的。只有部分主元法(行交换)可以有效地实现。然而，这已足够使算法稳定了。顺便提一下，这意味着并不是把矩阵 **A** 分解成为 LU 的形式，而是将其按行置换的方式分解(如果保存这个置换的次序，该分解就与原来的分解一样有用)。

Crout 算法中的主元法有个精妙之处。关键是要注意(2.3.12)式，在 $i=j$(最后一次应用)时，与(2.3.13)式(除了后者还要做一次除法外)是完全一样的，这两种情况求和的上限都是 $k=j-1(=i-1)$。这意味着，不必费心去考虑对角线元素 β_{jj} 是否会正落在对角线上，也不必考虑该列中，它下面的某个元素(未做除法的) $\alpha_{ij}, i=j+1,\ldots,N-1$ 是否会"提升"成为对角线元素 β。当所有该列元素都求得后，这点便会证实。现在应该能够想到，将选取最大的元素作为对角线元素 β(主元素)，然后用它去除其他元素。这便是**部分主元法的 Crout 方法**。实现方法中还有一个精妙之处：它首先找每行的最大元素，而后(在找最大主元素时)乘以一个比例系数，就像最初对所有方程进行比例变换，使它们的最大系数成为单位元素那样。这就是 2.1 节中提到过的"隐式主元法"。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::ludcmp(Mat_IO_DP &a, Vec_O_INT &indx, DP &d)
```

给定矩阵 a[0..n-1][0..n-1]，本程序对其进行按行置换的 LU 分解，结果存在该矩阵中。a 为输入矩阵，同时作为输出结果，其元素的排列形式与上面的(2.3.14)式一样。indx[0..n-1]为输出向量，它用来记录因部分主元法而改变了的行排列次序。输出变量 d 的值为 ±1，相应地表示行交换次数为偶数还是奇数。本程序跟程序 lubksb 一起用来求解线性方程组或者求矩阵的逆。

```
{
    const DP TINY=1.0e-20;          一个小数。
    int i,imax,j,k;
    DP big,dum,sum,temp;

    int n=a.nrows();
    Vec_DP vv(n);                   vv用于保存每行的内含比例因子。
    d=1.0;                          还未进行交换。
    for (i=0;i<n;i++) {             按行循环，求内含的比例因子。
        big=0.0;
        for (j=0;j<n;j++)
            if ((temp=fabs(a[i][j])) > big) big=temp;
        if (big == 0.0) nrerror("Singular matrix in routine ludcmp");
        No nonzero largest element
        vv[i]=1.0/big;              保存比例因子。
    }
    for (j=0;j<n;j++) {             Grout方法的列循环。
        for (i=0;i<j;i++) {         (2.3.12)式中i = j的情况。
            sum=a[i][j];
            for (k=0;k<i;k++) sum -= a[i][k]*a[k][j];
            a[i][j]=sum;
        }
        big=0.0;                    初始化最大主元素变量。
        for (i=j;i<n;i++) {         (2.3.12) 式中i = j的情况，以及式(2.3.13)中
            sum=a[i][j];            i = j + 1 ... N-1的情况。
                for (k=0;k<j;k++) sum -= a[i][k]*a[k][j];
                a[i][j]=sum;
                if ((dum=vv[i]*fabs(sum)) >= big) {
                新的候选主元是否比已选中的更合适？
                    big=dum;
                    imax=i;
                }
            }
            if (j != imax) {            是否需要进行交换？
                for (k=0;k<n;k++) {     是的，进行交换……
                    dum=a[imax][k];
                    a[imax][k]=a[j][k];
                    a[j][k]=dum;
                }
                d = -d;                 ……改变d的奇偶性。
                vv[imax]=vv[j];         还要交换比例因子。
            }
            indx[j]=imax;
            if (a[j][j] == 0.0) a[j][j]=TINY;
            如果主元素为零则矩阵是奇异的（至少对本算法精度而言）。对奇异矩阵的情况需要
            把TINY换成零。
            if (j != n-1) {             最后除以主元素。
                dum=1.0/(a[j][j]);
                for (i=j+1;i<n;i++) a[i][j] *= dum;
            }
        }                               转回约化下一列。
    }
```

下面的程序用向前替代过程和回代过程来实现(2.3.6)式和(2.3.7)式。

```
#include "nr.h"

void NR::lubksb(Mat_I_DP &a, Vec_I_INT &indx, Vec_IO_DP &b)
求解n维线性方程组 A·X = B，其中a[0..n-1][0..n-1]为输入矩阵，它不是原始矩阵 A，而是经程序 ludcmp 确定后得
到的其 LU 分解阵。indx[0..n-1]为输入向量，是 ludcmp 返回的记录行排列顺序的向量。b[0..n-1]为输入右端项向量
B，返回解向量 X。a 和 indx 不被该程序修改，可留下来再被调用，以求不同的右端项 b 的解。这个程序考虑了 b 开始时
可能有很多零元素的情况，因而在求矩阵的逆时可提高效率。
{
    int i,ii=0,ip,j;
    DP sum;

    int n=a.nrows();
    for (i=0;i<n;i++) {             当ii为正数时，它表示b的第一个非零元的位置。
        ip=indx[i];                 现在来进行（2.3.6）式的向前替换。其惟一的
        sum=b[ip];                  缺点是没有恢复原来的排列顺序。
        b[ip]=b[i];
```

```
            if (ii != 0)
                for (j=ii-1;j<i;j++) sum -= a[i][j]*b[j];
            else if (sum != 0.0)     遇到非零元素，下面要进入上面的求和循环。
                ii=i+1;
            b[i]=sum;
        }
        for (i=n-1;i>=0;i--) {       现在来进行(2.3.7)式的回代。
            sum=b[i];
            for (j=i+1;j<n;j++) sum -= a[i][j]*b[j];
            b[i]=sum/a[i][i];        保存解向量X的一个分量。
        }                            程序结束。
    }
```

程序 ludcmp 中，LU 分解大约需要执行$\frac{1}{3}N^3$次内层循环(每次包括一次乘法和一次加法)。这是求解一个(或少量几个)右端项时的运算次数，它要比 2.1 节中的 Gauss-Jordan 程序 gaussj 快 3 倍，比不计算逆矩阵的 Gauss-Jordan 程序快 1.5 倍。当要求解逆矩阵时，总的运算次数(包括(2.3.7)式后面讨论的向前替代和回代部分)为$\left(\frac{1}{3}+\frac{1}{6}+\frac{1}{2}\right)N^3=N^3$，与 gaussj 相同。

总结一下，求解线性方程组 $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$ 的较好方法是

```
const int N = ...
Mat_DP a(N,N);
Vec_DP b(N);
Vec_INT indx(N);
DP d;
...
NR::ludcmp(a,indx,d);
NR::lubksb(a,indx,b);
```

解 x 将保存在 b 中，且原来的矩阵 **A** 会被破坏。

如果随后还想求解具有相同的 **A** 但右端项 **b** 不同的线性方程组，则仅重写：

```
NR::lubksb(a,indx,b);
```

当然矩阵 **A** 已非原来的形式，这里的 a 和 indx 已经由程序 ludcmp 确定了。

2.3.2　矩阵的求逆

使用上一节中的 LU 分解和回代过程的程序，能完全直接地按列来求矩阵的逆。

```
const int N = ...
Mat_DP a(N,N),y(N,N);
Vec_DP col(N);
Vec_INT indx(N);
DP d;
int i,j;
...
NR::ludcmp(a,indx,d);                仅把矩阵分解一次。
for(j=0;j<N;j++) {                   按列求逆。
    for(i=0;i<N;i++) col[i]=0.0;
    col[j]=1.0;
    NR::lubksb(a,indx,col);
    for(i=0;i<N;i++) y[i][j]=col[i];
}
```

矩阵 y 保存的是原始矩阵 a 的逆，矩阵 a 会被破坏。使用像 gaussj(参见 2.1 节)这样的 Gauss-Jordan 程序求矩阵的逆也是不错的选择，它同样也破坏原始矩阵。这两种方法的运算次数基本相同。

如果要根据矩阵 **A** 和 **B** 来计算 $\mathbf{A}^{-1}\cdot\mathbf{B}$，应当对 **A** 进行 LU 分解，然后用 **B** 的列进行回代，而不应用单位矩阵先求 **A** 的逆。这样可以省去整个矩阵的乘法运算，并且精度较高。

2.3.3　矩阵的行列式

经 LU 分解的矩阵的行列式仅为对角线上元素的积，

$$\det=\prod_{j=0}^{N-1}\beta_{jj} \tag{2.3.15}$$

回顾一下,并没有求原始矩阵的分解,而是求矩阵行交换后的分解。幸运的是,记录了行交换次数是奇数还是偶数,因此就可以在乘积前面给以相应的符号(现在可知在程序 ludcmp 中设置变量 d 的原因了)。

计算行列式需要调用一次 ludcmp,而不必接着调用 lubksb 进行回代。

```
const int N = ...
Mat_DP a(N,N);
Vec_INT indx(N);
DP d;
int j;
...
NR::ludcmp(a,indx,d);          返回值d为±1。
for(j=0;j<N;j++) d *= a[j][j];
```

d 中现在保存的即为原始矩阵 a 的行列式值,a 将会被破坏。

对实际出现的大矩阵,很可能出现行列式值上溢或下溢,超出了计算机所能表示的浮点数的范围。在这种情况下,可以修改上述程序中的循环部分,例如除以十的幂次并单独保存该比例因子,或求出各乘数绝对值的对数和并单独保存其符号。

2.3.4 深入讨论:复数系统方程

如果矩阵 **A** 是实数的,而右端向量是复数的,即 $\mathbf{b}+i\mathbf{d}$,则:(i)用通常的方法对 **A** 进行 LU 分解;(ii) 用 **b** 回代求得解向量的实部;(iii)用 **d** 回代求得解向量的虚部。

如果矩阵本身是复的,则要求解的方程为

$$(\mathbf{A}+i\mathbf{C})\cdot(\mathbf{x}+i\mathbf{y})=(\mathbf{b}+i\mathbf{d}) \tag{2.3.16}$$

有两种方法求解。最好的方法是重写 ludcmp 和 lubksb 为复数程序。在构造比例向量 vv 及寻找最大主元素时,用复数的模代替绝对值。其他部分都很显然,需要时就进行复数的算术运算。

快速但不太好的求解复数系统的方法是,将(2.3.16)式的实部和虚部分开,即

$$\begin{aligned}\mathbf{A}\cdot\mathbf{x}-\mathbf{C}\cdot\mathbf{y}&=\mathbf{b}\\ \mathbf{C}\cdot\mathbf{x}+\mathbf{A}\cdot\mathbf{y}&=\mathbf{d}\end{aligned} \tag{2.3.17}$$

它可以写成一个 $2N\times 2N$ 的实数方程,

$$\begin{pmatrix}\mathbf{A} & -\mathbf{C}\\ \mathbf{C} & \mathbf{A}\end{pmatrix}\cdot\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix}=\begin{pmatrix}\mathbf{b}\\ \mathbf{d}\end{pmatrix} \tag{2.3.18}$$

然后用 ludcmp 和 lubksb 来求解目前的形式。该方案在存储上占用两倍的空间,因为 **A** 和 **C** 都被存储了两次。另外,在运算上也要花两倍的时间,因为在用复数形式写成的程序中,复数乘法仅需 4 次实数相乘,而求解 $2N\times 2N$ 问题的工作量是求解 $N\times N$ 问题工作量的 8 倍。如果能容忍这两个因素,则(2.3.18)式也是一种很方便的求解办法。

参考文献和深入阅读

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), Chapter 4.

Anderson, E., et al. 2000, LAPACK User's Guide, 3rd ed. (Philadelphia: S.I.A.M.).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §3.3, and p. 50.

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 9, 16, and 18.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §9.11.

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press).

2.4　三对角及带状对角系统方程

线性方程系统的特例之一是**三对角**形式的，也就是说，非零的元素仅出现在对角线及其前、后一列的位置上，这种情况会经常出现。**带状对角**形式的系统也常见，其非零的元素仅出现在主对角线附近（上或下）的几个对角线中。

对三对角系统，LU 分解过程、前代或回代每次只需 $O(N)$ 次操作，整个求解过程可以非常简洁地编码。求解程序 tridag 在后面的章节中还要用到。

一般来说，不必把整个 $N \times N$ 矩阵全部保存起来，只需保存非零的元素，存为三个向量。要求解的方程为

$$\begin{bmatrix} b_0 & c_0 & 0 & \cdots & & & \\ a_1 & b_1 & c_1 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-2} & b_{N-2} & c_{N-2} \\ & & & \cdots & 0 & a_{N-1} & b_{N-1} \end{bmatrix} \cdot \begin{bmatrix} u_0 \\ u_1 \\ \cdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ \cdots \\ r_{N-2} \\ r_{N-1} \end{bmatrix} \tag{2.4.1}$$

注意 a_0 和 c_{N-1} 没有定义，下面的程序并不引用它们。

```
#include "nr.h"

void NR::tridag(Vec_I_DP &a, Vec_I_DP &b, Vec_I_DP &c, Vec_I_DP &r, Vec_O_DP &u)
求解向量 u[0..n-1]，三对角线性矩阵由(2.4.1)式给出。a[0..n-1]，b[0..n-1]，c[0..n-1]及 r[0..n-1]为输入向
量，并不进行修改。
{
    int j;
    DP bet;

    int n=a.size();
    Vec_DP gam(n);                                      gam需要一个向量的工作空间。
    if (b[0] == 0.0) nrerror("Error 1 in tridag");
    若这种情况发生，应重写方程，u1忽略不计，成为N-1维的分解及前代。
    u[0]=r[0]/(bet=b[0]);
    for (j=1;j<n;j++) {                                 分解及前代。
        gam[j]=c[j-1]/bet;
        bet=b[j]-a[j]*gam[j];
        if (bet == 0.0) nrerror("Error 2 in tridag"); 算法失败，见下面的解释。
        u[j]=(r[j]-a[j]*u[j-1])/bet;
    }
    for (j=(n-2);j>=0;j--)
        u[j] -= gam[j+1]*u[j+1];                        回代。
}
```

在 tridag 中没有选主元。正是这个原因，即使矩阵是非奇异的，tridag 也可能失败，因为有些非奇异矩阵是可能遇到零主元的。在实践中，不必为此过于担心。那些能产生三对角线性系统的问题通常都有附加的属性，能保证算法 tridag 成功。举个例子，如果

$$|b_j| > |a_j| + |c_j| \qquad j = 0, \ldots, N-1 \tag{2.4.2}$$

（称为**对角占优**）则可以证明算法不会遇到零主元。

可以构造出特殊的例子，来说明由于算法没有选主元而导致数值不稳定。然而，实践中这种不稳定几乎从未遇到过，不像一般的矩阵问题中选主元是必不可少的。

三对角算法是算法中的一种少有情况，实践中其稳健性要比理论上强得多。当然，一旦遇到 tridag 算法失败的问题，则可换用更一般的方法，即求解带状对角系统的方法，见下面小节的讨论（程序 bandec 和 banbks）。

一些其他的矩阵形式包含了三对角及少数几个附加的非零元素（如右上角或左下角），也可以进行快速求解，参见 2.7 节。

2.4.1　深入讨论：带状对角系统

三对角系统的非零元素仅出现在对角线及前、后一个元素的位置上，而带状对角系统紧靠在对角线左边（下边）有（例如）$m_1 \geq 0$ 个非零元素，紧靠其右边（上边）有 $m_2 \geq 0$ 个非零元素。当然，这仅在 m_1 和 m_2 都 $\ll N$ 时才有意义。在这种情况下，用 LU 分解求解线性系统可以比通用 $N \times N$ 的情况完成得更快，占用空间更少。

用元素 a_{ij} 精确定义带状对角矩阵,表示为

$$a_{ij}=0 \quad 当 \quad j>i+m_2 \quad 或 \quad i>j+m_1 \tag{2.4.3}$$

带状对角矩阵用一种称为压缩形式的方式进行存储及操作,把矩阵顺时针旋转 45°,使非零元素排成一个细长的矩阵,它有 m_1+1+m_2 列和 N 行。最好用示例解释一下。

带状对角矩阵:

$$\begin{pmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 5 & 0 & 0 & 0 & 0 \\ 9 & 2 & 6 & 5 & 0 & 0 & 0 \\ 0 & 3 & 5 & 8 & 9 & 0 & 0 \\ 0 & 0 & 7 & 9 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 8 & 4 & 6 \\ 0 & 0 & 0 & 0 & 2 & 4 & 4 \end{pmatrix} \tag{2.4.4}$$

其中 $N=7, m_1=2, m_2=1$,存储的压缩形式为 7×4 矩阵

$$\begin{pmatrix} x & x & 3 & 1 \\ x & 4 & 1 & 5 \\ 9 & 2 & 6 & 5 \\ 3 & 5 & 8 & 9 \\ 7 & 9 & 3 & 2 \\ 3 & 8 & 4 & 6 \\ 2 & 4 & 4 & x \end{pmatrix} \tag{2.4.5}$$

其中 x 表示压缩格式中浪费的空间,它们在运算中不被引用,因而其值可随意。注意,原矩阵对角线元素出现在第 m_1 列中,对角线左下的元素在其左侧,对角线右上的元素在其右侧。

压缩存储的带状对角矩阵最简单的运算是乘以其右边的向量。尽管这是算法上的小问题,但可以仔细研究一下下面的程序,查看如何把压缩存储格式的非零元素 a_{ij} 变成顺序的方式。

```
#include "nr.h"

void NR::banmul(Mat_I_DP &a, const int m1, const int m2, Vec_I_DP &x,
    Vec_O_DP &b)
矩阵乘法 b=A·x,其中 A 是带状对角阵,对角线下有 m1 行,对角线上有 m2 行。输入向量 x 及输出向量 b 相应存为
x[0..n-1]和 b[0..n-1]。数组 a[0..n-1][0..m1+m2]存储 A:对角线元素在 a[0..n-1][m1]中。对角线左下的元素
在数组a[j..n-1][0..m1-1]中(j>0,对应于每个对角线左下元素的个数)。对角线右上元素在 a[0..j][m1+1..m1+
m2]中(j<n-1,对应于每个对角线右上元素的个数)。
{
    int i,j,k,tmploop;

    int n=a.nrows();
    for (i=0;i<n;i++) {
        k=i-m1;
        tmploop=MIN(m1+m2+1,int(n-k));
        b[i]=0.0;
        for (j=MAX(0,-k);j<tmploop;j++) b[i] += a[i][j]*x[j+k];
    }
}
```

不可能把一个带状对角矩阵 A 的 LU 分解也像其压缩形式本身一样紧凑地存储起来。分解(主要是用 Crout 方法,参见 2.3 节)产生了附加的非零元素填入。一种直接的存储方案是,把上三角因子(U)返回到 A 以前占有的相同的空间中,把下三角因子(L)返回到单独的 $N\times m_1$ 压缩矩阵中。U 的对角线元素(它们的乘积,乘上 $d=\pm 1$,便是行列式的值)被存放在 A 的存储空间的第一列。

下面的程序 bandec 类似于 2.3 节中的 ludcmp,不过是用来解带状对角系统的。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::bandec(Mat_IO_DP &a, const int m1, const int m2, Mat_O_DP &al,
    Vec_O_INT &indx, DP &d)
给定 n×n 带状对角矩阵 A,其对角线左下有 m1 行,对角线右上有 m2 行,被压缩存储在数组 a[0..n-1][0..m1+m2]
中,见程序 banmul 的说明部分。本程序构造一个 A 按行排列的 LU 分解。上三角矩阵替代 a,下三角矩阵返回到
al[0..n-1][0..m1-1]中,indx[0..n-1]是输出向量,记录部分主元法影响的行排列情况,d 输出值为±1,由行交换次数
的奇偶确定。本程序与 banbks 一起使用,求解带状对角方程组。
```

```
{
    const DP TINY=1.0e-20;
    int i,j,k,l,mm;
    DP dum;

    int n=a.nrows();
    mm=m1+m2+1;
    l=m1;
    for (i=0;i<m1;i++) {                    重新安排存储。
        for (j=m1-i;j<mm;j++) a[i][j-l]=a[i][j];
        l--;
        for (j=mm-l-1;j<mm;j++) a[i][j]=0.0;
    }
    d=1.0;
    l=m1;
    for (k=0;k<n;k++) {                     对每一行……
        dum=a[k][0];
        i=k;
        if (l<n) l++;
        for (j=k+1;j<l;j++) {               寻找主元素。
            if (fabs(a[j][0]) > fabs(dum)) {
                dum=a[j][0];
                i=j;
            }
        }
        indx[k]=i+1;
        if (dum == 0.0) a[k][0]=TINY;
        矩阵算法上是奇异的，但仍用TINY作为主元继续进行(在某些应用时需要这样做)。
        if (i != k) {                       行交换。
            d = -d;
            for (j=0;j<mm;j++) SWAP(a[k][j],a[i][j]);
        }
        for (i=k+1;i<l;i++) {               进行消元。
            dum=a[i][0]/a[k][0];
            al[k][i-k-1]=dum;
            for (j=1;j<mm;j++) a[i][j-1]=a[i][j]-dum*a[k][j];
            a[i][mm-1]=0.0;
        }
    }
}
```

在 bandec 存储限制允许范围内，有时候选主元是可能的，上面的程序利用了这种可能性。一般情况下，当返回 TINY 作为 0 的对角元时，则原来的矩阵(可能由于舍入误差的原因)事实上是奇异的。考虑到这点，bandec 某种程度上要比上面的 tridag 更稳健，因为 tridag 即使对非奇异的矩阵也可能导致算法上的失败。bandec 对某些效果不好的三对角系统也是有效的(取 $m_1 = m_2 = 1$)。

一旦矩阵 **A** 被分解，任意数目的右端项都可以通过重复调用 banbks 来求解，回代程序与 2.3 节中的 lubksb 类似。

```
#include "nr.h"

void NR::banbks(Mat_I_DP &a, const int m1, const int m2, Mat_I_DP &al,
    Vec_I_INT &indx, Vec_IO_DP &b)
对给定作为 bandec 返回值的数组 a,al 和 index，及给定的右端项向量 b[0..n-1]，求解带状对角线性方程 A·x = b。解向
量 x 覆盖写入 b[0..n-1]。不修改其他的输入数组，并且可用于以后不同右端项的连续调用。
{
    int i,j,k,l,mm;
    DP dum;

    int n=a.nrows();
    mm=m1+m2+1;
    l=m1;
    for (k=0;k<n;k++) {                     前代，不打乱行的排列顺序。
        j=indx[k]-1;
        if (j!=k) SWAP(b[k],b[j]);
        if (l<n) l++;
        for (j=k+1;j<l;j++) b[j] -= al[k][j-k-1]*b[k];
    }
```

```
        l=1;
        for (i=n-1;i>=0;i--) {          回代。
            dum=b[i];
            for (k=1;k<l;k++) dum -= a[i][k]*b[k+i];
            b[i]=dum/a[i][0];
            if (l<mm) l++;
        }
    }
```

程序 bandec 和 banbks 基于参考书目[1]中的程序 bandetl 和 bansolt。

参考文献和深入阅读

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems*; reprinted 1991 (New York: Dover), p. 74.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Example 5.4.3, p. 166.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §9.11.

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/6. [1]

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §4.3.

2.5 线性方程组解的迭代改进

显然,很难使线性方程组的解的精度超过计算机浮点数精度。遗憾的是,对于大型线性方程组,要想达到甚至是接近计算机所限定的精度也总是很难的。在直接求解的方法中,舍入误差的积累达到一定程度时,矩阵就接近奇异了。对于远非奇异的矩阵(认为如此),也很容易丢失两个或三个有效数字。

如果发生这种情况,有一个简单的方法能恢复到整个机器精度,称为解的**迭代改进**。这在理论上是很直接的(参见图 2.5.1)。

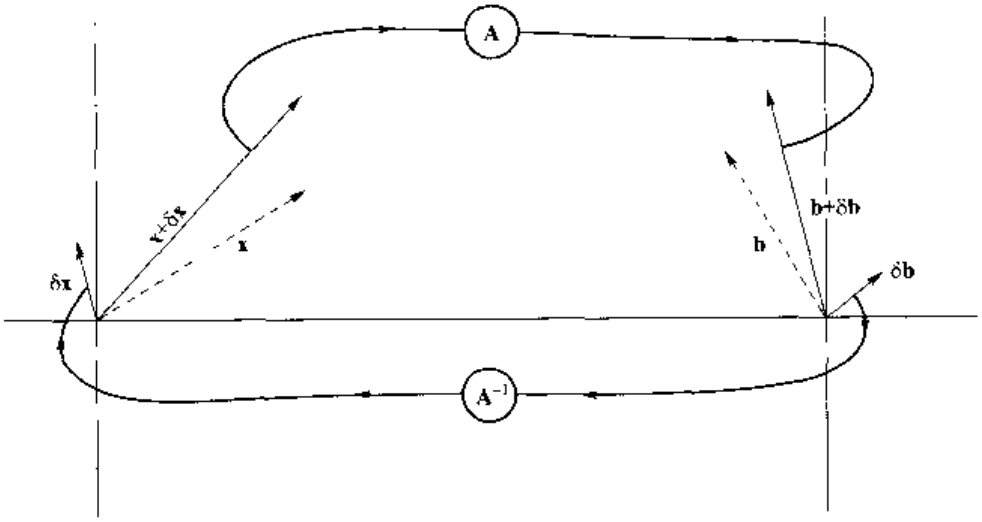


图 2.5.1 $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$ 的解的迭代改进。首先假定 $\mathbf{A}$ 与 $\mathbf{x}+\delta\mathbf{x}$ 相乘的积为 $\mathbf{b}+\delta\mathbf{b}$,减去已知向量 $\mathbf{b}$ 得 $\delta\mathbf{b}$。再以此为右端项反过来求解该线性方程组,得 $\delta\mathbf{x}$,将其从开始假定的解中减去即得改进后的解 $\mathbf{x}$

假定向量 $\mathbf{x}$ 是线性方程组

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{2.5.1}$$

的精确解,然而并不知道解 $\mathbf{x}$,而仅仅知道一些略有误差的解 $\mathbf{x}+\delta\mathbf{x}$,其中 $\delta\mathbf{x}$ 为未知的误差。当将其乘以矩阵 $\mathbf{A}$ 时,这个有误差的解使乘积与原来的右端项 $\mathbf{b}$ 也存在偏差,即

$$\mathbf{A}\cdot(\mathbf{x}+\delta\mathbf{x})=\mathbf{b}+\delta\mathbf{b} \tag{2.5.2}$$

(2.5.2)式减去(2.5.1)式,得

$$\mathbf{A}\cdot\delta\mathbf{x}=\delta\mathbf{b} \tag{2.5.3}$$

但由(2.5.2)式也可求得 $\delta\mathbf{b}$,将其代入(2.5.3)式中得

$$\mathbf{A}\cdot\delta\mathbf{x}=\mathbf{A}\cdot(\mathbf{x}+\delta\mathbf{x})-\mathbf{b} \tag{2.5.4}$$

该方程中整个右端项是已知量,因为 $\mathbf{x}+\delta\mathbf{x}$ 是有待改进的有误差的解。用双精度数计算右端项是个好方法,因为在减去 $\mathbf{b}$ 的过程中会产生很多抵消。接下来只需求解(2.5.4)式的偏差量 $\delta\mathbf{x}$,将其从有误差的解中减去便得到改进后的解。

如果是通过 LU 分解得到最初的解,则这么做还有一个很大的优点。因为这种情况下,已经有了矩阵 $\mathbf{A}$ 的 LU 分解形式,要求解(2.5.4)式,只需计算右端项再进行回代即可。

完成上述功能的代码既简洁又直接:

```
#include "nr.h"

void NR::mprove(Mat_I_DP &a, Mat_I_DP &alud, Vec_I_INT &indx, Vec_I_DP &b,
    Vec_IO_DP &x)
改进线性方程组 A·X = B 的解向量 x[0..n-1]。矩阵 a[0..n-1][0..n-1]以及向量 b[0..n-1]和 x[0..n-1]均为输
入变量。alud[0..n-1][0..n-1]也为输入变量,它是 ludcmp 返回的 LU 分解形式,向量 indx[0..n-1]也由该程序返回。
输出变量只有 x[0..n-1]被修改,它是改进的解集。
{
    int i,j;

    int n=x.size();
    Vec_DP r(n);
    for (i=0;i<n;i++) {                    计算右端项, 用双精度数表示差值。
        long double sdp = -b[i];
        for (j=0;j<n;j++)
            sdp += (long double) a[i][j]*(long double) x[j];
        r[i]=sdp;
    }
    lubksb(alud,indx,r);                   求解误差量。
    for (i=0;i<n;i++) x[i] -= r[i];        将其从原来的解中减去。
}
```

应当注意 2.3 节中的程序 ludcmp 破坏了输入矩阵,将其进行了 LU 分解。因为迭代改进既需要原来的矩阵又需要其 LU 分解的形式,在调用 ludcmp 之前需要复制 $\mathbf{A}$。同理 lubksb 破坏了 $\mathbf{b}$ 而得到 $\mathbf{x}$,所以要复制 $\mathbf{b}$。如果不在意这些额外增加的存储空间,则推荐迭代改进法,其运算量仅为 N^2 级(向量与矩阵相乘及回代,见(2.3.7)式下面的讨论)。

如果愿意,可以连续调用 mprove 几次。除非最初的解与真解相差太远,一般来说一次调用就足够了。但也可以再调用一次来证实其收敛性。

2.5.1　深入讨论:关于解的迭代改进的更多讨论

对(2.5.4)式进行更可靠的分析是具有启发意义的(而且本书后面还要用到),并且还能得到另外一些结果。在前面的讨论中,已指出解向量 $\mathbf{x}+\delta\mathbf{x}$ 有一个错误的项,但我们却忽略了一个事实,即 $\mathbf{A}$ 的 LU 分解本身并不精确。

一种不同的分析方法由矩阵 $\mathbf{B}_0$ 开始,假定它是矩阵 $\mathbf{A}$ 的一个近似的逆,则 $\mathbf{B}_0\cdot\mathbf{A}$ 近似等于单位矩阵 $\mathbf{1}$。定义 $\mathbf{B}_0$ 的**余量矩阵** $\mathbf{R}$ 为

$$\mathbf{R}\equiv\mathbf{1}-\mathbf{B}_0\cdot\mathbf{A} \tag{2.5.5}$$

假定其值都"很小"(下面会给出精确的说明)。因此

$$\mathbf{B}_0\cdot\mathbf{A}=\mathbf{1}-\mathbf{R} \tag{2.5.6}$$

接下来考虑下面的标准运算:

$$\begin{aligned}\mathbf{A}^{-1}&=\mathbf{A}^{-1}\cdot(\mathbf{B}_0^{-1}\cdot\mathbf{B}_0)=(\mathbf{A}^{-1}\cdot\mathbf{B}_0^{-1})\cdot\mathbf{B}_0=(\mathbf{B}_0\cdot\mathbf{A})^{-1}\cdot\mathbf{B}_0\\&=(\mathbf{1}-\mathbf{R})^{-1}\cdot\mathbf{B}_0=(\mathbf{1}+\mathbf{R}+\mathbf{R}^2+\mathbf{R}^3+\cdots)\cdot\mathbf{B}_0\end{aligned} \tag{2.5.7}$$

可以将最后一个表达式的前 n 项部分和定义为

$$\mathbf{B}_n \equiv (\mathbf{1} + \mathbf{R} + \cdots + \mathbf{R}^n) \cdot \mathbf{B}_0 \tag{2.5.8}$$

因此如果极限存在,则 $\mathbf{B}_\infty \to \mathbf{A}^{-1}$。

现在,直接验证(2.5.8)式满足一些递归关系。考虑到要求解方程组 $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$,其中 $\mathbf{x}$ 和 $\mathbf{b}$ 都是向量,定义

$$\mathbf{x}_n \equiv \mathbf{B}_n \cdot \mathbf{b} \tag{2.5.9}$$

则很容易得到

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{B}_0 \cdot (\mathbf{b} - \mathbf{A} \cdot \mathbf{x}_n) \tag{2.5.10}$$

立刻可以看出,它跟(2.5.4)式一样,其中 $-\delta\mathbf{x} = \mathbf{x}_{n+1} - \mathbf{x}_n$,$\mathbf{B}_0$ 代替了 $\mathbf{A}^{-1}$。因此,看到式(2.5.4)并不需要 $\mathbf{A}$ 的 LU 分解有多么精确,而只要求余量矩阵 $\mathbf{R}$ 很小即可。粗略而言,如果该余量小于计算机舍入误差的平方根,则使用一次(2.5.10)式之后(即由 $\mathbf{x}_0 = \mathbf{B}_0 \cdot \mathbf{b}$ 得到 $\mathbf{x}_1$),第一次忽略的项为 $\mathbf{R}^2$ 数量级,也就小于舍入误差了。而且,(2.5.10)式和(2.5.4)式一样可以使用多次,因为它仅仅使用了 $\mathbf{B}_0$,而没用更高阶的 $\mathbf{B}$。

由(2.5.8)式可以得到一个更令人吃惊的递推式,它每次递推的阶 n 要超过二倍:

$$\mathbf{B}_{2n+1} = 2\mathbf{B}_n - \mathbf{B}_n \cdot \mathbf{A} \cdot \mathbf{B}_n \qquad n = 0, 1, 3, 7, \ldots \tag{2.5.11}$$

从一个合适的矩阵 $\mathbf{B}_0$ 开始,重复使用(2.5.11)式,可以二次地收敛于未知的逆矩阵 $\mathbf{A}^{-1}$(**二次地**的定义参见 9.4 节)。(2.5.11)式有许多不同的名字,包括 **Schultz 方法**和 **Hotelling 方法**,参见 Pan and Reif [1]。事实上,(2.5.11)式只不过是求根的迭代 Newton-Raphson 方法应用于矩阵求逆罢了。

然后,应注意到(2.5.11)式,每次迭代它要进行两次完全的矩阵相乘。每次矩阵相乘包括 N^3 次加法和乘法。但是,在 2.1 节~2.3 节的 toto 中,已经知道直接求 $\mathbf{A}$ 的逆也只需要 N^3 次加法和 N^3 次乘法。因此(2.5.11)式仅在其运算比一般矩阵求解方法快时才有意义。这些情况将在 13.10 节中讨论。

我们来研究一下两个相关的问题:何时(2.5.7)式的级数收敛;什么样的初始假设 $\mathbf{B}_0$ 合适(例如,如果初始的 LU 分解不可行)。

矩阵的"范数"(norm)定义为对一个向量长度放大的最大倍数,

$$\|\mathbf{R}\| \equiv \max_{\mathbf{v} \neq 0} \frac{|\mathbf{R} \cdot \mathbf{v}|}{|\mathbf{v}|} \tag{2.5.12}$$

如果使(2.5.7)式取任意的右端项 $\mathbf{b}$,即需要矩阵的逆时,显然收敛的充分条件是

$$\|\mathbf{R}\| < 1 \tag{2.5.13}$$

Pan and Reif[1]中指出,一个合适的 $\mathbf{B}_0$ 的初始猜测是,充分小的常量 ϵ 乘以 $\mathbf{A}$ 的转置,即

$$\mathbf{B}_0 = \epsilon \mathbf{A}^T \qquad 或 \qquad \mathbf{R} = \mathbf{1} - \epsilon \mathbf{A}^T \cdot \mathbf{A} \tag{2.5.14}$$

要清楚为什么这里使用了第 11 章的概念;我们仅给出最简单的介绍:$\mathbf{A}^T \cdot \mathbf{A}$ 是对称的正定矩阵,故它有正实数的特征值。用其对角线表示,$\mathbf{R}$ 采用的形式为

$$\mathbf{R} = \mathrm{diag}(1 - \epsilon\lambda_0, 1 - \epsilon\lambda_1, \ldots, 1 - \epsilon\lambda_{N-1}) \tag{2.5.15}$$

其中所有的 λ_i 均为正数。显然对任何 ϵ,满足 $0 < \epsilon < 2/(\max_i \lambda_i)$ 都将得到 $\|\mathbf{R}\|_2 < 1$。不难发现,ϵ 的最优选择将使(2.5.11)式最快收敛,它是

$$\epsilon = 2/(\max_i \lambda_i + \min_i \lambda_i) \tag{2.5.16}$$

人们很少知道(2.5.16)式中 $\mathbf{A}^T \cdot \mathbf{A}$ 的特征值。Pan and Reif[1]中提供了几个可以由 $\mathbf{A}$ 直接计算出来的界。下面的选择保证了 $n \to \infty$ 时 $\mathbf{B}_n$ 的收敛性,

$$\epsilon \leq 1 \Big/ \sum_{j,k} a_{jk}^2 \qquad 或 \qquad \epsilon \leq 1 \Big/ \left(\max_i \sum_j |a_{ij}| \times \max_j \sum_i |a_{ij}| \right) \tag{2.5.17}$$

第二个表达式值得注意,它由 Pan 和 Reif 推得。他们注意到,(2.5.12)式中向量的范数不必是通常的 L_2 范数,但可被 L_∞(最大值)范数或是 L_1(绝对值)范数所代替。详见他们的著作。

另外一种方法就是,通过在 N 维空间中随机选取的方向上,选几个单位向量 $\mathbf{v}_i$ 来计算 $s_i \equiv |\mathbf{A} \cdot \mathbf{v}_i|^2$,用统计的方法来估算最大的特征值。最大的特征值 λ 可以被 $2 \max s_i$ 及 $2N\mathrm{Var}(s_i)/\mu(s_i)$ 的最大值所限定,其中 Var 和 μ 表示样本的方差和均值。

参考文献和深入阅读

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §2.3.4, p. 55.

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §3.5.3.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §5.5.6, p. 183.

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 13.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §9.5, p. 437.

Pan, V., and Reif, J. 1985, in Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (New York: Association for Computing Machinery). [1]

2.6　奇异值分解

有一套非常有用的技术，用来处理不论是奇异的，还是数值上非常接近奇异的方程组或矩阵。在许多情况下，高斯消去法和 LU 分解法都不能给出满意的结果，这套称为**奇异值分解**或 **SVD** 的技术，会精确地告诉我们什么样的情况属于该类问题。某些情况下，SVD 不仅可以判断该问题，还可以求解它，向读者提供一个有用的数值解，尽管将会发现这个解也不一定是我们想要的。

SVD 也可用来求解大多数的**线性最小二乘**问题，本节简要地介绍一下相关理论，详细讨论见第 15 章，其中的专题即为数据模型法。

SVD 技术基于下面的线性代数定理，其证明超出了我们的讨论范围：任意 $M \times N$ 的矩阵 $\mathbf{A}$，其行数 M 大于或等于列数 N，可以写成一个 $M \times N$ 的列正交矩阵 $\mathbf{U}$，一个 $N \times N$ 的元素均为正数或零（**奇异值**）的对角阵 $\mathbf{W}$，以及一个 $N \times N$ 正交阵 $\mathbf{V}$ 的转置矩阵的乘积形式。用表格（tableau）表示这些矩阵的各种形式会更清楚些：

$$\begin{pmatrix} \\ \mathbf{A} \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \mathbf{U} \\ \\ \end{pmatrix} \cdot \begin{pmatrix} w_0 & & & \\ & w_1 & & \\ & & \cdots & \\ & & \cdots & \\ & & & w_{N-1} \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{V}^T \\ \\ \end{pmatrix} \tag{2.6.1}$$

矩阵 $\mathbf{U}$ 和 $\mathbf{V}$ 都是正交的，它们的列是正交标准化的，

$$\sum_{i=0}^{M-1} U_{ik} U_{in} = \delta_{kn} \qquad \begin{matrix} 0 \leqslant k \leqslant N-1 \\ 0 \leqslant n \leqslant N-1 \end{matrix} \tag{2.6.2}$$

$$\sum_{j=0}^{N-1} V_{jk} V_{jn} = \delta_{kn} \qquad \begin{matrix} 0 \leqslant k \leqslant N-1 \\ 0 \leqslant n \leqslant N-1 \end{matrix} \tag{2.6.3}$$

或者用表格来表示：

$$\begin{pmatrix} \\ \mathbf{U}^T \\ \\ \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{U} \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \mathbf{V}^T \\ \\ \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{V} \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \mathbf{1} \\ \\ \end{pmatrix} \tag{2.6.4}$$

由于 **V** 是方阵,所以它也是行正交标准化的,$\mathbf{V}\cdot\mathbf{V}^T=1$。

当 $M<N$ 时,SVD 分解也可以执行。在这种情况中,奇异值 $w_j, j=M,\ldots,N-1$ 都为零,并且 **U** 中相应的列也都为零。这时,仅当 $k,n\leqslant M-1$ 时,(2.6.2)式成立。

不管矩阵是否是奇异的,(2.6.1)式的分解总是可以完成的,而且这个分解几乎是惟一的。也就是说,其分解形式惟一到:(i)对 **U** 的列、**W** 的元素和 **V** 的列(或 $\mathbf{V}^T$ 的行)能进行相同的置换;(ii)形成 **U** 和 **V** 的任意列的线性组合,其在 **W** 中对应的元素仍恰好完全相同。对于 $M<N$ 的情况,这种自由置换的重要结论是,分解的数值算法对 $w_j, j=M,\ldots,N-1$ 要求返回非零,而这 $N-M$ 个零奇异值应分散于所有 $j=0,1,\ldots,N-1$ 的位置上。

在本节的最后将给出一个程序 svdcmp,对任意矩阵 **A** 进行 SVD 变换,将其替换成 **U**(它们的形式是相同的)并且分别给出 **W** 和 **V**。程序 svdcmp 基于 Forsythe 等人提供的程序[1],而他们的这个程序又基于 Golub 和 Reinsch 原来的程序,本节参考文献[2~4]以及其他一些地方都可以找到它的各种形式。这些参考书中对所用算法进行了广泛深入的讨论。尽管我们也不喜欢使用黑箱程序,但我们还是要求读者接受它,因为这样就不必花太多时间去研究背景知识。有充足的理由表明,该算法是非常稳定的,出现失误是极例外的情况。关于该算法的大部分概念(将矩阵 Householder 约化成双对角矩阵的形式,通过带位移的 QR 过程将矩阵对角化),将在第 11 章中进行讨论。

如果对黑箱产生怀疑,可以亲自验证一下 svdcmp。这很容易做到:任意生成一个矩阵 **A**,调用该程序,然后验证一下矩阵的乘积是否满足(2.6.1)式和(2.6.4)式。由于这两个方程是定义 SVD 方法的惟一限定条件,所以对所选定的 **A** 这样做便是完全的验证了。

下面对 SVD 进行深入讨论。

2.6.1 方阵的 SVD

如果矩阵 **A** 为 $N\times N$ 阶的方阵,则 **U**,**V** 和 **W** 均为相同大小的方阵。它们的逆也很容易计算:**U** 和 **V** 是正交的,因而它们的逆就是它们的转置。**W** 为对角阵,它的逆也是对角阵,其元素为原来元素 w_j 的倒数。由(2.6.1)式可直接得到 **A** 的逆为

$$\mathbf{A}^{-1}=\mathbf{V}\cdot[\mathrm{diag}\,(1/w_j)]\cdot\mathbf{U}^T \tag{2.6.5}$$

这么做惟一可能出错的地方是 w_j 中有一个为零,或其(数值上)非常小,由于舍入误差导致丢失。如果不止一个 w_j 存在这种问题,则矩阵就更加奇异。因此,SVD 首先就让使用者清楚地知道这种情况。

形式上,矩阵的**条件数**定义为 w_j 的最大元与最小元的比值。如果一个矩阵的条件数无穷大,则该矩阵是奇异的;如果一个矩阵的条件数太大,即其倒数超出了机器的浮点精度(比如,单精度数小于 10^{-6} 或双精度数小于 10^{-12}),则称该矩阵是**病态的**。

对奇异矩阵,**零空间**和**值域**的概念很重要。考虑下面的联立方程组:

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{2.6.6}$$

其中 **A** 为方阵,**b** 和 **x** 为向量。(2.6.6)式将 **A** 定义为从向量空间 **x** 到向量空间 **b** 的一个线性映射。如果 **A** 是奇异的,则存在子空间 **x**,称之为零空间,它被映射为零,$\mathbf{A}\cdot\mathbf{x}=0$。零空间的维数(可在 **x** 中找到的线性无关的向量个数)称为 **A** 的**零维数**。

还有一些通过 **A** 可以"到达"的子空间 **b**,就是说存在一些 **x** 被映射到其上。子空间 **b** 被称为 **A** 的值域。值域的维数称为 **A** 的**秩**。如果 **A** 为非奇异的,则其值域为整个向量空间 **b**,即其秩为 N。如果 **A** 是奇异的,则秩小于 N。事实上,相应的定理为"秩与零维数之和等于 N"。

SVD 明确地构造了矩阵零空间和值域的正交标准化基。特别是对 **U** 的列,若与其标号相同的元素 w_j 为非零元,则其构成值域的正交标准化的基底;对 **V** 的列,若与其标号相同的 w_j 为零,则其构成零空间的正交标准化的基底。

现在,看一下在 **A** 为奇异的情况下,联立线性方程组(2.6.6)的求解过程。首先,对于**齐次**线性方程组,因其 $\mathbf{b}=0$,所以可以立刻用 SVD 方法求解:对应 w_j 为零的 **V** 中的任何一列得到一个解。

当右端项向量 **b** 不为零时,一个重要的问题是,右端项向量 **b** 是否存在 **A** 的值域。如果存在,则该奇异矩阵确实有解。事实上解不止一个,因为零空间的任意向量(**V** 中对应 w_j 为零的列)进行任意线性组合都可以添加到 **x** 上。

如果仅想求出该解向量集合中一个有代表性的特殊解，则可以求具有最小长度$|\mathbf{x}|^2$的那个解。下面介绍使用SVD求该向量的方法。如果$w_j=0$，简单地将$1/w_j$替换成零即可（设置$\infty=0$的情况并不常见）。然后计算（从右向左）

$$\mathbf{x}=\mathbf{V}\cdot[\mathrm{diag}\,(1/w_j)]\cdot(\mathbf{U}^T\cdot\mathbf{b}) \tag{2.6.7}$$

这就是具有最小长度的解向量，在零空间中$\mathbf{V}$的列完成对解集的要求。

证明：考虑$|\mathbf{x}+\mathbf{x}'|$，其中$\mathbf{x}'$属于零空间。设$\mathbf{W}^{-1}$表示有零元素的矩阵$\mathbf{W}$被修改过后的逆矩阵，

$$\begin{aligned}|\mathbf{x}+\mathbf{x}'|&=|\mathbf{V}\cdot\mathbf{W}^{-1}\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{x}'|\\&=|\mathbf{V}\cdot(\mathbf{W}^{-1}\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{V}^T\cdot\mathbf{x}')|\\&=|\mathbf{W}^{-1}\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{V}^T\cdot\mathbf{x}'|\end{aligned} \tag{2.6.8}$$

其中第一个等式由(2.6.7)式导出，第二个和第三个等式由$\mathbf{V}$是正交标准化得出。如果检验一下右端求和的两项，会发现第一项中仅当$w_j\neq0$时，第j个分量为非零；而对于第二项，由于$\mathbf{x}'$在零空间中，仅当$w_j=0$时，第j个分量为非零。故对$\mathbf{x}'=0$时，其长度最小，证毕。

如果$\mathbf{b}$不在奇异矩阵$\mathbf{A}$的值域范围内，则(2.6.6)式无解。但值得高兴的是，如果$\mathbf{b}$不在$\mathbf{A}$的值域中，(2.6.7)式依然可以构造一个"解"向量$\mathbf{x}$。该向量并不是$\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$的精确解，但在所有可能的向量$\mathbf{x}$中，它是在最小二乘意义上最可能接近的解。换言之，(2.6.7)式求得的$\mathbf{x}$，使

$$r\equiv|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}| \tag{2.6.9}$$

最小。数r称为解的**余量**。

证明与(2.6.8)式类似：假设对$\mathbf{x}$添加了某个任意的$\mathbf{x}'$，则$\mathbf{A}\cdot\mathbf{x}-\mathbf{b}$也添加了一项$\mathbf{b}'\equiv\mathbf{A}\cdot\mathbf{x}'$。显然$\mathbf{b}'$在$\mathbf{A}$的值域中，则有

$$\begin{aligned}|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}+\mathbf{b}'|&=|(\mathbf{U}\cdot\mathbf{W}\cdot\mathbf{V}^T)\cdot(\mathbf{V}\cdot\mathbf{W}^{-1}\cdot\mathbf{U}^T\cdot\mathbf{b})-\mathbf{b}+\mathbf{b}'|\\&=|(\mathbf{U}\cdot\mathbf{W}\cdot\mathbf{W}^{-1}\cdot\mathbf{U}^T-1)\cdot\mathbf{b}+\mathbf{b}'|\\&=|\mathbf{U}\cdot[(\mathbf{W}\cdot\mathbf{W}^{-1}-1)\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{U}^T\cdot\mathbf{b}']|\\&=|(\mathbf{W}\cdot\mathbf{W}^{-1}-1)\cdot\mathbf{U}^T\cdot\mathbf{b}+\mathbf{U}^T\cdot\mathbf{b}'|\end{aligned} \tag{2.6.10}$$

$(\mathbf{W}\cdot\mathbf{W}^{-1}-1)$为对角阵，只有当$w_j=0$时其第$j$个分量为非零。而由于$\mathbf{b}'$为$\mathbf{A}$的值域，故只有当$w_j\neq0$时，$\mathbf{U}^T\mathbf{b}'$的第$j$个分量为非零。因此，令$\mathbf{b}'=0$即得最小值，证毕。

图 2.6.1 是对所讨论 SVD 的总结。

(2.6.6)式以后的讨论中，都已经忽略矩阵是奇异的还是非奇异的，这从分析的角度来讲是对的。然而从数值的角度来讲，更一般的情况为，某些w_j非常小但不为零，即矩阵是"病态"的。在这种情况下，LU 分解法或高斯消去法等直接解法，可能给出方程组的形式解（即没遇到零主元素）。但解向量可能有非常大的分量，当与矩阵$\mathbf{A}$相乘时，由于代数上的相互抵消而得到右端项向量$\mathbf{b}$的一个非常差的近似解。遇到这些情况时，将这些小的w_j化为零，再用(2.6.7)式来求解，求得的解向量$\mathbf{x}$比用直接法和保留这些非零的小w_j的 SVD 解法都要好得多（在使余量$|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}|$较小的意义上看）。

看起来这可能有点矛盾，因为，将奇异值化为零相当于把要求解的方程组所生成的一个线性组合抛弃。产生此矛盾的原因是，我们抛弃了一个由于舍入误差而变得非常差的方程组合。最好的情况也是无用的，而通常比无用还要差，因为它将差不多已是零空间向量的解沿某个方向"拖"偏了一点。这样，它就掺进了舍入误差而使得余量$|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}|$更大了。

SVD 方法不能盲目使用。必须谨慎地确定将小w_j化为零的阈值，而且（或者）必须考虑清楚多大的计算余量$|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}|$可以接受。

作为示例，有一个"回代"程序 svbksb 用来评价(2.6.7)式，并得到右端项$\mathbf{b}$的一个解向量$\mathbf{x}$，要求先调用 svdcmp 计算出矩阵$\mathbf{A}$的 SVD 形式。注意该程序假定已经将小的w_j化为零了，因为该程序不提供此项功能。如果还没有将小w_j化零，则该程序与其他的直接方法一样是"病态"的，这样就误用了 SVD。

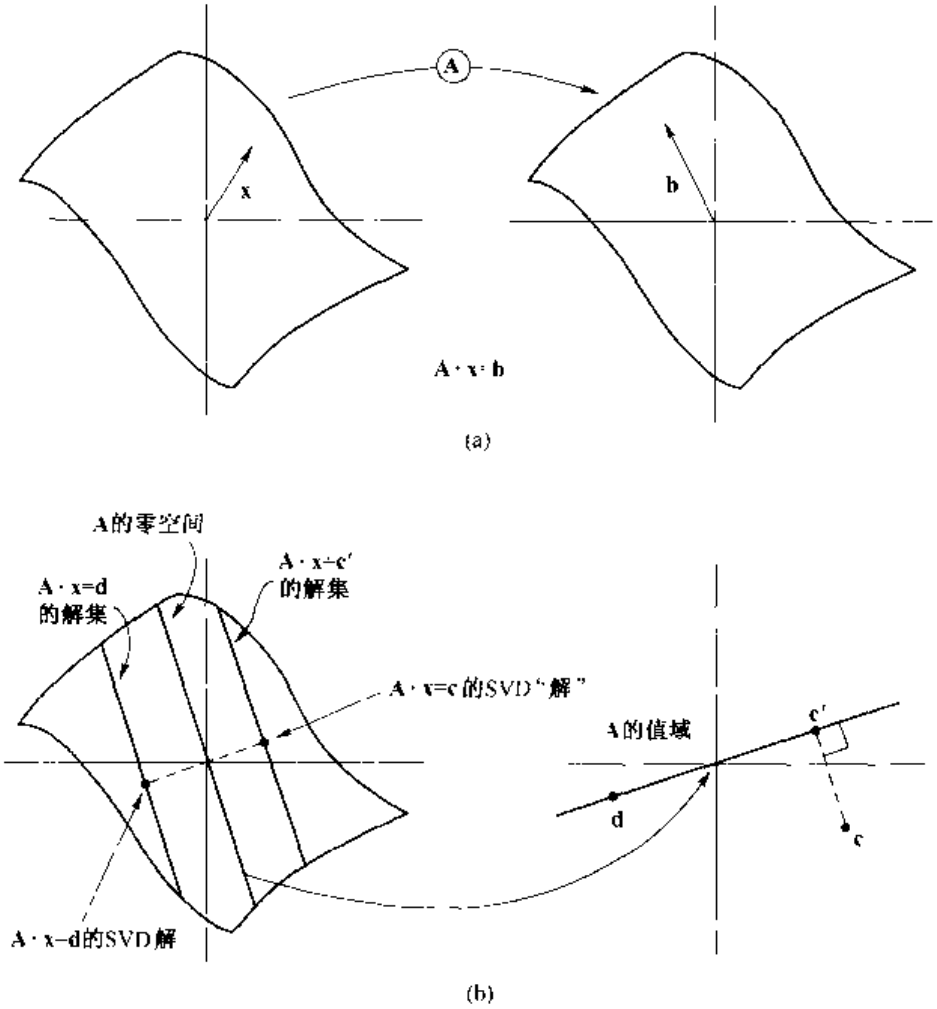


图 2.6.1 SVD 总结 (a)非奇异矩阵 **A** 将一个向量空间映射到一个具有相同维数的向量空间。向量 **x** 被映射成 **b**，因此 **x** 满足方程 $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$。(b)奇异矩阵 **A** 将一个向量空间映射到低维空间，这里用一个平面映射到一条直线表示，该直线称为 **A** 的"值域"。**A** 的"零空间"被映射成零。$\mathbf{A}\cdot\mathbf{x}=\mathbf{d}$ 的解集包含任意一个特殊解加上零空间的任意向量，这里用平行于零空间的直线表示。奇异值分解(SVD)选择离零最近的特殊解，如图示。点 **c** 不在 **A** 的值域范围内，所以 $\mathbf{A}\cdot\mathbf{x}=\mathbf{c}$ 无解。SVD 找到最小二乘意义上的最优的折中解，即 $\mathbf{A}\cdot\mathbf{x}=\mathbf{c}'$ 的解，如图所示

```
#include "nr.h"

void NR::svbksb(Mat_I_DP &u, Vec_I_DP &w, Mat_I_DP &v, Vec_I_DP &b, Vec_O_DP &x)
```

求 $A\cdot X=B$ 的解向量 X，其中 A 由 svdcmp 返回的数组 u[0..m-1][0..n-1]，w[0..n-1] 和 v[0..n-1][0..n-1] 确定，m 和 n 对方阵来说是相同的。b[0..m-1] 为输入右端项。x[0..n-1] 为输出的解向量。输入量不被破坏，因而程序可以用不同的 b 顺序调用。

```
{
    int jj,j,i;
    DP s;

    int m=u.nrows();
    int n=u.ncols();
    Vec_DP tmp(n);
    for (j=0;j<n;j++) {                 计算U^T B。
        s=0.0;
        if (w[j] != 0.0) {              仅当w_j为非零时，结果为非零。
            for (i=0;i<m;i++) s += u[i][j]*b[i];
            s /= w[j];                  这里用w_j除。
        }
        tmp[j]=s;
    }
```

```
        for (j=0;j<n;j++) {                 将矩阵乘以V得到解。
            s=0.0;
            for (jj=0;jj<n;jj++) s += v[j][jj]*tmp[jj];
            x[j]=s;
        }
    }
```

注意，svdcmp 和 svbksb 的典型用法表面上跟 ludcmp 和 lubksb 的典型用法很相似。这两种情况下，左端矩阵 **A** 都仅分解一次，然后对右端项使用该分解一次，或是用不同的右端项使用该分解多次。最重要的不同之处在于，在调用 svbksb 之前要对奇异数值进行"编译"：

```
const int N = ...
DP wmax,wmin;
Mat_DP a(N,N),u(N,N),v(N,N);
Vec_DP w(N),b(N),x(N);
int i,j;
...
for(i=0;i<N;i++)                    如果不想破坏a，将其副本存入u。
    for j=0;j<N;j++)
        u[i][j]=a[i][j];
NR::svdcmp(u,w,v);                  对方阵a进行SVD。
wmax=0.0;                           得到最大的奇异值。
for(j=0;j<N;j++) if (w[j] > wmax) wmax=w[j];
这里设置的奇异值的阈值允许为非零。此常数是典型的、非一般的，根据实际应用可以改动。
wmin=wmax*1.0e-12;
for(j=0;j<N;j++) if (w[j] < wmin) w[j]=0.0;
NR::svbksb(u,w,v,b,x);              现在可以回代了。
```

2.6.2　方程个数少于未知数个数的SVD

如果方程个数 M 少于未知数个数 N，则不能得到惟一解。通常存在一个 $N-M$ 维的基础解系，SVD 可找到此整个解空间。

因为 $M<N$，所以 SVD 分解将产生 $N-M$ 个零或可忽略不计的 w_j。M 个方程中的退化情况，还会产生额外的零 w_j。在调用 svbksb 之前，一定要保证找到这些小的 w_j，并将其化为零，则可以得到特殊的解向量 **x**。跟以前一样，**V** 中对应于零化 w_j 的列是基础解向量，其线性组合加上特定的解就构成一个解空间。

2.6.3　方程个数多于未知数个数的SVD

这种情况将在第 15 章中讨论。对超定的线性方程组，希望找到的是其最小二乘解。用表格表示，要求解的方程组为

$$\begin{pmatrix} & & \\ & \mathbf{A} & \\ & & \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{x} \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \mathbf{b} \\ \\ \end{pmatrix} \tag{2.6.11}$$

在上面方阵情况下所给出的证明，不用修改就适用于方程个数多于未知数个数的情况。最小二乘解向量 **x** 由(2.6.7)式给出，非方阵的情况如下：

$$\begin{pmatrix} \\ \mathbf{x} \\ \\ \end{pmatrix} = \begin{pmatrix} & & \\ & \mathbf{V} & \\ & & \end{pmatrix} \cdot \begin{pmatrix} \\ \mathrm{diag}(1/w_j) \\ \\ \end{pmatrix} \cdot \begin{pmatrix} & & \\ & \mathbf{U}^T & \\ & & \end{pmatrix} \cdot \begin{pmatrix} \\ \mathbf{b} \\ \\ \end{pmatrix} \tag{2.6.12}$$

一般情况下,矩阵 **W** 不是奇异的,且无需设 w_j 为零。然而,偶尔 **A** 也可能是列退化的。这种情况下,就需要将小的 w_j 值化为零了。**V** 中相应列给出 **x** 的线性组合,这时即使假定是超定方程组,其限定条件也要变差。

有时候,尽管由于计算上的原因不必零化任何 w_j,然而可能会注意那些特别小的值。它们在 **V** 中对应的列是 **x** 的线性组合,但它们对数据并不敏感。事实上,可能想要零化这些 w_j,以减少其自由参量的数目。这些问题将在第 15 章中进一步讨论。

2.6.4 构造标准正交基

假设在 M 维向量空间有 N 个向量,$N \leq M$,则 N 个向量跨越了整个向量空间的一些子空间。我们经常想要构造跨越相同子空间的 N 个向量的一个标准正交基。教科书中的做法是,通过 Gram-Schmidt 正交化,由一个向量开始,然后将子空间每次扩展一个维数。然而从数值上看,由于其内在的舍入误差,单纯的 Gram-Schmidt 正交化效果很差。

对一个子空间,构造标准正交基的正确办法是,通过 SVD 构造一个 $M \times N$ 的矩阵 **A**,其 N 列是已有的向量。对 **A** 运行 svdcmp,则矩阵 **U**(事实上 svdcmp 输出的 **U** 替代了 **A**)的列就是欲求的标准正交基向量。

也许还应检验一下输出的 w_j 中是否含有零值。如果有,则事实上跨越的子空间不是 N 维的,**U** 中对应于 w_j 为零的列应从标准正交基中除去。

本章 2.10 节中讨论的 QR 分解也可构造一个标准正交基,参见[5]。

2.6.5 矩阵的近似

注意(2.6.1)式可以重写,将任意矩阵 A_{ij} 表示成 **U** 的列和 $\mathbf{V}^T$ 的行的外积之和形式,奇异值 w_j 作为"权因子",

$$A_{ij} = \sum_{k=0}^{N-1} w_k U_{ik} V_{jk} \tag{2.6.13}$$

如果碰到矩阵 **A** 的大部分奇异值 w_j 都非常小的情况,则仅用(2.6.13)式的一小部分项就可得到 **A** 的很不错的近似值。这意味着仅需保存少量的 **U** 和 **V** 的列(相同的 k 列)就可以恢复整个矩阵,且精度很高。

同时注意到用此近似的矩阵去乘向量 **x** 也是非常高效的。可将 **x** 与每个存储的 **V** 的列做点积,并乘以相应的比例值 w_k,然后分别乘以对应的 **U** 的列,并累加求和。如果矩阵用很少的 K 个奇异值来近似,则计算 **A**·**x**仅需乘法大约 $K(M+N)$次,而不必像求整个矩阵那样用 MN 次。

2.6.6 SVD 算法

下面是用于构造任意矩阵的奇异值分解的算法。有关方法的讨论参见 11.2 节和 11.3 节,以及本节参考文献[4,5]

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::svdcmp(Mat_IO_DP &a, Vec_O_DP &w, Mat_O_DP &v)
给定矩阵 a[0..m-1][0..n-1],本程序求其奇异值分解,A = U·W·V^T。输出矩阵 U 保存在数组 a 中。代表奇异值的
对角阵 W 作为输出量用 w[0..n-1]表示。矩阵 V(不是其转置形式 V^T)作为输出量用 v[0..n-1][0..n-1]表示。
{
    bool flag;
    int i,its,j,jj,k,l,nm;
    DP anorm,c,f,g,h,s,scale,x,y,z;

    int m=a.nrows();
    int n=a.ncols();
    Vec_DP rv1(n);
    g=scale=anorm=0.0;                    Householder约化成双对角形式。
    for (i=0;i<n;i++) {
```

```
            l=i+2;
            rv1[i]=scale*g;
            g=s=scale=0.0;
            if (i < m) {
                for (k=i;k<m;k++) scale += fabs(a[k][i]);
                if (scale != 0.0) {
                    for (k=i;k<m;k++) {
                        a[k][i] /= scale;
                        s += a[k][i]*a[k][i];
                    }
                    f=a[i][i];
                    g = -SIGN(sqrt(s),f);
                    h=f*g-s;
                    a[i][i]=f-g;
                    for (j=l-1;j<n;j++) {
                        for (s=0.0,k=i;k<m;k++) s += a[k][i]*a[k][j];
                        f=s/h;
                        for (k=i;k<m;k++) a[k][j] += f*a[k][i];
                    }
                    for (k=i;k<m;k++) a[k][i] *= scale;
                }
            }
            w[i]=scale *g;
        g=s=scale=0.0;
        if (i+1 <= m && i != n) {
            for (k=l-1;k<n;k++) scale += fabs(a[i][k]);
            if (scale != 0.0) {
                for (k=l-1;k<n;k++) {
                    a[i][k] /= scale;
                    s += a[i][k]*a[i][k];
                }
                f=a[i][l-1];
                g = -SIGN(sqrt(s),f);
                h=f*g-s;
                a[i][l-1]=f-g;
                for (k=l-1;k<n;k++) rv1[k]=a[i][k]/h;
                for (j=l-1;j<m;j++) {
                    for (s=0.0,k=l-1;k<n;k++) s += a[j][k]*a[i][k];
                    for (k=l-1;k<n;k++) a[j][k] += s*rv1[k];
                }
                for (k=l-1;k<n;k++) a[i][k] *= scale;
            }
        }
        anorm=MAX(anorm,(fabs(w[i])+fabs(rv1[i])));
    }
    for (i=n-1;i>=0;i--) {                          求右端变换累加和。
        if (i < n-1) {
            if (g != 0.0) {
                for (j=l;j<n;j++)                   进行两次除法避免可能的下溢。
                    v[j][i]=(a[i][j]/a[i][l])/g;
                for (j=l;j<n;j++) {
                    for (s=0.0,k=l;k<n;k++) s += a[i][k]*v[k][j];
                    for (k=l;k<n;k++) v[k][j] += s*v[k][i];
                }
            }
            for (j=l;j<n;j++) v[i][j]=v[j][i]=0.0;
        }
        v[i][i]=1.0;
        g=rv1[i];
        l=i;
    }
    for (i=MIN(m,n)-1;i>=0;i--) {                   求左端变换累加和。
        l=i+1;
        g=w[i];
        for (j=l;j<n;j++) a[i][j]=0.0;
        if (g != 0.0) {
            g=1.0/g;
            for (j=l;j<n;j++) {
                for (s=0.0,k=l;k<m;k++) s += a[k][i]*a[k][j];
                f=(s/a[i][i])*g;
                for (k=i;k<m;k++) a[k][j] += f*a[k][i];
            }
            for (j=i;j<m;j++) a[j][i] *= g;
```

```
    } else for (j=i;j<m;j++) a[j][i]=0.0;
    ++a[i][i];
}
for (k=n-1;k>=0;k--) {                  双对角阵对角化: 对奇异值循环,对所要做的迭代循环。
    for (its=0;its<30;its++) {
        flag=true;
        for (l=k;l>=0;l--) {            做分解测试。
            nm=l-1;                     注意rv1[0]总为零。
            if (fabs(rv1[l])+anorm == anorm) {
                flag=false;
                break;
            }
            if (fabs(w[nm])+anorm == anorm) break;
        }
if (flag) {
    c=0.0;                              如果l>0,则消去rv1[l]。
    s=1.0;
    for (i=l-1;i<k+1;i++) {
        f=s*rv1[i];
        rv1[i]=c*rv1[i];
        if (fabs(f)+anorm == anorm) break;
        g=w[i];
        h=pythag(f,g);
        w[i]=h;
        h=1.0/h;
        c=g*h;
        s = -f*h;
        for (j=0;j<m;j++) {
            y=a[j][nm];
            z=a[j][i];
            a[j][nm]=y*c+z*s;
            a[j][i]=z*c-y*s;
        }
    }
}
z=w[k];
if (l == k) {                           收敛。
    if (z < 0.0) {                      使奇偶数值非负。
        w[k] = -z;
        for (j=0;j<n;j++) v[j][k] = -v[j][k];
    }
    break;
}
if (its == 29) nrerror("no convergence in 30 svdcmp iterations");
x=w[l];                                 从底部2-2子式移位。
nm=k-1;
y=w[nm];
g=rv1[nm];
h=rv1[k];
f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0*h*y);
g=pythag(f,1.0);
f=((x-z)*(x+z)+h*((y/(f+SIGN(g,f)))-h))/x;
c=s=1.0;                                下面一个QR变换。
for (j=l;j<=nm;j++) {
    i=j+1;
    g=rv1[i];
    y=w[i];
    h=s*g;
    g=c*g;
    z=pythag(f,h);
    rv1[j]=z;
    c=f/z;
    s=h/z;
    f=x*c+g*s;
    g=g*c-x*s;
    h=y*s;
    y *= c;
    for (jj=0;jj<n;jj++) {
        x=v[jj][j];
        z=v[jj][i];
        v[jj][j]=x*c+z*s;
        v[jj][i]=z*c-x*s;
```

```
                }
                z=pythag(f,h);
                w[j]=z;                    如果z=0可循环任意次。
                if (z) {
                    z=1.0/z;
                    c=f*z;
                    s=h*z;
                }
                f=c*g+s*y;
                x=c*y-s*g;
                for (jj=0;jj<m;jj++) {
                    y=a[jj][j];
                    z=a[jj][i];
                    a[jj][j]=y*c+z*s;
                    a[jj][i]=z*c-y*s;
                }
            }
            rv1[l]=0.0;
            rv1[k]=f;
            w[k]=x;
        }
    }
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::pythag(const DP a, const DP b)
计算(a^2 + b^2)^1/2，没有破坏性上溢和下溢。
{
    DP absa,absb;

    absa=fabs(a);
    absb=fabs(b);
    if (absa > absb) return absa*sqrt(1.0+SQR(absb/absa));
    else return (absb == 0.0 ? 0.0 : absb*sqrt(1.0+SQR(absa/absb)));
}
```

参考文献和深入阅读

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §8.6 and Chapter 12 (SVD). QR decomposition is discussed in §5.2.6. [5]

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 18.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I.10 by G.H. Golub and C. Reinsch. [2]

Anderson, E., et al. 2000, LAPACK User's Guide, 3rd ed. (Philadelphia: S.I.A.M.). [3]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag).

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §6.7. [4]

2.7　稀疏线性方程组

如果一个线性方程组，仅有相对很少的一部分元素 a_{ij} 为非零，则称其为稀疏的。对这类问题，无需使用通用的线性系统求解方法，因为在求解或求逆矩阵时，运算量为 $O(N^3)$ 的大部分算术运算都用来处理零元

素，而且，希望求解的问题可能大到已超出了可用的存储空间，且保留这些无用的零元素很浪费空间。注意，对任意稀疏矩阵的求解方法，不外乎有两个明确的目的(它们并不总能同时达到)：省时间并且(或者)省空间。

在2.4节中，已提及过一种原始的稀疏矩阵模型，带状对角矩阵。例如，对于三角矩阵情形，可以做到既省时间(N^3 降至 N 数量级)，又省空间(N^2 降至 N 数量级)。求解的算法原则上无异于一般的LU分解法，不同的是注意到了如何记录零元素而应用得更巧妙。大部分求解稀疏矩阵的实际问题都有同样的特征。它们基本上都采用分解法或者类似于Gauss-Jordan的消去法，但必须进行认真的优化，以减少称为**填充项**的数目，这些填充项最初是零元素，而在求解过程中必然变成非零元素，为此必须保留它们的存储空间。

求解稀疏方程组的直接方法主要依赖于矩阵的稀疏模式。常见的模式，或者在化简一般形式矩阵时所用的一些有用子模式，都已有了特定的名称及特定的解法。这里不进行详细介绍了，可以参见本节的参考文献，自学一些更专业的内容。下面列出的名词和图2.7.1可供使用。

- 三对角矩阵
- 带宽为 M 的带状对角矩阵(或带状矩阵)
- 带状三角矩阵
- 块状对角矩阵
- 块状三对角矩阵
- 块状三角矩阵
- 环带状矩阵
- 单边(或双边)镶边块状对角矩阵
- 单边(或双边)镶边块状三角矩阵
- 单边(或双边)镶边带状对角矩阵
- 单边(或双边)镶边带状三角矩阵
- 其他

此外，还应知道一些在求解二维或高维偏微分方程时遇到的特殊稀疏矩阵形式。参见第19章。

如果特殊的稀疏模式不是单一的，那么可能希望试用**分析/因子分解/运算**程序包，它能自动地计算出怎样减少填充项。对每个稀疏模式只进行一次分析步骤。对每个适合该模式的特殊矩阵进行一次因子分解步骤。对每个使用该特殊矩阵的右端项进行一次运算步骤。有关的内容参见本节参考文献[2,3]。NAG库[4]具有分析/因子分解/运算的能力。从IMSL[5]中也可得到一个稀疏矩阵计算的程序集，如同**耶鲁大学稀疏矩阵程序包**(*Yale Sparse Matrix Package*)[6]。

应当清楚，跟类似用主元消去法的正规LU分解法相比，稀疏矩阵解法为减少填充项和算术运算次数，进行了特殊的次序交换和消去，一般来说其算法的数值稳定性要有所降低。将求解的线性问题按比例变换其系数，使其非零元素之间可进行量值比较，这样(如果能做到的话)就可改善这种稳定性问题。

下面将提供两种方法，可用来求解一般的稀疏矩阵问题，而并不依赖于稀疏模式的具体情况。

2.7.1 Sherman-Morrison公式

假定已求出了方阵 $\mathbf{A}$ 的逆 $\mathbf{A}^{-1}$。现在，想对 $\mathbf{A}$ 进行些小变化，比如更换一个元素 a_{ij}，或更换几个元素，或更换一行，或更换一列。可以直接计算 $\mathbf{A}^{-1}$ 中相应有变化的项，而不必重复整个复杂的过程。如果对向量 $\mathbf{u}$ 和 $\mathbf{v}$ 来说变换形式为

$$\mathbf{A} \rightarrow (\mathbf{A}+\mathbf{u}\otimes\mathbf{v}) \tag{2.7.1}$$

如果 $\mathbf{u}$ 为单位向量 $\mathbf{e}_i$，则(2.7.1)式将 $\mathbf{v}$ 的元素添加到第 i 行上(回顾一下，$\mathbf{u}\otimes\mathbf{v}$ 表示矩阵的第 i 行、第 j 列元素值为 $\mathbf{u}$ 的第 i 项与 $\mathbf{v}$ 的第 j 项之积)。如果 $\mathbf{v}$ 是单位向量 $\mathbf{e}_j$，则(2.7.1)式将 $\mathbf{u}$ 的元素添加到第 j 列上。如果 $\mathbf{u}$ 和 $\mathbf{v}$ 恰好都是单位向量 $\mathbf{e}_i$ 和 $\mathbf{e}_j$，则相应的项仅添加到元素 a_{ij} 上。

Sherman-Morrison公式给出逆矩阵 $(\mathbf{A}+\mathbf{u}\otimes\mathbf{v})^{-1}$，推导如下：

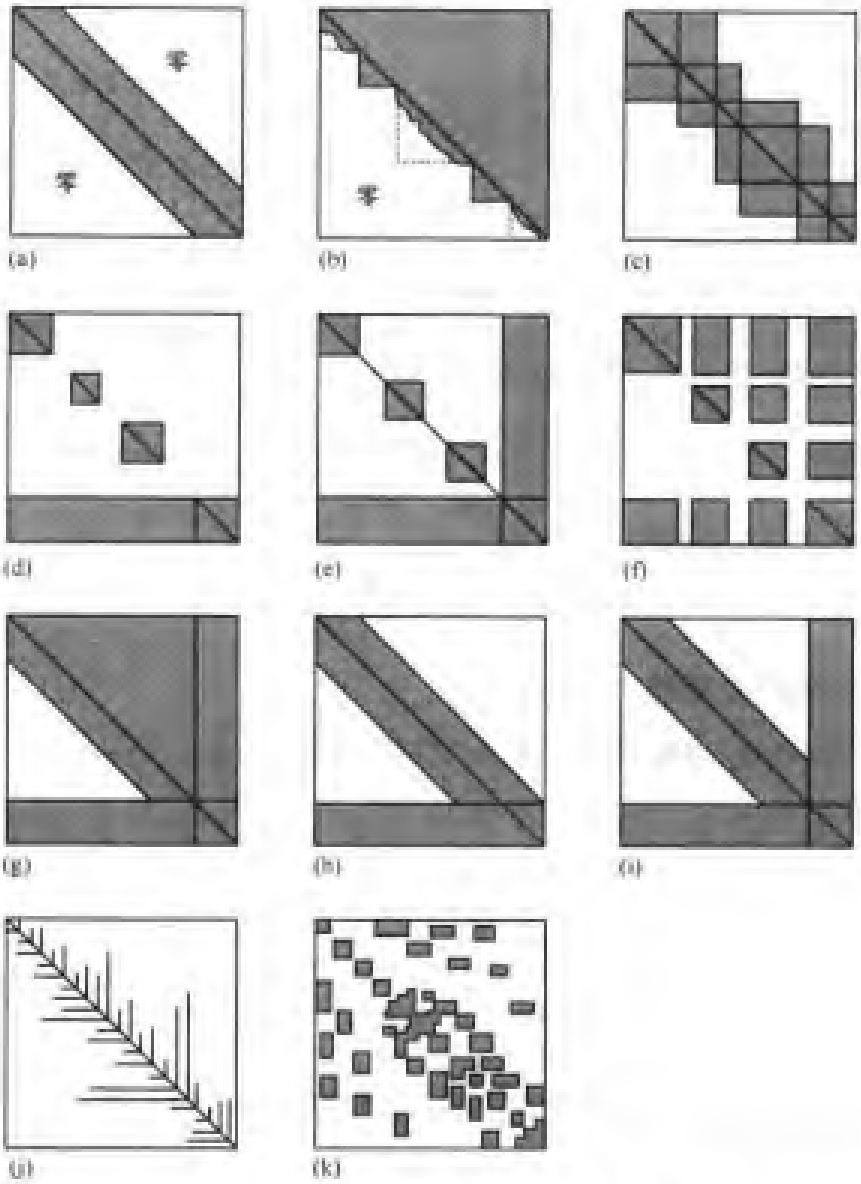


图 2.7.1 稀疏矩阵的一些标准形式。(a)带状对角矩阵;(b)块状三角矩阵;(c)块状三对角矩阵;(d)单边镶边块状对角矩阵;(e)双边镶边块状对角矩阵;(f)单边镶边块状三角矩阵;(g)镶边带状三角矩阵;(h)和(i)分别为单边和双边镶边带状对角矩阵;(j)和(k)为其他形式。参见本节参考文献[1]

$$
\begin{aligned}
(\mathbf{A}+\mathbf{u}\otimes\mathbf{v})^{-1} &= (\mathbf{1}+\mathbf{A}^{-1}\cdot\mathbf{u}\otimes\mathbf{v})^{-1}\cdot\mathbf{A}^{-1} \\
&= (\mathbf{1}-\mathbf{A}^{-1}\cdot\mathbf{u}\otimes\mathbf{v}+\mathbf{A}^{-1}\cdot\mathbf{u}\otimes\mathbf{v}\cdot\mathbf{A}^{-1}\cdot\mathbf{u}\otimes\mathbf{v}-\ldots)\cdot\mathbf{A}^{-1} \\
&= \mathbf{A}^{-1}-\mathbf{A}^{-1}\cdot\mathbf{u}\otimes\mathbf{v}\cdot\mathbf{A}^{-1}(1-\lambda+\lambda^2-\ldots) \\
&= \mathbf{A}^{-1}-\frac{(\mathbf{A}^{-1}\cdot\mathbf{u})\otimes(\mathbf{v}\cdot\mathbf{A}^{-1})}{1+\lambda}
\end{aligned}
\tag{2.7.2}
$$

其中,

$$
\lambda \equiv \mathbf{v}\cdot\mathbf{A}^{-1}\cdot\mathbf{u} \tag{2.7.3}
$$

(2.7.2)式中的第二行为标准幂级数展开。第三行是结合内积和外积分解出标量 λ。

(2.7.2)式的用途在于:给定 $\mathbf{A}^{-1}$ 及向量 $\mathbf{u}$ 和 $\mathbf{v}$,只需求出两个矩阵的乘积和一个向量的点积,

$$
\mathbf{z} \equiv \mathbf{A}^{-1}\cdot\mathbf{u} \qquad \mathbf{w} \equiv (\mathbf{A}^{-1})^{T}\cdot\mathbf{v} \qquad \lambda = \mathbf{v}\cdot\mathbf{z} \tag{2.7.4}
$$

便可得到逆中欲求的变化

$$\mathbf{A}^{-1} \quad \rightarrow \quad \mathbf{A}^{-1} - \frac{\mathbf{z} \otimes \mathbf{w}}{1+\lambda} \tag{2.7.5}$$

整个过程仅需要 $3N^2$ 次乘法和次数差不多的加法(如果 **u** 或 **v** 为单位向量,则次数更少)。

Sherman-Morrison 公式可直接用来求解一类稀疏矩阵问题。如果已有一个快速计算 **A** 的逆的算法(例如三对角阵或其他标准稀疏矩阵的形式),则(2.7.4)式和(2.7.5)式允许构造相关但形式更为复杂的矩阵,例如每次添加一行或一列。注意,Sherman-Morrison 公式可以连续使用一次以上,每次都用最近更新过的 $\mathbf{A}^{-1}$((2.7.5)式)。当然,如果不得不修改每一行,则又回到 N^3 次的方法了。虽然 N^3 的常数仅是较好的直接法的计算量的几倍,但失去了主元法稳定的优点,因而要谨慎使用。

对另外一些稀疏矩阵问题,Sherman-Morrison 公式不能直接应用,原因很简单,要存储整个逆矩阵 $\mathbf{A}^{-1}$ 是不可取的。如果仅要修改 $\mathbf{u}\otimes\mathbf{v}$ 的一处,且求解的线性问题为

$$(\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \cdot \mathbf{x} = \mathbf{b} \tag{2.7.6}$$

则可按下面的方法进行。假定对矩阵 **A** 应用的是通用的快速算法,来求解两个辅助问题

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \qquad \mathbf{A} \cdot \mathbf{z} = \mathbf{u} \tag{2.7.7}$$

求向量 **y** 和 **z**。由此,将(2.7.2)式右边乘以 **b**,得

$$\mathbf{x} = \mathbf{y} - \left[\frac{\mathbf{v} \cdot \mathbf{y}}{1 + (\mathbf{v} \cdot \mathbf{z})}\right] \mathbf{z} \tag{2.7.8}$$

2.7.2 周期三对角方程组

会经常遇到所谓的**周期三对角方程组**,而且是使用刚才所述的 Sherman-Morrison 方法求解的一个很好的示例。方程形式为

$$\begin{bmatrix} b_0 & c_0 & 0 & \cdots & & & \beta \\ a_1 & b_1 & c_1 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-2} & b_{N-2} & c_{N-2} \\ \alpha & & & \cdots & 0 & a_{N-1} & b_{N-1} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_{N-2} \\ x_{N-1} \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ \cdots \\ r_{N-2} \\ r_{N-1} \end{bmatrix} \tag{2.7.9}$$

这是一个三对角方程组,只是角上有矩阵元素 α 和 β。这种形式典型地由具有周期性边界条件的有限差分微分方程产生(参见 19.4 节)。

使用 Sherman-Morrison 公式,将该方程组视为三对角方程组加上一个修正处理。用(2.7.6)式中的表示方法(记号),将向量 **u** 和 **v** 定义为

$$\mathbf{u} = \begin{bmatrix} \gamma \\ 0 \\ \vdots \\ 0 \\ \alpha \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ \beta/\gamma \end{bmatrix} \tag{2.7.10}$$

此处 γ 暂时是任意的。矩阵 **A** 是(2.7.9)式中的三对角部分,其中有两项被修改过:

$$b_0' = b_0 - \gamma, \qquad b_{N-1}' = b_{N-1} - \alpha\beta/\gamma \tag{2.7.11}$$

现在,用标准的三对角算法来求解(2.7.7)式,然后由(2.7.8)式得到解。

下面的 cyclic 程序实现了该算法,取任意参数 $\gamma = -b_0$,以避免(2.7.11)式中的第一个方程由于减法运算造成精度损失。为了在第二个式中不会产生精度损失,可以进行不同的选择。

```
#include "nr.h"

void NR::cyclic(Vec_I_DP &a, Vec_I_DP &b, Vec_I_DP &c, const DP alpha,
    const DP beta, Vec_I_DP &r, Vec_O_DP &x)
```

求(2.7.9)式给出的"周期"线性方程组的一个解向量 x[0..n-1]。a,b,c 和 r 为输入向量,其维数均是[0..n-1],alpha 和 beta 是矩阵角上的元素值。输入量没有被修改。

```
{
    int i;
    DP fact,gamma;

    int n=a.size();
    if (n <= 2) nrerror("n too small in cyclic");
    Vec_DP bb(n),u(n),z(n);
    gamma = -b[0];                          形成bb[0]时避免出现减误差。
    bb[0]=b[0]-gamma;                       为修改过的三对角方程组设立对角线。
    bb[n-1]=b[n-1]-alpha*beta/gamma;
    for (i=1;i<n-1;i++) bb[i]=b[i];
    tridag(a,bb,c,r,x);                     求解 A · x = r。
    u[0]=gamma;                             设置向量 u。
    u[n-1]=alpha;
    for (i=1;i<n-1;i++) u[i]=0.0;
    tridag(a,bb,c,u,z);                     求解 A · z = u。
    fact=(x[0]+beta*x[n-1]/gamma)/          形成 v · x/(1 + v · z)。
        (1.0+z[0]+beta*z[n-1]/gamma);
    for (i=0;i<n;i++) x[i] -= fact*z[i];    得到解向量x。
}
```

2.7.3　深入讨论:Woodbury 公式

如果要修改多个项,则不能重复地使用(2.7.8)式,因为没有存储新的 $\mathbf{A}^{-1}$,所以不能在完成第一步后高效地求解辅助问题(2.7.7)。取而代之,**Woodbury 公式**是 Sherman-Morrison 公式的块矩阵版本形式,

$$
\begin{aligned}
&(\mathbf{A}+\mathbf{U}\cdot\mathbf{V}^T)^{-1}\\
&\quad=\mathbf{A}^{-1}-\left[\mathbf{A}^{-1}\cdot\mathbf{U}\cdot(\mathbf{1}+\mathbf{V}^T\cdot\mathbf{A}^{-1}\cdot\mathbf{U})^{-1}\cdot\mathbf{V}^T\cdot\mathbf{A}^{-1}\right]
\end{aligned}
\tag{2.7.12}
$$

其中 $\mathbf{A}$ 是 $N\times N$ 矩阵,而 $\mathbf{U}$ 和 $\mathbf{V}$ 是 $N\times P$ 矩阵,$P<N$,且通常是 $P\ll N$。如果用表格表示要修正项可能会更清楚些,

$$
\begin{bmatrix} & & \\ & \mathbf{U} & \\ & & \end{bmatrix}\cdot\begin{bmatrix}\mathbf{I}+\mathbf{V}^T\cdot\mathbf{A}^{-1}\cdot\mathbf{U}\end{bmatrix}^{-1}\cdot\begin{bmatrix} & & \\ & \mathbf{V}^T & \\ & & \end{bmatrix}
\tag{2.7.13}
$$

可以看出,其中要求逆的矩阵是 $P\times P$,而不是 $N\times N$。

如果注意到下面的事实,那么 Woodbury 公式与连续使用 Sherman-Morrison 公式之间的联系就很清楚了,即如果矩阵 $\mathbf{U}$ 由 P 个向量 $\mathbf{u}_0,\ldots,\mathbf{u}_{P-1}$ 按列构成,矩阵 $\mathbf{V}$ 由 P 个向量 $\mathbf{v}_0,\ldots,\mathbf{v}_{P-1}$ 按列构成,

$$
\mathbf{U}\equiv\begin{bmatrix}\mathbf{u}_0\end{bmatrix}\cdots\begin{bmatrix}\mathbf{u}_{P-1}\end{bmatrix}\qquad\mathbf{V}\equiv\begin{bmatrix}\mathbf{v}_0\end{bmatrix}\cdots\begin{bmatrix}\mathbf{v}_{P-1}\end{bmatrix}
\tag{2.7.14}
$$

则对 $\mathbf{A}$ 的修正可用两种方法表示:

$$
\left(\mathbf{A}+\sum_{k=0}^{P-1}\mathbf{u}_k\otimes\mathbf{v}_k\right)=(\mathbf{A}+\mathbf{U}\cdot\mathbf{V}^T)
\tag{2.7.15}
$$

(注意 $\mathbf{u}$ 和 $\mathbf{v}$ 的下标并不表示元素,而是用来区分不同的列向量。)

(2.7.15)式表明,如果 $\mathbf{A}^{-1}$ 已在存储器中,则可以用(2.7.12)式求一个 $P\times P$ 矩阵的逆,便可一次求出进行 P 次修改后的结果,或者也可用(2.7.5)式连续调用 P 次得到。

如果没有已存储好的 $\mathbf{A}^{-1}$,则必须按下列方法使用(2.7.12)式求解。欲求解的线性方程为

$$
\left(\mathbf{A}+\sum_{k=0}^{P-1}\mathbf{u}_k\otimes\mathbf{v}_k\right)\cdot\mathbf{x}=\mathbf{b}
\tag{2.7.16}
$$

首先求解 P 个辅助方程组:

$$\begin{aligned}\mathbf{A}\cdot\mathbf{z}_0&=\mathbf{u}_0\\\mathbf{A}\cdot\mathbf{z}_1&=\mathbf{u}_1\\&\cdots\\\mathbf{A}\cdot\mathbf{z}_{P-1}&=\mathbf{u}_{P-1}\end{aligned}\tag{2.7.17}$$

并根据得到的 $\mathbf{z}$ 按列构造矩阵 $\mathbf{Z}$:

$$\mathbf{Z}\equiv\begin{bmatrix}\mathbf{z}_0\end{bmatrix}\cdots\begin{bmatrix}\mathbf{z}_{P-1}\end{bmatrix}\tag{2.7.18}$$

第二步,求 $P\times P$ 矩阵的逆:

$$\mathbf{H}\equiv(\mathbf{1}+\mathbf{V}^T\cdot\mathbf{Z})^{-1}\tag{2.7.19}$$

最后,再求解一个辅助方程组:

$$\mathbf{A}\cdot\mathbf{y}=\mathbf{b}\tag{2.7.20}$$

根据已有这些量,可得到解为

$$\mathbf{x}=\mathbf{y}-\mathbf{Z}\cdot\left[\mathbf{H}\cdot(\mathbf{V}^T\cdot\mathbf{y})\right]\tag{2.7.21}$$

2.7.4 分区求逆

有时候会遇到一个矩阵(尽管不必是稀疏的),它可以通过分区来高效地求逆。假定 $N\times N$ 矩阵 $\mathbf{A}$ 被划分成

$$\mathbf{A}=\begin{bmatrix}\mathbf{P}&\mathbf{Q}\\\mathbf{R}&\mathbf{S}\end{bmatrix}\tag{2.7.22}$$

其中 $\mathbf{P}$ 和 $\mathbf{S}$ 分别为 $p\times p$ 和 $s\times s$ 的方阵($p+s=N$)。矩阵 $\mathbf{Q}$ 和 $\mathbf{R}$ 不必是方阵,分别为 $p\times s$ 和 $s\times p$ 的矩阵。

如果 $\mathbf{A}$ 的逆被划分为同样的形式,

$$\mathbf{A}^{-1}=\begin{bmatrix}\tilde{\mathbf{P}}&\tilde{\mathbf{Q}}\\\tilde{\mathbf{R}}&\tilde{\mathbf{S}}\end{bmatrix}\tag{2.7.23}$$

则 $\tilde{\mathbf{P}},\tilde{\mathbf{Q}},\tilde{\mathbf{R}},\tilde{\mathbf{S}}$ 与相应的 $\mathbf{P},\mathbf{Q},\mathbf{R},\mathbf{S}$ 大小相同,可以由下面的公式求出:

$$\begin{aligned}\tilde{\mathbf{P}}&=(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})^{-1}\\\tilde{\mathbf{Q}}&=-(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})^{-1}\cdot(\mathbf{Q}\cdot\mathbf{S}^{-1})\\\tilde{\mathbf{R}}&=-(\mathbf{S}^{-1}\cdot\mathbf{R})\cdot(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})^{-1}\\\tilde{\mathbf{S}}&=\mathbf{S}^{-1}+(\mathbf{S}^{-1}\cdot\mathbf{R})\cdot(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})^{-1}\cdot(\mathbf{Q}\cdot\mathbf{S}^{-1})\end{aligned}\tag{2.7.24}$$

或由其等价的公式求出:

$$\begin{aligned}\tilde{\mathbf{P}}&=\mathbf{P}^{-1}+(\mathbf{P}^{-1}\cdot\mathbf{Q})\cdot(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})^{-1}\cdot(\mathbf{R}\cdot\mathbf{P}^{-1})\\\tilde{\mathbf{Q}}&=-(\mathbf{P}^{-1}\cdot\mathbf{Q})\cdot(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})^{-1}\\\tilde{\mathbf{R}}&=-(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})^{-1}\cdot(\mathbf{R}\cdot\mathbf{P}^{-1})\\\tilde{\mathbf{S}}&=(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})^{-1}\end{aligned}\tag{2.7.25}$$

(2.7.24)式和(2.7.25)式中的小括号指出了只需计算一次重复因子(当然,根据结合律,矩阵相乘的次序可任意)。选择使用(2.7.24)式或是(2.7.25)式,取决于你是否要求 $\tilde{\mathbf{P}}$ 或 $\tilde{\mathbf{S}}$ 有简单的公式;或是是否重复的表达式 $(\mathbf{S}-\mathbf{R}\cdot\mathbf{P}^{-1}\cdot\mathbf{Q})^{-1}$ 要比表达式 $(\mathbf{P}-\mathbf{Q}\cdot\mathbf{S}^{-1}\cdot\mathbf{R})^{-1}$ 计算简单;或是 $\mathbf{P}$ 和 $\mathbf{S}$ 的相对大小;或是是否 $\mathbf{P}^{-1}$ 或 $\mathbf{S}^{-1}$ 已知。

另一个有用的公式是求分区矩阵的行列式，

$$\det \mathbf{A} = \det \mathbf{P} \det(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q}) = \det \mathbf{S} \det(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R}) \tag{2.7.26}$$

2.7.5　深入讨论：稀疏矩阵的索引存储

在 2.4 节中我们已经看到，二对角或带状对角矩阵可以用压缩格式存储，仅为那些可能不为零的元素分配存储空间，同时为保存方便，可能会浪费几个存储单元。对更为一般的稀疏矩阵情况又如何呢？对一个维数为 $N \times N$ 的稀疏矩阵，当它仅包含几倍于 N 的非零元时（一种典型情况），当然是低效的。而且经常是要为所有 N^2 个元素分配空间，在物理上这是不可能的。即使可以分配这么大的空间，它也是低效或应当禁止的，因为它把大量机器时间用于循环查找这些非零元。

显然需要一些索引存储方案，仅保存非零的矩阵元素，而且具有足够的辅助信息，以确定一个元素的逻辑位置，以及如何使各个元素对某共同的矩阵运算进行循环操作。遗憾的是，并不存在一个通用的标准方案。Knuth [7]、**耶鲁大学稀疏矩阵程序包**[6]和 ITPACK [8]中描述了几种方法。对大多数应用，建议使用 PCGPACK [9]中的存储方案，它跟 Bentley [10]描述的方法差不多相同，而且与耶鲁大学稀疏矩阵程序包[6]中的一种也很相似。这种可以称为**行索引稀疏存储方式**的方案的优点是，仅需大约两倍于非零矩阵元素数目的存储量（其他方法可能需要三至五倍）。为简便起见，仅讨论实践中经常出现的方阵的情况。

为表示一个 $N \times N$ 维的矩阵 $\mathbf{A}$，行索引方案设了两个一维的数组，称为 sa 和 ija。第一个存储矩阵元素的值，第二个存整数值。存储规则为

- sa 的前 N 个位置按顺序存储 $\mathbf{A}$ 的对角线上的矩阵元素。（注意，对角线元素即使为零也要存储，因为实际应用中大部分情况是对角线元素非零，故会造成一些轻微程度的存储浪费。）
- ija 的前 N 个位置保存数组 sa 的索引号，它指向矩阵相应行的第一个不在对角线上的元素。（如果那一行没有不在对角线上的元素，则其数值为上一行中最后存储元素在 sa 中的索引加 1。）
- ija 的第 0 个位置总是等于 $N+1$。（可据此判定 N。）
- ija 的第 N 个位置是最后一行最后一个非对角元在 sa 中的索引号加 1。（可据此判定矩阵非零元的数目，或数组 sa 和 ija 的元素个数。）sa 的第 N 个位置不用，其值任意。
- sa 中 $\geqslant N+1$ 的位置存有 $\mathbf{A}$ 不在对角线上的元素的值，它们按行排列，每行内按列排列。
- ija 中 $\geqslant N+1$ 的位置保存的是 sa 中对应索引号中元素的列值。

尽管这些规则看上去很任意，但实际上它们是一个非常精巧的存储方案。举例说明，考虑矩阵

$$\begin{bmatrix} 3. & 0. & 1. & 0. & 0. \\ 0. & 4. & 0. & 0. & 0. \\ 0. & 7. & 5. & 9. & 0. \\ 0. & 0. & 0. & 0. & 2. \\ 0. & 0. & 0. & 6. & 5. \end{bmatrix} \tag{2.7.27}$$

用行索引压缩存储，矩阵(2.7.27)用两个长度为 11 的数组来表示，如下所示：

索引 k	0	1	2	3	4	5	6	7	8	9	10
ija[k]	6	7	7	9	10	11	2	1	3	4	3
sa[k]	3.	4.	5.	0.	5.	x	1.	7.	9.	2.	6.

(2.7.28)

其中 x 的值是任意的。注意，根据存储规则可知，N 的值（即 5）为 ija[0] − 1，每个数组的长度为 ija[ija[0] − 1]，即 11。行 i 的对角线元素为 sa[i]，该行非对角线元素在 sa[k]中，其中 k 从 ija[i]到 ija[i + 1] − 1，当然上限要大于或等于下限（如在 C++ 语言中的循环）。

这里有一个程序 sprsin，它把一个矩阵从全部存储模式转换成行索引稀疏存储模式，它把小于特定阈值的元素都视为零元丢弃了。当然，稀疏存储模式的主要用途是对那些用全部存储模式机器根本存不下的矩阵，那时就不得不直接产生稀疏格式。然而，作为存储方案的一个精确的算法定义 sprsin，还是非常有用的，可用做大问题的子问题测试，以及对执行时间大于存储的情况，也迫使采用稀疏存储。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::sprsin(Mat_I_DP &a, const DP thresh, Vec_O_DP &sa, Vec_O_INT &ija)
将方阵 a[0..n-1][0..n-1]转换成行索引稀疏存储模式。仅保留数值大于阈值的元素。输出两个维数为 nmax(作为输入
参数)的线性数组:sa[0..]包含数组值,ija[0..]为索引号。输出 sa 和 ija 中的元素数目都是 ija[ija[0]-1](参见正文)。
{
    int i,j,k;

    int n=a.nrows();
    int nmax=sa.size();
    for (j=0;j<n;j++) sa[j]=a[j][j];        存储对角线元。
    ija[0]=n+1;                             如果存在,指向第零行的非对角元。
    k=n;
    for (i=0;i<n;i++) {                     行循环。
        for (j=0;j<n;j++) {                 列循环。
            if (fabs(a[i][j]) >= thresh && i != j) {
                if (++k > nmax) nrerror("sprsin: sa and ija too small");
                sa[k]=a[i][j];              存储非对角元及其列值。
                ija[k]=j;
            }
        }
        ija[i+1]=k+1;                       每行完成时,将指针指向下一个。
    }
}
```

用行索引稀疏存储模式保存的矩阵的惟一最重要用途就是在其右边乘以一个向量。事实上,存储模式的优化就是为了达到这个目的。因而下面的程序就非常简单了。

```
#include "nr.h"

void NR::sprsax(Vec_I_DP &sa, Vec_I_INT &ija, Vec_I_DP &x, Vec_O_DP &b)
将行索引稀疏矩阵存储为数组 sa 和 ija 乘以一个向量 x[0..n-1],得到向量 b[0..n-1]。
{
    int i,k;

    int n=x.size();
    if (ija[0] != n+1)
        nrerror("sprsax: mismatched vector and matrix");
    for (i=0;i<n;i++) {
        b[i]=sa[i]*x[i];                    从对角项开始。
        for (k=ija[i];k<ija[i+1];k++) {     对非对角项循环。
            b[i] += sa[k]*x[ija[k]];
        }
    }
}
```

在矩阵的转置的右边乘以一个向量也很简单(本节后面将要用到这种运算)。注意,转置矩阵并未实际构造。

```
#include "nr.h"

void NR::sprstx(Vec_I_DP &sa, Vec_I_INT &ija, Vec_I_DP &x, Vec_O_DP &b)
将存储成数组 sa 和 ija 的行索引稀疏矩阵的转置乘以向量 x[0..n-1],得到向量 b[0..n-1]。
{
    int i,j,k;

    int n=x.size();
    if (ija[0] != (n+1))
        nrerror("mismatched vector and matrix in sprstx");
    for (i=0;i<n;i++) b[i]=sa[i]*x[i];          从对角项开始。
    for (i=0;i<n;i++) {                         对非对角项循环。
        for (k=ija[i];k<ija[i+1];k++) {
            j=ija[k];
            b[j] += sa[k]*x[i];
        }
    }
}
```

事实上,因为选择行索引存储对行和列的处理完全不同,所以常遇到要对一个给定的、以行索引稀疏存

储模式存储的矩阵，构造其转置的操作。当该操作不可避免时，可以按下面的步骤操作：按列构造非对角元的索引（参见 8.4 节），则这些元素按列次序写入输出数组。在写每个元素时，要确定其行值并保存起来。最后，每列的元素按行次序存储。

```
#include "nr.h"

void NR::sprstp(Vec_I_DP &sa, Vec_I_INT &ija, Vec_O_DP &sb, Vec_O_INT &ijb)
构造稀疏方阵的转置，把按行索引稀疏存储的数组 sa 和 ija 转换成数组 sb 和 ijb。
{
    int j,jl,jm,jp,ju,k,m,n2,noff,inc,iv;
    DP v;

    n2=ija[0];                                          矩阵线性长度加2。
    for (j=0;j<n2-1;j++) sb[j]=sa[j];                   对角元。
    Vec_INT ija_vec((int *) &ija[n2],ija[n2-1]-ija[0]);
    Vec_INT ijb_vec(&ijb[n2],ija[n2-1]-ija[0]);
    indexx(ija_vec,ijb_vec);
    indexx对于长型变量的重载，按列构造非对角元的索引。
    for (j=n2,k=0;j < ija[n2-1];j++,k++) {
        ijb[j]=ijb_vec[k];
    }
    jp=0;
    for (k=ija[0];k<ija[n2-1];k++) {            对非对角元循环。
        m=ijb[k]+n2;                            使用索引表按(以前的)列存储。
        sb[k]=sa[m];
        for (j=jp;j<ija[m]+1;j++)               将索引填入省略的行中。
            ijb[j]=k;
        jp=ija[m]+1;                            用二分法查找元素m在哪一行，并将其存入
        jl=0;                                   ijb[k]。
        ju=n2-1;
        while (ju-jl > 1) {
            jm=(ju+jl)/2;
            if (ija[jm] > m) ju=jm; else jl=jm;
        }
        ijb[k]=jl;
    }
    for (j=jp;j<n2;j++) ijb[j]=ija[n2-1];
    for (j=0;j<n2-1;j++) {                      用Shell排序算法对每行进行最后一遍排序。
        jl=ijb[j+1]-ijb[j];
        noff=ijb[j];
        inc=1;
        do {
            inc *= 3;
            inc++;
        } while (inc <= jl);
        do {
            inc /= 3;
            for (k=noff+inc;k<noff+jl;k++) {
                iv=ijb[k];
                v=sb[k];
                m=k;
                while (ijb[m-inc] > iv) {
                    ijb[m]=ijb[m-inc];
                    sb[m]=sb[m-inc];
                    m -= inc;
                    if (m-noff+1 <= inc) break;
                }
                ijb[m]=iv;
                sb[m]=v;
            }
        } while (inc > 1);
    }
}
```

上面的程序中，包含了 8.1 节中介绍的一个排序算法，但调用了外部程序 indexx 来构造初始的列索引。

作为稀疏矩阵运算的最后一个示例，我们给出两个稀疏矩阵相乘的两个程序。它们对 13.10 节中所描述的技术是很有用的。

一般来说，两个稀疏矩阵的积本身并不是稀疏的。因此限制积矩阵的大小有两种方法：其一，事先已知

积为特定的稀疏模式,仅计算其中的这些元素;其二,计算所有的非零元,但仅保存那些数值在阈值以上的元素。第一种技术如果能够使用是非常有效的。稀疏模式可以通过用行索引稀疏存储格式设置的数组(如 ija)来说明,并且程序还构造一个相应的值数组(如 sa)。第二种技术有超出计算时间及不知输出规模的危险,故应谨慎使用。

对行索引存储,一种很自然的乘法运算是,左边的矩阵乘以右边矩阵的转置,这样可以进行行对行的运算,而不是行对列。因此程序是计算 $\mathbf{A}\cdot\mathbf{B}^T$,而不是 $\mathbf{A}\cdot\mathbf{B}$。这意味着在把右端矩阵提交矩阵的乘程序之前,必须运行 sprstp 程序对其进行转置。

两个实现程序,sprspm 进行"模式乘",而 sprstm 进行"阈值乘",它们在结构上非常相似。由于为两个输入流和输出流进行了不同的对角线或非对角线元素组合,两个算法都很复杂。

```
#include "nr.h"

void NR::sprspm(Vec_I_DP &sa, Vec_I_INT &ija, Vec_I_DP &sb, Vec_I_INT &ijb,
    Vec_O_DP &sc, Vec_I_INT &ijc)
矩阵乘法运算 A·B^T,其中 A 和 B 是以行索引存储方式存储的两个稀疏矩阵,B^T 为 B 的转置。其中 sa 和 ija 存储矩阵
A,sb 和 ijb 存储矩阵 B。本程序仅计算由输入索引数组 ijc 预说明的矩阵乘积中的元素,ijc 本身不被修改。输入数组 sc
和 ijc 一起给出了积矩阵的行索引存储方式。对稀疏矩阵乘法,执行本程序前,应经常调用 sprstp,以构造已知矩阵的转
置,并存入到数组 sb 和 ijb。
{
    int i,ijma,ijmb,j,m,ma,mb,mbb,mn;
    DP sum;

    if (ija[0] != ijb[0] || ija[0] != ijc[0])
        nrerror("sprspm: sizes do not match");
    for (i=0;i<ijc[0]-1;i++) {          行循环。
        m=i+1;
        j=m-1;                          设置初值,以便第一遍循环处理对角元。
        mn=ijc[i];
        sum=sa[i]*sb[i];
        for (;;) {                      对每个输出元进行的主循环。
            mb=ijb[j];
            for (ma=ija[i];ma<ija[i+1];ma++) {
            对A的行元素循环。下面的循环逻辑是对对角元及非对角元进行各种组合。
                ijma=ija[ma];
                if (ijma == j) sum += sa[ma]*sb[j];
                else {
                    while (mb < ijb[j+1]) {
                        ijmb=ijb[mb];
                        if (ijmb == i) {
                            sum += sa[i]*sb[mb++];
                            continue;
                        } else if (ijmb < ijma) {
                            mb++;
                            continue;
                        } else if (ijmb == ijma) {
                            sum += sa[ma]*sb[mb++];
                            continue;
                        }
                        break;
                    }
                }
            }
            for (mbb=mb;mbb<ijb[j+1];mbb++) {          处理完B的行的剩余部分。
                if (ijb[mbb] == i) sum += sa[i]*sb[mbb];
            }
            sc[m-1]=sum;
            sum=0.0;                                   为下一次循环重置索引。
            if (mn >= ijc[i+1]) break;
            j=ijc[(m= ++mn)-1];
        }
    }
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::sprstm(Vec_I_DP &sa, Vec_I_INT &ija, Vec_I_DP &sb, Vec_I_INT &ijb,
    const DP thresh, Vec_O_DP &sc, Vec_O_INT &ijc)
矩阵乘法运算 A·B^T.其中 A 和 B 是以行索引存储方式存储的两个稀疏矩阵,B^T 为 B 的转置。其中 sa 和 ija 存储矩阵
A,sb 和 ijb 存储矩阵 B。本程序计算矩阵积中的所有元素(可能不是稀疏的),但仅存储那些数值超过阈值的元素。输
出数组 sc 和 ijc 用行索引存储方式给出积矩阵。在调用程序中提供的这些矩阵,应具有至少使得矩阵相乘所满足的尺
寸大小。对稀疏矩阵乘法,执行本程序前应先调用 sprstp,以构造已知矩阵的转置,并存入到 sb 和 ijb 中。
{
    int i,ijma,ijmb,j,k,ma,mb,mbb;
    DP sum;

    if (ija[0] != ijb[0]) nrerror("sprstm: sizes do not match");
    int nmax=sc.size();
    ijc[0]=k=ija[0];
    for (i=0;i<ija[0]-1;i++) {                          A的行循环。
        for (j=0;j<ijb[0]-1;j++) {                      B的行循环。
            if (i == j) sum=sa[i]*sb[j]; else sum=0.0e0;
            mb=ijb[j];
            for (ma=ija[i];ma<ija[i+1];ma++) {
            对A的行元素循环，下面的循环逻辑是对角元及非对角元进行各种组合。
                ijma=ija[ma];
                if (ijma == j) sum += sa[ma]*sb[j];
                else {
                    while (mb < ijb[j+1]) {
                        ijmb=ijb[mb];
                        if (ijmb == i) {
                            sum += sa[i]*sb[mb++];
                            continue;
                        } else if (ijmb < ijma) {
                            mb++;
                            continue;
                        } else if (ijmb == ijma) {
                            sum += sa[ma]*sb[mb++];
                            continue;
                        }
                        break;
                    }
                }
            }
            for (mbb=mb;mbb<ijb[j+1];mbb++) {           处理完B的行的剩余部分。
                if (ijb[mbb] == i) sum += sa[i]*sb[mbb];
            }
            if (i == j) sc[i]=sum;                      输出结果。
            else if (fabs(sum) > thresh) {
                if (k > nmax-1) nrerror("sprstm: sc and ijc too small");
                sc[k]=sum;
                ijc[k++]=j;
            }
        }
        ijc[i+1]=k;
    }
}
```

2.7.6　深入讨论:共轭梯度法求解稀疏方程组

共轭梯度法为求解 $N \times N$ 线性方程组提供了一种相当通用的方法:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.7.29}$$

这些求解大型稀疏方程组的方法,其优点在于它们对 **A** 的引用仅包括 **A** 与一个向量相乘,或其转置与一个向量相乘。正如所见,这些运算对适当存储的稀疏矩阵是非常高效的。可能会要求提供一些函数,希望它们对这些稀疏矩阵乘法尽可能高效地运行。我们现在提供一些通用的程序,下面的 linbcg 能利用函数求解线性方程组(2.7.29)。

最简单的常见共轭梯度算法[11~13]是仅当 **A** 为对称正定时求解(2.7.29)式。其思想基于极小化下面的函数:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}\cdot\mathbf{A}\cdot\mathbf{x} - \mathbf{b}\cdot\mathbf{x} \tag{2.7.30}$$

当其梯度

$$\nabla f = \mathbf{A}\cdot\mathbf{x} - \mathbf{b} \tag{2.7.31}$$

为零时,该函数取极小值,这相当于(2.7.29)式。极小化的实现是通过产生一个搜索方向 $\mathbf{p}_k$ 的数列,从而改进极小化的因变量 $\mathbf{x}_k$。每一步都是要找一个量α_k 来极小化$f(\mathbf{x}_k + \alpha_k\mathbf{p}_k)$,并且把 $\mathbf{x}_{k+1}$ 设为等于新的点 $\mathbf{x}_k + \alpha_k\mathbf{p}_k$。$\mathbf{p}_k$ 和 $\mathbf{x}_k$ 也是这样构造的,$\mathbf{x}_{k+1}$是 f 在所走过方向$\{\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{k-1}\}$上的向量空间内的极小点。经过 N 次迭代后便可到达整个向量空间的极小点,即(2.7.29)式的解。

在后面的 10.6 节中,将把这个"普通的"共轭梯度算法推广到任意非线性函数的情况。这里,我们仍是求解线性的函数,但不必是正定的或对称的,其一种重要的推广是**双共轭梯度法**。一般来说,该算法并非是简单的函数极小化问题。它要构造四个向量序列 $\mathbf{r}_k, \bar{\mathbf{r}}_k, \mathbf{p}_k, \bar{\mathbf{p}}_k, k = 0, 1, \ldots$。只要提供初始向量 $\mathbf{r}_0$ 和 $\bar{\mathbf{r}}_0$,并设 $\mathbf{p}_0 = \mathbf{r}_0, \bar{\mathbf{p}}_0 = \bar{\mathbf{r}}_0$,则可得下面的递推式:

$$\begin{aligned}
\alpha_k &= \frac{\bar{\mathbf{r}}_k\cdot\mathbf{r}_k}{\bar{\mathbf{p}}_k\cdot\mathbf{A}\cdot\mathbf{p}_k} \\
\mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k\mathbf{A}\cdot\mathbf{p}_k \\
\bar{\mathbf{r}}_{k+1} &= \bar{\mathbf{r}}_k - \alpha_k\mathbf{A}^T\cdot\bar{\mathbf{p}}_k \\
\beta_k &= \frac{\bar{\mathbf{r}}_{k+1}\cdot\mathbf{r}_{k+1}}{\bar{\mathbf{r}}_k\cdot\mathbf{r}_k} \\
\mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k \\
\bar{\mathbf{p}}_{k+1} &= \bar{\mathbf{r}}_{k+1} + \beta_k\bar{\mathbf{p}}_k
\end{aligned} \tag{2.7.32}$$

该向量序列满足"双正交"条件

$$\bar{\mathbf{r}}_i\cdot\mathbf{r}_j = \mathbf{r}_i\cdot\bar{\mathbf{r}}_j = 0, \qquad j < i \tag{2.7.33}$$

及"双共轭"条件

$$\bar{\mathbf{p}}_i\cdot\mathbf{A}\cdot\mathbf{p}_j = \mathbf{p}_i\cdot\mathbf{A}^T\cdot\bar{\mathbf{p}}_j = 0, \qquad j < i \tag{2.7.34}$$

还是相互正交的,

$$\bar{\mathbf{r}}_i\cdot\mathbf{p}_j = \mathbf{r}_i\cdot\bar{\mathbf{p}}_j = 0, \qquad j < i \tag{2.7.35}$$

证明这些性质的直接推导见[14]。只要递推不因出现分母为零而提前中断,其结束必在 $m \leqslant N$ 步之内,且 $\mathbf{r}_m = \bar{\mathbf{r}}_m = 0$。这是因为,最多 N 步后就已经用完了所构造向量的新的正交方向。

使用该算法求解方程组(2.7.29),给定解的一个初始猜测 $\mathbf{x}_0$。将 $\mathbf{r}_0$选为"残差":

$$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\cdot\mathbf{x}_0 \tag{2.7.36}$$

并设 $\bar{\mathbf{r}}_0 = \mathbf{r}_0$。然后构造改进的估算序列:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k \tag{2.7.37}$$

实现(2.7.32)式的递推。(2.7.37)式保证递推得到的 $\mathbf{r}_{k+1}$事实上是对应于 $\mathbf{x}_{k+1}$的残差 $\mathbf{b} - \mathbf{A}\cdot\mathbf{x}_{k+1}$。因为 $\mathbf{r}_m = 0$,故 $\mathbf{x}_m$是(2.7.29)式的解。

尽管不能保证整个过程不会中断,或是对一般的矩阵 $\mathbf{A}$ 会变得不稳定,然而实践中,这种情况很少见。更重要的是,仅当有精确的算法时,才会至多 N 次迭代后便精确地结束。由于舍入误差,必须将运算看做是一个纯迭代的过程,当符合一些适当的误差准则时才中止迭代。

普通的共轭梯度算法是双共轭梯度算法的一个特例,即 $\mathbf{A}$ 是对称的,并选择 $\bar{\mathbf{r}}_0 = \mathbf{r}_0$。然后对所有的 k,令 $\bar{\mathbf{r}}_k = \mathbf{r}_k$ 及 $\bar{\mathbf{p}}_k = \mathbf{p}_k$。可以省去这部分计算,将算法的工作量减半。共轭梯度法可解释为极小化(2.7.30)式。如果 $\mathbf{A}$ 是正定及对称的,算法不会中断(理论上如此)。下面的程序 linbcg 在输入对称的 $\mathbf{A}$ 时,实际上简化成普通的共轭梯度法了,但它却进行了所有的冗余计算。

通用算法的另一种变化是对于对称但非正定的矩阵 $\mathbf{A}$,这时选择 $\bar{\mathbf{r}}_0 = \mathbf{A}\cdot\mathbf{r}_0$ 而不是 $\bar{\mathbf{r}}_0 = \mathbf{r}_0$。这种情况下,对所有的 k,$\bar{\mathbf{r}}_k = \mathbf{A}\cdot\mathbf{r}_k$ 及 $\bar{\mathbf{p}}_k = \mathbf{A}\cdot\mathbf{p}_k$。因此,算法等价于普通的共轭梯度算法,只是所有的点积运算 $\mathbf{a}\cdot\mathbf{b}$ 被 $\mathbf{a}\cdot\mathbf{A}\cdot\mathbf{b}$ 所代替。它称为"极小残差"算法,因为它对应于如下函数的连续极小化操作:

$$\Phi(\mathbf{x}) = \frac{1}{2}\mathbf{r}\cdot\mathbf{r} = \frac{1}{2}|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}|^2 \tag{2.7.38}$$

其中连续地迭代 $\mathbf{x}_k$，就可以在由共轭梯度法产生的搜索方向 $\mathbf{p}_k$ 的相同集上极小化 Φ。该算法已推广为各种非对称矩阵的算法。**通用极小残差法**(GMRES，参见[9,15])可能是这些方法中最为稳健的了。

注意(2.7.38)式给出

$$\nabla\Phi(\mathbf{x}) = \mathbf{A}^T\cdot(\mathbf{A}\cdot\mathbf{x}-\mathbf{b}) \tag{2.7.39}$$

对任意非奇异矩阵 $\mathbf{A}$，$\mathbf{A}^T\cdot\mathbf{A}$ 是对称和正定的。因此，可以试着通过将普通的共轭梯度算法应用于如下问题来求解(2.7.29)式：

$$(\mathbf{A}^T\cdot\mathbf{A})\cdot\mathbf{x} = \mathbf{A}^T\cdot\mathbf{b} \tag{2.7.40}$$

但实际不用这样做。矩阵 $\mathbf{A}^T\cdot\mathbf{A}$ 的条件数是 $\mathbf{A}$ 的条件数的平方(条件数的定义见 2.6 节)。一个大的条件数不仅增加了所需的迭代次数，而且限制了可能得到的解的精度。用双共轭梯度法对原始矩阵 $\mathbf{A}$ 求解总要好一些。

至此尚未言及这些算法的收敛速度问题。对于条件好的矩阵，即"靠近"单位阵的矩阵，普通的共轭梯度算法效果很好。这提示我们把这些方法应用于(2.7.29)式的**先决条件**形式，则

$$(\tilde{\mathbf{A}}^{-1}\cdot\mathbf{A})\cdot\mathbf{x} = \tilde{\mathbf{A}}^{-1}\cdot\mathbf{b} \tag{2.7.41}$$

对接近于 $\mathbf{A}$ 的某个 $\tilde{\mathbf{A}}$ 求解线性方程组可能要容易些，其中 $\tilde{\mathbf{A}}^{-1}\cdot\mathbf{A}\approx\mathbf{1}$，它允许算法在较少的步数内收敛。矩阵 $\tilde{\mathbf{A}}$ 称为一个**预条件器**[11]，这里给出的整个方案即为**预条件双共轭梯度法**，或称为 PBCG。

为了能高效实现，PBCG 算法介绍了一个附加的向量集 $\mathbf{z}_k$ 和 $\bar{\mathbf{z}}_k$，定义为

$$\tilde{\mathbf{A}}\cdot\mathbf{z}_k = \mathbf{r}_k \qquad 和 \qquad \tilde{\mathbf{A}}^T\cdot\bar{\mathbf{z}}_k = \bar{\mathbf{r}}_k \tag{2.7.42}$$

并修改了(2.7.32)式中 α_k、β_k、$\mathbf{p}_k$ 和 $\bar{\mathbf{p}}_k$ 的定义：

$$\begin{aligned}
\alpha_k &= \frac{\bar{\mathbf{r}}_k\cdot\mathbf{z}_k}{\bar{\mathbf{p}}_k\cdot\mathbf{A}\cdot\mathbf{p}_k}\\
\beta_k &= \frac{\bar{\mathbf{r}}_{k+1}\cdot\mathbf{z}_{k+1}}{\bar{\mathbf{r}}_k\cdot\mathbf{z}_k}\\
\mathbf{p}_{k+1} &= \mathbf{z}_{k+1}+\beta_k\mathbf{p}_k\\
\bar{\mathbf{p}}_{k+1} &= \bar{\mathbf{z}}_{k+1}+\beta_k\bar{\mathbf{p}}_k
\end{aligned} \tag{2.7.43}$$

对下面的程序 linbcg，我们要求读者自己来提供求解(2.7.42)式的辅助线性方程组的程序。如果不知道用什么做预条件器 $\tilde{\mathbf{A}}$，则不妨使用 $\mathbf{A}$ 的对角线部分，甚至是单位矩阵，这样收敛的负担完全落在双共轭梯度算法本身上。

下面的程序 linbcg，基于以前 Anne Greenbaum 所写的一个程序(参见本节参考文献[13]，它是一个不同的、稍简单些的实现)。

怎样能构成"好"的收敛依赖于应用。因此程序 linbcg 提供 4 种选择，可以对输入标记 itol 进行设置。如果设 itol = 1，则当值 $|\mathbf{A}\cdot\mathbf{x}-\mathbf{b}|/|\mathbf{b}|$ 小于输入量 tol 时，迭代停止。如果设 itol = 2，则所需的准则为

$$|\tilde{\mathbf{A}}^{-1}\cdot(\mathbf{A}\cdot\mathbf{x}-\mathbf{b})|/|\tilde{\mathbf{A}}^{-1}\cdot\mathbf{b}| < \mathrm{tol} \tag{2.7.44}$$

如果 itol = 3，则程序使用其自身对 $\mathbf{x}$ 误差的估计，并要求其大小除以 $\mathbf{x}$ 的值小于 tol。设置 itol = 4 与 itol = 3 情况相同，只是用误差的最大量(绝对值)和 $\mathbf{x}$ 的最大分量代替了向量的大小(即用 L_∞ 范数代替了 L_2 范数)。需要在实践中试着寻找到最适合自己的收敛准则。

输出方面，err 是实际所达到的容差。如果返回的数 iter 表明所允许的最大迭代次数 itmax 未被超过，则 err 应比 tol 小。如果要进一步迭代，可不改变返回的量再次调用程序。然而两次调用之间，程序会失去有关所组成的共轭梯度子空间的记忆，故不应使其经常返回，至少每次应有 N 次迭代。

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include "nr.h"
using namespace std;
```

```
void NR::linbcg(Vec_I_DP &b, Vec_IO_DP &x, const int itol, const DP tol,
    const int itmax, int &iter, DP &err)
```

用迭代双共轭梯度法求解 $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$,b[0..n-1]已知,求 x[0..n-1]。输入时 x[0..n-1]应设为解的初始猜测(或全为零);itol 为 1,2,3 或 4,分别说明使用何种收敛测试(见正文);itmax 为所允许迭代的最大次数;tol 为所需的收敛容差。输出 x[0..n-1]被置为改进后的解,iter 为实际进行的迭代次数。err 为估计的误差。矩阵 A 的引用仅通过用户提供的程序 atimes 引用,它计算 A 或其转置与一个向量的乘积,而程序 asolve,它对某个预条件矩阵 $\tilde{\mathbf{A}}$(可能仅是 A 的平凡对角线部分)求解 $\tilde{\mathbf{A}}\cdot\mathbf{x}=\mathbf{b}$ 或 $\tilde{\mathbf{A}}^T\cdot\mathbf{x}=\mathbf{b}$。

```
{
DP ak,akden,bk,bkden=1.0,bknum,bnrm,dxnrm,xnrm,zminrm,znrm;
本程序中使用双精度为好。
const DP EPS=1.0e-14;
int j;

int n=b.size();
Vec_DP p(n),pp(n),r(n),rr(n),z(n),zz(n);
iter=0;                          计算初始余量。
atimes(x,r,0);                   atimes的输入为x[0..n-1], 输出为r[0..n-1], 最后的0
for (j=0;j<n;j++) {              表示使用矩阵本身(而不是其转置)。
    r[j]=b[j]-r[j];
    rr[j]=r[j];
}
//atimes(r,rr,0);                去掉本行的注释符号即得到该算法的"最小余量"
if (itol == 1) {                 变形。
    bnrm=snrm(b,itol);
    asolve(r,z,0);               asolve的输入是的r[0..n-1],输出为z[0..n-1];
}                                最后的0表示使用的是矩阵Ã(而不是其转置)。
else if (itol == 2) {
    asolve(b,z,0);
    bnrm=snrm(z,itol);
    asolve(r,z,0);
}
else if (itol == 3 || itol == 4) {
    asolve(b,z,0);
    bnrm=snrm(z,itol);
    asolve(r,z,0);
    znrm=snrm(z,itol);
} else nrerror("illegal itol in linbcg");
cout << fixed << setprecision(6);
while (iter < itmax) {           主循环。
    ++iter;
    asolve(rr,zz,1);             最后的1表示使用转置矩阵 Ã^T。
    for (bknum=0.0,j=0;j<n;j++) bknum += z[j]*rr[j];
    Calculate coefficient bk and direction vectors p and pp.
    if (iter == 1) {
        for (j=0;j<n;j++) {
            p[j]=z[j];
            pp[j]=zz[j];
        }
    } else {
        bk=bknum/bkden;
        for (j=0;j<n;j++) {
            p[j]=bk*p[j]+z[j];
            pp[j]=bk*pp[j]+zz[j];
        }
    }
    bkden=bknum;                 计算系数ak, 新的迭代量x, 以及新的余量r和rr。
    atimes(p,z,0);
    for (akden=0.0,j=0;j<n;j++) akden += z[j]*pp[j];
    ak=bknum/akden;
    atimes(pp,zz,1);
    for (j=0;j<n;j++) {
        x[j] += ak*p[j];
        r[j] -= ak*z[j];
        rr[j] -= ak*zz[j];
    }
    asolve(r,z,0);               求解 Ã·z = r, 并检查停止准则。
    if (itol == 1)
        err=snrm(r,itol)/bnrm;
    else if (itol == 2)
        err=snrm(z,itol)/bnrm;
    else if (itol == 3 || itol == 4) {
        zminrm=znrm;
```

```
            znrm=snrm(z,itol);
            if (fabs(zminrm-znrm) > EPS*znrm) {
                dxnrm=fabs(ak)*snrm(p,itol);
                err=znrm/fabs(zminrm-znrm)*dxnrm;
            } else {
                err=znrm/bnrm;          误差可能不精确，故再次循环。
                continue;
            }
            xnrm=snrm(x,itol);
            if (err <= 0.5*xnrm) err /= xnrm;
            else {
                err=znrm/bnrm;          误差可能不精确，故再次循环。
                continue;
            }
        }
        cout << "iter=" << setw(4) << iter+1 << setw(12) << err << endl;
        if (err <= tol) break;
    }
}
```

程序 linbcg 使用了下面这个小实用程序来计算向量的范数。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::snrm(Vec_I_DP &sx, const int itol)
根据 itol 的值，计算向量 sx[0..n-1]的两个范数之一，用在程序 linbcg 中。
{
    int i,isamax;
    DP ans;

    int n=sx.size();
    if (itol <= 3) {
        ans = 0.0;
        for (i=0;i<n;i++) ans += sx[i]*sx[i]; 向量大小的范数。
        return sqrt(ans);
    } else {
        isamax=0;
        for (i=0;i<n;i++) {                     最大元的范数。
            if (fabs(sx[i]) > fabs(sx[isamax])) isamax=i;

        }
        return fabs(sx[isamax]);
    }
}
```

为使程序 atimes 和 asolve 的说明清楚些，在此列出简单的版本，它假定矩阵 A 以行索引稀疏格式存于某处。

```
#include "nr.h"

extern Vec_INT *ija_p;
extern Vec_DP *sa_p;                            矩阵存于某处。

void NR::atimes(Vec_I_DP &x, Vec_O_DP &r, const int itrnsp)
{
    if (itrnsp) sprstx(*sa_p,*ija_p,x,r);
    else sprsax(*sa_p,*ija_p,x,r);
}
#include "nr.h"

extern Vec_INT *ija_p;
extern Vec_DP *sa_p;                            矩阵存于某处。

void NR::asolve(Vec_I_DP &b, Vec_O_DP &x, const int itrnsp)
{
    int i;

    int n=b.size();
    for(i=0;i<n;i++) x[i]=((*sa_p)[i] != 0.0 ? b[i]/(*sa_p)[i] : b[i]);
    矩阵Ã是A的对角线部分，存于sa的前n个元素。因为该矩阵的转置对角元不变，
    故没有用标记itrnsp。
}
```

参考文献和深入阅读

Tewarson, R.P. 1973, *Sparse Matrices* (New York: Academic Press). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter I.3 (by J.K. Reid). [2]

George, A., and Liu, J.W.H. 1981, *Computer Solution of Large Sparse Positive Definite Systems* (Englewood Cliffs, NJ: Prentice-Hall). [3]

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.). [4]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [5]

Eisenstat, S.C., Gursky, M.C., Schultz, M.H., and Sherman, A.H. 1977, *Yale Sparse Matrix Package*, Technical Reports 112 and 114 (Yale University Department of Computer Science). [6]

Knuth, D.E. 1997, *Fundamental Algorithms*, 3rd ed., vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §2.2.6. [7]

Kincaid, D.R., Respess, J.R., Young, D.M., and Grimes, R.G. 1982, *ACM Transactions on Mathematical Software*, vol. 8, pp. 302–322. [8]

PCGPAK User's Guide (New Haven: Scientific Computing Associates, Inc.). [9]

Bentley, J. 1986, *Programming Pearls* (Reading, MA: Addison-Wesley), §9. [10]

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), Chapters 4 and 10, particularly §§10.2–10.3. [11]

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), Chapter 8. [12]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill). [13]

Fletcher, R. 1976, in *Numerical Analysis Dundee 1975*, Lecture Notes in Mathematics, vol. 506, A. Dold and B Eckmann, eds. (Berlin: Springer-Verlag), pp. 73–89. [14]

Saad, Y., and Schulz, M. 1986, *SIAM Journal on Scientific and Statistical Computing*, vol. 7, pp. 856–869. [15]

Bunch, J.R., and Rose, D.J. (eds.) 1976, *Sparse Matrix Computations* (New York: Academic Press).

Duff, I.S., and Stewart, G.W. (eds.) 1979, *Sparse Matrix Proceedings 1978* (Philadelphia: S.I.A.M.).

2.8 Vandermonde 矩阵和 Toeplitz 矩阵

在 2.4 节中,三对角矩阵的情况是特殊处理的,因为这种特殊类型的线性系统可使求解运算仅为 N 阶,而不是一般线性问题所需的 N^3 阶。如果存在这样的特殊类型,重要的是要了解它们。如果碰巧要求解含有特殊类型的问题,它所节省的计算量是很可观的。

本节中处理两种特殊形式的矩阵,它们可以在 N^2 阶运算内求解。虽然不像三对角形式那么好,但强于一般情况(除了运算量外,这两种类型的矩阵没有别的共同之处)。第一种类型的矩阵称为 Vandermonde(范德蒙德)矩阵,它经常出现在与多项式拟合、矩的分布重构以及其他一些有关内容的问题中。例如在 3.5 节中出现过的 Vandermonde 矩阵问题。第二种类型称为 Toeplitz(特普利茨)矩阵,很容易在解卷积和信号处理问题中遇到,在 13.7 节中还将讨论 Toeplitz 矩阵。

这并不是已知特殊类型矩阵的全部。**Hilbert 矩阵**的元素形式为 $a_{ij}=1/(i+j+1)$, $i,j=0,\ldots,N-1$,可以用非常精确的整数算法来求逆。由于其数值情况很特殊,用其他方法求逆是很困难的(详情请参考本节参考文献[1])。已在 2.7 节中讨论过的 Sherman-Morrison 和 Woodbury 公式,有时可以用来把新的特殊形式转换成已有的形式。本节参考文献[2]中还介绍了一些其他特殊形式。还没发现其他形式的矩阵像本节讨论的这两种矩阵这么常见。

2.8.1 深入讨论:Vandermonde 矩阵

$N\times N$ 的 Vandermonde 矩阵完全由 N 个数 $x_0,x_1,\ldots,x_{N-1}$ 确定,这 N^2 个元素按其整数次幂排列 x_i^j, $i,j=0,\ldots,N-1$。显然有两种可能的形式,它取决于是 i 作为行、j 作为列,还是相反的情况。如果是前面一种情况,得到线性系统的方程组如下:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \cdots & x_{N-1}^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} \tag{2.8.1}$$

将矩阵乘法展开就会看到，该式求解的未知系数 c_i，与 N 对横坐标和纵坐标(x_j, y_j)之间满足多项式关系。3.5 节中将精确地求解该问题，那里给出的程序是用我们将要说明的方法来求解(2.8.1)式的。

行列交换的方程组形式为

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_{N-1} \\ x_0^2 & x_1^2 & \cdots & x_{N-1}^2 \\ & & \cdots & \\ x_0^{N-1} & x_1^{N-1} & \cdots & x_{N-1}^{N-1} \end{bmatrix} \cdot \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \cdots \\ w_{N-1} \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ \cdots \\ q_{N-1} \end{bmatrix} \tag{2.8.2}$$

将其展开会发现它与**矩问题**(problem of moment)有关。给定 N 个点 x_i 的值，求未知的权 w_i，使其与前面的 N 个矩的给定值 q_i 相匹配(要了解关于此问题更多的内容，请参考本节参考文献[3])。本节中给出的程序用来求解(2.8.2)式。

求解(2.8.1)式和(2.8.2)式的方法，都与拉格朗日(Lagrange)多项式插值公式密切相关，这将在后面的 3.1 节中讨论。尽管如此，下面的推导还是可以理解的。

设 $P_j(x)$为 $N-1$ 阶多项式，其定义为

$$P_j(x) = \prod_{\substack{n=0 \\ (n\neq j)}}^{N-1} \frac{x - x_n}{x_j - x_n} = \sum_{k=0}^{N-1} A_{jk} x^k \tag{2.8.3}$$

最后一个等式的意义为，将矩阵乘积展开成多项式形式并合并同类项，再把其系数定义为矩阵 A_{ij} 的元素。

多项式 $P_j(x)$一般是 x 的函数。但会发现该函数的特殊之处，对所有 x_i，$i \neq j$，其值为零；对 $x = x_j$，其值为 1。换言之，

$$P_j(x_i) = \delta_{ij} = \sum_{k=0}^{N-1} A_{jk} x_i^k \tag{2.8.4}$$

(2.8.4)式表明 A_{jk} 是(2.8.2)式中元素 x_i^k 所构成的矩阵的逆，x_i^k 中的下标代表列索引。因此，(2.8.2)式的解就是该逆矩阵与右端项的乘积，

$$w_j = \sum_{k=0}^{N-1} A_{jk} q_k \tag{2.8.5}$$

至于该问题转置的形式(2.8.1)，根据转置矩阵的逆是逆矩阵的转置这一事实，有

$$c_j = \sum_{k=0}^{N-1} A_{kj} y_k \tag{2.8.6}$$

用 3.5 节中的程序来实现它。

下面的问题是如何找到一个好的方法，将(2.8.3)式中的各个单项乘出来，以便得到 A_{jk} 的元素。这个问题很有必要写出来，希望读者自己通过程序发现如何解决此问题。先定义 $P(x)$：

$$P(x) \equiv \prod_{n=0}^{N-1} (x - x_n) \tag{2.8.7}$$

求出其系数，然后通过对另外的一项做综合除法，得到 P_j 的分子和分母项(关于综合除法详见 5.3 节)。因为每一次这样的除法处理复杂度仅为 N 阶，所以总的处理过程为 N^2 阶。

必须说明，Vandermonde 问题由于其内在属性因而是严重病态的(在 5.8 节中还要提到，其原因与要使切比雪夫(Chebyshev)拟合达到相当的精度一样：存在高阶多项式，其形式非常接近于零。因此，舍入误差会引入到这些多项式前几项的实际系数中)。因此始终采用双精度数来计算 Vandermonde 问题是个好办法。

下面求解(2.8.2)式的程序是由 G. B. Rybicki 提供的。

```
#include "nr.h"

void NR::vander(Vec_I_DP &x, Vec_O_DP &w, Vec_I_DP &q)
求解 Vandermonde 线性系统 ∑_{i=0}^{N-1} x_i^k w_i = q_k,(k = 0,...,N-1)。输入为向量 x[0..n-1]和 q[0..n-1],输出为向量
w[0..n-1]。
{
    int i,j,k;
    DP b,s,t,xx;

    int n=q.size();
    Vec_DP c(n);
    if (n == 1) w[0]=q[0];
    else {
        for (i=0;i<n;i++) c[i]=0.0;          初始化数组。
        c[n-1] = -x[0];                      主多项式系数由递归得到。
        for (i=1;i<n;i++) {
            xx = -x[i];
            for (j=(n-1-i);j<(n-1);j++) c[j] += xx*c[j+1];
            c[n-1] += xx;
        }
        for (i=0;i<n;i++) {                  依次对每一个次因子……
            xx=x[i];
            t=b=1.0;
            s=q[n-1];
            for (k=n-1;k>0;k--) {            ……进行综合除法。
                b=c[k]+xx*b;
                s += q[k-1]*b;               用右端项进行矩阵乘,
                t=xx*t+b;
            }
            w[i]=s/t;                        用分母除。
        }
    }
}
```

2.8.2 深入讨论:Toeplitz 矩阵

一个 $N\times N$ 的 Toeplitz 矩阵由给出的 $2N-1$ 个数 R_k 组成,$k=-N+1,\ldots,-1,0,1,\ldots,N-1$。这些数作为矩阵的常量元素,按(左上至右下)对角线顺序排列:

$$\begin{bmatrix} R_0 & R_{-1} & R_{-2} & \cdots & R_{-(N-2)} & R_{-(N-1)} \\ R_1 & R_0 & R_{-1} & \cdots & R_{-(N-3)} & R_{-(N-2)} \\ R_2 & R_1 & R_0 & \cdots & R_{-(N-4)} & R_{-(N-3)} \\ \cdots & & & \cdots & & \\ R_{N-2} & R_{N-3} & R_{N-4} & \cdots & R_0 & R_{-1} \\ R_{N-1} & R_{N-2} & R_{N-3} & \cdots & R_1 & R_0 \end{bmatrix} \tag{2.8.8}$$

线性 Toeplitz 问题可写为

$$\sum_{j=0}^{N-1} R_{i-j}x_j = y_i \qquad (i=0,\ldots,N-1) \tag{2.8.9}$$

其中 $x_j, j=0,\ldots,N-1$ 为要求解的未知量。

如果对所有 k 都有 $R_k=R_{-k}$,则 Toeplitz 矩阵是对称的。Levinson[4]设计了一种求对称的 Toeplitz 问题的快速算法,即**边界法**。它是一个递归过程,用来求解 $M+1$ 维的 Toeplitz 问题:

$$\sum_{j=0}^{M} R_{i-j}x_j^{(M)} = y_i \qquad (i=0,\ldots,M) \tag{2.8.10}$$

依次取 $M=0,1,\ldots$ 直到 $M=N-1$,便可求得最后要求的结果。向量 $x_j^{(M)}$ 是第 M 次得到的结果,只有当 M 达到 $N-1$ 时,才是要求的解。

Levinson 的方法在标准的教科书中(例如[5])都有详细的介绍,但这种方法推广到**非对称矩阵**的情况却鲜为人知。这里给出一个由 G.B.Rybicki 提供的推导。

在下面由第 M 步到第 $M+1$ 步的推导中,会发现递推解 $x^{(M)}$ 的变化如下:

$$\sum_{j=0}^{M} R_{i-j}x_j^{(M)} = y_i \qquad i=0,\ldots,M \tag{2.8.11}$$

变成

$$\sum_{j=0}^{M} R_{i-j} x_j^{(M+1)} + R_{i-(M+1)} x_{M+1}^{(M+1)} = y_i \qquad i = 0, \ldots, M+1 \tag{2.8.12}$$

消去 y_i，得

$$\sum_{j=0}^{M} R_{i-j} \left(\frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \right) = R_{i-(M+1)} \qquad i = 0, \ldots, M \tag{2.8.13}$$

或者将 $i \to M - i$，并将 $j \to M - j$，

$$\sum_{j=0}^{M} R_{j-i} G_j^{(M)} = R_{-(i+1)} \tag{2.8.14}$$

其中，

$$G_j^{(M)} \equiv \frac{x_{M-j}^{(M)} - x_{M-j}^{(M+1)}}{x_{M+1}^{(M+1)}} \tag{2.8.15}$$

将其改写为

$$x_{M-j}^{(M+1)} = x_{M-j}^{(M)} - x_{M+1}^{(M+1)} G_j^{(M)} \qquad j = 0, \ldots, M \tag{2.8.16}$$

因此，如果由此递推式求得 M 阶的值 $x^{(M)}$ 和 $G^{(M)}$，以及一个 $M+1$ 阶的值 $x_{M+1}^{(M+1)}$，则其余所有的 $x_j^{(M+1)}$ 便可求得。幸运的是，值 $x_{M+1}^{(M+1)}$ 可以如下求得：在(2.8.12)式中，令 $i = M+1$，

$$\sum_{j=0}^{M} R_{M+1-j} x_j^{(M+1)} + R_0 x_{M+1}^{(M+1)} = y_{M+1} \tag{2.8.17}$$

对未知的 $M+1$ 阶量 $x_j^{(M+1)}$，可以用 G 中的低阶量来替换。由

$$G_{M-j}^{(M)} = \frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \tag{2.8.18}$$

运算的结果为

$$x_{M+1}^{(M+1)} = \frac{\sum_{j=0}^{M} R_{M+1-j} x_j^{(M)} - y_{M+1}}{\sum_{j=0}^{M} R_{M+1-j} G_{M-j}^{(M)} - R_0} \tag{2.8.19}$$

剩下的惟一问题是求 G 的递推关系。在构造递推关系之前必须指出，对原来的非对称矩阵的线性问题，有两个完全不同的解集，即右端项解(我们已讨论的)和左端项解 z_i。左端项解的形式差别仅在于要处理的方程为

$$\sum_{j=0}^{M} R_{j-i} z_j^{(M)} = y_i \qquad i = 0, \ldots, M \tag{2.8.20}$$

然后，对该集合进行类似的操作，得

$$\sum_{j=0}^{M} R_{i-j} H_j^{(M)} = R_{i+1} \tag{2.8.21}$$

其中，

$$H_j^{(M)} \equiv \frac{z_{M-j}^{(M)} - z_{M-j}^{(M+1)}}{z_{M+1}^{(M+1)}} \tag{2.8.22}$$

(与(2.8.14)式～(2.8.15)式比较。)提及左端项解的原因在于，在(2.8.21)式中，要是用 $y_i \to R_{i+1}$ 替换右端项，则 H_j 与 x_j 满足相同的方程。因此，由(2.8.19)式可以很快推出

$$H_{M+1}^{(M+1)} = \frac{\sum_{j=0}^{M} R_{M+1-j} H_j^{(M)} - R_{M+2}}{\sum_{j=0}^{M} R_{M+1-j} G_{M-j}^{(M)} - R_0} \tag{2.8.23}$$

如果使用替换 $y_i \to R_{-(i+1)}$，并用相同的符号表示，则 G 满足 z 所表示的相同的方程。即

$$G_{M+1}^{(M+1)} = \frac{\sum_{j=0}^{M} R_{j-M-1} G_j^{(M)} - R_{-M-2}}{\sum_{j=0}^{M} R_{j-M-1} H_{M-j}^{(M)} - R_0} \tag{2.8.24}$$

同理,变换(2.8.16)式以及 z 所表示方程的对应量,得最后的方程为

$$\begin{aligned} G_j^{(M+1)} &= G_j^{(M)} - G_{M+1}^{(M+1)} H_{M-j}^{(M)} \\ H_j^{(M+1)} &= H_j^{(M)} - H_{M+1}^{(M+1)} G_{M-j}^{(M)} \end{aligned} \tag{2.8.25}$$

现在,来看看方程的初值

$$x_0^{(0)} = y_0/R_0 \qquad G_0^{(0)} = R_{-1}/R_0 \qquad H_0^{(0)} = R_1/R_0 \tag{2.8.26}$$

可以递推下去。在第 M 步求解过程中,由(2.8.23)式和(2.8.24)式求得 $H_{M+1}^{(M+1)}$, $G_{M+1}^{(M+1)}$,然后由(2.8.25)式求得另外的 $H^{(M+1)}$ 和 $G^{(M+1)}$ 的内容。由此,向量 $x^{(M+1)}$ 和/或 $z^{(M+1)}$ 便很容易求出了。

下面的程序完成此功能。它将(2.8.25)式的第二个方程形式变为

$$H_{M-j}^{(M+1)} = H_{M-j}^{(M)} - H_{M+1}^{(M+1)} G_j^{(M)} \tag{2.8.27}$$

这样计算可以"同址"进行。

注意如果 $R_0 = 0$,则上述算法失败。事实上,由于边界法没有采用主元素法,如果将原来的 Toeplitz 矩阵中主对角线上的小元素化成零,算法也会失败(与 2.4 节中讨论的三对角矩阵算法对比一下)。如果算法失败,矩阵并不一定是奇异的——可能必须用一个慢一点但更为通用的算法,例如,使用主元素法的 LU 分解来求解该问题。

实现(2.8.23)式~(2.8.27)式的程序也是由 G. B. Rybicki 提供的。注意,程序中的 r[n-1+j]与上面的 R_j 相当,因此,数组 r 的下标变化范围为 0~2N-2。

```
#include "nr.h"

void NR::toeplz(Vec_I_DP &r, Vec_O_DP &x, Vec_I_DP &y)
求解 Toeplitz 系统 ∑_{j=0}^{N-1} R_{(N-1+i-j)} x_j = y_i (i = 0,...,N-1)。Toeplitz 矩阵不必是对称的。y[0..n-1]和 r[0..2*n-2]为
输入数组;x[0..n-1]为输出数组。
{
    int j,k,m,m1,m2,n,n1;
    DP pp,pt1,pt2,qq,qt1,qt2,sd,sgd,sgn,shn,sxn;

    n=y.size();
    n1=n-1;
    if (r[n1] == 0.0) nrerror("toeplz-1 singular principal minor");
    x[0]=y[0]/r[n1];                为递推初始化。
    if (n1 == 0) return;
    Vec_DP g(n1),h(n1);
    g[0]=r[n1-1]/r[n1];
    h[0]=r[n1+1]/r[n1];
    for (m=0;m<n;m++) {             递推的主循环。
        m1=m+1;
        sxn = -y[m1];               计算(2.8.19)式中x 的分子和分母项。
        sd = -r[n1];
        for (j=0;j<m+1;j++) {
            sxn += r[n1+m1-j]*x[j];
            sd += r[n1+m1-j]*g[m-j];
        }
        if (sd == 0.0) nrerror("toeplz-2 singular principal minor");
        x[m1]=sxn/sd;               由(2.8.16)式得到x。
        for (j=0;j<m+1;j++)
            x[j] -= x[n1]*g[m-j];
        if (m1 == n1) return;
        sgn = -r[n1-m1-1];          由(2.8.24)式和(2.8.23)式计算G和H的分子和分母项。
        shn = -r[n1+m1+1];
        sgd = -r[n1];
        for (j=0;j<m+1;j++) {
            sgn += r[n1+j-m1]*g[j];
            shn += r[n1+m1-j]*h[j];
            sgd += r[n1+j-m1]*h[m-j];
        }
        if (sgd == 0.0) nrerror("toeplz-3 singular principal minor");
        g[m1]=sgn/sgd;              由此得到G和H。
        h[m1]=shn/sd;
```

```
            k=m;
            m2=(m+2) >> 1;
            pp=g[m1];
            qq=h[m1];
            for (j=0;j<m2;j++) {
                pt1=g[j];
                pt2=g[k];
                qt1=h[j];
                qt2=h[k];
                g[j]=pt1-pp*qt2;
                g[k]=pt2-pp*qt1;
                h[j]=qt1-qq*pt2;
                h[k--]=qt2-qq*pt1;
            }
        }                                   返回进行下一次递推。
        nrerror("toeplz - should not arrive here!");
    }
```

如果在工作中遇到求解较大型的 Toeplitz 系统，将会发现有一些声称为“新的、快速的”算法，它只要求 $N(\log N)^2$ 阶操作，而 Levinson 算法需 N^2 阶操作，但这些算法非常复杂，可参考 Bunch[6]和 de Hoog[7]的论文，其中有这些算法的相关内容。

参考文献和深入阅读

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), Chapter 5 [also treats some other special forms].

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), §19. [1]

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley) [2]

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), pp. 394ff. [3]

Levinson, N., Appendix B of N. Wiener, 1949, *Extrapolation, Interpolation and Smoothing of Stationary Time Series* (New York: Wiley). [4]

Robinson, E.A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, NJ: Prentice-Hall), pp. 163ff. [5]

Bunch, J.R. 1985, *SIAM Journal on Scientific and Statistical Computing*, vol. 6, pp. 349–364. [6]

de Hoog, F. 1987, *Linear Algebra and Its Applications*, vol. 88/89, pp. 123–138. [7]

2.9　深入讨论：Cholesky 分解

如果一个方阵 **A** 恰好是对称和正定的，则它有一特殊的更有效的三角分解。“对称的”意味着 $a_{ij} = a_{ji}$，$i, j = 0, \ldots, N-1$，正定的意味着

$$\mathbf{v} \cdot \mathbf{A} \cdot \mathbf{v} > 0 \qquad \text{对所有向量 } \mathbf{v} \tag{2.9.1}$$

（在第 11 章中，将会看到正定的等价解释为，**A** 的所有特征值均为正数。）对称正定的矩阵相当特殊，它们在某些应用中会经常出现，故最好知道其特殊的分解，称为 Cholesky（楚列斯基）分解。使用这种方法求解线性方程组时，比其他方法快两倍。

Cholesky 分解并不是寻找任意的下三角和上三角因子 **L** 和 **U**，而是构造一个下三角矩阵 **L**，其转置 $\mathbf{L}^T$ 本身作为上三角部分。换言之，可以把(2.3.1)式换成

$$\mathbf{L} \cdot \mathbf{L}^T = \mathbf{A} \tag{2.9.2}$$

该分解有时被引用为对矩阵 **A**“取平方根”。$\mathbf{L}^T$ 的元素与 **L** 的元素的关系自然是

$$L^T_{ij} = L_{ji} \tag{2.9.3}$$

写出(2.9.2)式的元素，很容易得到与(2.3.12)式和(2.3.13)式类似的结果，

$$L_{ii} = \left(a_{ii} - \sum_{k=0}^{i-1} L_{ik}^2 \right)^{1/2} \tag{2.9.4}$$

及

$$L_{ji}=\frac{1}{L_{ii}}\left(a_{ij}-\sum_{k=0}^{i-1}L_{ik}L_{jk}\right)\qquad j=i+1,i+2,\ldots,N-1 \tag{2.9.5}$$

如果按顺序 $i=0,1,\ldots,N-1$ 使用(2.9.4)式和(2.9.5)式,就会发现右端出现的 L 在需要时已经确定了。还有,仅有 $j\geqslant i$ 的元素 a_{ij} 被引用(因为 **A** 是对称的,这些包含了全部的信息)。然后,很方便地就可把因子 **L** 覆盖写入 **A** 的子对角线部分(下三角,但不包括对角线),而保留输入的 **A** 的上三角的值不变。因而只需一个长度为 N 的附加向量存储 **L** 的对角线部分。其运算次数为 $N^3/6$ 次内层循环(包括一次乘和一次减)及 N 次平方根运算。前面已提到过,这差不多要比 **A** 的 LU 分解快两倍(因它没有考虑矩阵的对称性)。

一个直截了当的实现为

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::choldc(Mat_IO_DP &a, Vec_O_DP &p)
对给定的正定对称矩阵 a[0..n-1][0..n-1],本程序构造其 Cholesky 分解,A = L·L^T。输入量仅需给出 a 的上三角部
分,而且它不被程序修改。Cholesky 因子 L 返回于 a 的下三角部分,但其对角线元素返回到 p[0..n-1]。
{
    int i,j,k;
    DP sum;

    int n=a.nrows();
    for (i=0;i<n;i++) {
        for (j=i;j<n;j++) {
            for (sum=a[i][j],k=i-1;k>=0;k--) sum -= a[i][k]*a[j][k];
            if (i == j) {                    由于舍入误差的缘故, a不是正定的。
                if (sum <= 0.0)
                    nrerror("choldc failed");
                p[i]=sqrt(sum);
            } else a[j][i]=sum/p[i];
        }
    }
}
```

此时,可能会考虑是否需要选主元。令人高兴的是,Cholesky 分解在数值上极为稳定,无需选主元。而程序 choldc 的失败,只能表明矩阵 **A**(或是由于舍入误差的缘故,有另一个与其非常接近的矩阵)不是正定的。事实上,程序 choldc 是一种检测一个对称矩阵是否是正定的很有效的方法(对于这种应用,可以把对 nrerror 的调用换成某种信号缓和些的方法)。

一旦矩阵被分解,可通过回代过程用三角因子来求解线性方程。这种直接的实现方法为

```
#include "nr.h"

void NR::cholsl(Mat_I_DP &a, Vec_I_DP &p, Vec_I_DP &b, Vec_O_DP &x)
求解 n 维线性方程组 A·x = b,其中 a 为正定对称矩阵。输入量 a[0..n-1][0..n-1]和 p[0..n-1]是程序 choldc 输出的
结果。只有 a 的下三角部分被访问到。b[0..n-1]输入量是右端项向量。解向量返回至 x[0..n-1]中。a,n 和 p 没有
被修改,可留做不同右端项 b 的连续调用。b 也不被修改。
{
    int i,k;
    DP sum;

    int n=a.nrows();
    for (i=0;i<n;i++) {                  求解L·y=b,将y存于x。
        for (sum=b[i],k=i-1;k>=0;k--) sum -= a[i][k]*x[k];
        x[i]=sum/p[i];
    }
    for (i=n-1;i>=0;i--) {               求解 L^T · x = y。
        for (sum=x[i],k=i+1;k<n;k++) sum -= a[k][i]*x[k];
        x[i]=sum/p[i];
    }
}
```

程序 choldc 和 cholsl 的一个典型应用是,对描述数据适合某种模型的协方差矩阵的求逆,参见 15.6 节。在此应用及其他许多应用中,经常需要 $\mathbf{L}^{-1}$。该矩阵的下三角部分可以高效地从 choldc 的输出中求得。

```
for (i=0;i<n;i++) {
    a[i][i]=1.0/p[i];
    for (j=i+1;j<n;j++) {
        sum=0.0;
        for (k=i;k<j;k++) sum -= a[j][k]*a[k][i];
        a[j][i]=sum/p[j];
    }
}
```

参考文献和深入阅读

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/1.

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), §4.9.2.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.5.

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §4.2.

2.10　深入讨论:QR 分解

还有另一种矩阵的因式分解,有时非常有用,即所谓的 **QR** 分解,

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{2.10.1}$$

其中 **R** 是上三角的,而 **Q** 是正交的,即

$$\mathbf{Q}^T \cdot \mathbf{Q} = \mathbf{1} \tag{2.10.2}$$

其中 $\mathbf{Q}^T$ 为 **Q** 的转置。尽管对一般的长方形矩阵,这种分解都存在,但我们还是限定维数为 $N \times N$ 的方阵的情况

跟前面介绍的其他矩阵分解(LU,SVD,Cholesky)一样,QR 分解也可用于求解线性方程系统。欲求解

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.10.3}$$

首先构造 $\mathbf{Q}^T \cdot \mathbf{b}$,然后通过回代过程求解

$$\mathbf{R} \cdot \mathbf{x} = \mathbf{Q}^T \cdot \mathbf{b} \tag{2.10.4}$$

因为 QR 分解的运算操作差不多是 LU 分解的两倍,所以它不用于典型的线性方程系统。然而,会遇到一些特殊情况,QR 分解是可选的方法。

标准的 QR 分解算法包括连续的 Householder 变换(在 11.2 节讨论)。写一个 Householder 矩阵,格式为 $\mathbf{1} - \mathbf{u} \otimes \mathbf{u}/c$,其中 $c = \frac{1}{2}\mathbf{u} \cdot \mathbf{u}$。合适的 Householder 矩阵用在给定的矩阵上,可以使位于某选定元素之下的矩阵列元素全部零化。因此,设第一个 Householder 矩阵 $\mathbf{Q}_0$,使 **A** 的第零列第零个元素之下的元素全化为零。同样,$\mathbf{Q}_1$ 使第一列第一个元素之下的元素全化为零,如此直到 $\mathbf{Q}_{n-2}$。所以

$$\mathbf{R} = \mathbf{Q}_{n-2} \cdots \mathbf{Q}_0 \cdot \mathbf{A} \tag{2.10.5}$$

因为 Householder 矩阵是正交的,

$$\mathbf{Q} = (\mathbf{Q}_{n-2} \cdots \mathbf{Q}_0)^{-1} = \mathbf{Q}_0 \cdots \mathbf{Q}_{n-2} \tag{2.10.6}$$

在大部分应用中,不需要显式地形成 **Q**,只需将其存成(2.6.10)式的因子形式,除非矩阵 **A** 非常接近奇异,否则不必选主元。本节参考文献[1]中给出了一种通用的包括选主元在内的长方形矩阵的 QR 算法。对方阵,一种实现方法如下:

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::qrdcmp(Mat_IO_DP &a, Vec_O_DP &c, Vec_O_DP &d, bool &sing)
```

构造 a[0..n-1][0..n-1]的 QR 分解。上三角矩阵 **R** 返回到 a 的上三角部分,**R** 的对角元返回到 d[0..n-1]。正交矩阵 **Q** 被表示成 n-1 个 Householder 矩阵 $\mathbf{Q}_0 \ldots \mathbf{Q}_{n-2}$的积的形式,其中 $\mathbf{Q}_j = 1 - \mathbf{u}_j \otimes \mathbf{u}_j / c_j$。对 $i = 0, \ldots, j-1$,$\mathbf{u}_j$ 的第 i 个元素为零,非零元返回在 a[i][j]中,$i = j, \cdots, n-1$,如果分解中出现奇异的情况,sing 值返回为 true,但本例中仍要完成分解;否则,返回为 false。

```
{
    int i,j,k;
    DP scale,sigma,sum,tau;

    int n=a.nrows();
    sing=false;
    for (k=0;k<n-1;k++) {
        scale=0.0;
        for (i=k;i<n;i++) scale=MAX(scale,fabs(a[i][k]));
        if (scale == 0.0) {                    奇异情况。
            sing=true;
            c[k]=d[k]=0.0;
        } else {                               形成 Q_k 及 Q_k · A。
            for (i=k;i<n;i++) a[i][k] /= scale;
            for (sum=0.0,i=k;i<n;i++) sum += SQR(a[i][k]);
            sigma=SIGN(sqrt(sum),a[k][k]);
            a[k][k] += sigma;
            c[k]=sigma*a[k][k];
            d[k] = -scale*sigma;
            for (j=k+1;j<n;j++) {
                for (sum=0.0,i=k;i<n;i++) sum += a[i][k]*a[i][j];
                tau=sum/c[k];
                for (i=k;i<n;i++) a[i][j] -= tau*a[i][k];
            }
        }
    }
    d[n-1]=a[n-1][n-1];
    if (d[n-1] == 0.0) sing=true;
}
```

下面的程序 qrsolv 用于求解线性系统。在许多应用中，仅需要算法的(2.10.4)式部分，因此将其分离成单独的程序 rsolv。

```
#include "nr.h"

void NR::qrsolv(Mat_I_DP &a, Vec_I_DP &c, Vec_I_DP &d, Vec_IO_DP &b)
求解 n 维线性方程组 A·x=b。输入量 a[0..n-1][0..n-1]，c[0..n-1]和 d[0..n-1]是程序 qrdcmp 的输出结果，而且
不被修改。输入量 b[0..n-1]为右端向量，并被输出的解向量所覆盖
{
    int i,j;
    DP sum,tau;

    int n=a.nrows();
    for (j=0;j<n-1;j++) {          形成 Q^T · b。
        for (sum=0.0,i=j;i<n;i++) sum += a[i][j]*b[i];
        tau=sum/c[j];
        for (i=j;i<n;i++) b[i] -= tau*a[i][j];
    }
    rsolv(a,d,b);                  求解 R · x = Q^T · b。
}

#include "nr.h"

void NR::rsolv(Mat_I_DP &a, Vec_I_DP &d, Vec_IO_DP &b)
求解 n 维线性方程组 R·x=b，其中 R 为存于 a 和 d 的上三角矩阵。输入量 a[0..n-1][0..n-1]和 d[0..n-1]是程序
qrdcmp的输出结果，而且不被修改。输入量 b[0..n-1]为右端向量，并被输出的解向量所覆盖。
{
    int i,j;
    DP sum;

    int n=a.nrows();
    b[n-1] /= d[n-1];
    for (i=n-2;i>=0;i--) {
        for (sum=0.0,j=i+1;j<n;j++) sum += a[i][j]*b[j];
        b[i]=(b[i]-sum)/d[i];
    }
}
```

关于如何使用 QR 分解构造正交基并求解最小二乘的问题，可详细参见本节参考文献[2](对此问题，推荐使用 2.6 节的 SVD 方法，因为它对病态情况具有较强的诊断能力)。

2.10.1　更新 QR 分解

有些数值算法包含求解一连串的线性系统，而每个系统与前一个仅有很小的差别。不必每次都用 $O(N^3)$ 次运算从零开始求解，经常可以在 $O(N^2)$ 次运算内更新矩阵因子，再用这个新的因子来求解下一个线性方程组。由于选主元的原因，LU 分解的更新很复杂。然而 QR 分解对一种很常见的更新却是非常容易的，

$$\mathbf{A} \to \mathbf{A} + \mathbf{s} \otimes \mathbf{t} \tag{2.10.7}$$

（与(2.7.1)式比较）实际上，用其等价的形式来计算更方便，

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \quad \to \quad \mathbf{A}' = \mathbf{Q}' \cdot \mathbf{R}' = \mathbf{Q} \cdot (\mathbf{R} + \mathbf{u} \otimes \mathbf{v}) \tag{2.10.8}$$

可以在(2.10.7)式和(2.10.8)式之间，反复使用 $\mathbf{Q}$ 是正交的这一事实，得到

$$\mathbf{t} = \mathbf{v} \quad 且 \quad \mathbf{s} = \mathbf{Q} \cdot \mathbf{u} \quad 或 \quad \mathbf{u} = \mathbf{Q}^T \cdot \mathbf{s} \tag{2.10.9}$$

算法[2]有两个阶段。第一阶段，应用 $N-1$ 次雅可比旋转（见 11.1 节），将 $\mathbf{R} + \mathbf{u} \otimes \mathbf{v}$ 约化成上 Hessenberg 形式。另一个 $N-1$ 次雅可比旋转将该上 Hessenberg 矩阵转换成新的上三角矩阵 $\mathbf{R}'$。矩阵 $\mathbf{Q}'$ 为 $\mathbf{Q}$ 的 $2(N-1)$ 次雅可比旋转的乘积形式。在应用中通常需要 $\mathbf{Q}^T$，该算法很容易将 $\mathbf{Q}$ 换成 $\mathbf{Q}^T$ 的形式计算。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::qrupdt(Mat_IO_DP &r, Mat_IO_DP &qt, Vec_IO_DP &u, Vec_I_DP &v)
对某个给出的 n×n 矩阵的 QR 分解，计算矩阵 Q·(R + u⊗v) 的 QR 分解。这些量的维数为 r[0..n-1][0..n-1]，
qt[0..n-1][0..n-1]，u[0..n-1]及 v[0..n-1]。注意 Q^T 为输入量，从程序 qt 中返回得到。
{
    int i,k;

    int n=u.size();
    for (k=n-1;k>=0;k--)                    寻找最大的k使 u[k] ≠ 0。
        if (u[k] != 0.0) break;
    if (k < 0) k=0;
    for (i=k-1;i>=0;i--) {                  将R + u⊗v转换成上Hessenberg形式。
        rotate(r,qt,i,u[i],-u[i+1]);
        if (u[i] == 0.0)
            u[i]=fabs(u[i+1]);
        else if (fabs(u[i]) > fabs(u[i+1]))
            u[i]=fabs(u[i])*sqrt(1.0+SQR(u[i+1]/u[i]));
        else u[i]=fabs(u[i+1])*sqrt(1.0+SQR(u[i]/u[i+1]));
    }
    for (i=0;i<n;i++) r[0][i] += u[0]*v[i];
    for (i=0;i<k;i++)                       将上Hessenberg矩阵转换成上三角形式。
        rotate(r,qt,i,r[i][i],-r[i+1][i]);
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::rotate(Mat_IO_DP &r, Mat_IO_DP &qt, const int i, const DP a,
    const DP b)
对某个给定的矩阵 r[0..n-1][0..n-1]和 qt[0..n-1][0..n-1]，按每个矩阵的 i 行和 i+1 行进行雅可比旋转。a 和 b 为
旋转参数：cos θ = a/√(a²+b²)，sin θ = b/√(a²+b²)。
{
    int j;
    DP c,fact,s,w,y;

    int n=r.nrows();
    if (a == 0.0) {
        c=0.0;
        s=(b >= 0.0 ? 1.0 : -1.0);      避免不必要的上溢或下溢。
    } else if (fabs(a) > fabs(b)) {
        fact=b/a;
        c=SIGN(1.0/sqrt(1.0+(fact*fact)),a);
        s=fact*c;
    } else {
```

```
            fact=a/b;
            s=SIGN(1.0/sqrt(1.0+(fact*fact)),b);
            c=fact*s;
        }
        for (j=i;j<n;j++) {          r预乘以雅可比旋转。
            y=r[i][j];
            w=r[i+1][j];
            r[i][j]=c*y-s*w;
            r[i+1][j]=s*y+c*w;
        }
        for (j=0;j<n;j++) {          qt预乘以雅可比旋转。
            y=qt[i][j];
            w=qt[i+1][j];
            qt[i][j]=c*y-s*w;
            qt[i+1][j]=s*y+c*w;
        }
    }
```

9.7 节中将使用 QR 分解及其更新程序。

参考文献和深入阅读

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/8. [1]

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §§5.2, 5.3, 12.5. [2]

2.11 矩阵求逆是否是 N^3 阶运算

在结束本章之前做个算术游戏，进一步探讨矩阵求逆的问题。从一个看似简单的问题开始：

对两个 2×2 矩阵进行矩阵相乘，要进行多少次乘法运算，

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \cdot \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} \tag{2.11.1}$$

进行 8 次运算，明确地写下来即

$$\begin{aligned} c_{00} &= a_{00}\times b_{00} + a_{01}\times b_{10} \\ c_{01} &= a_{00}\times b_{01} + a_{01}\times b_{11} \\ c_{10} &= a_{10}\times b_{00} + a_{11}\times b_{10} \\ c_{11} &= a_{10}\times b_{01} + a_{11}\times b_{11} \end{aligned} \tag{2.11.2}$$

是否能仅用 7 次乘法，就写出求 c 的公式(在读下文之前，不妨自己试一试)。

事实上，这套公式是由 Strassen[1]总结的。这些公式为

$$\begin{aligned} Q_0 &\equiv (a_{00}+a_{11})\times(b_{00}+b_{11}) \\ Q_1 &\equiv (a_{10}+a_{11})\times b_{00} \\ Q_2 &\equiv a_{00}\times(b_{01}-b_{11}) \\ Q_3 &\equiv a_{11}\times(-b_{00}+b_{10}) \\ Q_4 &\equiv (a_{00}+a_{01})\times b_{11} \\ Q_5 &\equiv (-a_{00}+a_{10})\times(b_{00}+b_{01}) \\ Q_6 &\equiv (a_{01}-a_{11})\times(b_{10}+b_{11}) \end{aligned} \tag{2.11.3}$$

由此得

$$
\begin{aligned}
c_{00} &= Q_0 + Q_3 - Q_4 + Q_6 \\
c_{10} &= Q_1 + Q_3 \\
c_{01} &= Q_2 + Q_4 \\
c_{11} &= Q_0 + Q_2 - Q_1 + Q_5
\end{aligned}
\tag{2.11.4}
$$

虽然比(2.11.2)式中少了一次乘法，但却多了许多加法和减法，暂时还看不出来这样做有什么优点。但注意到(2.11.3)式中，a 和 b 并不互相引用，因此当 a 和 b 本身又是矩阵时，(2.11.3)式和(2.11.4)式就有效。对非常大的两个矩阵相乘问题(对某个整数 m，维数 $N=2^m$)，可将其递归地分解为四分之一、十六分之一等。关键是要注意到：节省后的运算量不仅是一个"7/8"的因子，而递归的每一级都是这个因子，它总共将矩阵乘的运算量由 N^3 级减少到 $N^{\log_2 7}$。

那么如何计算(2.11.3)式和(2.11.4)式中额外的加法呢？对大的 N 值，可证明(2.11.3)式和(2.11.4)式中蕴含的加法次数是乘法次数的6倍。但是，如果 N 非常大，该常数因子就不能与从 N^3 降到 $N^{\log_2 7}$ 的指数变化相提并论了。

在进行"快速"矩阵乘法的研究中，Strassen 还得到了关于矩阵求逆的令人吃惊的结果。假定矩阵

$$
\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \quad 和 \quad \begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix}
\tag{2.11.5}
$$

互为逆矩阵，则 c 可由 a 按下列操作得到[比较(2.7.11)式和(2.7.25)式]：

$$
\begin{aligned}
R_0 &= \mathrm{Inverse}(a_{00}) \\
R_1 &= a_{10} \times R_0 \\
R_2 &= R_0 \times a_{01} \\
R_3 &= a_{10} \times R_2 \\
R_4 &= R_3 - a_{11} \\
R_5 &= \mathrm{Inverse}(R_4) \\
c_{01} &= R_2 \times R_5 \\
c_{10} &= R_5 \times R_1 \\
R_6 &= R_2 \times c_{10} \\
c_{00} &= R_0 - R_0 \\
c_{11} &= -R_5
\end{aligned}
\tag{2.11.6}
$$

在(2.11.6)式中，"逆"运算出现两次。该运算被解释为，若 a 和 c 为数值，则求其倒数；若 a 和 c 本身为了矩阵，则求其逆矩阵。想像一下，对一个维数 $N=2^m$ 的大型矩阵求逆时，用二分法递归求解的情况。在每一步递归过程中，都要使求逆运算的次数加倍，但这仅意味着总共进行 N 次除法。所以在递归使用(2.11.6)式时，除法并不是主要运算。事实上，(2.11.6)式主要的运算是它的六次乘法。因为这些乘法可用复杂度为 $N^{\log_2 7}$ 的算法实现，故矩阵求逆也能如此。

其具有实用性：如果估算一下，要多大的 N 才能体现出指数 3 和指数 $\log_2 7 = 2.807$ 的差别，实际上这就是浪费笔墨了。由于递归 Strassen 算法的内在复杂性，就会发现 LU 分解法并不面临过时的危险。

另一方面，如果读者对这种游戏感兴趣，可以再试试：(1)能仅用三次实数乘法，就求出两个复数 $(a+ib)$ 和 $(c+id)$ 的乘积吗(答案参见 5.4 节)？(2)能在计算一般 x 的四次多项式时，对许多不同的 x 值，每次仅用三次乘法就能求出其值吗(答案参见 5.3 节)？

参考文献和深入阅读

Strassen, V. 1969, *Numerische Mathematik*, vol. 13, pp. 354–356. [1]

Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley).

Winograd, S. 1971, *Linear Algebra and Its Applications*, vol. 4, pp. 381–388.

Pan, V. Ya. 1980, *SIAM Journal on Computing*, vol. 9, pp. 321–342.

Pan, V. 1984, *How to Multiply Matrices Faster*, Lecture Notes in Computer Science, vol. 179 (New York: Springer-Verlag)

Pan, V. 1984, *SIAM Review*, vol. 26, pp. 393–415. [More recent results that show that an exponent of 2.496 can be achieved — theoretically!]

第3章　内插法和外推法

3.0　引言

有时，仅知道函数 $f(x)$ 在一系列点 $x_0, x_1, \ldots, x_{N-1}$ 上的数值（假如 $x_0 < \ldots < x_{N-1}$），但并没有 $f(x)$ 的解析式来求得任意一点的数值。例如，$f(x_i)$ 可能来源于一些物理测量值，或来源于不能用简单函数形式表示的冗长的数值运算。通常 x_i 是等间距的，但不是必须如此。

现在的任务是，画一条通过（可能越过）x_i 的平滑曲线，来对任意 x 估算 $f(x)$ 的值。如果所求的 x 处于 x_i 的最大值与最小值之间，则称之为**内插法**，如果 x 超出此范围则称为**外推法**，后者冒险性更大一些。

内插法和外推法是在已知点之间或越过已知点，用某些可能的函数形式来模拟此函数。函数的形式应该是很常见的，而且能够近似于应用中可能遇到的几大类函数。函数中最常用的形式为多项式（参见3.1节）。有理函数（多项式的商）也被证明是非常有用的（参见3.2节）。正弦和余弦等三角函数应用于**三角插值法**及相关的傅里叶方法中，将在第12章和第13章中讨论。

在内容全面的数学文献中，都有关于何类函数用做插值函数能很好地进行估算的定理。遗憾的是，这些定理通常几乎完全没用：如果我们对函数很熟悉，能够自如地应用有关的定理，也就不用必须根据数值表来内插函数了。

内插法与**函数逼近**有关，但也有不同之处。函数逼近的任务是，找一个近似的（但容易计算的）函数用以取代原来较复杂的函数。在内插法情况中，是要在不由自己选择的点上给出函数 f 的值。而对于函数逼近，是为了得到近似函数，并允许在任意设定点计算出函数 f 的值。我们将在第5章中讨论函数逼近问题。

可以很容易地构造一些病态函数使内插法失败。例如，考虑函数

$$f(x) = 3x^2 + \frac{1}{\pi^4}\ln\left[(\pi - x)^2\right] + 1 \tag{3.0.1}$$

它除了 $x = \pi$ 之外都有定义，而 $x = \pi$ 时无定义，其他情况值有正有负。任何基于数值 $x = 3.13, 3.14, 3.15, 3.16$ 的插值法，都肯定在 $x = 3.1416$ 处得到一个错误的解，尽管通过这5个点所画的曲线确实相当平滑（用计算器试试看）。

因为这种病态函数可能隐藏在任何地方，所以要求内插法和外推法能提供自身的错误评估。这样的错误评估自然不会是非常简单的。我们可能有个函数，它丢弃掉在已知两点之间那些相差太远或是不符合要求的点，其理由只有函数构造者才清楚。内插法总是假定进行内插的函数有一定程度的光滑性，在此假定下，偏离光滑性也应能被检测出来。

概念上讲，插值过程有两个阶段：(1)找一个插值函数满足已知数据点；(2)对目标点 x 进行插值函数评估。

然而在实际应用中，这个二阶段法并不总是最好的办法。此方法与每次需要时都由 N 个给定值直接构造估计函数 $f(x)$ 的方法相比，在计算上一般缺乏高效性，而且存在舍入误差问题。大多数实用方法是，由一个邻近点 $f(x_i)$ 开始，考虑其他相关的 $f(x_i)$ 信息，然后进行一连串的越来越小的修正（希望如此）。该过程的运算次数一般为 $O(N^2)$。如果进行得顺利，最后一次的修正规模最小，而且该修正值可作为误差的一个非正式的（然而并不是严格的）界限。

对多项式插值的情况，有时候会对插值多项式的系数感兴趣，尽管用它们来估计插值函数也并不太令人满意。我们将在3.5节中讨论这个问题。

局部插值，使用有限个"最邻近"点，给出插值函数 $f(x)$，一般说来它没有连续的一阶导数或高阶导数。这是因为 x 取的是表中给出的值 x_i，插值法所连接的这些表中的点都是"局部的"（如果这样的连接允许发生在任何地方，则在那一点插值函数本身将不连续，这非常糟糕！）。

在考虑导数连续性要求时,就必须使用一种称为**样条**(spline)函数的"呆板"的插值方法。样条是两个已知列表点之间的多项式函数,但它的系数却有些非局部确定的感觉。这种非局部性是用来保证该插值函数的若干阶导数的整体平滑性。三次样条插值(参见3.3节)是最为通用的方法。其产生的插值函数要求二阶导数连续。样条法比多项式法趋于稳定,它在列表点之间摇动的概率要小些。

插值法中使用的节点数目(减1)称为该**插值的阶**。尤其对多项式插值,增加阶数并不一定能提高精度。如果添加的点离感兴趣的点 x 相差太远,则造成带有额外限制点的高阶多项式,会在列表值之间产生振荡现象。这种振荡可能与"真实"函数的形状毫无关系(参见图3.0.1)。当然,增加所求节点附近节点的数目通常会有所帮助,但一点点改进意味着要增加大量表中的数值,故通常是不可取的

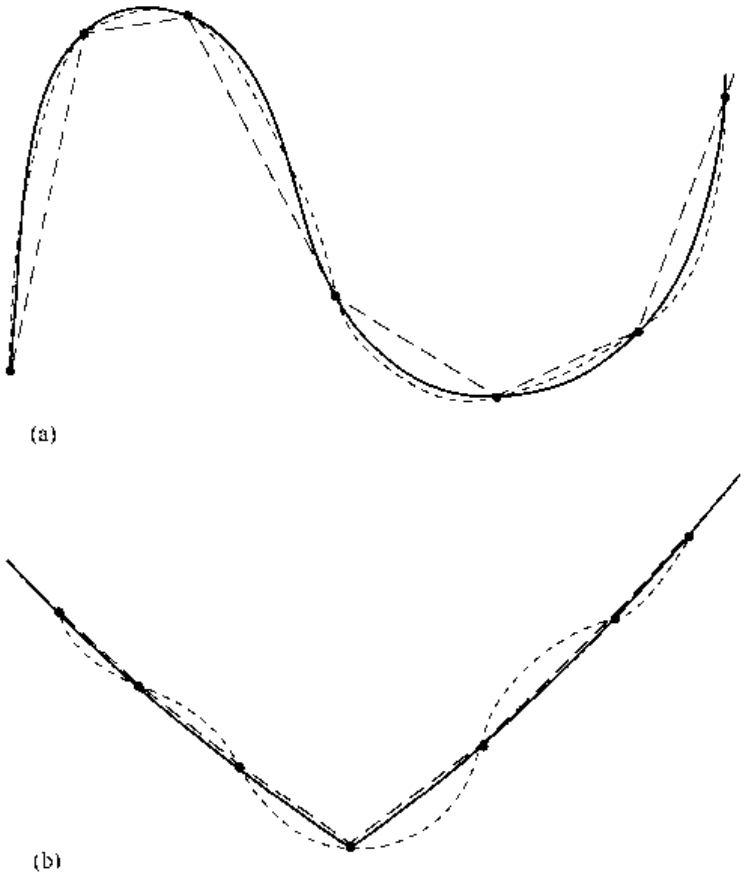


图3.0.1 (a)平滑函数(实线)用高阶多项式插值表示(虚线),比用低级多项式插值表示(一段段的折线)要更精确些。(b)拐点很陡或高阶导数变化迅速的函数,用高阶多项式近似(虚线),由于太"呆板",比用低阶多项式近似(折线)精度要差一些。即使像指数函数或有理函数这样的平滑函数,用高阶多项式近似也是很差的。

如果有可靠的证据证明插值函数的形式接近于真实函数 f,那么使用高阶插值是个好主意。我们推荐使用3至4个节点的插值,5至6个节点也是可以接受的。除非对估计的误差控制得相当严格,一般极少使用再多的节点了。

如果数据表中包含的点多于插值所需的数目,则在每次插值时,就必须选取表中合适位置的点。尽管这项工作严格来说并不是插值法专题的一部分,但却非常重要(经常做得很不好)。我们将在3.4节中讨论。

给出的内插法程序也是外推法程序。它在第16章中的一个重要应用是常微分方程的积分解法。在应用中要引起特别注意的是误差控制。否则,外推法的不可靠性就会突显:内插函数用做外推函数,当参数 x 在已知区间之外,且其距离大于表中点的典型间距时,所得结果与真实值会相差甚远。

插值可以是多维的,如函数 $f(x,y,z)$。一般情况下,多维插值伴随着一系列一维插值。这个问题将在3.6节中讨论。

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), Chapter 2.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 3.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 4.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 5.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), Chapter 3.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), Chapter 6.

3.1　多项式内插法和外推法

通过任意两点只有惟一一条直线，通过任意三点只有惟一一个平面，等等。通过 N 个点 $y_0=f(x_0)$，$y_1=f(x_1)$，…，$y_{N-1}=f(x_{N-1})$ 的 $N-1$ 维内插多项式，可用拉格朗日(Lagrange)经典公式明确地表示出来，

$$
\begin{aligned}
P(x)=&\frac{(x-x_1)(x-x_2)\dots(x-x_{N-1})}{(x_0-x_1)(x_0-x_2)\dots(x_0-x_{N-1})}y_0\\
&+\frac{(x-x_0)(x-x_2)\dots(x-x_{N-1})}{(x_1-x_0)(x_1-x_2)\dots(x_1-x_{N-1})}y_1+\cdots\\
&+\frac{(x-x_0)(x-x_1)\dots(x-x_{N-2})}{(x_{N-1}-x_0)(x_{N-1}-x_1)\dots(x_{N-1}-x_{N-2})}y_{N-1}
\end{aligned}
\tag{3.1.1}
$$

其中共有 N 个项，每个项都是一个 $N-1$ 维多项式。对每一项除了一个 x_i 使其值为 y_i 外，对其他所有 x_i 值均为零。

直接使用拉格朗日公式并无什么严重的错误，但也不是非常好的办法。这种求解算法没有给出误差估计，并且不适合编程。一个较好的算法(构造同样的惟一的插值多项式)是 Neville 算法，它与艾特肯(Aitken)算法有关联，有时甚至容易混淆，但后者现在已经过时了。

令 P_0 是经过点(x_0,y_0)的 x 零阶多项式的值(即常数)，故 $P_0=y_0$。用类似的方法定义 $P_1,P_2,\dots,P_{N-1}$。再令 P_{01} 为过点(x_0,y_0)和(x_1,y_1)的 x 一阶多项式的值。用类似方法定义 $P_{12},P_{23},\dots,P_{(N-2)(N-1)}$。对高阶多项式进行类似定义，直至 $P_{012\cdots(N-1)}$，它是对过所有 N 个点的插值多项式的值，即所求的解。各个不同的 P 形成一张"表"，左边为"祖先"，一直推到最右边的一个"后代"。比如，当 $N=4$ 时，

$$
\begin{array}{lllll}
x_0: & y_0=P_0 & & & \\
 & & P_{01} & & \\
x_1: & y_1=P_1 & & P_{012} & \\
 & & P_{12} & & P_{0123} \\
x_2: & y_2=P_2 & & P_{123} & \\
 & & P_{23} & & \\
x_3: & y_3=P_3 & & &
\end{array}
\tag{3.1.2}
$$

Neville 算法用递推的方法填写表中的数，每次一列，从左到右。它基于"子女"P 和其两个"双亲"之间的关系，得

$$
P_{i(i+1)\dots(i+m)}=\frac{(x-x_{i+m})P_{i(i+1)\dots(i+m-1)}+(x_i-x)P_{(i+1)(i+2)\dots(i+m)}}{x_i-x_{i+m}}
\tag{3.1.3}
$$

因为两个双亲值已由点 $x_{i+1},\dots,x_{i+m-1}$ 得到，故该递推式可求。

对递推式(3.1.3)的一个改进是，记下双亲值与子女值之间的微小"差值"，即定义($m=1,2,\dots,N-1$)

$$
\begin{aligned}
C_{m,i} &\equiv P_{i\ldots(i+m)} - P_{i\ldots(i+m-1)} \\
D_{m,i} &\equiv P_{i\ldots(i+m)} - P_{(i+1)\ldots(i+m)}
\end{aligned}
\tag{3.1.4}
$$

由(3.1.3)式很容易推出关系式

$$
\begin{aligned}
D_{m+1,i} &= \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \\
C_{m+1,i} &= \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}}
\end{aligned}
\tag{3.1.5}
$$

对每一层 m,修正后的 C 和 D 都使插值高一级。最后的解 $P_{0\ldots(N-1)}$ 等于任意 y_i 加上一些 C 和(或)D 之和,这些 C 和 D 形成一条穿过家族树到达右端了女的路径。

下面是一个多项式内插或外推程序。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::polint(Vec_I_DP &xa, Vec_I_DP &ya, const DP x, DP &y, DP &dy)
给定数组 xa[0..n-1]和 ya[0..n-1]及数值 x,本程序返回值 y 及误差估计 dy。如果 P(x)为使 P(xa_i) = ya_i, i = 0,...,
n-1 的 N-1 阶多项式,则返回值 y = P(x)。
{
    int i,m,ns=0;
    DP den,dif,dift,ho,hp,w;

    int n=xa.size();
    Vec_DP c(n),d(n);
    dif=fabs(x-xa[0]);
    for (i=0;i<n;i++) {             这里用ns作为表中最近入口的指针。
        if ((dift=fabs(x-xa[i])) < dif) {
            ns=i;
            dif=dift;
        }
        c[i]=ya[i];                 初始化表c和表d。
        d[i]=ya[i];
    }
    y=ya[ns--];                     y的初始逼近。
    for (m=1;m<n;m++) {             对表中的每一列。
        for (i=0;i<n-m;i++) {       对当前c和d进行循环, 并更新它们的值。
            ho=xa[i]-x;
            hp=xa[i+m]-x;
            w=c[i+1]-d[i];
            if ((den=ho-hp) == 0.0) nrerror("Error in routine polint");
            该错误仅当两个输入的xa值(在误差范围内)相同时才发生.
            den=w/den;
            d[i]=hp*den;            c和d在此处被更新.
            c[i]=ho*den;
        }
        y += (dy=(2*(ns+1) < (n-m) ? c[ns+1] : d[ns--]));
        表中的一列处理完后,判断要把修正值 c 还是 d 加到 y 的累积值上,就是说,选择表中的哪一条路径——走上
        边的分叉还是下边的分叉。我们的做法是走表中最"直"的路径到达顶点,并相应地更新 ns 值以记录所走的路
        径。本程序求得目标点 x 上有偏估计值。最后用 dy 表示误差。
    }
}
```

常常需要用涉及到一个数组的子序列(如 x[5]...x[8])的虚设变量 xa 和 ya 来调用程序 polint。然而,既然程序 polint 用到了类型 NRVec,并且 NRVec 没有具体化这种子数组的功能,我们就不得不将期望的点复制到适当长度的临时数组中去。关于这个问题的进一步讨论,参见 3.4 节的结尾部分。

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §2.1.

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.

3.2　有理函数内插法和外推法

有些函数用多项式近似效果不好，但用有理函数，即多项式的商来估计效果却很好。我们用 $R_{i(i+1)\ldots(i+m)}$ 表示过 $m+1$ 个点 $(x_i, y_i)\ldots(x_{i+m}, y_{i+m})$ 的有理函数。为更清楚起见，假定

$$R_{i(i+1)\ldots(i+m)} = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \cdots + p_\mu x^\mu}{q_0 + q_1 x + \cdots + q_\nu x^\nu} \tag{3.2.1}$$

因为有 $\mu+\nu+1$ 个未知的 p 和 q（q_0 任意），则有

$$m+1 = \mu+\nu+1 \tag{3.2.2}$$

在把插值函数定义为有理函数时，必须给出分子和分母多项式的阶数。

有理函数比多项式函数优越，粗略而言是因为它们能够模拟具有极点的函数。极点是指(3.2.1)式中分母的零点。如果要插值的函数本身有极点的话，则在实数 x 处就有可能发生这些极点。更为常见的情况是，函数 $f(x)$ 对所有有限实数 x 都是有穷的，但在 x 复平面内存在极点的解析连续性。即使限定为实数 x，这些极点也会使多项式逼近彻底失败，正如它们会使 x 的无穷级数达不到收敛一样。如果在复平面内，沿列表中的 m 个点画一个圆，除非最近的极点也远离此圆，否则多项式插值结果就不会很好。相比而言，只要在有理函数的分母中，x 有足够高的幂次以避免（消除）近处的极点，有理函数逼近的效果就还不错。

对插值问题，构造有理函数就是为了使其能够通过一组选定的列表函数值。然而应该提及的是，有理函数逼近可以用于解析的工作中。有人曾构造了一个有理函数逼近，其准则为：(3.2.1)式的有理函数本身的幂级数展开，应与欲求函数 $f(x)$ 的幂级数展开的前 $m+1$ 项一致。这被称为帕德(Padé)逼近，将在 5.12 节中讨论。

Bulirsch 和 Stoer 发现了一种 Neville 型的算法，可以对表中的数据进行有理函数外推。按列构造类似(3.1.2)式的表，进而导出最后的结果和误差估计。Bulirsch-Stoer 算法产生一个所谓的**对角有理函数**，分子和分母阶数相等（若 m 为偶数），或者分母的阶数比分子的阶数高一阶（若 m 为奇数，参见上面的(3.2.2)式）。该算法的推导参见本节参考文献[1]。类似于多项式逼近的(3.1.3)式，该算法也可用递推关系精确地总结为

$$\begin{aligned} R_{i(i+1)\ldots(i+m)} &= R_{(i+1)\ldots(i+m)} \\ &\quad + \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{\left(\frac{x-x_i}{x-x_{i+m}}\right)\left(1 - \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{R_{(i+1)\ldots(i+m)} - R_{(i+1)\ldots(i+m-1)}}\right) - 1} \end{aligned} \tag{3.2.3}$$

该递推式表示，可由过 m 个点的有理函数和过 $m-1$ 个点的有理函数（(3.2.3)式中的 $R_{(i+1)\ldots(i+m-1)}$）导出过 $m+1$ 个点的有理函数。初始值为

$$R_i = y_i \tag{3.2.4}$$

及

$$R \equiv [R_{i(i+1)\ldots(i+m)} \quad \text{with} \quad m=-1] = 0 \tag{3.2.5}$$

现在，跟上面的(3.1.4)式和(3.1.5)式完全一样，将递推式(3.2.3)转化为仅含细小差别的式子

$$\begin{aligned} C_{m,i} &\equiv R_{i\ldots(i+m)} - R_{i\ldots(i+m-1)} \\ D_{m,i} &\equiv R_{i\ldots(i+m)} - R_{(i+1)\ldots(i+m)} \end{aligned} \tag{3.2.6}$$

注意它们满足关系式

$$C_{m+1,i} - D_{m+1,i} = C_{m,i+1} - D_{m,i} \tag{3.2.7}$$

该式在证明下面的递推式时是非常有用的：

$$D_{m+1,i}=\frac{C_{m,i+1}(C_{m,i+1}-D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right)D_{m,i}-C_{m,i+1}}$$

$$C_{m+1,i}=\frac{\left(\frac{x-x_i}{x-x_{i+m+1}}\right)D_{m,i}(C_{m,i+1}-D_{m,i})}{\left(\frac{x-x_i}{x-x_{i+m+1}}\right)D_{m,i}-C_{m,i+1}} \tag{3.2.8}$$

该递推关系由下面的函数来实现，其用法与3.1节中的polint程序完全相同。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::ratint(Vec_I_DP &xa, Vec_I_DP &ya, const DP x, DP &y, DP &dy)
给定数组xa[0..n-1]和ya[0..n-1]及数值x，本程序返回y的值及精度估计dy。返回值是通过n个点(xa_i, ya_i)，i=0,
...,n-1的对角有理函数在x处的值。
{
    const DP TINY=1.0e-25;              一个小的数。
    int m,i,ns=0;
    DP w,t,hh,h,dd;
    int n=xa.size();
    Vec_DP c(n),d(n);
    hh=fabs(x-xa[0]);
    for (i=0;i<n;i++) {
        h=fabs(x-xa[i]);
        if (h == 0.0) {
            y=ya[i];
            dy=0.0;
            return;
        } else if (h < hh) {
            ns=i;
            hh=h;
        }
        c[i]=ya[i];
        d[i]=ya[i]+TINY;            须用TINY来防止少见的0除0的情况。
    }
    y=ya[ns--];
    for (m=1;m<n;m++) {
        for (i=0;i<n-m;i++) {
            w=c[i+1]-d[i];
            h=xa[i+m]-x;            因为在初始循环中已测试过，这里h不会为零。
            t=(xa[i]-x)*d[i]/h;
            dd=t-c[i+1];
            if (dd == 0.0) nrerror("Error in routine ratint");
            这种错误情况表明该内插函数在欲求的x处存在极点。
            dd=w/dd;
            d[i]=c[i+1]*dd;
            c[i]=t*dd;
        }
        y += (dy=(2*(ns+1) < (n-m) ? c[ns+1] : d[ns--]));
    }
}
```

参考文献和深入阅读

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §2.2. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.2.

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 3.

3.3　三次样条插值

给定一个列表显示的函数 $y_i = y(x_i)$，$i = 0 \ldots N-1$，特别注意在 x_j 和 x_{j+1} 之间的一个特殊的区间。该区间的线性插值公式为

$$y = Ay_j + By_{j+1} \tag{3.3.1}$$

其中，

$$A \equiv \frac{x_{j+1} - x}{x_{j+1} - x_j} \qquad B \equiv 1 - A = \frac{x - x_j}{x_{j+1} - x_j} \tag{3.3.2}$$

(3.3.1)式和(3.3.2)式是拉格朗日插值公式(3.1.1)的特殊情况。

因为它是(分段)线性的，(3.3.1)式在的每一区间内的二阶导数为零，在横坐标为 x_j 处的二阶导数不定义或无限。三次样条插值的目的就是要得到一个内插公式，不论在区间内亦或其边界上，其一阶导数平滑，二阶导数连续。

做一个与事实相反的假设，除 y_i 的列表值之外，我们还有函数二阶导数 y'' 的列表值，即一系列的 y_i'' 值，则在每个区间内，可以在(3.3.1)式的右边加上一个三次多项式，其二阶导数从左边的 y_j'' 值线性变化到右边的 y_{j+1}'' 值。这么做便得到了所需的连续二阶导数。如果还将该三次多项式构造成在 x_j 和 x_{j+1} 处为零，则不会破坏在终点 x_j 和 x_{j+1} 处与列表函数值 y_j 和 y_{j+1} 的一致性。

进行一些辅助计算便可知，仅有一种办法才能进行这种构造，即用

$$y = Ay_j + By_{j+1} + Cy_j'' + Dy_{j+1}'' \tag{3.3.3}$$

来代替(3.3.1)式，其中 A 和 B 由(3.3.2)式定义，且

$$C \equiv \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2 \qquad D \equiv \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2 \tag{3.3.4}$$

注意，(3.3.3)式和(3.3.4)式对自变量 x 的依赖，是完全通过 A 和 B 对 x 的线性依赖，以及 C 和 D(通过 A 和 B)对 x 的三次依赖而实现的。

可以很容易地验证，y'' 事实上是该插值多项式的二阶导数。使用 A, B, C, D 的定义对 x 求(3.3.3)式的导数，计算 dA/dx，dB/dx，dC/dx 及 dD/dx。结果为一阶导数

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y_j'' + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y_{j+1}'' \tag{3.3.5}$$

二阶导数

$$\frac{d^2y}{dx^2} = Ay_j'' + By_{j+1}'' \tag{3.3.6}$$

因为 $x = x_j$ 时 $A = 1$，$x = x_{j+1}$ 时 $A = 0$，而 B 正相反，则(3.3.6)式表明 y'' 恰为列表函数的二阶导数。而且该二阶导数在两个区间 (x_{j-1}, x_j) 和 (x_j, x_{j+1}) 上是连续的。

现在惟一的问题是，假设 y_i'' 是已知的，而实际上并不知道。然而，仍不要求从(3.3.5)式算出的一阶导数在两个区间的边界处是连续的。三次样条的关键思想就在于要求这种连续性，并用它求得等式的二阶导数 y_i''。

设(3.3.5)式在区间 (x_{j-1}, x_j) 上对 $x = x_j$ 求得的值，等于同一等式在区间 (x_j, x_{j+1}) 上对 $x = x_j$ 求得的值，便可得到所求方程。重新整理得到(对 $j = 1, \ldots, N-2$)

$$\frac{x_j - x_{j-1}}{6}y_{j-1}'' + \frac{x_{j+1} - x_{j-1}}{3}y_j'' + \frac{x_{j+1} - x_j}{6}y_{j+1}'' = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}} \tag{3.3.7}$$

这意味着有 $N-2$ 个线性方程，但却有 N 个未知数 y_i''，$i = 0, \ldots, N-1$。因此，存在一个具有两个参数的可能解集。

为求得惟一解，需要给出两个进一步的条件，一般取 x_0 和 x_{N-1} 处的边界条件。最常见的做法有：

- 设 y_0'' 和 y_{N-1}'' 之一或两个都为零，得到所谓的**自然三次样条**函数，其一个或两个边界的二次导数为零。

- 设 y''_0 和 y''_{N-1} 为(3.3.5)式计算得到的值,使得该插值函数的一阶导数在一个或两个边界处有特定的值。

三次样条插值特别实用的原因之一在于,有两个附加边界条件的方程组(3.3.7),它不仅是线性的,而且是三对角的。每个 y''_j 仅与其最邻近的 $j\pm1$ 的值有关。因此,方程可以用三对角算法(参见2.4节)在 $O(N)$ 次运算内求解。该算法非常简明,很容易正确地构造出样条计算的程序。但是这使得程序不像(3.3.7)式的实现那样完全透明,所以建议读者将下面程序与 tridag 程序(参见2.4节)比较一下,仔细地研究。

```
#include "nr.h"

void NR::spline(Vec_I_DP &x, Vec_I_DP &y, const DP yp1, const DP ypn,
    Vec_O_DP &y2)
给定数组 x[0..n-1]和 y[0..n-1]构成一个列表函数,即 y_i = f(x_i),其中 x_0 < x_1 < ... < x_{N-1},还有插值函数在第零个点和
第 n-1 个点处的一阶导数值 yp1 和 ypn。相应地,本程序返回数组 y2[0..n-1],其值为插值函数在表中 x_i 点的二阶导数。
如果 yp1 和(或)ypn 大于或等于 1×10^30,则本程序将相应的边界条件设为自然样条,即边界的二阶导数为零。
{
    int i,k;
    DP p,qn,sig,un;

    int n=y2.size();
    Vec_DP u(n-1);
    if (yp1 > 0.99e30)              下边界条件设成"自然的",
        y2[0]=u[0]=0.0;
    else {                          否则有特定的一阶导数。
        y2[0] = -0.5;
        u[0]=(3.0/(x[1]-x[0]))*((y[1]-y[0])/(x[1]-x[0])-yp1);
    }
    for (i=1;i<n-1;i++) {           三对角算法的分解循环,y2和u为被分解因子的临时变量。
        sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);
        p=sig*y2[i-1]+2.0;
        y2[i]=(sig-1.0)/p;
        u[i]=(y[i+1]-y[i])/(x[i+1]-x[i]) - (y[i]-y[i-1])/(x[i]-x[i-1]);
        u[i]=(6.0*u[i]/(x[i+1]-x[i-1])-sig*u[i-1])/p;

    }
    if (ypn > 0.99e30)              上边界条件设成"自然的",
        qn=un=0.0;
    else {                          否则有特定的一阶导数。
        qn=0.5;
        un=(3.0/(x[n-1]-x[n-2]))*(ypn-(y[n-1]-y[n-2])/(x[n-1]-x[n-2]));
    }
    y2[n-1]=(un-qn*u[n-2])/(qn*y2[n-2]+1.0);
    for (k=n-2;k>=0;k--)            三对角算法的回代循环。
        y2[k]=y2[k]*y2[k+1]+u[k];
}
```

重要的是要记住,在处理数组 x_i 和 y_i 构成的列表函数时,程序 spline 仅能调用一次。一旦调用过后,对任意的 x,插值函数的值通过调用(要多少次就调多少次)一个独立的程序 splint(spline interpolation 的字头缩写)来得到。

```
#include "nr.h"

void NR::splint(Vec_I_DP &xa, Vec_I_DP &ya, Vec_I_DP &y2a, const DP x, DP &y)
给定数组 xa[0..n-1]和 ya[0..n-1]作为列表函数(按 xa_i 的顺序),数组 y2a[0..n-1]由上面 spline 程序的输出给定,并给
定 x 的值,本程序返回 x 的三次样条插值结果 y。
{
    int k;
    DP h,b,a;

    int n=xa.size();
    int klo=0;                      用二分法查找表的正确位置。如果用随机的x值连续调
    int khi=n-1;                    用本函数,这么做是最佳的。如果连续调用时,x是有
    while (khi-klo > 1) {           序的且间距不大,最好把前一次调用时klo和khi的值保
        k=(khi+klo) >> 1;           存起来,在下次调用时先检测一下它们的值是否还合适。
        if (xa[k] > x) khi=k;
        else klo=k;
```

```
    }                                   klo和 khi 为x输入值的上下界。
    h=xa[khi]-xa[klo];
    if (h == 0.0) nrerror("Bad xa input to routine splint");    xa的值不能相同。
    a=(xa[khi]-x)/h;
    b=(x-xa[klo])/h;                    三次样条多项式求值。
    y=a*ya[klo]+b*ya[khi]+((a*a*a-a)*y2a[klo]
        +(b*b*b-b)*y2a[khi])*(h*h)/6.0;
}
```

参考文献和深入阅读

De Boor, C. 1978, *A Practical Guide to Splines* (New York: Springer-Verlag).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §§4.4–4.5.

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §2.4.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §3.8.

3.4　搜索有序表的方法

假设决定用一些特殊的插值方法，如用四阶多项式插值，根据表中 x_i 和 f_i 的值计算出函数 $f(x)$。那么，对给出的特定值 x，要求计算其函数值，就需要一个快速的方法找到 x_i 在表中的位置。将这个问题划归于数值分析不一定合适，但实际中却很容易碰到，所以不能忽略。

这个问题的标准描述为：已知横坐标数组 xx[j]，j = 0, 1, . . . , n − 1，按递增或递减的顺序排列，对给出的数 x，求整数 j，使得 x 在 xx[j]与 xx[j + 1]之间。为了查找方便，首先定义两个附加的数组元素 xx[− 1]和xx[n]，分别代表正负无穷大(这样仍然保持表中元素的单调性)。这样 j 将总是在 − 1 到 n − 1 之间，即可用 − 1 作为表的一端界限，n − 1 作为另一端界限。

在大多数情况下，当上面所述的要求满足时，**二分法**是最好的解决办法，它能在大约 $\log_2 n$ 次试验内找到表中正确的位置。在上一节的样条法求值程序 splint 中，已经使用了二分法，所以可以参考一下。另外，二分法程序一般可如下构造：

```
#include "nr.h"

void NR::locate(Vec_I_DP &xx, const DP x, int &j)
给定数组 xx[0..n-1]及值 x，返回值 j，使得 x 在 xx[j]和 xx[j+1]之间。xx 必须是单调的，递增或递减均可。若返回 j=
-1 或 j=n-1，则表明 x 越界了。
{
    int ju,jm,jl;
    bool ascnd;

    int n=xx.size();
    jl=-1;                          初始化上下边界。
    ju=n;
    ascnd=(xx[n-1] >= xx[0]);       递增的顺序为真，否则为假。
    while (ju-jl > 1) {             如果还没有完成，则计算中点，
        jm=(ju+jl) >> 1;
        if (x >= xx[jm] == ascnd)
            jl=jm;                  并替换下界，
        else
            ju=jm;                  否则应替换上界。
    }                               循环至条件满足。
    if (x == xx[0]) j=0;            设置输出量，
    else if (x == xx[n-1]) j=n-2;
    else j=jl;
}                                   并返回。
```

3.4.1 用相关数值进行搜索

有时会遇到多次查表的情况,而且是用几乎相同的横坐标进行连续查询。例如,可能正在构造一个用在微分方程左端的函数;如将在第16章中看到的那样,大部分微分方程的积分要在向前或向后跳移的一些点上,调用右端项的值。但是,在此积分方向上,它的移动趋势很慢。

在这种情况下,每次调用都从头开始进行完全二分处理是很浪费的。下面的程序,不是从头而是从表中某个预测的位置开始查表。它的第一次搜寻,不论递增还是递减,变化量都为1,下一次为2,再一次为4,等等。直到要查的值落在该区间内为止。然后,在所得的区间内进行二分处理。在最坏的情况下,本程序大概要比上面的locate程序慢两倍(如果搜寻的范围遍及整张表)。在最好的情况下,即所求点通常离预测值很近的情况,要比locate程序快$\log_2 n$倍。图3.4.1是对这两个程序的比较。

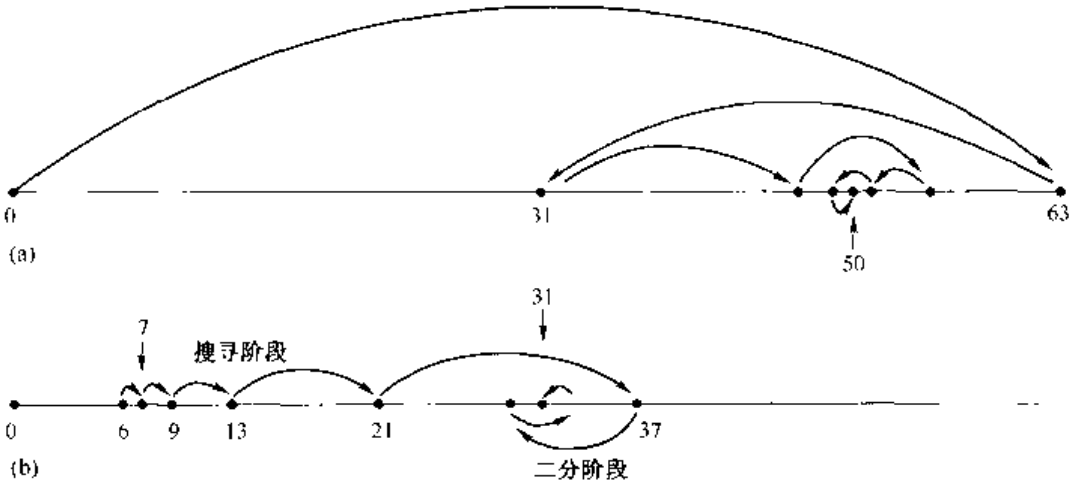


图3.4.1 (a)程序locate用二分法寻找有序表。这里显示的是在长度为64的表中,按步顺序收敛到元素50的过程。(b)程序hunt是从表中的已知位置开始,按递增步长搜寻,然后用二分法收敛。这里显示的是一个不太令人满意的例子,从元素6开始收敛到元素31。比较满意的例子应是查找6附近的元素,比如8,仅需要三次查找

```
#include "nr.h"

void NR::hunt(Vec_I_DP &xx, const DP x, int &jlo)
给定数组 xx[0..n-1]及数值 x,返回值 jlo,使得 x 在 xx[jlo]与 xx[jlo+1]之间。xx[0..n-1]必须是单调的,递增或递减。
若返回 jlo=-1 或 jlo=n-1 则表明 x 越界。输入的 jlo 值是作为其输出值的预测值。
{
    int jm,jhi,inc;
    bool ascnd;

    int n=xx.size();
    ascnd=(xx[n-1] >= xx[0]);          递增的顺序为真,否则为假。
    if (jlo < 0 || jlo > n-1) {        预测值不准,无参考价值,直接转到二分处理。
        jlo=-1;
        jhi=n;
    } else {
        inc=1;                         设置搜索步长,
        if (x >= xx[jlo] == ascnd) {   向上搜索:
            if (jlo == n-1) return;
            jhi=jlo+1;
            while (x >= xx[jhi] == ascnd) {        搜索未结束,
                jlo=jhi;
                inc += inc;            步长加倍,
                jhi=jlo+inc;
                if (jhi > n-1) {       搜寻结束,已至表尾。
```

```
                jhi=n;
                break;
            }                           再试。
        }                               搜量结束，值区间找到。
    } else {                            向下搜索;
        if (jlo == 0) {
            jlo=-1;
            return;
        }
        jhi=jlo--;
        while (x < xx[jlo] == ascnd) {          搜寻未结束,
            jhi=jlo;
            inc <<= 1;                  加倍步长,
            if (inc >= jhi) {           搜寻结束，已至表尾。
                jlo=-1;
                break;
            }
            else jlo=jhi-inc;
        }                               再试。
    }                                   搜寻结束，值区间找到。
}                                       搜寻完成，开始进入最后的二分阶段。
while (jhi-jlo != 1) {
    jm=(jhi+jlo) >> 1;
    if (x >= xx[jm] == ascnd)
        jlo=jm;
    else
        jhi=jm;
}
if (x == xx[n-1]) jlo=n-2;
if (x == xx[0]) jlo=0;
}
```

3.4.2　写在 Hunt 之后

问题：程序 locate 和 hunt 返回索引指针 j，使求的值处在表项 xx[j]与 xx[j+1]之间，其中 xx[0..n-1]为表的总长度。但是如果想使用如 polint(参见 3.1 节)或 ratint(参见 3.2 节)这样的程序得到 m 个点插值，就必须提供长度为 m 的较短的数组 xx 和 yy。那么，这些子数组的索引指针具体该如何定义呢？

解答：计算

$$k = \mathrm{MIN}(\mathrm{MAX}(j-(m-1)/2,0),n-m)$$

(函数 MIN 和 MAX 是求两个整型参数的最小值和最大值，参见 1.2 节及附录 B_2)该表达式产生的是 m 个点集合中最左边元素的索引指针，这 m 个点的中心在 j 和 j+1 之间(在可能范围内)，但左边界为 0，右边界为 n-1。于是，需要将 xx[k..k+m-1]和 yy[k..k+m-1]的值复制到临时的 NRVecs 中以过渡到程序 polint。

参考文献和深入阅读

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §6.2.1.

3.5　插值多项式的系数

有些时候想要知道的，并非是经过少数几个点的插值多项式的函数值，而是该多项式的系数。这些系数的实际用处可以是，同时计算函数的插值和它的几个导数(参见 5.3 节)，或者求列表函数段与某个其他函数的卷积，这个其他函数的矩(即它与 x 的幂函数卷积)是解析已知的。

然而，一定要肯定这些系数确为所需的。一般来说，插值多项式系数的确定，不像对某特定点求函数值那么精确。因此，仅仅为计算插值而确定系数并不是个好办法。那样计算得到的数值并不能精确地通过列表中的点，而用 3.1 节～3.3 节中的程序所计算的值，能精确地通过这些列表点。

读者也不应该将插值多项式(及其系数)错当成它的类似问题：经过一系列数据点进行多项式的**最佳拟合**。拟合是个求平滑函数的过程，因为拟合的系数个数一般要远比数据点的数目少。所以，即使在列表值出

现统计误差时,拟合的系数也可以精确而稳定地确定(参见14.8节)。而系数个数等于列表点个数的插值法,是把所有列表中的数据都当成准确无误的。如果其中含有统计误差,则可能导致列表点之间产生插值多项式的摆动

跟以前一样,令列表中的点为 $y_i = y(x_i)$。如果插值多项式写成

$$y = c_0 + c_1 x + c_2 x^2 + \cdots + c_{N-1} x^{N-1} \tag{3.5.1}$$

则 c_i 应满足线性方程

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \cdots & x_{N-1}^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} \tag{3.5.2}$$

这是一个 **Vandermonde 矩阵**,参见2.8节所述,原则上可以用解线性方程的标准技术(参见2.3节)来求解(3.5.2)式。然而,用2.8节推导出来的特殊方法对大的维数 N 更为有效,因此这个方法要好一些。

记住,Vandermonde 问题可能是相当病态的。在这种情况下,找不到能给出非常精确解的数值方法。但这些情况并不意味着采用3.1节中的方法计算插值会有困难,只是在计算系数时才会有困难。

和2.8节中的程序一样,下面的程序也是由 G.B.Rybicki 提供的。

```
#include "nr.h"

void NR::polcoe(Vec_I_DP &x, Vec_I_DP &y, Vec_O_DP &cof)
给定数组 x[0..n-1] 和 y[0..n-1] 表示列表函数 y_i = f(x_i),本程序返回系数数组 cof[0..n-1],使得 y_i = Σ_{j=0}^{n-1} cof_j x_i^j。
{
    int k,j,i;
    DP phi,ff,b;

    int n=x.size();
    Vec_DP s(n);
    for (i=0;i<n;i++) s[i]=cof[i]=0.0;
    s[n-1]= -x[0];
    for (i=1;i<n;i++) {                由递推求得主多项式 P(x) 的原系数 s_i。
        for (j=n-1-i;j<n-1;j++)
            s[j] -= x[i]*s[j+1];
        s[n-1] -= x[i];
    }
    for (j=0;j<n;j++) {
        phi=n;
        for (k=n-1;k>0;k--)        量 phi = Π_{j≠k}(x_j - x_k) 作为 P(x_j) 的导数已找到。
            phi=k*s[k]+x[j]*phi;
        ff=y[j]/phi;
        b=1.0;                     拉格朗日公式中每一项的多项式系数由综合除法 P(x)
        for (k=n-1;k>=0;k--) {     除以(x - x_j)求得。解 c_k 为累计值。
            cof[k] += b*ff;
            b=s[k]+x[j]*b;
        }
    }
}
```

3.5.1 其他方法

还有一种技术是,采用已经介绍过的函数值的插值程序(参见3.1节中的 polint 程序)。如果使用内插法(或外推法)求插值多项式在 $x=0$ 处的值,则此值显然为 c_0。现在可以将每个 y_i 减去 c_0,再除以各自相应的 x_i。然后,去掉一个点(具有最小 x_i 的点是很好的候选点),再重复此过程求 c_1,如此继续。

此处理过程的稳定性并不能马上看出来,但我们发现一般情况下,它比直接处理的程序更加稳定。这种方法的复杂度为 N^3,而处理一次为 N^2。然而,会发现对 N 很大时这两种做法效果都不好,这是因为 Vandermonde 问题本身是病态的。用单精度表示时,N 达到8或10效果还比较令人满意;用双精度表示时,N 值差不多可加倍。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::polcof(Vec_I_DP &xa, Vec_I_DP &ya, Vec_O_DP &cof)
给定数组 xa[0..n-1]和 ya[0..n-1]表示列表函数 ya_i = f(xa_i)，本程序返回系数数组 cof[0..n-1]，使得 ya_i =
∑_{j=0}^{n-1} cof_j xa_i^j。
{
    int k,j,i;
    DP xmin,dy;

    int n=xa.size();
    Vec_DP x(n),y(n);
    for (j=0;j<n;j++) {
        x[j]=xa[j];
        y[j]=ya[j];
    }
    for (j=0;j<n;j++) {
        Vec_DP x_t(n-j),y_t(n-j);
        for (k=0;k<n-j;k++) {
            x_t[k]=x[k];
            y_t[k]=y[k];
        }
        polint(x_t,y_t,0.0,cof[j],dy);          外推到 x = 0。
        xmin=1.0e38;
        k = -1;
        for (i=0;i<n-j;i++) {                   寻找最小绝对值 x_i，
            if (fabs(x[i]) < xmin) {
                xmin=fabs(x[i]);
                k=i;
            }
            if (x[i] != 0.0)                    (同时化简所有的项)
                y[i]=(y[i]-cof[j])/x[i];
        }
        for (i=k+1;i<n-j;i++) {                 并进行消去。
            y[i-1]=y[i];
            x[i-1]=x[i];
        }
    }
}
```

如果点 $x=0$ 不在(或者至少不接近)列表 x_i 值的范围，则插值多项式的系数一般来说会变得非常大。然而，这些系数真正的“信息内容”和“平移引起”的大值只有微小差别。这是一种病态的情况，会导致有效数字的丢失，并使求出的系数结果很差。这时，应当考虑重新定义原来的问题，把 $x=0$ 放到合适的位置。

另外一种比较差的情况是，如果试图用太高阶的插值求平滑函数，则插值多项式也会试图用高阶系数，即这些系数很大，但差不多相等而互相抵消了，仅靠最后几位小数的精确度来使列表值满足插值函数。这样做的效果与插值多项式的值在其限定点之间摆动的内在倾向是一致的，即便机器的浮点精度非常好时这种情况也会发生。上面的程序 polcoe 和 polcof 对这种异常情况都有不同程度的敏感性。

读者是否仍然认为应用系数是一种好主意呢？

参考文献和深入阅读

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), §5.2.

3.6　二维或高维插值

对多维插值的情况，我们要根据一个列表值 y 的 n 维网格点及 n 个一维向量，求函数 $y(x_1, x_2, \ldots, x_n)$ 的估计，这 n 个一维向量给出每组自变量 $x_1, x_2, \ldots, x_n$ 的列表值。这里我们将不考虑非笛卡儿系统的插值问题，即不考虑表中函数值是建立在 n 维空间中的一些“随机”点上，而不是直角坐标系中的情况。为清楚起见，仅考虑二维的情况，三维或高维的情况可以完全类推。

对二维情况，假定已知函数值的矩阵 ya[0..m-1][0..n-1]、另外还已知数组 x1a[0..m-1]及数组

x2a[0..n-1],这些输入量的关系由函数 $y(x_1, x_2)$确定

$$\texttt{ya[j][k]} = y(\texttt{x1a[j]}, \texttt{x2a[k]}) \tag{3.6.1}$$

我们想用插值法估计函数 y 在表中未列出点(x_1, x_2)的值。

一个重要的概念是**方格网格**,点(x_1, x_2)落在其中,也就是说由列表中的四个点包围起了所求的内部点。为方便起见,从左下角起按逆时针方向从0到3给这些点进行标号(见图3.6.1)。更精确地表示为,如果由

$$\begin{aligned} \texttt{x1a[j]} \leqslant x_1 \leqslant \texttt{x1a[j+1]} \\ \texttt{x2a[k]} \leqslant x_2 \leqslant \texttt{x2a[k+1]} \end{aligned} \tag{3.6.2}$$

定义 j 和 k,则

$$\begin{aligned} y_0 &\equiv \texttt{ya[j][k]} \\ y_1 &\equiv \texttt{ya[j+1][k]} \\ y_2 &\equiv \texttt{ya[j+1][k+1]} \\ y_3 &\equiv \texttt{ya[j][k+1]} \end{aligned} \tag{3.6.3}$$

最简单的二维插值是方格网格上的二维**线性插值**。其公式为

$$\begin{aligned} t &\equiv (x_1 - \texttt{x1a[j]})/(\texttt{x1a[j+1]} - \texttt{x1a[j]}) \\ u &\equiv (x_2 - \texttt{x2a[k]})/(\texttt{x2a[k+1]} - \texttt{x2a[k]}) \end{aligned} \tag{3.6.4}$$

(因此 t 和 u 都在0和1之间)且

$$y(x_1, x_2) = (1-t)(1-u)y_0 + t(1-u)y_1 + tuy_2 + (1-t)uy_3 \tag{3.6.5}$$

二维线性插值类似于"政府部门的行政工作"。在插值点从一个方格网格移到另一个方格网格时,插值函数的值是连续变化的。然而,插值函数的梯度在每个方格网格的边界上是不连续变化的。

将二维线性插值推广到高维时,有明显不同于原来方法的两个方面:可以使用更高的阶获得插值函数精度的提高(是足够平滑的函数),而不必非要满足梯度和高阶导数的连续性。或者使用更高的阶在对方格网格边界点插值时,使一些导数更加平滑。下面讨论一下这两个方面的问题。

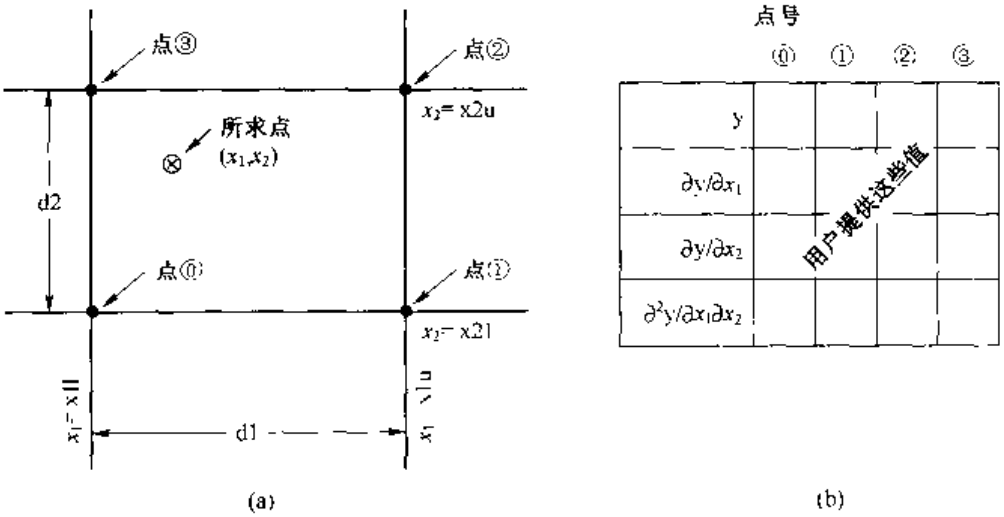


图3.6.1 (a)在二维插值程序 bcuint 和 bcucof 中对点进行标号。(b)对(a)中4个点的每个点,用户要提供一个函数值,两个一阶导数值和一个混合导数值,总共16个数值

3.6.1 用更高的阶获得高精度

基本思想是,将高维插值问题分解成一系列一维插值问题。如果想在 x_1 方向上做 m-1 阶插值,在 x_2 方向上做 n-1 阶插值,就首先设置一个 m×n 的由列表函数矩阵构成的子块,它包含了所求的点(x_1, x_2)。

接下来在 x_2 方向，即子块的行方向上做 m 次一维插值，求得在点(x1a[j], x_2)，j = 0, . . . , m − 1 处的函数值。最后，在 x_1 方向上进行最后一次插值，便得到了解。如果程序使用 3.1 节中的多项式插值程序 polint，且假定已设置好子块(存储在矩阵 ya 中)，则程序如下：

```
#include "nr.h"

void NR::polin2(Vec_I_DP &x1a, Vec_I_DP &x2a, Mat_I_DP &ya, const DP x1,
    const DP x2, DP &y, DP &dy)
给定数组 x1a[0..m-1]和 x2a[0..n-1]为自变量，子矩阵 ya[0..m-1][0..n-1]表示由 x1a 和 x2a 定义的网格点的列表
函数值。另外还知道自变量 x1 和 x2 的值，本程序返回插值函数值 y 及精度表示 dy(仅根据 x1 方向上的插值)。
{
    int j,k;

    int m=x1a.size();
    int n=x2a.size();
    Vec_DP ymtmp(m),ya_t(n);
    for (j=0;j<m;j++) {                          行循环。
        for (k=0;k<n;k++) ya_t[k]=ya[j][k];
        polint(x2a,ya_t,x2,ymtmp[j],dy);         插值结果暂时存储。
    }
    polint(x1a,ymtmp,x1,y,dy);                   进行最后一次插值。
}
```

3.6.2　用更高的阶获得高平滑度：双三次插值

下面给出两个不相关的常用方法。第一个方法通常称为**双三次插值**。

双三次插值要求使用者在每个坐标格点上不仅指定函数 $y(x_1, x_2)$ 的值，而且还要指定梯度 $\partial y/\partial x_1 \equiv y_{,1}$，$\partial y/\partial x_2 \equiv y_{,2}$ 及混合偏导数 $\partial^2 y/\partial x_1 \partial x_2 \equiv y_{,12}$。然后，便可在坐标刻度为 t 和 u((3.6.4)式)处求得**三次**插值函数，它具有下述性质：(i)函数值及指定的导数都是在坐标格点上重新精确产生的；(ii)当插值点由一个方格网格进入另一方格网格时，函数值及指定的导数值是连续变化的。

双三次插值方程并不需要正确说明多余的导数，理解这一点很重要。平滑的性质是有多重"保证"的，与指定导数的"精度"无关。如何获得这些指定的值，那是另一个单独的课题。获得的值越好，插值也就越精确，但是无论怎样插值都是平滑的。

最好是能知道导数的解析值，或是能够用数值方法精确地计算出方格点上的导数。次好的办法是能够从列表中已有的方格点的函数值，通过数值差分来确定这些导数。有关的代码类似下面的形式(使用中心差分)：

```
y1a[j][k]=(ya[j+1][k]-ya[j-1][k])/(x1a[j+1]-x1a[j-1]);
y2a[j][k]=(ya[j][k+1]-ya[j][k-1])/(x2a[k+1]-x2a[k-1]);
y12a[j][k]=(ya[j+1][k+1]-ya[j+1][k-1]-ya[j-1][k+1]+ya[j-1][k-1])
        /((x1a[j+1]-x1a[j-1])*(x2a[k+1]-x2a[k-1]));
```

在方格网格的 4 个顶角点上，已知每个顶角的函数值 y 及导数 y1, y2, y12，构造方格网格内的双三次插值，有两个步骤：首先，用下面的程序 bcucof 得到 16 个量 c_{ij}，$i, j = 0, \ldots, 3$(由函数值及导数值求 c 的公式就是一个复杂的线性变换，系数在求解过程中仅确定一次。可将其制成数表，以后便不用再考虑它了)。其次，将 c 代换到下面任意几个或全部想要求的双三次函数值及导数的公式中：

$$
\begin{aligned}
y(x_1, x_2) &= \sum_{i=0}^{3}\sum_{j=0}^{3} c_{ij} t^i u^j \\
y_{,1}(x_1, x_2) &= \sum_{i=0}^{3}\sum_{j=0}^{3} i c_{ij} t^{i-1} u^j (dt/dx_1) \\
y_{,2}(x_1, x_2) &= \sum_{i=0}^{3}\sum_{j=0}^{3} j c_{ij} t^i u^{j-1} (du/dx_2) \\
y_{,12}(x_1, x_2) &= \sum_{i=0}^{3}\sum_{j=0}^{3} i j c_{ij} t^{i-1} u^{j-1} (dt/dx_1)(du/dx_2)
\end{aligned}
\tag{3.6.6}
$$

其中 t 和 u 也由(3.6.4)式给出。

```
#include "nr.h"

void NR::bcucof(Vec_I_DP &y, Vec_I_DP &y1, Vec_I_DP &y2, Vec_I_DP &y12,
    const DP d1, const DP d2, Mat_O_DP &c)
```

给定数组 y[0..3],y1[0..3],y2[0..3]及 y12[0..3],分别表示矩形坐标区域 4 个顶点(从左下角起按逆时针方向标号)的函数值、梯度及偏导数,而且已知沿 1,2 方向上网格的单位长度 d1 和 d2,本程序返回表 c[0..3][0..3],供程序 bcuint 求双三次插值使用。

```
{
    static int wt_d[16*16]=
        {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
        -3, 0, 0, 3, 0, 0, 0, 0,-2, 0, 0,-1, 0, 0, 0, 0,
        2, 0, 0,-2, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0,
        0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
        0, 0, 0, 0,-3, 0, 0, 3, 0, 0, 0, 0,-2, 0, 0,-1,
        0, 0, 0, 0, 2, 0, 0,-2, 0, 0, 0, 0, 1, 0, 0, 1,
        -3, 3, 0, 0,-2,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,-3, 3, 0, 0,-2,-1, 0, 0,
        9,-9, 9,-9, 6, 3,-3,-6, 6,-6,-3, 3, 4, 2, 1, 2,
        -6, 6,-6, 6,-4,-2, 2, 4,-3, 3, 3,-3,-2,-1,-1,-2,
        2,-2, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 2,-2, 0, 0, 1, 1, 0, 0,
        -6, 6,-6, 6,-3,-3, 3, 3,-4, 4, 2,-2,-2,-2,-1,-1,
        4,-4, 4,-4, 2, 2,-2,-2, 2,-2,-2, 2, 1, 1, 1, 1};
    int l,k,j,i;
    DP xx,d1d2;
    Vec_DP cl(16),x(16);
    static Mat_INT wt(wt_d,16,16);

    d1d2=d1*d2;
    for (i=0;i<4;i++) {          压缩临时向量 x。
        x[i]=y[i];
        x[i+4]=y1[i]*d1;
        x[i+8]=y2[i]*d2;
        x[i+12]=y12[i]*d1d2;
    }
    for (i=0;i<16;i++) {         矩阵乘以存好的表。
        xx=0.0;
        for (k=0;k<16;k++) xx += wt[i][k]*x[k];
        cl[i]=xx;
    }
    l=0;
    for (i=0;i<4;i++)            将结果解压缩成输出表。
        for (j=0;j<4;j++) c[i][j]=cl[l++];
}
```

(3.6.6)式求双三次插值,其实现非常简单,它调用上面的程序 bcucof,并返回插值函数值及两个梯度值:

```
#include "nr.h"

void NR::bcuint(Vec_I_DP &y, Vec_I_DP &y1, Vec_I_DP &y2, Vec_I_DP &y12,
    const DP x1l, const DP x1u, const DP x2l, const DP x2u,
    const DP x1, const DP x2, DP &ansy, DP &ansy1, DP &ansy2)
```

在方格坐标区域内求双三次插值。输入量包括 y,y1,y2,y12(跟 bcucof 要求的一样);x1l 和 x1u 为在第一个方向上,方形坐标区域的上下界坐标;x2l 和 x2u 同理为第二个方向上的上下界坐标;x1,x2 为所求插值点的坐标。ansy 为返回的插值函数值,ansy1 和 ansy2 为插值梯度值。本程序调用 bcucof。

```
{
    int i;
    DP t,u,d1,d2;
    Mat_DP c(4,4);

    d1=x1u-x1l;
    d2=x2u-x2l;
    bcucof(y,y1,y2,y12,d1,d2,c);        得到c值。
    if (x1u == x1l || x2u == x2l)
        nrerror("Bad input in routine bcuint");
```

```
    t=(x1-x1l)/d1;                    (3.6.4)式。
    u=(x2-x2l)/d2;
    ansy=ansy2=ansy1=0.0;
    for (i=3;i>=0;i--) {              (3.6.6)式。
        ansy=t*ansy+((c[i][3]*u+c[i][2])*u+c[i][1])*u+c[i][0];
        ansy2=t*ansy2+(3.0*c[i][3]*u+2.0*c[i][2])*u+c[i][1];
        ansy1=u*ansy1+(3.0*c[3][i]*t+2.0*c[2][i])*t+c[1][i];
    }
    ansy1 /= d1;
    ansy2 /= d2;
}
```

3.6.3　用更高的阶获得高平滑度:双三次样条

获得二维插值高平滑度的另一种常用技术是**双三次样条**。实际上,它相当于双三次插值的一种特殊情况:插值函数与(3.6.6)式的函数形式完全相同,不过坐标方格点的导数值是由一维样条"全局"确定的。然而,双三次样条通常的实现形式,看起来跟上面的双三次插值程序相差甚远,倒是更接近上面的 polin2 程序的形式:求函数插值时,先按列表的行方向求 m 次一维样条插值,再按新建的列方向做一次一维样条插值,这样做在所需的预计算和存储等方面都很得体(时间和空间的权衡)。无需预先计算并存好所有的导数信息(像双三次插值那样),而是像样条插值一样,只需先计算并保存一张辅助表,它是仅沿一个方向上的二阶导数。接下来,只需对 m 行样条做样条求值(不用构造),还要对最后的列样条做一次构造并且求值(回忆一下,构造样条的复杂度为 N 阶,而样条求值仅为 $\log N$ 就可查到表中的位置)。

这里的程序是预先计算辅助的二阶导数表:

```
#include "nr.h"

void NR::splie2(Vec_I_DP &x1a, Vec_I_DP &x2a, Mat_I_DP &ya, Mat_O_DP &y2a)
给定一个 m×n 阶的列表函数 ya[0..m-1][0..n-1]及列表自变量 x2a[0..n-1],本程序构造 ya 的行的一维自然三次
样条,返回的二阶导数存于数组 y2a[0..m-1][0..n-1]中(数组 x1a[0..m-1]包含在变量列表中,这主要是为了保持与
程序 splin2 的一致性)。
{
    int m,n,j,k;

    m=x1a.size();
    n=x2a.size();
    Vec_DP ya_t(n),y2a_t(n);
    for (j=0;j<m;j++) {
        for (k=0;k<n;k++) ya_t[k]=ya[j][k];
        spline(x2a,ya_t,1.0e30,1.0e30,y2a_t);        值1×10^30标志自然样条。
        for (k=0;k<n;k++) y2a[j][k]=y2a_t[k];
    }
}
```

上面的程序执行过一次以后,可以连续调用下面的程序执行任意次求双三次样条插值:

```
#include "nr.h"

void NR::splin2(Vec_I_DP &x1a, Vec_I_DP &x2a, Mat_I_DP &ya, Mat_I_DP &y2a,
    const DP x1, const DP x2, DP &y)
给定 x1a,x2a,ya,m,n,它们的说明见程序 splie2,并由该程序生成 y2a,对给定所求的插值点 x1,x2,本程序用双三次
样条插值求得插值函数值 y。
{
    int j,k;

    int m=x1a.size();
    int n=x2a.size();
    Vec_DP ya_t(n),y2a_t(n),yytmp(m),ytmp(m);
    for (j=0;j<m;j++) {              使用一维样条求值程序splint,并由splie2构造
        for (k=0;k<n;k++) {          的行样条执行m个求值计算。
            ya_t[k]=ya[j][k];
            y2a_t[k]=y2a[j][k];
        }
```

```
            splint(x2a,ya_t,y2a_t,x2,yytmp[j]);
        }
        spline(x1a,yytmp,1.0e30,1.0e30,ytmp);      构造一维列样条并求值计算。
        splint(x1a,yytmp,ytmp,x1,y);
    }
```

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.2.

Kinahan, B.F., and Harm, R. 1975, *Astrophysical Journal*, vol. 200, pp. 330–335.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §5.2.7.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.7.

第4章 函数积分

4.0 引言

数值积分方法又称为**求积法**(quadrature),其历史要追溯到微积分发明之始,甚至更早。初等函数的导数一般能解析地计算出来,而积分却不能,这使得数值积分这一领域的作用更为显著,而不必像在整个18世纪和19世纪中那样,进行繁琐的数值分析的算术运算了。

随着自动计算的发明,求积分成为了一项并不很有趣的数值计算工作。自动计算,即使是最原始的形式,如使用台式计算器和能装满一间房子的"计算机"在内(直到20世纪50年代,其主体是人而不是机器),使微分方程的数值积分这一极为丰富的领域展示了其可行性。求积分仅是其中一种最简单的特例:积分值

$$I = \int_a^b f(x)dx \tag{4.0.1}$$

与对微分方程(4.0.2)求 $I = y(b)$ 的值等价:

$$\frac{dy}{dx} = f(x) \tag{4.0.2}$$

上式有边界条件

$$y(a) = 0 \tag{4.0.3}$$

本书第16章将讨论微分方程的数值积分,着重强调步长的"可变"选择,或"自适应"选择的概念。这里不进一步探讨。如果待积分的函数明显地集中在一个或几个峰值点处,或其形状很难用单个长度标尺描述,那么应该将问题化成(4.0.2)式和(4.0.3)式的形式,然后再采用第16章中介绍的方法。

本章求积分法的原理就是,以一定方式在积分区间内,按横坐标顺序将被积函数的函数值加起来。目标是用最少的求被积函数值的次数,得到尽可能精确的积分值。和内插法(参见第3章)的情况类似,可自由选择不同阶的各种方法,有时(但不总是)越高阶的精度越高。在4.3节讨论的龙贝格(Romberg)积分,是利用各种不同阶的积分方法的一般形式,它是一种值得推荐的方法。

除了本章和第16章的方法外,求积分的方法很多。其中很重要的一种是利用逼近函数。在5.9节将详细讨论切比雪夫(Chebyshev)逼近函数积分。读者应了解如何利用3.3节中程序 spline 的结果来求**三次样条函数**的积分,这里将不进行详细讨论。(提示:用(3.3.3)式对 x 解析积分,参见[1])。

傅里叶变换中涉及的积分可由快速傅里叶变换(FFT)算法算出,将在13.9节中讨论。

多重积分的内容也很多,在4.6节将进行一般性讨论。比较重要的蒙特卡罗积分方法在第7章中介绍。

参考文献和深入阅读

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), Chapter 2.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), Chapter 7.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 4.

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), Chapter 3.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.4.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs,

NJ: Prentice Hall), Chapter 5.
Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.2, p. 89. [1]
Davis, P., and Rabinowitz, P. 1984, *Methods of Numerical Integration*, 2nd ed. (Orlando, FL: Academic Press).

4.1 坐标等距划分的经典公式

每本关于数值分析的书都会谈及 Simpson 先生和他的"法则"。当一个函数在等距点处的值已知,求这个函数积分的经典公式是极精巧的。它让人们联想到它的历史。正是通过这些公式,现代数值分析家们可以与几个世纪前甚至远至牛顿时代的前辈交流思想。但时代的确在变化,除了其中最根本的两个公式("扩展梯形法"(4.1.11)式和"扩展中点法"(4.1.19)式,参见4.2节)以外,这些经典公式几乎都没什么用处了,但它们确实是很经典的。

先做一些约定:一组坐标序列 $x_0, x_1, \ldots, x_{N-1}, x_N$,其间隔为定步长 h,

$$x_i = x_0 + ih \qquad i = 0, 1, \ldots, N \tag{4.1.1}$$

已知函数 $f(x)$ 在各 x_i 处值,

$$f(x_i) \equiv f_i \tag{4.1.2}$$

求函数 $f(x)$ 在下限 a 和上限 b 之间的积分,其中 a 和 b 分别等于其中某一个 x_i 值。使用了在端点的函数值 $f(a)$ 或 $f(b)$ 的积分公式,称为**闭型**公式。有时被积函数在一个或两个端点处的函数值难以计算(如求 f 变成求 0/0 型极限,或者更糟的情况是,端点处有可积奇异性)。此时,就需要一个**开型**公式,即只用严格限于 a 和 b 之间的 x_i 值来估计积分(见图 4.1.1)。

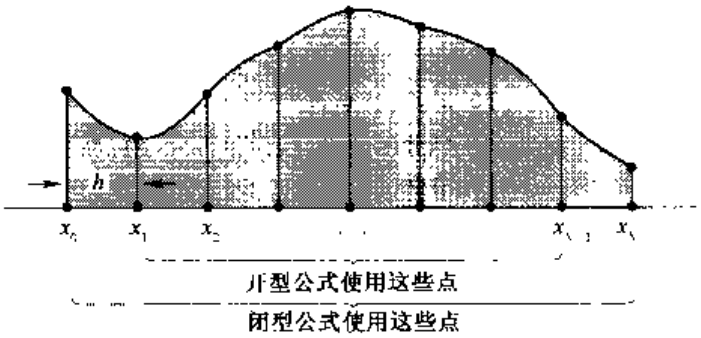


图 4.1.1 用等距划分的坐标来求函数从 x_0 到 x_N 之间积分的求积公式。闭型公式要计算在边界点处的值,而开型公式不这样做(以防边界点上的求值使算法失效)。

经典公式的基本构造模块是,计算函数在一个小区间上的积分的"方法"。随着区间数目的增多,就会发现这些方法对阶数逐渐增高的多项式是精确的(记住在实数情况下,高阶并非意味着高精度)。下面给出一些这样的闭型公式。

4.1.1 Newton-Cotes 闭型公式

梯形法:

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{1}{2}f_0 + \frac{1}{2}f_1\right] \quad + O(h^3 f'') \tag{4.1.3}$$

其中误差项 $O()$ 表示真实值与估计值之差,其大小等于某一数值系数乘以 h^3,再乘以函数在被积区间内某一点处的二阶导数的积。这个数值系数是可知的,可在任何一本关于这一内容的标准参考资料中找到。但应在哪一点求二阶导数却是未知的。要是能知道这一点,就能求出该点的值。由于一个已知数与一个未知数之积仍是一个未知数,因此还照原样书写公式,只写误差项 $O()$,而不写出数值系数。

(4.1.3)式为一个两点公式(x_0 和 x_1)。它对小于等于一次的多项式是精确的,如 $f(x) = x$。有人希望能

有一个适用于小于等于二次多项式的三点公式。确实有这样的公式，而且由于公式左右对称引起系数相消，这种三点公式对小于等于三次的多项式也是精确的，如 $f(x)=x^3$。

Simpson 法：

$$\int_{x_0}^{x_2} f(x)dx = h\left[\frac{1}{3}f_0+\frac{4}{3}f_1+\frac{1}{3}f_2\right] \quad +O(h^5 f^{(4)}) \tag{4.1.4}$$

其中 $f^{(4)}$ 表示函数在积分区间某未知点处的四阶导数值，还要注意，公式给出的是距离为 $2h$ 的区间上的积分，所以系数加起来等于2。

四点公式就没有那么巧的系数相消了，但对于小于等于三次的多项式也是精确的。

Simpson 3/8 法：

$$\int_{x_0}^{x_3} f(x)dx = h\left[\frac{3}{8}f_0+\frac{9}{8}f_1+\frac{9}{8}f_2+\frac{3}{8}f_3\right] \quad +O(h^5 f^{(4)}) \tag{4.1.5}$$

五点公式就又有消去性质了。

Bode 法：

$$\int_{x_0}^{x_4} f(x)dx = h\left[\frac{14}{45}f_0+\frac{64}{45}f_1+\frac{24}{45}f_2+\frac{64}{45}f_3+\frac{14}{45}f_4\right] \quad +O(h^7 f^{(6)}) \tag{4.1.6}$$

上式对小于等于五次的多项式是精确的。

至此，公式不再以人名命名，这里也不再接着探讨下去。其他公式可查阅[1]。

4.1.2 单个区间的外推公式

我们暂时偏离历史发展的顺序，针对这个问题，大多数书上会给出一些 Newton-Cotes 开型公式，例如，

$$\int_{x_0}^{x_5} f(x)dx = h\left[\frac{55}{24}f_1+\frac{5}{24}f_2+\frac{5}{24}f_3+\frac{55}{24}f_4\right] \quad +O(h^5 f^{(4)})$$

注意，上式是求从 $a=x_0$ 到 $b=x_5$ 的积分，却只用了内部点 x_1,x_2,x_3,x_4。我们认为，这种开型公式用处不大，原因是：(i)不能像处理闭型公式那样串起来构成有用的扩展方法；(ii)其他用处都受到高斯积分公式的支配，这一点将在4.5节中介绍。

现在，不从 Newton-Cotes 公式着手，而从 x_0 到 x_1 的单区间内估计积分的公式着手，这一积分用 f 在 x_1，x_2，...处的值表示。这些值对于为"扩展"开型公式建立模块将会十分有用。

$$\int_{x_0}^{x_1} f(x)dx = h[f_1] \quad +O(h^2 f') \tag{4.1.7}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{3}{2}f_1-\frac{1}{2}f_2\right] \quad +O(h^3 f'') \tag{4.1.8}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{23}{12}f_1-\frac{16}{12}f_2+\frac{5}{12}f_3\right] \quad +O(h^4 f^{(3)}) \tag{4.1.9}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{55}{24}f_1-\frac{59}{24}f_2+\frac{37}{24}f_3-\frac{9}{24}f_4\right] \quad +O(h^5 f^{(4)}) \tag{4.1.10}$$

也许读者会问上面的公式是怎样得到的。有很多不错的方法，但最直接的方法是在公式的基本形式中用未知数，比如用 p,q,r,s 代替其中的系数。不失一般性，取 $x_0=0$，$x_1=1$，则 $h=1$。依次用 $f(x)=1$，$f(x)=x$，$f(x)=x^2$，$f(x)=x^3$ 代替 $f(x)$（还有 f_1,f_2,f_3,f_4）。每次求一次积分，使左边简化成为一个数，而右边成为未知数 p,q,r,s 的线性方程。求解由此得到的4个方程，即可求得系数。

4.1.3 扩展公式(闭型)

如果使用(4.1.3)式 $N-1$ 次，在区间 (x_0,x_1)，(x_1,x_2)，...，(x_{N-2},x_{N-1}) 中计算积分，再把结果加起来，就可得到从 x_0 到 x_{N-1} 积分的"扩展"或"合成"公式。

扩展梯形法：

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{1}{2}f_0 + f_1 + f_2 + \cdots + f_{N-2} + \frac{1}{2}f_{N-1}\right] + O\left(\frac{(b-a)^3 f''}{N^2}\right) \tag{4.1.11}$$

这里误差项用区间长度 $b-a$ 和点数 N 来表示,而不用 h。通常 a 和 b 固定,欲知若取步长个数为原来的两倍时,误差项将减少多少,从公式看就很清楚了(此时按4倍关系改变)。所以在后面的公式中,我们仅用步数来标度误差项。

事实上,(4.1.11)式是本节最重要的公式,是大多数实用积分方法的基础,这一点到4.2节会更清楚。

$1/N^3$ 阶的扩展公式为

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{5}{12}f_0 + \frac{13}{12}f_1 + f_2 + f_3 + \cdots + f_{N-3} + \frac{13}{12}f_{N-2} + \frac{5}{12}f_{N-1}\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.12}$$

后面将详细说明怎样得到。

对连续的、不相重叠的区间对,应用(4.1.4)式得到**"扩展 Simpson 法"**:

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{2}{3}f_2 + \frac{4}{3}f_3 + \cdots + \frac{2}{3}f_{N-3} + \frac{4}{3}f_{N-2} + \frac{1}{3}f_{N-1}\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.13}$$

注意,上式中括号内2/3和4/3一直交替出现。许多人以为这种摆动的交替过程,包含了关于被积函数积分的某种不易察觉的性质。事实上,这种交替只是由于用了(4.1.4)式构造模块的结果。

与 Simpson 法同阶的还有另外一个扩展公式:

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{3}{8}f_0 + \frac{7}{6}f_1 + \frac{23}{24}f_2 + f_3 + f_4 + \cdots + f_{N-5} + f_{N-4} + \frac{23}{24}f_{N-3} + \frac{7}{6}f_{N-2} + \frac{3}{8}f_{N-1}\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.14}$$

本式通过对每连续四个一组的点集上拟合三次多项式得到。关于这一点的细节将在18.3节中讨论,该节中为解积分方程使用了相似的方法。现在,可以说明(4.1.12)式怎么得到的了。在第一个区间及最后一个区间里,使用(4.1.3)式的梯形法,其他区间用 Simpson 法,这样得到的改进结果再与直接使用 Simpson 扩展法得到的结果取平均。梯形法比 Simpson 法低**两阶**,这样使得积分的阶次降为 N 的另外一个幂次(因为梯形法只用了两次而不是 N 次)。因此,这使得到的公式比 Simpson 法低一阶。

4.1.4 扩展公式(开型与半开型)

(4.1.11)式~(4.1.14)式加上(4.1.7)式~(4.1.10)式,即可以推导出开型的或半开型的扩展公式,其中将(4.1.7)式~(4.1.10)式用于第一步的外推的开型公式,而(4.1.11)式~(4.1.14)式用于求第二步及后面各步值的闭型公式。正如上面刚刚讨论过的那样,在区间边界求积分的各步,可选用比区间之内各步低一阶的公式。这样,所得区间两端都为开型的公式如下。

由(4.1.7)式和(4.1.11)式得到

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{3}{2}f_1 + f_2 + f_3 + \cdots + f_{N-3} + \frac{3}{2}f_{N-2}\right] + O\left(\frac{1}{N^2}\right) \tag{4.1.15}$$

由(4.1.8)式及(4.1.12)式得到

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{23}{12}f_1 + \frac{7}{12}f_2 + f_3 + f_4 + \cdots + f_{N-4} + \frac{7}{12}f_{N-3} + \frac{23}{12}f_{N-2}\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.16}$$

由(4.1.9)式及(4.1.13)式得到

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{27}{12}f_1 + 0 + \frac{13}{12}f_3 + \frac{4}{3}f_4 + \cdots + \frac{4}{3}f_{N-5} + \frac{13}{12}f_{N-4} + 0 + \frac{27}{12}f_{N-2}\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.17}$$

区间内部各点系数为4/3和2/3交替变化。为消去这种交替变化，由(4.1.9)式及(4.1.14)式可得

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{55}{24}f_1 - \frac{1}{6}f_2 + \frac{11}{8}f_3 + f_4 + f_5 + f_6 + \cdots + f_{N-6} + f_{N-5} + \frac{11}{8}f_{N-4} - \frac{1}{6}f_{N-3} + \frac{55}{24}f_{N-2}\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.18}$$

再简单提一下另一种扩展开型公式，它用于积分限位于标定坐标中点的情况，一般称为**扩展中点法**，其精度与(4.1.15)式是相同的：

$$\int_{x_0}^{x_{N-1}} f(x)dx = h[f_{1/2} + f_{3/2} + f_{5/2} + \cdots + f_{N-5/2} + f_{N-3/2}] + O\left(\frac{1}{N^2}\right) \tag{4.1.19}$$

这种情况也有高阶公式，这里就不介绍了。

半开型公式显然就是(4.1.11)式～(4.1.14)式分别与(4.1.15)式～(4.1.18)式的组合。积分的闭端使用前4个式子中的权，而积分的开端使用后4个式子中的权。举例说明，一个右闭左开误差按$1/N^3$减小的公式为

$$\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{23}{12}f_1 + \frac{7}{12}f_2 + f_3 + f_4 + \cdots + f_{N-3} + \frac{13}{12}f_{N-2} + \frac{5}{12}f_{N-1}\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.20}$$

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.4. [1]

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover), §7.1.

4.2 基本算法

从(4.1.11)式的梯形法开始讨论。至于为什么要从梯形法开始来研究众多其他算法，原因有两个。

显然的原因是,在固定积分限 a 和 b 之间求固定函数 $f(x)$ 的积分时,可以将扩展梯形法中分划区间的数目增加一倍,而不丢失前面工作的结果。实现梯形法最粗糙的步骤是,在积分区域端点 a 和 b 对函数取平均。而改进的第一步是,将中点处函数值加入到这个平均值中,第二步加进 1/4 和 3/4 点处的函数值。依次类推(参见图 4.2.1):

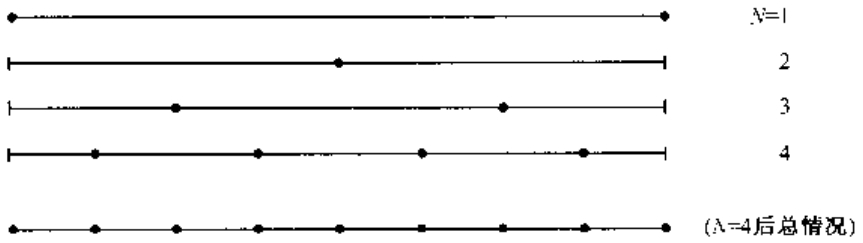


图 4.2.1 顺序调用程序 trapzd,每次可利用前面各次调用的结果,改进时仅需在新增点处求被积函数值。最下面一条线表示经过第4次调用后求函数值的点数。程序 qsimp 给中间结果加权,这样无需多加修改就可由梯形法转变为Simpson法。

可简单地写出程序实现这种算法:

```
#include "nr.h"

DP NR::trapzd(DP func(const DP), const DP a, const DP b, const int n)
本程序计算扩展梯形法第 n 次的加工结果。func 作为积分限为 a 和 b 的被积函数为输入,a 和 b 也为输入。n=1 时调用
程序,返回 ∫_a^b f(x)dx 的粗略估计。随后 n=2,3,...,调用(按顺序)时,每次增加 2^(n-2) 个内点以改善精度。
{
    DP x,tnm,sum,del;
    static DP s;
    int it,j;

    if (n == 1) {
        return (s=0.5*(b-a)*(func(a)+func(b)));
    } else {
        for (it=1,j=1;j<n-1;j++) it <<= 1;
        tnm=it;
        del=(b-a)/tnm;                    这是所增加点的间隔。
        x=a+0.5*del;
        for (sum=0.0,j=0;j<it;j++,x+=del) sum += func(x);
        s=0.5*(s+(b-a)*sum/tnm);          用修正值替代s。
        return s;
    }
}
```

调用上面的程序 trapzd 有很多方式。最简单的一种就是,事先知道(无法想像如何知道)需要多少步的情况下,用扩展梯形法求一个函数的积分。若需 2^M+1 步,可用下面的程序完成,返回结果 s:

```
for(j=1;j<=m+1;j++) s=NR::trapzd(func,a,b,j);
```

当然更好的办法是改进梯形法,以达到某种特殊的精度要求。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::qtrap(DP func(const DP), const DP a, const DP b)
返回函数 func 从 a 到 b 的积分值。参数 EPS 可设置为要求的相对精度,同时设置 JMAX,使允许的最多步数为 2^(JMAX-1),用
梯形法求积分。
{
    const int JMAX=20;
    const DP EPS=1.0e-10;
    int j;
    DP s,olds=0.0;                    olds的初值是任意的。

    for (j=0;j<JMAX;j++) {
```

```
        s=trapzd(func,a,b,j+1);
        if (j > 5)                  避免早期的假收敛性。
            if (fabs(s-olds) < EPS*fabs(olds) ||
                (s == 0.0 && olds == 0.0)) return s;
        olds=s;
    }
    nrerror("Too many steps in routine qtrap");
    return 0.0;                     永远不到此。
}
```

程序 qtrap 虽然不很复杂，但用于求一般不很平滑函数的积分很有效。复杂性增加后，往往可变换成高阶方法，但高阶方法仅当被积函数足够平滑时才有效果。若被积函数的变量是测量数据点之间的线性插值时，则可选用 qtrap 程序方法。注意对 EPS 不能要求过于严格，因为若 qtrap 为达到要求的精度而迭代步数过多时，累积的舍入误差会增大，程序就不能收敛了。一般在双精度情况下，当收敛度适中时，达到 10^{-10} 或更小的精度是没问题的，否则不行。在单精度情况下，精度最高限制在 10^{-6} 水平。

从梯形法出发研究其他算法的另一个原因是，当用 $1/N$ 的幂次表达时，以 $1/N^2$ 阶项开始的逼近误差事实上完全是偶数阶的，由 **Euler-Maclaurin 求和公式**可直接看出

$$\begin{aligned}\int_{x_0}^{x_{N-1}} f(x)dx = h\left[\frac{1}{2}f_0 + f_1 + f_2 + \cdots + f_{N-2} + \frac{1}{2}f_{N-1}\right] \\ - \frac{B_2h^2}{2!}(f'_{N-1} - f'_0) - \cdots - \frac{B_{2k}h^{2k}}{(2k)!}(f^{(2k-1)}_{N-1} - f^{(2k-1)}_0) - \cdots\end{aligned} \tag{4.2.1}$$

其中 B_{2k} 为 **Bernoulli 数**，由下面的生成函数给出：

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!} \tag{4.2.2}$$

前几个偶数项值(除 $B_1 = -1/2$ 外，奇数项都为零)为

$$\begin{aligned}B_0 = 1 \quad B_2 = \frac{1}{6} \quad B_4 = -\frac{1}{30} \quad B_6 = \frac{1}{42} \\ B_8 = -\frac{1}{30} \quad B_{10} = \frac{5}{66} \quad B_{12} = -\frac{691}{2730}\end{aligned} \tag{4.2.3}$$

(4.2.1)式并不是一个收敛展开式，仅为一个近似展开式，表示式中不论从哪一项开始截去，其后面项产生的误差总是小于第一个略去项的两倍。不收敛的原因是 Bernoulli 数会变得很大，如

$$B_{50} = \frac{495057205241079648212477525}{66}$$

关键是，(4.2.1)式中误差级数只有 h 的偶数次幂。一般来说，在4.1节中的高阶求积分方法并不具备这种性质。例如，(4.1.12)式误差的级数就是从 $O(1/N^3)$ 开始，但紧接着还有 N 的所有顺序幂次，如 $1/N^4$，$1/N^5$ 等。

设用 N 步求(4.1.11)式的值，结果为 S_N。再由 $2N$ 步得到 S_{2N}(只需任意接连调用 trapzd 两次)。第二次结果的首项误差只是第一次的 1/4，故组合式

$$S = \frac{4}{3}S_{2N} - \frac{1}{3}S_N \tag{4.2.4}$$

就会消去误差的首项，但(4.2.1)式中 $1/N^3$ 阶的误差项不存在。剩下的误差就是 $1/N^4$ 阶的，与 Simpson 法相同。事实上，不久就会看到(4.2.4)式其实**正是** Simpson 法，系数 2/3，4/3 交替变化。用 Simpson 法求积分，更为可取。由此得到一个与上面的 qtrap 极其类似的程序：

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::qsimp(DP func(const DP), const DP a, const DP b)
```

返回函数 func 从 a 到 b 的积分值。参数 EPS 设置为要求的相对精度，同时设置 JMAX，使得允许的最大步数为 2^{JMAX-1}。用 Simpson 法求积分。

```
{
    const int JMAX=20;
    const DP EPS=1.0e-10;
    int j;
    DP s,st,ost=0.0,os=0.0;

    for (j=0;j<JMAX;j++) {
        st=trapzd(func,a,b,j+1);
        s=(4.0*st-ost)/3.0;          与上面(4.2.4)式比较。
        if (j > 5)                   避免早期的假收敛性。
            if (fabs(s-os) < EPS*fabs(os) ||
                (s == 0.0 && os == 0.0)) return s;
        os=s;
        ost=st;
    }
    nrerror("Too many steps in routine qsimp");
    return 0.0;                      永远不到此。
}
```

一般来说,若被积函数的四阶导数有界(即三阶导数连续),则 qsimp 比 qtrap 效果好(即要求的函数值次数较少)。有时,对于一些简易的工作,此时将 qsimp 与它的必需部分 trapzd 结合起来是一种好方法。

参考文献和深入阅读

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §3.3.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §§7.4.1–7.4.2.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.

4.3 龙贝格积分

可以将龙贝格(Romberg)积分看成是上节的程序 qsimp 的自然推广,其积分格式比 Simpson 法具有更高的阶次。其基本思想是,利用扩展梯形法(由 trapzd 实现)中连续进行 k 次细分的结果,消除误差级数直到(不包括)$O(1/N^{2k})$的所有项。qsimp 程序是 $k=2$ 的特例。这是一例关于通常称为 **Richardson 延迟极限逼近**的一般思想:对不同的参数值 h 完成数值算法,然后将结果外推到连续极限 $h=0$。

(4.2.4)式中减去了误差首项,是多项式外推法的一个特例。在更一般的龙贝格情形下,还可以采用 Neville 算法(参见 3.1 节),将其逐次地精确外推到零步长的情形。在龙贝格积分程序中,能很简洁地编写 Neville 算法。为使程序简洁,调用 3.1 节给出的函数 polint 来做外推,似乎要更好些。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::qromb(DP func(const DP), DP a, DP b)
返回函数 func 从 a 到 b 的积分值,积分采用龙贝格法,阶数为 2K,当 K = 2 时为 Simpson 法。
{
    const int JMAX=20, JMAXP=JMAX+1, K=5;
    const DP EPS=1.0e-10;
    这里EPS为要求的精度,由外推误差估算确定,JMAX限制步长总数;K是用于外推的点数。
    DP ss,dss;
    Vec_DP s(JMAX),h(JMAXP),s_t(K),h_t(K);    存储连续梯形法逼近及其相应步长。
    int i,j;

    h[0]=1.0;
    for (j=1;j<=JMAX;j++) {
        s[j-1]=trapzd(func,a,b,j);
        if (j >= K) {
            for (i=0;i<K;i++) {
```

```
                h_t[i]=h[j-K+i];
                s_t[i]=s[j-K+i];
            }
            polint(h_t,s_t,0.0,ss,dss);
            if (fabs(dss) <= EPS*fabs(ss)) return ss;
        }
        h[j]=0.25*h[j-1];
        关键步:步长减为0.5倍而因子却是0.25,这样使得按(4.2.1)式外推到 h^2 的多项式,而不仅仅是 h 的多项式。
    }
    nrerror("Too many steps in routine qromb");
    return 0.0;                          永远不到此。
}
```

连同所需的 trapzd 及 polint，在不含有奇点且端点为非奇点的区域上用程序 qromb 求足够平滑的(比如，解析的)积分，是很有效的。此时 qromb 求函数值的次数远少于 4.2 节中的任一种方法。例如，积分

$$\int_0^2 x^4 \log(x+\sqrt{x^2+1})dx$$

用程序 qromb 仅需调用 6 次 trapzd，在第二次外推就收敛(应用上述参数)，而 qsimp 需调用 11 次(做了 32 倍之多的积分值计算)，qtrap 需调用 19 次(做了 8192 倍的数值计算)。

参考文献和深入阅读

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §§3.4–3.5.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §§7.4.1–7.4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §4.10–2.

4.4　广义积分

就当前的目的而言，称满足下列条件之一的积分为"广义"的：

- 在有限的积分上下限处函数有有限极限值，但正好在端点之一处不能直接求函数值(如 $\sin x/x$ 在 $x=0$ 处)。
- 上限为 $+\infty$，或下限为 $-\infty$。
- 区间两端有可积奇异点(如 $x^{-1/2}$ 在 $x=0$ 处)。
- 上下限之间某一已知点为可积奇异点。
- 上下限之间某一未知点为可积奇异点。

若积分是无穷大(如 $\int_1^\infty x^{-1}dx$)，或在极限意义下不存在(如 $\int_{-\infty}^\infty \cos x dx$)，都不能称为广义的，而称为不可积的。若本身是不可积的积分，算法再多也没有用。

本节将对上两节的方法进行推广，使其对上面列出的前四个情形仍有效。关于带有可积奇异点的求积问题将在第 18 章进行讨论，详见 18.3 节。关于不可知奇异点的第五个问题，则只能用可变步长的微分方程积分算法，这个问题将在第 16 章讨论。

首先，必须有一个像扩展梯形法((4.1.11)式)那样的基本方法，但必须按 4.1 节的约定是**开型**公式，即不必在端点处求被积函数的值。最好选用扩展中点法的(4.1.19)式。原因是(4.1.19)式与(4.1.11)式一样，具有误差级数都是 h 的偶次幂的性质。有一个不太有名但很有用的公式，称为 **Euler-Maclaurin 第二求和公式**：

$$\int_{x_0}^{x_{N-1}} f(x)dx = h[f_{1/2} + f_{3/2} + f_{5/2} + \cdots + f_{N-5/2} + f_{N-3/2}] + \frac{B_2h^2}{4}(f'_{N-1} - f'_0) + \cdots + \frac{B_{2k}h^{2k}}{(2k)!}(1-2^{-2k+1})(f_{N-1}^{(2k-1)} - f_0^{(2k-1)}) + \cdots \tag{4.4.1}$$

这个公式可由(4.2.1)式导出,先用步长 h 代入(4.2.1)式,再用 $h/2$ 代入,然后用第一次结果减去第二次结果的两倍即可。

扩展中点法不可能将步数增加一倍而仍可以利用前面函数求值的结果(试试看)。但将步数增加到原来的三倍却是可以的。是增加到三倍好,还是增加到两倍而承认这种损失好呢?一般来说,增加到三倍后不必要的工作也增加到$\sqrt{3}$倍,因为达到期望的精度所必需的步数,可能落在三倍引起的对数区间的任何地方。而增加到二倍的话,这个因子仅为$\sqrt{2}$,但不能忽略一个额外的因素,即因不能利用前面的计算结果致使工作量翻倍。由于 $1.732 < 2 \times 1.414$,得知步数增加到原来的三倍要好些。

下面就是要求的程序,可与 trapzd 直接相比。

```
#include "nr.h"

DP NR::midpnt(DP func(const DP), const DP a, const DP b, const int n)
本程序计算扩展中点法第 n 阶段的改进结果。func 为输入,是待积函数。积分限 a 和 b 也为输入。当 n = 1 调用时返回
∫_a^b f(x) dx 的粗略估计。随后 n = 2,3,...调用(按顺序)时,每次增加(2/3) × 3^(n-1)个内点以改善精度。
{
    int it,j;
    DP x,tnm,sum,del,ddel;
    static DP s;

    if (n == 1) {
        return (s=(b-a)*func(0.5*(a+b)));
    } else {
        for(it=1,j=1;j<n-1;j++) it *= 3;
        tnm=it;
        del=(b-a)/(3.0*tnm);
        ddel=del+del;                    增加的点交替地在del和ddel之间划分。
        x=a+0.5*del;
        sum=0.0;
        for (j=0;j<it;j++) {
            sum += func(x);
            x += ddel;
            sum += func(x);
            x += del;
        }
        s=(s+(b-a)*sum/tnm)/3.0;         由新旧积分的组合得到精细的积分值。
        return s;
    }
}
```

在如 4.2 节的 qtrap 驱动程序中,midpnt 程序可以精确地替换 trapzd,可以很容易将 trapzd(func,a,b,j)改为 midpnt(func,a,b,j),可能也要减小参数 JMAX,因为 3^{JMAX-1}(来自三倍步数)比 2^{JMAX-1}(二倍)大得多。

实现开型公式类似于 Simpson 法(4.2 节中的 qsimp),只需用 midpnt 替换 trapzd,再如上述减小 JMAX,现将外推步也改为

```
s=(9.0*st-ost)/8.0;
```

因为步数增加到三倍后,误差降到 1/9 大小,而不像两倍时那样减小到 1/4。

无论是修改后的 qtrap 还是修改后的 qsimp,都适用于本节开头列出的第一类问题。更复杂的是按下面的方式对龙贝格积分进行推广:

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::qromo(DP func(const DP), const DP a, const DP b,
    DP choose(DP (*)(const DP), const DP, const DP, const int))
```

开区间上的龙贝格积分，采用任意指定的积分程序 choose 及龙贝格方法，返回函数 func 从 a 到 b 的积分值。choose 应为开型公式，不用求函数在端点处的值。choose 每次调用时将步数增加三倍，其误差级数只含有步数的偶次幂项。choose 可以选用 midpnt，midinf，midsql，midsqu，midexp 中任一程序。以下常数与程序 qromb 中的意义相同。

```
{
    const int JMAX=14, JMAXP=JMAX+1, K=5;
    const DP EPS=3.0e-9;
    int i,j;
    DP ss,dss;
    Vec_DP h(JMAXP),s(JMAX),h_t(K),s_t(K);

    h[0]=1.0;
    for (j=1;j<=JMAX;j++) {
        s[j-1]=choose(func,a,b,j);
        if (j >= K) {
            for (i=0;i<K;i++) {
                h_t[i]=h[j-K+i];
                s_t[i]=s[j-K+i];
            }
            polint(h_t,s_t,0.0,ss,dss);
            if (fabs(dss) <= EPS*fabs(ss)) return ss;
        }
        h[j]=h[j-1]/9.0;          这里步长增加二倍，且误差级数为偶次。
    }
    nrerror("Too many steps in routine qromo");
    return 0.0;                   永远不到此。
}
```

不要为程序 qromo 中难懂的参数声明而烦恼，下例是一种典型的调用形式（贝塞尔函数 $Y_0(x)$ 从 0 至 2 的积分）：

```
#include "nr.h"
using namespace NR;
DP answer;
...
answer=qromo(bessy0,0.0,2.0,midpnt);
```

qromo 与 qromb（参见 4.3 节）的差别非常微小，以至于似乎没有必要把 qromo 全部写出来。但用它来解决我们前面列出的广义积分问题的确非常好（除较难的第 h 个问题外）。

广义积分的基本技巧是进行变量替换，消去奇异点，或将无穷的积分区间映射到有限的积分区间，例如恒等式

$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right) dt \qquad ab > 0 \tag{4.4.2}$$

适用于 $b \to \infty$，a 为正数；或是 $a \to -\infty$，b 为负数的情况，并且任意函数的收敛速度要比 $1/x^2$ 的收敛速度快。

可先进行(4.4.2)式中的变量替换，然后用 qromo 和 midpnt 进行数值计算，或者也可以在数值算法中进行变量替换。倾向于用后一种方法，因为它对用户更加透明。仅对 midpnt 进行一些修改即可实现(4.4.2)式，称为 midinf。b 可为无穷大（准确地说，应是一个很大的数，如 1×10^{30}），或 a 为负无穷大。

```
#include "nr.h"

namespace {
    DP func(DP funk(const DP), const DP x)
    {
        return funk(1.0/x)/(x*x);          变量的变换效果。
    }
}

DP NR::midinf(DP funk(const DP), const DP aa, const DP bb, const int n)
```

本程序可用于精确替换 midpnt，即返回值为从 aa 到 bb 对 funk 求积分的第 n 次的改进结果，只是应按 $1/x$ 均匀分布的点求函数值，而不是按 x。这样上限 bb 可以是计算机允许的正的极大值，或是下限 aa 为负的极小值，但这两种情况不能同时出现，aa 和 bb 符号必须相同。

```
{
    DP x,tnm,sum,del,ddel,b,a;
    static DP s;
    int it,j;

    b=1.0/aa;                    这两行改变积分限。
    a=1.0/bb;
    if (n == 1) {                从这里起，例程与midpnt完全一样。
        return (s=(b-a)*func(funk,0.5*(a+b)));
    } else {
        for(it=1,j=1;j<n-1;j++) it *= 3;
        tnm=it;
        del=(b-a)/(3.0*tnm);
        ddel=del+del;
        x=a+0.5*del;
        sum=0.0;
        for (j=0;j<it;j++) {
            sum += func(funk,x);
            x += ddel;
            sum += func(funk,x);
            x += del;
        }
        return (s=(s+(b-a)*sum/tnm)/3.0);
    }
}
```

若需从负的下极限到正无穷大进行积分，可在某个正数点处将积分分成两部分进行，例如

```
answer=qromo(funk,-5.0,2.0,midpnt)+qromo(funk,2.0,1.0e30,midinf);
```

如何选择分割点呢？应选取一个充分大的正数点，使函数 funk 至少在这点就开始下降，而一直下降到无穷大处，函数趋近零。隐含于第二次调用 qromo 中的多项式外推是处理 $1/x$ 的多项式，而不是 x 的多项式。

当积分下限是一个可积的、满足幂规律的奇异点时，也可进行变量替换。若被积函数在 $x=a$ 附近按 $(x-a)^{-\gamma}$，$0\leqslant\gamma<1$形式发散，则用恒等式

$$\int_a^b f(x)dx=\frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}}f(t^{\frac{1}{1-\gamma}}+a)dt \qquad (b>a) \tag{4.4.3}$$

若奇异点在上限处，则用恒等式

$$\int_a^b f(x)dx=\frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}}f(b-t^{\frac{1}{1-\gamma}})dt \qquad (b>a) \tag{4.4.4}$$

若上下限都是奇异点，则在中间某处将积分分开，如上面的例子

在逆平方根奇异点的情况下，(4.4.3)式和(4.4.4)式就特别简单，这种情况在实际中经常出现：

$$\int_a^b f(x)dx=\int_0^{\sqrt{b-a}} 2tf(a+t^2)dt \qquad (b>a) \tag{4.4.5}$$

上式中 a 处为奇异点。而 b 处为奇异点时，则

$$\int_a^b f(x)dx=\int_0^{\sqrt{b-a}} 2tf(b-t^2)dt \qquad (b>a) \tag{4.4.6}$$

这里再一次编写对用户透明的替换 midpnt 的程序，以实现变量替换，它能自动进行变量替换：

```
#include <cmath>
#include "nr.h"
using namespace std;

namespace {
    DP func(DP funk(const DP), const DP aa, const DP x)
    {
        return 2.0*x*funk(aa+x*x);
    }
}

DP NR::midsql(DP funk(const DP), const DP aa, const DP bb, const int n)
```

本程序可精确替换 midpnt，只是被积函数在下限 aa 处允许有一个逆平方根的奇异点。

```
{
    DP x,tnm,sum,del,ddel,a,b;
    static DP s;
    int it,j;

    b=sqrt(bb-aa);
    a=0.0;
    if (n == 1) {
    省略剩下的程序,因为除了函数 func 的调用序列不同外,与 midinf 相同。
```

同样有

```
#include <cmath>
#include "nr.h"
using namespace std;

namespace {
    DP func(DP funk(const DP), const DP bb, const DP x)
    {
        return 2.0*x*funk(bb-x*x);
    }
}

DP NR::midsqu(DP funk(const DP), const DP aa, const DP bb, const int n)
本程序可精确替换 midpnt,只是被积函数在上限 bb 处允许有一个逆平方根奇异点。
{
    DP x,tnm,sum,del,ddel,a,b;
    static DP s;
    int it,j;

    b=sqrt(bb-aa);
    a=0.0;
    if (n == 1) {
  省略剩下的程序,因为除了函数 func 的调用序列不同外,与 midinf 相同。
```

最后举一个例子,它足以概括地说明这些公式是如何导出的。设积分上限为无穷大,被积函数呈指数衰减,则需有变量替换,即将 $e^{-x}dx$ 变换为$(\pm)dt$(选择正负号以使新变量的上限大于下限)。可由下面的替换求积分:

$$t = e^{-x} \qquad 或 \qquad x = -\log t \tag{4.4.7}$$

这样就使得

$$\int_{x=a}^{x=\infty} f(x)dx = \int_{t=0}^{t=e^{-a}} f(-\log t)\frac{dt}{t} \tag{4.4.8}$$

实现对用户透明的程序为

```
#include <cmath>
#include "nr.h"
using namespace std;

namespace {
    DP func(DP funk(const DP), const DP x)
    {
        return funk(-log(x))/x;
    }
}

DP NR::midexp(DP funk(const DP), const DP aa, const DP bb, const int n)
本程序可精确替换 midpnt,只是 bb 假定为无穷大(这个无穷值在使用时并不需要)。假设函数 funk 在无限远处呈指数迅
速衰减。
{
    DP x,tnm,sum,del,ddel,a,b;

     static DP s;
     int it,j;

     b=exp(-aa);
     a=0.0;
     if (n == 1) {
     省略剩下的程序,因为与 midinf 相同。
```

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.4.3, p. 294.

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §3.7.

4.5 高斯求积法与正交多项式

在4.1节的公式中,函数积分是用一系列等间隔点上的函数值,乘以一些精心选择的权值系数后求和来近似的。我们看到,只要选择系数时自由性越大,就可以得到越来越高阶的积分公式。**高斯求积法**的思想是不仅能给我们选择系数的自由,而且还能自由选择在哪些坐标位置进行函数求值,这些坐标点不再是等间隔的。这种处理有两个自由度,这就导致了高斯求积分公式的阶次是有同样多个函数值的Newton-Cotes求积公式的阶次的两倍。

听来好得令人难以置信,但的确如此,其中关键的一点已熟悉得不必再强调了,那就是高阶次并非就意味着高精度。仅当被积函数很平滑时(指在可被多项式很好地逼近的意义下),阶次高才有高精度。

此外,高斯求积分公式的另外一个特性增加了它的实用性,就是对由"多项式乘以某已知函数 $W(x)$"得到的一类被积函数,可以通过选择权系数和坐标值使求得的积分很精确,而对一般"多项式"被积函数并不能如此。这时,可选择函数 $W(x)$以从所求的积分中消去可积奇异点。也就是说,给定 $W(x)$及整数 N,可以找到一系列权 w_j 和一系列坐标 x_j,当 $f(x)$为多项式时,使得逼近式

$$\int_a^b W(x)f(x)dx \approx \sum_{j=0}^{N-1} w_j f(x_j) \tag{4.5.1}$$

是精确的,例如求积分

$$\int_{-1}^{1} \frac{\exp(-\cos^2 x)}{\sqrt{1-x^2}}dx \tag{4.5.2}$$

(必须承认,它看上去不是很自然的积分。)我们感兴趣的是,基于下面选择的高斯求积分公式

$$W(x) = \frac{1}{\sqrt{1-x^2}} \tag{4.5.3}$$

积分区间为$(-1,1)$。这种变换方法称为**高斯-切比雪夫积分**,这样做的原因后面会说明。

注意,积分公式(4.5.1)也可用权函数 $W(x)$写成并不显而易见的形式:令$g(x) \equiv W(x)f(x)$且 $v_j \equiv w_j/W(x_j)$,则(4.5.1)式变成

$$\int_a^b g(x)dx \approx \sum_{j=0}^{N-1} v_j g(x_j) \tag{4.5.4}$$

函数 $W(x)$正隐含在其中。这里暗示说明,对形如多项式乘以 $W(x)$的被积函数能达到较高阶精度,而对非常光滑且性质很好的函数却反而达不到高阶精度。首先要确定是采用(4.5.1)式的形式,还是采用(4.5.4)式的形式来求积分,然后再对给定的 $W(x)$求权和坐标列表。

下面的例子是 $W(x)=1$、$N=10$时的积分程序,其中有已给定的坐标列表和权。此时权和坐标点关于积分中心点对称,故实际上只有5个不同值:

```
#include "nr.h"

DP NR::qgaus(DP func(const DP), const DP a, const DP b)
```

返回函数 func 从 a 到 b 的积分,由10点高斯-勒让德积分完成:仅在积分区域的内点上求10次函数值。

```
{
    坐标点和权。
    static const DP x[]={0.1488743389816312,0.4333953941292472,
        0.6794095682990244,0.8650633666889845,0.9739065285171717};
    static const DP w[]={0.2955242247147529,0.2692667193099963,
        0.2190863625159821,0.1494513491505806,0.0666713443086881};
    int j;
    DP xr,xm,dx,s;

    xm=0.5*(b+a);
    xr=0.5*(b-a);
    s=0;                        将为函数平均值的两倍，因为10个权（上面5个数
    for (j=0;j<5;j++) {         每个用2次）总和为2。
        dx=xr*x[j];
        s += w[j]*(func(xm+dx)+func(xm-dx));
    }
    return s *= xr;             改变答案以达到积分的范围。
}
```

上面的程序表明，即使不懂高斯求积法理论，也同样可以用这种方法：只要在书上（例如本节参考文献[1]和[2]）找到列表的权和坐标点即可。不过这一理论十分精彩，若选用某一特别的 $W(x)$ 需自己构造权和坐标点列表时，则了解一些这方面的理论将很有帮助。因此，这里不加证明地给读者提供一些有用结论，结论中假定 $W(x)$ 在 (a,b) 区间内不改变符号，实际中通常也是如此。

高斯求积分理论要追溯到1814年，当时高斯是用连分数来解决这个问题的。1826年雅可比(Jacobi)用正交多项式又推出高斯的结果。用正交多项式对任意权函数系统的处理方法，则主要归功于克里斯托费尔(Christoffel)在1877年的研究。在介绍正交多项式之前，我们先将感兴趣的积分区间固定为 (a,b)。定义"两个函数 f 和 g 对权函数 W 的**数积**"为

$$\langle f|g\rangle \equiv \int_a^b W(x)f(x)g(x)dx \tag{4.5.5}$$

数积是一个数，而非 x 的函数。若数积为零，则称两个函数**正交**。若函数与其自身的数积为1，则称函数是**归一的**。一组两两正交且分别归一化了的函数的集合称为**正交归一集**。

存在一个多项式集合：(i)对每一个 $j=0,1,2,\ldots$，恰好只包含一个 j 次多项式，记为 $p_j(x)$；(ii)对某一特定的权函数 $W(x)$，这些多项式两两正交。以下的递推关系是用于寻找这种集合的构造过程。

$$\begin{aligned} p_{-1}(x) &\equiv 0 \\ p_0(x) &\equiv 1 \\ p_{j+1}(x) &= (x-a_j)p_j(x) - b_j p_{j-1}(x) \qquad j=0,1,2,\ldots \end{aligned} \tag{4.5.6}$$

其中

$$\begin{aligned} a_j &= \frac{\langle xp_j|p_j\rangle}{\langle p_j|p_j\rangle} \qquad j=0,1,\ldots \\ b_j &= \frac{\langle p_j|p_j\rangle}{\langle p_{j-1}|p_{j-1}\rangle} \qquad j=1,2,\ldots \end{aligned} \tag{4.5.7}$$

系数 b_0 是任意的，可以取它为零。

由(4.5.6)式得到的多项式为**首一多项式**，即其首项（如 $p_j(x)$ 的 x^j）系数为1。将每一个 $p_j(x)$ 除以常数 $[\langle p_j|p_j\rangle]^{1/2}$ 即可得到正交归一的多项式集。也可用其他方法对多项式归一化。要把某已知归一多项式转换成首一多项式，只要知道 p_j 中 x^j 的系数，设为 λ_j，然后用 λ_j 除以 p_j 中各项即可得到首一多项式。注意在递推关系式(4.5.6)中，这个系数依赖于所采用的归一化方程。

多项式 $p_j(x)$ 在 (a,b) 区间内，恰好有 j 个不相等的根，且这些根分别插在 $p_{j-1}(x)$ 的 $j-1$ 个根之间，即 $p_{j-1}(x)$ 的每两个根之间恰好有一个 $p_j(x)$ 的根。利用此性质可方便找到根：可以从 $p_1(x)$ 的一个根开始，依次将每个 j，用牛顿法或其他求根方法精确地找到 $p_j(x)$ 的根（参见第9章）。

为什么要找到正交归一多项式 $p_j(x)$ 的所有根呢？因为在区间 (a,b) 内，权函数为 $W(x)$ 的 N 点高斯求

积分式为(4.5.1)式和(4.5.4)式,它的坐标点恰好是同样区间内同样权函数的正交多项式 $p_N(x)$的根。这就是高斯求积法的基本定理,由此可在任何特殊情况下找到坐标点。

找到横坐标点 $x_0,\ldots,x_{N-1}$后,就得找权 $w_j, j=0,\ldots,N-1$,其方法之一(不是最有效的)是解线性方程组

$$\begin{bmatrix} p_0(x_0) & \cdots & p_0(x_{N-1}) \\ p_1(x_0) & \cdots & p_1(x_{N-1}) \\ \vdots & & \vdots \\ p_{N-1}(x_0) & \cdots & p_{N-1}(x_{N-1}) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \end{bmatrix} = \begin{bmatrix} \int_a^b W(x)p_0(x)dx \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.5.8}$$

(4.5.8)式简单地解决了权的问题,根据(4.5.1)式就能计算出前 N 个正交多项式的正确积分值。注意(4.5.8)式中右边出现零,这是因为 $p_1(x),\ldots,p_{N-1}(x)$都与 $p_0(x)$正交,而 $p_0(x)$为常量。可以看出,由所得到的权,接下来 $N-1$ 个多项式的积分也是精确的。这就是说,求积公式对小于等于 $2N-1$ 次的多项式都是精确的。另外,还可以用以下公式求权(其证明超出了本书的讨论范围):

$$w_j = \frac{\langle p_{N-1}|p_{N-1}\rangle}{p_{N-1}(x_j)p'_N(x_j)} \tag{4.5.9}$$

其中 $p'_N(x_j)$为正交多项式在其零点 x_j 处的导数。

可见高斯求积计算法明显分两步:(i)得到正交多项式 $p_0,\ldots,p_N$,即求(4.5.6)式中系数 a_j 和 b_j;(ii)求 $p_N(x)$的零点及其相应的权值。对于一些经典的正交多项式,系数 a_j 和 b_j 已经算出(下面(4.5.10)式~(4.5.14)式),故第一步可以省略。但是若碰到"非经典"的权函数 $W(x)$,而且不知道系数 a_j 和 b_j,此时构造一个相关正交多项式集就很重要。本节的最后将讨论这个问题。

4.5.1 横坐标和权的计算

这件工作难易的程度,取决于读者对权函数及其相关多项式的了解程度。若是很经典、已被仔细研究过的正交多项式,那问题就很清楚了,包括其零点的良好逼近值。此时可将其逼近值作为初始猜测值,应用牛顿法(将在9.4节中讨论)就可很快地收敛,牛顿法中需要的导数 $p'_N(x)$也可以根据 p_N 和 p_{N-1}由标准关系式导出。权则由(4.5.9)式方便地得到。在下面谈到的著名示例中,这种直接求根方法比任何其他方法都快3到5倍。

下面列出一些权函数、区间及递推关系式,由这些可以求得最常用的正交多项式及其相应的高斯求积分公式。

高斯-勒让德(Gauss-Legendre):

$$\begin{gathered} W(x)=1 \qquad -1<x<1 \\ (j+1)P_{j+1}=(2j+1)xP_j-jP_{j-1} \end{gathered} \tag{4.5.10}$$

高斯-切比雪夫(Gauss-Chebyshev):

$$\begin{gathered} W(x)=(1-x^2)^{-1/2} \qquad -1<x<1 \\ T_{j+1}=2xT_j-T_{j-1} \end{gathered} \tag{4.5.11}$$

高斯-拉盖尔(Gauss-Laguerre):

$$\begin{gathered} W(x)=x^\alpha e^{-x} \qquad 0<x<\infty \\ (j+1)L_{j+1}^\alpha=(-x+2j+\alpha+1)L_j^\alpha-(j+\alpha)L_{j-1}^\alpha \end{gathered} \tag{4.5.12}$$

高斯-埃尔米特(Gauss-Hermite):

$$\begin{gathered} W(x)=e^{-x^2} \qquad -\infty<x<\infty \\ H_{j+1}=2xH_j-2jH_{j-1} \end{gathered} \tag{4.5.13}$$

高斯-雅可比(Gauss-Jacobi):

$$W(x) = (1-x)^{\alpha}(1+x)^{\beta} \qquad -1 < x < 1$$
$$c_j P_{j+1}^{(\alpha,\beta)} = (d_j + e_j x) P_j^{(\alpha,\beta)} - f_j P_{j-1}^{(\alpha,\beta)} \tag{4.5.14}$$

其中系数 c_j, d_j, e_j, f_j 为

$$\begin{aligned} c_j &= 2(j+1)(j+\alpha+\beta+1)(2j+\alpha+\beta) \\ d_j &= (2j+\alpha+\beta+1)(\alpha^2-\beta^2) \\ e_j &= (2j+\alpha+\beta)(2j+\alpha+\beta+1)(2j+\alpha+\beta+2) \\ f_j &= 2(j+\alpha)(j+\beta)(2j+\alpha+\beta+2) \end{aligned} \tag{4.5.15}$$

现在,分别给出每种情况下计算横坐标点和权的程序。首先是最常用的高斯－勒让德横坐标和权集。本程序由 G.B.Rybicki 设计,(4.5.9)式在高斯－勒让德情况下的特殊形式为

$$w_j = \frac{2}{(1-x_j^2)[P_N'(x_j)]^2} \tag{4.5.16}$$

在程序中,还将积分区间(x_1, x_2)规划为$(-1,1)$,输出结果为如下高斯求积公式需要的坐标点 x_j 和权值 w_j:

$$\int_{x_1}^{x_2} f(x)dx = \sum_{j=0}^{N-1} w_j f(x_j) \tag{4.5.17}$$

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::gauleg(const DP x1, const DP x2, Vec_O_DP &x, Vec_O_DP &w)
给定积分上下限 x1 和 x2,本程序返回长度为 n 的数组 x[0..n-1]和 w[0..n-1],其中分别存放 n 点高斯－勒让德求积
分公式的坐标点权值。
{
    const DP EPS=1.0e-14;                    EPS为相对精度。
    int m,j,i;
    DP z1,z,xm,xl,pp,p3,p2,p1;               高精度对本例程是个好主意。

    int n=x.size();
    m=(n+1)/2;                               区间中根是对称的，这样只需求其中的一半。
    xm=0.5*(x2+x1);
    xl=0.5*(x2-x1);
    for (i=0;i<m;i++) {                      对求的根作循环。
        z=cos(3.141592654*(i+0.75)/(n+0.5));
        从上面对第 i 个根的逼近值开始,进入用牛顿法进行细划分的主循环。
        do {
            p1=1.0;
            p2=0.0;
            for (j=0;j<n;j++) {          从递推关系循环求得勒让德多项式在z点的值。
                p3=p2;
                p2=p1;
                p1=((2.0*j+1.0)*z*p2-j*p3)/(j+1);
            }
            现在 p1 就是所求勒让德多项式,下面将由它与 p2 的标准关系式计算其导数 pp。p2 是比 p1 低一阶的多
            项式。
            pp=n*(z*p1-p2)/(z*z-1.0);
            z1=z;
            z=z1-p1/pp;                      牛顿法。
        } while (fabs(z-z1) > EPS);
        x[i]=xm-xl*z;                        将根值映射到指定区间。
        x[n-1-i]=xm+xl*z;                    再找其对称的关联点。
        w[i]=2.0*xl/((1.0-z*z)*pp*pp);
        w[n-1-i]=w[i];                       计算权及其对称关联值。
    }
}
```

下面的三个程序是根据 Stroud 和 Secrest[2]给出的根作为初始逼近值。第一个是高斯－拉盖尔的坐标点和权值,应用于下面的积分公式:

$$\int_0^\infty x^\alpha e^{-x} f(x)dx = \sum_{j=0}^{N-1} w_j f(x_j) \tag{4.5.18}$$

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::gaulag(Vec_O_DP &x, Vec_O_DP &w, const DP alf)
给定 alf,即拉盖尔多项式的参数α,本程序返回数组 x[0..n-1]和 w[0..n-1],其中分别存放 n 点高斯-拉盖尔求积分
公式的坐标点和权。最小坐标以 x[0]返回,最大以 x[n-1]返回。
{
    const int MAXIT=10;
    const DP EPS=1.0e-14;                    EPS是相对精度。
    int i,its,j;
    DP ai,p1,p2,p3,pp,z,z1;                  高精度对本例程是个好主意。

    int n=x.size();
    for (i=0;i<n;i++) {                      在求的根作循环。
        if (i == 0) {                        最小根的初始猜测值。
            z=(1.0+alf)*(3.0+0.92*alf)/(1.0+2.4*n+1.8*alf);
        } else if (i == 1) {                 第二个根的初始猜测值。
            z += (15.0+6.25*alf)/(1.0+0.9*alf+2.5*n);
        } else {                             其他根的初始猜测值。
            ai=i-1;
            z += ((1.0+2.55*ai)/(1.9*ai)+1.26*ai*alf/
                (1.0+3.5*ai))*(z-x[i-2])/(1.0+0.3*alf);
        }
        for (its=0;its<MAXIT;its++) {        牛顿法进行细分。
            p1=1.0;
            p2=0.0;
            for (j=0;j<n;j++) {              从递推关系求得拉盖尔多项式在z点的值。
                p3=p2;
                p2=p1;
                p1=((2*j+1+alf-z)*p2-(j+alf)*p3)/(j+1);
            }
            现在 p1 就是所求拉盖尔多项式。下面将由它与 p2 的标准关系式计算其导数 pp。p2 是比 p1 低一阶的多
            项式。
            pp=(n*p1-(n+alf)*p2)/z;
            z1=z;
            z=z1-p1/pp;                      牛顿公式。
            if (fabs(z-z1) <= EPS) break;
        }
        if (its >= MAXIT) nrerror("too many iterations in gaulag");
        x[i]=z;                              存储根和权。
        w[i] = -exp(gammln(alf+n)-gammln(DP(n)))/(pp*n*p2);
    }
}
```

下面的程序计算的是高斯-埃尔米特坐标点和权。若使用函数的"标准"归一化值,如(4.5.13)式,我们发现当 N 很大时计算会溢出,这是因为会产生不同的阶乘。为避免产生这种情况,可以使用多项式 $\widetilde{H}_j$ 的正交归一化集,此集合可由下面的递推式产生:

$$\widetilde{H}_{-1} = 0, \quad \widetilde{H}_0 = \frac{1}{\pi^{1/4}}, \quad \widetilde{H}_{j+1} = x\sqrt{\frac{2}{j+1}}\widetilde{H}_j - \sqrt{\frac{j}{j+1}}\widetilde{H}_{j-1} \tag{4.5.19}$$

求权的公式变成

$$w_j = \frac{2}{[\widetilde{H}'_N(x_j)]^2} \tag{4.5.20}$$

求归一化导数的公式为

$$\widetilde{H}'_j = \sqrt{2j}\widetilde{H}_{j-1} \tag{4.5.21}$$

由程序 gauher 返回的坐标点和权可用于积分公式

$$\int_{-\infty}^{\infty} e^{-x^2} f(x)dx = \sum_{j=0}^{N-1} w_j f(x_j) \tag{4.5.22}$$

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::gauher(Vec_O_DP &x, Vec_O_DP &w)
本程序返回数组 x[0..n-1]和 w[0..n-1],其中分别存放 n 点高斯-埃尔米特求积分公式的坐标点和权。最大坐标以
x[0]返回,最小负坐标以 x[n-1]返回.
{
    const DP EPS=1.0e-14,PIM4=0.7511255444649425;
    Relative precision and 1/π^(1/4).
    const int MAXIT=10;                          最多迭代次数。
    int i,its,j,m;
    DP p1,p2,p3,pp,z,z1;                         高精度对本例程是一个好主意。

    int n=x.size();
    m=(n+1)/2;
    根对原点对称, 所以只求它一半。
    for (i=0;i<m;i++) {                          对所求的根循环。
        if (i == 0) {                            最大根的初始猜测值。
            z=sqrt(DP(2*n+1))-1.85575*pow(DP(2*n+1),-0.16667);
        } else if (i == 1) {                     第二大根的初始猜测值。
            z -= 1.14*pow(DP(n),0.426)/z;
        } else if (i == 2) {                     第三大根的初始猜测值。
            z=1.86*z-0.86*x[0];
        } else if (i == 3) {                     第四大根的初始猜测值。
            z=1.91*z-0.91*x[1];
        } else {                                 其他根的初始猜测值。
            z=2.0*z-x[i-2];
        }
        for (its=0;its<MAXIT;its++) {            由牛顿法进行细分。
            p1=PIM4;
            p2=0.0;
            for (j=0;j<n;j++) {                  由递推式循环求得埃尔米特多项式
                p3=p2;                           在z处的值。
                p2=p1;
                p1=z*sqrt(2.0/(j+1))*p2-sqrt(DP(j)/(j+1))*p3;
            }
            p1 就是所求埃尔米特多项式,下面将利用比 p1 低一阶的多项式 p2 的(4.5.21)式求得其导数 pp。
            pp=sqrt(DP(2*n))*p2;
            z1=z;
            z=z1-p1/pp;                          牛顿公式。
            if (fabs(z-z1) <= EPS) break;
        }
        if (its >= MAXIT) nrerror("too many iterations in gauher");
        x[i]=z;                                  存储根及权。
        x[n-1-i] = -z;
        w[i]=2.0/(pp*pp);                        计算权及其对称关联值。
        w[n-1-i]=w[i];
    }
}
```

最后,给出计算高斯-雅可比坐标点和权的程序,这些坐标值和权值用于下面的积分公式:

$$\int_{-1}^{1} (1-x)^{\alpha}(1+x)^{\beta} f(x)dx = \sum_{j=0}^{N-1} w_j f(x_j) \tag{4.5.23}$$

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::gaujac(Vec_O_DP &x, Vec_O_DP &w, const DP alf, const DP bet)
给定 alf 及 bet,即雅可比多项式的参数α和β,本程序返回数组 x[0..n-1]和 w[0..n-1],其中分别存放 n 点高斯-雅可
比求积分公式的坐标点和权。最大坐标以 x[0]返回,最小坐标以 x[n-1]返回。
```

```
{
    const int MAXIT=10;
    const DP EPS=1.0e-14;                    EPS是相对精度。
    int i,its,j;
    DP alfbet,an,bn,r1,r2,r3;
    DP a,b,c,p1,p2,p3,pp,temp,z,z1;          高精度对本例程是一个好主意。

    int n=x.size();
    for (i=0;i<n;i++) {                      对所求的根循环。
        if (i == 0) {                        最大根的初始猜测值。
            an=alf/n;
            bn=bet/n;
            r1=(1.0+alf)*(2.78/(4.0+n*n)+0.768*an/n);
            r2=1.0+1.48*an+0.96*bn+0.452*an*an+0.83*an*bn;
            z=1.0-r1/r2;
        } else if (i == 1) {                 第二大根的初始猜测值。
            r1=(4.1+alf)/((1.0+alf)*(1.0+0.156*alf));
            r2=1.0+0.06*(n-8.0)*(1.0+0.12*alf)/n;
            r3=1.0+0.012*bet*(1.0+0.25*fabs(alf))/n;
            z -= (1.0-z)*r1*r2*r3;
        } else if (i == 2) {                 第三大根的初始猜测值。
            r1=(1.67+0.28*alf)/(1.0+0.37*alf);
            r2=1.0+0.22*(n-8.0)/n;
            r3=1.0+8.0*bet/((6.28+bet)*n*n);
            z -= (x[0]-z)*r1*r2*r3;
        } else if (i == n-2) {               次小根的初始猜测值。
            r1=(1.0+0.235*bet)/(0.766+0.119*bet);
            r2=1.0/(1.0+0.639*(n-4.0)/(1.0+0.71*(n-4.0)));
            r3=1.0/(1.0+20.0*alf/((7.5+alf)*n*n));
            z += (z-x[n-4])*r1*r2*r3;
        } else if (i == n-1) {               最小根的初始猜测值。

            r1=(1.0+0.37*bet)/(1.67+0.28*bet);
            r2=1.0/(1.0+0.22*(n-8.0)/n);
            r3=1.0/(1.0+8.0*alf/((6.28+alf)*n*n));
            z += (z-x[n-3])*r1*r2*r3;
        } else {                             其他根的初始猜测值。
            z=3.0*x[i-1]-3.0*x[i-2]+x[i-3];
        }
        alfbet=alf+bet;
        for (its=1;its<=MAXIT;its++) {       由牛顿法进行细分。
            temp=2.0+alfbet;                 从 P0 、P1 开始递推以避免 α + β = 0
            p1=(alf-bet+temp*z)/2.0;         或-1时零作除数。
            p2=1.0;
            for (j=2;j<=n;j++) {             由递推式循环求得z点的雅可比式项式的值。
                p3=p2;
                p2=p1;
                temp=2*j+alfbet;
                a=2*j*(j+alfbet)*(temp-2.0);
                b=(temp-1.0)*(alf*alf-bet*bet+temp*(temp-2.0)*z);
                c=2.0*(j-1+alf)*(j-1+bet)*temp;
                p1=(b*p2-c*p3)/a;
            }
            pp=(n*(alf-bet-temp*z)*p1+2.0*(n+alf)*(n+bet)*p2)/(temp*(1.0-z*z));
            p1 就是所求雅可比多项式。下面将由其与 p2 的标准关系式求其导数 pp,p2 是比 p1 低一阶的多项式。
            z1=z;
            z=z1-p1/pp;                      牛顿公式。
            if (fabs(z-z1) <= EPS) break;
        }
        if (its > MAXIT) nrerror("too many iterations in gaujac");
        x[i]=z;                              存储根及权。
        w[i]=exp(gammln(alf+n)+gammln(bet+n)-gammln(n+1.0)-
            gammln(n+alfbet+1.0))*temp*pow(2.0,alfbet)/(pp*p2);
    }
}
```

勒让德多项式是雅比可多项式当$\alpha=\beta=0$时的特例,不过最好还是对它们分别编写一个程序,例如上面给出的程序 gauleg。而切比雪夫多项式对应于$\alpha=\beta=-1/2$(参见 5.8 节)。可以解析地写出其坐标点和权为

$$x_j = \cos\left(\frac{\pi(j+\frac{1}{2})}{N}\right)$$
$$w_j = \frac{\pi}{N} \tag{5.2.24}$$

4.5.2 深入讨论:递推式已知时的情况

现在讨论一下,不知道函数正交多项式零点的最佳初始估计值,但可用系数 a_j 和 b_j 求得它们的情况。正如我们已经看到的,$p_N(x)$的零点就是 N 点高斯求积分公式中的坐标值。最有效的求权方法是用(4.5.9)式,因为一般情况下,其中导数 p'、很容易由(4.5.6)式微分得到,对于某些经典多项式则可由其特殊关系式得到。注意,(4.5.9)式仅对写成为首一多项式的形式才有效。对其他整理形式,则需另有一个因子 λ_N/λ_{N-1},其中 λ_N 为 p_N 中 x^N 的系数。

除了某些已讨论过的特殊情况外,求坐标点的最好方法并**不是**像对 $P_N(x)$用牛顿法那样的求根法。一般更快的方法有 Golub-Welsch[3]算法,此算法基于 Wilf[4]的研究成果。这个算法就是,将项 xp_j 放到(4.5.6)式左边,而将 p_{j+1}项放到右边。递推关系式可写成矩阵形式

$$x\begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} = \begin{bmatrix} a_0 & 1 & & & \\ b_1 & a_1 & 1 & & \\ & \vdots & \vdots & & \\ & & b_{N-2} & a_{N-2} & 1 \\ & & & b_{N-1} & a_{N-1} \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ p_N \end{bmatrix}$$

或是

$$x\mathbf{p} = \mathbf{T}\cdot\mathbf{p} + p_N\mathbf{e}_{N-1} \tag{4.5.25}$$

其中 $\mathbf{T}$ 为三角矩阵,$\mathbf{p}$ 为 $p_0, p_1, \ldots, p_{N-1}$的列向量,$\mathbf{e}_{N-1}$为单位向量,即其第 $N-1$ 位(最后一位)为 1,其余为零。$\mathbf{T}$ 矩阵可由对角相似变换矩阵 $\mathbf{D}$ 变成对称形式

$$\mathbf{J} = \mathbf{DTD}^{-1} = \begin{bmatrix} a_0 & \sqrt{b_1} & & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & & \\ & \vdots & \vdots & & \\ & & \sqrt{b_{N-2}} & a_{N-2} & \sqrt{b_{N-1}} \\ & & & \sqrt{b_{N-1}} & a_{N-1} \end{bmatrix} \tag{4.5.26}$$

$\mathbf{J}$ 矩阵称为**雅可比矩阵**(注意不要与其他有相同名字但出现在完全不同问题中的矩阵混淆)。从(4.5.25)式可以看出,$p_N(x_j)=0$ 等价于 x_j 为 $\mathbf{T}$ 的特征值。因为相似变换后特征值不变,可见 x_j 是对称三对角矩阵 $\mathbf{J}$ 的特征值。此外,Wilf[4]还证明了,若 $\mathbf{v}_j$ 为对应于特征值 x_j 的特征向量,经归一化使得 $\mathbf{v}\cdot\mathbf{v}=1$,则

$$w_j = \mu_0 v_{j,0}^2 \tag{4.5.27}$$

其中,

$$\mu_0 = \int_a^b W(x)\,dx \tag{4.5.28}$$

上式中 $v_{j,0}$是 $\mathbf{v}$ 的第零个分量。在第 11 章中将介绍,寻找一个对称三对角矩阵的特征值和特征向量并不困难。这里给出一个程序 gaucof,可由系数 a_j 和 b_j 找到坐标点和权值。记住,若正交多项式的递推式未整理成首一形式,则通过乘以一个数 λ_j,可以很容易转化成首一多项式的形式。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::gaucof(Vec_IO_DP &a, Vec_IO_DP &b, const DP amu0, Vec_O_DP &x,
    Vec_O_DP &w)
```

由雅可比矩阵计算高斯求积分公式的坐标点和权值。输入是 a[0..n-1]和 b[0..n-1],分别为首一正交多项式集合的递推关系式系数。量 $\mu_0 \equiv \int_a^b W(x)\,dx$ 作为 amu0 的输入。按递减顺序返回坐标点 x[0..n-1],相应的权为 w[0..n-1]。数组 a 和 b 被修改。若修改程序 tqli 和 eigsrt,只计算每个特征向量的第零个分量,则执行速度可以加快。

```
{
    int i,j;

    int n=a.size();
    Mat_DP z(n,n);
    for (i=0;i<n;i++) {
        if (i != 0) b[i]=sqrt(b[i]);        设置雅可比矩阵的上对角。
        for (j=0;j<n;j++) z[i][j]=DP(i == j);
        设置单位矩阵, 由times计算特征向量.
    }
    tqli(a,b,z);
    eigsrt(a,z);                            按递减顺序排放特征值 。
    for (i=0;i<n;i++) {
        x[i]=a[i];
        w[i]=amu0*z[0][i]*z[0][i];          (4.5.27) 式 。
    }
}
```

4.5.3 深入讨论:具有非经典权的正交多项式

本节有点特殊,讨论若权函数不是上面处理的经典形式,而且也不知道用于 gaucof 中递推关系式(4.5.6)的 a_j 和 b_j 值的情况。此时,需要另外一种求 a_j 和 b_j 的方法。

Stieltjes 方法由(4.5.7)式求得 a_0,由(4.5.6)式求得 $p_1(x)$。已知 p_0 和 p_1 后,则可由(4.5.7)式求得 a_1 和 b_1,如此递推下去,但是怎么计算(4.5.7)式中的内积呢?

教科书上的方法是,将每个 $p_j(x)$明确地表示为 x 的多项式形式,然后逐项相乘求内积。当知道权函数的前 $2N$ 个矩时,这种方法似乎是可行的,此矩为

$$\mu_j = \int_a^b x^j W(x)dx \qquad j = 0, 1, \ldots, 2N-1 \tag{4.5.29}$$

然而一般来说,根据 μ_j 的值求解系数 a_j 和 b_j 的代数方程,得到的结果呈现严重的病态。即使用双精度,当 $N=12$ 时也总会丢掉全部精度。所以不能使用任何基于矩(4.5.29)的方法。

Sack 和 Donovan[5]发现,如果表示 p_j 时,不用 x 的幂作为基本函数集,而是使用另外一组正交多项式如 $\pi_j(x)$,则数值的稳定性会大大地改善。大概地讲,稳定性改善是因为当求内积积分时,多项式作为基在(a,b)区间采样比幂作为基要好,特别是当权函数类似 $W(x)$时。

假设已知修正矩值

$$\nu_j = \int_a^b \pi_j(x)W(x)dx \qquad j = 0, 1, \ldots, 2N-1 \tag{4.5.30}$$

其中 π_j 满足类似(4.5.6)式的递推关系式,

$$\begin{aligned} \pi_{-1}(x) &\equiv 0 \\ \pi_0(x) &\equiv 1 \\ \pi_{j+1}(x) &= (x-\alpha_j)\pi_j(x) - \beta_j\pi_{j-1}(x) \qquad j = 0, 1, 2, \ldots \end{aligned} \tag{4.5.31}$$

系数α_j 和 β_j 是确切已知的。Wheeler[6]找到一个 $O(N^2)$有效算法,与 Sack 和 Donovan 算法是等效的,通过一组中间过渡量求 a_j 和 b_j,

$$\sigma_{k,l} = \langle p_k | \pi_l \rangle \qquad k, l \geqslant -1 \tag{4.5.32}$$

初始化

$$\begin{aligned} \sigma_{-1,l} &= 0 & l &= 1, 2, \ldots, 2N-2 \\ \sigma_{0,l} &= \nu_l & l &= 0, 1, \ldots, 2N-1 \\ a_0 &= \alpha_0 + \frac{\nu_1}{\nu_0} \\ b_0 &= 0 \end{aligned} \tag{4.5.33}$$

然后,对于 $k = 1, 2, \ldots, N-1$,计算

$$\sigma_{k,l} = \sigma_{k-1,l+1} - (a_{k-1} - \alpha_l)\sigma_{k-1,l} - b_{k-1}\sigma_{k-2,l} + \beta_l\sigma_{k-1,l-1}$$

$$l = k, k+1, \ldots, 2N-k-1$$

$$a_k = \alpha_k - \frac{\sigma_{k-1,k}}{\sigma_{k-1,k-1}} + \frac{\sigma_{k,k+1}}{\sigma_{k,k}} \tag{4.5.34}$$

$$b_k = \frac{\sigma_{k,k}}{\sigma_{k-1,k-1}}$$

注意,如果需要,归一化因子也很容易算出

$$\langle p_0|p_0\rangle = \nu_0$$

$$\langle p_j|p_j\rangle = b_j\langle p_{j-1}|p_{j-1}\rangle \qquad j = 1, 2, \ldots \tag{4.5.35}$$

本节参考文献[7]中有此算法的推理。

Wheeler算法需要精确计算(4.5.30)式的修正矩值。实际情况中通常为闭型的形式,或可使用递推关系式。对有限区间(a,b),此算法非常有效;对无限区间,该算法并未能完全消除病态性。在这种情况下,Gautschi[8,9]提出用变换变量将无限区间转化为一个有限区间,然后再用一个合适的离散化过程计算其内积。详情可见参考文献。

下面给出由Wheeler算法产生系数a_j和b_j的程序orthog,给定系数α_j和β_j及修正矩ν_j。注意在此程序中sig[k][l] = $\sigma_{k,l-1}$。

```
#include "nr.h"

void NR::orthog(Vec_I_DP &anu, Vec_I_DP &alpha, Vec_I_DP &beta, Vec_O_DP &a,
    Vec_O_DP &b)
由 Wheeler 算法,计算权函数为 W(x)的首一正交多项式的递推关系式中的系数 a_j 和 b_j, j = 0, ..., N-1。输入数组 al-
pha[0..2*n-2]和 beta[0..2*n-2]为选定的基为正交多项式的递推关系式中的系数α_j 和β_j, j = 0, ..., 2N-2。输入
修正矩 ν_j 为 anu[0..2*n-1]。前 n 个系数返回数组 a[0..n-1]和 b[0..n-1]中。
{
    int k,l,looptmp;

    int n=a.size();
    Mat_DP sig(2*n+1,2*n+1);
    looptmp=2*n;
    for (l=2;l<looptmp;l++) sig[0][l]=0.0;          初始化,(4.5.33)式。
    looptmp++;
    for (l=1;l<looptmp;l++) sig[1][l]=anu[l-1];
    a[0]=alpha[0]+anu[1]/anu[0];
    b[0]=0.0;
    for (k=2;k<n+1;k++) {                           (4.5.34)式。
        looptmp=2*n-k+2;
        for (l=k;l<looptmp;l++) {
            sig[k][l]=sig[k-1][l+1]+(alpha[l-1]-a[k-2])*sig[k-1][l]
                -b[k-2]*sig[k-2][l]+beta[l-1]*sig[k-1][l-1];
        }
        a[k-1]=alpha[k-1]+sig[k][k+1]/sig[k][k]-sig[k-1][k]/sig[k-1][k-1];
        b[k-1]=sig[k][k]/sig[k-1][k-1];
    }
}
```

作为应用orthog的示例,考察问题[7]:如何在区间(0,1)上,产生权函数$W(x) = -\log x$的正交多项式。一个合适的π_j集是移位了的勒让德多项式

$$\pi_j = \frac{(j!)^2}{(2j)!}P_j(2x-1) \tag{4.5.36}$$

P_j前面的因子使多项式为首一的。递推关系式(4.5.31)中的系数为

$$\alpha_j = \frac{1}{2} \qquad j = 0, 1, \ldots$$

$$\beta_j = \frac{1}{4(4-j^{-2})} \qquad j = 1, 2, \ldots \tag{4.5.37}$$

而修正矩为

$$\nu_j = \begin{cases} 1 & j = 0 \\ \dfrac{(-1)^j (j!)^2}{j(j+1)(2j)!} & j \geqslant 1 \end{cases} \tag{4.5.38}$$

用这些作为输入去调用 orthog，即可在 N 很大时得到满足机器精度的多项式，然后用此权函数就可以计算高斯求积分公式了。这看起来似乎是很简单的问题，然而在 Sack 和 Donovan 之前，这其实是很困难的。

4.5.4 高斯积分推广

推广高斯求积法有多种途径。很重要的一种推广是**预设节点**，即先指定某些必须包含在坐标集中的点，那么，问题就是选择权及其余坐标点，使得积分法尽可能准确。最常见的例子是 Gauss-Radau 积分，其节点之一必须为区间的端点，即 a 或 b；而 Gauss-Lobatto 积分则必须 a 和 b 都为节点。Golub[10]为这些情况设计了与 gaucof 类似的算法。

第二种重要的推广是 Gauss-Kronrod 公式。对一般高斯求积分公式，当 N 增加时，坐标集中没有相同的点。这就意味着如果像估计积分误差那样用增大的 N 比较，不能再使用以前的函数值。Kronrod[11]提出了寻找最优序列的方法，每次都能使用前一次的所有坐标值。若从 $N = m$ 开始，新增加 n 个点，则会有 $2n + m$ 个自由参数量：n 个新坐标点、n 个权值和对应于以前固定坐标点的 m 个新的权值。所以期望能达到的最大精确度为 $2n + m - 1$。现在问题是在实际中，当要求的所有坐标点都在 (a, b) 中时，是否还能达到这个最大准确度，这一问题的答案现在还不清楚。

Kronrod 发现当 $n = m + 1$ 时，可以得到高斯－勒让德积分的最佳推广。Patterson[12]找到了如何连续计算这种推广的方法。在自动求积分程序[13]中，求一个函数的积分并能达到所指定的精度要求，通常使用的序列是 $N = 10, 21, 43, 87, \ldots$。

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.4. [1]

Stroud, A.H., and Secrest, D. 1966, *Gaussian Quadrature Formulas* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Golub, G.H., and Welsch, J.H. 1969, *Mathematics of Computation*, vol. 23, pp. 221–230 and A1–A10. [3]

Wilf, H.S. 1962, *Mathematics for the Physical Sciences* (New York: Wiley), Problem 9, p. 80. [4]

Sack, R.A., and Donovan, A.F. 1971/72, *Numerische Mathematik*, vol. 18, pp. 465–478. [5]

Wheeler, J.C. 1974, *Rocky Mountain Journal of Mathematics*, vol. 4, pp. 287–296. [6]

Gautschi, W. 1978, in *Recent Advances in Numerical Analysis*, C. de Boor and G.H. Golub, eds. (New York: Academic Press), pp. 45–72. [7]

Gautschi, W. 1981, in *E.B. Christoffel*, P.L. Butzer and F. Fehér, eds. (Basel: Birkhauser Verlag), pp. 72–147. [8]

Gautschi, W. 1990, in *Orthogonal Polynomials*, P. Nevai, ed. (Dordrecht: Kluwer Academic Publishers), pp. 181–216. [9]

Golub, G.H. 1973, *SIAM Review*, vol. 15, pp. 318–334. [10]

Kronrod, A.S. 1964, *Doklady Akademii Nauk SSSR*, vol. 154, pp. 283–286 (in Russian). [11]

Patterson, T.N.L. 1968, *Mathematics of Computation*, vol. 22, pp. 847–856 and C1–C11; 1969, *op. cit.*, vol. 23, p. 892. [12]

Piessens, R., de Doncker, E., Uberhuber, C.W., and Kahaner, D.K. 1983, *QUADPACK: A Subroutine Package for Automatic Integration* (New York: Springer-Verlag). [13]

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §3.6.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.5.

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), §§2.9–2.10.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §§4.4–4.8.

4.6 多维积分

在大于一维的区域上，多个变量的函数积分不太容易，有两个原因：第一，N 维空间函数求值的数目，将按照一维情况下的 N 次幂增长。大概地说，求一个一维积分需求30个点的函数值，则三维情况下达到同样精度水平，就需要30 000阶次的函数求值；第二，N 维积分区域需由 $N-1$ 维的边界来定义，而边界本身极其复杂，例如不一定就是凸的或是连通的。而相比之下，一维积分边界则只有两个数，即上限和下限。

碰到多维积分问题时，首先要问的问题是能否解析地将维数降低。例如，称为单变量函数 $f(t)$ 的**多重积分**可由下面的公式降为一维积分：

$$\begin{aligned}&\int_0^x dt_n \int_0^{t_n} dt_{n-1} \cdots \int_0^{t_3} dt_2 \int_0^{t_2} f(t_1)dt_1 \\ &= \frac{1}{(n-1)!}\int_0^x (x-t)^{n-1} f(t)dt\end{aligned} \tag{4.6.1}$$

或者函数有某种依赖自变量的特殊对称性质。若边界也有这种对称性，则维数可降低。例如在三维空间中，球对称函数在球形域中的积分，就可以在极坐标下降为一维积分。

对下面提出的问题有两种完全不同的解决办法。问题是：积分区域边界的形状是简单的还是复杂的？该区域内，被积函数是简单平滑的还是复杂的，或者是局部有显著峰值的？要求精度高不高，要求结果的精度是准确到百分之一还是百分之几？

若边界复杂，被积函数在很小区域内峰值不显著，且允许相对较低的精度，则可用蒙特卡罗(Monte Carlo)积分方法。这种方法编程简单，特别是对粗略的形式。仅需知道一个带有简单边界的区域，它包含了积分区域的复杂区域，外加一种方法，能判别随机点是落在积分区域的内部还是外部。蒙特卡罗方法先求随机点处的函数值，然后由这些随机值来估算积分。第7章将讨论细节及较复杂的情形。

若边界简单，函数很平滑，则用残留逼近将多维积分分成多重一维积分，或者多维高斯求积分，这样做效果好而且速度比较快。若精度要求较高，则无论如何这种逼近方法是惟一的选择，因为蒙特卡罗方法本质上渐近地收敛速度较慢。

精度要求不高时，若被积函数变化较慢，在积分区域内又很平滑，则用多重一维积分或多维高斯求积分法。若被积函数振荡或不连续，但在小区域内峰值不显著时，则用蒙特卡罗方法。

若被积函数在小区域内峰值确实显著，并且已知这些区域，则可将积分分成在几个区域内分别进行，使其在每个区域内都平滑。若这些区域不可知，那就没有其他办法了(在本书程度范围内)。不能指望积分程序会搜索出 N 维空间内对积分影响较大的未知小块(但可参考7.8节)。

读了上面的内容，读者可能想了解多重一维积分的逼近方法。下面介绍其原理。我们的考察限定在 x、y、z 空间的三维积分。二维或高于三维的情况完全类似。

第一步，确定积分区域：(i) x 的上下限，记为 x_1 和 x_2；(ii)在确定的 x 值处 y 的上下限，记为 $y_1(x)$、$y_2(x)$；(iii)在确定的 x 和 y 处 z 的上下限，记为 $z_1(x,y)$ 和 $z_2(x,y)$。换言之，找到 x_1 和 x_2 值及函数 $y_1(x)$、$y_2(x)$、$z_1(x,y)$ 和 $z_2(x,y)$ 使得

$$\begin{aligned}I &\equiv \iiint dx\,dy\,dz\,f(x,y,z) \\ &= \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x,y)}^{z_2(x,y)} dz\, f(x,y,z)\end{aligned} \tag{4.6.2}$$

例如，单位圆上的二维积分为

$$\int_{-1}^{1} dx \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} dy\, f(x,y) \tag{4.6.3}$$

现将最内层积分定义为 $G(x,y)$,

$$G(x,y) \equiv \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)dz \tag{4.6.4}$$

对 $G(x,y)$的积分记为函数 $H(x)$,

$$H(x) \equiv \int_{y_1(x)}^{y_2(x)} G(x,y)dy \tag{4.6.5}$$

最后,对 $H(x)$的积分即为所求:

$$I = \int_{x_1}^{x_2} H(x)dx \tag{4.6.6}$$

可递归地调用一些基本的一维积分程序,来实现(4.6.4)式~(4.6.6)式(如下面程序中的 qgaus):求外层积分 I 只需调用一次,求中间积分 H 则需调用多次,求内层积分 G 调用次数就更多了(参见图 4.6.1)。当前的 x 和 y 值,以及指向函数 func 的指针,通过在名字空间 NRquad3d 中定义的全局变量传递给中间的每次调用。

```
#include "nr.h"

extern DP yy1(const DP),yy2(const DP);
extern DP z1(const DP, const DP);
extern DP z2(const DP, const DP);

namespace NRquad3d {
    DP xsav,ysav;
    DP (*nrfunc)(const DP, const DP, const DP);

    DP f3(const DP z)                    x和y处被积函数(x,y,z)。
    {
        return nrfunc(xsav,ysav,z);
    }

    DP f2(const DP y)                    这是(4.6.4)式的G。
    {
        ysav=y;
        return NR::qgaus(f3,z1(xsav,y),z2(xsav,y));
    }

    DP f1(const DP x)                    这是(4.6.5)式的H。
    {
        xsav=x;
        return NR::qgaus(f2,yy1(x),yy2(x));
    }
}

DP NR::quad3d(DP func(const DP, const DP, const DP), const DP x1, const DP x2)
程序返回由用户提供的三维区域上函数 func 的积分,而三维区域由 x1,x2 的值和(4.6.2)式所定义的由用户提供的函数
yy1,yy2,z1 和 z2 确定(这里函数 y1 和 y2 写为 yy1 和 yy2 以免与某些 C 函数库中的贝塞尔函数名字混淆)。积分通过递
归地调用 qgaus 求得。
{
    NRquad3d::nrfunc=func;
    return qgaus(NRquad3d::f1,x1,x2);
}
```

用户必须提供的函数如以下范例:

```
DP func(const DP x, const DP y, const DP z);
The 3-dimensional function to be integrated.
DP yy1(const DP x);
DP yy2(const DP x);
DP z1(const DP x, const DP y);
DP z2(const DP x, const DP y);
```

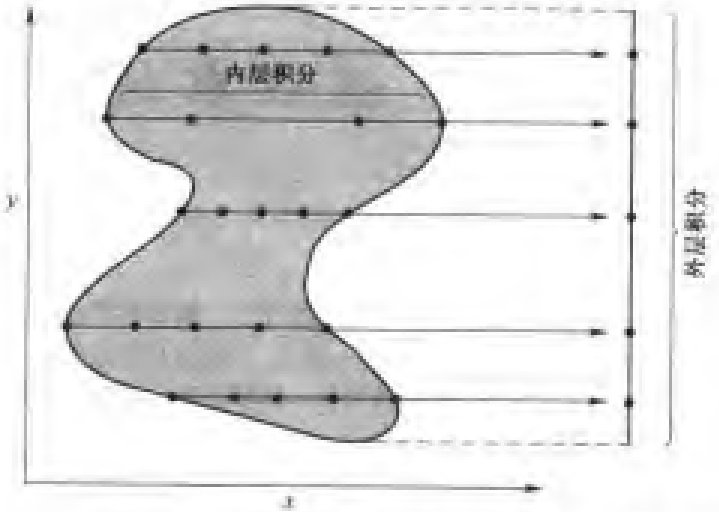


图4.6.1 在不规则区域上进行二维积分时函数值计算。外层对 y 的积分程序需有内层对 x 沿它所选定的 y 轴上的积分。然后，内层积分程序求函数在 x 处的值。这样求函数值一般比笛卡儿网点法精确

参考文献和深入阅读

Stroud, A.H. 1971, *Approximate Calculation of Multiple Integrals* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.7, p. 318.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.2.5, p. 307.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), equations 25.4.58ff.

第5章 函数求值

5.0 引言

函数求值的方法有很多，本章讨论如何选择一种合适的方法。第6章将通过给出一些特殊函数的程序，来解释并应用这些方法。本章和下一章的学习目的是一致的，不过还有些区别：本章阐述的一些最清晰明了的一般方法，并不一定被下一章一些奇特的特殊函数选用。比较这两章，我们会看到实际应用中所谓"一般"方法与"特殊"方法之间的协调关系。

本章将讨论一般方法以及如何编写程序来估计函数值，这些函数也许是"特殊"的，但还没有特殊到包括在第6章或标准程序库中。

参考文献和深入阅读

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall).

Lanczos, C. 1956, *Applied Analysis*; reprinted 1988 (New York: Dover), Chapter 7.

5.1 级数与其收敛性

解析函数可在某点 x_0 的邻域内展开成级数：

$$f(x)=\sum_{k=0}^{\infty}a_k(x-x_0)^k \tag{5.1.1}$$

用这个级数可直接估值。当然没有必要每次都计算出 $x-x_0$ 的 k 次幂，只需记下 $k-1$ 次幂再进行一次乘法。同样，系数 a 的形式通常也可利用前面的工作：$k!$或$(2k)!$项都可以用一次或两次乘法求得。

如何知道加项已足够多了呢？实际应用时，级数的高次项最好能快速减小，不然级数方法就不是最好的方法。通常的做法是，当新增加的一项的数值小于前面和的ϵ（一个小正数）倍时，就不再加项。这样做有时数学上是不太严格的（注意，可能有 $a_k=0$ 的极端情况）。

级数表示的缺点之一是，**在复平面**上远离 x_0 处遇到奇异性，此幂级数保证**不**收敛。不过这也不要紧，书上找的级数（或自己设计的）一般可知其收敛半径。然而处处收敛（数学意义上）的级数有一个严重问题，就是在任何位置收敛都不够快，不能用数值计算方法。sin 函数和第一类贝塞尔函数就是两个常见的例子：

$$\sin x=\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k+1)!}x^{2k+1} \tag{5.1.2}$$

$$J_n(x)=\left(\frac{x}{2}\right)^n\sum_{k=0}^{\infty}\frac{(-\frac{1}{4}x^2)^k}{k!(k+n)!} \tag{5.1.3}$$

这两个级数对所有 x 都收敛，但都要到 $k>>|x|$时才开始收敛，在此之前，项值一直增加。当 x 很大时，这种级数计算方法便没有实用价值。

5.1.1 加速级数收敛

加快级数收敛速度的方法很多（等价地用部分和的序列收敛）。对于像(5.1.2)式或(5.1.3)式那种通项需一直增加的级数，这些方法作用不大。而对于通项的模是递减的级数，某些加速收敛的效果是很好的。**艾特肯 δ^2 过程**是一个近似几何收敛的、对级数部分和式进行外推的简单公式。设 S_{n-1}, S_n, S_{n+1}为三个相邻的部分和，更好的估计是

$$S'_n \equiv S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n+1} - 2S_n + S_{n-1}} \tag{5.1.4}$$

使用(5.1.4)式时，对任意整数 p 可用 $n+p$ 和 $n-p$ 分别替换 $n+1$ 和 $n-1$。得到 S'_n 序列后，再次使用(5.1.4)式，依此类推下去(实际上，经过第一次迭代后已大部分收敛，以后迭代收敛已很少)。注意，(5.1.4)式必须按所给式计算。代数上，有一种等价的对舍入误差更敏感的形式。

对**交错级数**(指和式中项的符号交替变化)，欧拉(Euler)变换是强大的工具。通常它适宜计算少量 $n-1$ 项的和，然后对从第 n 项开始级数的其余部分作变换，公式(n 为偶数)为

$$\sum_{s=0}^{\infty}(-1)^s u_s = u_0 - u_1 + u_2 \ldots - u_{n-1} + \sum_{s=0}^{\infty} \frac{(-1)^s}{2^{s+1}}[\Delta^s u_n] \tag{5.1.5}$$

其中 Δ 是**前向差分算子**，例如

$$\begin{aligned} \Delta u_n &\equiv u_{n+1} - u_n \\ \Delta^2 u_n &\equiv u_{n+2} - 2u_{n+1} + u_n \\ \Delta^3 u_n &\equiv u_{n+3} - 3u_{n+2} + 3u_{n+1} - u_n \qquad \text{等} \end{aligned} \tag{5.1.6}$$

当然，不必算出(5.1.5)式右边无穷项之和，可只计算前 p 项和，用(5.1.6)式从 u_n 开始得到前 p 个差分。

欧拉变换不仅可用于交错级数，有时某个形式上发散的级数，仅根据前几项就可得到精确的求和结果。它也被大量用于渐近级数求和。比较明智的方法是，只计算到该项的量值不再增加为止，同时进行一些独立的数值检验工作，确保得到的结果是有意义的。

van Wijngaarden[1]发现欧拉变换的一个精巧的实现方式，即每次按顺序合并原交错级数的一项。每次合并时，**或是**将 p 增1，即再做一次高次差分(5.1.6)；或是**例行地**将 n 加1，此时不必在原 n 值上再重做一遍所有的差分运算。根据使级数收敛最快来决定是增加 n 还是增加 p。van Wijngaarden 方法仅需一个向量来存放部分差分。算法如下：

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::eulsum(DP &sum, const DP term, const int jterm, Vec_IO_DP &wksp)
和式中合并第 jterm 项，其值为 term。sum 是输入即先前计算出的部分和，并是新的部分和的输出。首次调用此程序，令
jterm = 0，输入级数首项。第二次调用时，term 置为级数第二项，其符号与首次调用时相反，令 jterm = 1，依次类推。wksp
是调用程序所提供的工作空间的数组，其维数至少要和需合并项的最大数一样大。
{
    int j;
    static int nterm;
    DP tmp,dum;

    if (jterm == 0) {                          初始化:
        nterm=1;                               wksp中存放差值数。
        sum=0.5*(wksp[0]=term);                返回初次估计。
    } else {
        if (nterm+1 > wksp.size()) nrerror("wksp too small in euler");
        tmp=wksp[0];
        wksp[0]=term;
        for (j=1;j<nterm;j++) {                用van wijingaarden算法更新当前存放值。
            dum=wksp[j];
            wksp[j]=0.5*(wksp[j-1]+tmp);
            tmp=dum;
        }
        wksp[nterm]=0.5*(wksp[nterm-1]+tmp);
        if (fabs(wksp[nterm]) <= fabs(wksp[nterm-1]))      增加p,
            sum += (0.5*wksp[nterm++]);                    表加长。
        else                                               增加n,
            sum += wksp[nterm];                            表不加长。
    }
}
```

这种欧拉技巧很有效,但不能直接用于正项级数。有时将正项级数转换成交错级数,再用欧拉变换。为此,van Wijngaarden 找到一个转换公式[1]:

$$\sum_{r=1}^{\infty} v_r = \sum_{r=1}^{\infty} (-1)^{r-1} w_r \tag{5.1.7}$$

其中,

$$w_r \equiv v_r + 2v_{2r} + 4v_{4r} + 8v_{8r} + \cdots \tag{5.1.8}$$

(5.1.7)式和(5.1.8)式用二维和代替一维和,(5.1.7)式中每一项是(5.1.8)式的一个无穷和。这种方法能节省工作量,看上去比较奇怪。这是由于(5.1.8)式下标指数按 2 的幂增长很快,仅需很少几项就能收敛得很精确,只要对"随机"数 r 有效地算出 v_r。对于 r 这种标准的"更新"技巧,上述由(5.1.1)式导出的方法已不能再使用。

事实上,欧拉变换是一个更为一般的级数变换的特例。设某已知函数$g(z)$的级数

$$g(z) = \sum_{n=0}^{\infty} b_n z^n \tag{5.1.9}$$

现在对另一未知级数求和:

$$f(z) = \sum_{n=0}^{\infty} c_n b_n z^n \tag{5.1.10}$$

不难看出(参见本节参考文献[2])(5.1.10)式可写为

$$f(z) = \sum_{n=0}^{\infty} [\Delta^{(n)} c_0] \frac{g^{(n)}}{n!} z^n \tag{5.1.11}$$

通常它收敛要快得多。其中的 $\Delta^{(n)} c_0$ 是 n 阶有限差分算子((5.1.6)式),$\Delta^{(0)} c_0 \equiv c_0$,$g^{(n)}$ 则为$g(z)$的 n 阶导数。可得到通常的欧拉变换((5.1.5)式且 $n=0$),例如将

$$g(z) = \frac{1}{1+z} = 1 - z + z^2 - z^3 + \cdots \tag{5.1.12}$$

代入(5.1.11)式,并令 $z=1$。

有时虽然计算效果不好,也还得用级数来计算函数。例如,可以利用所得到的数值逼近为函数的近似形式,它在后面将要用到(参见 5.8 节)。在计算缓慢收敛级数的许多项之和时,应注意舍入误差。浮点数表示时,按顺序求一列数之和,从最小项开始比从最大项开始的结果更为准确。将项两两合成一对,再两两合成一组的效果更好,这样在加法中就包含了比较大小的操作。

参考文献和深入阅读

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), Chapter 13 [van Wijngaarden's transformations]. [1]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 3.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §3.6.

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), §2.3. [2]

5.2 连分式求值

通常,连分式可作为科学计算中求函数值的有效工具。连分式如下面的形式:

$$f(x) = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \cfrac{a_5}{b_5 + \cdots}}}}} \tag{5.2.1}$$

印刷时写成

$$f(x) = b_0 + \frac{a_1}{b_1 +}\ \frac{a_2}{b_2 +}\ \frac{a_3}{b_3 +}\ \frac{a_4}{b_4 +}\ \frac{a_5}{b_5 +}\cdots \tag{5.2.2}$$

(5.2.1)式和(5.2.2)式中 a 和 b 可以是 x 的函数，一般为线性的，或最差的为 x 的二次单项式(如 x 或 x^2 的常数倍)。例如，正切函数连分式表示为

$$\tan x = \frac{x}{1 -}\ \frac{x^2}{3 -}\ \frac{x^2}{5 -}\ \frac{x^2}{7 -}\cdots \tag{5.2.3}$$

连分式通常比级数展开收敛要快得多，并且收敛区域更大(但不总是包括级数的收敛域)。有时级数收敛最差的地方却连分式收敛最好，虽然不总是这样。Blanch[1]给出了有关连分式最有用的收敛测试的详细介绍。

有一些标准技术，包括重要的**商－差算法**在内，都不外乎是连分式逼近、幂级数逼近和有理函数逼近。有关这方面的内容参见 Acton[2]，更详细的讨论见 Fike[3]。

计算连分式何时能停止呢？跟级数不同，不能从左到右估计(5.2.1)式，当变化很小时就停下来。而写成(5.2.1)式的形式后，计算连分式的惟一办法是从右到左，首先估计从哪里开始。然而，正确方法不是如此。

正确的方法是，应用一个联系连分式和有理数逼近的结果，并给出一个从左到右估计(5.2.1)式或(5.2.2)式的方法。设 f_n 表示由 a_n 和 b_n 估计(5.2.2)式的结果，那么

$$f_n = \frac{A_n}{B_n} \tag{5.2.4}$$

其中 A_n 和 B_n 由下列递推式给出：

$$\begin{aligned} A_{-1} &\equiv 1 \qquad B_{-1} \equiv 0 \\ A_0 &\equiv b_0 \qquad B_0 \equiv 1 \\ A_j = b_j A_{j-1} + a_j A_{j-2} &\qquad B_j = b_j B_{j-1} + a_j B_{j-2} \qquad j = 1, 2, \ldots, n \end{aligned} \tag{5.2.5}$$

此方法是在 1655 年由 J. Wallis 发明，并在"Arithmetica Infinitorum"[4]一书中讨论。读者能容易地证明它。

实际上，这个算法并不是很好。由(5.2.5)式经常会递推出一些非常大或非常小的部分分子 A_j 和分母 B_j，这样用浮点数表示时，可能有上溢或下溢的危险。然而递推式(5.2.5)对 A 和 B 是线性的，因此每次迭代时，都可重新整定当前存放的递推结果，例如，可将 A_j，B_j，A_{j-1} 和 B_{j-1} 除以 B_j，使 $A_j = f_j$。还可供判断是否还需递推下去：只需看最新迭代出的 f_j 和 f_{j-1} 是不是如所期望那样接近(若有时 B_j 恰好为零，则跳出本次循环，重新归一化。更好的优化方法是，仅当溢出即将发生时才重新整定，以省去不必要的除法，但这样会使算法复杂化)。

有两种新的求连分式的方法。**Steed 方法**不显式使用 A_j 和 B_j，而只用了**比值** $D_j = B_{j-1}/B_j$。只要递推地用下式计算 D_j 和 $\Delta f_j = f_j - f_{j-1}$：

$$D_j = 1/(b_j + a_j D_{j-1}) \tag{5.2.6}$$

$$\Delta f_j = (b_j D_j - 1)\Delta f_{j-1} \tag{5.2.7}$$

Steed 方法(参见本节参考文献[5])避免了重新调整当前结果。但对某些连分数可能会碰到这种情况，即(5.2.6)式中的分母几乎为零，所以 D_j 和 Δf_j 很大。下一个 Δf_{j+1} 虽会抵消这种变化，但在计算 f_j 的数值和时使精度降低。此时程序运行不尽如人意，所以一般 Steed 方法只在预先知道分母不很小的情况下才推荐使用。在 6.7 节 bessik 程序中，此方法将用于某个特殊目的。

求连分式最一般的方法大概要算**改进的 Lentz 方法**[6]。利用下面两个比值就可以不必对当前结果整定：

$$C_j = A_j/A_{j-1}, \qquad D_j = B_{j-1}/B_j \tag{5.2.8}$$

再由下式计算 f_j：

$$f_j = f_{j-1} C_j D_j \tag{5.2.9}$$

由(5.2.5)式可知，显然这两个比值满足递推关系式

$$D_j = 1/(b_j + a_j D_{j-1}), \qquad C_j = b_j + a_j/C_{j-1} \tag{5.2.10}$$

此算法中，D_j 表达式的分母或 C_j 本身有趋近零的危险。这样(5.2.10)式就没有用了。对此，Thompson 和 Barnett[5]提出修正 Lentz 算法来解决这个问题：它将为零的项用一个很小的量代替，如 10^{-30}。只需用这种方法试着将算法运行一个循环，就会发现 f_{j+1} 可精确地计算出来。

下面详细列出改进的 Lentz 算法：

- 令 $f_0 = b_0$；若 $b_0 = 0$，则令 $f_0 = tiny$
- 令 $C_0 = f_0$
- 令 $D_0 = 0$
- 对 $j = 1, 2, \ldots$

  令 $D_j = b_j + a_j D_{j-1}$

  若 $D_j = 0$，令 $D_j = tiny$

  令 $C_j = b_j + a_j / C_{j-1}$

  若 $C_j = 0$，令 $C_j = tiny$

  令 $D_j = 1/D_j$

  令 $\Delta j = C_j D_j$

  令 $f_j = f_{j-1} \Delta j$

  若 $|\Delta_j - 1| < eps$ 则停止

其中 eps 为浮点精度值，可以是 10^{-7} 或 10^{-15}。参数 $tiny$ 必须小于 $eps|b_j|$，如 10^{-30}。

上面算法是假定 $|f_j - f_{j-1}|$ 足够小时，可以结束连分式求值。通常情况下正是如此，但不能保证总是这样。Jones[7]为判断求各种连分数值的结束条件，证明了一系列定理。

Lentz 算法目前尚无误差扩散的一般分析，不过经验表明此算法至少不比其他算法差。

5.2.1 连分式处理

利用连分式的几个重要性质可将连分数写成另外的形式，以加速数值运算过程。一个**等价变换**是

$$a_n \to \lambda a_n, \quad b_n \to \lambda b_n, \quad a_{n+1} \to \lambda a_{n+1} \tag{5.2.11}$$

它不会改变连分式的值。选择合适的修整因子 λ 可以简化 a 和 b 的形式。当然还可以进行连续的等价变换，连分式的相邻连续项可用不同的 λ 值。

连分式的奇部分和偶部分也是连分式，其分别连续收敛到 f_{2n} 和 f_{2n+1}。这里所利用的，主要是因为它们比原来的连分式收敛快两倍。如果它们各部分中的项并不比原分式复杂多少，则可以节省很多计算。(5.2.2)式偶部分公式为

$$f_{\text{even}} = d_0 + \frac{c_1}{d_1 +} \frac{c_2}{d_2 +} \cdots \tag{5.2.12}$$

其中根据当前变量

$$\begin{aligned} \alpha_1 &= \frac{a_1}{b_1} \\ \alpha_n &= \frac{a_n}{b_n b_{n-1}}, \qquad n \geqslant 2 \end{aligned} \tag{5.2.13}$$

我们有

$$\begin{gathered} d_0 = b_0, \quad c_1 = \alpha_1, \quad d_1 = 1 + \alpha_2 \\ c_n = -\alpha_{2n-1}\alpha_{2n-2}, \quad d_n = 1 + \alpha_{2n-1} + \alpha_{2n}, \qquad n \geqslant 2 \end{gathered} \tag{5.2.14}$$

在 Blanch[1]的著作中可以找到奇部分的类似公式。通常联合利用(5.2.11)式和(5.2.14)式得到数值计算的最佳格式。

下一章将要经常运用连分式。

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §3.10.

Blanch, G. 1964, *SIAM Review*, vol. 6, pp. 383–421. [1]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 11. [2]

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 1.

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall), §§8.2, 10.4, and 10.5. [3]

Wallis, J. 1695, in *Opera Mathematica*, vol. 1, p. 355, Oxoniae e Theatro Shedoniano. Reprinted by Georg Olms Verlag, Hildeshein, New York (1972). [4]

Thompson, I.J., and Barnett, A.R. 1986, *Journal of Computational Physics*, vol. 64, pp. 490–509. [5]

Lentz, W.J. 1976, *Applied Optics*, vol. 15, pp. 668–671. [6]

Jones, W.B. 1973, in *Padé Approximants and Their Applications*, P.R. Graves-Morris, ed. (London: Academic Press), p. 125. [7]

5.3 多项式和有理函数

N 次多项式在数值上，可用一个存储系数的数组 c[j], j = 0, . . . , N 来表示。其中 c[0] 为多项式中常数项，c[N]为 x^N 系数，当然其他习惯表示法也是可以的。多项式有两种处理方法，一是**数值**处理(如求值)，此时应给定自变量的数值，另一种方法是**代数**处理，只是以某种方式变换系数数组，而不需要某个特定的自变量。首先讨论数值处理方法。

首先，假定读者已知道永远不能用如下方式求多项式的值：

```
p=c[0]+c[1]*x+c[2]*x*x+c[3]*x*x*x+c[4]*x*x*x*x;
```

或者(更糟糕)，

```
p=c[0]+c[1]*x+c[2]*pow(x,2.0)+c[3]*pow(x,3.0)+c[4]*pow(x,4.0);
```

计算机革命已经来临，这种错误不应该再犯！至于是写成

```
p=c[0]+x*(c[1]+x*(c[2]+x*(c[3]+x*c[4])));
```

还是写成

```
p=(((c[4]*x+c[3])*x+c[2])*x+c[1])*x+c[0];
```

就只是习惯问题了。

若系数 c[0..n]数量很多，程序应为

```
p=c[n];
for(j=n-1;j>=0;j--) p=p*x+c[j];
```

或

```
p=c[j=n];
while (j>0) p=p*x+c[--j];
```

另外，还有一个同时计算多项式 $P(x)$ 及其导数 $dP(x)/dx$ 之值的方法：

```
p=c[n];
dp=0.0;
for(j=n-1;j>=0;j--) {dp=dp*x+p; p=p*x+c[j];}
```

或

```
p=c[j=n];
dp=0.0;
while (j>0) {dp=dp*x+p; p=p*x+c[--j];}
```

程序结果是计算多项式值 p 及其导数值 dp。

以上方法基本上为**综合除法**[1,2],可以推广到一般地同时求多项式及其 nd 阶导数值。

```
#include "nr.h"

void NR::ddpoly(Vec_I_DP &c, const DP x, Vec_O_DP &pd)
给定 nc 次多项式以数组 c[0..nc]表示的 nc+1 个系数,c[0]为常数项,x 值给定,程序返回输出数组 pd[0..nd]。程序在
pd[0]中返回多项式在 x 处的值,pd[1..nd]中返回多项式在 x 处的第一个 nd 阶导数。
{
    int nnd,j,i;
    DP cnst=1.0;

    int nc=c.size()-1;
    int nd=pd.size()-1;
    pd[0]=c[nc];
    for (j=1;j<nd+1;j++) pd[j]=0.0;
    for (i=nc-1;i>=0;i--) {
        nnd=(nd < (nc-i) ? nd : nc-i);
        for (j=nnd;j>0;j--)
            pd[j]=pd[j]*x+pd[j-1];
        pd[0]=pd[0]*x+c[i];
    }
    for (i=2;i<nd+1;i++) {      一阶导数后, 出现阶乘因子。
        cnst *= i;
        pd[i] *= cnst;
    }
}
```

读者可能奇怪并想知道,$n>3$ 次多项式却用少于 n 次的乘法就能求得结果,为此,需先算出一些辅助系数,并再做些加法。例如多项式

$$P(x)=a_0+a_1x+a_2x^2+a_3x^3+a_4x^4 \tag{5.3.1}$$

可由 3 次乘法 5 次加法算出,其中 $a_4>0$,

$$P(x)=[(Ax+B)^2+Ax+C][(Ax+B)^2+D]+E \tag{5.3.2}$$

其中 A,B,C,D,E 由下式给出:

$$\begin{aligned}
A&=(a_4)^{1/4}\\
B&=\frac{a_3-A^3}{4A^3}\\
D&=3B^2+8B^3+\frac{a_1A-2a_2B}{A^2}\\
C&=\frac{a_2}{A^2}-2B-6B^2-D\\
E&=a_0-B^4-B^2(C+D)-CD
\end{aligned} \tag{5.3.3}$$

五次多项式可由 4 次乘法和 5 次加法求出,六次多项式可由 4 次乘法和 7 次加法求出。若感兴趣的话,可参见本节参考文献[3~5]。这个主题虽说不很实用,但也是很有意思的,如同 2.11 节中讨论的快速矩阵乘法。

现在讨论代数处理。一个 $n-1$ 次多项式(范围在[0..n-1]的数组)乘一个多项式因子 $x-a$,可用下面的一段程序:

```
c[n]=c[n-1];
for (j=n-1;j>=1;j--) c[j]=c[j-1]-c[j]*a;
c[0] *= (-a);
```

同样地,n 次多项式除以多项式因子 $x-a$(又是综合除法)可用下述程序:

```
rem=c[n];
c[n]=0.0;
for(i=n-1;i>=0;i--) {
    swap=c[i];
    c[i]=rem;
    rem=swap+rem*a;
}
```

最后得到一个新的多项式和一个数值余数 rem。

两个一般多项式相乘，每次从每个多项式取一个系数相乘，然后直接求和。两个一般多项式相除，使用不同优化算法影响却很大。注意，下面的程序基于本节参考文献[3]中的算法。

```
#include "nr.h"

void NR::poldiv(Vec_I_DP &u, Vec_I_DP &v, Vec_O_DP &q, Vec_O_DP &r)
给定 n 次多项式的 n+1 个系数 u[0..n]，以及另一个 nv 次多项式的 nv+1 个系数 v[0..nv]，用多项式 v 去除多项式
u("u"/"v")，商的多项式系数返回到 q[0..n]中，余数多项式系数返回到 r[0..n]中。数组元素 r[nv..n]和 q[n-nv+1..n]
返回零值。
{
    int k,j;

    int n=u.size()-1;
    int nv=v.size()-1;
    for (j=0;j<=n;j++) {
        r[j]=u[j];
        q[j]=0.0;
    }
    for (k=n-nv;k>=0;k--) {
        q[k]=r[nv+k]/v[nv];
        for (j=nv+k-1;j>=k;j--) r[j] -= q[k]*v[j-k];
    }
    for (j=nv;j<=n;j++) r[j]=0.0;
}
```

5.3.1 有理函数

求一个如下所示的有理函数之值：

$$R(x)=\frac{P_\mu(x)}{Q_\nu(x)}=\frac{p_0+p_1x+\cdots+p_\mu x^\mu}{q_0+q_1x+\cdots+q_\nu x^\nu} \tag{5.3.4}$$

显然，只需分别求两个多项式值后再相除。通常为方便起见，选择 $q_0=1$，只需分母分子上下同除以 q_0 即可，此外可将两组系数存入一个数组，调用下面求值的标准函数：

```
#include "nr.h"

DP NR::ratval(const DP x, Vec_I_DP &cof, const int mm, const int kk)
给定 mm,kk 和 cof[0..mm+kk]，计算并返回有理函数(cof[0]+cof[1]x+…+cof[mm]x^mm)/(1+cof[mm+1]x+…+cof
[mm+kk]x^kk)。
{
    int j;
    DP sumd,sumn;

    for (sumn=cof[mm],j=mm-1;j>=0;j--) sumn=sumn*x+cof[j];
    for (sumd=0.0,j=mm+kk;j>mm;j--) sumd=(sumd+cof[j])*x;
    return sumn/(1.0+sumd);
}
```

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 183, 190. [1]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), pp. 361–363. [2]

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.6. [3]

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 4.

Winograd, S. 1970, *Communications on Pure and Applied Mathematics*, vol. 23, pp. 165–179. [4]

Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley). [5]

5.4 复数运算

由于C++中有一个内建的复数类,通常读者可以用编译器和类库做一些复数运算。一般情况下是这样,但不总是如此。如果一个程序里只有少量的复数运算,也可以考虑自己编写这些代码。或者我们有时也会遇到:虽然复数操作数和结果都可很好地表示,但运算过程中会出现上溢和下溢之类的问题。这种情况之所以发生,大概是因为软件公司认为实现复数运算是微不足道的事情,从而没有给以特别的注意。

事实上,复数运算并不是件小事。加和减自然很容易做,只需对运算数的实部和虚部分别进行运算。乘法也不难,只要做4次乘法、1次加法和1次减法:

$$(a+ib)(c+id)=(ac-bd)+i(bc+ad) \tag{5.4.1}$$

(符号 i 前的加法不能算,它只是显式地分开实部和虚部。)但有时如下乘法更快:

$$(a+ib)(c+id)=(ac-bd)+i[(a+b)(c+d)-ac-bd] \tag{5.4.2}$$

它只有3次乘法(ac,bd,$(a+b)(c+d)$),加上2次加法和3次减法。总的操作次数增加了2次,但是许多机器上乘法是很慢的。

的确,有时候(5.4.1)式和(5.4.2)式的最后结果是可表示的,而中间过程却发生溢出。这种情况仅会在最后结果勉强可表示时才会产生。求复数模时,不要按下式计算:

$$|a+ib|=\sqrt{a^2+b^2} \tag{5.4.3}$$

当 a 或 b 大到最大可表示数的平方根时(如 10^{19},相当于 10^{38} 的平方根),中间结果会溢出。正确的方法是

$$|a+ib|=\begin{cases}|a|\sqrt{1+(b/a)^2} & |a|\geqslant|b|\\ |b|\sqrt{1+(a/b)^2} & |a|<|b|\end{cases} \tag{5.4.4}$$

复数进行除法时,应用类似的方法以避免上溢、下溢或失去精度,

$$\frac{a+ib}{c+id}=\begin{cases}\dfrac{[a+b(d/c)]+i[b-a(d/c)]}{c+d(d/c)} & |c|\geqslant|d|\\ \dfrac{[a(c/d)+b]+i[b(c/d)-a]}{c(c/d)+d} & |c|<|d|\end{cases} \tag{5.4.5}$$

当然其中的子表达式,如 c/d 或 d/c,只需要计算一次。

复数平方根就更复杂了,因为除了必须监视中间结果外,还得讨论选择求得的结果(发生在负实轴上)。求 $c+id$ 的平方根,先计算

$$w\equiv\begin{cases}0 & c=d=0\\ \sqrt{|c|}\sqrt{\dfrac{1+\sqrt{1+(d/c)^2}}{2}} & |c|\geqslant|d|\\ \sqrt{|d|}\sqrt{\dfrac{|c/d|+\sqrt{1+(c/d)^2}}{2}} & |c|<|d|\end{cases} \tag{5.4.6}$$

求得结果为

$$\sqrt{c+id}=\begin{cases}0 & w=0\\ w+i\left(\dfrac{d}{2w}\right) & w\neq0,\ c\geqslant0\\ \dfrac{|d|}{2w}+iw & w\neq0,\ c<0,\ d\geqslant0\\ \dfrac{|d|}{2w}-iw & w\neq0,\ c<0,\ d<0\end{cases} \tag{5.4.7}$$

参考文献和深入阅读

Midy, P., and Yakovlev, Y. 1991, *Mathematics and Computers in Simulation*, vol. 33, pp. 33–49.

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley) [see solutions to exercises 4.2.1.16 and 4.6.4.41].

5.5　递推关系及 Clenshaw 递推公式

许多有用的函数满足递推关系，如

$$(n+1)P_{n+1}(x)=(2n+1)xP_n(x)-nP_{n-1}(x) \tag{5.5.1}$$

$$J_{n+1}(x)=\frac{2n}{x}J_n(x)-J_{n-1}(x) \tag{5.5.2}$$

$$nE_{n+1}(x)=e^{-x}-xE_n(x) \tag{5.5.3}$$

$$\cos n\theta=2\cos\theta\cos(n-1)\theta-\cos(n-2)\theta \tag{5.5.4}$$

$$\sin n\theta=2\cos\theta\sin(n-1)\theta-\sin(n-2)\theta \tag{5.5.5}$$

其中前三个函数分别为勒让德多项式、第一类贝塞尔函数和指数积分(表示方法参见本节参考文献[1])。当从两个相邻 n 项的值往大或往小的方向扩展计算序列项时，这些关系式很有用。

关于三角函数(5.5.4)式和(5.5.5)式多介绍一下。如果说程序运行的大部分时间总是在计算三角函数，那一定是什么位置出了问题。对于自变量为线性序列 $\theta=\theta_0+n\delta$，$n=0,1,2,\ldots$ 的三角函数，使用下面的递推关系很有效：

$$\begin{aligned}\cos(\theta+\delta)&=\cos\theta-[\alpha\cos\theta+\beta\sin\theta]\\ \sin(\theta+\delta)&=\sin\theta-[\alpha\sin\theta-\beta\cos\theta]\end{aligned} \tag{5.5.6}$$

其中 α 和 β 是预先计算的系数

$$\alpha\equiv 2\sin^2\left(\frac{\delta}{2}\right)\qquad \beta\equiv\sin\delta \tag{5.5.7}$$

不用通常方法(等价地)求角度而采用这种方法的原因是，当增量 δ 很小时，α 和 β 仍在起作用。同样，(5.5.6)式中加法应按方括号中的顺序求和。在第 12 章中处理傅里叶变换时，将反复利用(5.5.6)式。

有时，另外一个技巧也很有用，就是注意到 $\sin\theta$ 和 $\cos\theta$ 都可以通过 tan 计算：

$$t\equiv\tan\left(\frac{\theta}{2}\right)\qquad \cos\theta=\frac{1-t^2}{1+t^2}\qquad \sin\theta=\frac{2t}{1+t^2} \tag{5.5.8}$$

当求 $\sin\theta$ 和 $\cos\theta$ 时都必须求一次 tan，另外还有 2 次乘法、2 次除法及 2 次加法。当在机器上求三角函数比较慢时，这样做能节省计算。但当 $\theta\to\pm\pi$ 时，要求特殊处理。另外也应注意，现在许多机器上求三角函数也很快，只有上机试一试才能知道(5.5.8)式是否更快些。

5.5.1　递推式的稳定性

在递推方向上(n 增加或减小)，递推式对于舍入误差并不总是**稳定**的。一个二项线性递推关系式

$$y_{n+1}+a_ny_n+b_ny_{n-1}=0,\qquad n=1,2,\ldots \tag{5.5.9}$$

有两组线性独立的根 f_n 和 g_n。其中只有一组根 f_n 的序列才是我们要求的。另一组 g_n 可能沿指定方向是指数增长，或指数下降，或指数中性(如按某种幂规律增长或下降)。若为指数增长，则在指定方向上此递推式没有实际用处。例如(5.5.2)式在 n 增加方向上，当 $x<n$ 时就是这种情况。故在 n 很大时，不能用(5.5.2)式前向递推出贝塞尔函数。

严格地说，若

$$f_n/g_n\to 0\quad 当\quad n\to\infty \tag{5.5.10}$$

则 f_n 称为递推关系式(5.5.9)的**最小解**。非最小解如 g_n 称为**强解**。最小解若存在，则是惟一的。强解则不然，例如可以在 g_n 上增加 f_n 的任意倍数。利用前向递推可以求任意强解值，但不是最小解(遗憾的是最小解恰恰就是我们要求的解)。

Abramowitz 和 Stegun[1]给出了一个在 n 增大或减小方向上都稳定的递推关系式列表。当然不能包括所有可能的公式。给定某个函数 $f_n(x)$ 的递推关系，读者可以自己检验一下：在感兴趣范围内取一个固定 x，不要从 $f_j(x)$ 和 $f_{j+1}(x)$ 的实际值开始递推，而是对它们分别赋 1 和 0 值，然后再分别赋 0 和 1 值。对此两种不

同的初始情况,在递推方向上(从 j 开始增大或减小),做出递归序列 10 至 20 项。注意,两个序列对应项之差,若差值小于允许值(如绝对值小于 10),则递推是稳定的。若增长缓慢,递推可能轻微不稳定但可以忍受。若快速增长,则存在指数增长的解。如果已知所求函数是与增长的解对应,则可保留递推公式,例如 n 增长时的贝塞尔函数,参见 6.5 节。如果不知道这个解是所求函数对应的解,此时应舍弃递推公式。应该注意,为计算两个起始函数 $f_j(x)$ 和 $f_{j+1}(x)$ 而寻找数值方法之前,应先进行检验工作:稳定性是递推关系本身的属性,与初始值无关。

还有一种很有启发性的检验稳定性的方法,就是把具有常系数的线性递推关系代替原有的递推关系,例如(5.5.2)式可写成

$$y_{n+1} - 2\gamma y_n + y_{n-1} = 0 \tag{5.5.11}$$

其中 $\gamma \equiv \frac{n}{x}$ 视为常数。设解的形式为 $y_n = a^n$ 来求解递推关系式,代入上式得到

$$a^2 - 2\gamma a + 1 = 0 \qquad 或 \qquad a = \gamma \pm \sqrt{\gamma^2 - 1} \tag{5.5.12}$$

对所有解 a,当 $|a| \leqslant 1$ 时,递推是稳定的。此式成立(可验证)应有 $|\gamma| \leqslant 1$ 或 $n \leqslant x$。所以当 n 很大时,不能用(5.5.2)式从 $J_0(x)$ 和 $J_1(x)$ 起始来计算 $J_n(x)$。

在这个问题上,若能有某些实际的定理做保证,就可能会更好些(虽然我们一直采取一种试探性的方法)。下面两个定理是由 Perron[2]给出的。

定理 A (5.5.9)式中若 $a_n \sim an^\alpha$, $b_n \sim bn^\beta$(当 $n \to \infty$ 且 $\beta < 2\alpha$ 时),则

$$g_{n+1}/g_n \sim -an^\alpha, \qquad f_{n+1}/f_n \sim -(b/a)n^{\beta-\alpha} \tag{5.5.13}$$

f_n 就是(5.5.9)式的最小解。

定理 B 与定理 A 的条件相同,只是 $\beta = 2\alpha$,考察特征多项式

$$t^2 + at + b = 0 \tag{5.5.14}$$

若(5.5.14)式的根 t_1 和 t_2 数值不等,不妨设 $|t_1| > |t_2|$,则

$$g_{n+1}/g_n \sim t_1 n^\alpha, \qquad f_{n+1}/f_n \sim t_2 n^\alpha \tag{5.5.15}$$

f_n 也是(5.5.9)式的最小解。若不具备上述两定理的条件,则不存在最小解(关于递推稳定性的更详细讨论,参见本节参考文献[3])。

若求得的解就是最小解,那么正如我们所知的,事情总是有好有坏的:在某一方向上递推关系严重不稳定,则在其反方向上,(不希望的)解总是收敛非常快。这意味着可以用连续项 f_j 和 f_{j+1} 的任意种子值开始,(在稳定方向上递推足够多步后)收敛到所求的函数与一个未知的归一化因子之积。若还存在其他对序列进行归一化的方法(如通过 f_n 的求和公式),则它确实是求函数值的一种实用方法。这种方法称为 **Miller 算法**。通常(5.5.2)式的应用示例[1,4]就是用这种方法,其归一化公式为

$$1 = J_0(x) + 2J_2(x) + 2J_4(x) + 2J_6(x) + \cdots \tag{5.5.16}$$

正好三项递推式和连分式之间有一个重要的关系。(5.5.9)式可重写成

$$\frac{y_n}{y_{n-1}} = -\frac{b_n}{a_n + y_{n+1}/y_n} \tag{5.5.17}$$

从 n 开始对上式反复迭代,得到

$$\frac{y_n}{y_{n-1}} = -\frac{b_n}{a_n -}\,\frac{b_{n+1}}{a_{n+1} -}\cdots \tag{5.5.18}$$

由 **Pincherle 定理**[2]可知,当且仅当(5.5.9)式存在最小解 f_n 时,(5.5.18)式收敛,此时收敛到 f_n/f_{n-1}。通常对于 $n=1$ 时的情况,还需结合一些其他方法求解 f_0,这些结果都是下一章计算各种特殊函数值的各种实用方法的基础。

5.5.2 Clenshaw 递推公式

Clenshaw 递推公式[5]是计算求和式的一种优秀而有效的方法,该和式是系数乘上遵循递推公式的函数之积,例如

$$f(\theta)=\sum_{k=0}^{N}c_k\cos k\theta \qquad 或 \qquad f(x)=\sum_{k=0}^{N}c_kP_k(x)$$

其原理如下：假设求和为

$$f(x)=\sum_{k=0}^{N}c_kF_k(x) \tag{5.5.19}$$

且 F_k 满足递推关系

$$F_{n+1}(x)=\alpha(n,x)F_n(x)+\beta(n,x)F_{n-1}(x) \tag{5.5.20}$$

$\alpha(n,x)$和 $\beta(n,x)$为函数。现由以下递推式定义量 $y_k(k=N,N-1,\ldots,1)$：

$$\begin{aligned}&y_{N+2}=y_{N+1}=0\\&y_k=\alpha(k,x)y_{k+1}+\beta(k+1,x)y_{k+2}+c_k \quad (k=N,N-1,\ldots,1)\end{aligned} \tag{5.5.21}$$

为求 c_k 解(5.5.21)式，然后显式整理和式(5.5.19)，得到如下形式(一部分)：

$$\begin{aligned}f(x)=&\cdots\\&+[y_8-\alpha(8,x)y_9-\beta(9,x)y_{10}]F_8(x)\\&+[y_7-\alpha(7,x)y_8-\beta(8,x)y_9]F_7(x)\\&+[y_6-\alpha(6,x)y_7-\beta(7,x)y_8]F_6(x)\\&+[y_5-\alpha(5,x)y_6-\beta(6,x)y_7]F_5(x)\\&+\cdots\\&+[y_2-\alpha(2,x)y_3-\beta(3,x)y_4]F_2(x)\\&+[y_1-\alpha(1,x)y_2-\beta(2,x)y_3]F_1(x)\\&+[c_0+\beta(1,x)y_2-\beta(1,x)y_2]F_0(x)\end{aligned} \tag{5.5.22}$$

注意，最后一行增加并减去一项 $\beta(1,x)y_2$。认真考察(5.5.22)式中含有 y_8 因子的项，可以发现按(5.5.20)式的递推关系式其中和为零，其他 y_k 类似，直到 y_2。最后(5.5.22)式所剩项为

$$f(x)=\beta(1,x)F_0(x)y_2+F_1(x)y_1+F_0(x)c_0 \tag{5.5.23}$$

(5.5.21)式和(5.5.23)式是计算(5.5.19)式的 **Clenshaw 递推公式**。先用(5.5.21)式递推计算 y_k，求得 y_2 和 y_1 后，用(5.5.23)式求得所求之和。

Clenshaw 递推方法是按 k 的递减顺序合并系数 c_k。在每一步，所有以前 c_k 的效果作为两个系数"记忆"下来，然后用它们乘函数 F_{k+1} 和 F_k(最后为 F_0 和 F_1)。当 k 很大而函数 F_k 很小，并且 k 很小而 c_k 也很小时，和式主要由小的 F_k 决定。此时，被记忆的系数将被消去而失去重要性。例如求一个级数和

$$J_{15}(1)=0\times J_0(1)+0\times J_1(1)+\ldots+0\times J_{14}(1)+1\times J_{15}(1) \tag{5.5.24}$$

其中 J_{15}很小，最终表示为消去 J_0 和 J_1 的线性组合，这里 J_0 和 J_1 都是单位阶的。

这种情况下，解决办法是用另一种 Clenshaw 递推方式，即按上升顺序合并 c_k。相应的方程是

$$\begin{aligned}&y_{-2}=y_{-1}=0\\&y_k=\frac{1}{\beta(k+1,x)}[y_{k-2}-\alpha(k,x)y_{k-1}-c_k],\end{aligned} \tag{5.5.25}$$

$$(k=0,1,\ldots,N-1) \tag{5.5.26}$$

$$f(x)=c_NF_N(x)-\beta(N,x)F_{N-1}(x)y_{N-1}-F_N(x)y_{N-2} \tag{5.5.27}$$

自动验证(5.5.23)式的第一个求和运算项是否符号相反，并且数值是否基本相等，从而判断是否应该采用(5.5.25)式～(5.5.27)式，而不能采用(5.5.21)式和(5.5.23)式，但这种情况比较少见。除了这种特殊情况外，Clenshaw 递推总是稳定的，并且与函数 F_k 的递推是在上升方向还是下降方向上的稳定性无关。

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), pp. xiii, 697. [1]

Gautschi, W. 1967, *SIAM Review*, vol. 9, pp. 24–82. [2]

Lakshmikantham, V., and Trigiante, D. 1988, *Theory of Difference Equations: Numerical Methods and Applications* (San Diego: Academic Press). [3]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 20ff. [4]

Clenshaw, C.W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory (London: H.M. Stationery Office). [5]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §4.4.3, p. 111.

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), p. 76.

5.6 二次方程和三次方程

简单代数方程的根可视为该方程系数的函数，这些函数曾在初等代数中学习过。但遗憾的是，很多人不知道如何求解有两个实根的二次方程，或者如何求一个三次方程的实数解。

二次方程

$$ax^2 + bx + c = 0 \tag{5.6.1}$$

其中 a, b, c 为实数，它的解有两种写法，即

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5.6.2}$$

和

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}} \tag{5.6.3}$$

如果用(5.6.2)式或(5.6.3)式求根，则将产生问题：若 a 或 c(或两者都)很小，则其中一根是用 b 去减一个非常接近的量(即判别式)，这样得到的结果很不准确。正确的求解方式是计算

$$q \equiv -\frac{1}{2}\left[b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}\right] \tag{5.6.4}$$

则两根为

$$x_1 = \frac{q}{a} \qquad 和 \qquad x_2 = \frac{c}{q} \tag{5.6.5}$$

若系数 a, b, c 为复数而不是实数，上面公式仍然有效，只是(5.6.4)式中平方根的符号必须满足

$$\mathrm{Re}(b^* \sqrt{b^2 - 4ac}) \geqslant 0 \tag{5.6.6}$$

其中 Re 表示实部，星号(*)表示共轭复数。

关于二次方程，首先回忆一下反双曲函数，实际上它就是此方程解的对数：

$$\sinh^{-1}(x) = \ln\left(x + \sqrt{x^2 + 1}\right) \tag{5.6.7}$$

$$\cosh^{-1}(x) = \pm \ln\left(x + \sqrt{x^2 - 1}\right) \tag{5.6.8}$$

$x \geqslant 0$ 时，(5.6.7)式是稳健的。当 x 为负数时，利用对称性 $\sinh^{-1}(-x) = -\sinh^{-1}(x)$。而显然，(5.6.8)式仅对 $x \geqslant 1$ 时才有效。

三次方程

$$x^3 + ax^2 + bx + c = 0 \tag{5.6.9}$$

其中 a, b, c 为实系数或复系数，首先计算

$$Q \equiv \frac{a^2 - 3b}{9} \qquad 和 \qquad R \equiv \frac{2a^3 - 9ab + 27c}{54} \tag{5.6.10}$$

若满足Q和 R 是实数(当 a,b,c 为实数时通常满足)和 $R^2<Q^3$,则三次方程有三个实根。求根时先计算

$$\theta=\arccos(R/\sqrt{Q^3}) \tag{5.6.11}$$

由此得三个根为

$$\begin{aligned}x_1&=-2\sqrt{Q}\cos\left(\frac{\theta}{3}\right)-\frac{a}{3}\\x_2&=-2\sqrt{Q}\cos\left(\frac{\theta+2\pi}{3}\right)-\frac{a}{3}\\x_3&=-2\sqrt{Q}\cos\left(\frac{\theta-2\pi}{3}\right)-\frac{a}{3}\end{aligned} \tag{5.6.12}$$

这些方程式在 1615 年出版的 François Viète 的著作"De emendatione"第 6 章中第一次出现。

否则,计算

$$A=-\left[R+\sqrt{R^2-Q^3}\right]^{1/3} \tag{5.6.13}$$

其中平方根的符号应满足

$$\mathrm{Re}(R^*\sqrt{R^2-Q^3})\geqslant 0 \tag{5.6.14}$$

星号表示共轭复数。若Q和 R 均为实数,(5.6.13)式和(5.6.14)式等价于

$$A=-\mathrm{sgn}(R)\left[|R|+\sqrt{R^2-Q^3}\right]^{1/3} \tag{5.6.15}$$

其中假设平方根为正。下一步计算

$$B=\begin{cases}Q/A & (A\neq 0)\\0 & (A=0)\end{cases} \tag{5.6.16}$$

由此可知三个根为

$$x_1=(A+B)-\frac{a}{3} \tag{5.6.17}$$

(a,b,c 为实数是惟一的实根)和

$$\begin{aligned}x_2&=-\frac{1}{2}(A+B)-\frac{a}{3}+i\frac{\sqrt{3}}{2}(A-B)\\x_3&=-\frac{1}{2}(A+B)-\frac{a}{3}-i\frac{\sqrt{3}}{2}(A-B)\end{aligned} \tag{5.6.18}$$

(同样为一对共轭复数。)(5.6.13)式～(5.6.16)式的排列形式可将舍入误差降至最小,而且正如 A.J. Glassman 所指出的,还可以保证所选择计算立方根的方法不会导致根的丢失。

若要解一组仅系数稍有不同的三次方程组,则用牛顿法更有效(参见 9.4 节)。

参考文献和深入阅读

Weast, R.C. (ed.) 1967, *Handbook of Tables for Mathematics*, 3rd ed. (Cleveland: The Chemical Rubber Co.), pp. 130–133.

Pachner, J. 1983, *Handbook of Numerical Analysis Applications* (New York: McGraw-Hill), §6.1.

McKelvey, J.P. 1984, *American Journal of Physics*, vol. 52, pp. 269–270; see also vol. 53, p. 775, and vol. 55, pp. 374–375.

5.7　数值求导

若已求得一函数 $f(x)$,现在欲求其导数 $f'(x)$。根据导数定义,$h\to 0$ 时求极限

$$f'(x)\approx\frac{f(x+h)-f(x)}{h} \tag{5.7.1}$$

上式表明:选一很小的 h,计算 $f(x+h)$。$f(x)$值很可能已经有了,若还没有,也求一下。最后运用(5.7.1)式就可计算出导数。

事实上还有很多方法。直接按上面过程求导数,几乎肯定得到的是不精确的结果。只有当函数 f 值的计算极其费事,而又已经计算了 $f(x)$,并且只要最多再求一次函数值就可求得导数时,上述方法才可考虑使用。在这种情况下,剩下的任务就是寻找合适的 h,下面开始讨论。

(5.7.1)式有两个误差源,截尾误差和舍入误差。截尾误差来源于泰勒级数扩展式的高次项:

$$f(x+h)=f(x)+hf'(x)+\frac{1}{2}h^2f''(x)+\frac{1}{6}h^3f'''(x)+\cdots \tag{5.7.2}$$

即有

$$\frac{f(x+h)-f(x)}{h}=f'+\frac{1}{2}hf''+\cdots \tag{5.7.3}$$

而舍入误差来源较多。首先 h 中有舍入误差:假定在 $x=10.3$ 处随便选定 $h=0.0001$。$x=10.3$ 及 $x+h=10.300\,01$ 都不是具有能用精确二进制表示的数。因此,都具有机器浮点格式 ϵ_m(单精度时可达 10^{-7})的某些相对误差。h 的"有效值"误差,也就是机器表示的 $x+h$ 和 x 之间的差值,其数量级为 $\epsilon_m x$,其中隐含量级为 $\epsilon_m x/h\sim 10^{-2}$ 的 h 的相对误差。由(5.7.1)式即看出,导数中也存在同样大小的相对误差。

结论 1:选择 h,使得 $x+h$ 和 x 能被一个可表示的数精确地区分。通常用下面的程序步骤实现:

$$\begin{aligned}\texttt{temp}&=x+h\\ h&=\texttt{temp}-x\end{aligned} \tag{5.7.4}$$

某些优化的编译器和一些带浮点处理芯片的计算机,其内部精度高于外部存储器,这种方法就是失效。此时,通常说明 temp 是易失的,或者在(5.7.4)两式之间调用一次空函数 donothing(temp),以强制 temp 位于并超出可寻址存储器范围。

当 h 为精确数时,(5.7.1)式舍入误差为 $e_r\sim\epsilon_f|f(x)/h|$,其中 ϵ_f 为计算 f 的部分精度。对于简单的函数 ϵ_f 等于机器精度,$\epsilon_f\approx\epsilon_m$;但对于计算复杂而且具有不只一个误差源的情况,误差可能比较大。(5.7.3)式的截尾误差数量级为 $e_t\sim|hf''(x)|$。变化 h 使得和值 e_r+e_t 为最小,求得 h 最优值:

$$h\sim\sqrt{\frac{\epsilon_f f}{f''}}\approx\sqrt{\epsilon_f x_c} \tag{5.7.5}$$

其中 $x_c\equiv(f/f'')^{1/2}$ 为函数 f 的曲率尺度,或称其为变化的特性尺度。若无其他提示信息,则通常假定 $x_c=x$(除了在 $x=0$ 附近,此时应使用其他 x 尺度的典型估计值)。

使用(5.7.5)式后,计算导数的相对误差为

$$(e_r+e_t)/|f'|\sim\sqrt{\epsilon_f(ff''/f'^2)^{1/2}}\sim\sqrt{\epsilon_f} \tag{5.7.6}$$

上式中最后等式假定 f、f'、f'' 都具有相同的特性长度尺度,通常也正是如此。可以看出,简单的有限差分方程式(5.7.1)最多能精确到机器精度 ϵ_m 的平方根。

如果每次求导数时可以求两个函数值,运用下面的对称形式效果会好得多

$$f'(x)\approx\frac{f(x+h)-f(x-h)}{2h} \tag{5.7.7}$$

此时由(5.7.2)式得截尾误差为 $e_t\sim h^2f'''$。舍入误差与前面一样。由类似下面的简短计算,最优的 h 值为

$$h\sim\left(\frac{\epsilon_f f}{f'''}\right)^{1/3}\sim(\epsilon_f)^{1/3}x_c \tag{5.7.8}$$

相对误差为

$$(e_r+e_t)/|f'|\sim(\epsilon_f)^{2/3}f^{2/3}(f''')^{1/3}/f'\sim(\epsilon_f)^{2/3} \tag{5.7.9}$$

上式与(5.7.6)式相比,一般情况比较精确一个数量级(单精度)或二个数量级(双精度)。由此得出结论 2:选择 h 等于 ϵ_f 或 ϵ_m 的正确次幂乘以特征尺度 x_c。

对其他情况(参见本节参考文献[1])很容易导出正确的幂次。例如对二维函数,混合导数公式为

$$\frac{\partial^2 f}{\partial x\partial y}=\frac{[f(x+h,y+h)-f(x+h,y-h)]-[f(x-h,y+h)-f(x-h,y-h)]}{4h^2} \tag{5.7.10}$$

正确尺度是 $h\sim\epsilon_f^{1/4}x_c$。

没有简单的有限差分公式，如(5.7.1)式或(5.7.7)式能给出机器的精度ϵ_m的公式，或者甚至比求f值的估算精度ϵ_f更差的精度公式，这一点很令人失望。有没有更好的办法呢？

有的，不过要涉及到考察相对于x_c尺度上函数的行为，还得假定函数平滑，或可解析的，这样泰勒展开(5.7.2)式中的高次项才有意义。同时还涉及求函数值的多种方法，所以精度的提高是以计算的费用增大为代价的。

Richardson的延迟极限逼近思想很有意义。数值积分中龙贝格积分就来自这种思想(参见4.3节)。求导数时，可将由越来越小的h值求得的结果外推至$h\to 0$。由Neville算法(参见3.1节)，应用每次有限差分新的计算结果求得高阶的外推。Ridders[2]优秀地实现了这种思想。下面的程序dfridr就是基于他的算法，只是修改了终止条件。程序的输入是函数f(称为func)，正数x，最大步长h(与h相比它更像前面称为x_c的量)。输出为导数的返回值及误差估计值err。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

DP NR::dfridr(DP func(const DP), const DP x, const DP h, DP &err)
返回由Ridders多项式外推法求得的函数func在点x处的导数值。输入h值为估计初始步长，不必很小，而必须为能够
使func显著改变变量x的增量。导数的误差估计值的返回值为err。
{
    const int NTAB=10;                      设置表的最大尺度。
    const DP CON=1.4, CON2=(CON*CON);       每次迭代步长减小CON。
    const DP BIG=numeric_limits<DP>::max();
    const DP SAFE=2.0;                      当误差SAFE比目前求得最好结果还坏时返回。
    int i,j;
    DP errt,fac,hh,ans;
    Mat_DP a(NTAB,NTAB);

    if (h == 0.0) nrerror("h must be nonzero in dfridr.");
    hh=h;
    a[0][0]=(func(x+hh)-func(x-hh))/(2.0*hh);
    err=BIG;
    for (i=1;i<NTAB;i++) {
    Neville表中的连续列将进入更小的步长和更高阶的外推。
        hh /= CON;
        a[0][i]=(func(x+hh)-func(x-hh))/(2.0*hh);     试验新的更小步长。
        fac=CON2;
        for (j=1;j<=i;j++) {          不需再求函数值，计算不同阶的外推。
            a[j][i]=(a[j-1][i]*fac-a[j-1][i-1])/(fac-1.0);
            fac=CON2*fac;
            errt=MAX(fabs(a[j][i]-a[j-1][i]),fabs(a[j][i]-a[j-1][i-1]));
            误差策略是将每次新的外推结果与低一阶结果比较，包括当前步长及前一次步长相比较。
             if (errt <= err) {       若误差减小，保存改进的结果。
                 err=errt;
                 ans=a[j][i];
             }
         }
         if (fabs(a[i][i]-a[i-1][i-1]) >= SAFE*err) break;
         若高阶结果比SAFE坏，则尽早停止。
     }
     return ans;
}
```

程序dfridr中求函数func值的次数一般是6到12次，但可大到$2\times$NTAB次。h为输入量，h越大求得的精度也越好，直至在某突变点处，无意义的外推导致一个很大的误差返回值。所以必须选择适当大小的h，注意返回值err，若这个返回值不够小时减小h。函数的特征x尺度为单位数量级时，一般典型地选取h为十分之几。

除Ridders方法外还有其他一些方法。若函数很平滑，并且在某区间内任意点处要多次计算函数的导数，此时最好在此区间内构造函数的切比雪夫多项式逼近式，直接由求得的切比雪夫系数求导数。下面5.8节～5.9节将讨论这种方法。

另外还有一种方法，适用于在区间的等分点上的数据表组成的函数，可能有干扰。在每个所求点x值

处,用表中该点左边 n_L 个点及右边 n_R 个点,由最小二乘方拟合 M 阶多项式。导数估计值即为求得的拟合多项式的导数。Savitzky-Golay 平滑滤波器用于构造多项式时很有效,在 14.8 节中再讨论。那里将给出一个程序求滤波器系数,这些系数不仅可以构造拟合多项式,而且还可用来计算数据点乘滤波器系数和的累积值。给定的程序 savgol 中有一个参数 ld,它表示求拟合多项式的哪一阶导数。求第一阶导数时置 ld = 1,导数值是累积和除以采样区间 h。

参考文献和深入阅读

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*; reprinted 1996 (Philadelphia: S.I.A.M.), §§5.4–5.6. [1]

Ridders, C.J.F. 1982, *Advances in Engineering Software*, vol. 4, no. 2, pp. 75–76. [2]

5.8 切比雪夫逼近

n 阶切比雪夫多项式用 $T_n(x)$ 表示,写成公式为

$$T_n(x) = \cos(n \arccos x) \tag{5.8.1}$$

看上去它像三角函数公式(事实上,切比雪夫多项式和离散傅里叶变换有紧密联系),结合三角恒等式由(5.8.1)式可得 $T_n(x)$ 的显式表达式为(参见图 5.8.1)

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\cdots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geqslant 1 \end{aligned} \tag{5.8.2}$$

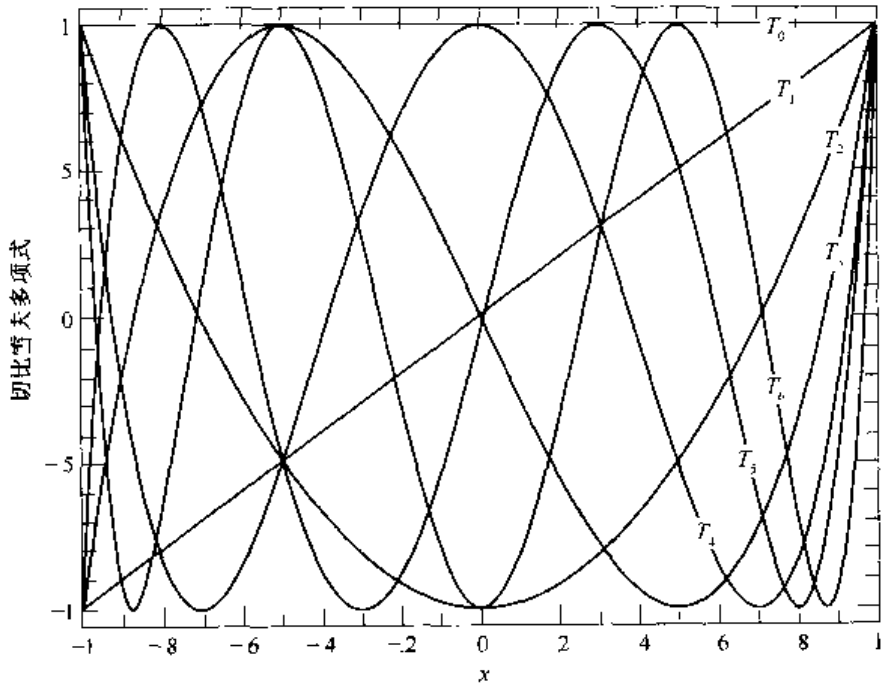


图 5.8.1 切比雪夫多项式 $T_0(x)$ 到 $T_6(x)$。注意 T_j 在[-1,1]区间上有 j 个根,且多项式值介于 ±1 之间

也存在上式反变换公式，即用 $T_n(x)$ 项表示的 x 幂次公式，见(5.11.2)式和(5.11.3)式。

切比雪夫多项式在 $[-1,1]$ 区间上对权 $(1-x^2)^{-1/2}$ 是正交的，特别是

$$\int_{-1}^{1}\frac{T_i(x)T_j(x)}{\sqrt{1-x^2}}dx=\begin{cases}0 & i\neq j\\ \pi/2 & i=j\neq 0\\ \pi & i=j=0\end{cases} \tag{5.8.3}$$

多项式 $T_n(x)$ 在 $[-1,1]$ 区间上有 n 个零点，分布在如下的点上：

$$x=\cos\left(\frac{\pi(k+\frac{1}{2})}{n}\right)\qquad k=0,1,\ldots,n-1 \tag{5.8.4}$$

在同样的区间上，有 $n+1$ 个极值(极大值和极小值)，其分布在如下的点上：

$$x=\cos\left(\frac{\pi k}{n}\right)\qquad k=0,1,\ldots,n \tag{5.8.5}$$

所有极大值处 $T_n(x)=1$，所有极小值处 $T_n(x)=-1$，正是这种性质使切比雪夫多项式在函数多项式逼近中非常有用。

切比雪夫多项式不仅满足连续正交关系，如(5.8.3)式，而且满足离散正交关系：若 $x_k(k=0,\ldots,m-1)$ 为由(5.8.4)式给出的 $T_m(x)$ 的 m 个零点，且 $i,j<m$，则

$$\sum_{k=0}^{m-1}T_i(x_k)T_j(x_k)=\begin{cases}0 & i\neq j\\ m/2 & i=j\neq 0\\ m & i=j=0\end{cases} \tag{5.8.6}$$

综合(5.8.1)式、(5.8.4)式和(5.8.6)式，不难证明下面的定理：设 $f(x)$ 为 $[-1,1]$ 区间上任意函数，定义 N 个系数 $c_j,j=0,\ldots,N-1$ 为

$$\begin{aligned}c_j&=\frac{2}{N}\sum_{k=0}^{N-1}f(x_k)T_j(x_k)\\&=\frac{2}{N}\sum_{k=0}^{N-1}f\left[\cos\left(\frac{\pi(k+\frac{1}{2})}{N}\right)\right]\cos\left(\frac{\pi j(k+\frac{1}{2})}{N}\right)\end{aligned} \tag{5.8.7}$$

那么逼近公式为

$$f(x)\approx\left[\sum_{k=0}^{N-1}c_kT_k(x)\right]-\frac{1}{2}c_0 \tag{5.8.8}$$

在 $T_N(x)$ 的 N 个零点处，上式精确相等。

N 确定时，(5.8.8)式就是 $[-1,1]$ 区间内逼近 $f(x)$ 的多项式($T_N(x)$ 的所有零点也在此区间内)。为什么说这种特殊多项式逼近比在此区间内其他任何严格的逼近函数要好呢？这并不是因为(5.8.8)式比其他 N 阶多项式更精确(对特定意义上的精确性来说)，而是因为(5.8.8)式可非常优秀地截为低阶($m\ll N$)多项式，能得到 m 阶最好的逼近(在某种准确性的意义上来说)。设 N 很大，以致可认为(5.8.8)式是 $f(x)$ 的完全逼近。现考察截尾逼近

$$f(x)\approx\left[\sum_{k=0}^{m-1}c_kT_k(x)\right]-\frac{1}{2}c_0 \tag{5.8.9}$$

c_j 同样由(5.8.7)式得到。因 $T_k(x)$ 限定在 ±1 之间，故(5.8.9)式与(5.8.8)式之差不可能大于舍弃的 $c_k(k=m,\ldots,N-1)$ 之和。若 c_k 衰减很快(通常如此)，则误差主要由 $c_mT_m(x)$ 确定，$c_mT_m(x)$ 为一振荡函数，在区间 $[-1,1]$ 上平滑地分布着 $m+1$ 个相同的极值。这种误差的平滑扩散性质很重要，这样(5.8.9)式的切比雪夫多项式就是我们想得到的逼近多项式——**极小极大多项式**，它在所有相同阶数的多项式中，与原函数 $f(x)$ 的最大偏差达到最小。极小极大多项式很难找，而切比雪夫逼近多项式几乎是恒等的而且很容易计算。

给定计算函数 $f(x)$ 值的一些方法后，需找到算法以实现(5.8.7)式，以及(检验得到的 c_k 和截尾值 m 后)求(5.8.9)式之值。以后用此式求 $f(x)$ 的值就很方便了。

第一项任务可直接计算。这里用(5.8.7)式的一般推广形式，即允许在任意两个数 a 和 b 范围内逼近，

而不仅仅限于 -1 到 1 之间. 对此可按下面的方式进行变量变换:

$$y \equiv \frac{x - \frac{1}{2}(b+a)}{\frac{1}{2}(b-a)} \tag{5.8.10}$$

以及实现对 y 的 $f(x)$ 的切比雪夫多项式逼近:

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::chebft(const DP a, const DP b, Vec_O_DP &c, DP func(const DP))
切比雪夫拟合:给定函数 func,区间上下极限[a,b],n 维输出数组c[0..n-1]。本程序先计算切比雪夫逼近的 n 个系数
c[0..n-1],使 func(x)≈[Σ_{k=0}^{n-1} c_k T_k(y)] - c_0/2,其中 y 与 x 关系由(5.8.10)式确定。调用时 n 大小选取应适当(如 30 或
50),然后在较小的 m 值处截去 c 数组,使 c_m 及其以后元素均可忽略。
{
    const DP PI=3.141592653589793;
    int k,j;
    DP fac,bpa,bma,y,sum;

    int n=c.size();
    Vec_DP f(n);
    bma=0.5*(b-a);
    bpa=0.5*(b+a);
    for (k=0;k<n;k++) {                    由(5.8.7)式求函数在n个点上的值。
        y=cos(PI*(k+0.5)/n);
        f[k]=func(y*bma+bpa);
    }
    fac=2.0/n;
    for (j=0;j<n;j++) {                    现在计算(5.8.7)。
        sum=0.0;
        for (k=0;k<n;k++)
            sum += f[k]*cos(PI*j*(k+0.5)/n);
        c[j]=fac*sum;
    }
}
```

(若发现程序 chebft 执行的大部分时间是在计算 N^2 次余弦,而不是求 N 次函数值,此时应参见 12.3 节,尤其是(12.3.22)式,讨论如何用快速余弦变换方法求(5.8.7)式的值。)

下面介绍在切比雪夫系数值后如何求逼近值。有人可能会用(5.8.2)式递推关系从 $T_0 = 1, T_1 = x$ 来求 $T_k(x)$,再求(5.8.9)式之和。更好的办法是用 Clenshaw 递推公式(参见 5.5 节),同时完成两个过程。应用于切比雪夫级数(5.8.9)的递推为

$$\begin{aligned} d_{m+1} &\equiv d_m \equiv 0 \\ d_j &= 2xd_{j+1} - d_{j+2} + c_j \qquad j = m-1, m-2, \ldots, 1 \\ f(x) &\equiv d_0 = xd_1 - d_2 + \frac{1}{2}c_0 \end{aligned} \tag{5.8.11}$$

```
#include "nr.h"

DP NR::chebev(const DP a, const DP b, Vec_I_DP &c, const int m, const DP x)
切比雪夫求值:参数均为输入。c[0..m-1]为切比雪夫系数数组,c 中前 m 个元素,应由 chebft(调用时 a 和 b 应相同)输
出。在 y=[x-(b+a)/2]/[(b-a)/2]处求值切比雪夫多项式Σ_{k=0}^{m-1} c_k T_k(y) - c_0/2。返回的结果是函数值。
{
    DP d=0.0,dd=0.0,sv,y,y2;
    int j;

    if ((x-a)*(x-b) > 0.0)
        nrerror("x not in range in routine chebev");
    y2=2.0*(y=(2.0*x-a-b)/(b-a));          变量替换。
    for (j=m-1;j>0;j--) {                  clenshaw递推式。
        sv=d;
        d=y2*d-dd+c[j];
```

```
        dd=sv;
    }
    return y*d-dd+0.5*c[0];                最后一步是不同的。
}
```

若区间[-1,1]上的逼近函数为偶函数,其展开式只包含偶次切比雪夫多项式。奇系数全为零时,调用程序 chebev 是没有用的[1]。对(5.8.1)式应用余弦半角恒等式,得到关系式

$$T_{2n}(x) = T_n(2x^2 - 1) \tag{5.8.12}$$

这样,可将偶系数连续存放在数组 c 中,而变量 x 由 $2x^2-1$ 代替,然后调用 chebev 函数后,就可求一系列偶切比雪夫多项式的值。

奇函数的展开式则只有奇切比雪夫多项式。最好是将其改写为函数 $f(x)/x$ 的扩展式,这样仅含偶切比雪夫多项式。在 $x=0$ 附近可求得精确值。$f(x)/x$ 的系数 c'_n 可由 $f(x)$ 的系数按下面的递推式求得:

$$\begin{aligned} c'_{N+1} &= 0 \\ c'_{n-1} &= 2c_n - c'_{n+1}, \qquad n = N-1, N-3, \ldots \end{aligned} \tag{5.8.13}$$

(5.8.13)式服从(5.8.2)式的递推关系式。

若一定要求奇切比雪夫级数的值,较好的方法是再用 $y=2x^2-1$ 替换 x,并将奇系数连续存放在数组 c 中,现将(5.8.11)式中最后一个公式改为

$$f(x) = x[(2y-1)d_1 - d_2 + c_0] \tag{5.8.14}$$

并修改程序 chebev 中的相应行。

参考文献和深入阅读

Clenshaw, C.W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory, (London: H.M. Stationery Office). [1]

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), Chapter 8.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §4.4.1, p. 104

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.5.2, p. 334.

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), §1.10, p. 39.

5.9 切比雪夫逼近函数的微分和积分

如果得到某一范围内逼近一个函数的切比雪夫系数之后(如5.8节的 chebft 程序),把它们变换成与该函数微分和积分相对应的切比雪夫系数就很简单了。完成这种变换后,就可以计算其导数与积分值,如同一开始就在进行拟合切比雪夫函数。

相应的公式为:设 $c_i, i=0,\ldots,m-1$ 为(5.8.9)式中逼近函数 f 的系数,C_i 为逼近 f 的不定积分的系数,c'_i 为逼近 f 的导数的系数,则有

$$C_i = \frac{c_{i-1} - c_{i+1}}{2i} \qquad (i > 0) \tag{5.9.1}$$

$$c'_{i-1} = c'_{i+1} + 2ic_i \qquad (i = m-1, m-2, \ldots, 1) \tag{5.9.2}$$

相应于积分常数的任意性,(5.9.1)式中 C_0 的选择也是任意的,递推式(5.9.2)起始项 $c'_m = c'_{m-1} = 0$,这些值表示关于原函数 f 的第 $m+1$ 项切比雪夫多项式系数没有信息。

下面是实现(5.9.1)式和(5.9.2)式的程序。

```
#include "nr.h"

void NR::chder(const DP a, const DP b, Vec_I_DP &c, Vec_O_DP &cder, const int n)
```

给定 a,b,c[0..n-1]为5.8节中程序 chebft 的输出,并给定 n 为要求逼近阶数(即所使用的 c 的长度)。本程序返回数组 cder[0..n-1],是系数为 c 的函数导数的切比雪夫系数。

```
{
    int j;
    DP con;

    cder[n-1]=0.0;                          n-1和n-2是特殊情况。
    cder[n-2]=2*(n-1)*c[n-1];
    for (j=n-2;j>0;j--)
        cder[j-1]=cder[j+1]+2*j*c[j];       (5.9.2)式。
    con=2.0/(b-a);
    for (j=0;j<n;j++)                       区间b-a内归一化。
        cder[j] *= con;
}


#include "nr.h"

void NR::chint(const DP a, const DP b, Vec_I_DP &c, Vec_O_DP &cint, const int n)
给定a,b,c[0..n-1]为5.8节中程序chebft的输出,并给定n为要求逼近阶数(即所使用的c的长度),本程序返回数组
cint[0..n-1],是系数为c的函数积分的切比雪夫系数。选择积分常数使得积分在a处为零。
{
    int j;
    DP sum=0.0,fac=1.0,con;

    con=0.25*(b-a);                         区间b-a内的归一化因子。
    for (j=1;j<n-1;j++) {
        cint[j]=con*(c[j-1]-c[j+1])/j;      (5.9.1)式。
        sum += fac*cint[j];                 累积积分常数。
        fac = -fac;                         将等于±1.
    }
    cint[n-1]=con*c[n-2]/(n-1);             对于n-1,(5.9.1)式的特殊情况。
    sum += fac*cint[n-1];
    cint[0]=2.0*sum;                        设置积分常数。
}
```

5.9.1 深入讨论:Clenshaw-Curtis 积分法

因为平滑函数的切比雪夫系数 c_j 一般按指数迅速减小,故(5.9.1)式通常是一种高效求积分方法的基础。若需在 $a \leqslant x \leqslant b$ 范围中,对 x 的许多不同值求函数 $\int_a^x f(x)\,dx$,则可按顺序使用程序 chebft 和 chint,接着反复调用 chebev。

若仅求单个定积分 $\int_a^b f(x)\,dx$,由(5.9.1)式可导出,此时 chint 和 chebev 需由下面的简单公式替换:

$$\int_a^b f(x)dx = (b-a)\left[\frac{1}{2}c_0 - \frac{1}{3}c_2 - \frac{1}{15}c_4 - \cdots - \frac{1}{(2k+1)(2k-1)}c_{2k} - \cdots\right] \tag{5.9.3}$$

其中的 c_j 由程序 chebft 求出。当 c_{2k} 可忽略时,截断级数,并且第一个忽略项为估计误差。

这种方法称为 Clenshaw-Curtis(克伦肖-柯蒂斯)积分法[1]。通常还能自适应选择 N,它是(5.8.7)式计算的切比雪夫系数的个数,同时也是函数 $f(x)$ 的求值次数。若 N 的选择使得(5.9.3)式中 c_{2k} 不是足够小,则应试一下 N 大一点的值。此时更好的方法是,用"梯形"或 Gauss-Lobatto(参见 4.5 节)变量来替换(5.8.7)式,

$$c_j = \frac{2}{N}\sum_{k=0}^{N}{}'' f\left[\cos\left(\frac{\pi k}{N}\right)\right]\cos\left(\frac{\pi jk}{N}\right) \qquad j = 0,\ldots,N-1 \tag{5.9.4}$$

上式中的两撇(")表示和式中的第一项和最后一项都要乘以 1/2。若(5.9.4)式中 N 值增加 1 倍,则求函数值的点有一半与前一次一样,这样前面计算的函数值可重新使用。这种特性,再加上权和坐标的解析性((5.9.4)式中的余弦函数),使得 Clenshaw-Curtis 求积方法比高阶自适应高斯求积方法(参见 4.5 节)稍好一些。

若问题中要求 N 值很大,此时应注意到,同时对所有 j 值,通过快速余弦变换可以很快求得(5.9.4)式之值(参见 12.3 节,尤其是(12.3.17)式)。已讨论过(5.8.7)式的非梯形形式也可以由快速余弦方法求出,参见(12.3.22)式。

参考文献和深入阅读

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), pp. 78–79.

Clenshaw, C.W., and Curtis, A.R. 1960, *Numerische Mathematik*, vol. 2, pp. 197–205. [1]

5.10 切比雪夫系数的多项式逼近

读过前面两节后可能会问,"是否对变换后的变量 y 必须存储和计算这个切比雪夫系数数组的切比雪夫逼近式。难道不能将 c_k 转换成原变量 x 的实际多项式的系数,而得到下面的多项式逼近形式吗?"

$$f(x) \approx \sum_{k=0}^{m-1} g_k x^k \tag{5.10.1}$$

是的,可以这样做(为此,我们还将提供算法),但必须注意:对(5.10.1)式的求值(其中系数g隐含了切比雪夫逼近)通常比直接对切比雪夫和式求值需要更多的有效数字(参见程序 chebev)。这是由于切比雪夫多项式本身有一种很巧妙的消去性质:如 $T_n(x)$的首项系数为 2^{n-1},$T_n(x)$的其他系数甚至更大,但经它们组合而成的多项式界于 ± 1 之间。仅当 m 不大于 7 或 8 时,才能考虑将切比雪夫逼近式写成直接多项式形式。即使这样,也还必须能容许精度比机器的舍入极限低 2 个左右的有效数字。

在(5.10.1)式中由 chebft(在恰当的 m 值处适当地截尾)输出的 c 来求得g,只需顺序地调用下面两个函数:

```
#include "nr.h"

void NR::chebpc(Vec_I_DP &c, Vec_O_DP &d)
切比雪夫多项式系数。给定系数数组 c[0..n-1],本程序生成系数数组 d[0..n-1],使得∑_{k=0}^{n-1} d_k y^k = [∑_{k=0}^{n-1} c_k T_k(y) - c_0/2]。方法是 Clenshaw 递推式(5.8.11),只是代数地而不是算术地应用。
{
    int k,j;
    DP sv;

    int n=c.size();
    Vec_DP dd(n);
    for (j=0;j<n;j++) d[j]=dd[j]=0.0;
    d[0]=c[n-1];
    for (j=n-2;j>0;j--) {
        for (k=n-j;k>0;k--) {
            sv=d[k];
            d[k]=2.0*d[k-1]-dd[k];
            dd[k]=sv;
        }
        sv=d[0];
        d[0] = -dd[0]+c[j];
        dd[0]=sv;
    }
    for (j=n-1;j>0;j--)
        d[j]=d[j-1]-dd[j];
    d[0] = -dd[0]+0.5*c[0];
}

#include "nr.h"

void NR::pcshft(const DP a, const DP b, Vec_IO_DP &d)
多项式系数转换。给定系数数组 d[0..n-1],本程序生成系数数组g[0..n-1],使得∑_{k=0}^{n-1} d_k y^k = ∑_{k=0}^{n-1} g_k x^k,其中 x 和 y 的关系由(5.8.10)式确定,即将 -1 < y < 1 区间映射到 a < x < b 区间上。数组g返回到 d 中。
{
    int k,j;
    DP fac,cnst;

    int n=d.size();
    cnst=2.0/(b-a);
    fac=cnst;
```

```
        for (j=1;j<n;j++) {          先用因子const调整比例……
            d[j] *= fac;
            fac *= cnst;
        }
        cnst=0.5*(a+b);              …… 然后重新定义所期望的转换。
        for (j=0;j<=n-2;j++)         这里用综合除法完成转换，综合除法是中学代数课内容，
            for (k=n-2;k>=j;k--)     若没有学过的读者，仍可以照此做下去。
                d[k] -= cnst*d[k+1];
    }
```

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 59, 182–183 [synthetic division].

5.11 深入讨论:幂级数的化简

切比雪夫方法比较特殊的应用例子是**幂级数化简**,这是一种很有用的方法,即使看上去无太大用处。

假定用收敛幂级数求函数值,例如

$$f(x) \equiv 1 - \frac{x}{3!} + \frac{x^2}{5!} - \frac{x^3}{7!} + \cdots \tag{5.11.1}$$

(此函数实际上就是 $\sin(\sqrt{x})/\sqrt{x}$,不过我们还是假定不知道。)读者可能要求在某特定区间如$[0,(2\pi)^2]$内,多次求此级数值。那么,现在就要找到级数中某一项,使得在此项之前的有限级数引起的误差(近似等于忽略项的第一项)是允许的。在上面示例中,若 $x=(2\pi)^2$,第一个小于 10^{-7} 的项为 $x^{13}/(27!)$。因此,此项近似等于最后一项为 $x^{12}/(25!)$的有限级数的误差。

注意,因为 x^{13} 的指数很大,因此在特定区间中除了在 x 很大的位置外,误差处处比 10^{-7} **小得多**。此特性便于"简化":若我们让区间内误差大到等于第一个忽略项在此区间端点处的值,那么就可以用一个很短的级数替代 13 项级数 f。

分下面几步完成:

1. 将 x 变换到变量 y,如(5.8.10)式,将 x 区间映射到 $-1 \leqslant y \leqslant 1$。
2. 求正好等于截尾级数(项数足够多可满足精度)的切比雪夫和式(如(5.8.8)式)的系数。
3. 将切比雪夫级数截成项数少的级数,用第一个被忽略的切比雪夫多项式的系数作为估计误差。
4. 转换成 y 的多项式。
5. 变量再转换回 x。

当给定原级数展开式的系数,上面各步都可以数值计算。第一步正好是程序 pcshft(参见 5.10 节)的逆过程,那里是将 y 的多项式(区间$[-1,1]$)映射为 x 的多项式(区间$[a,b]$)。因为(5.8.10)式为 x 和 y 的线性关系式,所以也可将 pcshft 用于此逆过程。

$$\texttt{pcshft}(a,b,\texttt{d},\texttt{n})$$

的逆过程是(可以验证一下)

$$\texttt{pcshft}\left(\frac{-2-b-a}{b-a},\frac{2-b-a}{b-a},\texttt{d},\texttt{n}\right)$$

第二步是求程序 chebpc(将切比雪夫系数用于多项式系数)的逆操作。下面给出的程序 pccheb 正是完成此过程,应用公式[1]

$$x^k = \frac{1}{2^{k-1}}\left[T_k(x) + \binom{k}{1}T_{k-2}(x) + \binom{k}{2}T_{k-4}(x) + \cdots\right] \tag{5.11.2}$$

上式中最后一项依 k 是奇数还是偶数而定,

$$\cdots + \binom{k}{(k-1)/2}T_1(x)\ (k\text{为奇数}),\quad \cdots + \frac{1}{2}\binom{k}{k/2}T_0(x)\ (k\text{为偶数}) \tag{5.11.3}$$

```
#include "nr.h"

void NR::pccheb(Vec_I_DP &d, Vec_O_DP &c)
程序 chebpc 的逆过程:给定多项式系数数组 d[0..n-1],返回等价切比雪夫系数数组 c[0..n-1]。
{
    int j,jm,jp,k;
    DP fac,pow;

    int n=d.size();
    pow=1.0;                    2的幂次。
    c[0]=2.0*d[0];
    for (k=1;k<n;k++) {         在多项式的x阶次上循环。
        c[k]=0.0;               零对应切比雪夫阶次。
        fac=d[k]/pow;
        jm=k;
        jp=1;
        for (j=k;j>=0;j-=2,jm--,jp++) {
        循环,以求具有组合系数乘以 d[k]的切比雪夫的低阶项,见文中公式。
            c[j] += fac;
            fac *= DP(jm)/DP(jp);
        }
        pow += pow;
    }
}
```

第四步和第五步分别由 chebpc 和 pcshft 完成。整个过程表述为下面形式:

```
const int NFEW = ... NMANY = ...
DP a,b;
Vec_DP c(NMANY),d(NFEW),e(NMANY);
在区间(a,b)中,化简 NMANY 幂级数的系数 e[0..NMANY-1]为 NFEW 系数 d[0..NFEW-1]。

NR::pcshft((-2.0-b-a)/(b-a),(2.0-b-a)/(b-a),e);
NR::pccheb(e,c);
...
这里一般需检查切比雪夫系数 c[0..NMANY-1]以决定 NFEW 该多小。
NR::chebpc(c,d);
NR::pcshft(a,b,d);
```

上面的例子中,第8个至第10个切比雪夫系数算出数量级分别为 -7×10^{-6},3×10^{-7} 和 -9×10^{-9},这样适当截尾后允许误差落在 10^{-7} 范围之内(单精度计算),得到的是8~10项的多项式,而不是原来的13项。

13项的多项式可用一个10项多项式替换而没有减少精度,看上去确实没有什么作用。决定函数 $f(x)$ 是13项多项式。而与化简级数等价的是,我们可以在感兴趣的区间内,用5.8节中的方法,在足够多点上构造切比雪夫逼近式来替换求函数 $f(x)$ 值。这样也能得到同样低阶多项式。主要是因为,切比雪夫系数的收敛速率与幂级数系数级数的收敛速率无关,而正是由前者决定多项式逼近式需要的项数。函数在某感兴趣的区域内可能有发散级数,但若函数本身平滑,则一定存在性能优良的多项式逼近式。可由5.8节中的方法找出,不要通过级数化简法寻找。

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 12.

Arfken, G. 1970, *Mathematical Methods for Physicists*, 2nd ed. (New York: Academic Press), p. 631. [1]

5.12 深入讨论:帕德逼近

所谓**帕德(Padé)逼近**是一种(特定阶的)有理函数,其幂级数展开式与给定级数的最高可能阶相同。若有理函数为

$$R(x) \equiv \frac{\sum_{k=0}^{M} a_k x^k}{1+\sum_{k=1}^{N} b_k x^k} \tag{5.12.1}$$

则称 $R(x)$ 为如下级数的帕德逼近式：

$$f(x) \equiv \sum_{k=0}^{\infty} c_k x^k \tag{5.12.2}$$

若

$$R(0) = f(0) \tag{5.12.3}$$

且

$$\frac{d^k}{dx^k} R(x)\bigg|_{x=0} = \frac{d^k}{dx^k} f(x)\bigg|_{x=0}, \qquad k = 1, 2, \ldots, M+N \tag{5.12.4}$$

(5.12.3)式及(5.12.4)式组成 $M+N+1$ 个方程可以求未知数 $a_0, \ldots, a_M$ 和 $b_1, \ldots, b_N$。得到这些方程最简单的方法是，让(5.12.1)式和(5.12.2)式相等，两边同时乘以(5.12.1)式的分母，再对系数含 a 或 b 的 x 幂次项列成等式。考查 $M=N$ 的对角有理逼近的特殊情况(参见 3.2 节)，则有 $a_0 = c_0$，剩下 a 和 b 满足

$$\sum_{m=1}^{N} b_m c_{N-m+k} = -c_{N+k}, \qquad k = 1, \ldots, N \tag{5.12.5}$$

$$\sum_{m=0}^{k} b_m c_{k-m} = a_k, \qquad k = 1, \ldots, N \tag{5.12.6}$$

注意(5.12.1)式中 $b_0 = 1$。解方程时从(5.12.5)式开始，因为它是一组对所有未知数 b 的线性方程，尽管方程组为 Toeplitz 矩阵形式(参见(2.8.8)式)，但经验表明方程通常表现为退化形式，因此不要用 2.8 节中的方法求解，而用 LU 分解法。此外，用迭代改进修正解是一个好方法(2.5 节中程序 mprove)[1]。

一旦 b 已知，则由(5.12.6)式得到求解未知数 a 的明晰公式，就可完成求解。

帕德逼近式典型地用于求某未知基础函数 $f(x)$。假定要计算 $f(x)$ 及 $f(x)$ 在 $x=0$ 的导数值：$f(0)$，$f'(0)$，$f''(0)$ 等，而且可能必须由繁琐的解析展开式计算得到。当然，这些只是 $f(x)$ 幂级数展开式的前面几项，但这些项数不必要变少，因为也不知道幂级数在何处(是否会)收敛。

与切比雪夫逼近式(参见 5.8 节)，或者幂级数化简(参见 5.11 节)这类已知的函数信息的方法相比，帕德逼近式能给出关于函数值新的信息。下面举例说明。

假定经过努力，已求得某未知函数 $f(x)$ 的幂级数展开式的前五项为

$$f(x) \approx 2 + \frac{1}{9}x + \frac{1}{81}x^2 - \frac{49}{8748}x^3 + \frac{175}{78732}x^4 + \cdots \tag{5.12.7}$$

(当然不必将系数写成精确的有理形式，写成数值的值即可。这里写成有理形式是为了提醒读者这些系数是通过分析计算得到的。)图 5.12.1 中将(5.12.7)式描绘成标有"幂级数"的曲线。可以看出 $x \gtrsim 4$ 时，主要是由最大的四次方项决定。

取(5.12.7)式中的 5 个系数，并输入给下面的程序 padé 中运行。返回 5 个有理系数，3 个 a 值和 2 个 b 值，它是将 $M=N=2$ 代入(5.12.1)式中得到的。图 5.12.1 中标有"帕德逼近"的曲线绘出所得到的有理函数。注意，图中两条实线都是由 5 个相同的原始系数值导出的。

为求得结果，需要用 Deus ex machina (拉丁语，意为上帝的机器——译者注)，它的存在对后面有用。结果可知(5.12.7)式实际上就是如下函数的幂级数展开式：

$$f(x) = [7 + (1+x)^{4/3}]^{1/3} \tag{5.12.8}$$

见图中绘出的点线。此函数在 $x=-1$ 处存在分支点，故级数只在 $-1<x<1$ 范围内收敛。在图 5.12.1 中大部分区间，级数发散，因此对第五项的截尾项就没有意义了。但是，同样五项，转换成帕德逼近式后，得到的函数值至少在 $x \sim 10$ 处仍有相当好的表示式。

是否有其他前五项幂级数相同的函数，但在 $2<x<10$ 范围内具有完全不同的性质，确实有。帕德逼近

式具有一种从想得到的各种可能性中挑选合适函数的功能。帕德逼近式的缺点是难于控制。一般无法确知其精度有多高，或者能有效地扩展到 x 的哪个位置。这是一种有效的方法。

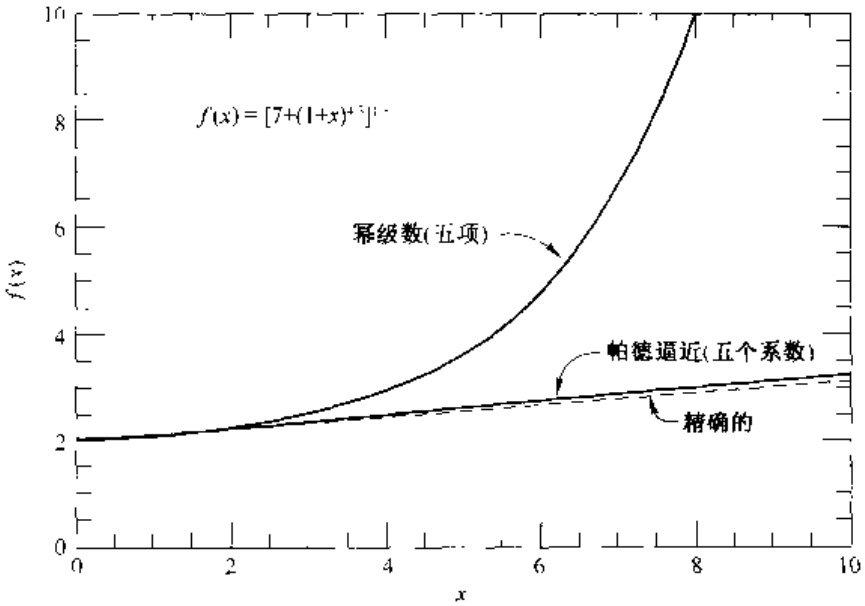


图 5.12.1　五项幂级数展开式及导出的同一函数 $f(x)$ 的五项系数帕德逼近式。全级数仅对 $x<1$ 收敛。注意帕德逼近式在远远超过级数收敛半径的位置仍能保证精确性

下面程序从 c 中计算出 a 和 b。注意，程序是针对 $M=N$ 的特殊情况，而且输出的有理系数排放成便于求值程序 ratval(参见 5.3 节)使用的格式。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::pade(Vec_IO_DP &cof, DP &resid)
给定函数的幂级数展开式前几项系数 cof[0..2*n]，解线性帕德方程，返回同一函数的对角有理函数逼近式的系数，函
数为(cof[0] + cof[1]x + ··· + cof[n]x^n)/(1 + cof[n+1]x + ··· + cof[2*n]x^n)。值 resid 是剩余向量的范数，其值很小表明为
收敛很好的解。
{
    const DP BIG=1.0e30;
    int j,k;
    DP d,rr,rrold,sum;

    int n=(cof.size()-1)/2;
    Mat_DP q(n,n),qlu(n,n);
    Vec_INT indx(n);
    Vec_DP x(n),y(n),z(n);
    for (j=0;j<n;j++) {                 为求解设置矩阵。
        y[j]=x[j]=cof[n+j+1];
        for (k=0;k<n;k++) {
            q[j][k]=cof[j-k+n];
            qlu[j][k]=q[j][k];
        }
    }
    ludcmp(qlu,indx,d);                 由 LU 分解和回代法求解。
    lubksb(qlu,indx,x);
    rr=BIG;
    do {                                使用迭代优化很重要，因为 Padé 方法有病态。
        rrold=rr;
        for (j=0;j<n;j++) z[j]=x[j];
        mprove(q,qlu,indx,y,x);
        for (rr=0.0,j=0;j<n;j++)        计算剩余量。
            rr += SQR(z[j]-x[j]);
    } while (rr < rrold);               若不能再优化，退出。
    resid=sqrt(rrold);
    for (k=0;k<n;k++) {                 计算其他系数。
```

```
            for (sum=cof[k+1],j=0;j<=k;j++)
                sum -= z[j]*cof[k-j];
            y[k]=sum;
        }
        for (j=0;j<n;j++) {
            cof[j+1]=y[j];
            cof[j+1+n] = -z[j];
        }
    }
```

输出结果。

参考文献和深入阅读

Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), p. 14.

Cuyt, A., and Wuytack, L. 1987, *Nonlinear Methods in Numerical Analysis* (Amsterdam: North-Holland), Chapter 2.

Graves-Morris, P.R. 1979, in *Padé Approximation and Its Applications*, Lecture Notes in Mathematics, vol. 765, L. Wuytack, ed. (Berlin: Springer-Verlag). [1]

5.13 深入讨论:有理切比雪夫逼近

在5.8节和5.10节中我们已经知道如何对给定区间 $a \leqslant x \leqslant b$ 内的给定函数 $f(x)$,求出其较好的多项式逼近式。本节中对此工作进行一般化推广,寻找有理函数的逼近式。这样做的原因是,对有些区间和函数,从最优有理函数逼近式能比具有相同系数个数的最优多项式逼近式得到更高的精度。必须权衡的是,求有理函数逼近式不如求多项式逼近式来得简单直接。正如我们已知的,由切比雪夫多项式求多项式的逼近式很方便。

设有理函数 $R(x)$ 的分子为 m 次,分母为 k 次。则有

$$R(x) \equiv \frac{p_0 + p_1 x + \cdots + p_m x^m}{1 + q_1 x + \cdots + q_k x^k} \approx f(x) \qquad 对\ a \leqslant x \leqslant b \tag{5.13.1}$$

要求的未知量是 $p_0, \ldots, p_m$ 和 $q_1, \ldots, q_k$,即共有 $m+k+1$ 个量。设 $r(x)$ 表示 $R(x)$ 与 $f(x)$ 之差,且 r 表示其绝对值的最大值,

$$r(x) \equiv R(x) - f(x) \qquad r \equiv \max_{a \leqslant x \leqslant b} |r(x)| \tag{5.13.2}$$

理想的最优解就是求适当 p 和 q 之值使得 r 最小。显然存在最小解,因为 r 的下限为零。但如何去求解,或者如何合适地逼近它呢?

首先,可从下面基本定理获得一些启发:若 $R(x)$ 非退化(即分子、分母中没有相同的多项式因子),则使得 r 最小的 p 和 q 的值是惟一的。在选择此最优值时,$r(x)$ 在 $a \leqslant x \leqslant b$ 中有 $m+k+2$ 个极值,**大小为 r 且符号交替变化**(定理中省去了许多假设,准确描述见 Ralston[1])。由此可知,有理函数情况与最小多项式情况非常相似:在5.8节中已看到,具有 $n+1$ 个切比雪夫系数的 n 阶逼近式的误差项,主要是由第一个被忽略的切比雪夫项构成,即 T_{n+1},它本身有 $n+2$ 个幅度相等的极点,且符号交替变化。因此,这里有理系数的个数 $m+k+1$ 与多项式系数的个数 $n+1$,所起的作用是相同的。

$r(x)$ 有 $m+k+2$ 个极值的另一种解释方法是,可使得 $R(x)$ 在任意 $m+k+1$ 个点 x_i 处精确地等于 $f(x)$。对(5.13.1)式乘以它的分母得到方程

$$\begin{gathered} p_0 + p_1 x_i + \cdots + p_m x_i^m = f(x_i)(1 + q_1 x_i + \cdots + q_k x_i^k) \\ i = 0, 1, \ldots, m+k \end{gathered} \tag{5.13.3}$$

这是一组未知量 p 和 q 的 $m+k+1$ 个线性方程组,可由标准方法(如LU分解)求解。若选定所有 x_i 值使其落在区间 (a,b) 中,则在每个选定的 x_i 和 x_{i+1} 之间有一个极值点,再加上不在区间 (a,b) 中的极值点,总共 $m+k+2$ 个极值点。对 x_i 的任意极值并不总是有相同幅度。定理中表明选定一组特别的 x_i,幅度可能相同,且等于最小值 r。

若不用在 x_i 处使 $f(x_i)$ 和 $R(x_i)$ 相等,也可以通过解下面线性方程使剩余量 $r(x_i)$ 等于任意设定值 y_i:

$$p_0 + p_1x_i + \cdots + p_mx_i^m = [f(x_i) - y_i](1 + q_1x_i + \cdots + q_kx_i^k)$$
$$i = 0, 1, \ldots, m + k \tag{5.13.4}$$

事实上，若 x_i 选定为最小解的极值点(非零点)，则满足的方程为

$$p_0 + p_1x_i + \cdots + p_mx_i^m = [f(x_i) \pm r](1 + q_1x_i + \cdots + q_kx_i^k)$$
$$i = 0, 1, \ldots, m + k + 1 \tag{5.13.5}$$

其中 ± 符号随交替极值点而交替变化。注意(5.13.5)式，在 $m + k + 2$ 个极值点处均满足，而(5.13.4)式仅在 $m + k + 1$ 个任意点处满足。原因是(5.13.5)式中的 r 也是未知量，故未知数共有 $m + k + 2$ 个。方程组有点非线性(对 r 而言)，但用第 9 章介绍的方法还是可求解的。

这样，只要给定最优有理函数的极点位置，即可解得系数及最大偏差。另外有一条可导出的称为"Remes(列梅兹)算法"[1]的定理，它讨论如何通过迭代过程收敛到这些位置。例如，下面是 **Remes 第二算法**的表述(稍微有些简化)：(1)求有 $m + k + 2$ 个极点 x_i 的初始有理函数(偏差不同)；(2)解(5.13.5)式求得新的有理系数及 r；(3)估算得到的 $R(x)$ 找出它的实际极点(与猜测值不一样)；(4)用具有相同符号最接近的极值点取代猜测值；(5)回到第二步并迭代直至收敛。在很多假定条件下，此方法收敛。Ralston[1]详细讨论了细节，包括如何找 x_i 的初始集。

下面让我们脱离常规，不用 Remes 算法。此算法内含的两种迭代(非线性集(5.13.5)的 r 和新得到 x_i 的集)要求过于苛刻，而且对于退化情况要求一套新的特殊逻辑设计。

当我们使用有理函数逼近时，目标通常非常实际：在某些内部循环中多次地反复求函数值，且要求加快求值速度。我们几乎从来不要求精确到机器精度的最后一位。假设使用的逼近式的误差有 $m + k + 2$ 个极点，偏差为因子 2。Remes 算法作为基础的定理保证，最优解的极点在因子 2 范围内——使上面极点下降，就会引起下面极点上升，直至都相等。这样，不严格逼近式实际上至少在最优解的一位有效位的范围内。

这已经足够了，特别当寻找一种求不严格逼近式的方法时，这就更好了。这种方法就是，由奇异值分解得到的超定线性方程的最小二乘方解(参见 2.6 节和 15.4 节)。按下面过程进行：首先，解(由最小二乘方)(5.13.3)式，不是求 x_i 的 $m + k + 1$ 个极值，而是求 x_i 的非常大的值。它的空间分布跟高阶切比雪夫多项式零点分布相似。这就是 $R(x)$ 的初始估计。第二步，对结果的偏差列成表，求平均绝对偏差，称为 r，然后将 r 固定，并对每点 x_i 的偏差量的符号分别选定为 ±，以求解(再次在最小二乘方意义下)(5.13.5)式。第三步，重复第二步几次。

算法隐含有一些 Remes 的正统算法：求解方程不是将偏差置向零，而是加减一些一致性的值。然而我们没有跟踪实际极点，而只是每次求解线性方程。还有一个方法是，解加权最小二乘方问题，其中权的选择是使最大偏差迅速减小的。

下面是一段实现算法的程序。注意，只在初始填写表 fs 时才调用一次函数 fn。很容易修改代码，以便在程序外面填写表。甚至坐标 xs 不必刚好就是我们用的那些。虽然在最优解的每个极点之间坐标选得不够多时，会使拟合质量受到影响。注意，输出有理系数的格式适合 5.3 节中程序 ratval 的调用。

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include "nr.h"
using namespace std;

void NR::ratlsq(DP fn(const DP), const DP a, const DP b, const int mm,
    const int kk, Vec_O_DP &cof, DP &dev)
以 cof[0..mm+kk]返回函数 fn 在区间(a,b)的有理函数逼近系数。输入量 mm 和 kk 分别表示分子和分母的次数。最大
逼近偏差绝对值返回为 dev。
{
    const int NPFAC=8,MAXIT=5;
    const DP BIG=1.0e30,PIO2=1.5707963267948966 19;
    int i,it,j,ncof,npt;
    DP devmax,e,hth,power,sum;

    ncof=mm+kk+1;
    npt=NPFAC*ncof;
    函数值的点数，即网格密度。
```

```
    Vec_DP bb(npt),coff(ncof),ee(npt),fs(npt),w(ncof),wt(npt),xs(npt);
    Mat_DP u(npt,ncof),v(ncof,ncof);
    dev=BIG;
    for (i=0;i<npt;i++) {                    数组装入求值点坐标和函数值。
        if (i < (npt/2)-1) {
            hth=PIO2*i/(npt-1.0);            每次末尾，用公式使得舍入灵敏度最小。
            xs[i]=a+(b-a)*SQR(sin(hth));
        } else {
            hth=PIO2*(npt-i)/(npt-1.0);
            xs[i]=b-(b-a)*SQR(sin(hth));
        }
        fs[i]=fn(xs[i]);
        wt[i]=1.0;                           后面迭代中调整数值以减小最大偏差。
        ee[i]=1.0;
    }
    e=0.0;
    for (it=0;it<MAXIT;it++) {               反复迭代。
        for (i=0;i<npt;i++) {                为最小二乘方拟合设置“设计”矩阵。
            power=wt[i];
            bb[i]=power*(fs[i]+SIGN(e,ee[i]));
            这里是关键思想:偏离为正,拟合 fn(x) + e;偏差为负,拟合 fn(x) - e。假设 e 是使逼近成为等波动偏差。
            for (j=0;j<mm+1;j++) {
                u[i][j]=power;
                power *= xs[i];
            }
            power = -bb[i];
            for (j=mm+1;j<ncof;j++) {
                power *= xs[i];
                u[i][j]=power;
            }
        }
        svdcmp(u,w,v);                       奇异值分解。
        在奇异或困难的情况下,这里修正奇异值 w[0..ncof-1],用零替换最小值。
        svbksb(u,w,v,bb,coff);
        devmax=sum=0.0;
        for (j=0;j<npt;j++) {                对偏离值列表，修正数值。
            ee[j]=ratval(xs[j],coff,mm,kk)-fs[j];
            wt[j]=fabs(ee[j]);               用数值突出最偏离的点。
            sum += wt[j];
            if (wt[j] > devmax) devmax=wt[j];
        }
        e=sum/npt;                           将e更新平均绝对值偏离。
        if (devmax <= dev) {                 只存储找到的最好系数集。
            for (j=0;j<ncof;j++) cof[j]=coff[j];
            dev=devmax;
        }
        cout << " ratlsq iteration= " << it;
        cout << " max error= " << setw(10) << devmax << endl;
    }
}
```

图 5.13.1 描述了在区间$(0,\pi)$内求当 $m=k=4$ 时，函数 $f(x)=\cos x/(1+e^x)$ 的有理数拟合，用程序 ratlsq 迭代前 5 次的不同情况。可看出经过第一次迭代，结果已经差不多和最优解一样好，迭代收敛并不是如图形显示的那样，其实收敛最好的是第二次迭代(偏差最小)。程序 ratlsq 返回迭代最好的结果，并非就是最后一次，迭代超过 5 次并没有多大好处。

参考文献和深入阅读

Ralston, A. and Wilf, H.S. 1960, *Mathematical Methods for Digital Computers* (New York: Wiley), Chapter 13. [1]

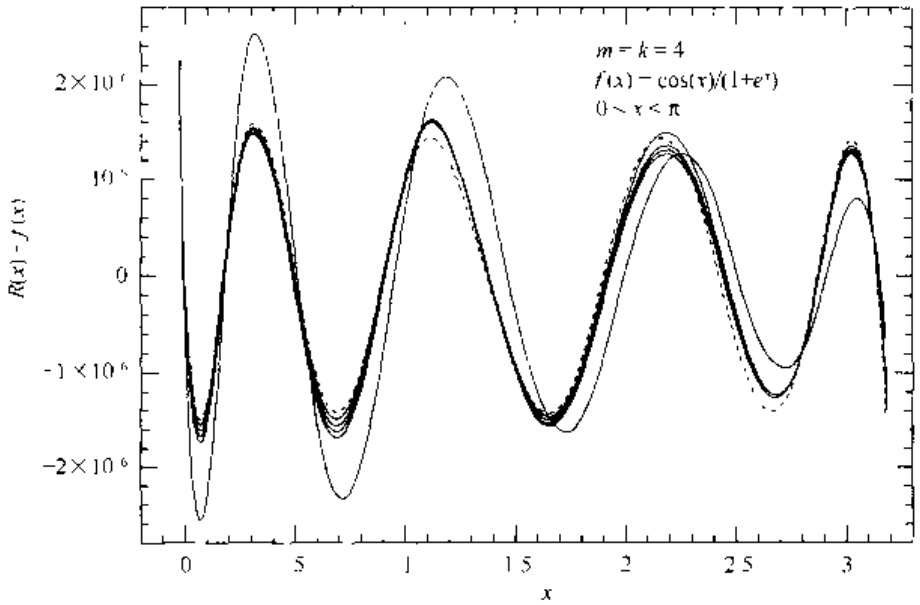


图 5.13.1　对于任意试验问题，实曲线显示程序 ratlsq 5 次连续迭代的偏差 $r(x)$。此算法不能精确收敛到最小最大解（点曲线表示），但是，第一次迭代后，差异只是精度的最低有效位的很小部分

5.14　线积分求函数值

编写计算机程序时，选择的算法并非必须是效率最高、实现最优秀，或者是执行速度最快的。实际上，可能选择的是易于实现、通用而且便于检测的算法。

有时可能需要求几次或几千次某个特殊函数值，也许是复变量的复数函数，函数可能有很多不同参数或渐近区域，或者两种情况都存在。用通常的方法（级数、连分式、有理函数逼近、递推关系式等）设计出的程序是一系列条件测试及对不同情况分支的大混合。程序执行时可能效率很高，但从一开始起通常不是最简便直接的方法。

一种极其通用但又不同的方法是，对一个函数定义的微分方程直接积分——对每个指定函数值从头开始积分——必要时，在复平面内沿一条路径积分。初看起来，这样做似乎是没有必要的，但实际情况正好相反。

作为一个特例，考察复超几何函数 ${}_2F_1(a,b,c;z)$，根据超几何级数的解析连续性定义有

$$
\begin{aligned}
{}_2F_1(a,b,c;z) = 1 + \frac{ab}{c}\frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!} + \cdots \\
+ \frac{a(a+1)\ldots(a+j-1)b(b+1)\ldots(b+j-1)}{c(c+1)\ldots(c+j-1)}\frac{z^j}{j!} + \cdots
\end{aligned}
\tag{5.14.1}
$$

级数仅在单位圆 $|z|<1$ 内才收敛（参见本节参考文献[1]），但我们感兴趣的不只是这个区域。

超几何函数 ${}_2F_1$ 是超几何差分方程，写为

$$
z(1-z)F'' = abF - [c-(a+b+1)z]F' \tag{5.14.2}
$$

的一个解（在原点处此解是规则的）。上式中撇号表示 d/dz。可以看出方程在 $z=0,1$ 和 ∞ 处有规则奇异点。因为所求解在 $z=0$ 处规则，这样 1 和 ∞ 成为分支点。若想使 ${}_2F_1$ 成为单值函数，则必须求一个联系两个点的切断分支。通常切断位置在正实轴从 1 到 ∞ 之间，在某些其他场合下可能需要另外的处理方法。

此处的方法包括几个在第 16 章里将详细介绍的常微分方程组开始积分的程序集。现在只需一个高级“黑箱”程序，从用自变量值表示的初始条件开始积分，直到此变量用另一值表示的终止条件为止，并自动调整积分步长以保证结果精度。此程序为 odeint，程序中由复杂的 Bulirsch-Stoer 方法计算步长。

假设已知某点 z_0 处的函数 F 及其导数 F' 的值，现在需求函数 F 在复平面的某个其他点处的值。连接两点的直线用参数表示为

$$z(s) = z_0 + s(z_1 - z_0) \tag{5.14.3}$$

其中 s 为实参数。微分方程(5.14.2)可改写为两个一阶方程：

$$\begin{aligned} \frac{dF}{ds} &= (z_1 - z_0)F' \\ \frac{dF'}{ds} &= (z_1 - z_0)\left(\frac{abF - [c-(a+b+1)z]F'}{z(1-z)}\right) \end{aligned} \tag{5.14.4}$$

从 $s=0$ 到 $s=1$ 的积分。F 和 F' 被视为两个独立的复变量。撇号表示 d/dz 可以忽略，因为它可以呈现为(5.14.4)式中第一个方程的结果。此外，(5.14.4)式的实部和虚部决定了4个具有自变量 s 的实微分方程。式中右边的复数运算可视为耦合这4个部分的一种简便写法。程序 odeint 输入正是根据这种思想，因为它既不知道复数函数，也不知道复自变量。

剩下的任务是确定从哪里开始，沿复平面内的哪条路径，达到任一点 z。此时，还要考虑函数奇异点以及采用的分支切断点。图5.14.1表示了我们采用的策略。当 $|z| \leq 1/2$ 时，(5.14.1)式的级数一般收敛很快，此时直接使用它就可以了。其他情况下，沿从一个起点 $(\pm 1/2, 0)$ 或 $(0, \pm 1/2)$ 开始的直线进行积分。选前一个起点分别对应于 $0 < \mathrm{Re}(z) < 1$ 及 $\mathrm{Re}(z) < 0$ 的情况。选后一个起点则用于在切断分支点的上或下 $\mathrm{Re}(z) > 1$。从实轴开始的原因是为了避免太过于靠近 $z=1$ 的奇异点(参见图5.14.1)。分支切断点的位置是根据所采用的策略在 $\mathrm{Re}(z) > 1$ 时从不越过实轴进行积分这一事实来确定的。

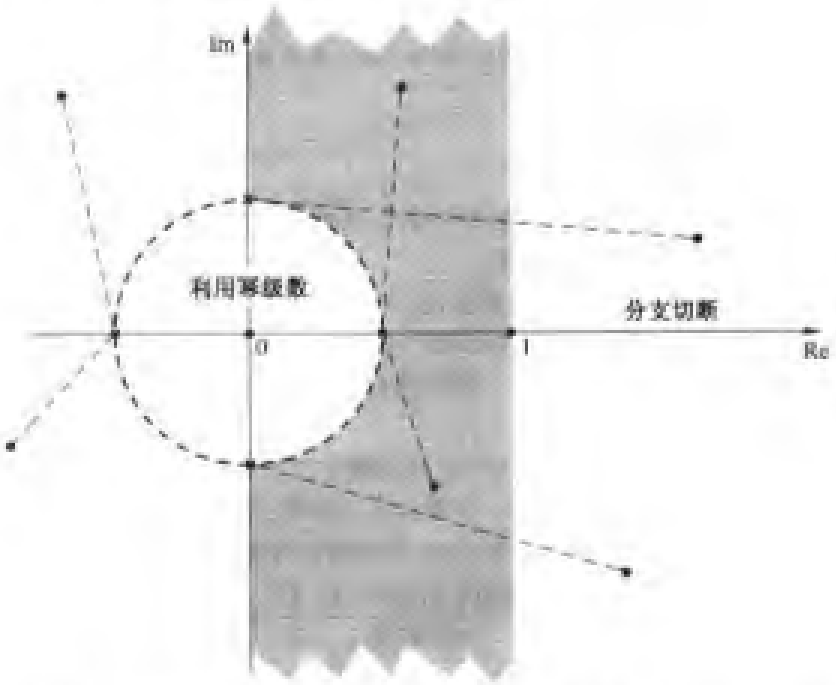


图5.14.1 表示超几何函数奇异点、分支切断点，以及几条虚线是从 $|z| = 1/2$ (这里级数收敛很快)到其他点的几条积分路径的复平面

此算法的实现参见6.12节中的程序 hypgeo。

基于上面描述过程的实施有各种衍生形式，都易于编程实现。若成功调用的 z 值互相靠近(a、b 和 c 值相同)，则在每次调用时，可存储状态向量(F, F')及 z 的相应值，然后在下一次调用时，用这些值作为起始值。增加的积分可能只需一至两步。应当避免无意中越过分支切断线进行积分：函数值可能是"正确的"，但不是所求的。

或者，也许想沿越过实轴 $\mathrm{Re}\ z > 1$ 的折线路径作积分，把它作为一种移动分支切断点的方法。例如，若想从(0,1/2)到(3/2,1/2)积分，然后，从这里到任何 $\mathrm{Re}\ z > 1$ 的地方——$\mathrm{Im}\ z$ 的符号正负均可(例如，若想用迭代方法求一个函数的根，而不想在分支点处沿不同路径的附近值求积分。若正是这样，在求根时会发现不连续函数值，而且不收敛)。

在任何情况下，都要注意以下丢失数值精度的结果：积分路径通过大函数值区域而到达函数值很小的终止（对超几何函数，一种特殊情形是：a 和 b 都取大的正值，而 $c \geq 1$，$x \geq 1$）。此时，需找一条较好的折线路径

在复平面上，通过对微分方程积分来求函数值的方法，也可用于其他特殊函数。例如，复贝塞尔函数、艾里（Airy）函数、Coulomb 波形函数及韦伯（Weber）函数，这些都是**合流超几何函数**的特例，其微分方程与上面使用过的类似（见本节参考文献[1]中 13.6 节的特例表）。合流超几何函数在有限 z 值处没有奇异点，这样积分就容易些。但是，无穷远处的本质奇异性，意味着沿某些路径对某些参数，积分具有很强的振荡性或指数衰减性，这样积分就更困难了，所以有时候分不同情况讨论。

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [1]

第6章 特殊函数

6.0 引言

特殊函数并没有什么特别之处，只是因为某些权威人士或教科书的作者这么称呼。特殊函数又称为**高等超越函数**、或称**数学物理函数**(此概念也经常出现在其他领域里)，或称**满足某些频繁出现的二阶微分方程的函数**(但并不是所有特殊函数都如此)。许多人简单地称其为"有用函数"而不作深究，本章选择了一些函数进行讨论。

一些很好的商业软件包，如 NAG 或 IMSL 就包含许多特殊函数的程序。这种程序只对那些不想知道其内部运行情况的用户是很有用的。通常这种"黑箱"里面状态杂乱，程序中分支很多，依调用参数的不同，所使用方法也不完全不同。黑箱能够(而且也应该能够)仔细地控制精度，以便在各种条件状况下达到所要求的精度。

我们给出的示例将不太讲究这些，一是因为我们想阐述第 5 章的方法，二是因为希望读者能理解所提供程序的内部运行情况。一些程序中含有精度参数，可按要求设置其大小，其他一些程序中(尤其是多项式拟合时)，只给出某种一般可行(通常 6 位左右有效数字)的精度。不能保证程序黑箱是完美的，只希望读者在程序中遇到问题时，能根据程序提供的材料，诊断并修正问题。

总之，本章的特殊函数可直接使用，但了解其内部工作情况也是必要的。

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York) [full of useful numerical approximations to a great variety of functions].

IMSL Sfun/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042).

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.), Chapter S.

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley).

Hastings, C. 1955, *Approximations for Digital Computers* (Princeton: Princeton University Press).

Luke, Y.L. 1975, *Mathematical Functions and Their Approximations* (New York: Academic Press).

6.1 Γ 函数、*B*函数、阶乘、二项式系数

Γ 函数(伽马函数)定义为积分形式

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt \tag{6.1.1}$$

z 若为整数，Γ 函数就是熟知的阶乘，但必须偏移 1，即

$$n! = \Gamma(n+1) \tag{6.1.2}$$

Γ 函数满足递推关系式

$$\Gamma(z+1) = z\Gamma(z) \tag{6.1.3}$$

若参数 $z>1$，或者更一般地在复半平面 $\mathrm{Re}(z)>1$ 上函数已知，则可由反射公式得到 $z<1$ 或 $\mathrm{Re}(z)<1$ 时，

$$\Gamma(1-z) = \frac{\pi}{\Gamma(z)\sin(\pi z)} = \frac{\pi z}{\Gamma(1+z)\sin(\pi z)} \tag{6.1.4}$$

注意 $\Gamma(z)$在 $z=0$ 及所有负整数值处有极点。

计算函数 $\Gamma(z)$的数值方法有多种，但都不如由 Lanczos[1]导出的近似公式清晰。这只对 Γ 函数完全有

效。这里不打算去导出近似公式，只给出其结果：对某一选定的 γ 和 N，以及一组系数 $c_1, c_2, \ldots, c_N$，Γ 函数由下式给出：

$$\Gamma(z+1) = (z+\gamma+\tfrac{1}{2})^{z+\frac{1}{2}} e^{-(z+\gamma+\frac{1}{2})} \times \sqrt{2\pi}\left[c_0 + \frac{c_1}{z+1} + \frac{c_2}{z+2} + \cdots + \frac{c_N}{z+N} + \epsilon\right] \quad (z>0) \tag{6.1.5}$$

可以看出上式思路类似于斯特林(Stirling)近似公式，只是考察到左半复平面的一些一阶极点而进行了一些修正。常数 c_0 很显然应等于 1。误差项用 ϵ 表示。对 $\gamma=5, N=6$，及一组 c 值，误差项 $|\epsilon| < 2\times10^{-10}$。读者一定会对下面事实留有深刻印象：即(在相同参数下)(6.1.5)式和 ϵ 的上下限**在 Re(z) > 0 的复半平面的任何地方**同样适用于复 Γ 函数。

计算 $\ln\Gamma(x)$ 比 $\Gamma(x)$ 更好，因为后者在许多计算机浮点表示中，对一般 x 值也会发生溢出。通常凡计算中用到 Γ 函数处，一般总是用 $\Gamma(x)$ 去除以另一个很大的数，这样所得结果就是普通大小的值了。这种运算可转化为对数减法运算。应用(6.1.5)式，只需调用 2 次对数函数和 25 次左右的算术运算，立即可得到 Γ 函数对数值。这并不比某些函数，如 $\sin x$ 或 e^x 的计算困难。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::gammln(const DP xx)
返回值 ln[Γ(xx)]，xx>0。
{
    int j;
    DP x,y,tmp,ser;
    static const DP cof[6]={76.18009172947146,-86.50532032941677,
        24.01409824083091,-1.231739572450155,0.1208650973866179e-2,
        -0.5395239384953e-5};

    y=x=xx;
    tmp=x+5.5;
    tmp -= (x+0.5)*log(tmp);
    ser=1.000000000190015;
    for (j=0;j<6;j++) ser += cof[j]/++y;
    return -tmp+log(2.5066282746310005*ser/x);
}
```

怎样编写求阶乘函数 $n!$ 的程序呢？一般情况下，小整数值的时候去调用阶乘函数(数值大后总是溢出)。多数应用情况下，同一个整数值需多次调用。若每次需要阶乘都调用 exp (gammln(n + 1.0))，这样就太浪费机时。最好返本求源，非必要时不要调用 gammln。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::factrl(const int n)
按浮点数返回 n! 的值。
{
    static int ntop=4;
    static DP a[33]={1.0,1.0,2.0,6.0,24.0};        仅按要求填表。
    int j;

    if (n < 0) nrerror("Negative factorial in routine factrl");
    if (n > 32) return exp(gammln(n+1.0));
    需要大于表长的数，事实上多数计算机上如此大的数一般会溢出，但不妨一试。
    while (ntop<n) {                               填表直到设置的值。
        j=ntop++;
        a[ntop]=a[j]*ntop;
    }
    return a[n];
}
```

n值较小时,factrl结果将是精确的,因为在任何计算机上,浮点数乘以一个小整数的结果总是精确的。若要用阶乘对数,则这种精确性就不能保证了,但对于二项式系数,必须精确地求得阶乘,因为在二项式系数本身溢出以前,二项式中的阶乘早就已经溢出了。

二项式系数定义为

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad 0 \leqslant k \leqslant n \tag{6.1.6}$$

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::bico(const int n, const int k)
按浮点数返回二项式系数(n k)的值。
{
    return floor(0.5+exp(factln(n)-factln(k)-factln(n-k)));
    n和k较小时, floor函数清除舍入误差。
}
```

这里用到

```
#include "nr.h"

DP NR::factln(const int n)
返回 ln(n!)。
{
    static DP a[101];                 静态数组自动初始化为零。

    if (n < 0) nrerror("Negative factorial in routine factln");
    if (n <= 1) return 0.0;
    if (n <= 100)                     在表内。
        return (a[n] != 0.0 ? a[n] : (a[n]=gammln(n+1.0)));
    else return gammln(n+1.0);        超出表。
}
```

求解问题时常需用到二项式系数,比较好的方法是利用递推公式,例如

$$\begin{aligned} \binom{n+1}{k} &= \frac{n+1}{n-k+1}\binom{n}{k} = \binom{n}{k} + \binom{n}{k-1} \\ \binom{n}{k+1} &= \frac{n-k}{k+1}\binom{n}{k} \end{aligned} \tag{6.1.7}$$

暂时放下整值变量的组合函数,最后讨论一下Beta函数(β函数,下面表示为B函数)

$$B(z,w) = B(w,z) = \int_0^1 t^{z-1}(1-t)^{w-1}dt \tag{6.1.8}$$

与Γ函数的关系为

$$B(z,w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \tag{6.1.9}$$

因此

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::beta(const DP z, const DP w)
返回 B 函数 B(z,w)的值。
{
    return exp(gammln(z)+gammln(w)-gammln(z+w));
}
```

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 6.

Lanczos, C. 1964, *SIAM Journal on Numerical Analysis*, ser. B, vol. 1, pp. 86–96. [1]

6.2 不完全Γ函数、误差函数、χ^2概率函数、累积泊松函数

不完全Γ函数定义为

$$P(a,x) \equiv \frac{\gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)}\int_0^x e^{-t}t^{a-1}dt \qquad (a>0) \tag{6.2.1}$$

它的极限值为

$$P(a,0)=0 \qquad 和 \qquad P(a,\infty)=1 \tag{6.2.2}$$

不完全Γ函数 $P(a,x)$是单调的，当 a 大于1时，在大约以 $a-1$ 为中心，宽为$\sqrt{a}$的区域内，从"接近零"上升到"接近1"(参见图6.2.1)。

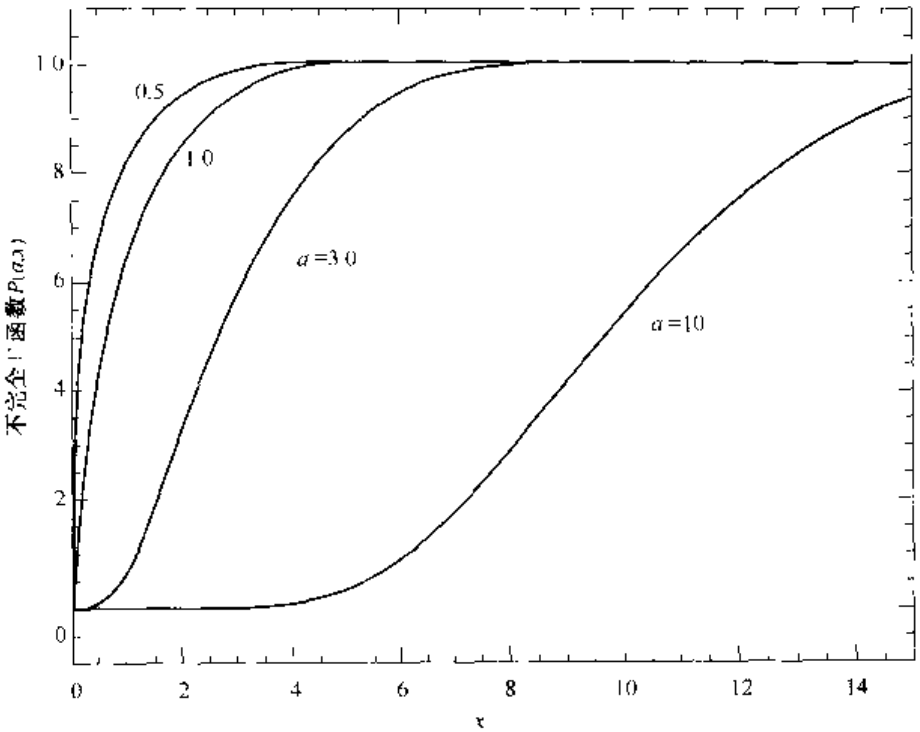


图6.2.1 四个 a 值的不完全Γ函数 $P(a,x)$

$P(a,x)$的互补形式也统称为不完全Γ函数，

$$Q(a,x) \equiv 1-P(a,x) \equiv \frac{\Gamma(a,x)}{\Gamma(a)} \equiv \frac{1}{\Gamma(a)}\int_x^\infty e^{-t}t^{a-1}dt \qquad (a>0) \tag{6.2.3}$$

它的极限值为

$$Q(a,0)=1 \qquad 和 \qquad Q(a,\infty)=0 \tag{6.2.4}$$

记号 $P(a,x)$，$\gamma(a,x)$和 $\Gamma(a,x)$是标准记号，而记号$Q(a,x)$是本书专用记号。

$\gamma(a,x)$有如下级数展开式：

$$\gamma(a,x)=e^{-x}x^a\sum_{n=0}^{\infty}\frac{\Gamma(a)}{\Gamma(a+1+n)}x^n \tag{6.2.5}$$

不必每次对 n 重新计算一次 $\Gamma(a+1+n)$，最好利用(6.1.3)式及以前的系数。

$\Gamma(a,x)$的连分式展开为

$$\Gamma(a,x)=e^{-x}x^{a}\left(\frac{1}{x+}\ \frac{1-a}{1+}\ \frac{1}{x+}\ \frac{2-a}{1+}\ \frac{2}{x+}\cdots\right)\qquad(x>0)\tag{6.2.6}$$

用(6.2.6)式偶项部分计算较好,它收敛快二倍(参见5.2节):

$$\Gamma(a,x)=e^{-x}x^{a}\left(\frac{1}{x+1-a-}\ \frac{1\cdot(1-a)}{x+3-a-}\ \frac{2\cdot(2-a)}{x+5-a-}\ \cdots\right)\qquad(x>0)\tag{6.2.7}$$

可以发现(6.2.5)式当 x 小于 $a+1$ 时收敛很快,而(6.2.6)式或(6.2.7)式,当 x 大于 $a+1$ 时收敛很快。每种情况下,均需到几倍$\sqrt{a}$的项数后才收敛。在 $x=a$ 附近,不完全 Γ 函数变化最剧烈。因此(6.2.5)式及(6.2.7)式一起可对任何正数 a 及 x 求函数值。附加的优点就是,再也不必通过两个几乎相等的数相减来求零附近的函数值。返回 $P(a,x)$和$Q(a,x)$的高阶函数是:

```
#include "nr.h"

DP NR::gammp(const DP a, const DP x)
返回不完全Γ函数 P(a,x)的值。
{
    DP gamser,gammcf,gln;

    if (x < 0.0 || a <= 0.0)
        nrerror("Invalid arguments in routine gammp");
    if (x < a+1.0) {                用级数表示。
        gser(gamser,a,x,gln);
        return gamser;
    } else {                        用连分数表示,
        gcf(gammcf,a,x,gln);
        return 1.0-gammcf;          并取互补形式。
    }
}
```

```
#include "nr.h"

DP NR::gammq(const DP a, const DP x)
返回不完全Γ函数Q(a,x)≡1-P(a,x)。
{
    DP gamser,gammcf,gln;

    if (x < 0.0 || a <= 0.0)
        nrerror("Invalid arguments in routine gammq");
    if (x < a+1.0) {                用级数表示,
        gser(gamser,a,x,gln);
        return 1.0-gamser;          并取互补形式。
    } else {                        用连分数表示。
        gcf(gammcf,a,x,gln);
        return gammcf;
    }
}
```

级数和连分式都将变量 gln 置为 $\ln\Gamma(a)$,原因是使用户方便地将上面两个程序修改成求 $\gamma(a,x)$和$\Gamma(a,x)$,而不只是求 $P(a,x)$和$Q(a,x)$。参见(6.2.1)式和(6.2.3)式。

实现(6.2.5)式及(6.2.7)式的函数 gser 和 gcf 为

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

void NR::gser(DP &gamser, const DP a, const DP x, DP &gln)
用级数表达式如 gamser 求不完全Γ函数 P(a,x)值并返回。同时由 gln 返回 lnΓ(a)。
```

```
{
    const int ITMAX=100;
    const DP EPS=numeric_limits<DP>::epsilon();
    int n;
    DP sum,del,ap;

    gln=gammln(a);
    if (x <= 0.0) {
        if (x < 0.0) nrerror("x less than 0 in routine gser");
        gamser=0.0;
        return;
    } else {
        ap=a;
        del=sum=1.0/a;
        for (n=0;n<ITMAX;n++) {
            ++ap;
            del *= x/ap;
            sum += del;
            if (fabs(del) < fabs(sum)*EPS) {
                gamser=sum*exp(-x+a*log(x)-gln);
                return;
            }
        }
        nrerror("a too large, ITMAX too small in routine gser");
        return;
    }
}

#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

void NR::gcf(DP &gammcf, const DP a, const DP x, DP &gln)
```

由连分数表示形式 gammcf 求不完全 Γ 函数 $Q(a,x)$ 并返回。同时由 gln 返回 $\ln\Gamma(a)$。

```
{
    const int ITMAX=100;
    const DP EPS=numeric_limits<DP>::epsilon();
    const DP FPMIN=numeric_limits<DP>::min()/EPS;
    ITMAX is the maximum allowed number of iterations; EPS is the relative accuracy; FPMIN is
    a number near the smallest representable floating-point number.
    int i;
    DP an,b,c,d,del,h;

    gln=gammln(a);
    b=x+1.0-a;                          这些设置为了用 $b_0=0$ 的修正 Lentz 方法
    c=1.0/FPMIN;                        (参见 5.2 节)来计算连分式。
    d=1.0/b;
    h=d;
    for (i=1;i<=ITMAX;i++) {            迭代到收敛。
        an = -i*(i-a);
        b += 2.0;
        d=an*d+b;
        if (fabs(d) < FPMIN) d=FPMIN;
        c=b+an/c;
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        del=d*c;
        h *= del;
        if (fabs(del-1.0) <= EPS) break;
    }
    if (i > ITMAX) nrerror("a too large, ITMAX too small in gcf");
    gammcf=exp(-x+a*log(x)-gln)*h;      将因子放在前面。
}
```

6.2.1 误差函数

误差函数及其互补形式是不完全 Γ 函数的特例，故可由前述过程方便地推得。其定义式为

$$\mathrm{erf}(x)=\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt \tag{6.2.8}$$

和

$$\mathrm{erfc}(x) \equiv 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt \tag{6.2.9}$$

它们有如下极限值和对称性质：

$$\mathrm{erf}(0) = 0 \qquad \mathrm{erf}(\infty) = 1 \qquad \mathrm{erf}(-x) = -\mathrm{erf}(x) \tag{6.2.10}$$

$$\mathrm{erfc}(0) = 1 \qquad \mathrm{erfc}(\infty) = 0 \qquad \mathrm{erfc}(-x) = 2 - \mathrm{erfc}(x) \tag{6.2.11}$$

与不完全Γ函数的关系为

$$\mathrm{erf}(x) = P\left(\frac{1}{2}, x^2\right) \qquad (x \geqslant 0) \tag{6.2.12}$$

和

$$\mathrm{erfc}(x) = Q\left(\frac{1}{2}, x^2\right) \qquad (x \geqslant 0) \tag{6.2.13}$$

为了避免与C语言库中函数名冲突，所以程序名中多加进一个“f”：

```
#include "nr.h"

DP NR::erff(const DP x)
返回误差函数 erf(x)。
{
    return x < 0.0 ? -gammp(0.5,x*x) : gammp(0.5,x*x);
}


#include "nr.h"

DP NR::erffc(const DP x)
返回互补误差函数 erfc(x)。
{
    return x < 0.0 ? 1.0+gammp(0.5,x*x) : gammq(0.5,x*x);
}
```

如果愿意，还可以很容易补救 erff 和 erffc 中较差的效率，即调用 gammp 或 gammq 后，$\Gamma(0.5) = \sqrt{\pi}$也就不必计算了。首先，需要考虑下面一段程序，它是基于对函数形式猜测的切比雪夫拟合。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::erfcc(const DP x)
返回互补误差函数 erfc(x)，其相对误差处处小于 1.2×10^-7。
{
    DP t,z,ans;

    z=fabs(x);
    t=1.0/(1.0+0.5*z);
    ans=t*exp(-z*z-1.26551223+t*(1.00002368+t*(0.37409196+t*(0.09678418+
        t*(-0.18628806+t*(0.27886807+t*(-1.13520398+t*(1.48851587+
        t*(-0.82215223+t*0.17087277)))))))));
    return (x >= 0.0 ? ans : 2.0-ans);
}
```

还有一些不完全Γ函数的特例，它们带有两个变量。

6.2.2 累积泊松概率函数

$P_x(<k)$表示**累积泊松(Poisson)概率**函数，其中 x 为正数，整数 $k \geqslant 1$。若期望均值为 x，则此函数定义为泊松随机事件发生次数介于0到 $k-1$(包括0和 $k-1$ 在内)的概率。其极限值为

$$P_x(<1) = e^{-x} \qquad P_x(<\infty) = 1 \tag{6.2.14}$$

与不完全Γ函数的简单关系为

$$P_x(<k) = Q(k, x) = \mathbf{gammq}(k, x) \tag{6.2.15}$$

6.2.3 χ^2 概率函数

$P(\chi^2|\nu)$定义为正确模型下观察 chi 平方值小于值χ^2 的概率(将在第 15 章讨论这一函数的用处)。其互补 $Q(\chi^2|\nu)$为正确模型下观察 chi 平方值随机超过χ^2 值的概率。两式中 ν 为整数,表示自由度数。函数有极限值

$$P(0|\nu)=0 \qquad P(\infty|\nu)=1 \tag{6.2.16}$$

$$Q(0|\nu)=1 \qquad Q(\infty|\nu)=0 \tag{6.2.17}$$

与不完全 Γ 函数有如下关系:

$$P(\chi^2|\nu)=P\left(\frac{\nu}{2},\frac{\chi^2}{2}\right)=\mathrm{gammp}\left(\frac{\nu}{2},\frac{\chi^2}{2}\right) \tag{6.2.18}$$

$$Q(\chi^2|\nu)=Q\left(\frac{\nu}{2},\frac{\chi^2}{2}\right)=\mathrm{gammq}\left(\frac{\nu}{2},\frac{\chi^2}{2}\right) \tag{6.2.19}$$

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapters 6, 7, and 26.

Pearson, K. (ed.) 1951, *Tables of the Incomplete Gamma Function* (Cambridge: Cambridge University Press).

6.3 指数积分

指数积分的标准定义为

$$E_n(x)=\int_1^\infty \frac{e^{-xt}}{t^n}dt, \qquad x>0, \quad n=0,1,\ldots \tag{6.3.1}$$

用积分的主值定义的函数

$$\mathrm{Ei}(x)=-\int_{-x}^{\infty}\frac{e^{-t}}{t}dt=\int_{-\infty}^{x}\frac{e^t}{t}dt, \qquad x>0 \tag{6.3.2}$$

也称为指数积分。注意,由于解析连续性,$\mathrm{Ei}(-x)$等于$-E_1(x)$。

函数 $E_n(x)$是不完全 Γ 函数的特例:

$$E_n(x)=x^{n-1}\Gamma(1-n,x) \tag{6.3.3}$$

这样可用类似的方法求其值。整理(6.2.6)式写成连分式形式,对所有 $x>0$ 都收敛:

$$E_n(x)=e^{-x}\left(\frac{1}{x+}\,\frac{n}{1+}\,\frac{1}{x+}\,\frac{n+1}{1+}\,\frac{2}{x+}\cdots\right) \tag{6.3.4}$$

它的偶项形式收敛更快:

$$E_n(x)=e^{-x}\left(\frac{1}{x+n-}\,\frac{1\cdot n}{x+n+2-}\,\frac{2(n+1)}{x+n+4-}\cdots\right) \tag{6.3.5}$$

仅当 $x\geq 1$ 时,连分式形式才很快收敛,这时才有实用价值。对 $0<x\leq 1$,可用级数表示

$$E_n(x)=\frac{(-x)^{n-1}}{(n-1)!}[-\ln x+\psi(n)]-\sum_{\substack{m=0\\ m\neq n-1}}^{\infty}\frac{(-x)^m}{(m-n+1)m!} \tag{6.3.6}$$

参量$\psi(n)$为双 Γ 函数,对整数变量定义为

$$\psi(1) = -\gamma, \qquad \psi(n) = -\gamma + \sum_{m=1}^{n-1} \frac{1}{m} \tag{6.3.7}$$

其中 $\gamma = 0.5772156649\ldots$,即欧拉(Euler)常数。按 x 的幂次递增顺序求(6.3.6)式的值:

$$\begin{aligned} E_n(x) = &-\left[\frac{1}{(1-n)} - \frac{x}{(2-n)\cdot 1} + \frac{x^2}{(3-n)(1\cdot 2)} - \cdots + \frac{(-x)^{n-2}}{(-1)(n-2)!}\right] \\ &+ \frac{(-x)^{n-1}}{(n-1)!}[-\ln x + \psi(n)] - \left[\frac{(-x)^n}{1\cdot n!} + \frac{(-x)^{n+1}}{2\cdot(n+1)!} + \cdots\right] \end{aligned} \tag{6.3.8}$$

$n=1$ 时则省去第一个方括号。这种方法的优点是,n 很大时级数项在到达含 $\psi(n)$ 项之前就开始收敛。这样就只需对较小的 n,如 $n \le 20-40$ 找一个算法求 $\psi(n)$。虽然查找表方法效率稍高些,但我们仍用(6.3.7)式。

Amos[1]详细讨论了求(6.3.8)式时截尾误差的影响,并且给出一个相当精巧的结束条件。但我们发现,若仅仅在加的最后一项小于允许值时停止,效果也不错。

两种特殊情况需分别处理:

$$\begin{aligned} E_0(x) &= \frac{e^{-x}}{x} \\ E_n(0) &= \frac{1}{n-1}, \qquad n > 1 \end{aligned} \tag{6.3.9}$$

下面给出的程序 expint 允许以精度 EPS 快速计算 $E_n(x)$,EPS 达到机器浮点数允许的字长范围内。若仍想更精确的结果,则只能在欧拉常数后面再增添几位数字。Wrench[2]给出了前 328 位数字。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

DP NR::expint(const int n, const DP x)
计算指数积分 E_n(x)。
{
    const int MAXIT=100;
    const DP EULER=0.577215664901533;
    const DP EPS=numeric_limits<DP>::epsilon();
    const DP BIG=numeric_limits<DP>::max()*EPS;
    其中,MAXIT 是允许的最大迭代次数;EULER 是欧拉常数 γ;EPS 是期望的相对误差,不小于机器精度;BIG 是接近
    最大可表示的浮点数。
    int i,ii,nm1;
    DP a,b,c,d,del,fact,h,psi,ans;

    nm1=n-1;
    if (n < 0 || x < 0.0 || (x==0.0 && (n==0 || n==1)))
    nrerror("bad arguments in expint");
    else {
    if (n == 0) ans=exp(-x)/x;                特殊情况。
    else {
        if (x == 0.0) ans=1.0/nm1;            另一种特殊情况。
        else {
            if (x > 1.0) {                    Lentz算法(参见 5.2 节)。
                b=x+n;
                c=BIG;
                d=1.0/b;
                h=d;
                for (i=1;i<=MAXIT;i++) {
                    a = -i*(nm1+i);
                    b += 2.0;
                    d=1.0/(a*d+b);            分母不能为零。
                    c=b+a/c;
                    del=c*d;
                    h *= del;
                    if (fabs(del-1.0) <= EPS) {
                        ans=h*exp(-x);
```

```
                        return ans;
                    }
                }
                nrerror("continued fraction failed in expint");
            } else {                                求级数值。
                ans = (nm1!=0 ? 1.0/nm1 : -log(x)-EULER);      设置第一项。
                fact=1.0;
                for (i=1;i<=MAXIT;i++) {
                    fact *= -x/i;
                    if (i != nm1) del = -fact/(i-nm1);

                    else {
                        psi = -EULER;           计算ψ(n)。
                        for (ii=1;ii<=nm1;ii++) psi += 1.0/ii;
                        del=fact*(-log(x)+psi);
                    }
                    ans += del;
                    if (fabs(del) < fabs(ans)*EPS) return ans;
                }
                nrerror("series failed in expint");
            }
        }
    }
}
return ans;
}
```

求 Ei 值较好的方法是，x 比较小时用幂级数，而 x 较大时用渐近级数。幂级数为

$$\mathrm{Ei}(x) = \gamma + \ln x + \frac{x}{1\cdot 1!} + \frac{x^2}{2\cdot 2!} + \cdots \tag{6.3.10}$$

其中 γ 为欧拉常数。渐近级数是

$$\mathrm{Ei}(x) \sim \frac{e^x}{x}\left(1 + \frac{1!}{x} + \frac{2!}{x^2} + \cdots\right) \tag{6.3.11}$$

使用渐近级数的下限 x 大约为 $|\ln \mathrm{EPS}|$，其中 EPS 是要求的相对误差。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

DP NR::ei(const DP x)
x>0时计算指数函数 Ei(x)。
{
    const int MAXIT=100;
    const DP EULER=0.577215664901533;
    const DP EPS=numeric_limits<DP>::epsilon();
    const DP FPMIN=numeric_limits<DP>::min()/EPS;
    其中 EULER 是欧拉常数 γ。MAXIT 是允许的最大迭代数。FPMIN 是接近最小可表示的浮点数。EPS 是相对误差，
    或者在 x=0.3725 处，Ei 接近零的绝对误差。
    int k;
    DP fact,prev,sum,term;

    if (x <= 0.0) nrerror("Bad argument in ei");
    if (x < FPMIN) return log(x)+EULER;        特殊情况，防止因为下溢导致收敛判断失败。
    if (x <= -log(EPS)) {
        sum=0.0;                               用幂级数。
        fact=1.0;
        for (k=1;k<=MAXIT;k++) {
            fact *= x/k;
            term=fact/k;
            sum += term;
            if (term < EPS*sum) break;
        }
        if (k > MAXIT) nrerror("Series failed in ei");
        return sum+log(x)+EULER;
    } else {                                   用渐近级数。
        sum=0.0;                               从第二项开始。
        term=1.0;
```

```
            for (k=1;k<=MAXIT;k++) {
                prev=term;
                term *= k/x;
                if (term < EPS) break;
                因为最后一项大于 1,term 本身近似于相对误差。
                if (term < prev) sum += term;      继续收敛: 添加新的项。
                else {
                    sum -= prev;                   发散: 减去前一项并退出。
                    break;
                }
            }
            return exp(x)*(1.0+sum)/x;
        }
    }
```

参考文献和深入阅读

Stegun, I.A., and Zucker, R. 1974, *Journal of Research of the National Bureau of Standards*, vol. 78B, pp. 199–216; 1976, *op. cit.*, vol. 80B, pp. 291–311.

Amos D.E. 1980, *ACM Transactions on Mathematical Software*, vol. 6, pp. 365–377 [1]; also vol. 6, pp. 420–428.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 5.

Wrench J.W. 1952, *Mathematical Tables and Other Aids to Computation*, vol. 6, p. 255. [2]

6.4 不完全B函数、学生分布、F分布、累积二项式分布

不完全 B 函数(Beta 函数)定义为

$$I_x(a,b) \equiv \frac{B_x(a,b)}{B(a,b)} \equiv \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1}dt \qquad (a,b>0) \tag{6.4.1}$$

有限定值

$$I_0(a,b)=0 \qquad I_1(a,b)=1 \tag{6.4.2}$$

及对称关系

$$I_0(a,b)=0 \qquad I_1(a,b)=1 \tag{6.4.3}$$

若 a 和 b 远大于 1,则 $I_x(a,b)$在约 $x=a/(a+b)$处从接近零快速增长到接近 1。图 6.4.1 画出了几对(a,b)的函数曲线。

不完全 B 函数级数展开式为

$$I_x(a,b) = \frac{x^a(1-x)^b}{aB(a,b)} \left[1+\sum_{n=0}^{\infty} \frac{B(a+1,n+1)}{B(a+b,n+1)} x^{n+1}\right] \tag{6.4.4}$$

此式没有证明,但它在数值计算过程中是很有用的(不过,能够运用(6.1.9)式和(6.1.3)式,对每个 n 值由前面结果和一些乘法后,就可求得系数中的 B 函数值)。

连分式表示已证明是更为有用的,

$$I_x(a,b) = \frac{x^a(1-x)^b}{aB(a,b)} \left[\frac{1}{1+}\frac{d_1}{1+}\frac{d_2}{1+}\cdots\right] \tag{6.4.5}$$

其中

$$\begin{aligned} d_{2m+1} &= -\frac{(a+m)(a+b+m)x}{(a+2m)(a+2m+1)} \\ d_{2m} &= \frac{m(b-m)x}{(a+2m-1)(a+2m)} \end{aligned} \tag{6.4.6}$$

连分式当 $x<(a+1)/(a+b+2)$时收敛很快,最坏的情况是 $O(\sqrt{\max(a,b)})$次迭代。当 $x>(a+1)/(a+b+2)$时,可利用(6.4.3)式的对称系获得等价的计算,那时连分式也快速收敛。因此有

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::betai(const DP a, const DP b, const DP x)
返回不完全 B 函数值 I_x(a,b).
{
    DP bt;

    if (x < 0.0 || x > 1.0) nrerror("Bad x in routine betai");
    if (x == 0.0 || x == 1.0) bt=0.0;
    else                                        连分子前的因子。
        bt=exp(gammln(a+b)-gammln(a)-gammln(b)+a*log(x)+b*log(1.0-x));
    if (x < (a+1.0)/(a+b+2.0))                  直接使用连分式。
        return bt*betacf(a,b,x)/a;
    else                                        进行对称变换以后使用分式。
        return 1.0-bt*betacf(b,a,1.0-x)/b;
}
```

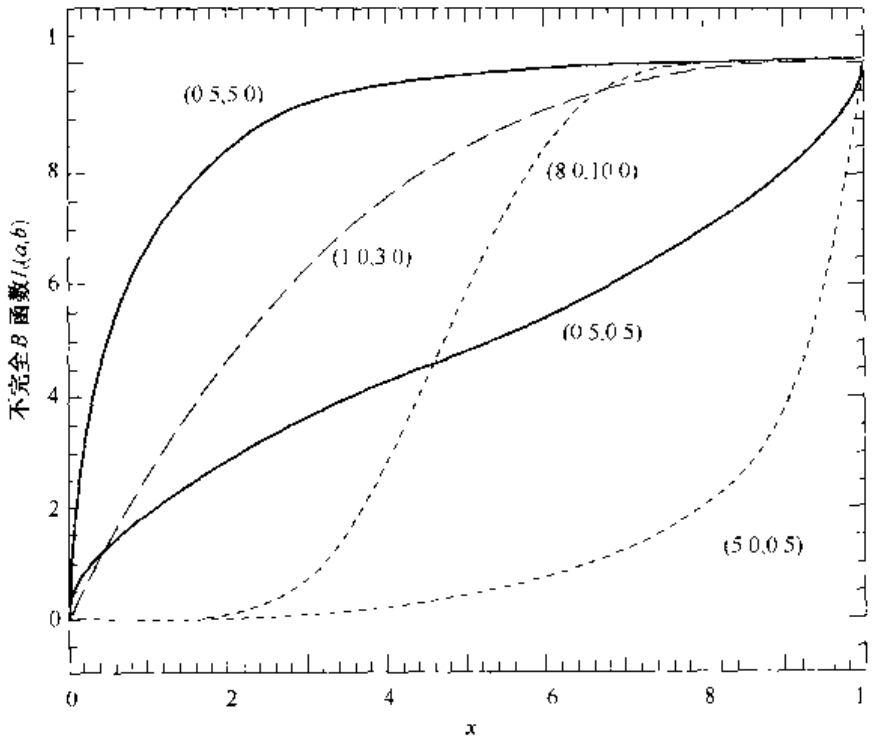


图 6.4.1 五对不同(a,b)的不完全 B 函数 $I_x(a,b)$、注意(0.5,5.0)和(5.0,0.5)关于对角线反射对称，参见(6.4.3)式

下面的程序用连分式求值：

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

DP NR::betacf(const DP a, const DP b, const DP x)
用修正 Lentz 方法(参见 5.2 节)对不完全 B 函数的连分式求值，被 betai 调用。
{
    const int MAXIT=100;
    const DP EPS=numeric_limits<DP>::epsilon();
    const DP FPMIN=numeric_limits<DP>::min()/EPS;
    int m,m2;
    DP aa,c,d,del,h,qab,qam,qap;

    qab=a+b;                                    这些 q 用在 (6.4.6)式系数的因子中。
    qap=a+1.0;
    qam=a-1.0;
    c=1.0;                                      Lentz方法的第一步。
```

```
    d=1.0-qab*x/qap;
    if (fabs(d) < FPMIN) d=FPMIN;
    d=1.0/d;
    h=d;
    for (m=1;m<=MAXIT;m++) {
        m2=2*m;
        aa=m*(b-m)*x/((qam+m2)*(a+m2));
        d=1.0+aa*d;                          递推式的一步(偶数步)。
        if (fabs(d) < FPMIN) d=FPMIN;
        c=1.0+aa/c;
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        h *= d*c;
        aa = -(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
        d=1.0+aa*d;                          递推式的下一步(奇数步)。
        if (fabs(d) < FPMIN) d=FPMIN;
        c=1.0+aa/c;
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        del=d*c;
        h *= del;
        if (fabs(del-1.0) <= EPS) break;   是否完成?
    }
    if (m > MAXIT) nrerror("a or b too big, or MAXIT too small in betacf");
    return h;
}
```

6.4.1 学生分布概率函数

学生分布用 $A(t|\nu)$表示,它在统计学中很有用,特别是用于检验两个分布是否有相同均值。$A(t|\nu)$为有 ν 重自由度的概率,若两均值相等,则某统计量 t(用于度量均值差)小于观测值(详见第 14 章)。如果 $A(t|\nu)>0.99$,则两均值相差很大。换言之,$1-A(t|\nu)$就很小,说明均值相等的假设不成立。

函数的数学定义为

$$A(t|\nu)=\frac{1}{\nu^{1/2}B(\frac{1}{2},\frac{\nu}{2})}\int_{-t}^{t}\left(1+\frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}dx \tag{6.4.7}$$

限定值为

$$A(0|\nu)=0 \qquad A(\infty|\nu)=1 \tag{6.4.8}$$

$A(t|\nu)$与不完全 B 函数 $I_x(a,b)$关系为

$$A(t|\nu)=1-I_{\frac{\nu}{\nu+t^2}}\left(\frac{\nu}{2},\frac{1}{2}\right) \tag{6.4.9}$$

这样,可用(6.4.9)式及上述程序 betai 来求函数值。

6.4.2 F 分布概率函数

此函数用于统计检验两个观察样本是否有同样的方差。统计量 F 定义为,第一个样本的观测离差与第二个样本的观测离差之比,它不难计算得到(详见第 14 章)。若第一个样本分布方差小于第二个样本分布方差,则与 F 一样大小的概率记为$Q(F|\nu_1,\nu_2)$,其中 ν_1 和 ν_2 分别为第一个和第二个样本的自由度数。换言之,$Q(F|\nu_1,\nu_2)$是表示假设"第一个方差小于第二个方差"被否决的显著性水平。其数值很小,表明被否决的可能性很大,而其反过来的含义就是,假设"第一个方差大于等于第二个方差"的可信度很大。

$Q(F|\nu_1,\nu_2)$有限定值

$$Q(0|\nu_1,\nu_2)=1 \qquad Q(\infty|\nu_1,\nu_2)=0 \tag{6.4.10}$$

它与不完全 B 函数 $I_x(a,b)$的关系($I_x(a,b)$由前面 betai 得到)为

$$Q(F|\nu_1,\nu_2)=I_{\frac{\nu_2}{\nu_2+\nu_1 F}}\left(\frac{\nu_2}{2},\frac{\nu_1}{2}\right) \tag{6.4.11}$$

6.4.3 累积二项式概率分布

假设每次试验中某一事件发生概率为 p，则 n 次试验中它发生次数超过 k(或等于 k)的概率 P 称为**累积二项式概率**，与不完全 B 函数 $I_x(a,b)$ 的关系如下：

$$P\equiv\sum_{j=k}^{n}\binom{n}{j}p^j(1-p)^{n-j}=I_p(k,n-k+1) \tag{6.4.12}$$

当 n 大于 12 左右时，最好用 betai 求(6.4.12)式中的和，而不用计算二项式系数再直接求和(当 n 小于 12 时，两种方法均可)。

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapters 6 and 26.

Pearson, E., and Johnson, N. 1968, *Tables of the Incomplete Beta Function* (Cambridge: Cambridge University Press).

6.5 整数阶贝塞尔函数

本节及下一节将给出计算各种整数阶贝塞尔函数的一些实用算法。在 6.7 节讨论分数阶情况。实际上计算分数阶情况的程序对整数阶情况也能很好运行。不过，本节计算整数阶的函数要更简单快速，惟一的不足是受到以有理逼近为基础的精度的限制。双精度情况时最好用 6.7 节中的分数阶函数。

对任何实数 ν，贝塞尔函数 $J_\nu(x)$ 由级数表示定义为

$$J_\nu(x)=\left(\frac{1}{2}x\right)^\nu\sum_{k=0}^{\infty}\frac{(-\frac{1}{4}x^2)^k}{k!\Gamma(\nu+k+1)} \tag{6.5.1}$$

此级数对任何 x 都收敛，但 $x\gg 1$ 时，计算上不是很有用。

当 ν 不是整数时，贝塞尔函数 $Y_\nu(x)$ 由下式给出：

$$Y_\nu(x)=\frac{J_\nu(x)\cos(\nu\pi)-J_{-\nu}(x)}{\sin(\nu\pi)} \tag{6.5.2}$$

ν 趋近整数 n 时，上式右边趋近于正确限定值 $Y_n(x)$，但在实用时没有多大用处。

若变量 $x<\nu$，定性地看，上面两类贝塞尔函数都像简单的幂关系，对 $0<x\ll\nu$ 的渐近形式为

$$\begin{aligned}
J_\nu(x)&\sim\frac{1}{\Gamma(\nu+1)}\left(\frac{1}{2}x\right)^\nu \qquad \nu\geqslant 0\\
Y_0(x)&\sim\frac{2}{\pi}\ln(x)\\
Y_\nu(x)&\sim-\frac{\Gamma(\nu)}{\pi}\left(\frac{1}{2}x\right)^{-\nu} \qquad \nu>0
\end{aligned} \tag{6.5.3}$$

若 $x>\nu$，定性地看，上面两类贝塞尔函数像正弦或余弦波，其幅度按 $x^{-1/2}$ 衰减。$x\gg\nu$ 的渐近形式为

$$\begin{aligned}
J_\nu(x)&\sim\sqrt{\frac{2}{\pi x}}\cos\left(x-\frac{1}{2}\nu\pi-\frac{1}{4}\pi\right)\\
Y_\nu(x)&\sim\sqrt{\frac{2}{\pi x}}\sin\left(x-\frac{1}{2}\nu\pi-\frac{1}{4}\pi\right)
\end{aligned} \tag{6.5.4}$$

在 $x \sim \nu$ 的过渡区间内,贝塞尔函数的典型幅度按以下的阶衰减:

$$
\begin{aligned}
J_\nu(\nu) &\sim \frac{2^{1/3}}{3^{2/3}\Gamma(\frac{2}{3})}\frac{1}{\nu^{1/3}} \sim \frac{0.4473}{\nu^{1/3}} \\
Y_\nu(\nu) &\sim -\frac{2^{1/3}}{3^{1/6}\Gamma(\frac{2}{3})}\frac{1}{\nu^{1/3}} \sim -\frac{0.7748}{\nu^{1/3}}
\end{aligned}
\tag{6.5.5}
$$

ν 很大时,上述渐近式有效。图 6.5.1 画出了每类贝塞尔函数的前几个。

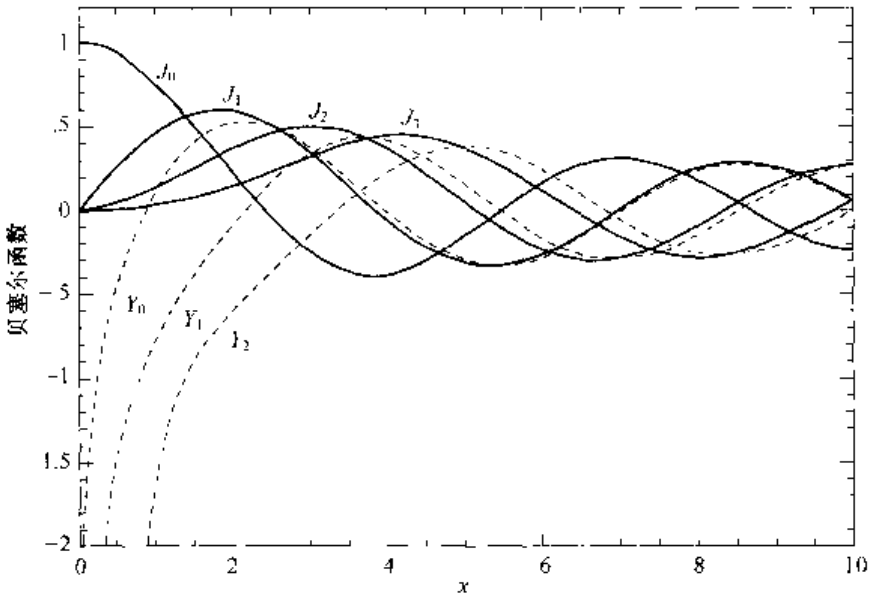


图 6.5.1 $J_0(x)$到 $J_3(x)$, $Y_0(x)$到 $Y_2(x)$的贝塞尔函数

贝塞尔函数满足递推关系

$$
J_{n+1}(x) = \frac{2n}{x}J_n(x) - J_{n-1}(x) \tag{6.5.6}
$$

和

$$
Y_{n+1}(x) = \frac{2n}{x}Y_n(x) - Y_{n-1}(x) \tag{6.5.7}
$$

如 5.5 节已指出,当 $x < n$ 时,沿 n 增加的方向只有(6.5.7)式的第二部分是稳定的。沿 n 增加的方向(6.5.6)式是不稳定的,其原因是它与(6.5.7)式的递推关系一样:由舍入误差引入的少量"污染"了的 Y_n 很快就使求 J_n 遇到麻烦,参见(6.5.3)式。

计算整数阶贝塞尔函数的实用策略分成两步:第一步,如何计算 J_0, J_1, Y_0 和 Y_1;第二步,如何使用稳定递推关系找到其他 J 和 Y。首先论述第一步的任务。

对落在 0 至任意值(取为 8)之间的 x,用 x 的有理函数逼近 $J_0(x)$和 $J_1(x)$。同样地,用有理函数去逼近 $Y_0(x)$和 $Y_1(x)$的"正则部分",此部分定义为

$$
Y_0(x) - \frac{2}{\pi}J_0(x)\ln(x) \qquad 和 \qquad Y_1(x) - \frac{2}{\pi}\left[J_1(x)\ln(x) - \frac{1}{x}\right] \tag{6.5.8}
$$

当 $8 < x < \infty$ 时,用逼近式($n = 0, 1$)

$$
J_n(x) = \sqrt{\frac{2}{\pi x}}\left[P_n\left(\frac{8}{x}\right)\cos(X_n) - Q_n\left(\frac{8}{x}\right)\sin(X_n)\right] \tag{6.5.9}
$$

$$Y_n(x) = \sqrt{\frac{2}{\pi x}}\left[P_n\left(\frac{8}{x}\right)\sin(X_n) + Q_n\left(\frac{8}{x}\right)\cos(X_n)\right] \tag{6.5.10}$$

其中

$$X_n \equiv x - \frac{2n+1}{4}\pi \tag{6.5.11}$$

当 $0 < 8/x < 1$ 时，P_0、P_1、Q_0 和 Q_1 是其变量的多项式。P 为偶多项式、Q 为奇多项式。

Hart[1]对不同的精度要求给出了不同有理函数和多项式的系数。直接实现为：

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::bessj0(const DP x)
对任意实数 x 返回贝塞尔函数值 J0(x)。
{
    DP ax,z,xx,y,ans,ans1,ans2;

    if ((ax=fabs(x)) < 8.0) {            直接有理函数逼近。
        y=x*x;
        ans1=57568490574.0+y*(-13362590354.0+y*(651619640.7
            +y*(-11214424.18+y*(77392.33017+y*(-184.9052456)))));
        ans2=57568490411.0+y*(1029532985.0+y*(9494680.718
            +y*(59272.64853+y*(267.8532712+y*1.0))));
        ans=ans1/ans2;
    } else {                             拟合函数(6.5.9)。
        z=8.0/ax;
        y=z*z;
        xx=ax-0.785398164;
        ans1=1.0+y*(-0.1098628627e-2+y*(0.2734510407e-4
            +y*(-0.2073370639e-5+y*0.2093887211e-6)));
        ans2 = -0.1562499995e-1+y*(0.1430488765e-3
            +y*(-0.6911147651e-5+y*(0.7621095161e-6
            -y*0.934945152e-7)));
        ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
    }
    return ans;
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::bessy0(const DP x)
对正数 x 返回贝塞尔函数值 Y0(x)。
{
    DP z,xx,y,ans,ans1,ans2;

    if (x < 8.0) {                       (6.5.8)式有理函数逼近。
        y=x*x;
        ans1 = -2957821389.0+y*(7062834065.0+y*(-512359803.6
            +y*(10879881.29+y*(-86327.92757+y*228.4622733))));
        ans2=40076544269.0+y*(745249964.8+y*(7189466.438
            +y*(47447.26470+y*(226.1030244+y*1.0))));
        ans=(ans1/ans2)+0.636619772*bessj0(x)*log(x);
    } else {                             拟合函数(6.5.10)。
        z=8.0/x;
        y=z*z;
        xx=x-0.785398164;
        ans1=1.0+y*(-0.1098628627e-2+y*(0.2734510407e-4
            +y*(-0.2073370639e-5+y*0.2093887211e-6)));
        ans2 = -0.1562499995e-1+y*(0.1430488765e-3
            +y*(-0.6911147651e-5+y*(0.7621095161e-6
```

```
            +y*(-0.934945152e-7))));
        ans=sqrt(0.636619772/x)*(sin(xx)*ans1+z*cos(xx)*ans2);
    }
    return ans;
}


#include <cmath>
#include "nr.h"
using namespace std;

DP NR::bessj1(const DP x)
对任意实数 x 返回贝塞尔函数值 J1(x)。
{
    DP ax,z,xx,y,ans,ans1,ans2;

    if ((ax=fabs(x)) < 8.0) {          直接有理逼近。
        y=x*x;
        ans1=x*(72362614232.0+y*(-7895059235.0+y*(242396853.1
            +y*(-2972611.439+y*(15704.48260+y*(-30.16036606))))));
        ans2=144725228442.0+y*(2300535178.0+y*(18583304.74
            +y*(99447.43394+y*(376.9991397+y*1.0))));
        ans=ans1/ans2;
    } else {                           拟合函数(6.5.9)。
        z=8.0/ax;
        y=z*z;
        xx=ax-2.356194491;
        ans1=1.0+y*(0.183105e-2+y*(-0.3516396496e-4
            +y*(0.2457520174e-5+y*(-0.240337019e-6))));
        ans2=0.04687499995+y*(-0.2002690873e-3
            +y*(0.8449199096e-5+y*(-0.88228987e-6
            +y*0.105787412e-6)));
        ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
        if (x < 0.0) ans = -ans;
    }
    return ans;
}


#include <cmath>
#include "nr.h"
using namespace std;

DP NR::bessy1(const DP x)
对正数 x 返回贝塞尔函数值 Y1(x)。
{
    DP z,xx,y,ans,ans1,ans2;

    if (x < 8.0) {                     (6.5.8)式有理函数逼近。
        y=x*x;
        ans1=x*(-0.4900604943e13+y*(0.1275274390e13
            +y*(-0.5153438139e11+y*(0.7349264551e9
            +y*(-0.4237922726e7+y*0.8511937935e4)))));
        ans2=0.2499580570e14+y*(0.4244419664e12
            +y*(0.3733650367e10+y*(0.2245904002e8
            +y*(0.1020426050e6+y*(0.3549632885e3+y)))));
        ans=(ans1/ans2)+0.636619772*(bessj1(x)*log(x)-1.0/x);
    } else {                           拟合函数(6.5.10)。
        z=8.0/x;
        y=z*z;
        xx=x-2.356194491;
        ans1=1.0+y*(0.183105e-2+y*(-0.3516396496e-4
            +y*(0.2457520174e-5+y*(-0.240337019e-6))));
        ans2=0.04687499995+y*(-0.2002690873e-3
            +y*(0.8449199096e-5+y*(-0.88228987e-6
            +y*0.105787412e-6)));
        ans=sqrt(0.636619772/x)*(sin(xx)*ans1+z*cos(xx)*ans2);
    }
    return ans;
}
```

现在讨论第二步，即如何由递推关系式(6.5.6)和(6.5.7)得到贝塞尔函数 $J_n(x)$ 和 $Y_n(x)$ $(n \geq 2)$。$Y_n(x)$可直接计算，因其向前递推总是稳定的。

```
#include "nr.h"

DP NR::bessy(const int n, const DP x)
对正数 x 且 n≥2 返回贝塞尔函数值 Yn(x)。
{
    int j;
    DP by,bym,byp,tox;

    if (n < 2) nrerror("Index n less than 2 in bessy");
    tox=2.0/x;
    by=bessy1(x);                    递推式的起始值。
    bym=bessy0(x);
    for (j=1;j<n;j++) {              递推式(6.5.7)。
        byp=j*tox*by-bym;
        bym=by;
        by=byp;
    }
    return by;
}
```

本算法需调用 bessy1 和 bessy0(导致调用各自的 bessj1 和 bessj0)，再加上递推中 $O(n)$次运算。

至于 $J_n(x)$就有点复杂了。可以从 J_0 和 J_1 开始，按 n 向上递推，但仅当 n 不超过 x 时，递推保持稳定。这样仅当 x 很大而 n 较小时才适合，不过实际经常正是如此。

困难的是当 $x < n$，这时最好用 Miller 算法，参见(5.5.16)式的讨论，从任意起始值开始向后递推，利用递推式的向前不稳定性而到达正确解。最后得到 J_0 和 J_1 时，再用过程中的累积和式(5.5.16)归一化所得到的解。

惟一的问题是，应从多大的 n 处开始向后递推，使达到希望的 n 值时能得到要求的精度。考察渐近式(6.5.3)和(6.5.5)，可以得出结论，应从比期望的 n 值大一个增量$[\text{常数} \times n]^{1/2}$的地方开始，其中常数的平方根粗略地等于精度的有效位数。

由上面讨论得到下面函数。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

DP NR::bessj(const int n, const DP x)
对实数 x 和 n≥2 返回贝塞尔函数值 Jn(x)。
{
    const DP ACC=160.0;                      增加 ACC 以提高精度。
    const int IEXP=numeric_limits<DP>::max_exponent/2;
    bool jsum;
    int j,k,m;
    DP ax,bj,bjm,bjp,dum,sum,tox,ans;

    if (n < 2) nrerror("Index n less than 2 in bessj");
    ax=fabs(x);
    if (ax*ax <= 8.0*numeric_limits<DP>::min()) return 0.0;
    else if (ax > DP(n)) {                   从 J0和 J1开始向前递推。
        tox=2.0/ax;
        bjm=bessj0(ax);
        bj=bessj1(ax);
        for (j=1;j<n;j++) {
            bjp=j*tox*bj-bjm;
            bjm=bj;
            bj=bjp;
        }
        ans=bj;
    } else {                                 在此算得的偶数 m 开始向后递推。
```

```
            tox=2.0/ax;
            m=2*((n+int(sqrt(ACC*n)))/2);
            jsum=false;
            bjp=ans=sum=0.0;                    jsum取0或1，当取1时，求(5.5.16)式中偶数项之和。
            bj=1.0;
            for (j=m;j>0;j--) {                 向后递推。
                bjm=j*tox*bj-bjp;
                bjp=bj;
                bj=bjm;
                dum=frexp(bj,&k);
                if (k > IEXP) {                 重新归一化以防溢出。
                    bj=ldexp(bj,-IEXP);
                    bjp=ldexp(bjp,-IEXP);
                    ans=ldexp(ans,-IEXP);
                    sum=ldexp(sum,-IEXP);
                }
                if (jsum) sum += bj;            求累积和。
                jsum=!jsum;                     把0变为1,或把1变为0。
                if (j == n) ans=bjp;            存储未归一化的结果。
            }
            sum=2.0*sum-bj;                     计算(5.5.16)式.
            ans /= sum;                         用于答案的归一化。
        }
        return x < 0.0 && (n & 1) ? -ans : ans;
    }
```

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 9.

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley), §6.8, p. 141. [1]

6.6 修正的整数阶贝塞尔函数

修正贝塞尔函数 $I_n(x)$ 和 $K_n(x)$ 对纯虚数变量与一般贝塞尔函数 J_n 和 Y_n 等价，详细地说，其关系式为

$$
\begin{aligned}
I_n(x) &= (-i)^n J_n(ix) \\
K_n(x) &= \frac{\pi}{2} i^{n+1}[J_n(ix) + iY_n(ix)]
\end{aligned}
\tag{6.6.1}
$$

仔细选择特殊前因子和组成 K_n 的 J_n 和 Y_n 的线性组合，使得变量 x 为实数时，函数值为实数。

x 很小($x \ll n$)时 $I_n(x)$ 和 $K_n(x)$ 都逐渐逼近其自变量的简单幂函数

$$
\begin{aligned}
I_n(x) &\approx \frac{1}{n!}\left(\frac{x}{2}\right)^n \qquad n \geqslant 0 \\
K_0(x) &\approx -\ln(x) \\
K_n(x) &\approx \frac{(n-1)!}{2}\left(\frac{x}{2}\right)^{-n} \qquad n > 0
\end{aligned}
\tag{6.6.2}
$$

上面表达式实质上与 $J_n(x)$ 和 $Y_n(x)$ 在此区域内的渐近式相同，只是 $Y_n(x)$ 和 $K_n(x)$ 相差一个因子 $-2/\pi$。$x \gg n$ 区域内，修正贝塞尔函数具有与贝塞尔函数完全不同的性质，

$$
\begin{aligned}
I_n(x) &\approx \frac{1}{\sqrt{2\pi x}} \exp(x) \\
K_n(x) &\approx \frac{\pi}{\sqrt{2\pi x}} \exp(-x)
\end{aligned}
\tag{6.6.3}
$$

自变量很大时，修正函数明显具有指数而不是正弦性质(参见图 6.6.1)，若去掉指数因子，则修正贝塞尔函数较平滑，这一点使只含有少数项的简单多项式的逼近很适用于 I_0，I_1，K_0 及 K_1。下面程序基于由Abramowitz和Stegun[1]给出的多项式系数，计算此4个函数值，同时可用于 $x > n$ 时，对 $n > 1$ 的向前递推，

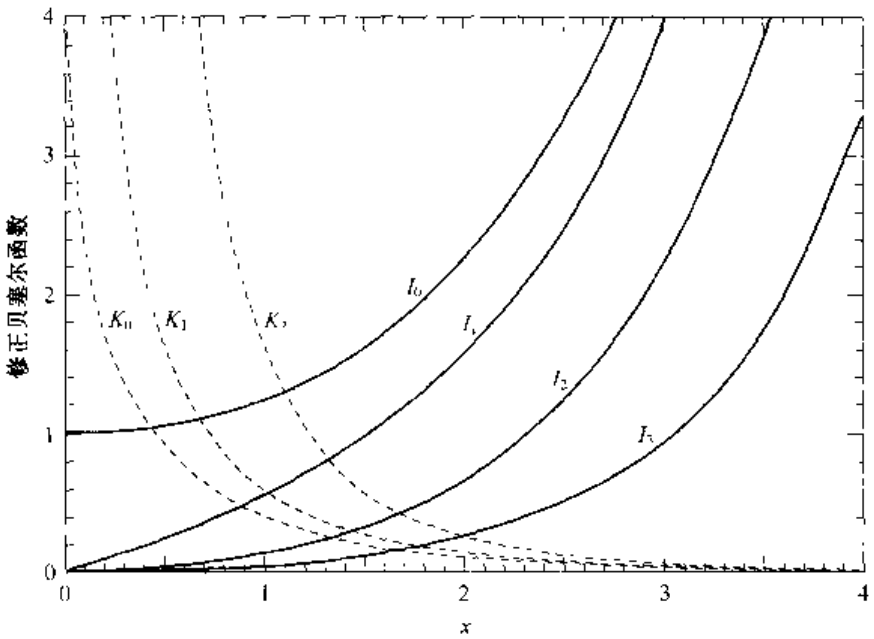


图6.6.1 $I_0(x)$到$I_3(x)$，$K_0(x)$到$K_2(x)$的修正贝塞尔函数

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::bessi0(const DP x)
对任意实数x返回修正贝塞尔函数值 I0(x)。
{
    DP ax,ans,y;

    if ((ax=fabs(x)) < 3.75) {          多项式拟合。
        y=x/3.75;
        y*=y;
        ans=1.0+y*(3.5156229+y*(3.0899424+y*(1.2067492
            +y*(0.2659732+y*(0.360768e-1+y*0.45813e-2)))));
    } else {
        y=3.75/ax;
        ans=(exp(ax)/sqrt(ax))*(0.39894228+y*(0.1328592e-1
            +y*(0.225319e-2+y*(-0.157565e-2+y*(0.916281e-2
            +y*(-0.2057706e-1+y*(0.2635537e-1+y*(-0.1647633e-1
            +y*0.392377e-2))))))));
    }
    return ans;
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::bessk0(const DP x)
对正实数x返回修正贝塞尔函数值 K0(x)。
{
    DP y,ans;

    if (x <= 2.0) {                 多项式拟合。
        y=x*x/4.0;
        ans=(-log(x/2.0)*bessi0(x))+(-0.57721566+y*(0.42278420
            +y*(0.23069756+y*(0.3488590e-1+y*(0.262698e-2
            +y*(0.10750e-3+y*0.74e-5))))));
    } else {
        y=2.0/x;
        ans=(exp(-x)/sqrt(x))*(1.25331414+y*(-0.7832358e-1
            +y*(0.2189568e-1+y*(-0.1062446e-1+y*(0.587872e-2
```

```
            +y*(-0.251540e-2+y*0.53208e-3))))));
    }
    return ans;
}


#include <cmath>
#include "nr.h"
using namespace std;

DP NR::bessi1(const DP x)
对正实数x返回修正贝塞尔函数值K1(x)。
{
    DP ax,ans,y;

    if ((ax=fabs(x)) < 3.75) {          多项拟合。
        y=x/3.75;
        y*=y;
        ans=ax*(0.5+y*(0.87890594+y*(0.51498869+y*(0.15084934
            +y*(0.2658733e-1+y*(0.301532e-2+y*0.32411e-3))))));
    } else {
        y=3.75/ax;
        ans=0.2282967e-1+y*(-0.2895312e-1+y*(0.1787654e-1
            -y*0.420059e-2));
        ans=0.39894228+y*(-0.3988024e-1+y*(-0.362018e-2
            +y*(0.163801e-2+y*(-0.1031555e-1+y*ans))));
        ans *= (exp(ax)/sqrt(ax));
    }
    return x < 0.0 ? -ans : ans;
}


#include <cmath>
#include "nr.h"
using namespace std;

DP NR::bessk1(const DP x)
对正实数x返回修正贝塞尔函数值K1(x)。
{
    DP y,ans;

    if (x <= 2.0) {                 多项式拟合。
        y=x*x/4.0;
        ans=(log(x/2.0)*bessi1(x))+(1.0/x)*(1.0+y*(0.15443144
            +y*(-0.67278579+y*(-0.18156897+y*(-0.1919402e-1
            +y*(-0.110404e-2+y*(-0.4686e-4)))))));
    } else {
        y=2.0/x;
        ans=(exp(-x)/sqrt(x))*(1.25331414+y*(0.23498619
            +y*(-0.3655620e-1+y*(0.1504268e-1+y*(-0.780353e-2
            +y*(0.325614e-2+y*(-0.68245e-3)))))));
    }
    return ans;
}
```

$I_n(x)$和$K_n(x)$的递推关系与$J_n(x)$和$Y_n(x)$相同,只是用ix代替x,这样在关系式中就需改变符号,

$$
\begin{aligned}
I_{n+1}(x) &= -\left(\frac{2n}{x}\right) I_n(x) + I_{n-1}(x) \\
K_{n+1}(x) &= +\left(\frac{2n}{x}\right) K_n(x) + K_{n-1}(x)
\end{aligned}
\tag{6.6.4}
$$

向前递推时,这些关系式常常是**不稳定**的。但对K_n,本身增长没有问题。对I_n需再次用向后递推的方法,并按与程序bessj相同的方法选择递推起始点。根本的区别就在于,$I_n(x)$的归一化公式中连续项交替出现减号,这也是因为在先前对J_n使用的公式中,用ix代替x的缘故,

$$
1 = I_0(x) - 2I_2(x) + 2I_4(x) - 2I_6(x) + \cdots \tag{6.6.5}
$$

实际上，简单地调用 bessi0 进行归一化更好。

递推程序 bessj 和 bessy 经过简单修改后就成为新程序 bessi 和 bessk：

```
#include "nr.h"

DP NR::bessk(const int n, const DP x)
对任意实数 x 且 n≥2 返回修正贝塞尔函数值 K_n(x)。
{
    int j;
    DP bk,bkm,bkp,tox;

    if (n < 2) nrerror("Index n less than 2 in bessk");
    tox=2.0/x;
    bkm=bessk0(x);                  对所有 x 向前递推……
    bk=bessk1(x);
    for (j=1;j<n;j++) {             ……这里进行递推。
        bkp=bkm+j*tox*bk;
        bkm=bk;
        bk=bkp;
    }
    return bk;
}
```

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

DP NR::bessi(const int n, const DP x)
对任意实数 x 且 n≥2 返回修正贝塞尔函数值 I_n(x)。
{
    const DP ACC=200.0;                     增加 ACC 以提高精度。
    const int IEXP=numeric_limits<DP>::max_exponent/2;
    int j,k;
    DP bi,bim,bip,dum,tox,ans;

    if (n < 2) nrerror("Index n less than 2 in bessi");
    if (x*x <= 8.0*numeric_limits<DP>::min()) return 0.0;
    else {
        tox=2.0/fabs(x);
        bip=ans=0.0;
        bi=1.0;
        for (j=2*(n+int(sqrt(ACC*n)));j>0;j--) {     从一个偶数 m 起向后递推。
            bim=bip+j*tox*bi;
            bip=bi;
            bi=bim;
            dum=frexp(bi,&k);
            if (k > IEXP) {                 重新归一化了防溢出。
                ans=ldexp(ans,-IEXP);
                bi=ldexp(bi,-IEXP);
                bip=ldexp(bip,-IEXP);
            }
            if (j == n) ans=bip;
        }
        ans *= bessi0(x)/bi;                用 bessi0 进行归一化。
        return x < 0.0 && (n & 1) ? -ans : ans;
    }
}
```

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §9.8. [1]

Carrier, G.F., Krook, M. and Pearson, C.E. 1966, *Functions of a Complex Variable* (New York: McGraw-Hill), pp. 220ff.

6.7 深入讨论:分数阶贝塞尔函数、艾里函数、球面贝塞尔函数

数值计算分数阶贝塞尔函数有很多方法,只是大多数算法并不实用。本节给出的程序虽然比较复的确值得推荐使用。

6.7.1 一般贝塞尔函数

基本思想是 Steed 方法,此方法来源于求 Coulomb 波形函数[1],这个方法必须同时求 J_ν、J'_ν、Y_ν 和样必然涉及四者之间的关系。有三个关系式来自两个连分式,其中有一式是复数。第四个关系式满足 skian 关系:

$$W \equiv J_\nu Y'_\nu - Y_\nu J'_\nu = \frac{2}{\pi x} \tag{6.}$$

第一个连分式(CF1)定义为

$$\begin{aligned} f_\nu \equiv \frac{J'_\nu}{J_\nu} &= \frac{\nu}{x} - \frac{J_{\nu+1}}{J_\nu} \\ &= \frac{\nu}{x} - \frac{1}{2(\nu+1)/x -} \; \frac{1}{2(\nu+2)/x -} \cdots \end{aligned} \tag{6.}$$

此式很容易由贝塞尔函数的三项递推式导出:从(6.5.6)式开始并利用(5.5.18)式。5.2 节中连分式的前向值本质上与递推关系式的后向递推等价。CF1 的收敛速率由**转折点** $x_{tp} = \sqrt{\nu(\nu+1)} \approx \nu$ 的位置决定,在此外贝塞尔函数为振荡性。当 $x \le x_{tp}$ 时,收敛很快。若 $x \ge x_{tp}$,每次 ν 增加 1,迭代连分式直到 $x \approx x_{tp}$,此后敛加快。因此 x 很大时,CF1 的迭代次数与 x 同阶。程序 bessjy 中,设定允许迭代次数为 10 000 次。x 很时,则使用贝塞尔函数的一般渐近表达式。

可以看出一旦 CF1 收敛,它的分母便与 J_ν 的符号一致。

复连分式 CF2 定义为

$$p + iq \equiv \frac{J'_\nu + iY'_\nu}{J_\nu + iY_\nu} = -\frac{1}{2x} + i + \frac{i}{x} \; \frac{(1/2)^2 - \nu^2}{2(x+i) +} \; \frac{(3/2)^2 - \nu^2}{2(x+2i) +} \cdots \tag{6.7.3}$$

(在下一小节中,讨论如何按类似修正贝塞尔函数的方式导出 CF2。)$x \ge x_{tp}$ 时,连分式收敛很快,$x \to 0$ 时,不收敛。x 很小时,必须采用一种特殊方法,下面将介绍。x 不是很小时,保证 $x \ge x_{tp}$ 则将 J_ν 和 J'_ν 稳定地向后递推到 $\nu = \mu \le x$,在此较小 ν 值处得到比值 f_μ。递推关系式在此方向上是稳定的。递推初始值是

$$J_\nu = \text{任意值}, \qquad J'_\nu = f_\nu J_\nu \tag{6.7.4}$$

注意,J_ν 任意初始值的符号与 CF1 数的分母的符号相同。J_ν 的初始值应很小,以免在递推时产生溢出的可能。递推式为

$$\begin{aligned} J_{\nu-1} &= \frac{\nu}{x} J_\nu + J'_\nu \\ J'_{\nu-1} &= \frac{\nu-1}{x} J_{\nu-1} - J_\nu \end{aligned} \tag{6.7.5}$$

一旦求得 CF2 在 $\nu = \mu$ 处值,则利用 Wronskian(6.7.1)式,这样就有足够的关系式来求解 4 个量,引入下面变量可简化公式

$$\gamma \equiv \frac{p - f_\mu}{q} \tag{6.7.6}$$

有

$$J_\mu = \pm \left(\frac{W}{q + \gamma(p - f_\mu)} \right)^{1/2} \tag{6.7.7}$$

$$J'_\mu = f_\mu J_\mu \tag{6.7.8}$$

$$Y_\mu = \gamma J_\mu \tag{6.7.9}$$

$$Y'_\mu = Y_\mu \left(p + \frac{q}{\gamma} \right) \tag{6.7.10}$$

)式中 J_ν 的符号应与(6.7.4)式中 J_ν 的初始值的符号相同。

一旦得到 $\nu=\mu$ 处4个函数，就可以求得 ν 初始值处的函数值，对 J_ν 和 J'_ν，只需用(6.7.7)式与使用5)式递推后得到的值，代入(6.7.4)式就可简单地求得。又由(6.7.9)式和(6.7.10)式中的值开始并使用向前递推式，则可得 Y_ν 和 Y'_ν：

$$Y_{\nu+1}=\frac{2\nu}{x}Y_\nu-Y_{\nu-1} \tag{6.7.11}$$

应用关系式

$$Y'_\nu=\frac{\nu}{x}Y_\nu-Y_{\nu+1} \tag{6.7.12}$$

现在讨论 x 较小时 CF2 不合适的情况。Temme[2]找到一种很好的方法，即通过精确处理 $x\to0$ 时奇异点数展开来求 Y_ν 和 $Y_{\nu+1}$，然后由(6.7.12)式得到 Y'_ν。展开式仅对 $|\nu|\leqslant1/2$ 时有效，故在此范围内采用递式(6.7.5)，求 f_ν 在 $\nu=\mu$ 处的值。接着由下式求 J_μ：

$$J_\mu=\frac{W}{Y'_\mu-Y_\mu f_\mu} \tag{6.7.13}$$

6.7.8)式得到 f'_μ。在 ν 初始值处的值 f_ν 和 f'_ν 可用如前所述方法求得，Y 也如前所述可递推求得。

Temme 级数为

$$Y_\nu=-\sum_{k=0}^{\infty}c_kg_k \qquad Y_{\nu+1}=-\frac{2}{x}\sum_{k=0}^{\infty}c_kh_k \tag{6.7.14}$$

中

$$c_k=\frac{(-x^2/4)^k}{k!} \tag{6.7.15}$$

数 g_k 和 h_k 由 p_k，q_k 和 f_k 按递推式定义：

$$\begin{aligned}
g_k&=f_k+\frac{2}{\nu}\sin^2\left(\frac{\nu\pi}{2}\right)q_k\\
h_k&=-kg_k+p_k\\
p_k&=\frac{p_{k-1}}{k-\nu}\\
q_k&=\frac{q_{k-1}}{k+\nu}\\
f_k&=\frac{kf_{k-1}+p_{k-1}+q_{k-1}}{k^2-\nu^2}
\end{aligned} \tag{6.7.16}$$

递推起始值为

$$\begin{aligned}
p_0&=\frac{1}{\pi}\left(\frac{x}{2}\right)^{-\nu}\Gamma(1+\nu)\\
q_0&=\frac{1}{\pi}\left(\frac{x}{2}\right)^{\nu}\Gamma(1-\nu)\\
f_0&=\frac{2}{\pi}\frac{\nu\pi}{\sin\nu\pi}\left[\cosh\sigma\Gamma_1(\nu)+\frac{\sinh\sigma}{\sigma}\ln\left(\frac{2}{x}\right)\Gamma_2(\nu)\right]
\end{aligned} \tag{6.7.17}$$

其中

$$\begin{aligned}
\sigma&=\nu\ln\left(\frac{2}{x}\right)\\
\Gamma_1(\nu)&=\frac{1}{2\nu}\left[\frac{1}{\Gamma(1-\nu)}-\frac{1}{\Gamma(1+\nu)}\right]\\
\Gamma_2(\nu)&=\frac{1}{2}\left[\frac{1}{\Gamma(1-\nu)}+\frac{1}{\Gamma(1+\nu)}\right]
\end{aligned} \tag{6.7.18}$$

这样书写公式的优越性是，通过小心计算 $\nu\pi/\sin\nu\pi$，$\sinh\sigma/\sigma$ 及 Γ_1 的值，能控制 $\nu\to0$ 时可能出现的问题。特别是，Temme 给出了 $\Gamma_1(\nu)$ 和 $\Gamma_2(\nu)$ 的切比雪夫展开式。在这里将 Γ_1 的展开式重新排成明晰的 ν 的偶级数形式，以便可以调用 5.8 节中的程序 chebev。

程序假定 $\nu \geqslant 0$. 若 ν 为负数,则可使用反射公式

$$J_{-\nu} = \cos \nu\pi \, J_\nu - \sin \nu\pi \, Y_\nu$$
$$Y_{-\nu} = \sin \nu\pi \, J_\nu + \cos \nu\pi \, Y_\nu \tag{6.7.19}$$

程序还假定 $x > 0$. $x < 0$ 时一般函数很复杂,但可由 $x > 0$ 时的函数来表示。$x = 0$ 时,Y_ν 呈奇异性。程序中算法应当使用双精度,即使想要得到的结果只需单精度。复数运算通过实变量计算。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

void NR::bessjy(const DP x, const DP xnu, DP &rj, DP &ry, DP &rjp, DP &ryp)
对任意自数 x 和 xnu = ν≥0 返回贝塞尔函数 rj = J_ν, ry = Y_ν 及其导数 rjp = J'_ν, ryp = Y'_ν。除在某一函数零点附近处,相
对精度在 EPS(EPS 控制绝对精度)的一或两位有效数字范围之内,FPMIN 是接近机器最小浮点数的某一数。
{
    const int MAXIT=10000;
    const DP EPS=numeric_limits<DP>::epsilon();
    const DP FPMIN=numeric_limits<DP>::min()/EPS;
    const DP XMIN=2.0, PI=3.141592653589793;
    DP a,b,br,bi,c,cr,ci,d,del,del1,den,di,dlr,dli,dr,e,f,fact,fact2,
        fact3,ff,gam,gam1,gam2,gammi,gampl,h,p,pimu,pimu2,q,r,rjl,
        rjl1,rjmu,rjp1,rjpl,rjtemp,ry1,rymu,rymup,rytemp,sum,sum1,
        temp,w,x2,xi,xi2,xmu,xmu2;
    int i,isign,l,nl;

    if (x <= 0.0 || xnu < 0.0)
        nrerror("bad arguments in bessjy");
    nl=(x < XMIN ? int(xnu+0.5) : MAX(0,int(xnu-x+1.5)));
    nl 是 J 的向后递推数及 Y 的向前递推数。当 x< XMIN,xmu 处于 -1/2 至 1/2 之间,而当 x≥XMIN,xmu 可选择,以使
    x 大于转折点。
    xmu=xnu-nl;
    xmu2=xmu*xmu;
    xi=1.0/x;
    xi2=2.0*xi;
    w=xi2/PI;                          Wronskian 关系式。
    isign=1;                           由修正 Lentz 方法(参 5.2 节)求 CF1。isign 记录分母符号的变化。
    h=xnu*xi;
    if (h < FPMIN) h=FPMIN;
    b=xi2*xnu;
    d=0.0;
    c=h;
    for (i=0;i<MAXIT;i++) {
        b += xi2;
        d=b-d;
        if (fabs(d) < FPMIN) d=FPMIN;
        c=b-1.0/c;
        if (fabs(c) < FPMIN) c=FPMIN;
        d=1.0/d;
        del=c*d;
        h=del*h;
        if (d < 0.0) isign = -isign;
        if (fabs(del-1.0) <= EPS) break;
    }
    if (i >= MAXIT)
        nrerror("x too large in bessjy; try asymptotic expansion");
    rjl=isign*FPMIN;                   初始化 J_ν 和 J'_ν 准备向后递推。
    rjpl=h*rjl;
    rjl1=rjl;                          存储值供以后调整。
    rjp1=rjpl;
    fact=xnu*xi;
    for (l=nl-1;l>=0;l--) {
        rjtemp=fact*rjl+rjpl;
```

```
        fact -= xi;
        rjpl=fact*rjtemp-rjl;
        rjl=rjtemp;
    }
    if (rjl == 0.0) rjl=EPS;
    f=rjpl/rjl;                          得到未归一化的 J_n 和 J_μ'。
    if (x < XMIN) {                      使用级数。
        x2=0.5*x;
        pimu=PI*xmu;
        fact = (fabs(pimu) < EPS ? 1.0 : pimu/sin(pimu));
        d = -log(x2);
        e=xmu*d;
        fact2 = (fabs(e) < EPS ? 1.0 : sinh(e)/e);
        beschb(xmu,gam1,gam2,gampl,gammi);        Γ1和Γ2 的切比雪夫计算 f_0。
        ff=2.0/PI*fact*(gam1*cosh(e)+gam2*fact2*d);
        e=exp(e);
        p=e/(gampl*PI);                  p_0。
        q=1.0/(e*PI*gammi);              q_0。
        pimu2=0.5*pimu;
        fact3 = (fabs(pimu2) < EPS ? 1.0 : sin(pimu2)/pimu2);
        r=PI*pimu2*fact3*fact3;
        c=1.0;
        d = -x2*x2;
        sum=ff+r*q;
        sum1=p;
        for (i=1;i<=MAXIT;i++) {
            ff=(i*ff+p+q)/(i*i-xmu2);
            c *= (d/i);
            p /= (i-xmu);
            q /= (i+xmu);
            del=c*(ff+r*q);
            sum += del;
            del1=c*p-i*del;
            sum1 += del1;
            if (fabs(del) < (1.0+fabs(sum))*EPS) break;
        }
        if (i > MAXIT)
            nrerror("bessy series failed to converge");
        rymu = -sum;
        ry1 = -sum1*xi2;
        rymup=xmu*xi*rymu-ry1;
        rjmu=w/(rymup-f*rymu);           (6.7.13)式。
    } else {                             由修正 Lentz 方法(参见 5.2 节)求 CF2。
        a=0.25-xmu2;
        p = -0.5*xi;
        q=1.0;
        br=2.0*x;
        bi=2.0;
        fact=a*xi/(p*p+q*q);
        cr=br+q*fact;
        ci=bi+p*fact;
        den=br*br+bi*bi;
        dr=br/den;
        di = -bi/den;
        dlr=cr*dr-ci*di;
        dli=cr*di+ci*dr;
        temp=p*dlr-q*dli;
        q=p*dli+q*dlr;
        p=temp;
        for (i=1;i<MAXIT;i++) {
            a += 2*i;
            bi += 2.0;
            dr=a*dr+br;
            di=a*di+bi;
                if (fabs(dr)+fabs(di) < FPMIN) dr=FPMIN;
                fact=a/(cr*cr+ci*ci);
                cr=br+cr*fact;
                ci=bi-ci*fact;
                if (fabs(cr)+fabs(ci) < FPMIN) cr=FPMIN;
                den=dr*dr+di*di;
                dr /= den;
```

```
            di /= -den;
            dlr=cr*dr-ci*di;
            dli=cr*di+ci*dr;
            temp=p*dlr-q*dli;
            q=p*dli+q*dlr;
            p=temp;
            if (fabs(dlr-1.0)+fabs(dli) <= EPS) break;
        }

        if (i >= MAXIT) nrerror("cf2 failed in bessjy");
        gam=(p-f)/q;                    (6.7.6)式-(6.7.10)式。
        rjmu=sqrt(w/((p-f)*gam+q));
        rjmu=SIGN(rjmu,rjl);
        rymu=rjmu*gam;
        rymup=rymu*(p+q/gam);
        ry1=xmu*xi*rymu-rymup;
    }
    fact=rjmu/rjl;
    rj=rjl1*fact;                       调整 J_ν 和 J'_ν 的起始值。
    rjp=rjp1*fact;
    for (i=1;i<=nl;i++) {               Y_ν 的向前递推。
        rytemp=(xmu+i)*xi2*ry1-rymu;
        rymu=ry1;
        ry1=rytemp;
    }
    ry=rymu;
    ryp=xmu*xi*rymu-ry1;
}

#include "nr.h"

void NR::beschb(const DP x, DP &gam1, DP &gam2, DP &gampl, DP &gammi)
对于|x|≤1/2 用切比雪夫展开式计算 Γ1 和 Γ2，同时返回 1/Γ(1+x)和 1/Γ(1-x)。
{
    const int NUSE1=7, NUSE2=8;
    static const DP c1_d[7] = {
        -1.142022680371168e0,6.5165112670737e-3,
        3.087090173086e-4,-3.4706269649e-6,6.9437664e-9,
        3.67795e-11,-1.356e-13};
    static const DP c2_d[8] = {
        1.843740587300905e0,-7.68528408447867e-2,
        1.2719271366546e-3,-4.9717367042e-6,-3.31261198e-8,
        2.423096e-10,-1.702e-13,-1.49e-15};
    DP xx;
    static Vec_DP c1(c1_d,7),c2(c2_d,8);

    xx=8.0*x*x-1.0;                     x乘以2改变范围为-1到1，然后
    gam1=chebev(-1.0,1.0,c1,NUSE1,xx);  使用变换求偶切比雪夫级数值。
    gam2=chebev(-1.0,1.0,c2,NUSE2,xx);
    gampl= gam2-x*gam1;
    gammi= gam2+x*gam1;
}
```

6.7.2 修正贝塞尔函数

Steed方法不能用于修正贝塞尔函数，因为此时CF2为纯虚数，4个函数中只有3个关系式。Temme[3]给出一个正则条件得到第4个关系式。

Wronskian关系式为

$$W \equiv I_\nu K'_\nu - K_\nu I'_\nu = -\frac{1}{x} \tag{6.7.20}$$

连分式CF1变成为

$$f_\nu \equiv \frac{I'_\nu}{I_\nu} = \frac{\nu}{x} + \frac{1}{2(\nu+1)/x+}\ \frac{1}{2(\nu+2)/x+}\cdots \tag{6.7.21}$$

为方便地得到CF2及正则条件，考察合流超几何函数序列

$$z_n(x) = U(\nu+1/2+n, 2\nu+1, 2x) \tag{6.7.22}$$

ν 固定，则

$$K_\nu(x) = \pi^{1/2}(2x)^\nu e^{-x} z_0(x) \tag{6.7.23}$$

$$\frac{K_{\nu+1}(x)}{K_\nu(x)} = \frac{1}{x}\left[\nu + \frac{1}{2} + x + \left(\nu^2 - \frac{1}{4}\right)\frac{z_1}{z_0}\right] \tag{6.7.24}$$

(6.7.23)式是合流超几何函数 K_ν 的标准表示式，而(6.7.24)式则从相邻连续超几何函数之间的关系式得到(在 Abramowitz 和 Stegun 所著的上节参考文献[1]中的(13.4.16)式和(13.4.18)式)。函数 z_n 满足三项递推关系式(Abramowitz 和 Stegun 书中(13.4.15)式)

$$z_{n-1}(x) = b_n z_n(x) + a_{n+1} z_{n+1} \tag{6.7.25}$$

其中

$$\begin{aligned} b_n &= 2(n+x) \\ a_{n+1} &= -[(n+1/2)^2 - \nu^2] \end{aligned} \tag{6.7.26}$$

按照得到(5.5.18)式的步骤，得到连分式 CF2

$$\frac{z_1}{z_0} = \frac{1}{b_1 +}\ \frac{a_2}{b_2 +}\cdots \tag{6.7.27}$$

由此从(6.7.24)式给出 $K_{\nu+1}/K_\nu$ 及 K'_ν/K_ν。

Temme 的正则化条件为

$$\sum_{n=0}^{\infty} C_n z_n = \left(\frac{1}{2x}\right)^{\nu+1/2} \tag{6.7.28}$$

其中

$$C_n = \frac{(-1)^n}{n!}\frac{\Gamma(\nu+1/2+n)}{\Gamma(\nu+1/2-n)} \tag{6.7.29}$$

注意 C_n 可由下面的递推式决定：

$$C_0 = 1, \qquad C_{n+1} = -\frac{a_{n+1}}{n+1} C_n \tag{6.7.30}$$

用(6.7.28)式条件求

$$S = \sum_{n=1}^{\infty} C_n \frac{z_n}{z_0} \tag{6.7.31}$$

则

$$z_0 = \left(\frac{1}{2x}\right)^{\nu+1/2} \frac{1}{1+S} \tag{6.7.32}$$

连同(6.7.23)式得到 K_ν。

Thompson 和 Barnett[4]找到一个很好的方法，同时求和(6.7.31)及前向求连分式 CF2。假定连分式求值时得到

$$\frac{z_1}{z_0} = \sum_{n=0}^{\infty} \Delta h_n \tag{6.7.33}$$

其中增量 Δh_n 可由 Steed 算法或 5.2 节中的修正 Lentz 算法得到。S 的前 N 项和的逼近式为

$$S_N = \sum_{n=1}^{N} Q_n \Delta h_n \tag{6.7.34}$$

其中

$$Q_n = \sum_{k=1}^{n} C_k q_k \tag{6.7.35}$$

q_k 则由下面的递推式得到：

$$q_{k+1} = (q_{k-1} - b_k q_k)/a_{k+1} \tag{6.7.36}$$

起始值 $q_0 = 0, q_1 = 1$。对现在的情况，S 收敛所需的项大约是使 CF2 收敛所需项数的三倍。

x 较小时，求 K_ν 和 $K_{\nu+1}$ 可用类似(6.7.14)式的级数：

$$K_\nu = \sum_{k=0}^{\infty} c_k f_k \qquad K_{\nu+1} = \frac{2}{x}\sum_{k=0}^{\infty} c_k h_k \tag{6.7.37}$$

其中

$$\begin{aligned} c_k &= \frac{(x^2/4)^k}{k!} \\ h_k &= -k f_k + p_k \\ p_k &= \frac{p_{k-1}}{k-\nu} \\ q_k &= \frac{q_{k-1}}{k+\nu} \\ f_k &= \frac{k f_{k-1} + p_{k-1} + q_{k-1}}{k^2-\nu^2} \end{aligned} \tag{6.7.38}$$

递推式的初始值为

$$\begin{aligned} p_0 &= \frac{1}{2}\left(\frac{x}{2}\right)^{-\nu}\Gamma(1+\nu) \\ q_0 &= \frac{1}{2}\left(\frac{x}{2}\right)^{\nu}\Gamma(1-\nu) \\ f_0 &= \frac{\nu\pi}{\sin\nu\pi}\left[\cosh\sigma\Gamma_1(\nu) + \frac{\sinh\sigma}{\sigma}\ln\left(\frac{2}{x}\right)\Gamma_2(\nu)\right] \end{aligned} \tag{6.7.39}$$

x 较小时的级数,以及 CF2 和正则关系式(6.7.28)都需 $|\nu| \leqslant 1/2$. 所以在此范围内可将 I_ν 递推到 $\nu=\mu$ 处,求得 K_μ,再将 K_ν 反向递推至 ν 的初始值.

程序假定 $\nu \geqslant 0$. ν 为负数时则用反射公式

$$\begin{aligned} I_{-\nu} &= I_\nu + \frac{2}{\pi}\sin(\nu\pi)K_\nu \\ K_{-\nu} &= K_\nu \end{aligned} \tag{6.7.40}$$

注意,当 x 很大时,$I_\nu \sim e^x$,$K_\nu \sim e^{-x}$,这样该函数会下溢或上溢。较好的办法是调整变量为 $e^{-x}I_\nu$ 和 $e^x K_\nu$。略去(6.7.23)式中的 e^{-x} 因子可保证 4 个量均有合适的调整范围。若使用(6.7.37)式中的级数,在 x 很小时要想调整 4 个量,则仅需对每个级数乘以 e^x。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

void NR::bessik(const DP x, const DP xnu, DP &ri, DP &rk, DP &rip, DP &rkp)
对正数 x 和 xnu = ν≥0,返回修正贝塞尔函数 ri = I_ν,rk = K_ν 及其导数 rip = I'_ν,rkp = K'_ν。相对精度在 EPS 的一至两位有
效数字范围之内。FPMIN 接近机器最小可表示浮点数。
{
    const int MAXIT=10000;
    const DP EPS=numeric_limits<DP>::epsilon();
    const DP FPMIN=numeric_limits<DP>::min()/EPS;
    const DP XMIN=2.0, PI=3.141592653589793;
    DP a,a1,b,c,d,del,del1,delh,dels,e,f,fact,fact2,ff,gam1,gam2,
        gammi,gampl,h,p,pimu,q,q1,q2,qnew,ril,ril1,rimu,rip1,ripl,
        ritemp,rk1,rkmu,rkmup,rktemp,s,sum,sum1,x2,xi,xi2,xmu,xmu2;
    int i,l,nl;

    if (x <= 0.0 || xnu < 0.0) nrerror("bad arguments in bessik");
    nl=int(xnu+0.5);                          nl 为 l 的后向递推次数和 k 的前向递推次数, xmu
    xmu=xnu-nl;                               落在 -1/2 至 1/2 之间。
    xmu2=xmu*xmu;
    xi=1.0/x;
    xi2=2.0*xi;
    h=xnu*xi;                                 由修正 Lentz 方法(参见 5.2 节)求 CF1。
    if (h < FPMIN) h=FPMIN;
    b=xi2*xnu;
    d=0.0;
    c=h;
    for (i=0;i<MAXIT;i++) {
        b += xi2;
        d=1.0/(b+d);                          分母不能为零, 所以不必过于担心。
```

```
    c=b+1.0/c;
    del=c*d;
    h=del*h;
    if (fabs(del-1.0) <= EPS) break;
}
if (i >= MAXIT)
    nrerror("x too large in bessik; try asymptotic expansion");
ril=FPMIN;                                  初始化 I_ν 和 I'_ν 准备向后递推。
ripl=h*ril;
ril1=ril;                                   存储值供以后调整。
ripl=ripl;
fact=xnu*xi;
for (l=nl-1;l >= 0;l--) {
    ritemp=fact*ril+ripl;
    fact -= xi;
    ripl=fact*ritemp+ril;
    ril=ritemp;
}
f=ripl/ril;                                 得到未归一化的 I_μ 和 I'_μ。
if (x < XMIN) {                             使用级数
    x2=0.5*x;
    pimu=PI*xmu;
    fact = (fabs(pimu) < EPS ? 1.0 : pimu/sin(pimu));
    d = -log(x2);
    e=xmu*d;
    fact2 = (fabs(e) < EPS ? 1.0 : sinh(e)/e);
    beschb(xmu,gam1,gam2,gampl,gammi);      求 Γ1 和 Γ2 的切比雪夫数值。
    ff=fact*(gam1*cosh(e)+gam2*fact2*d);    f0。
    sum=ff;
    e=exp(e);
    p=0.5*e/gampl;                          p0。
    q=0.5/(e*gammi);                        q0。
    c=1.0;
    d=x2*x2;
    sum1=p;
    for (i=1;i<=MAXIT;i++) {
        ff=(i*ff+p+q)/(i*i-xmu2);
        c *= (d/i);
        p /= (i-xmu);
        q /= (i+xmu);
        del=c*ff;
        sum += del;
        del1=c*(p-i*ff);
        sum1 += del1;
        if (fabs(del) < fabs(sum)*EPS) break;
    }
    if (i > MAXIT) nrerror("bessk series failed to converge");
    rkmu=sum;
    rki=sum1*xi2;
} else {                                    通过 Steed's 算法 (参见 5.2 节) 求
    b=2.0*(1.0+x);                          CF2,因分母不能为零，故可行。
    d=1.0/b;
    h=delh=d;
    q1=0.0;                                 递推式 (6.7.35) 初始化。
    q2=1.0;
    a1=0.25-xmu2;
    q=c=a1;                                 (6.7.34) 式第一项。
    a = -a1;
    s=1.0+q*delh;
    for (i=1;i<MAXIT;i++) {
        a -= 2*i;
        c = -a*c/(i+1.0);
        qnew=(q1-b*q2)/a;
        q1=q2;
        q2=qnew;
        q += c*qnew;
        b += 2.0;
        d=1.0/(b+a*d);
        delh=(b*d-1.0)*delh;
        h += delh;
```

```
            dels=q*delb;
            s += dels;
            if (fabs(dels/s) <= EPS) break;
            因为CF2本身很快地收敛，所以只需测试和项的收敛性。
        }
        if (i >= MAXIT) nrerror("bessik: failure to converge in cf2");
        h=ai*h;
        rkmu=sqrt(PI/(2.0*x))*exp(-x)/s;          略去因子exp(-x)，当x ≥ XMIN时，用exp(x)调整所
        rk1=rkmu*(xmu+x+0.5-h)*xi;                有返回函数。
    }
    rkmup=xmu*xi*rkmu-rk1;
    rimu=xi/(f*rkmu-rkmup);                       由Wronskian得到I_μ
    ri=(rimu*ril1)/ril;                           调整初始I_ν和I'_ν。
    rip=(rimu*rip1)/ril;
    for (i=1;i <= nl;i++) {                       向前递推K_ν。
        rktemp=(xmu+i)*xi2*rk1+rkmu;
        rkmu=rk1;
        rk1=rktemp;
    }
    rk=rkmu;
    rkp=xmu*xi*rkmu-rk1;
}
```

6.7.3 艾里函数

对正数 x，艾里(Airy)函数定义为

$$\mathrm{Ai}(x) = \frac{1}{\pi}\sqrt{\frac{x}{3}}K_{1/3}(z) \tag{6.7.41}$$

$$\mathrm{Bi}(x) = \sqrt{\frac{x}{3}}[I_{1/3}(z) + I_{-1/3}(z)] \tag{6.7.42}$$

其中

$$z = \frac{2}{3}x^{3/2} \tag{6.7.43}$$

应用反射公式(6.7.40)，可将(6.7.42)式转换成更有用的可计算形式：

$$\mathrm{Bi}(x) = \sqrt{x}\left[\frac{2}{\sqrt{3}}I_{1/3}(z) + \frac{1}{\pi}K_{1/3}(z)\right] \tag{6.7.44}$$

故 Ai 和 Bi 只需调用一次 bessik 即可求出值。

导数不能用上面表达式进行简单地微分求得，因为在 $x=0$ 附近可能会相消。可以使用如下等价形式：

$$\begin{aligned}\mathrm{Ai}'(x) &= -\frac{x}{\pi\sqrt{3}}K_{2/3}(z)\\ \mathrm{Bi}'(x) &= x\left[\frac{2}{\sqrt{3}}I_{2/3}(z) + \frac{1}{\pi}K_{2/3}(z)\right]\end{aligned} \tag{6.7.45}$$

负变量的相应公式为

$$\begin{aligned}\mathrm{Ai}(-x) &= \frac{\sqrt{x}}{2}\left[J_{1/3}(z) - \frac{1}{\sqrt{3}}Y_{1/3}(z)\right]\\ \mathrm{Bi}(-x) &= -\frac{\sqrt{x}}{2}\left[\frac{1}{\sqrt{3}}J_{1/3}(z) + Y_{1/3}(z)\right]\\ \mathrm{Ai}'(-x) &= \frac{x}{2}\left[J_{2/3}(z) + \frac{1}{\sqrt{3}}Y_{2/3}(z)\right]\\ \mathrm{Bi}'(-x) &= \frac{x}{2}\left[\frac{1}{\sqrt{3}}J_{2/3}(z) - Y_{2/3}(z)\right]\end{aligned} \tag{6.7.46}$$

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::airy(const DP x, DP &ai, DP &bi, DP &aip, DP &bip)
返回艾里函数Ai(x)、Bi(x)及其导数Ai'(x)、Bi'(x)。
```

```
{
    const DP PI=3.141592653589793238, ONOVRT=0.577350269189626;
    const DP THIRD=(1.0/3.0), TWOTHR=2.0*THIRD;
    DP absx,ri,rip,rj,rjp,rk,rkp,rootx,ry,ryp,z;

    absx=fabs(x);
    rootx=sqrt(absx);
    z=TWOTHR*absx*rootx;
    if (x > 0.0) {
        bessik(z,THIRD,ri,rk,rip,rkp);
        ai=rootx*ONOVRT*rk/PI;
        bi=rootx*(rk/PI+2.0*ONOVRT*ri);
        bessik(z,TWOTHR,ri,rk,rip,rkp);
        aip = -x*ONOVRT*rk/PI;
        bip=x*(rk/PI+2.0*ONOVRT*ri);
    } else if (x < 0.0) {
        bessjy(z,THIRD,rj,ry,rjp,ryp);
        ai=0.5*rootx*(rj-ONOVRT*ry);
        bi = -0.5*rootx*(ry+ONOVRT*rj);
        bessjy(z,TWOTHR,rj,ry,rjp,ryp);
        aip=0.5*absx*(ONOVRT*ry+rj);
        bip=0.5*absx*(ONOVRT*rj-ry);
    } else {                    Case x = 0.
        ai=0.355028053887817;
        bi=ai/ONOVRT;
        aip = -0.258819403792807;
        bip = -aip/ONOVRT;
    }
}
```

6.7.4 球面贝塞尔函数

对整数 n,球面贝塞尔函数定义为

$$
\begin{aligned}
j_n(x) &= \sqrt{\frac{\pi}{2x}} J_{n+(1/2)}(x) \\
y_n(x) &= \sqrt{\frac{\pi}{2x}} Y_{n+(1/2)}(x)
\end{aligned}
\tag{6.7.47}
$$

只需调用一次 bessjy 即可求其值,导数也可由(6.7.47)式的微分求出。

注意,(6.7.3)式的连分式 CF2 当 $\nu = 1/2$ 时仅存在第一项。这样对球面函数,按程序 bessjy 可找到很简便的算法,通常只需对 j_n 递推到 $n=0$,由 CF2 的第一项确定 p 和 q,然后对 y_n 向前递推。在 $x=0$ 处,不需另外特殊的级数。程序 bessjy 已经完全有效,没有必要为球面贝塞尔函数再去设计一个独立的程序。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::sphbes(const int n, const DP x, DP &sj, DP &sy, DP &sjp, DP &syp)
对整数 n 返回球面贝塞尔函数 j_n(x), y_n(x) 及其导数 j'_n(x), y'_n(x)。
{
    const DP RTPI02=1.253314137315500251;
    DP factor,order,rj,rjp,ry,ryp;

    if (n < 0 || x <= 0.0) nrerror("bad arguments in sphbes");
    order=n+0.5;
    bessjy(x,order,rj,ry,rjp,ryp);
    factor=RTPI02/sqrt(x);
    sj=factor*rj;
    sy=factor*ry;
    sjp=factor*rjp-sj/(2.0*x);
    syp=factor*ryp-sy/(2.0*x);
}
```

参考文献和深入阅读

Barnett, A.R., Feng, D.H., Steed, J.W., and Goldfarb, L.J.B. 1974, *Computer Physics Communications*, vol. 8, pp. 377–395. [1]

Temme, N.M. 1976, *Journal of Computational Physics*, vol. 21, pp. 343–350 [2]; 1975, *op. cit.*, vol. 19, pp. 324–337. [3]

Thompson, I.J., and Barnett, A.R. 1987, *Computer Physics Communications*, vol. 47, pp. 245–257. [4]

Barnett, A.R. 1981, *Computer Physics Communications*, vol. 21, pp. 297–314.

Thompson, I.J., and Barnett, A.R. 1986, *Journal of Computational Physics*, vol. 64, pp. 490–509.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 10.

6.8 球面调和函数

球面调和函数出现在大量物理问题中。例如，波动方程或拉普拉斯(Laplace)方程，就是通过在球面坐标下用变量分离法求解的。球面调和函数 $Y_{lm}(\theta,\phi)$，$-l\leqslant m\leqslant l$，是球面上两个坐标 θ,ϕ 的函数。

对于不同 l 和 m 球面调和函数是正交的，且经过归一化，使之在球面上的平方积分为单位 1：

$$\int_0^{2\pi} d\phi \int_{-1}^{1} d(\cos\theta) Y_{l'm'}{}^*(\theta,\phi) Y_{lm}(\theta,\phi) = \delta_{l'l}\delta_{m'm} \tag{6.8.1}$$

其中星号(*)表示复共轭。

数学上，联系球面调和函数与**连带勒让德(Legendre)多项式**的关系式为

$$Y_{lm}(\theta,\phi) = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta) e^{im\phi} \tag{6.8.2}$$

应用关系式

$$Y_{l,-m}(\theta,\phi) = (-1)^m Y_{lm}{}^*(\theta,\phi) \tag{6.8.3}$$

总能将一个球面调和函数，与一个 $m\geqslant 0$ 的连带勒让德多项式联系起来。取 $x\equiv\cos\theta$，这种多项式由普通勒让德多项式定义(参见 4.5 节和 5.5 节)：

$$P_l^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_l(x) \tag{6.8.4}$$

前面几项连带勒让德多项式及与其对应的归一化后的球面调和函数，如下所示。

$$\begin{array}{ll}
P_0^0(x) = 1 & Y_{00} = \sqrt{\frac{1}{4\pi}} \\
\hline
P_1^1(x) = -(1-x^2)^{1/2} & Y_{11} = -\sqrt{\frac{3}{8\pi}}\sin\theta e^{i\phi} \\
P_1^0(x) = x & Y_{10} = \sqrt{\frac{3}{4\pi}}\cos\theta \\
\hline
P_2^2(x) = 3(1-x^2) & Y_{22} = \frac{1}{4}\sqrt{\frac{15}{2\pi}}\sin^2\theta e^{2i\phi} \\
P_2^1(x) = -3(1-x^2)^{1/2}x & Y_{21} = -\sqrt{\frac{15}{8\pi}}\sin\theta\cos\theta e^{i\phi} \\
P_2^0(x) = \frac{1}{2}(3x^2-1) & Y_{20} = \sqrt{\frac{5}{4\pi}}(\frac{3}{2}\cos^2\theta - \frac{1}{2}) \\
\hline
\end{array} \tag{6.8.5}$$

数值计算连带勒让德多项式的方法很多，但许多方法都不很好。例如，有显式表达式如

$$\begin{aligned}
P_l^m(x) = \frac{(-1)^m(l+m)!}{2^m m!(l-m)!}(1-x^2)^{m/2}\Bigg[1 - \frac{(l-m)(m+l+1)}{1!(m+1)}\left(\frac{1-x}{2}\right) \\
+ \frac{(l-m)(l-m-1)(m+l+1)(m+l+2)}{2!(m+1)(m+2)}\left(\frac{1-x}{2}\right)^2 - \cdots\Bigg]
\end{aligned} \tag{6.8.6}$$

其中多项式只到含有 $(1-x)^{l-m}$ 的项(关于此式及相关公式参见本节参考文献[1])。这种方法不能叫人满

意，因为逐项符号交替，因此求多项式值时，符号相反的连续项可巧妙地消去。l 很大时，多项式中单独项却比和式大得多，而且精度很差。

实际应用中依精度要求，l 达到6或8时，(6.8.6)式按单精度(32位)使用；l 达到15或18时，按双精度(64位)使用。因而希望能找到一种更有效的计算方法，如下述。

连带勒让德多项式满足很多递推关系[1,2]。有仅对 l 递推的，有仅对 m 递推的，也有同时对 l 和 m 递推的。大多数对 m 的递推不稳定，不适于数值计算。下面这个递推式对 l 的递推是稳定的，试与(5.5.1)式比较：

$$(l-m)P_l^m = x(2l-1)P_{l-1}^m - (l+m-1)P_{l-2}^m \tag{6.8.7}$$

这个式子很有用，因为对初始值存在闭形表达式

$$P_m^m = (-1)^m(2m-1)!!(1-x^2)^{m/2} \tag{6.8.8}$$

(记号$n!!$表示小于等于 n 的所有奇整数的乘积。)对(6.8.7)式，令 $l=m+1$ 并设 $P_{m-1}^m=0$，得到

$$P_{m+1}^m = x(2m+1)P_m^m \tag{6.8.9}$$

(6.8.7)式中对一般 l 所要求的两个初始值可由(6.8.8)式和(6.8.9)式给出。

实现该算法的函数为：

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::plgndr(const int l, const int m, const DP x)
计算连带勒让德多项式 P_l^m(x)。其中 m 和 l 为整数，满足 0≤m≤l，x 满足 -1≤x≤1。
{
    int i,ll;
    DP fact,pll,pmm,pmmp1,somx2;

    if (m < 0 || m > l || fabs(x) > 1.0)
        nrerror("Bad arguments in routine plgndr");
    pmm=1.0;                    计算 P_m^m。
    if (m > 0) {
        somx2=sqrt((1.0-x)*(1.0+x));
        fact=1.0;
        for (i=1;i<=m;i++) {
            pmm *= -fact*somx2;
            fact += 2.0;
        }
    }
    if (l == m)
        return pmm;
    else {                      计算 P_{m+1}^m。
        pmmp1=x*(2*m+1)*pmm;
        if (l == (m+1))
            return pmmp1;
        else {                  计算 P_l^m, l > m+1。
            for (ll=m+2;ll<=l;ll++) {
                pll=(x*(2*ll-1)*pmmp1-(ll+m-1)*pmm)/(ll-m);
                pmm=pmmp1;
                pmmp1=pll;
            }
            return pll;
        }
    }
}
```

参考文献和深入阅读

Magnus, W., and Oberhettinger, F. 1949, *Formulas and Theorems for the Functions of Mathematical Physics* (New York: Chelsea), pp. 54ff. [1]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 8. [2]

6.9 Fresnel 积分、余弦和正弦积分

6.9.1 Fresnel 积分

定义两个 Fresnel(菲涅耳)积分为

$$C(x)=\int_0^x \cos\left(\frac{\pi}{2}t^2\right)dt,\qquad S(x)=\int_0^x \sin\left(\frac{\pi}{2}t^2\right)dt \tag{6.9.1}$$

x 较小时,对任意的精度要求,计算函数值最方便的方法是幂级函数;x 较大时,则用连分式。级数为

$$\begin{aligned}C(x)&=x-\left(\frac{\pi}{2}\right)^2\frac{x^5}{5\cdot 2!}+\left(\frac{\pi}{2}\right)^4\frac{x^9}{9\cdot 4!}-\cdots\\ S(x)&=\left(\frac{\pi}{2}\right)\frac{x^3}{3\cdot 1!}-\left(\frac{\pi}{2}\right)^3\frac{x^7}{7\cdot 3!}+\left(\frac{\pi}{2}\right)^5\frac{x^{11}}{11\cdot 5!}-\cdots\end{aligned} \tag{6.9.2}$$

存在一种复连分式,可同时求出 $S(x)$和 $C(x)$:

$$C(x)+iS(x)=\frac{1+i}{2}\operatorname{erf} z,\qquad z=\frac{\sqrt{\pi}}{2}(1-i)x \tag{6.9.3}$$

其中

$$\begin{aligned}e^{z^2}\operatorname{erfc} z&=\frac{1}{\sqrt{\pi}}\left(\frac{1}{z+}\,\frac{1/2}{z+}\,\frac{1}{z+}\,\frac{3/2}{z+}\,\frac{2}{z+}\cdots\right)\\ &=\frac{2z}{\sqrt{\pi}}\left(\frac{1}{2z^2+1-}\,\frac{1\cdot 2}{2z^2+5-}\,\frac{3\cdot 4}{2z^2+9-}\cdots\right)\end{aligned} \tag{6.9.4}$$

上式第二行中将连分式由通常形式转换为偶次形式(参见 5.2 节),这样收敛快两倍。必须小心,不要在 x 太大时,求交错级数(6.9.2)式的值。从级数项中可以看出,$x=1.5$ 是切换到使用连分式比较合适的点。

注意,当 x 很大时,

$$C(x)\sim\frac{1}{2}+\frac{1}{\pi x}\sin\left(\frac{\pi}{2}x^2\right),\qquad S(x)\sim\frac{1}{2}-\frac{1}{\pi x}\cos\left(\frac{\pi}{2}x^2\right) \tag{6.9.5}$$

这样,程序 frenel 的精度可能会受到 x 较大时库函数中正弦和余弦精度的影响。

```
#include <cmath>
#include <complex>
#include <limits>
#include "nr.h"
using namespace std;

void NR::frenel(const DP x, complex<DP> &cs)
对所有实数 x 计算 Fresnel 积分 S(x)和 C(x)。C(x)为 cs 的实部,S(x)为虚部。
{
    const int MAXIT=100;
    const DP EPS=numeric_limits<DP>::epsilon();
    const DP FPMIN=numeric_limits<DP>::min();
    const DP BIG=numeric_limits<DP>::max()*EPS;
    const DP PI=3.141592653589793238, PIBY2=(PI/2.0), XMIN=1.5;
    其中 EPS 为相对误差。MAXIT 为允许最大迭代次数。FPMIN 为最小可表示的浮点数的邻近数。BIG 为接近机器上
    溢极限的数。XMIN 为使用级数和连分式之间的区分线。
    bool odd;
    int k,n;
    DP a,ax,fact,pix2,sign,sum,sumc,sums,term,test;
    complex<DP> b,cc,d,h,del;

    ax=fabs(x);
    if (ax < sqrt(FPMIN)) {                    特殊情况,避免因为下溢而使收敛测试失败。
        cs=ax;
    } else if (ax <= XMIN) {                   同时求两级数值。
        sum=sums=0.0;
        sumc=ax;
        sign=1.0;
```

```
            fact=PIBY2*ax*ax;
            odd=true;
            term=ax;
            n=3;
            for (k=1;k<=MAXIT;k++) {
                term *= fact/k;
                sum += sign*term/n;
                test=fabs(sum)*EPS;
                if (odd) {
                    sign = -sign;
                    sums=sum;
                    sum=sumc;
                } else {
                    sumc=sum;
                    sum=sums;
                }
                if (term < test) break;
                odd=!odd;
                n += 2;
            }
            if (k > MAXIT) nrerror("series failed in frenel");
            cs=complex<DP>(sumc,sums);
        } else {                                    由修正Lentz方法求连分式值(5.2节)。
            pix2=PI*ax*ax;
            b=complex<DP>(1.0,-pix2);
            cc=BIG;
            d=h=1.0/b;
            n = -1;
            for (k=2;k<=MAXIT;k++) {
                n += 2;
                a = -n*(n+1);
                b += 4.0;
                d=1.0/(a*d+b);                      分母不能为零。
                cc=b+a/cc;
                del=cc*d;
                h *= del;
                if (fabs(real(del)-1.0)+fabs(imag(del)) <= EPS) break;
            }
            if (k > MAXIT) nrerror("cf failed in frenel");
            h *= complex<DP>(ax,-ax);
            cs=complex<DP>(0.5,0.5)
                *(1.0-complex<DP>(cos(0.5*pix2),sin(0.5*pix2))*h);
        }
        if (x < 0.0) {                              应用反对称性。
            cs = -cs;
        }
        return;
    }
```

6.9.2 余弦和正弦积分

余弦和正弦积分定义为

$$\begin{aligned}\mathrm{Ci}(x) &= \gamma + \ln x + \int_0^x \frac{\cos t - 1}{t}\,dt \\ \mathrm{Si}(x) &= \int_0^x \frac{\sin t}{t}\,dt\end{aligned} \tag{6.9.6}$$

其中 $\gamma \approx 0.5772\ldots$ 为欧拉常数，因

$$\mathrm{Si}(-x) = -\mathrm{Si}(x), \qquad \mathrm{Ci}(-x) = \mathrm{Ci}(x) - i\pi \tag{6.9.7}$$

故仅需对 $x>0$ 找出求函数的方法。

还可以用幂级数和复连分式相结合的方法求此函数值。级数为

$$\begin{aligned}\mathrm{Si}(x) &= x - \frac{x^3}{3\cdot 3!} + \frac{x^5}{5\cdot 5!} - \cdots \\ \mathrm{Ci}(x) &= \gamma + \ln x + \left(-\frac{x^2}{2\cdot 2!} + \frac{x^4}{4\cdot 4!} - \cdots\right)\end{aligned} \tag{6.9.8}$$

指数积分$E_1(ix)$的连分式为

$$
\begin{aligned}
E_1(ix) &= -\mathrm{Ci}(x) + i[\mathrm{Si}(x) - \pi/2] \\
&= e^{-ix}\left(\frac{1}{ix+}\,\frac{1}{1+}\,\frac{1}{ix+}\,\frac{2}{1+}\,\frac{2}{ix+}\cdots\right) \\
&= e^{-ix}\left(\frac{1}{1+ix-}\,\frac{1^2}{3+ix-}\,\frac{2^2}{5+ix-}\cdots\right)
\end{aligned}
\tag{6.9.9}
$$

上式中最后一行是连分式的偶形式,对同样计算量它收敛可快两倍。现在从交错级数到连分式的转换点为$x=2$。当x很大时,Fresnel积分的精度会受到正弦和余弦程序的精度的限制。

```
#include <cmath>
#include <complex>
#include <limits>
#include "nr.h"
using namespace std;

void NR::cisi(const DP x, complex<DP> &cs)
计算余弦和正弦积分 Ci(x)和 Si(x)。Ci(x)为 cs 的实部,Si(x)为虚部。Ci(0)返回一个大负数,不给出错误信息。x < 0
时程序返回 Ci(-x),用户必须自己另加 -ix 因子。
{
    const int MAXIT=100;                          允许最大迭代次数。
    const DP EULER=0.577215664901533, PIBY2=1.570796326794897, TMIN=2.0;
    const DP EPS=numeric_limits<DP>::epsilon();
    const DP FPMIN=numeric_limits<DP>::min()*4.0;
    const DP BIG=numeric_limits<DP>::max()*EPS;
    Here EULER is Euler's constant γ; PIBY2 is π/2; TMIN is the dividing line between using
    the series and continued fraction; EPS is the relative error, or absolute error near a zero of
    Ci(x); FPMIN is a number close to the smallest representable floating-point number; BIG is
    a number near the machine overflow limit.
    int i,k;
    bool odd;
    DP a,err,fact,sign,sum,sumc,sums,t,term;
    complex<DP> h,b,c,d,del;

    t=fabs(x);
    if (t == 0.0) {                               特殊情况。


        cs= -BIG;
        return;
    }
    if (t > TMIN) {                               由修正 Lentz 方法求连分式(5.2 节)。
        b=complex<DP>(1.0,t);
        c=complex<DP>(BIG,0.0);
        d=h=1.0/b;
        for (i=1;i<MAXIT;i++) {
            a= -i*i;
            b += 2.0;
            d=1.0/(a*d+b);                        分母不能为零。
            c=b+a/c;
            del=c*d;
            h *= del;
            if (fabs(real(del)-1.0)+fabs(imag(del)) <= EPS) break;
        }
        if (i >= MAXIT) nrerror("cf failed in cisi");
        h=complex<DP>(cos(t),-sin(t))*h;
        cs= -conj(h)+complex<DP>(0.0,PIBY2);
    } else {                                      同时求两级数值。
        if (t < sqrt(FPMIN)) {                    特殊情况,避免因为下溢出现而使收敛检测失败。
            sumc=0.0;
            sums=t;
        } else {
            sum=sums=sumc=0.0;
            sign=fact=1.0;
            odd=true;
            for (k=1;k<=MAXIT;k++) {
                fact *= t/k;
```

```
                    term=fact/k;
                    sum += sign*term;
                    err=term/fabs(sum);
                    if (odd) {
                        sign = -sign;
                        sums=sum;
                        sum=sumc;
                    } else {
                        sumc=sum;
                        sum=sums;
                    }
                    if (err < EPS) break;
                    odd=!odd;
                }
                if (k > MAXIT) nrerror("maxits exceeded in cisi");
            }
            cs=complex<DP>(sumc+log(t)+EULER,sums);
        }
        if (x < 0.0) cs = conj(cs);
    }
```

参考文献和深入阅读

Stegun, I.A., and Zucker, R. 1976, *Journal of Research of the National Bureau of Standards*, vol. 80B, pp. 291–311; 1981, *op. cit.*, vol. 86, pp. 661–686.

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapters 5 and 7.

6.10 Dawson 积分

Dawson 积分 $F(x)$定义为

$$F(x) = e^{-x^2}\int_0^x e^{t^2}\,dt \tag{6.10.1}$$

此函数与复误差函数关系为

$$F(z) = \frac{i\sqrt{\pi}}{2}e^{-z^2}\left[1-\operatorname{erfc}(-iz)\right] \tag{6.10.2}$$

Rybicki[1]找到一个极其接近 $F(z)$的表达

$$F(z) = \lim_{h\to 0}\frac{1}{\sqrt{\pi}}\sum_{n\ \mathrm{odd}}\frac{e^{-(z-nh)^2}}{n} \tag{6.10.3}$$

(6.10.3)式独特之处在于,其精度在 h 较小时呈**指数**增长,所以一般的 h 值(级数有相当快的收敛速度)就能得到相当精确的逼近结果。

稍后在 13.11 节中将讨论导出(6.10.3)式的理论,它是傅里叶方法的一个有趣应用。这里仅给出基于公式的实现程序。

将和式的下标移至指数项的最大项附近是最方便的。所以定义 n_0 为最接近 x/h 的偶整数,$x_0 \equiv n_0 h$,$x' \equiv x - x_0$和 $n' \equiv n - n_0$,则有

$$F(x) \approx \frac{1}{\sqrt{\pi}}\sum_{\substack{n'=-N\\ n'\ \mathrm{odd}}}^{N}\frac{e^{-(x'-n'h)^2}}{n'+n_0} \tag{6.10.4}$$

h 足够小并 N 充分大时,这个近似等号是足够精确的。若注意到下式

$$e^{-(x'-n'h)^2} = e^{-x'^2}e^{-(n'h)^2}\left(e^{2x'h}\right)^{n'} \tag{6.10.5}$$

则(6.10.4)式的计算速度还可以大大加快。第一个因子只需计算一次,第二项则为可存储的常数数组,第三项可递推地计算,因此只需计算两个指数项。另外还有一个优点就是,将求和号按 n' 分成正值和负值两部分时,系数 $e^{-(n'h)^2}$ 具有对称性。

下面程序中选定 $h=0.4$、$N=11$。因为求和的对称性及 n 限定为奇数,所以 for 循环限定为 0~5。在浮点型版本中,此程序计算结果的精度大约为 2×10^{-7}。$x=0$ 附近,$F(x)$ 值为零,为保证精度,程序中在 $|x|<0.2$ 时,用 $F(x)$ 的幂级数[2]求值。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::dawson(const DP x)
对任何实数 x 返回 Dawson 积分 F(x) = exp(-x^2)∫_0^x exp(t^2)dt。
{
    const int NMAX=6;
    const DP H=0.4, A1=2.0/3.0, A2=0.4, A3=2.0/7.0;
    int i,n0;
    static bool init = true;        若需初始化,标志为0,否则为1。
    DP d1,d2,e1,e2,sum,x2,xp,xx,ans;
    static Vec_DP c(NMAX);

    if (init) {
        init=false;
        for (i=0;i<NMAX;i++) c[i]=exp(-SQR((2.0*i+1.0)*H));
    }
    if (fabs(x) < 0.2) {            利用级数扩展。
        x2=x*x;
        ans=x*(1.0-A1*x2*(1.0-A2*x2*(1.0-A3*x2)));
    } else {                        利用取样定理表达式。
        xx=fabs(x);
        n0=2*int(0.5*xx/H+0.5);
        xp=xx-n0*H;
        e1=exp(2.0*xp*H);
        e2=e1*e1;
        d1=n0+1;
        d2=d1-2.0;
        sum=0.0;
        for (i=0;i<NMAX;i++,d1+=2.0,d2-=2.0,e1*=e2)
            sum += c[i]*(e1/d1+1.0/(d2*e1));
        ans=0.5641895835*SIGN(exp(-xp*xp),x)*sum;        常数为1/√π。
    }
    return ans;
}
```

还有其他一些计算 Dawson 积分的方法,见本节参考文献[2,3]。

参考文献和深入阅读

Rybicki, G.B. 1989, *Computers in Physics*, vol. 3, no. 2, pp. 85–87. [1]

Cody, W.J., Pociorek, K.A., and Thatcher, H.C. 1970, *Mathematics of Computation*, vol. 24, pp. 171–178. [2]

McCabe, J.H. 1974, *Mathematics of Computation*, vol. 28, pp. 811–816. [3]

6.11 椭圆积分和雅可比椭圆函数

在很多应用中要用到椭圆积分,因为任何形式如积分都可由椭圆积分求得:

$$\int R(t,s)\,dt \tag{6.11.1}$$

其中 R 为 t 和 s 的有理函数,s 为 t 的三次或四次多项式的平方根。一般文献[1]中描述了如何进行简化,这个最初工作是勒让德完成的。勒让德证明了只需要三个基本椭圆函数。其中最简单的一种是

$$I_1=\int_y^x \frac{dt}{\sqrt{(a_1+b_1t)(a_2+b_2t)(a_3+b_3t)(a_4+b_4t)}} \tag{6.11.2}$$

上式中，将 s^2 的四次方写成因子相乘形式。标准积分表[2]中，积分限总有一个是四次方的零点，另外一个积分限则是靠近下一个零点，这样在积分区间内没有奇异点。求积分 I_1，可将积分区间 $[y, x]$ 分成子区间，每个区间的起点和终点都是奇异点。这样，只需区分将 4 个零点(按照阶的大小)分别作为积分上下限的 8 种情况。此外，当(6.11.2)式中有一个 b 为零时，四次方简化变为三次方，最大或最小奇异点移至 $\pm\infty$ 处，这样会产生多于 8 种情况(实际上，有些为前 8 种情况的特例)。根据第一类勒让德标准椭圆积分(下面将要定义)，一般将 16 种情况制成表。积分变量 t 改变时，四次方的零点映射到实轴上的标准位置。这样只需两个无量纲的参量就可为勒让德积分制表。然而勒让德的表示形式掩盖了在根置换下原积分式(6.11.2)的对称性。后面再讨论勒让德表示。首先给出一个较好的方法：

Carlson[3]给出了第一类椭圆标准积分的新定义，

$$R_F(x,y,z)=\frac{1}{2}\int_0^\infty\frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}\tag{6.11.3}$$

其中 x，y 和 z 非负且最多有一个零点。通过将积分区间标准化，可保留零点的置换对称性(Weierstrass 的正则形式也具有此性质)。Carlson 首先证明了，若 x 或 y 是(6.11.2)式四次多项式的一个零点，则积分 I_1 可根据 R_F 写成在**剩余**三个零点置换下仍对称的形式。一般情况下若 x 或 y 都不为零点，则两个这样的 R_F 函数可由**加法定理**合成一个，导出下面的基本公式：

$$I_1=2R_F(U_{12}^2,U_{13}^2,U_{14}^2)\tag{6.11.4}$$

其中

$$U_{ij}=(X_iX_jY_kY_m+Y_iY_jX_kX_m)/(x-y)\tag{6.11.5}$$

$$X_i=(a_i+b_ix)^{1/2},\qquad Y_i=(a_i+b_iy)^{1/2}\tag{6.11.6}$$

i,j,k,m 可由 1,2,3,4 的任意排列置换。一个更简便的计算方法是

$$\begin{aligned}U_{13}^2&=U_{12}^2-(a_1b_4-a_4b_1)(a_2b_3-a_3b_2)\\U_{14}^2&=U_{12}^2-(a_1b_3-a_3b_1)(a_2b_4-a_4b_2)\end{aligned}\tag{6.11.7}$$

其中各个 U 分别对应于将 4 个零点分成一对一对的三种方式，所以 I_1 在零点置换下有明显的对称性。因此，当有一个积分限为零时，(6.11.4)式重新产生所有 16 种情况，也包括了上下限都不为零的情况。

这样 Carlson 函数允许有任何积分区间，并且在积分区间中，被积函数的分支点的位置也是任意的。为了处理第二类和第三类椭圆积分，Carlson 定义了第三类标准积分为

$$R_J(x,y,z,p)=\frac{3}{2}\int_0^\infty\frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}}\tag{6.11.8}$$

它对于 x，y 和 z 对称。当两个参量相等时，退化形式可表示为

$$R_D(x,y,z)=R_J(x,y,z,z)\tag{6.11.9}$$

它对 x 和 y 对称。可用函数 R_D 取代第二类勒让德积分、R_E 的退化形式表示为

$$R_C(x,y)=R_F(x,y,y)\tag{6.11.10}$$

其中包含对数、反三角和反双曲线函数。

Carlson[4~7]根据(6.11.1)式中四次项的线性因子指数给出了积分表。例如，指数为$(\frac{1}{2},\frac{1}{2},-\frac{1}{2},-\frac{3}{2})$的积分，可表示为 R_D 的一个单积分，它说明了 Gradshteyn 和 Ryzhik[2]列举的 144 种情况。

将椭圆积分降为标准型时的一些实用性细节，如处理复共轭零点，可参见 Carlson 的论文[3~7]。

现在，讨论椭圆积分的数值算法。通常使用高斯或兰登(Landen)变换。**降阶**变换可将勒让德积分的模数 k 减少而趋于零，而**增阶**变换则可增大而趋于 1。在这些极限位置，函数可解析地表达。这些方法以四次方收敛，对第一类和第二类求积分很有效，而对第三类积分的某些情况可能不太精确。Carlson 算法[9,10]则对三类积分提供一种统一的方法，而没有明显的消去律。

算法的关键是**加倍定理**：

$$R_F(x,y,z) = 2R_F(x+\lambda, y+\lambda, z+\lambda) = R_F\left(\frac{x+\lambda}{4}, \frac{y+\lambda}{4}, \frac{z+\lambda}{4}\right) \tag{6.11.11}$$

其中

$$\lambda = (xy)^{1/2} + (xz)^{1/2} + (yz)^{1/2} \tag{6.11.12}$$

仅需变换积分变量即可证明此定理[11]。(6.11.11)式可反复迭代,直到 R_F 中的变量几乎相等为止。变量相等时有

$$R_F(x,x,x) = x^{-1/2} \tag{6.11.13}$$

若变量足够接近,则可由(6.11.13)式固定的泰勒(Taylor)展开式中第五次项来求函数值。算法中迭代部分只是线性地收敛,误差在每次迭代中最终按因子 $4^6 = 4096$ 减少。一般只需迭代 2~3 次,若变量初始值相差很大,则可能需要迭代 6~7 次。下面列出 R_F 的算法,其他情况可参阅 Carlson 的论文[9]。

第一步:对 $n = 0,1,2,\ldots$,计算

$$\mu_n = (x_n + y_n + z_n)/3$$

$$X_n = 1 - (x_n/\mu_n), \quad Y_n = 1 - (y_n/\mu_n), \quad Z_n = 1 - (z_n/\mu_n)$$

$$\epsilon_n = \max(|X_n|, |Y_n|, |Z_n|)$$

若 $\epsilon_n <$ tol 到第二步,否则计算

$$\lambda_n = (x_n y_n)^{1/2} + (x_n z_n)^{1/2} + (y_n z_n)^{1/2}$$

$$x_{n+1} = (x_n + \lambda_n)/4, \quad y_{n+1} = (y_n + \lambda_n)/4, \quad z_{n+1} = (z_n + \lambda_n)/4$$

并在此步内循环。

第二步:计算

$$E_2 = X_n Y_n - Z_n^2, \quad E_3 = X_n Y_n Z_n$$

$$R_F = (1 - \tfrac{1}{10}E_2 + \tfrac{1}{14}E_3 + \tfrac{1}{24}E_2^2 - \tfrac{3}{44}E_2E_3)/(\mu_n)^{1/2}$$

有时候 R_J 中的变量 p 或 R_C 中的变量 y 为负数,而且需要积分的柯西(Cauchy)主值。这时可由如下公式容易求得:

$$R_J(x,y,z,p) = [(\gamma - y)R_J(x,y,z,\gamma) - 3R_F(x,y,z) + 3R_C(xz/y, p\gamma/y)]/(y-p) \tag{6.11.14}$$

其中

$$\gamma \equiv y + \frac{(z-y)(y-x)}{y-p} \tag{6.11.15}$$

p 为负数时上式为正数,且

$$R_C(x,y) = \left(\frac{x}{x-y}\right)^{1/2} R_C(x-y, -y) \tag{6.11.16}$$

$p<0$ 的某些情况,R_J 的柯西主值可能为零,因此(6.11.14)式在零附近可能不精确。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::rf(const DP x, const DP y, const DP z)
计算第一类 Carlson 椭圆积分 Rf(x,y,z)。x,y 和 z 必须为非负,且至多只能有一个为零。TINY 必须至少为机器下溢极
限的 5 倍,BIG 则必须最多为机器上溢极限的 1/5。
{
    const DP ERRTOL=0.0025, TINY=1.5e-38, BIG=3.0e37, THIRD=1.0/3.0;
    const DP C1=1.0/24.0, C2=0.1, C3=3.0/44.0, C4=1.0/14.0;
    DP alamb,ave,delx,dely,delz,e2,e3,sqrtx,sqrty,sqrtz,xt,yt,zt;

    if (MIN(MIN(x,y),z) < 0.0 || MIN(MIN(x+y,x+z),y+z) < TINY ||
        MAX(MAX(x,y),z) > BIG)
            nrerror("invalid arguments in rf");
```

```
    xt=x;
    yt=y;
    zt=z;
    do {
        sqrtx=sqrt(xt);
        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        zt=0.25*(zt+alamb);
        ave=THIRD*(xt+yt+zt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
    } while (MAX(MAX(fabs(delx),fabs(dely)),fabs(delz)) > ERRTOL);
    e2=delx*dely-delz*delz;
    e3=delx*dely*delz;
    return (1.0+(C1*e2-C2-C3*e3)*e2+C4*e3)/sqrt(ave);
}
```

在双精度(16位有效数字)的情况,允许的误差精度参数取为0.0025已经足够了。因为误差尺度为ϵ_n^6,可以看出0.08将产生单精度(7位有效数字),并且最多需要两到三次迭代。因为对于其他椭圆函数,第六阶截尾误差的系数是不同的,所以R_C算法中允许的误差值改为0.04(单精度)和0.0012(双精度),R_J和R_D中分别改为0.05和0.0015。R_F的算法除了用于结合某些基本函数外,也反复用于计算R_J。

在C++语言的实现程序中,根据两个依赖于机器的常数(TINY和BIG)来检测输入变量,以防止计算中出现上溢和下溢。对应于机器的最小值为3×10^{-39}和最大值1.7×10^{38},我们较保守地选取TINY和BIG值。应用下面给出的齐次性关系式(6.11.22)可以扩展允许的变量值的范围。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::rd(const DP x, const DP y, const DP z)
计算第一类 Carlson 椭圆积分 R_D(x,y,z)。x 和 y 必须非负,且最多一个为零,z 必须为正值。TINY 必须至少是机器上溢
极限的负 2/3 次幂的 2 倍。BIG 必须最多是 0.1 × ERRTOL 再乘以机器下溢极限负 2/3 次幂。
{
    const DP ERRTOL=0.0015, TINY=1.0e-25, BIG=4.5e21;
    const DP C1=3.0/14.0, C2=1.0/6.0, C3=9.0/22.0;
    const DP C4=3.0/26.0, C5=0.25*C3, C6=1.5*C4;
    DP alamb,ave,delx,dely,delz,ea,eb,ec,ed,ee,fac,sqrtx,sqrty,
        sqrtz,sum,xt,yt,zt;

    if (MIN(x,y) < 0.0 || MIN(x+y,z) < TINY || MAX(MAX(x,y),z) > BIG)
        nrerror("invalid arguments in rd");
    xt=x;
    yt=y;
    zt=z;
    sum=0.0;
    fac=1.0;
    do {
        sqrtx=sqrt(xt);
        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz;
        sum += fac/(sqrtz*(zt+alamb));
        fac=0.25*fac;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        zt=0.25*(zt+alamb);
        ave=0.2*(xt+yt+3.0*zt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
    } while (MAX(MAX(fabs(delx),fabs(dely)),fabs(delz)) > ERRTOL);
    ea=delx*dely;
    eb=delz*delz;
```

```
    ec=ea-eb;
    ed=ea-6.0*eb;
    ee=ed+ec+ec;
    return 3.0*sum+fac*(1.0+ed*(-C1+C5*ed-C6*delz*ee)
        +delz*(C2*ee+delz*(-C3*ec+delz*C4*ea)))/(ave*sqrt(ave));
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::rj(const DP x, const DP y, const DP z, const DP p)
```

计算第三类 Carlson 椭圆积分 $R_J(x,y,z,p)$。x,y 和 z 必须为非负,最多一个为零,p 必须不为零。若 $p<0$,返回柯西主值。TINY 必须至少是机器下溢极限的立方根的 2 倍。BIG 不能超过机器上溢极限的立方根的 1/5。

```
{
    const DP ERRTOL=0.0015, TINY=2.5e-13, BIG=9.0e11;
    const DP C1=3.0/14.0, C2=1.0/3.0, C3=3.0/22.0, C4=3.0/26.0;
    const DP C5=0.75*C3, C6=1.5*C4, C7=0.5*C2, C8=C3+C3;
    DP a,alamb,alpha,ans,ave,b,beta,delp,delx,dely,delz,ea,eb,ec,ed,ee,
        fac,pt,rcx,rho,sqrtx,sqrty,sqrtz,sum,tau,xt,yt,zt;

    if (MIN(MIN(x,y),z) < 0.0 || MIN(MIN(x+y,x+z),MIN(y+z,fabs(p))) < TINY
        || MAX(MAX(x,y),MAX(z,fabs(p))) > BIG)
            nrerror("invalid arguments in rj");
    sum=0.0;
    fac=1.0;
    if (p > 0.0) {
        xt=x;
        yt=y;
        zt=z;
        pt=p;
    } else {
        xt=MIN(MIN(x,y),z);
        zt=MAX(MAX(x,y),z);
        yt=x+y+z-xt-zt;
        a=1.0/(yt-p);
        b=a*(zt-yt)*(yt-xt);
        pt=yt+b;
        rho=xt*zt/yt;
        tau=p*pt/yt;
        rcx=rc(rho,tau);
    }
    do {
        sqrtx=sqrt(xt);
        sqrty=sqrt(yt);
        sqrtz=sqrt(zt);
        alamb=sqrtx*(sqrty+sqrtz)+sqrty*sqrtz;
        alpha=SQR(pt*(sqrtx+sqrty+sqrtz)+sqrtx*sqrty*sqrtz);
        beta=pt*SQR(pt+alamb);
        sum += fac*rc(alpha,beta);
        fac=0.25*fac;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        zt=0.25*(zt+alamb);
        pt=0.25*(pt+alamb);
        ave=0.2*(xt+yt+zt+pt+pt);
        delx=(ave-xt)/ave;
        dely=(ave-yt)/ave;
        delz=(ave-zt)/ave;
        delp=(ave-pt)/ave;
    } while (MAX(MAX(fabs(delx),fabs(dely)),
        MAX(fabs(delz),fabs(delp))) > ERRTOL);
    ea=delx*(dely+delz)+dely*delz;
    eb=delx*dely*delz;
    ec=delp*delp;
    ed=ea-3.0*ec;
    ee=eb+2.0*delp*(ea-ec);
    ans=3.0*sum+fac*(1.0+ed*(-C1+C5*ed-C6*ee)+eb*(C7+delp*(-C8+delp*C4)
```

```
        +delp*ea*(C2-delp*C3)-C2*delp*ec)/(ave*sqrt(ave));
    if (p <= 0.0) ans=a*(b*ans+3.0*(rcx-rf(xt,yt,zt)));
    return ans;
}


#include <cmath>
#include "nr.h"
using namespace std;

DP NR::rc(const DP x, const DP y)
计算 Carlson 退化椭圆积分 Rc(x,y)。x 必须非负，y 必须非零。若 y<0，返回柯西主值。TINY 必须至少是机器下溢极
限的 5 倍。BIG 不能超过机器最大上溢极限的 1/5。
{
    const DP ERRTOL=0.0012, TINY=1.69e-38, SQRTNY=1.3e-19, BIG=3.0e37;
    const DP TNBG=TINY*BIG, COMP1=2.236/SQRTNY, COMP2=TNBG*TNBG/25.0;
    const DP THIRD=1.0/3.0, C1=0.3, C2=1.0/7.0, C3=0.375, C4=9.0/22.0;
    DP alamb,ave,s,w,xt,yt;

    if (x < 0.0 || y == 0.0 || (x+fabs(y)) < TINY || (x+fabs(y)) > BIG ||
        (y<-COMP1 && x > 0.0 && x < COMP2))
            nrerror("invalid arguments in rc");
    if (y > 0.0) {
        xt=x;
        yt=y;
        w=1.0;
    } else {
        xt=x-y;
        yt= -y;
        w=sqrt(x)/sqrt(xt);
    }
    do {
        alamb=2.0*sqrt(xt)*sqrt(yt)+yt;
        xt=0.25*(xt+alamb);
        yt=0.25*(yt+alamb);
        ave=THIRD*(xt+yt+yt);
        s=(yt-ave)/ave;
    } while (fabs(s) > ERRTOL);
    return w*(1.0+s*s*(C1+s*(C2+s*(C3+s*C4))))/sqrt(ave);
}
```

有时候可能希望以勒让德记号来表示答案，或者也许是已有了以这种方法表示的结果，而希望用上面给出的程序计算其值。这种来回转换并不难。**第一类勒让德椭圆积分**定义为

$$F(\phi,k)\equiv\int_0^{\phi}\frac{d\theta}{\sqrt{1-k^2\sin^2\theta}} \tag{6.11.17}$$

第一类完全椭圆积分由下式给定：

$$K(k)\equiv F(\pi/2,k) \tag{6.11.18}$$

根据 R_F 有

$$\begin{aligned}F(\phi,k)&=\sin\varphi R_F(\cos^2\varphi,1-k^2\sin^2\phi,1)\\K(k)&=R_F(0,1-k^2,1)\end{aligned} \tag{6.11.19}$$

第二类勒让德椭圆积分及**第二类完全椭圆积分**由下式给定：

$$\begin{aligned}E(\phi,k)&\equiv\int_0^{\phi}\sqrt{1-k^2\sin^2\theta}\,d\theta\\&=\sin\phi R_F(\cos^2\phi,1-k^2\sin^2\phi,1)\\&\quad-\tfrac{1}{3}k^2\sin^3\phi R_D(\cos^2\phi,1-k^2\sin^2\phi,1)\\E(k)&\equiv E(\pi/2,k)=R_F(0,1-k^2,1)-\tfrac{1}{3}k^2R_D(0,1-k^2,1)\end{aligned} \tag{6.11.20}$$

最后给出**第三类勒让德椭圆积分**

$$\begin{aligned}\Pi(\phi,n,k) &\equiv \int_0^{\phi}\frac{d\theta}{(1+n\sin^2\theta)\sqrt{1-k^2\sin^2\theta}}\\ &=\sin\phi R_F(\cos^2\phi,1-k^2\sin^2\phi,1)\\ &\quad-\tfrac{1}{3}n\sin^3\phi R_J(\cos^2\phi,1-k^2\sin^2\phi,1,1+n\sin^2\phi)\end{aligned}\tag{6.11.21}$$

(注意,这里 n 的记号习惯与 Abramowitz 和 Stegun[12]的记号相反,其中的 $\sin\alpha$ 即为这里的 k。)

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::ellf(const DP phi, const DP ak)
用 Carlson 函数 R_F 求第一类勒让德椭圆积分 F(φ,k)  变量取值范围为 0≤φ≤π/2,0≤k sin φ≤1.
{
    DP s;

    s=sin(phi);
    return s*rf(SQR(cos(phi)),(1.0-s*ak)*(1.0+s*ak),1.0);
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::elle(const DP phi, const DP ak)
用 Carlson 函数 R_D 和 R_F 求第二类勒让德椭圆积分 E(φ,k)。变量取值范围为 0≤φ≤π/2,0≤k sin φ≤1。
{
    DP cc,q,s;

    s=sin(phi);
    cc=SQR(cos(phi));
    q=(1.0-s*ak)*(1.0+s*ak);
    return s*(rf(cc,q,1.0)-(SQR(s*ak))*rd(cc,q,1.0)/3.0);
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::ellpi(const DP phi, const DP en, const DP ak)
用 Carlson 函数 R_J 和 R_F 求第三类勒让德椭圆积分 Π(φ,n,k)(注意 n 的记号习惯与 Abramowitz 和 Stegun 的相反),变量
取值范围为 0≤φ≤π/2,0≤k sin φ≤1。
{
    DP cc,enss,q,s;

    s=sin(phi);
    enss=en*s*s;
    cc=SQR(cos(phi));
    q=(1.0-s*ak)*(1.0+s*ak);
    return s*(rf(cc,q,1.0)-enss*rj(cc,q,1.0,1.0+enss)/3.0);
}
```

Carlson 函数是 $-\frac{1}{2}$ 阶和 $-\frac{3}{2}$ 阶的齐次函数,所以

$$\begin{aligned}R_F(\lambda x,\lambda y,\lambda z)&=\lambda^{-1/2}R_F(x,y,z)\\ R_J(\lambda x,\lambda y,\lambda z,\lambda p)&=\lambda^{-3/2}R_J(x,y,z,p)\end{aligned}\tag{6.11.22}$$

这样,按勒让德记号表示 Carlson 函数,就将变量置换成升阶形式,利用齐次性将第三个变量映射到 1,然后应用(6.11.19)式~(6.11.21)式。

6.11.1 雅可比椭圆函数

定义雅可比椭圆函数 sn 时,不是考察椭圆积分

$$u(y,k) \equiv u = F(\phi,k) \tag{6.11.23}$$

而是考察其反函数

$$y = \sin\phi = \mathrm{sn}(u,k) \tag{6.11.24}$$

等价地，

$$u = \int_0^{\mathrm{sn}} \frac{dy}{\sqrt{(1-y^2)(1-k^2y^2)}} \tag{6.11.25}$$

当 $k=0$ 时，sn 即为正弦函数。函数 cn 和 dn 由以下关系式定义：

$$\mathrm{sn}^2 + \mathrm{cn}^2 = 1, \qquad k\ \mathrm{sn}^2 + \mathrm{dn}^2 = 1 \tag{6.11.26}$$

下面给出的程序实际上取输入参数 $m_c \equiv k_c^2 = 1-k^2$，同时计算出所有三个函数 sn, cn 和 dn，因为计算三个并不比计算其中任何一个更困难。详细的方法参见本节参考文献[8]。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::sncndn(const DP uu, const DP emmc, DP &sn, DP &cn, DP &dn)
返回雅可比椭圆函数 sn(u,k_c),cn(u,k_c)和 dn (u,k_c)。其中 uu = u,emmc = k_c^2。
{
    const DP CA=1.0e-8;          CA平方的精度。
    bool bo;
    int i,ii,l;
    DP a,b,c,d,emc,u;
    Vec_DP em(13),en(13);

    emc=emmc;
    u=uu;
    if (emc != 0.0) {
        bo=(emc < 0.0);
        if (bo) {
            d=1.0-emc;
            emc /= -1.0/d;
            u *= (d=sqrt(d));
        }
        a=1.0;
        dn=1.0;
        for (i=0;i<13;i++) {
            l=i;
            em[i]=a;
            en[i]=(emc=sqrt(emc));
            c=0.5*(a+emc);
            if (fabs(a-emc) <= CA*a) break;
            emc *= a;
            a=c;
        }
        u *= c;
        sn=sin(u);
        cn=cos(u);
        if (sn != 0.0) {
            a=cn/sn;
            c *= a;
            for (ii=l;ii>=0;ii--) {
                b=em[ii];
                a *= c;
                c *= dn;
                dn=(en[ii]+a)/(b+a);
                a=c/b;
            }
            a=1.0/sqrt(c*c+1.0);
            sn=(sn >= 0.0 ? a : -a);
            cn=c*sn;
        }
        if (bo) {
            a=dn;
            dn=cn;
            cn=a;
            sn /= d;
        }
    } else {
        cn=1.0/cosh(u);
```

```
            dn=cn;
            sn=tanh(u);
        }
    }
```

参考文献和深入阅读

Erdélyi, A., Magnus, W., Oberhettinger, F., and Tricomi, F.G. 1953, *Higher Transcendental Functions*, Vol. II, (New York: McGraw-Hill). [1]

Gradshteyn, I.S., and Ryzhik, I.W. 1980, *Table of Integrals, Series, and Products* (New York: Academic Press). [2]

Carlson, B.C. 1977, *SIAM Journal on Mathematical Analysis*, vol. 8, pp. 231–242. [3]

Carlson, B.C. 1987, *Mathematics of Computation*, vol. 49, pp. 595–606 [4]; 1988, *op. cit.*, vol. 51, pp. 267–280 [5]; 1989, *op. cit.*, vol. 53, pp. 327–333 [6]; 1991, *op. cit.*, vol. 56, pp. 267–280. [7]

Bulirsch, R. 1965, *Numerische Mathematik*, vol. 7, pp. 78–90; 1965, *op. cit.*, vol. 7, pp. 353–354; 1969, *op. cit.*, vol. 13, pp. 305–315. [8]

Carlson, B.C. 1979, *Numerische Mathematik*, vol. 33, pp 1–16. [9]

Carlson, B.C., and Notis, E.M. 1981, *ACM Transactions on Mathematical Software*, vol. 7, pp. 398–403. [10]

Carlson, B.C. 1978, *SIAM Journal on Mathematical Analysis*, vol. 9, p. 524–528. [11]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Chapter 17. [12]

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley), pp. 78–79.

6.12 超几何函数

正如5.14节讨论过的那样，要找到一个计算复超几何函数 ${}_2F_1(a,b,c;z)$ 的快速通用程序是非常困难的。该函数定义为下面的超几何级数的解析延拓：

$$
\begin{aligned}
{}_2F_1(a,b,c;z) = 1 &+ \frac{ab}{c}\frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!} + \cdots \\
&+ \frac{a(a+1)\ldots(a+j-1)b(b+1)\ldots(b+j-1)}{c(c+1)\ldots(c+j-1)}\frac{z^j}{j!} + \cdots
\end{aligned}
\tag{6.12.1}
$$

此级数仅在单位圆 $|z|<1$ 内收敛(参见本节参考文献[1])，但我们感兴趣的不只是该区域。

在5.14节讨论过通过复平面上的直线积分求此函数值的方法。这里仅列出得到的结果程序。

函数 hypgeo 的实现，正如程序中注解描述的那样，是很直接显然的。第16章中对微分的程序 odeint 很好用，甚至不必完全理解：使用 odeint 需要一个为零的全程变量、一次函数的调用，以及对求导程序 hypdrv 预先设定格式。

当然，当 z 值太靠近奇异点1时，函数 hypgeo 会失效(若想靠近此奇异点，或在 ∞ 处的奇异点，可用[1]中15.3节介绍的"线性转换公式")。若远离 $z=1$，且 a,b,c 值适当，则积分往往只需要很少几步即可完成。一般为6步。

```
#include <cmath>
#include <complex>
#include "nr.h"
using namespace std;

complex<DP> aa,bb,cc,z0,dz;                    传递给 hypdrv。

int kmax,kount;                                 由 odeint使用。
DP dxsav;
Vec_DP *xp_p;
Mat_DP *yp_p;

complex<DP> NR::hypgeo(const complex<DP> &a, const complex<DP> &b,
    const complex<DP> &c, const complex<DP> &z)
```

通过直接对复平面上超几何方程积分,对复数 a,b,c 和 z 计算复超几何函数 ${}_2F_1$. 分支切换点取在沿实轴Re $z>1$处.

```
{
    const DP EPS=1.0e-14;                          精度参数。
    int nbad,nok;
    complex<DP> ans,y[2];
    Vec_DP yy(4);

    kmax=0;
    if (norm(z) <= 0.25) {                         使用级数……
        hypser(a,b,c,z,ans,y[1]);
        return ans;
    }
    或为线积分选择起点。
    else if (real(z) < 0.0) z0=complex<DP>(-0.5,0.0);
    else if (real(z) <= 1.0) z0=complex<DP>(0.5,0.0);
    else z0=complex<DP>(0.0,imag(z) >= 0.0 ? 0.5 : -0.5);
    aa=a;                                          通过对 odeint前面的全程变量赋值, 给 hypdrv 传递参数。
    bb=b;
    cc=c;
    dz=z-z0;
    hypser(aa,bb,cc,z0,y[0],y[1]);                 求起始函数和导数。
    yy[0]=real(y[0]);
    yy[1]=imag(y[0]);
    yy[2]=real(y[1]);
    yy[3]=imag(y[1]);
    odeint(yy,0.0,1.0,EPS,0.1,0.0001,nok,nbad,hypdrv,bsstep);
    odeint 的变元是自变数向量、因变量的起始值和终值、精度参数、步长的初始猜想、最小的步长、取定的(返回)成功步
    数和失败步数、导数的程序名及(此处的 Bulirsch-Stoer)步进程序
    y[0]=complex<DP>(yy[0],yy[1]);
    return y[0];
}
```

```
#include <complex>
#include "nr.h"
using namespace std;

void NR::hypser(const complex<DP> &a, const complex<DP> &b,
    const complex<DP> &c, const complex<DP> &z,
    complex<DP> &series, complex<DP> &deriv)
返回超几何级数 ₂F₁ 及其导数,迭代到机器精度。|z|≤1/2 时收敛很快
{
    int n;
    complex<DP> aa,bb,cc,fac,temp;
    deriv=0.0;
    fac=1.0;
    temp=fac;
    aa=a;
    bb=b;
    cc=c;
    for (n=1;n<=1000;n++) {
        fac *= ((aa*bb)/cc);
        deriv += fac;
        fac *= ((1.0/n)*z);
        series=temp+fac;
        if (series == temp) return;
        temp=series;
        aa += 1.0;
        bb += 1.0;
        cc += 1.0;
    }
    nrerror("convergence failure in hypser");
}
```

```
#include <complex>
```

```
#include "nr.h"
using namespace std;

extern complex<DP> aa,bb,cc,z0,dz;          在hypgeo中定义。

void NR::hypdrv(const DP s, Vec_I_DP &yy, Vec_O_DP &dyyds)
计算超几何方程的导数,见文中(5.14.4)式。
{
    complex<DP> z,y[2],dyds[2];

    y[0]=complex<DP>(yy[0],yy[1]);
    y[1]=complex<DP>(yy[2],yy[3]);
    z=z0+s*dz;
    dyds[0]=y[1]*dz;
    dyds[1]=(aa*bb*y[0]-(cc-(aa+bb+1.0)*z)*y[1])*dz/(z*(1.0-z));
    dyyds[0]=real(dyds[0]);
    dyyds[1]=imag(dyds[0]);
    dyyds[2]=real(dyds[1]);
    dyyds[3]=imag(dyds[1]);
}
```

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [1]

第7章 随 机 数

7.0 引言

利用计算机，这种人类所设计的各种机器中最精确、最能做出确切判断的机器，来产生“随机数”，这看上去似乎有些自相矛盾，甚至在概念上是讲不通的。任何程序必将产生出完全可以预计的结果，因而不是真正的“随机”数。

然而，计算机的实用“随机数生成程序”却得到了广泛的使用。关于这个自相矛盾的问题，可以参见Knuth编写的本节参考文献[1]3.5节中的讨论。人们有时把计算机生成的序列称为“伪随机”，而将“随机”一词专用来指本质上的随机物理过程所产生的输出。在此不对伪随机和随机进行严格的区分。

随机性的定义虽不精确，但在计算机生成序列的叙述中指出，产生随机序列的程序应该不同于使用其输出的计算机程序，并且(在所有可测量的方面)与后一程序在统计上是不相关的。换言之，任何两个不同的随机数生成程序，当它们与用户的特定应用程序相结合时，都应该产生统计上相同的结果。如果不是这样，(按用户的观点)至少其中一个不是好的随机数生成程序。

上述解释看起来是循环的，因为它将一个程序与另一个程序相互进行了比较。然而，确实存在一批随机数生成程序，这些程序对很广泛的一类应用程序都能满足定义。根据经验还发现，用物理过程产生的随机数进行检验，能获得统计上相同的结果。

那么从实用的观点来看，随机性是使用者(或程序员)自己认可的事。对某一应用问题来说已足够随机的生成程序未必对另一个应用问题也具有足够的随机性。当然，可以用一些现有的统计测试表来进行测试，其中某些表是很实用的。总的来说，它们在测试相关性方面都工作得比较出色，如同通过应用程序(用户的应用程序)进行测试一样。好的随机数生成程序应该能够通过所有这些测试，或者使用者至少能够知道它们中哪一个会失败，所以使用者应该有能力判断这些生成程序是否与所探讨的情形有关。

关于这一主题的参考资料，建议阅读本节参考文献[1,2]。关于数值方法的书籍(参见本节参考文献[3,4])中，只有很少几本是将随机数作为主题来论述的。

参考文献和深入阅读

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 3, especially §3.5. [1]

Brattey, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation*, 2nd ed. (New York: Springer-Verlag). [2]

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 11. [3]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10. [4]

7.1 一致偏离

一致偏离就是在一特定范围内(典型为0至1之间)的随机数。在这个范围内，任一个数出现的可能性是等同的。换言之，一致偏离大概就是人们所认为的“随机数”。不过，要将一致偏离与其他种类的“随机数”区分开来，例如，将它与给定的均值和标准差的正态(高斯)分布所产生的数进行区分。正如将在后面各节看到的那样，其他种类的偏离几乎总是通过对一个或多个一致偏离进行适当的运算而生成的。所以这一节的主题是随

机一致偏离,它是建立任何一种随机模型或蒙特卡罗(Monte Carlo)计算工作的基本构造模块。

7.1.1 系统提供的随机数生成程序

C++从 ANSI C 库继承了一对函数,用来初始化并生成"随机数"。在 cstdlib 中,典型的声明为

```
#include <cstdlib>
#define RAND_MAX ...

void srand(unsigned seed);
int rand(void);
```

给 seed 赋一个任意的初始值,通过调用 srand(seed)来初始化随机数成生程序。每个初始值都会典型地返回不同的随机序列,或者至少是某个极长序列的不同的子序列。不过对相同的初始值 seed 总是返回相同的随机序列。

通过连续调用 rand(),可以在序列中得到连续的随机数。这个函数返回一个整数,它通常处于 0 至 RAND_MAX 之间(含后者)。通常,RAND_MAX 表示整型所能表示的最大正值,但有时它只代表短整型的最大值。如果需在 0.0(含 0.0)和 1.0(不含 1.0)之间得到一个随机 double 值,可用如下的表达式:

```
x = rand()/(RAND_MAX+1.0);
```

在本章中最重要的内容是,对系统提供的 rand()产生很大的怀疑,因为它同我们刚刚叙述的内容很类似。由于不合理的函数 rand()使一些科学论文的结果值得怀疑。因为系统提供的 rand()几乎总是"线性同余生成程序",它是按以下的递推关系生成整数序列 $I_1, I_2, I_3, \ldots$,每个数都在 0 和 $m-1$ 之间(例如,RAND_MAX):

$$I_{j+1} = aI_j + c \pmod m \tag{7.1.1}$$

其中 m 称为"模",a 和 c 是正整数,分别称为"乘子"和"增量"。递推式(7.1.1)一直反复进行,其周期显然不超过 m。如果恰当地选取 m、a 和 c,则将获得最大周期为长度 m。在这种情况下,0 与 $m-1$ 之间所有可能的整数都在某点出现,因而任何初始"种子"I_0 的选择都与其他选择一样:序列恰好从该点产生。

尽管这种一般的结构已足以能提供相当不错的随机数,但在许多 ANSI C 库中,它的实现方法是相当有缺陷的,其中一定数量的实施方法不能很好地工作。这个责任将由ANSI C 委员会及其实施者来承担。典型的问题如下:

首先,由 ANSI C 标准规定 rand()返回一个整型的数值,这个整数在许多机器中只有两个字节的长度,使 RAND_MAX 通常不是很大。ANSI C 标准要求 RAND_MAX 最多只是 32 767。在许多情况下,这可能是灾难性的。对蒙特卡罗积分(参见 7.6 节和 7.8 节)来说,大概需要对 10^6 个不同点进行计算求值。于是,原来不相同的每一次计算求值,实际上却是对同样的 32 767 个点进行 30 次运算,这不同于我们需要的计算。因此,必须明确地丢弃一些库内只具有两个字节返回值的随机数程序

其次,ANSI C 委员会发表的理论说明中包含了以下一段话:"委员会规定,一种实施方法所提供的 rand 函数,它能在该实施过程中产生优随机系列,则这种实施方法是允许的。因此,没有规定标准的算法。这就是承认在不同的实施方法中,函数能产生相同的伪随机序列,正如已发表的示例……",这个"示例"是

```
unsigned long next=1;

int rand(void) /* NOT RECOMMENDED (see text) */
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
    next=seed;
}
```

相当于(7.1.1)式中取 $a = 1\,103\,515\,245$, $c = 12\,345$, $m = 2^{32}$(因此,算法在无符号长精度数据类型下计算,能保证返回正确的低阶位)。在该例中 a 和 c 虽并非是最好的选择,但总的来说,对它们的选择不是特别复杂。

然而，真正弄巧成拙的是，执行者们把委员会的上述陈述作为依据，试着在发表的示例上进行"改进"。例如，在一台普通的32位PC兼容机上，装上一个与上述相符的长型类的生成程序，但它交换了返回值的高16位和低16位。有人认为这种额外的变换能增加随机性，但实际上这破坏了生成程序。虽然这类的错误可以修复，但在线性同余生成程序中存在一种根本的缺陷。这就是我们现在正要讨论的问题。

线性同余的方法具有速度快、每次调用只需少量运算等优点，因此几乎是通用的。但它的缺点是在连续调用时，不能避免序列的相关性。如果同时使用 k 个随机数在 k 维空间中描点做图(每个坐标都在0和1之间)，这些点不会出现"充满"k 维空间的趋势，相反地将位于 $k-1$ 维"平面"上。这样的平面最多大约有 $m^{1/k}$ 个，并且如果常数 m，a 和 c 没有仔细选择的话，则平面的个数还要少许多。如果数 m 为32 768，那么在三维空间中，三重点位于的平面数将不超过32 768(2^{15})的立方根，即32。即使 m 接近机器所表示的最大整数，例如 $\sim 2^{32}$，二重点所位于的平面数通常不超过 2^{32} 的立方根，约为1600。那么，用户不如将注意力只集中在总体积中一小部分范围内发生的物理过程上，所以平面的离散性非常明显。

更遗憾的是，读者可能正在使用一个生成程序，它选择的 m，a 和 c 往往弄巧成拙。一个很差的程序如RANDU，其 $a=65\,539$ 且 $m=2^{31}$，在IBM大型机上使用了许多年，并被广泛复制在其他系统上[1]。我们的一位同事检测产生了只具有11个平面的一个"随机"图。他所在的计算机中心程序设计顾问告诉他，这是误用了随机数生成程序因此得出这种结论："我们保证每一个数单独来说都是随机的，但是并不保证一个以上的数也是随机的"。一定要认清这一点。

k 维空间的相关性并非是线性同余生成程序的惟一缺陷。这种生成程序的低阶位(最小有效位)常常比它们的高阶位随机性差得多。如果想在1和10之间生成一个随机整数，应该一直利用高阶位，如

```
j=1+int(10.0*rand()/(RAND_MAX+1.0));
```

而不应该用

```
j=1+(rand() % 10);
```

(它使用了低阶位。)同样，决不能将"rand()"数分成几个想像的随机段，而是应该对每段采取不同的调用。

7.1.2 可移植的随机数生成程序

Park和Miller[1]研究出大量的随机数生成程序，这些程序迄今为止已被使用了近30年。经过严格的理论检验，他们指出了一些正在被广泛使用的不适当的程序示例。

这里有一个很好的经过理论与实际检验的论证，即简单的乘法同余算法：

$$I_{j+1}=aI_j \pmod m \tag{7.1.2}$$

如果乘数 a 和模 m 精确选择，它可以和 $c\neq 0$((7.1.1)式)的线性同余生成程序一样好。Park和Miller提出了一个基于以下选择的"最低标准"生成程序：

$$a=7^5=16\,807 \qquad m=2^{31}-1=2\,147\,483\,647 \tag{7.1.3}$$

这个生成程序在1969年首先由Lewis，Goodman和Miller提出，在以后的几年里，此生成程序通过了所有新型理论的检验，并且(也许更重要)积累了大量成功的应用经验。Park和Miller从未自称这一生成程序是"完美"的(在下面可看到它确实并非完美)，但是经鉴定与其他生成程序相比较，这个最低标准生成程序是最好的。

在高级语言中，直接实施(7.1.2)式和(7.1.3)式是不可能的，因为 a 和 $m-1$ 的乘积超过了32位整数的最大值。在汇编语言的执行过程中，使用64位乘积寄存器是很简单的，但机器与机器之间不可移植。当两个32位整数之积以一个32位的常数为模时，不用借助其他大于32位(包括符号位)的辅助寄存器，这个技巧由Schrage[2,3]提出。这个技巧极其有趣：它容许最低标准生成程序在所有机器中，用任何一种基本的程序语言来执行。

Schrage的算法是基于 m 的"近似因子分解"：

$$m=aq+r, \quad 即, \quad q=[m/a],\ r=m \bmod a \tag{7.1.4}$$

方括号表示整数部分。若 r 很小，特别是 $r<q$ 且 $0<z<m-1$，则可以看出 $a(z \bmod q)$ 和 $r[z/q]$ 都处在 $0,\ldots,m-1$ 范围内，且

$$az \bmod m = \begin{cases} a(z \bmod q) - r[z/q] & \text{如果} \geqslant 0, \\ a(z \bmod q) - r[z/q] + m & \text{其他情况} \end{cases} \tag{7.1.5}$$

对(7.1.3)式中的常数,Schrage 的算法取 $q = 127\,773$ 和 $r = 2836$。

下面是一个最低标准生成程序:

```
#include "nr.h"

DP NR::ran0(int &idum)
Park 和 Miller 的"最低"随机数生成程序。返回在 0.0 至 1.0 之间的一致随机偏离。设置 idum 为任意整数(除 MASK 外)
用于初始赋值。在序列中连续调用时,idum 值不允许改变。
{
    const int IA=16807,IM=2147483647,IQ=127773;
    const int IR=2836,MASK=123459876;
    const DP AM=1.0/DP(IM);
    int k;
    DP ans;

    idum ^= MASK;                          对于 idum,与 MASK 异或允许使用零或其他简单
    k=idum/IQ;                             的位模式。
    idum=IA*(idum-k*IQ)-IR*k;              用 Schrage 的方法计算 idum=(IA*idum)% IM,不会
    if (idum < 0) idum += IM;              溢出。
    ans=AM*idum;                           变换 idum 为浮点型结果。
    idum ^= MASK;                          返回前取消屏蔽。
    return ans;
}
```

ran0 的周期为 $2^{31} - 2 \approx 2.1 \times 10^9$。在(7.1.2)形式的特殊生成程序中,0 值是决不允许作为初始种子,它只会维持自身不变,若采用任何非 0 的初始种子,上述情形绝不会出现。然而经验证明,使用者总是设法用种子 idum = 0 来调用随机数生成程序。这就是 ran0 在入口和出口处都用任意常数来执行它的异或的原因。如果用户能指出前人的错误,就可以用 A 操作将这两行清除。

Park 和 Miller 讨论了另外两个乘子 a,它们都可用于同一个 $m = 2^{31} - 1$。它们是 $a = 48\,271$($q = 44\,488$, $r = 3399$)和 $a = 69\,621$($q = 30\,845$, $r = 23\,902$)。如果需要的话,这些值可在 ran0 程序中替换。它们比 Lewis 等所采用的较长的试验值更好。除这些值之外,其他的值都不能用。

ran0 程序是一个最低标准,它的大多数应用是令人满意的。但我们并不把它作为随机数生成程序的最优方法推荐。理由是,这个最低标准非常地简单。不难想像这种情况,在需使用连续的随机数处,它与生成程序的算法可能有抵触。例如,由于连续数与 2×10^9 的模数只相差一个 1.6×10^4 的倍数。因此,非常小的随机数之后趋于跟有一个小于平均值的数。例如,在 10^4 的范围内,一旦有一个小于 10^{-6} 的值返回,在这个值的后面总是跟有一个略小于 0.0168 的数。这就很容易想到,如果把这些参数应用于特殊事件,就会导致错误的结果。

还有更模糊的一系列相关性出现在 ran0 中。例如,如果连续的点(I_i, I_{i+1})属于二维平面,$i = 1, 2, \ldots, N$,那么当 N 稍比 10^7 大一点,且远小于周期 $m - 2$ 时,其结果使 χ^2 检验失败。因为低阶顺序相关性如此难控制,而且现有消除它们的方法又是非常简单,因此我们认为应谨慎地使用它。

下面的 ran1 程序使用了最低标准产生随机值,但它为消除低阶顺序相关性而搅乱了输出。随机偏离要获取序列 I_j 中的第 j 个值,它不是取在第 j 次调用中的输出值,而是取在调用平均 $j + 32$ 次后的随机值。这种混乱的算法由 Bays 和 Durham 提出,在 Knuth_4]中有所描述,如图 7.1.1 所示。

```
#include "nr.h"

DP NR::ran1(int &idum)
Park 和 Miller 的"最低"随机数生成程序,包括 Bays-Durham 混洗算法且加了保护。返回 0.0 ~ 1.0 之间的一致随机偏离
(除终点值外)。以 idum 的负整数作为初值调用,所以在连续偏离的序列中 idum 值不允许改变。RNMX 近似为小于 1 的
最大浮动值。
{
    const int IA=16807,IM=2147483647,IQ=127773,IR=2836,NTAB=32;
    const int NDIV=(1+(IM-1)/NTAB);
    const DP EPS=3.0e-16,AM=1.0/IM,RNMX=(1.0-EPS);
    static int iy=0;
    static Vec_INT iv(NTAB);
```

```
    int j,k;
    DP temp;

    if (idum <= 0 || !iy) {                 初始赋值。
        if (-idum < 1) idum=1;              确保防止 idum=0。
        else idum = -idum;
        for (j=NTAB+7;j>=0;j--) {           装载混洗表(8以后逆增)。
            k=idum/IQ;
            idum=IA*(idum-k*IQ)-IR*k;
            if (idum < 0) idum += IM;
            if (j < NTAB) iv[j] = idum;
        }
        iy=iv[0];
    }
    k=idum/IQ;                              非初始值启动。
    idum=IA*(idum-k*IQ)-IR*k;               用 Schrage 的方法计算 idum=(IA*idum) % IM
    if (idum < 0) idum += IM;               不会溢出。
    j=iy/NDIV;                              在 0-NTAB-1 范围内。
    iy=iv[j];                               预先保存值后输出并填充混洗表。
    iv[j] = idum;
    if ((temp=AM*iy) > RNMX) return RNMX;   使用者不要求终点值。
    else return temp;
}
```

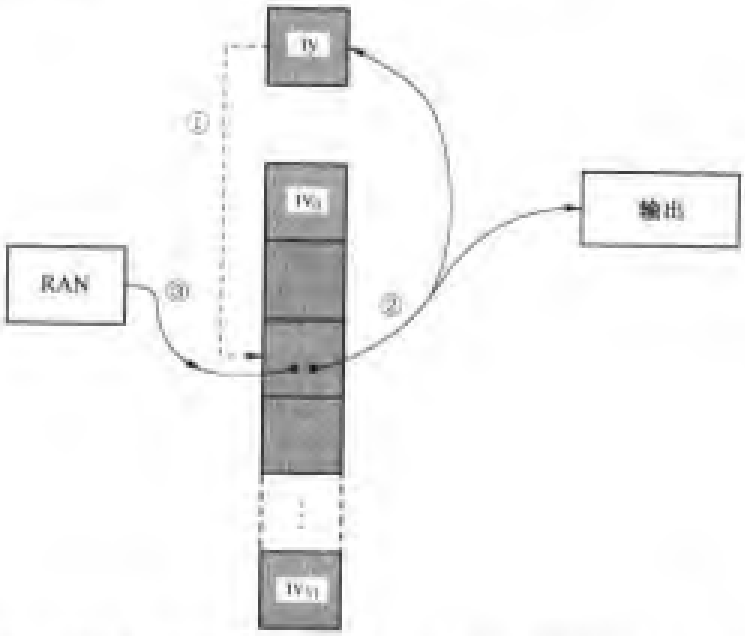


图 7.1.1 用于 ran1的混洗过程。它破坏了在最低标准随机数生成程序中的顺序相关性。图中加圆圈的数表示事件的顺序：在每次调用中，iy中的随机数是用来选择数组iv中的随机元素。该元素变成输出随机数，并且就是下一个iy，它在iv中的位置由最低标准程序重新填补

程序 ran1 通过了 ran0 没能通过的统计测试。事实上，我们并不知道 ran1 未能通过的任何统计测试，除了起始调用的次数变成周期 m 的阶，例如大于 $10^9 \approx m/20$。

对于某些需要更长的随机序列的场合，L'Ecuyer [6]给出了可将两个不同周期的序列组合形成一个新序列的方法，该序列的周期是这两个周期的最小公倍数。基本原理是把两序列简单地相加，它们二者的模之一作为模，称它为 m。为避免中间值溢出整数值范围，一个好方法是相减而不是相加，然后再把常数 $m-1$ 加回去(如果结果≤ 0)，这样就把数限制在所期望的区间 $0,\ldots,m-1$ 范围内。

值得注意的是，这种受约束的减法可以从第一序列的每一个值得到所有的值 $0,\ldots,m-1$，但这并不是必需的，考虑到不合理的极端情况，被减值只在 1 和 10 之间；所得的序列在随机性方面不亚于它自己的第一序列。实际上，第二个序列的值域基本上覆盖了第一个序列的值域，这是必不可少的。L'Ecuyer 推荐使用两个生成程序 $m_1 = 2\,147\,483\,563$($a_1 = 40\,014$，$q_1 = 53\,668$，$r_1 = 12\,211$)和 $m_2 = 2\,147\,483\,399$($a_2 = 40\,692$，$q_2 = 52\,774$，$r_2 = 3791$)。两个模数都略小于 2^{31}。周期 $m_1 - 1 = 2\times 3\times 7\times 631\times 81\,031$，$m_2 - 1 = 2\times 19\times 31\times 1019\times 1789$，只有一

个公因子2,所以组合后的生成程序周期约为2.3×10^{18}。对于目前的计算机来说,超出这个周期是不可能的。

两个生成程序的组合,在很大程度上破坏了顺序相关性。尽管如此,我们(在下面程序ran2中)将介绍相加混洗的执行过程。我们认为在它的浮点精度范围内,ran2提供了很好的随机数。

```
#include "nr.h"

DP NR::ran2(int &idum)
L'Ecuyer的长周期(>2×10^18)随机数生成程序包括Bays-Durham混洗算法而且添加了保护。返回0.0~1.0之间的一致
随机偏离(除终点值外)。以idum的负整数作为初值调用,所以在连续偏离的序列中idum值不允许改变,RNMX近似为
小于1的最大浮动值。
{
    const int IM1=2147483563,IM2=2147483399;
    const int IA1=40014,IA2=40692,IQ1=53668,IQ2=52774;
    const int IR1=12211,IR2=3791,NTAB=32,IMM1=IM1-1;
    const int NDIV=1+IMM1/NTAB;
    const DP EPS=3.0e-16,RNMX=1.0-EPS,AM=1.0/DP(IM1);
    static int idum2=123456789,iy=0;
    static Vec_INT iv(NTAB);
    int j,k;
    DP temp;

    if (idum <= 0) {                          初始赋值。
        idum=(idum==0 ? 1 : -idum);           确保防止idum=0。
        idum2=idum;
        for (j=NTAB+7;j>=0;j--) {             装载混洗表(8以后递增)。
            k=idum/IQ1;
            idum=IA1*(idum-k*IQ1)-k*IR1;
            if (idum < 0) idum += IM1;
            if (j < NTAB) iv[j] = idum;
        }
        iy=iv[0];
    }
    k=idum/IQ1;                               非初始值启动。
    idum=IA1*(idum-k*IQ1)-k*IR1;              用Schrage的方法计算idum=(IA1*idum)%
    if (idum < 0) idum += IM1;                IM1不会溢出。
    k=idum2/IQ2;
    idum2=IA2*(idum2-k*IQ2)-k*IR2;            计算idum2=(IA2*idum)%IM2不会溢出。
    if (idum2 < 0) idum2 += IM2;
    j=iy/NDIV;                                在0~NTAB-1范围内。
    iy=iv[j]-idum2;                           这里idum已被混洗,idum和idum2组合而
    iv[j] = idum;                             产生输出。
    if (iy < 1) iy += IMM1;
    if ((temp=AM*iy) > RNMX) return RNMX;     使用者不要求终点值。
    else return temp;
}
```

L'Ecuyer[6]列出了一些附加的小生成程序,这些生成程序可以合并为一个大程序,包括可以执行16位整数计算。

最后,把Knuth建议[4]的可移植程序提供给大家,已将它转换成为现在的程序,称为ran3。这个ran3根本不是基于线性同余的方法,而是以一个相减方法为基础(参见本节参考文献[5])。人们也许希望,如果这个程序有缺点,它应与前面的ran1的缺点具有截然不同的特征。如果用户总是对一个程序产生怀疑,那么一个好的办法是,在同一个应用程序中试用一下其他程序。ran3具有一个不错的功能:如果所用的机器在整数算术方面(限于16位的整数)很差,可以说明将mj,mk和ma[]设为浮点数,并且定义mbig和mseed分别为4 000 000和1 618 033,于是相应程序将全部变成浮点型程序。

```
#include <cstdlib>
#include "nr.h"
using namespace std;

DP NR::ran3(int &idum)
返回0.0~1.0之间的一致随机偏离。设定idum为任意负值,作为序列的初始赋值或重新初始赋值。
{
    static int inext,inextp;
    static int iff=0;
    const int MBIG=1000000000,MSEED=161803398,MZ=0;
    const DP FAC=(1.0/MBIG);
According to Knuth, any large MBIG, and any smaller (but still large) MSEED can be sub-
stituted for the above values.
```

```
    static Vec_INT ma(56);              值56(其范围ma[1..55])为一特殊值，故不应改动;
    int i,ii,k,mj,mk;                   参见Knuth。

    if (idum < 0 || iff == 0) {         初始赋值。
        iff=1;
        mj=labs(MSEED-labs(idum));      使用种子idum和大数MSEED对ma[55]初始
        mj %= MBIG;                     赋值。
        ma[55]=mj;
        mk=1;
        for (i=1;i<=54;i++) {           现在按稍微随机的顺序，并用不是特别随机的
            ii=(21*i) % 55;             数，对表的其余部分初始赋值。
            ma[ii]=mk;
            mk=mj-mk;
            if (mk < int(MZ)) mk += MBIG;
            mj=ma[ii];
        }
        for (k=0;k<4;k++)               我们用“提高生成程序的办法”来随机化它们。
            for (i=1;i<=55;i++) {
                ma[i] -= ma[1+(i+30) % 55];
                if (ma[i] < int(MZ)) ma[i] += MBIG;
            }
        inext=0;                        准备第一个生成数指标。
        inextp=31;                      常数31为特殊数，参见Knuth。
        idum=1;
    }
    除初始化外，在此处开始。
    if (++inext == 56) inext=1;         增量inext和inextp在56至1间循环。
    if (++inextp == 56) inextp=1;
    mj=ma[inext]-ma[inextp];            相减而生成新随机数。
    if (mj < int(MZ)) mj += MBIG;       确保它在范围之内，并输出得到一致随机偏离。
    ma[inext]=mj;
    return mj*FAC;
}
```

7.1.3 深入讨论:快速而略有缺陷的生成程序

人们有时需要将快速而略有缺陷的随机数生成程序嵌套在程序之中，也许只取一行或二行代码，仅稍微进行一些随机化处理。例如，人们可能希望处理从某种不总是以确定顺序进行的实验中得到的数据，所以要求第一个输出要比可能出现的其他情形更“典型”。

对于这种应用，实际上真正需要的就是“合理”地在(7.1.1)式中选择一组 m，a 和 c。如果并不需要一个比 10^4 到 10^6 更长的周期，则可以维持 $(m-1)a+c$ 的值足够小，使其小到足以在 Schrage 的复杂方法中能避免溢出。每当需要一个快速而略有缺陷的一致偏离时，可以很容易地将

```
unsigned long jran,ia,ic,im;
DP ran;
...
jran=(jran*ia+ic) % im;
ran=DP(jran)/DP(im);
```

嵌入程序中。或想得到在 jlo 和 jhi(包括两个端点在内)之间的一个整数时，则可以将

```
jran=(jran*ia+ic) % im;
j=jlo+((jhi-jlo+1)*jran)/im;
```

嵌入程序中(在以上两种情况中，jran 曾取 0 和 im − 1 之间的任何种子值进行初始化)。

应当确认，当 im 很小时，它的第 k 个根，即为 k 维空间平面的个数，甚至会更小。所以一个快速却略有缺陷的程序将绝对不能用来选择 $k>1$ 的 k 维空间的点。

有了以上的说明，对于“好”的常数的选择由下面的表 7.1.1 给出。这些常数：(i)给出最大长度 im 的周期，并且更重要的是，(ii)对 2 维、3 维、4 维、5 维、6 维的情形通过了 Knuth 的“谱检验”。增量 ic 是一个接近 $\left(\frac{1}{2}-\frac{1}{6}\sqrt{3}\right)$ im 值的素数。实际上几乎与 im 互素的 ic 值都会通过这一检验，不过有些“经验知识”有利于这种选择(参见本节参考文献[4]中第 84 页)。

表 7.1.1 快速而略有缺陷的生成程序的常数

上溢出	im	ia	ic	上溢出	im	ia	ic
	6075	106	1283		86 436	1093	18 257
2^{20}					121 500	1021	25 673
	7875	211	1663		259 200	421	54 773
2^{21}				2^{27}			
	7875	421	1663		117 128	1277	24 749
2^{22}					121 500	2041	25 673
	6075	1366	1283		312 500	741	66 037
	6655	936	1399	2^{28}			
	11 979	430	2531		145 800	3661	30 809
2^{23}					175 000	2661	36 979
	14 406	967	3041		233 280	1861	49 297
	29 282	419	6173		244 944	1579	51 749
	53 125	171	11 213	2^{29}			
2^{24}					139 968	3877	29 573
	12 960	1741	2731		214 326	3613	45 289
	14 000	1541	2957		714 025	1366	150 889
	21 870	1291	4621	2^{30}			
	31 104	625	6571		134 456	8121	28 411
	139 968	205	29 573		259 200	7141	54 773
2^{25}				2^{31}			
	29 282	1255	6173		233 280	9301	49 297
	81 000	421	17 117		714 025	4096	150 889
	134 456	281	28 411	2^{32}			
2^{26}							

7.1.4 深入讨论:更快的生成程序

在 C++ 语言中,若在 32 位长整型表示的机器中,要将两个无符号长整型整数相乘,返回值只是 64 位乘积中的低 32 位。如果现在选择 $m = 2^{32}$,则(7.1.1)式中的这个"模"是任意的,而且简单地有

$$I_{j+1} = aI_j + c \tag{7.1.6}$$

Knuth 建议 $a = 1\,664\,525$,作为 m 的一个适当的乘子,H.W.Lewis 用 $c = 1\,013\,904\,223$ 对 a 值进行了大量测试,它是一个接近$(\sqrt{5}-2)m$ 值的素数。这样产生的生成程序(称它为 ranqd1)是

```
unsigned long idum;
...
idum = 1664525L*idum + 1013904223L;
```

这个程序中任何 32 位线性同余程序一样有效,完全满足许多应用的需要,而且只有一个单独的乘法和加法,速度非常快。

为了检查机器是否具有所要求的整数性能,可查看它是否能产生下列 32 位值的序列(这里是以十六进制给出的):00000000, 3C6EF35F, 47502932, D1CCF6E9, AAF95334, 6252E503, 9F2EC686, 57FE6C2D, A3D95FA8, 81FDBEE7, 94F0AF1A, CBF633B1。

如果需要的是浮点值而不是 32 位整数,而且要避开被浮点数除 2^{32},一个不明显的方法是,把这个值掩蔽在某一个幂指数中,使其处于 1 和 2 之间,然后减 1.0。这样产生的生成程序(称其为 ranqd2)看起来如下:

```
    unsigned long idum,itemp;
    float rand;
#ifdef vax
    static unsigned long jflone = 0x00004080;
    static unsigned long jflmsk = 0xffff007f;
#else
    static unsigned long jflone = 0x3f800000;
    static unsigned long jflmsk = 0x007fffff;
#endif
    ...
    idum = 1664525L*idum + 1013904223L;
    itemp = jflone | (jflmsk & idum);
    rand = (*(float *)&itemp)-1.0;
```

十六进制常数 3F800000 和 007FFFFF 适合于用 IEEE 表示的计算机的 32 位浮点数(如 Pentium 机和大多数 Linux 或 UNIX 工作站)。对 DEC VAX 机,正确的十六进制常数分别为 00004080 和 FFFF007F。提醒注意,IEEE 的掩蔽法会引起构造的浮点数超出整型数的 23 位低阶位,这不是理想的(作者已尽最大努力,力求使本书的全部内容独立于机器及编译器,甚至与程序语言无关。本小节可能略有偏离,请谅解在很多时候,会出现不可抗拒的存在缺陷的结果)。

7.1.5 相对的执行时间和建议

执行程序的时间必然取决于机器。然而表 7.1.2 表现出了典型机器运行本节讨论的多种一致随机偏离生成程序的相对执行时间,包括 7.5 节中的 ran4。在表 7.1.2 中,较小的数值表明程序执行速度较快。程序 ranqd1 和 ranqd2 可参考上面介绍的快速而略有缺陷的生成程序。

表 7.1.2 典型机器的相对执行时间

生成程序	相对执行时间
ran0	≈1.0
ran1	≈1.3
ran2	≈2.0
ran3	≈0.6
ranqd1	≈0.10
ranqd2	≈0.25
ran4	≈4.0

总的来说,建议一般情况使用 ran1。基于 Park 和 Miller 的附加混洗的最低标准生成程序是可移植的,并且就我们所知,它除了周期逸出之外没有其他缺陷。

如果在单个计算中要产生大于 100 000 000 的随机数(即大于 ran1 周期的 5%),则建议用 ran2 程序,它的周期要长得多。

看上去 Knuth 的减法程序 ran3 就执行时间而言在可移植程序中是佼佼者,遗憾的是,这种方法不易掌握,而且不标准。一般把 ran3 作为"第二位的建议",当怀疑某个生成程序在相关计算上有缺点时,可用它来代替。

ran4 可生成非常好的随机序列,而且还具有其他一些优良性能,但它速度慢,参阅 7.5 节。

最后,生成程序 ranqd1 和 ranqd2 的速度是非常快的,但它们依赖机器,而且最多与 32 位线性同余生成程序一样好。但以我们的观点,它们在很多场合下并不能令人满意。建议只在速度受限制的特殊情况下才使用它们。

参考文献和深入阅读

Park, S.K., and Miller, K.W. 1988, *Communications of the ACM*, vol. 31, pp. 1192–1201. [1]

Schrage, L. 1979, *ACM Transactions on Mathematical Software*, vol. 5, pp. 132–138. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation*, 2nd ed. (New York: Springer-Verlag). [3]

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §§3.2–3.3. [4]

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 10. [5]

L'Ecuyer, P. 1988, *Communications of the ACM*, vol. 31, pp. 742–774. [6]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

7.2 变换方法:指数偏离和正态偏离

前一节中学习了如何用一致概率分布生成随机偏离的方法,所以在 x 和 $x+dx$ 之间生成一个数的概率,记成 $p(x)dx$,可由下式给出:

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{其他情况} \end{cases} \tag{7.2.1}$$

概率分布 $p(x)$ 当然是归一化的,所以

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{7.2.2}$$

现在,假设生成一个一致偏离 x,然后取 x 的某个指定的函数 $y(x)$。y 的概率分布,记成 $p(y)dy$,由概率的基本变换定律确定。简单地表示为

$$|p(y)dy| = |p(x)dx| \tag{7.2.3}$$

或者

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \tag{7.2.4}$$

7.2.1 指数偏离

例如,假设 $y(x) \equiv -\ln(x)$,并且 $p(x)$ 由一致偏离的(7.2.1)式给出。于是

$$p(y)dy = \left|\frac{dx}{dy}\right| dy = e^{-y}dy \tag{7.2.5}$$

呈指数分布。在实际问题中,这种指数分布经常出现。通常,独立的泊松(Poisson)随机事件之间的等待时间的分布就是这种分布,例如,核放射性衰变。还可以很容易地从(7.2.4)式看出,量 y/λ 具有的概率分布为 $\lambda e^{-\lambda y}$。

所以有

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::expdev(int &idum)
利用 ran1(idum)作为一致偏离的来源,返回一个具有单位均值、指数分布的正规随机偏离。
{
    DP dum;

    do
        dum=ran1(idum);
    while (dum == 0.0);
    return -log(dum);
}
```

在使用上面的**变换方法**生成 y 的任意某个所期望的分布过程中,例如对某个积分等于 1 的正函数 f,生成具有 $p(y)=f(y)$ 的分布的过程,让我们看看它包括一些什么内容(参见图 7.2.1)。根据(7.2.4)式,需要求解微分方程

$$\frac{dx}{dy} = f(y) \tag{7.2.6}$$

这个方程的解正好是 $x = F(y)$，其中 $F(y)$ 是 $f(y)$ 的不定积分。所以，取 $f(y)$ 分布为一致偏离，它所期望的变换为

$$y(x) = F^{-1}(x) \tag{7.2.7}$$

其中 F^{-1} 是 F 的反函数。(7.2.7)式是否可行取决于 **$f(y)$ 积分的反函数**本身计算起来是否可行，无论从解析方面还是从数值方面都是如此。计算起来有时候是可行的，有时则不可行。

下面给(7.2.7)式一个直观的几何解释：因为 $F(y)$ 是 y 的概率曲线下从0至 y 的面积，(7.2.7)式就可这样规定：选取一个一致随机的 x，然后寻找 y 值，使从0到 y 的概率面积等于 x，此时 y 为所求。

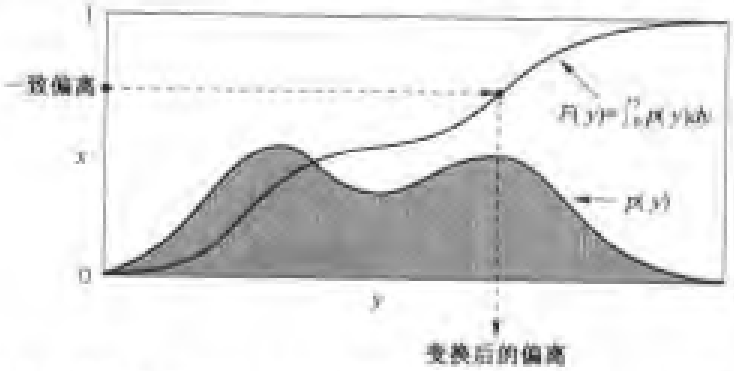


图7.2.1 由已知概率分布 $p(y)$ 生成随机偏离 y 的变换方法。$p(y)$ 的不定积分必须是已知的并且是可逆的。在0和1之间选择一致偏离 x。它在给定的积分曲线上所对应的 y 就是所求的偏离

7.2.2 正态(高斯)偏离

变换方法可以推广到高于一维的情形。如果 $x_1, x_2, \ldots$ 是具有**联合概率分布** $p(x_1 x_2, \ldots) dx_1 dx_2, \ldots$ 的随机偏离，并且如果 $y_1, y_2, \ldots$ 每个都是所有 x 的函数(y 的个数与 x 的个数相同)，则 y 的联合概率分布为

$$p(y_1, y_2, \ldots) dy_1 dy_2 \ldots = p(x_1, x_2, \ldots) \left| \frac{\partial(x_1, x_2, \ldots)}{\partial(y_1, y_2, \ldots)} \right| dy_1 dy_2 \ldots \tag{7.2.8}$$

其中 $|\partial()/\partial()|$ 为 x 关于 y 的雅可比行列式(或者是 y 关于 x 的雅可比行列式的倒数)。

运用(7.2.8)式的一个重要例子是，生成具有正态(高斯)分布的随机偏离的 Box-Muller 方法，

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{7.2.9}$$

考虑(0,1)内两个一致偏离 x_1, x_2 与两个量 y_1, y_2 之间的变换，

$$\begin{aligned} y_1 &= \sqrt{-2\ln x_1} \cos 2\pi x_2 \\ y_2 &= \sqrt{-2\ln x_1} \sin 2\pi x_2 \end{aligned} \tag{7.2.10}$$

等价地，可以写成

$$\begin{aligned} x_1 &= \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right] \\ x_2 &= \frac{1}{2\pi} \arctan \frac{y_2}{y_1} \end{aligned} \tag{7.2.11}$$

现在的雅可比行列式可以很容易地计算：

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}} e^{-y_1^2/2}\right] \left[\frac{1}{\sqrt{2\pi}} e^{-y_2^2/2}\right] \tag{7.2.12}$$

因为此式是单个 y_2 函数与单个 y_1 函数的乘积，并且可以看出，每个 y 都是服从正态分布式(7.2.9)的独立分布。

在应用(7.2.10)式时,还有一个很有用的技巧。假设不在单位正方形中挑选一致偏离 x_1 和 x_2,而代之在原点为圆心的单位圆内挑选一个随机点 v_1 和 v_2 作为它的纵坐标和横坐标。于是,它们的平方和 $R^2 \equiv v_1^2 + v_2^2$ 是一个一致偏离,它可以用于 x_1。而由 (v_1, v_2) 定义的,关于 v_1 的夹角可以起到随机角度 $2\pi x_2$ 的作用。正是因为这点,现在(7.2.10)式中,余弦和正弦可以写成 $v_1/\sqrt{R^2}$ 和 $v_2/\sqrt{R^2}$,避免了对三角函数的调用。

于是有

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::gasdev(int &idum)
使用 ran1(idum)作为一致偏离的来源,返回一个具有零均值和单位方差的正态分布偏离。
{
    static int iset=0;
    static DP gset;
    DP fac,rsq,v1,v2;

    if (idum < 0) iset=0;                    重新初始化。
    if (iset == 0) {                         没有额外的偏离,所以……
        do {
            v1=2.0*ran1(idum)-1.0;           ……由每个方向-1到+1扩充而成的正方形
            v2=2.0*ran1(idum)-1.0;           中选出两个一致数。
            rsq=v1*v1+v2*v2;                 检查它们是否在单位圆内,
        } while (rsq >= 1.0 || rsq == 0.0); 如果不在单位圆内,再试一次。
        fac=sqrt(-2.0*log(rsq)/rsq);
        现在进行 Box-Muller变换,以得到两个正态偏离。返回一个并为下次变换保存一个。
        gset=v1*fac;
        iset=1;                              设置标志。
        return v2*fac;
    } else {                                 有额外的偏离,
        iset=0;                              故清除标记,
        return gset;                         并返回。
    }
}
```

参考文献和深入阅读

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), §9.1. [1]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation*, 2nd ed. (New York: Springer-Verlag). [2]

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 121ff.

7.3 拒绝方法:伽马偏离、泊松偏离、二项偏离

拒绝方法是生成随机偏离的强有力的一般技巧,这些随机偏离的分布函数 $p(x)\,dx$(在 x 和 $x+dx$ 之间数值发生的概率)是已知并可计算的。拒绝方法并不要求累积分布函数($p(x)$的不定积分)是容易计算的,更不要求累积分布函数的反函数易计算,这是前一节的变换方法所要求的。

拒绝方法是以简单的几何依据为基础的:

描绘一个想生成的概率分布 $p(x)$的图形,使得在任意 x 范围内曲线下的面积,对应于在该范围内产生一个 x 所期望的概率。如果有某种方法选择一个二维情形的随机点,使其在曲线下的面积具有一致概率,则该随机点 x 值便是所期望的分布。

在这一图上画任一条曲线 $f(x)$,使之具有有限(并非无限)的面积,并且每一处都位于原来概率分布之上(由概率的定义这始终是可能的,因为原来的曲线只围住单位面积)。将这个 $f(x)$称为**比较函数**。设想一下,若有在二维情形中选择一个随机点的方式,则选择的随机点在比较函数下面积是均匀的。每当该点位于原概率分布的面积以外时,就**拒绝**它并选另一个随机点。每当该点位于原概率分布面积之内时,就**接受**它。

很显然，接受点在被接受面积中是一致的，所以，它们的 x 值都有所期望的分布。同样易见，被拒绝的点部分取决于比较函数的面积与概率分布函数的面积之比，而不取决于这两个函数之中任一个的具体形状。例如，面积小于2的比较函数将拒绝不超过一半的点。即使在某些 x 值处，它非常差地逼近概率函数，如在某个 $p(x)$ 为零的区域中，被拒绝的点仍保持有限。

下面讨论如何在比较函数 $f(x)$ 下选择二维的一致随机点的方法。变换方法(参见7.2节)的变形能很好地解决这一问题：应当确信已选择了一个比较函数，它的不定积分是解析可知的，并且还是解析可逆的，以给出 x 作为"在比较函数面积下直到 x 左边"的函数。现在选出位于0与 A 之间的一个一致偏离，其中 A 是 $f(x)$ 下总面积，并且用这个一致偏离得到一个对应的 x。然后，选择0与 $f(x)$ 之间的一个一致偏离作为二维点的 y 值。应该确信，点 (x,y) 在比较函数 $f(x)$ 下为均匀分布。

一个等价的过程就是，在0和1之间找出第二个一致偏离值，并根据它是否小于或大于比率 $p(x)/f(x)$ 来决定接受或拒绝它。

综上所述，对某个给定的 $p(x)$，拒绝方法要求断然地找到某个合理的比较函数 $f(x)$。此后每生成一个偏离值，要求有两个一致随机偏离值，一个是 f 值的计算(以得到坐标 y)，一个是 p 值的计算(以决定是否接受或拒绝点 x,y)。图7.3.1表明了这一过程。当然这个过程平均重复 A 次，然后才获得最终的偏离。

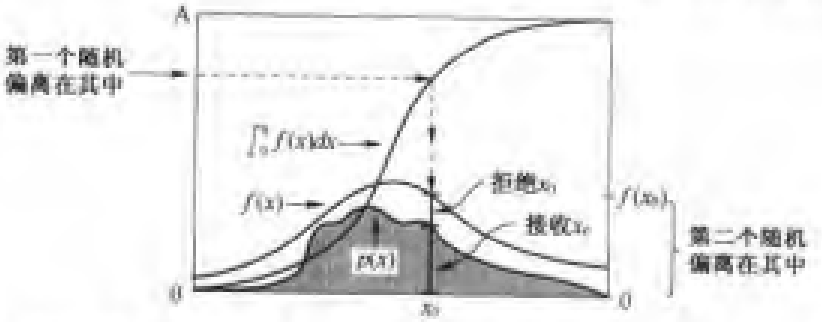


图7.3.1 由已知概率分布 $p(x)$ 生成随机偏离 x 的拒绝方法。其中 $p(x)$ 处处小于某个其他的函数 $f(x)$。变换方法首先用于生成分布 $f(x)$ 的随机偏离 x(比较图7.2.1)。第二个一致偏离用来决定是否接受或拒绝该 x。如果它被拒绝，就寻找 f 的一个新偏离，依次类推。接受点与拒绝点的比率是 p 下的面积与 p 和 f 之间面积的比

7.3.1 伽马分布

整数阶($a>0$)的伽马分布是，具有单位均值的泊松随机过程中第 a 个事件的等待时间。例如，当 $a=1$ 时，它就是7.2节的指数分布，即第一个事件的等待时间。

一个伽马偏离具有在 x 和 $x+dx$ 之间出现值的概率 $p_a(x)dx$，其中

$$p_a(x)dx = \frac{x^{a-1}e^{-x}}{\Gamma(a)}dx \qquad x>0 \tag{7.3.1}$$

为了对小的 a 值生成(7.3.1)式的偏离，最好加上 a 个指数分布等待时间，即一致偏离的对数。因为对数之和就是乘积的对数，实际上只需生成 a 个一致偏离的乘积，然后取对数。

对大的 a 值，分布(7.3.1)式具有典型的"钟型"形式，在 $x=a$ 处有峰值，并且半宽度大约为 $\sqrt{a}$。

我们对具有同样性质形式的几个概率分布很感兴趣。对于这种情况的一个有用的比较函数由 **Lorent** 分布推出：

$$p(y)dy = \frac{1}{\pi}\left(\frac{1}{1+y^2}\right)dy \tag{7.3.2}$$

它的不定积分的反函数恰好是正切函数。由此可得，对任何常数 a_0，c_0 和 x_0，在比较函数

$$f(x) = \frac{c_0}{1+(x-x_0)^2/a_0^2} \tag{7.3.3}$$

面积下，一致随机点 x 的坐标可由以下公式确定：

$$x = a_0 \tan(\pi U) + x_0 \tag{7.3.4}$$

其中 U 是 0 和 1 之间的一致偏离。因此,对某个特殊"钟型"$p(x)$概率分布,只需找出常数 a_0, c_0 和 x_0。其中乘积 $a_0 c_0$(它决定面积)应尽可能小,使得(7.3.3)式在任意点大于 $p(x)$。

Ahrens 已解决了伽马分布的这一问题,得出了如下算法(如 Knuth 书中描述(参见本节参考文献[1])):

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::gamdev(const int ia, int &idum)
使用 ran1(idum)作为一致偏离的来源,返回一个整数阶 ia 的伽马分布的偏离,即在具有单位均值的泊松分布过程中,对
第 ia 个事件的等待时间。
{
    int j;
    DP am,e,s,v1,v2,x,y;

    if (ia < 1) nrerror("Error in routine gamdev");
    if (ia < 6) {                                   使用直接方法,加上等待时间。
        x=1.0;
        for (j=1;j<=ia;j++) x *= ran1(idum);
        x = -log(x);
    } else {                                        使用拒绝方法。
        do {
            do {
                do {                                这四行生成随机角的正切,即等价于
                    v1=ran1(idum);                  y=tan(π*ran1(idum))。
                    v2=2.0*ran1(idum)-1.0;
                } while (v1*v1+v2*v2 > 1.0);
                y=v2/v1;
                am=ia-1;
                s=sqrt(2.0*am+1.0);                 决定是否拒绝 x:
                x=s*y+am;                           在零概率区域内拒绝。
            } while (x <= 0.0);                     概率 fn与比较 fn的比率。
            e=(1.0+y*y)*exp(am*log(x/am)-s*y);      在第二个一致偏离基础上的拒绝。
        } while (ran1(idum) > e);
    }
    return x;
}
```

7.3.2 泊松偏离

从概念上讲,泊松(Poisson)分布与伽马分布有关。泊松分布给出了在时间 x 的给定区间内,某整数 m 个单位速率的泊松随机事件出现的概率,而伽马分布是在 x 和 $x+dx$ 之间对第 m 个事件等待时间的概率。注意 m 只取整数值 ≥ 0,所以将泊松分布看做一个连续分布函数 $p_x(m)dm$,除了 m 取 ≥ 0 的整数之外,泊松分布处处为零。在这样的区域里,它是无穷的,而使得在包含整数的区域上的积分概率是某些有限数。在整数 j 上的总概率为

$$\mathrm{Prob}(j) = \int_{j-\epsilon}^{j+\epsilon} p_x(m)dm = \frac{x^j e^{-x}}{j!} \tag{7.3.5}$$

初看上去,它未必是拒绝方法的候选分布,因为没有任何连续比较函数能够大于这个无穷高且又无限窄的 $p_x(m)$中的**狄拉克(Dirac δ)函数**。不过有这样一个技巧:将 j 处尖峰信号的有限面积均匀地扩展到 j 和 $j+1$ 之间的区间中。这就定义了一个连续分布 $q_x(m)dm$,表示为

$$q_x(m)dm = \frac{x^{[m]}e^{-x}}{[m]!}dm \tag{7.3.6}$$

其中$[m]$表示小于 m 的最大整数。如果现在使用拒绝方法由(7.3.6)式生成一个(非整数)偏离,然后取该偏离的整数部分,它好像是由所求的分布式(7.3.5)得到的(参见图 7.3.2)。这一技巧对任何整数概率分布都是通用的。

对于足够大的 x,分布式(7.3.6)可定性地看成是"钟型"的,这样就可以采用前面已使用过的同样类型的 Lorent 变换。对小的 x,可以生成独立的指数偏离(事件之间的等待时间)。若这些偏离的和首次超出 x,那

么，在等待时间 x 内可能发生事件的个数就成为已知，并且它等于此和式的项数减 1。

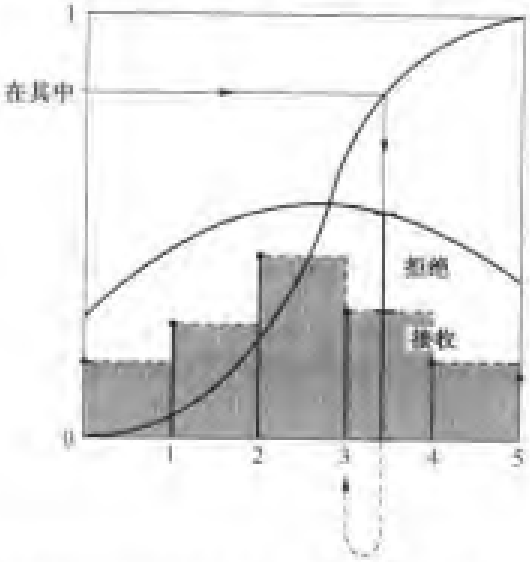


图 7.3.2 作为应用到整数值分布的拒绝方法。方法是在表示成直线的阶梯函数上执行，产生一个实值偏离，这个偏离被截断成下一个较低的整数，这就是输出

用这些想法可写出如下程序：

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::poidev(const DP xm, int &idum)
用 ran1(idum)作为一致随机偏离的来源，按浮点数返回整数值，它是从具有均值 xm 的泊松分布导出的随机偏离。
{
    const DP PI=3.141592653589793238;
    static DP sq,alxm,g,oldm=(-1.0);    oldm 是自上一次调用后，xm 是否有所改变
    DP em,t,y;                          的标志。

    if (xm < 12.0) {                    使用直接方法。
        if (xm != oldm) {
            oldm=xm;
            g=exp(-xm);                 如果 xm 是新的，计算指数。
        }
        em = -1;
        t=1.0;
        do {                            替换叠加指数偏离，它等价于乘上一致偏离。
            ++em;
            t *= ran1(idum);            实际上并不需要取对数，只比较一下事先计算出
        } while (t > g);                的指数。
    } else {                            使用拒绝方法。
        if (xm != oldm) {               如果自下次调用后，xm 有所改变，
            oldm=xm;                    则预先计算下面出现的函数。
            sq=sqrt(2.0*xm);
            alxm=log(xm);
            g=xm*alxm-gammln(xm+1.0);
            函数 gammln 是如 6.1 节中所给出的 Γ 函数的自然对数。
        }
        do {
            do {                        y 是来自 Lorentz 比较函数的偏离。
                y=tan(PI*ran1(idum));
                    em=sq*y+xm;             em 是进行了位移并改变了尺度的 y。
                } while (em < 0.0);         若在零概率区域中则拒绝。
                    em=floor(em);           整值分布的技巧。
                    t=0.9*(1.0+y*y)*exp(em*alxm-gammln(em+1.0)-g);
                    所求的概率与比较函数的比，比较它和另一个一致偏离来决定是接受还是拒
                    绝。因子 0.9 的选择是使 t 不超出 1。
```

```
        } while (ran1(idum) > t);
    }
    return em;
}
```

7.3.3 二项偏离

如果一个事件出现的概率为 q，并且我们进行 n 次试验，那么它出现 m 次的个数具有二项分布：

$$\int_{j-\epsilon}^{j+\epsilon} p_{n,q}(m)dm = \binom{n}{j} q^j (1-q)^{n-j} \tag{7.3.7}$$

二项分布是整数值的，在 0 到 n 的可能值中选取 m 次。它取决于两个参数 n 和 q，所以它的实现相对于前面的情况更难一些。不过，已阐述的技巧对完成这样的工作是不成问题的。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::bnldev(const DP pp, const int n, int &idum)
用 ran1(idum)作为一致偏离的来源，按浮点数返回一整数值，该值是一个由 n 个随机试验(每个试验的概率为 pp)的二
项式分布导出的随机偏离。
{
    const DP PI=3.141592653589793238;
    int j;
    static int nold=(-1);
    DP am,em,g,angle,p,bnl,sq,t,y;
    static DP pold=(-1.0),pc,plog,pclog,en,oldg;

    p=(pp <= 0.5 ? pp : 1.0-pp);
    如果将答案也变为 n 减去它本身，则二项分布在 pp 到 1 - pp 的替换之下是不变的，记住下面将这样做。
    am=n*p;                         这是需要产生偏置的均值。
    if (n < 25) {                   当 n 不是太大时，使用直接方法，这可能需要对
        bnl=0.0;                    ran1 至多 25 次调用。
        for (j=0;j<n;j++)
            if (ran1(idum) < p) ++bnl;
    } else if (am < 1.0) {          如果在 25 次或更多次试验中，期望少于一次的事
        g=exp(-am);                 件发生，则这个分布就是相当准确的泊松分布，使
        t=1.0;                      用直接泊松方法。
        for (j=0;j<=n;j++) {
            t *= ran1(idum);
            if (t < g) break;
        }
        bnl=(j <= n ? j : n);
    } else {                        使用拒绝方法。
        if (n != nold) {            如果 n 改变，则计算有用的量。
            en=n;
            oldg=gammln(en+1.0);
            nold=n;
        } if (p != pold) {          如果 p 改变，则计算有用的量。
            pc=1.0-p;
            plog=log(p);
            pclog=log(pc);
            pold=p;
        }
        sq=sqrt(2.0*am*pc);         下面的代码现在看来应该很熟悉，这是具有
        do {                        Lorentz 比较函数的拒绝方法。
            do {
                angle=PI*ran1(idum);
                y=tan(angle);
                em=sq*y+am;
            } while (em < 0.0 || em >= (en+1.0));      拒绝。
            em=floor(em);           整数值分布的技巧。
            t=1.2*sq*(1.0+y*y)*exp(oldg-gammln(em+1.0)
                -gammln(en-em+1.0)+em*plog+(en-em)*pclog);
        } while (ran1(idum) > t);   拒绝，对每个偏离平均要发生大约 1.5 次。
        bnl=em;
    }
```

```
        if (p != pp) bnl=n-bnl;              记住要取消对称变换。
        return bnl;
    }
```

许多其他算法参阅 Devroye [2]和 Bratley [3]。

参考文献和深入阅读

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 125ff. [1]

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag), §X.4. [2]

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation*, 2nd ed. (New York: Springer-Verlag). [3].

7.4 随机位的生成

C++语言可以提供一个很有用的操作,以进行机器水平的逐位运算,诸如 << (左移)。本节将阐明如何充分利用这种性能。

问题是,当0和1是等可能的情形,如何生成单个随机位。当然,可以只在0和1之间生成一致随机偏离,并使用它们的高阶位(也就是检验它们是否大于或小于0.5),但这要进行很多运算。然而对某些特殊的应用,如实时信号处理,我们希望能非常快地生成随机位。

有一种生成随机位的方法基于"模2本原多项式"理论,它有两个不同的具体实现。对这些多项式理论的介绍超出了本书的讨论范围。虽然7.7节和20.3节将讨论它的一些内容,在此只给出以下说明。存在这样一种特殊多项式,它们的系数不是0就是1。例如,

$$x^{18}+x^5+x^2+x^1+x^0 \tag{7.4.1}$$

可简化而仅写出(系数)非零项的 x 的指数,即

$$(18,5,2,1,0)$$

每个 n 阶的(上面是 $n=18$)模2本原多项式都定义了,由前 n 个随机位得到一个新随机位的递推公式。这个递推公式保证可以产生一个极大长度的序列,也就是在它重复之前循环所有可能的 n 位的序列(全部为0除外)。因而,可以得到具有任意初始模式的(全部为0除外)序列,并在序列重复之前得到 2^n-1 个随机位。

将这些位从1(最近生成的)到 n(n 步以前生成的)进行编号,并记为 $a_1,a_2,\ldots,a_n$。对一个新位 a_0 给出公式。生成 a_0 之后,将所有的位做移位一步,使旧的 a_n 消失,并且新的 a_0 成为 a_1。然后再次应用此公式,依次类推。

"方法I"最容易用硬件实施,它只需要一个 n 位长的移位寄存器以及几个XOR门("异或"或按位模2加),C++语言用"^"表示这一操作。对以上给出的本原多项式,递推公式是

$$a_0=a_{18}\wedge a_5\wedge a_2\wedge a_1 \tag{7.4.2}$$

用^连在一起的这些项可视为移位寄存器上的"流出孔",经异或后作为寄存器的输入。更一般地说,除去常数项(零位项)外,对本原多项式的每一个非零系数恰好都有一项与之对应。因而对一般 n 阶本原多项式,第一项始终是 a_n,最后一项可能是 a_1 或不是 a_1,它取决于本原多项式中是否有 x^1 的项。

尽管用硬件实施很简单,但用C++语言来实现方法I是相当繁琐的,因为所有位必须被一个全字屏蔽序列收集:

```
#include "nr.h"

int NR::irbit1(unsigned long &iseed)
以整数返回一个随机位,它是以 iseed 中18个低有效位为基础的(对下一次调用,iseed 被修改)。
{
    unsigned long newbit;                    累加的 XOR。

    newbit = ((iseed >> 17) & 1)             得到位18。
        ^ ((iseed >> 4) & 1)                 具有位5的异或。
```

```
        ^ ((iseed >> 1) & 1)              具有位2的异或。
        ^ (iseed & 1);                    具有位1的异或。
    iseed=(iseed << 1) | newbit;          将种子值向左移位并在它的位1中放置异或的结果。
    return int(newbit);
}
```

"方法Ⅱ"不太适合直接硬件实施(尽管仍有可能),但更适合C++语言。随着每个新位生成,它在保存的 n 个位中修正了不止一位(参见图7.4.1)。它能生成极大长度序列,但与"方法Ⅰ"顺序不同。

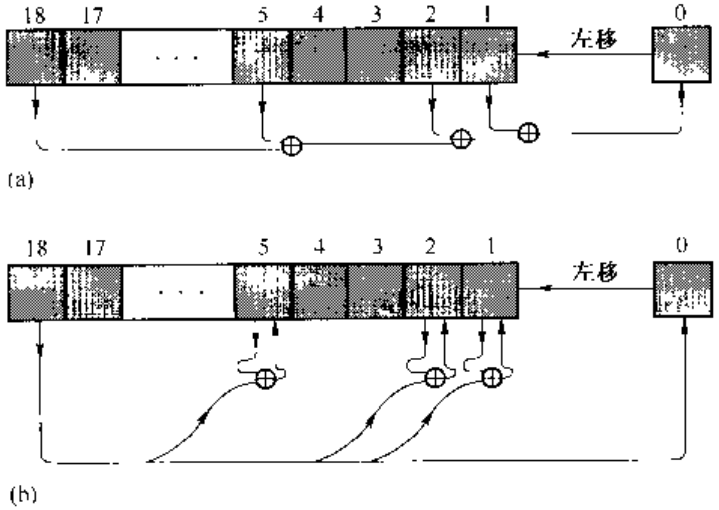


图7.4.1 从移位寄存器和模2本原多项式得到随机位的两个有关方法。(a)所挑选的"流出孔"的内容由"异或"(按位模2加)所组合,其结果从右边进行移位,这个方法最容易用硬件实施,(b)挑选出来的位通过它与最左边位的"异或"得以修正,然后最左边的位从右边移入,这个方法最容易用软件实施

对于本原多项式(7.4.1)应有以下关系式:

$$\begin{aligned} a_0 &= a_{18} \\ a_5 &= a_5 \wedge a_0 \\ a_2 &= a_2 \wedge a_0 \\ a_1 &= a_1 \wedge a_0 \end{aligned} \tag{7.4.3}$$

一般在本原多项式中,除0次项和 n 次项外,对每个非零项都要进行一次异或操作。方法Ⅱ的优点是,所有的异或操作通常都可由单个屏蔽字的异或操作来完成。

```
#include "nr.h"

int NR::irbit2(unsigned long &iseed)
以整数返回一个随机位,它是以iseed中18个低有效位为基础的(对下一次调用,iseed被修改)。
{
    const unsigned long IB1=1,IB2=2,IB5=16,IB18=131072;      2的幂。
    const unsigned long MASK=IB1+IB2+IB5;

    if (iseed & IB18) {            改变所有屏蔽位,位移,并将1放进位1中。
        iseed=((iseed ^ MASK) << 1) | IB1;
        return 1;
    } else {                       位移并将0放进位1中。
        iseed <<= 1;
        return 0;
    }
}
```

注意:不要使用这些程序的连续位,作为一个很大而且猜想为随机的整数位,或者作为一个猜想为随机浮点数的尾数位。在那些情况下,它们并不是十分随机的,参见Knuth[1]。这些随机位可利用的示例是:(i)以快速的"芯片速率"随机地用±1乘上一个信号,使得它的谱在一给定的带通内,一致地(可恢复)展开,或者(ii)二叉树的蒙特卡罗探测,其中究竟向左或向右分支是随机做出的。

现在，不必仔细研究前面的例子中18次本原多项式具有哪些独特之处（之所以选择18次，是为了读者能直接用数值实验证实我们的陈述，而2^{18}已足够小了）。表7.4.1列出了一些模2的本原多项式[2]，阶数高至100（事实上，每一阶都存在一些本原多项式，例如，可参阅7.7节高至10阶的完整列表）。

表7.4.1 模2本原多项式

(1,0)	(51,6,3,1,0)
(2,1,0)	(52,3,0)
(3,1,0)	(53,6,2,1,0)
(4,1,0)	(54,6,5,4,3,2,0)
(5,2,0)	(55,6,2,1,0)
(6,1,0)	(56,7,4,2,0)
(7,1,0)	(57,5,3,2,0)
(8,4,3,2,0)	(58,6,5,1,0)
(9,4,0)	(59,6,5,4,3,1,0)
(10,3,0)	(60,1,0)
(11,2,0)	(61,5,2,1,0)
(12,6,4,1,0)	(62,6,5,3,0)
(13,4,3,1,0)	(63,1,0)
(14,5,3,1,0)	(64,4,3,1,0)
(15,1,0)	(65,4,3,1,0)
(16,5,3,2,0)	(66,8,6,5,3,2,0)
(17,3,0)	(67,5,2,1,0)
(18,5,2,1,0)	(68,7,5,1,0)
(19,5,2,1,0)	(69,6,5,2,0)
(20,3,0)	(70,5,3,1,0)
(21,2,0)	(71,5,3,1,0)
(22,1,0)	(72,6,4,3,2,1,0)
(23,5,0)	(73,4,3,2,0)
(24,4,3,1,0)	(74,7,4,3,0)
(25,3,0)	(75,6,3,1,0)
(26,6,2,1,0)	(76,5,4,2,0)
(27,5,2,1,0)	(77,6,5,2,0)
(28,3,0,)	(78,7,2,1,0)
(29,2,0)	(79,4,3,2,0)
(30,6,4,1,0)	(80,7,5,3,2,1,0)
(31,3,0)	(81,4,0)
(32,7,5,3,2,1,0)	(82,8,7,6,4,1,0)
(33,6,4,1,0)	(83,7,4,2,0)
(34,7,6,5,2,1,0)	(84,8,7,5,3,1,0)

(续表)

(35,2,0)	(85,8,2,1,0)
(36,6,5,4,2,1,0)	(86,6,5,2,0)
(37,5,4,3,2,1,0)	(87,7,5,1,0)
(38,6,5,1,0)	(88,8,5,4,3,1,0)
(39,4,0)	(89,6,5,3,0)
(40,5,4,3,0)	(90,5,3,2,0)
(41,3,0)	(91,7,6,5,3,2,0)
(42,5,4,3,2,1,0)	(92,6,5,2,0)
(43,6,4,3,0)	(93,2,0)
(44,6,5,2,0)	(94,6,5,1,0)
(45,4,3,1,0)	(95,6,5,4,2,1,0)
(46,8,5,3,2,1,0)	(96,7,6,4,3,2,0)
(47,5,0,)	(97,6,0)
(48,7,5,4,2,1,0)	(98,7,4,3,2,1,0)
(49,6,5,4,0)	(99,7,5,4,0)
(50,4,3,2,0)	(100,8,7,2,0)

参考文献和深入阅读

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), pp. 30ff. [1]

Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (Cambridge: Cambridge University Press), §§9.32–9.37.

Tausworthe, R.C. 1965, *Mathematics of Computation*, vol. 19, pp. 201–209.

Watson, E.J. 1962, *Mathematics of Computation*, vol. 16, pp. 368–369. [2]

7.5 深入讨论:基于数据加密的随机序列

在本书的第一版中,讲述了如何利用数据加密标准(Data Encryption Standard, DES)[1~3]来产生随机数。遗憾的是,当用高级语言,如C++语言来实现软件时,数据加密标准慢得难以让人接受。其实,上述这种实施明显地看出是弊大于利。在这里给出一个简洁的算法,尽管它的加密性能稍差些,但它能产生同样有用的随机数。

数据加密标准,像先前的加密系统LUCIFER一样,就是通常所说的"分组乘积加密"[4]。它以迭代方式(16次)采用一种高度非线性的位混合操作,作用于64位输入。图7.5.1表示出这种混合过程的DES信息流程。函数g称为"密码函数",它接受32位输出32位。Meyer和Matyas[4]论述了这种密码函数的重要性在于它的非线性及其他设计标准。

数据加密标准用一个复杂的作用于短序列的连续位的查找表,和位的置换系统来构成它的密码函数g。显然,这个函数有很强的加密功能(或许如某些批评家所争论的那样,有极微妙难辨认的保密缺陷)。为了我们的目的,最好采用另一种函数g,它在用高级语言时能快速计算。这种函数可能会减弱一些算法的加密性。但目的不仅是加密,还要最快地找到函数g,迭代最少的混合过程(参见图7.5.1)。这个可通过标准测试的输出随机序列,经常可用来产生随机数。这个合成的算法将不是数据加密标准,而多半是一种更适合我们目标的"伪随机加密标准"。

按上述准则,函数g应是非线性的。在函数g中把整数乘法操作放在一个重要的位置。因为在高级语言中,一般不采用64位寄存器,必须把16位乘法操作限制在32位的结果中。因而,关于函数g的大致构思是,

必须分别计算输入半字的高16位、低16位的三个乘法，得出三个不同32位乘积，然后经过快速运算（如用"加"或者"异或"运算）组合它们，或许再加上个固定常数，或为一个32位结果。

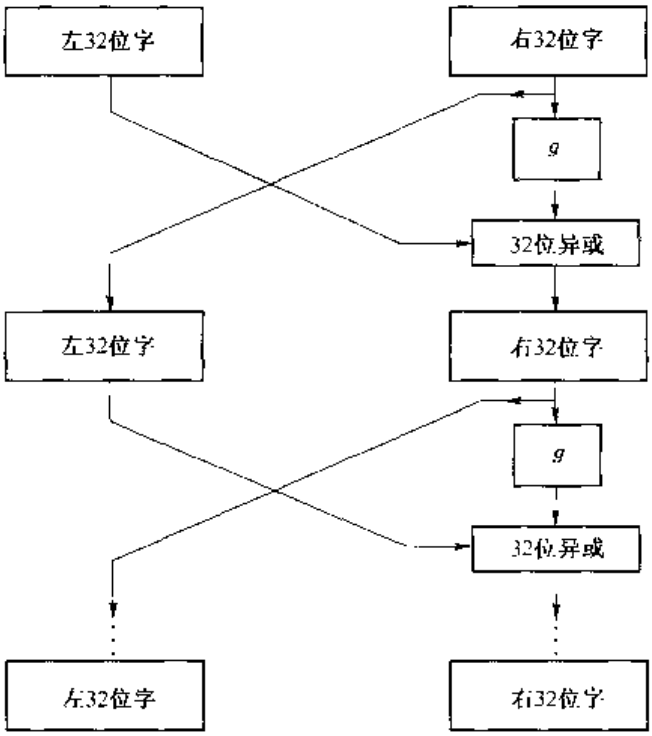


图7.5.1 图中显示了数据加密标准（DES）。对两组32位码字进行非线性函数g的迭代（根据Meyer和Matyas）[4]

有几个有效的实现方法，可选择其中之一进行系统研究。对输出随机性的测试和实验得出图7.5.2所示操作步骤。图中几个新元件需要解释一下：值C_1与C_2是固定常数，它们是随机选择的，但其中必须准确包含有16位1和16位0。通过异或操作组合这些常数以保证整个g没有倾向性偏0位或1位。

在图7.5.2中，弄清楚"反转半字"操作是非常重要的，否则最低位与最高位将不会被三个乘积适当地组合。因而在g中隐含的几项选择是：在纵向"流水线"的什么位置进行反转？按什么次序组合这三个乘积与常数C_2？每一个组合都需要进行"加"及"异或"操作吗？在确定图中算法之前，对这些选择需要多进行一些试验

最少迭代次数N_u的确定可以简略，有意义的N_u最小值显然是2，因为一次迭代只能简单地移动一个32位而不会改变它。可以利用常数C_1和C_2来帮助确定合适的N_u值：当$N_u=2$，且$C_1=C_2=0$（有意粗劣的选择），生成程序使一些随机性的测试失败。另一方面，当$N_u=4$或$N_u=2$，但常数C_1和C_2不太大，这时从这个结构得到的相当于10^9个浮点数r_i的序列里，我们还没有发现对随机性的任何统计偏差。因而$N_u=4$及不太小的C_1和C_2的组合给出的序列是随机的，对其随机性的测试已远超出我们的实际实验。在$N_u=2$（当然快两倍）时，并没有发现它的不随机性，尽管事实如此，还是推荐使用较为保守的参数值（$N_u=4$）。

这些思想的实现简单易懂。下述程序不是可随便移植的。因为已设无符号长整数是32位，是大多数机器的实际情形。然而，并不是说**较长**的整数总是不好的（其常数C_1，C_2有适当的范围）。C++语言并不提供便于移植的方法，以便分割一个长整数为半字，故必须使用逻辑与屏蔽（&0xffff），并左移或右移16位（<<16或>>16）。在某些机器上半字提取比使用C++的"逻辑与"结构要快得多，但这在"长结尾"与"短结尾"机器之间一般是不可移植的（长、短结尾指的是将字节存放到一个字里的次序）。

```
#include "nr.h"

void NR::psdes(unsigned long &lword, unsigned long &irword)
```

64位字（lword，irword）的"伪DES"混洗。这两个32位量在所有位都被混洗了。

```
{
    const int NITER=4;
    static const unsigned long c1[NITER]={
        0xbaa96887L, 0x1e17d32cL, 0x03bcdc3cL, 0x0f33d1b2L};
    static const unsigned long c2[NITER]={
        0x4b0f3b58L, 0xe874f0c3L, 0x6955c5a6L, 0x55a7ca46L};
    unsigned long i,ia,ib,iswap,itmph=0,itmpl=0;

    for (i=0;i<NITER;i++) {
    实现DES逻辑运算的niter次迭代，采用简单的、非加密的非线性函数替换DES。
        ia=(iswap=irword) ^ c1[i];              常数c1和c2(下面)保证了非线性混合。
        itmpl = ia & 0xffff;
        itmph = ia >> 16;
        ib=itmpl*itmpl+ ~(itmph*itmph);
        irword=lword ^ (((ia = (ib >> 16) |
            ((ib & 0xffff) << 16)) ^ c2[i])+itmpl*itmph);
        lword=iswap;
    }
}
```

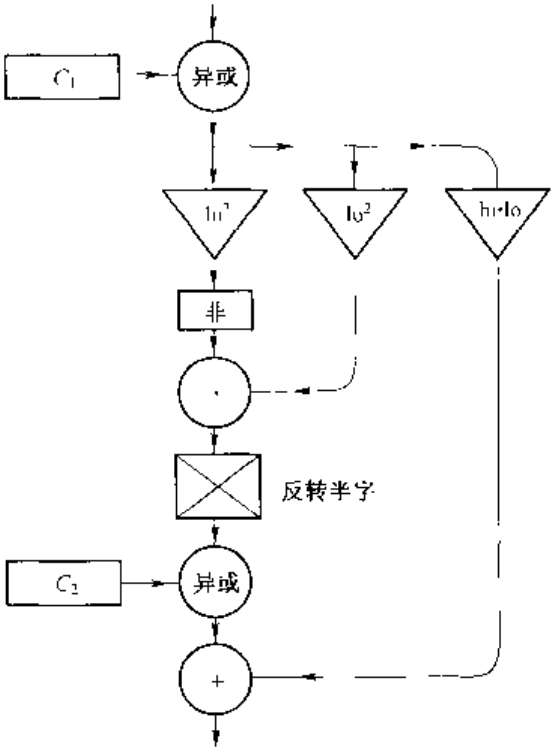


图 7.5.2 程序 psdes 使用的非线性函数 g

列在下面的程序 ran4 使用 psdes 以产生均匀的随机偏差。采用习惯做法，将变量 idum 的负值放在左 32 位，而正值 i 放在右 32 位，返回第 i 个随机偏差，及 idum 到 $i+1$ 的增量。这是为了便于定义许多不同的序列(idum 的负值)，但还要随机存取每一个序列(idum 的正值)。为了从 32 位整数中得到一个浮点数，我们采用了 7.1 节末尾叙述的掩蔽技巧，十六进制常数 3F800000 和 007FFFFF 适合于用 IEEE 表示的计算机的 32 位浮点(如 Pentium 机和许多 Linux 或 UNIX 工作站)。对于 DEC VAX 机，正确的十六进制常数分别为 00004080 和 FFFF007F。若要扩大可移植性，可以用有符号非负的 32 位整数来构成一个浮点数(典型地，如是负数，则刚好加 2^{31})，然后用一个浮点常数乘它(典型地为 2^{-31})。

程序 ran4 一个有用的特点是，它允许在序列中随机访问第 n 个随机数值，而不需要先生成 $1\cdots n-1$ 个数值。任何基于**混洗**技术的随机数生成程序都会具有这一特性(混洗技术，数值高度群集的数据键的映射技术，几乎均匀地进入一个存储地址空间)[5,6]。有时会遇到一种模拟问题，其中存在的罕见情况，往往只有等到它出现并产生相当大的影响时才被人们认识到。我们希望使用相同的随机值，也就是说改变某些控制参数，使这种情况能模拟返回并重新开始。这类的问题诸如"在循环数337 098 901中使用了什么随机数?"，然而，在提出这个问题之前它可能已经是循环数395 100 273了。基于递归的随机数生成程序比基于混洗技术的随机数生成程序更不易回答这类问题。

```
#include "nr.h"

DP NR::ran4(int &idum)
返回在0.0到1.0范围内的一致随机偏离.它由64位字(idums,idum)经伪DES(与DES类似)混洗而生成的,其中idums
是用负值idum前次调用的设置,并增加idum.通过连续调用,本程序可用来产生随机序列,而在连续调用之间保持idum
不变;当用idum=n来调用时,本程序能够在一序列中随机访问第n位偏离值。当用不同的idum的负值调用,则不同的
序列被初始化.
{
The hexadecimal constants jflone and jflmsk below are used to produce a floating number
between 1 and 2 by bitwise masking. They are machine-dependent. See text.
#if defined(vax) || defined(_vax_) || defined(__vax__) || defined(VAX)
    static const unsigned long jflone = 0x00004080;
    static const unsigned long jflmsk = 0xffff007f;
#else
    static const unsigned long jflone = 0x3f800000;
    static const unsigned long jflmsk = 0x007fffff;
#endif
    unsigned long irword,itemp,lword;
    static int idums = 0;

    if (idum < 0) {                          重新设置idums并且准备返回序列中的第
        idums = -idum;                        一个偏离值。
        idum=1;
    }
    irword=idum;
    lword=idums;
    psdes(lword,irword);                     用"伪DES"对字编码。
    itemp=jflone | (jflmsk & irword);        将浮点数掩蔽在1至2之间。
    ++idum;
    return (*(float *)&itemp)-1.0;           相减后移到0至1的范围。
}
```

表7.5.1给出一些数据,以检验用户的机器上ran4和psdes是否能正常工作。不推荐应用ran4,除非能复制所示的十六进制的数值。一般情况下,ran4的速度是ran0(参见7.1节)的1/4,是ran1的1/3。

表7.5.1 检验psdes能否实现的数值

idum	psdes调用前		psdes调用后(十六进制)		ran4(idum)	
	lword	irword	lword	irword	VAX	Pentium
-1	1	1	604D1DCE	509C0C23	0.275 898	0.219 120
99	1	99	D97F8571	A66CB41A	0.208 204	0.849 246
-99	99	1	7822309D	64300984	0.034 307	0.375 290
99	99	99	D7F376F0	59BA89EB	0.838 676	0.457 334

逐次用变量-1,99,-99和99调用psdes,应得到表中显示的正确的lword与irword数值。掩蔽转换返回的浮点随机值将由机器决定,表中显示了VAX和Pentium的数值。

参考文献和深入阅读

Data Encryption Standard, 1977 January 15, Federal Information Processing Standards Publication, number 46 (Washington: U.S. Department of Commerce, National Bureau of Standards). [1]

Guidelines for Implementing and Using the NBS Data Encryption Standard, 1981 April 1, Federal Information Processing Standards Publication, number 74 (Washington: U.S. Department of Commerce, National Bureau of Standards). [2]

Validating the Correctness of Hardware Implementations of the NBS Data Encryption Standard, 1980, NBS Special Publication 500-20 (Washington: U.S. Department of Commerce, National Bureau of Standards). [3]

Meyer, C.H. and Matyas, S.M. 1982, *Cryptography: A New Dimension in Computer Data Security* (New York: Wiley). [4]

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), Chapter 6. [5]

Vitter, J.S., and Chen, W-C. 1987, *Design and Analysis of Coalesced Hashing* (New York: Oxford University Press). [6]

7.6 简单的蒙特卡罗积分

数值方法上的灵感也可能来自于不相同的信源。"样条"来源于绘图员使用的木质条板。"模拟退火"(将在 10.9 节讨论)是来源于热力学模拟。更何况大家至少耳闻过"蒙特卡罗积分方法"。

假设在一个多维体积 V 中,选取 N 个均匀分布的随机点,称它们为 $x_0,\ldots,x_{N-1}$。于是,蒙特卡罗(Monte Carlo)积分的基本定理指出,函数 f 在多维体积上积分近似为

$$\int f\,dV \approx V\langle f\rangle \pm V\sqrt{\frac{\langle f^2\rangle-\langle f\rangle^2}{N}} \tag{7.6.1}$$

其中角括号($\langle\rangle$)表示对 N 个取样点上的算术平均值,

$$\langle f\rangle \equiv \frac{1}{N}\sum_{i=0}^{N-1} f(x_i) \qquad \langle f^2\rangle \equiv \frac{1}{N}\sum_{i=0}^{N-1} f^2(x_i) \tag{7.6.2}$$

(7.6.1)式中的"加或减"项表示积分的标准偏差,而不是严格的界。进一步说,它不保证误差服从高斯分布,所以此误差项只可看成可能误差的粗略估计。

假设要对函数g在一个不容易随机取样的区域 W 上积分。例如,W 可能具有非常复杂的形状。毫无疑问,只需找一个既包含 W 又能够很容易取样的区域V(参见图 7.6.1),然后定义函数 f,它在 W 内等于g,在 W 外(但仍在取样的 V 中)等于零,取 V 尽可能靠近 W,因 f 的零值会增加(7.6.1)式的误差估计项。最好是,W 之外所选择的取样点没有信息内容,使有效点数 N 减少。(7.6.1)式中的误差估计项要考虑这个问题。

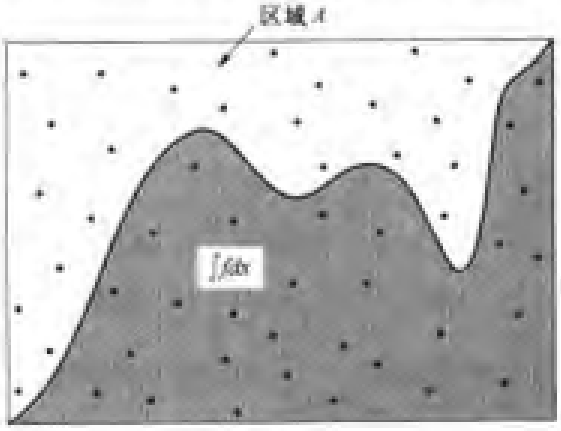


图 7.6.1 蒙特卡罗积分。随机点在区域 A 中选取,函数 f 的积分近似为曲线f下随机点的部分乘上A的面积。这个过程的重新细分可以改进这个方法的准确性,详见正文

给出蒙特卡罗积分的一个通用程序是相当复杂的(参见 7.8 节),而一个示例将表明这种方法的简要原理。假设要对一个形状复杂的物体,例如一个由大方框定界的圆环的交,寻求它的重量和质心的位置。具体说,设该物体受如下三个联合条件限制:

$$z^2+\left(\sqrt{x^2+y^2}-3\right)^2 \leqslant 1 \tag{7.6.3}$$

(圆环中心位于原点,长半径为 4,短半径为 2。)

$$x \geqslant 1 \qquad y \geqslant -3 \tag{7.6.4}$$

(方框的两个面,参见图 7.6.2。)此时假设,物体具有定常密度 ρ。

要估计如下复杂物体内部的积分:

$$\int \rho\, dx\, dy\, dz \qquad \int x\rho\, dx\, dy\, dz \qquad \int y\rho\, dx\, dy\, dz \qquad \int z\rho\, dx\, dy\, dz \tag{7.6.5}$$

其质心坐标是后三个积分(线性矩量)与第一个积分(重量)之比。

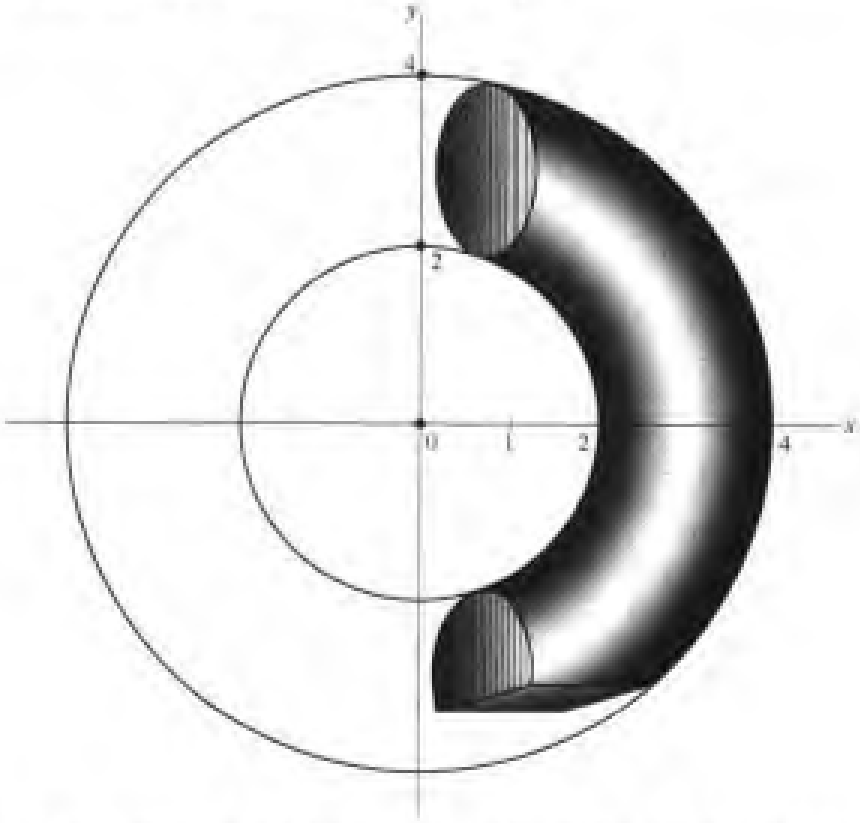


图7.6.2 蒙特卡罗积分的示例(见正文)。积分区域是一段圆环,由两个平面的交界限定。区域的界限不容易写成解析封闭形式,所以蒙特卡罗是一个有用的方法

在下面的程序段中,包围圆环段 W 的区域 V 是一个长方箱体,其范围是 $x=1\sim4, y=-3\sim4, z=-1\sim1$。

```
#include "nr.h"
...
n=...                                   设置所求的样本点个数。
den=...                                 设置密度的常数值。
sw=swx=swy=swz=0.0;                     零化累加的各种和式。
varw=varx=vary=varz=0.0;
vol=3.0*7.0*2.0;                        取样区域的体积。
for(j=0;j<n;j++) {
    x=1.0+3.0*ran2(idum);               在取样区域中随机挑选一点。
    y=(-3.0)+7.0*ran2(idum);
    z=(-1.0)+2.0*ran2(idum);
    if (z*z+SQR(sqrt(x*x+y*y)-3.0) < 1.0) {      它在圆环中吗?
        sw += den;                      如果在圆环中,则加上各种累积量。
        swx += x*den;
        swy += y*den;
        swz += z*den;
        varw += SQR(den);
        varx += SQR(x*den);
        vary += SQR(y*den);
        varz += SQR(z*den);
    }
}
w=vol*sw/n;                             积分式(7.6.5)的值,
x=vol*swx/n;
y=vol*swy/n;
z=vol*swz/n;
dw=vol*sqrt((varw/n-SQR(sw/n))/n);      及其对应的误差估计。
dx=vol*sqrt((varx/n-SQR(swx/n))/n);
dy=vol*sqrt((vary/n-SQR(swy/n))/n);
dz=vol*sqrt((varz/n-SQR(swz/n))/n);
```

在蒙特卡罗积分中,变量的替换是很有用的。例如,假设要计算同样的积分,但只是圆环段的密度依赖于坐标 z 而变化,事实上它按照公式

$$\rho(x,y,z)=e^{5z} \tag{7.6.6}$$

变化。计算这个积分的一个方法是,刚好是在第一次使用 den 之前,将语句

```
den=exp(5.0*z);
```

嵌在 if(...)块中。这样是可行的,但是个糟糕方法。因为(7.6.6)式随 z 下降(降到它的下限 -1)迅速到零,这时大多数取样点对重量或矩量的和式几乎无用。这些点实际上浪费掉了就像这些点都位于区域 W 之外似的。而变量替换,如同 7.2 节的交换方法所做的,能解决这个问题。令

$$ds=e^{5z}dz \qquad \text{使得} \qquad s=\frac{1}{5}e^{5z},\quad z=\frac{1}{5}\ln(5s) \tag{7.6.7}$$

则 $\rho dz=ds$,由界限 $-1<z<1$,变成界限 $0.00135<s<29.682$。现在程序段如下:

```
#include "nrutil.h"
...
n=...                                   设置需求的样本点个数。
sw=swx=swy=swz=0.0;
varw=varx=vary=varz=0.0;
ss=0.2*(exp(5.0)-exp(-5.0))             需要随机取样的s区间。
vol=3.0*7.0*ss                          在x,y,s空间中的体积。
for(j=0;j<n;j++) {
    x=1.0+3.0*ran2(idum);
    y=(-3.0)+7.0*ran2(idum);
    s=0.00135+ss*ran2(idum);            挑选s中的一点。
    z=0.2*log(5.0*s);                   (7.6.7)式。
    if (z*z+SQR(sqrt(x*x+y*y)-3.0) < 1.0) {
        sw += 1.0;                      因为考虑了s的定义,密度是1。
        swx += x;
        swy += y;
        swz += z;
        varw += 1.0;
        varx += x*x;
        vary += y*y;
        varz += z*z;
    }
}
w=vol*sw/n;                             积分(7.6.5)式的值。
x=vol*swx/n;
y=vol*swy/n;
z=vol*swz/n;
dw=vol*sqrt((varw/n-SQR(sw/n))/n);      及其对应的误差估计。
dx=vol*sqrt((varx/n-SQR(swx/n))/n);
dy=vol*sqrt((vary/n-SQR(swy/n))/n);
dz=vol*sqrt((varz/n-SQR(swz/n))/n);
```

稍加思考就可看出,(7.6.7)式之所以有用,是因为我们那时想要消去(e^{5z})的被积函数部分是解析可积的,并且有一个能够解析求逆的积分(比较 7.2 节)。一般情况,这些性质不具备。可去掉被积函数可以求积分和求逆的"最佳"因子。"最佳"的准则就是,试图把剩下的被积函数约化成一个尽可能靠近常数的函数。

极限情形是有启发性的:如果设法使得被积函数 f 准确地为常数,如果已知体积的区域 V 准确包住所求的区域 W,则计算的 f 的平均值将准确为它的常数值,并且(7.6.1)式中误差估计准确地为零。事实上,将能够准确地求得积分,蒙特卡罗数值求值却成了多余的。所以,由极端情形稍作后退,以达到能用变量替换使函数 f 近似为常数的程度,并且达到能够在一个只稍比 W 大的区域内取样的程度,这就能提高蒙特卡罗积分的准确性。这种技术在文献中一般称为**方差退化**。

蒙特卡罗积分的缺点是,它的准确性只随样本点个数 N 的平方根增长。如果对准确性要求适中,或计算机的预算量很大,那么这个技术作为极为普遍的技术是很值得推荐的。在后面的两节中,将了解到"消除 N 的平方根这一障碍"的现有技术和成果,至少在某些场合,少数几个函数的计算具有较高的精确度。

参考文献和深入阅读

Hammersley, J.M., and Handscomb, D.C. 1964, *Monte Carlo Methods* (London: Methuen).
Shreider, Yu. A. (ed.) 1966, *The Monte Carlo Method* (Oxford: Pergamon).
Sobol', I.M. 1974, *The Monte Carlo Method* (Chicago: University of Chicago Press).
Kalos, M.H., and Whitlock, P.A. 1986, *Monte Carlo Methods* (New York: Wiley).

7.7 准随机序列

已经知道，在 n 维空间中，均匀随机地选择 N 个点，会在蒙特卡罗积分中引起一个误差项，它以 $1/\sqrt{N}$ 形式下降。总体上，每一个新方程的取样点都线性地加到累加的和上，并成为函数平均值，同时也线性地加到累加的平方和上，并成为方差((7.6.2)式)。误差估计来自这个方差的平方根，即幂 $N^{-1/2}$。

这个平方根的收敛性是常见的，但并非总是如此。一个简单的反例是，在笛卡儿网格上选择取样点，一个网格取一点(无论什么次序)。于是，蒙特卡罗方法成为一个确定的求积结构——尽管是一个简单的——它的相对误差下降至少有 N^{-1} 那样快(如果函数在取样区域的边界上平滑趋于零，或在此区域内是周期性函数，则它下降得会更快)。

有关网格的一个问题是，必须**预先决**定它应有什么精度，然后完成全部取样点。用网格进行取样，直到收敛或遇到临界时终止，这是不方便的。有人也许会问，是否存在某种中间方案，随机地拾取样点的方法，并且扩展出某种自身回避的方法，它能回避均匀随机点的群集的发生机会。

除此以外，还有一种类似的问题。可能要为某个点而搜索一个 n 维空间，那里(局部可计算)的条件是满足的。当然，为使计算有意义，最好空间必须是连续的，以使在一个有限的 n 维邻域内，所求的条件是满足的。尽管可能事先不知道这个邻域究竟有多大，但我们将平滑地更加细分尺度以增加取样点，直到发现所求的点为止。是否还存在比不相关的随机取样法更好的方法呢?

对上面问题的回答是肯定的。比不相关的随机点更均匀地充满 n 维空间的 n 元序列，被称为**准随机序列**，这个术语有点用词不当，因为准随机序列根本不"随机"，事实上它们很明显地是次随机。严格地说，在准随机序列中，取样点是尽可能互相"最大地回避"。

一个简单的、概念上的例子是 **Halton 序列**[1]。在一维空间，这个序列第 j 个数 H_j 按以下步骤得到：(i)写出 j 作为以 b 为基的数，b 为某个素数(例如，$j=17$ 时得出以 $b=3$ 为基的数为 122)。(ii)反转这个数，并在这个序列前置一个小数点(即以基数 b 为基的小数点，例如在上例中可得以 3 为基的数 0.221)，所得结果就是 H_j。为在 n 维空间得到 n 元序列，可以用不同的素数为基数 b，作为 Halton 序列的每一个成分。一般情况采用前 n 个素数。

Halton 序列的工作原理是：每次数字 j 的个数增加一位，j 的反转数字的尾数变成了精细网格 b 的一个因子。因此这个过程是在越来越精细的笛卡儿网格中填充所有点的每一个点——并且在每一网格上的扩展数量级是最大的(因为数字 j 中最急剧变化的数，控制着最有意义的小数)。

Faure、Sobol'、Niederreiter 等人提出过产生准随机序列的其他方法。Bratley 和 Fox [2]给出了一个很好的论述，并讨论了由 Antonov 和 Saleev [4]提拱的 Sobol'[3]序列的特别有效的变形。我们现在要讨论的就是 Antonov-Saleev 变形的实现。

从一组特殊的 w 个二进制尾数 V_i，$i=1,2,\ldots,w$ 中，Sobol'序列产生 0 与 1 之间的数，直接作为长度为 w 位的二进制尾数，这称为**方向数**。在 Sobol'原来的方法中，第 j 个数 X_j 是由满足条件的一些数组 V_i(逐位异或)产生的，其条件是 V_i 的 i 要满足"j 的第 i 位为非零"。换言之，当 j 增加时，V_i 中的不同的 1 以不同的时间出现或消失在 X_j 中。这就使 V_1 交替地出现和消失，变化得非常快，而 V_k 从出现到消失(或反之)要每 2^{k-1} 步产生一次。

Antonov 和 Saleev 的贡献是用整数 j 的比特位替换方向数。人们也可用 j 的**格雷码** $G(j)$ 的比特位(格雷码的简要论述参见 20.2 节)。

现在，$G(j)$ 与 $G(j+1)$ 正好只有一位不同，即 j 的二进制表示的最右边零位的位置(如需要，在 j 中加上一个前置零)。这样，第 $j+1$ 个 Sobol'-Antonov-Saleev 数，可以从第 j 个数与单个 V_i"异或"来得到，即 j 中最右边的零位 i。将看到，这使得序列的计算非常有效

图 7.7.1 画出了二维 Sobol'序列产生的前 1024 个点。可以看出从相继点能"知道"前边留下的空隙,并分层地填充这些空隙

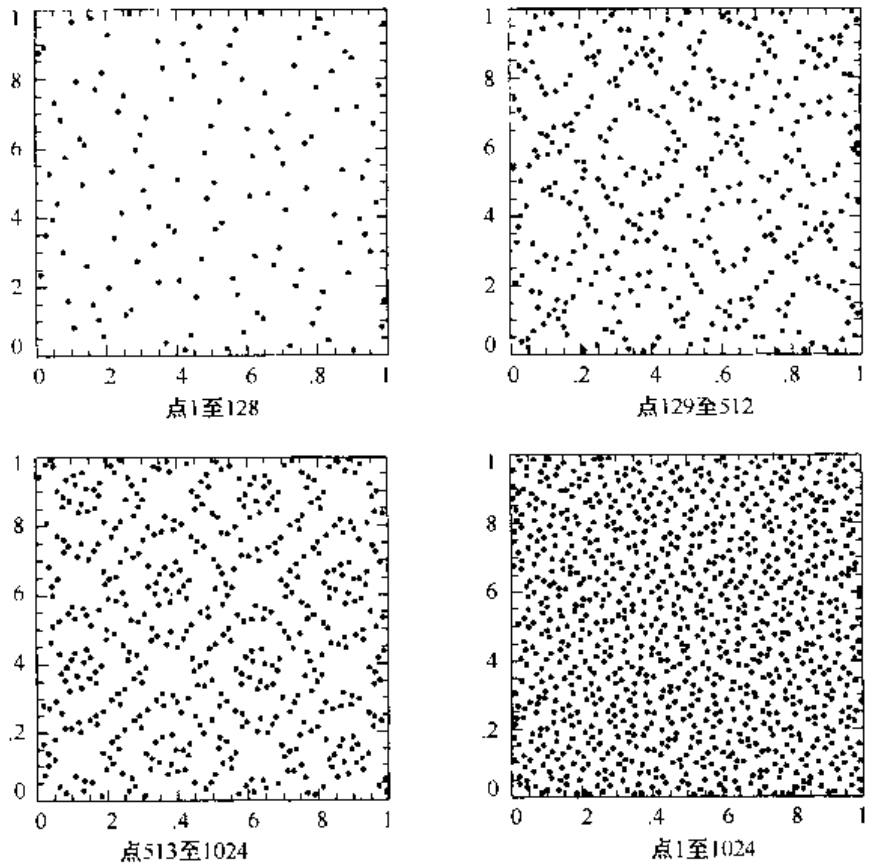


图 7.7.1 二维 Sobol'序列的前 1024 个点。该序列由数论生成,而不是随机的,故在任何阶段,相继点都"知道"应该如何填充到前面已有分布的空隙中

现在讨论方向数 V_i 是怎样产生的。许多优秀的数学家正在从事这项工作,所以此处只进行简介以满足说明问题的需要:每一个不同的 Sobol'序列(或 n 维序列的分量)是基于整数模 2 的各种本原多项式,即该多项式系数为 0 或 1,并且产生了一个最大长度的移位寄存器(模 2 的本原多项式在 7.4 节中使用过,在 20.3 节中还将进一步讨论)。假设 P 是这样一个 q 次多项式

$$P = x^q + a_1 x^{q-1} + a_2 x^{q-2} + \cdots + a_{q-1} x + 1 \tag{7.7.1}$$

用 q 项递推关系定义一个整数序列 M_i,

$$M_i = 2a_1 M_{i-1} \oplus 2^2 a_2 M_{i-2} \oplus \cdots \oplus 2^{q-1} M_{i-q+1} a_{q-1} \oplus (2^q M_{i-q} \oplus M_{i-q}) \tag{7.7.2}$$

其中逐位异或以⊕表示。这个递推关系的开始值 $M_1, \ldots M_q$ 是可以分别小于 $2, \ldots, 2^q$ 的任意奇整数。则方向数 V_i 为

$$V_i = M_i / 2^i \qquad i = 1, \ldots, w \tag{7.7.3}$$

表 7.7.1 列出了所有级次 $q \leq 10$ 的模 2 本原多项式。由于各项系数为 0 或 1,且 x^q 和 1 项的系数总是 1,故用中间各项的系数作为一个二进制数的各比特位,由此来表示多项式是很方便的(x 的幂次越高有效位数越多)。该表使用如下约定。

表 7.7.1 模 2 本原多项式

阶次	模 2^n 本原多项式
1	0(即 $x+1$)
2	1(即 x^2+x+1)

（续表）

阶次	模2* 本原多项式
3	1,2(即 x^3+x+1 和 x^3+x^2+1)
4	1,4 (即 x^4+x+1 和 x^4+x^3+1)
5	2,4,7,11,13,14
6	1,13,16,19,22,25
7	1,4,7,8,14,19,21,28,31,32,37,41,42,50,55,56,59,62
8	14,21,22,38,47,49,50,52,56,67,70,84,97,103,115,122
9	8,13,16,22,25,44,47,52,55,59,62,67,74,81,82,87,91,94,103,104,109,122,124,137,138,143,145,152,157,167,173,176,181,182,185,191,194,199,218,220,227,229,230,234,236,241,244,253
10	4,13,19,22,50,55,64,69,98,107,115,121,127,134,140,145,152,158,161,171,181,194,199,203,208,227,242,251,253,265,266,274,283,289,295,301,316,319,324,346,352,361,367,382,395,398,400,412,419,422,426,428,433,446,454,457,472,493,505,508

*用1进制整数表示内部位(即忽略最高次的位和单位位)。

现在讨论 Sobol'序列的实现过程。连续调用 sobseq 函数(在初始调用后),返回一个 n 维 Sobol'序列的连续点,它是基于表7.7.1中前 n 个本原多项式。正如已给出的程序中初始化 n,使其最大为6维,字长 w 为30位。这些参数可通过 MAXBIT($=w$)和 MAXDIM 来改变,还可以通过在数组 ip(表中的本原多项式),mdeg(它的阶次),以及 iv(递归的初始数据,(7.7.2)式)中添加更多的初始数据来改变。在表7.7.2中,表明了程序中使用的初始数据。

表7.7.2 用于 sobseq 的初始值

阶次	多项式	初始值			
1	0	1	(3)	(5)	(15)...
2	1	1	1	(7)	(11)...
3	1	1	3	7	(5)...
3	2	1	3	3	(15)...
4	1	1	1	3	13...
4	4	1	1	5	9...

括号中的值不能随便使用,而要用这一阶次的递推关系得出。

```
#include "nr.h"

void NR::sobseq(const int n, Vec_O_DP &x)
当 n 为负值时,对 MAXDIM 每个不同的 Sobol'序列,内部初始化一组 MAXBIT 方向数。当 n 是正值时(但≤MAXDIM),以
向量 x[0...n-1]返回这 n 维序列的下一个值(在不同的初始化中 n 不能改变)。
{
    const int MAXBIT=30,MAXDIM=6;
    int j,k,l;
    unsigned long i,im,ipp;
    static int mdeg[MAXDIM]={1,2,3,3,4,4};
    static unsigned long in;
    static Vec_ULNG ix(MAXDIM);
    static Vec_ULNG_p iu(MAXBIT);
    static unsigned long ip[MAXDIM]={0,1,1,2,1,4};
    static unsigned long iv[MAXDIM*MAXBIT]=
```

```
        {1,1,1,1,1,1,3,1,3,3,1,1,5,7,7,3,3,5,15,11,5,15,13,9};
    static DP fac;

    if (n < 0) {                                    初始化，不返回向量。
        for (k=0;k<MAXDIM;k++) ix[k]=0;
        in=0;
        if (iv[0] != 1) return;
        fac=1.0/(1 << MAXBIT);
        for (j=0,k=0;j<MAXBIT;j++,k+=MAXDIM) iu[j] = &iv[k];
        允许1D和2D寻址。
        for (k=0;k<MAXDIM;k++) {
            for (j=0;j<mdeg[k];j++) iu[j][k] <<= (MAXBIT-1-j);
            所存储的值只要求正规化。
            for (j=mdeg[k];j<MAXBIT;j++) {          用递推得到另一个值。
                ipp=ip[k];
                i=iu[j-mdeg[k]][k];
                i ^= (i >> mdeg[k]);
                for (l=mdeg[k]-1;l>=1;l--) {
                    if (ipp & 1) i ^= iu[j-l][k];
                    ipp >>= 1;
                }
                iu[j][k]=i;
            }
        }
    } else {                                        计算序列中的下一个向量。
        im=in++;
        for (j=0;j<MAXBIT;j++) {                    找出最右边的零位。
            if (!(im & 1)) break;
            im >>= 1;
        }
        if (j >= MAXBIT) nrerror("MAXBIT too small in sobseq");
        im=j*MAXDIM;
        for (k=0;k<MIN(n,MAXDIM);k++) {             异或适当的方向数成为向量的每个
            ix[k] ^= iv[im+k];                      分量，并且转换成浮点数。
            x[k]=ix[k]*fac;
        }
    }
}
```

Sobol'序列的优点是，n 维平滑函数的蒙特卡罗积分，对随机取样数 N 相对误差以 $(\ln N)^n/N$ 的方式减小，也就是几乎和 $1/N$ 一样快。作为示例，设一个积分函数 f，它在三维空间的圆环内是非零的。如果圆环的主半径是 R_0，辅助半径的坐标 r 定义为

$$r = \left([(x^2+y^2)^{1/2} - R_0]^2 + z^2\right)^{1/2} \tag{7.7.4}$$

函数

$$f(x,y,z) = \begin{cases} 1+\cos\left(\dfrac{\pi r^2}{a^2}\right) & r < r_0 \\ 0 & r \geqslant r_0 \end{cases} \tag{7.7.5}$$

可在圆柱形坐标系得解析积分，有

$$\int\int\int dx\,dy\,dz\,f(x,y,z) = 2\pi^2 a^2 R_0 \tag{7.7.6}$$

(7.7.4)式在区间 $-1<x,y,z<1$ 内均匀取样，并以参数 $R_0=0.6$，$r_0=0.3$ 和由 sobseq 程序产生的两个不相关的随机点和 Sobol'序列，进行相继 100 次蒙特卡罗积分，结果示于图 7.7.2。图中曲线表示 100 次积分的均方根平均误差与取样点数之间的函数关系(对任一单个积分，固有漂移误差从正值到负值，或相反，所以相对误差的对数图没有多少信息)。图中细短曲线对应于不相关的随机取样点，并显示了常用的 $N^{-1/2}$ 渐近线。细实曲线显示了 Sobol'序列的结果。表达式 $(\ln N)^3/N$ 中的对数项已表示在曲线的曲率中，而渐近线 N^{-1} 是明确的。

为了理解图 7.7.2 的重要性，假设要求函数 f 的蒙特卡罗积分的精确度在 1%以内。Sobol'序列要进行几千次取样方可达到这个精确度，而伪随机取样几乎需要 10 万次。对于更高精确度的要求。这个比率甚至会更高。

与上不同，当被积函数在取样区域内有硬的边界(不连续的边界)，情况就不太理想了。例如，函数值在圆环内取 1，外面取 0：

$$f(x,y,z)=\begin{cases}1 & r<r_0\\ 0 & r\geqslant r_0\end{cases} \tag{7.7.7}$$

其中 r 由(7.7.4)式定义。恰巧,这个函数有与(7.7.5)式函数相同的解析积分,即 $2\pi^2 a^2 R_0$。

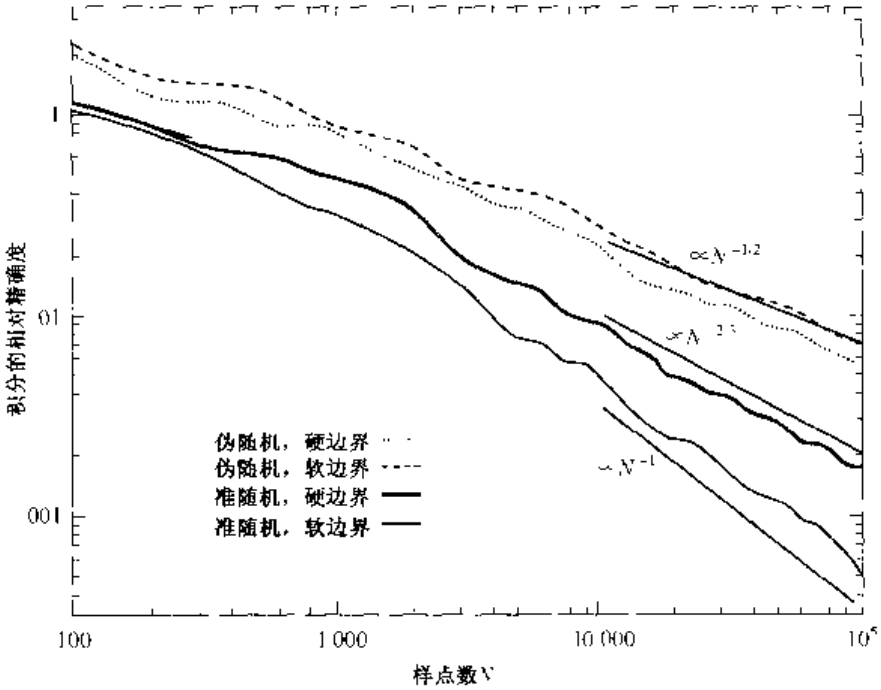


图7.7.2 蒙特卡罗积分的相对精确度和取样点数的函数关系。图中对应于两个不同被积函数与两个不同选择随机点的方法,准随机Sobol'序列收敛得比传统的伪随机序列快得多。对准随机取样来说,平滑的被积函数(软边界)将比阶跃跳变的被积函数(硬边界)好得多。曲线显示了100次试验的均方根平均值

细致分层次的Sobol'序列是基于一组笛卡儿网格,但(圆环面)边界与这些网格没有明确关系。结果是在圆环表面薄层上有量级为 $N^{2/3}$ 的取样点,它无论来自圆环面内,还是圆环外大致都是随机的。对这薄层应用平方根定律得到总数的 $N^{1/3}$ 偏离,或蒙特卡罗积分 $N^{-2/3}$ 相对误差。图7.7.2的粗实线证实了这个特性。图7.7.2中粗的短划线是函数(7.7.7)式用独立随机点的积分结果,尽管Sobol'序列的优点不像在平滑函数时那么突出,但它的精确度仍然适中,比如1%时,平滑函数的情况要优越一个因子(~5),当精确度提高时,这个因子还将增大。

我们提供的sobseq程序产生的序列都是从起点开始的,若要满足从某点而不从起点开始产生序列,这是很容易做的。一旦方向数iv初始化了,第 j 个点能够直接由 j 的格雷码中的非零位所对应的方向数进行异或得到。正如上面描述的那样。

7.7.1 拉丁超立方

我们在这里简要叙述一下**拉丁(Latin)平方或拉丁超立方**取样的不相关技术,当必须在 N 维空间中极稀疏地取样时(取样点为 M),这种技术是很有用的。例如,当要试验汽车的抗撞性能,它是4个不同的设计参数的联立函数,但从经济考虑只有3部汽车可供试验(这是不是一个好的方案倒不是关键,而是如何尽量利用现有的条件)。

设计思想是,将每一个设计参数(维数)分为 M 段,故整个空间分成 M^N 个小单元(每一维选择的段 M,可以相等或不相等)。例如,3部汽车4个参数就共有 $3\times3\times3\times3=81$ 个小单元。

其次,按下列算法选择 M 个小单元包含的取样点:随机地选择 M^N 个小单元中的一个,作为第一点。现除去与第一点有某一相同参数的所有小单元(也就是,划去与第一点在同一行或同一列的所有小单元),还剩下 $(M-1)^N$ 个小单元。再随机地选取其中之一,并划去新的行与列。一直重复这个步骤直到只剩下一个小单元,它包含最后一个样点。

这种结构的特点是,每一个设计参数都将在它的每一个子区域里经过测试。如果测试时系统响应由某

一个设计参数所控制,则这个参数会在这个取样技术中被发现。另一方面,如果各参数之间有强烈的相互影响,拉丁超立方将失去它的优势,所以使用时要小心。

参考文献和深入阅读

Halton, J.H. 1960, *Numerische Mathematik*, vol. 2, pp. 84–90. [1]

Bratley P., and Fox, B.L. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 88–100. [2]

Lambert, J.P. 1988, in *Numerical Mathematics – Singapore 1988*, ISNM vol. 86, R.P. Agarwal, Y.M. Chow, and S.J. Wilson, eds. (Basel: Birkhaüser), pp. 273–284.

Niederreiter, H. 1988, in *Numerical Integration III*, ISNM vol. 85, H. Brass and G. Hämmerlin, eds. (Basel: Birkhaüser), pp. 157–171.

Sobol', I.M. 1967, *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 88–112. [3]

Antonov, I.A., and Saleev, V.M 1979, *USSR Computational Mathematics and Mathematical Physics*, vol. 19, no. 1, pp. 252–256. [4]

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York, Wiley) [discusses Latin Square].

7.8 深入讨论:自适应及递归蒙特卡罗方法

这一节讨论蒙特卡罗积分的更先进技术。作为应用这些技术的例子,举两个很不相同但很有技巧的多维蒙特卡罗程序:vegas[1,2]与 miser[3]。讨论的技术都属于方差退化的一般程序(参见 7.6 节),但是又很有特色。

7.8.1 重要取样

重要取样的应用已隐含在(7.6.6)式与(7.6.7)式中。现在用比较正规的方法讨论它。设被积函数 f 可以写为两个函数的乘积,一个函数 h 几乎是常数,另一个函数 g 是正值函数,则它对多维体积 V 的积分是

$$\int f\,dV = \int (f/g)\,g dV = \int h\,g dV \tag{7.8.1}$$

根据(7.6.7)式,可以将(7.8.1)式进行变量变换成 G,而 G 是 g 的不定积分。这使 gdV 成为一个全微分。然后,应用蒙特卡罗积分的基本定理——(7.6.1)式进行处理。(7.8.1)式更一般的理解是,积分 f 可以由取样 h 代替,但此时不是均匀概率密度 dV,而是非均匀密度 gdV。前面第一种理解是第二种理解的特殊情形,产生非均匀取样 gdV 的方法就是通过使用不定积分 G 的变换方法(参见 7.2 节)。

更直接地,可以把基本定理(7.6.1)式推广到非均匀取样的情形:设在体积 V 中选取具有概率密度 p 的取样点 x_i,它满足

$$\int p\,dV = 1 \tag{7.8.2}$$

推广后的基本定理是,用 N 个取样点 $x_0,\ldots,x_{N-1}$,任意函数 f 的积分是估算值 I:

$$I \equiv \int f\,dV = \int \frac{f}{p}\,p dV \approx \left\langle \frac{f}{p} \right\rangle \pm \sqrt{\frac{\langle f^2/p^2\rangle - \langle f/p\rangle^2}{N}} \tag{7.8.3}$$

其中角括号表示 N 个点的算术平均值,如(7.6.2)式。式中的正负项是误差估计的标准偏差,如(7.6.1)式。要注意,(7.6.1)式是(7.8.3)式在 $p=$ 常数 $=1/V$ 的特殊情形。

取样密度 p 的最佳选择是什么呢?我们已直观地看到,就是使 $h=f/p$ 尽可能接近常数。可以把注意力集中在(7.8.3)式根号内的分子上,即每个取样点的方差。两个角括号都是积分的蒙特卡罗估计量,所以可以写成

$$S \equiv \left\langle \frac{f^2}{p^2} \right\rangle - \left\langle \frac{f}{p} \right\rangle^2 \approx \int \frac{f^2}{p^2}\,p dV - \left[\int \frac{f}{p}\,p dV\right]^2 = \int \frac{f^2}{p}\,dV - \left[\int f\,dV\right]^2 \tag{7.8.4}$$

现在用函数变分,求(7.8.2)式限制的条件下 p 的最佳值,

$$0=\frac{\delta}{\delta p}\left(\int\frac{f^2}{p}dV-\left[\int f\,dV\right]^2+\lambda\int p\,dV\right) \tag{7.8.5}$$

其中 λ 是拉格朗日乘子，注意中间项并不依赖 p。变分(来自积分内部)给出 $0=-f^2/p^2+\lambda$ 或

$$p=\frac{|f|}{\sqrt{\lambda}}=\frac{|f|}{\int|f|\,dV} \tag{7.8.6}$$

其中的 λ 是受(7.8.2)式限制的。

如果在积分区域 f 不变号，如果有计算取样结果的特殊方法，则可得出明显的结果，p 的最佳值正比于 $|f|$。于是方差值将减小到0. 虽不太直观，但的确是：即使 f 要变号，最佳值也将有 $p\propto|f|$。在这种情形下，每个取样点的方差((7.8.4)式与(7.8.6)式)是

$$S=S_{\text{optimal}}=\left(\int|f|\,dV\right)^2-\left(\int f\,dV\right)^2 \tag{7.8.7}$$

有意思的是，在被积函数中，可以加一个常数而保持它不变号，因为这仅使积分变化一个已知量，常数乘 V。p 的最佳选择总是给出零方差，积分完全精确。对这个看来是似而非的问题(已在7.6节结尾叙述过)的回答是这样的，要确切知道(7.8.6)式中 p，就要确切知道积分 $\int|f|\,dV$，这就等价于确切知道待计算的积分。

如果函数 f 在体积 V 的大部分区域有一已知常数值，一个肯定好的方法是再加一常数使那个常数值为零。这样做大概只是一个粗略不成熟想法。重要取样所能达到的精确度，不取决于(7.8.7)式取多小，而实际上取决于对**可执行的** p，(7.8.4)式取多小。

7.8.2 分层取样

分层取样的思路完全不同于重要抽样。将符号扩充一下，令$\langle\langle f\rangle\rangle$表示函数 f 对体积 V 的真实平均(即积分被 V 除)，$\langle f\rangle$如前一样是上述平均的最简单的(均匀取样的)蒙特卡罗估计：

$$\langle\langle f\rangle\rangle\equiv\frac{1}{V}\int f\,dV\qquad\langle f\rangle\equiv\frac{1}{N}\sum_i f(x_i) \tag{7.8.8}$$

估计值的方差 $\mathrm{Var}(\langle f\rangle)$是蒙特卡罗积分误差平方的量度，它由关系式

$$\mathrm{Var}(\langle f\rangle)=\frac{\mathrm{Var}(f)}{N} \tag{7.8.9}$$

确定(对比(7.6.1)式)，与函数的方差 $\mathrm{Var}(f)\equiv\langle\langle f^2\rangle\rangle-\langle\langle f\rangle\rangle^2$ 有关。

设将体积 V 分割成两个相等的、不相交的子体积 a 和 b，在每一子体积中取 $N/2$ 个样点。不同于(7.8.8)式，$\langle\langle f\rangle\rangle$的估计值$\langle f\rangle'$表示为

$$\langle f\rangle'\equiv\frac{1}{2}\left(\langle f\rangle_a+\langle f\rangle_b\right) \tag{7.8.10}$$

换言之，它是在两个半区域内的取样平均的均值。估计值(7.8.10)的方差是

$$\begin{aligned}\mathrm{Var}\left(\langle f\rangle'\right)&=\frac{1}{4}\left[\mathrm{Var}\left(\langle f\rangle_a\right)+\mathrm{Var}\left(\langle f\rangle_b\right)\right]\\&=\frac{1}{4}\left[\frac{\mathrm{Var}_a(f)}{N/2}+\frac{\mathrm{Var}_b(f)}{N/2}\right]\\&=\frac{1}{2N}\left[\mathrm{Var}_a(f)+\mathrm{Var}_b(f)\right]\end{aligned} \tag{7.8.11}$$

其中 $\mathrm{Var}_a(f)$表示 f 在了体积 a 中的方差$\langle\langle f^2\rangle\rangle_a-\langle\langle f\rangle\rangle_a^2$，对 b 也有对应量。

从已有的定义(在物理上，这个有关组合二阶矩的公式称为"平行移轴定理")不难证明关系式：

$$\mathrm{Var}(f)=\frac{1}{2}\left[\mathrm{Var}_a(f)+\mathrm{Var}_b(f)\right]+\frac{1}{4}\left(\langle\langle f\rangle\rangle_a-\langle\langle f\rangle\rangle_b\right)^2 \tag{7.8.12}$$

比较(7.8.9)式、(7.8.11)式和(7.8.12)式可以看出，分层(分为两个子体积)取样法给出的方差——当$\langle\langle f\rangle\rangle_a$与$\langle\langle f\rangle\rangle_b$不同时，它是较小的，决不比简单蒙特卡罗情形中的大。

我们还没有利用在两个子体积中，取**不同数量**的点数的可能性，如在子体积 a 中取 N_a 点，而在于体积 b 中取 $N_b\equiv N-N_a$ 点。现在就来研究这个问题。估计值的方差为

$$\mathrm{Var}\left(\langle f\rangle'\right)=\frac{1}{4}\left[\frac{\mathrm{Var}_a(f)}{N_a}+\frac{\mathrm{Var}_b(f)}{N-N_a}\right] \tag{7.8.13}$$

很容易证明它在

$$\frac{N_a}{N}=\frac{\sigma_a}{\sigma_a+\sigma_b} \tag{7.8.14}$$

时取极小值。

这里选用了简化标记 $\sigma_a\equiv[\mathrm{Var}_a(f)]^{1/2}$，$b$ 亦类似。如果 N_a 满足(7.8.14)式，则(7.8.13)式简化为

$$\mathrm{Var}\left(\langle f\rangle'\right)=\frac{(\sigma_a+\sigma_b)^2}{4N} \tag{7.8.15}$$

如果 $\mathrm{Var}(f)=\mathrm{Var}_a(f)=\mathrm{Var}_b(f)$，(7.8.15)式简化为(7.8.9)式，在这种情况下分层取样没起作用。

概括归纳上面的标准方法是，将体积 V 分割成多于两个相等的子区域。在各区域间分配取样点数的最佳方案是，令每一区域 j 的点数正比于 σ_j(即在这子区域中函数 f 的方差的平方根)。但在高维空间中(如 $d\gtrsim 4$)，这实际上并不是很有用。沿每一维将体积分割为 K 部分，其意味着有 K^d 个子体积，在着重估计全部对应的 σ_j 时，一般这个数目太大了。

7.8.3 混合策略

初看起来，重要取样与分层取样是互不相容的，前者在被积函数值 $|f|$ 最大的地方集中取样点，而后者集中在 f 的方差最大的地方。为什么二者都可取呢？

回答是，它们取决于你知道什么和你知道多少。重要取样依赖于已经掌握的积分近似值，故能够产生具有所期望的概率密度 p 的随机点 x_i。由于 p 在一定程度上不理想，所以只能使误差仅以 $N^{-1/2}$ 减小。如果在被积函数 f 急剧变化的区域，p 很不理想，情况会很糟，因为取样函数 $h=f/p$ 会产生很大的方差。重要取样是研究光滑取样函数 h 的值，只有满足这些条件才是有效的。

相反，分层取样不需要知道 f 的任何信息。分层取样是研究消除各子区域点数的起伏，而不是研究各点平滑的数值。最简单的分层策略：分割 V 为相等的 N 个子区域，并在每一个子区域中随机地选取一个点，这个方法的误差近似按 N^{-1} 下降，比 $N^{-1/2}$ 快得多(在7.7节中的准随机数是消除点密度起伏的另一个方法，它近似给出与"盲目"分层取样一样好的结果)。

然而，"近似"是一个很重要的提示。例如，假设被积函数除在某一个子区域外都可以忽略，则在这单个子区域内取样积分的结果完全无用。粗略可知，重要取样允许在这子区域中取很多点，这要比盲目分层取样好得多。

分层取样还是有它的优点，只要能估计出方差值，就能按照(7.8.14)式及其推论在不同子区域取不同数目的点，同时能找到将一区域划分为适当数目的子区域(注意，不是多维数 d 的 K^d 个)的方法，每一子区域的函数方差将比总体积中的方差显著减少。要做到这些，需要掌握函数 f 的许多信息，虽然重要取样也需要掌握 f 的各种信息。

实际上，重要取样与分层取样，即使不是在很多方面，但也是在许多重要方面是可兼容的。许多被积函数 f 的值，除了称为"作用区"的小部分体积外，在 V 中处处都是很小。而在这些作用区，$|f|$ 的数值与标准偏差 $\sigma=[\mathrm{Var}(f)]^{1/2}$ 的大小是可比较的，所以，这两个技术将给出大致相同的点密度。在较复杂的执行过程中，可能要把这两种技术"嵌套"起来。例如，基于粗网格上的重要取样后，可以在每一网格中，再用分层取样。

7.8.4 自适应蒙特卡罗：VEGAS

Peter Lepage [1,2]发明的 VEGAS 算法广泛应用于多维积分，这种积分在基本粒子物理学中常用到。VEGAS 基本上采用重要取样，但若维数 d 足够小时，可以避免 K^d 的迅速增长，有时也采用分层取样(特别是如果 $(K/2)^d<N/2$，N 为取样点数)。VEGAS 中重要取样的基本方法是，自适应地构造一个多维加权函数 g，它是**可分解**的，

$$p\propto g(x,y,z,\ldots)=g_x(x)g_y(y)g_z(z)\cdots \tag{7.8.16}$$

这样的函数有两种方法避免 K^d 的迅速增长：(i)可以把作为 d 个分割的一维函数存储在计算机中，每一函数由 K 个列表值定义，即用 $K\times d$ 代替 K^d；(ii)对 d 个一维函数，以概率密度进行连续取样，而得到坐标向量分

量$(x,y,z,\ldots)$。

最佳可分解加权函数可写为[1]

$$g_x(x) \propto \left[\int dy \int dz \ldots \frac{f^2(x,y,z,\ldots)}{g_y(y)g_z(z)\ldots}\right]^{1/2} \tag{7.8.17}$$

(对 $y,z,\ldots$ 也有类似方程。)此式在一维时,退化为 $g \propto |f|$(7.8.6)式。(7.8.17)式显然提示了 VEGAS 的自适应策略:给出一组g函数(如初始可以都取为常数),对函数 f 取样,不仅累积积分所有的估值,还累积(7.8.17)式右边的 Kd 估计值(d 维的每一独立自变量分为 K 段)。由此决定下一迭代过程改进的g函数。

若被积函数 f 局限于 d 维空间的一个或少数几个区域,则当坐标值为这些区域在该坐标轴的投影时,这些加权函数g很快变大。蒙特卡罗积分的精确度将显著提高,超过简单蒙特卡罗的精确度。

VEGAS 的缺点是显而易见的:函数 f 在个别坐标方向投影是均匀的,VEGAS 在这些方向上的取样点不集中。例如,VEGAS 的最坏情况是被积函数集中在全体对角线上,例如,从$(0,0,0,\ldots)$到$(1,1,1,\ldots)$。因为这种几何形状是完全不可分割的,VEGAS 完全体现不出优点。更一般地,若被积函数集中在一维(或更高维)曲线上(或超曲面上),VEGAS 算法也许根本不能使用,除非刚好处于坐标方向。

下面列出的程序 vegas 基本上是 Lepage 的标准版本,为适应我们的需要进行了很少的修改(感谢 Lepage 允许在本书中引用程序)。为了与 VEGAS 算法的其他版本在计算上相协调,保留了原来的变量名称。参数 NDMX 就是我们称为 K 的变量名,是沿每一轴的增量最大值;MXDIM 是 d 的最大值。其他参数待注释中说明。

vegas 程序执行 m = itmx 次对所求积分进行统计独立的计算,每一次具有 N = ncall 个函数计算。它们虽然是统计不相关的,但这些迭代是互相支持的,因为每一次迭代都用来为下一次迭代改进取样网格。所有迭代结果,组合得到最佳答案及其估计误差,如下式:

$$I_{\text{best}} = \sum_{i=0}^{m-1} \frac{I_i}{\sigma_i^2} \bigg/ \sum_{i=0}^{m-1} \frac{1}{\sigma_i^2} \qquad \sigma_{\text{best}} = \left(\sum_{i=0}^{m-1} \frac{1}{\sigma_i^2}\right)^{-1/2} \tag{7.8.18}$$

返回这个参量

$$\chi^2/m \equiv \frac{1}{m-1} \sum_{i=0}^{m-1} \frac{(I_i - I_{\text{best}})^2}{\sigma_i^2} \tag{7.8.19}$$

如果它远远大于 1,则迭代结果是统计不相容的,且答案是值得怀疑的。

输入标记 init 可方便使用。可以调用 init = 0, ncall = 1000, itmx = 5;然后调用 init = 1, ncall = 100 000, itmx = 1。执行结果可形成小取样数的 5 次迭代的取样网格,然后在最佳网格上进行一次高精确度积分。

用户提供的被积函数 fxn 除所期望的求值点 x 外,还有一个自变量 wgt,在多数应用场合可忽略函数内的 wgt。但有时可能要对一些附加函数或主函数 f 相一致的函数进行积分,如某些函数g的积分可以估值

$$I_g = \sum_i w_i g(\mathbf{x}) \tag{7.8.20}$$

其中 w_i 与 $\mathbf{x}$ 分别是自变量 wgt 与 x。在函数 fxn 内部直接累加求和,并通过全程变量将此求和结果返回到用户的主程序。当然,由于取样将对 f 优化,所以$g(\mathbf{x})$最好是与主函数 f 有某种程度的类似。

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include "nr.h"
using namespace std;

extern int idum;                          在主程序中对随机数进行初始化。

void NR::vegas(Vec_I_DP &regn, DP fxn(Vec_I_DP &, const DP), const int init,
    const int ncall, const int itmx, const int nprn, DP &tgral, DP &sd,
    DP &chi2a)
```

此程序实现对用户提供的 ndim 维函数 fxn 在以 regn[0..2*ndim-1]标志的长方体内进行蒙特卡罗积分,这个 regn[0..2*ndim-1]向量由 ndim 的"较低左边"坐标,其后跟随 ndim 的"较高右边"坐标而组成。积分包括 itmx 次迭代,每次近似 ncall 次调用函数。每次迭代后网格被精练;大于 5 或 10 次迭代是相当有用的。输入标志 init 表示这次调用是新的开始,还是为附加迭代的子调用(见下面注释)。输入标志 nprn(通常为 0)是控制诊断量的输出。返回的答案是 tgral(积分的最佳估值),sd(它的标准差)及 chi2a(每自由度的 χ^2,一个是否一致的结果已获得的标记)。详情见后面正文。

```
{
    const int NDMX=50, MXDIM=10;
    const DP ALPH=1.5, TINY=1.0e-30;
    static int i,it,j,k,mds,nd,ndo,ng,npg;
    static DP calls,dv2g,dxg,f,f2,f2b,fb,rc,ti;
    static DP tsi,wgt,xjac,xn,xnd,xo,schi,si,swgt;
    static Vec_INT ia(MXDIM),kg(MXDIM);
    static Vec_DP dt(MXDIM),dx(MXDIM),r(NDMX),x(MXDIM),xin(NDMX);
    static Mat_DP d(NDMX,MXDIM),di(NDMX,MXDIM),xi(MXDIM,NDMX);
    最好设为静态，允许重新开始。

    int ndim=regn.size()/2;
    if (init <= 0) {                    标准输入，在冷启动时输入。
        mds=ndo=1;                      改变到 mds=0使不能分层取样，即仅用重要取样。
        for (j=0;j<ndim;j++) xi[j][0]=1.0;
    }
    if (init <= 1) si=swgt=schi=0.0;
    这里输入以继承前次调用的栅格，但不输入它的结果。
    if (init <= 2) {                    这里输入以继承前次调用的栅格和它的结果。
        nd=NDMX;
        ng=1;
        if (mds != 0) {                 为分层设置循环。
            ng=int(pow(ncall/2.0+0.25,1.0/ndim));
            mds=1;
            if ((2*ng-NDMX) >= 0) {
                mds = -1;
                npg=ng/NDMX+1;
                nd=ng/npg;
                ng=npg*nd;
            }
        }
        for (k=1,i=0;i<ndim;i++) k *= ng;
        npg=MAX(int(ncall/k),2);
        calls=DP(npg)*DP(k);
        dxg=1.0/ng;
        for (dv2g=1,i=0;i<ndim;i++) dv2g *= dxg;
        dv2g=SQR(calls*dv2g)/npg/npg/(npg-1.0);
        xnd=nd;
        dxg *= xnd;
        xjac=1.0/calls;
        for (j=0;j<ndim;j++) {
            dx[j]=regn[j+ndim]-regn[j];
            xjac *= dx[j];
        }
        if (nd != ndo) {                若需要，反复量化分级。
            for (i=0;i<MAX(nd,ndo);i++) r[i]=1.0;
            for (j=0;j<ndim;j++)
                rebin(ndo/xnd,nd,r,xin,xi,j);
            ndo=nd;
        }
        if (nprn >= 0) {
            cout << " Input parameters for vegas";
            cout << " ndim= " << setw(4) << ndim;
            cout << " ncall= " << setw(8) << calls << endl;
            cout << setw(34) << " it=" << setw(5) << it;
            cout << " itmx=" << setw(5) << itmx << endl;
            cout << setw(34) << " nprn=" << setw(5) << nprn;
            cout << " ALPH=" << setw(9) << ALPH << endl;
            cout << setw(34) << " mds=" << setw(5) << mds;
            cout << " nd=" << setw(5) << nd << endl;
            for (j=0;j<ndim;j++) {
                cout << setw(30) << " xl[" << setw(2) << j;
                cout << "]= " << setw(11) << regn[j] << " xu[";
                cout << setw(2) << j << "]= ";
                cout << setw(11) << regn[j+ndim] << endl;
            }
        }
    }
    for (it=0;it<itmx;it++) {
    主要的迭代循环：在此输入(init ≥3)以完成附加的 itmx 次迭代，而所有其他参数不变。
```

```
ti=tsi=0.0;
for (j=0;j<ndim;j++) {
    kg[j]=1;
    for (i=0;i<nd;i++) d[i][j]=di[i][j]=0.0;
}
for (;;) {
    fb=f2b=0.0;
    for (k=0;k<npg;k++) {
        wgt=xjac;
        for (j=0;j<ndim;j++) {
            xn=(kg[j]-ran2(idum))*dxg+1.0;
            ia[j]=MAX(MIN(int(xn),NDMX),1);
            if (ia[j] > 1) {
                xo=xi[j][ia[j]-1]-xi[j][ia[j]-2];
                rc=xi[j][ia[j]-2]+(xn-ia[j])*xo;
            } else {
                xo=xi[j][ia[j]-1];
                rc=(xn-ia[j])*xo;
            }
            x[j]=regn[j]+rc*dx[j];
            wgt *= xo*xnd;
        }
        f=wgt*fxn(x,wgt);
        f2=f*f;
        fb += f;
        f2b += f2;
        for (j=0;j<ndim;j++) {
            di[ia[j]-1][j] += f;
            if (mds >= 0) d[ia[j]-1][j] += f2;
        }
    }
    f2b=sqrt(f2b*npg);
    f2b=(f2b-fb)*(f2b+fb);
    if (f2b <= 0.0) f2b=TINY;
    ti += fb;
    tsi += f2b;
    if (mds < 0) {                使用分层取样.
        for (j=0;j<ndim;j++) d[ia[j]-1][j] += f2b;
    }
    for (k=ndim-1;k>=0;k--) {
        kg[k] %= ng;
        if (++kg[k] != 1) break;
    }
    if (k < 0) break;
}
tsi *= dv2g;                      为迭代计算最后结果.
wgt=1.0/tsi;
si += wgt*ti;
schi += wgt*ti*ti;
swgt += wgt;
tgral=si/swgt;
chi2a=(schi-si*tgral)/(it+0.0001);
if (chi2a < 0.0) chi2a = 0.0;
sd=sqrt(1.0/swgt);
tsi=sqrt(tsi);
if (nprn >= 0) {
    cout << " iteration no. " << setw(3) << (it+1);
    cout << " : integral = " << setw(14) << ti;
    cout << " +/- " << setw(9) << tsi << endl;
    cout << " all iterations: " << " integral =";
    cout << setw(14) << tgral << "+-" << setw(9) << sd;
    cout << " chi**2/IT n =" << setw(9) << chi2a << endl;
    if (nprn != 0) {
        for (j=0;j<ndim;j++) {
            cout << " DATA FOR axis " << setw(2) << j << endl;
            cout << " X delta i X delta i";
            cout << " X deltai" << endl;
            for (i=nprn/2;i<nd;i += nprn+2) {
                cout << setw(8) << xi[j][i] << setw(12) << di[i][j];
                cout << setw(12) << xi[j][i+1] << setw(12) << di[i+1][j];
                cout << setw(12) << xi[j][i+2] << setw(12) << di[i+2][j];
```

```
                    cout << endl;
                }
            }
        }
    }
    for (j=0;j<ndim;j++) {
        xo=d[0][j];
        xn=d[1][j];
        d[0][j]=(xo+xn)/2.0;
        dt[j]=d[0][j];
        for (i=2;i<nd;i++) {
            rc=xo+xn;
            xo=xn;
            xn=d[i][j];
            d[i-1][j] = (rc+xn)/3.0;
            dt[j] += d[i-1][j];
        }
        d[nd-1][j]=(xo+xn)/2.0;
        dt[j] += d[nd-1][j];
    }
    for (j=0;j<ndim;j++) {
        rc=0.0;
        for (i=0;i<nd;i++) {
            if (d[i][j] < TINY) d[i][j]=TINY;
            r[i]=pow((1.0-d[i][j]/dt[j])/
                (log(dt[j])-log(d[i][j])),ALPH);
            rc += r[i];
        }
        rebin(rc/xnd,nd,r,xin,xi,j);
    }
  }
}
```

精练栅格。查阅参考文献以便理解这些步骤的奥妙。为避免快速、不稳定的变化，精练被阻止，并且被指数 ALPH 压缩在范围内。

```
#include "nr.h"

void NR::rebin(const DP rc, const int nd, Vec_I_DP &r, Vec_O_DP &xin,
    Mat_IO_DP &xi, const int j)
vegas 所用的实用程序，把包括为 xi 的第 j 行的密度向量重新分成由向量 r 定义的新向量。
{
    int i,k=0;
    DP dr=0.0,xn=0.0,xo=0.0;

    for (i=0;i<nd-1;i++) {
        while (rc > dr)
            dr += r[(++k)-1];
        if (k > 1) xo=xi[j][k-2];
        xn=xi[j][k-1];
        dr -= rc;
        xin[i]=xn-(xn-xo)*dr/r[k-1];
    }
    for (i=0;i<nd-1;i++) xi[j][i]=xin[i];
    xi[j][nd-1]=1.0;
}
```

7.8.5 递归分层取样

已经看到，分层取样存在的问题是不可避免 K^d 的迅速增长，这显然是笛卡儿坐标 d 维值所固有的。一种称为**递归分层取样**[3]的技术试图解决这个问题，它不是沿所有 d 维，而是一次仅沿一维逐次将体积分半，对起点应用逐次减小的子区域分半法的是方程(7.8.10)与(7.8.13)。

假设对函数 f 定量地进行 N 次计算，并求它在长方体 $R=(\mathbf{x}_a,\mathbf{x}_b)$ 内的中值 $\langle f\rangle$（这样的区域由两个对顶角坐标向量表示）。首先，指定 N 的一小部分 p 来确定 R 中 f 的方差：在 R 中均匀地抽样 pN 个函数值并累加求和，对应于 d 个不同的坐标方向（R 能沿其方向分半），它将给出 d 对不同的方差。换言之，在 pN 个抽样中，在每一个基于 R 的可能的分半区域中估计 $\mathrm{Var}(f)$。

$$R_{ai} \equiv (\mathbf{x}_a, \mathbf{x}_b - \frac{1}{2}\mathbf{e}_i \cdot (\mathbf{x}_b - \mathbf{x}_a)\mathbf{e}_i)$$
$$R_{bi} \equiv (\mathbf{x}_a + \frac{1}{2}\mathbf{e}_i \cdot (\mathbf{x}_b - \mathbf{x}_a)\mathbf{e}_i, \mathbf{x}_b) \tag{7.8.21}$$

其中 $\mathbf{e}_i$ 是第 i 个坐标方向的单位向量，$i = 1, 2, \ldots, d$。

其次，研究方差以找出分半的最佳维数 i。例如，可用(7.8.15)式选择 i，以使在区域 R_{ai} 和 R_{bi} 中估计方差的平方根之和取最小值(实际做法稍有不同)。

再次，限定余下的 $(1-p)N$ 个函数在区域 R_{ai} 和 R_{bi} 之间计算求值。如果用(7.8.15)式选择 i，则就应该按照(7.8.14)式做这样的限定。

为估计 f 的平均值，我们有两个长方体，每一个有自己的函数计算界限。"RSS"算法本身是递归的：为计算每一区域的平均值，返回到(7.8.21)式上面段落中的"首先"部分重新开始(当然，如果此区域的点限定在某数以下，则常用简单蒙特卡罗法而不用连续递归)。

最后，综合这些方法，并用(7.8.10)式和(7.8.11)式的第一行，来估算两个子体积的方差。

这样以最简形式完成了 RSS 算法。在按"执行细节"的一般规程描述某些附加技巧之前，需要简要回顾一下(7.8.13)式～(7.8.15)式，并推导出实际使用这些公式的替代公式。(7.8.13)式右边应用了熟知的标度定律(7.8.9)式两次，一次对 a，一次对 b。如果估值 $\langle f\rangle_a$ 与 $\langle f\rangle_b$ 是用简单蒙特卡罗均匀随机取样点计算的话，这将是对的。然而，事实上这两个平均估计值是用递归法得到的，所以说明(7.8.9)式成立是没有理由的。不如用下式代替(7.8.13)式：

$$\mathrm{Var}\left(\langle f\rangle'\right) = \frac{1}{4}\left[\frac{\mathrm{Var}_a(f)}{N_a^\alpha} + \frac{\mathrm{Var}_b(f)}{(N-N_a)^\alpha}\right] \tag{7.8.22}$$

其中 α 是一个未知常数 ≥ 1($=1$ 对应简单蒙特卡罗)。在这种情况下，经简单推导可以证明当

$$\frac{N_a}{N} = \frac{\mathrm{Var}_a(f)^{1/(1+\alpha)}}{\mathrm{Var}_a(f)^{1/(1+\alpha)} + \mathrm{Var}_b(f)^{1/(1+\alpha)}} \tag{7.8.23}$$

时，$\mathrm{Var}(\langle f\rangle')$ 有极小值，其极小值为

$$\mathrm{Var}\left(\langle f\rangle'\right) \propto \left[\mathrm{Var}_a(f)^{1/(1+\alpha)} + \mathrm{Var}_b(f)^{1/(1+\alpha)}\right]^{1+\alpha} \tag{7.8.24}$$

当 $\alpha = 1$ 时，(7.8.22)式～(7.8.24)式还原成(7.8.13)式～(7.8.15)式。为找出的 α 自相容值，用数值实验发现 $\alpha \approx 2$。也就是当 $\alpha = 2$ 作为限定取样机会时，应用(7.8.23)式递归，观察到的 RSS 算法方差近似按 N^{-2} 变化；而以 α 的其他值代入(7.8.23)式时，情况会更恶劣(然而，对 α 的敏感性并不算很高，但也不知 $\alpha = 2$ 究竟是严格分析的结果，还是仅有的一种有用的直观推断)

到现在为止，miser 的实现与上面描述的算法的主要区别在于(7.8.23)式右边的方差如何估值。从实验中发现，以取样函数极大值与极小值之差的平方代替纯粹的取样二阶矩是更健壮的。当然，估计值随着取样量的增加也增加，而(7.8.23)式用它仅仅是比较近似相等取样数的两个子体积(a 和 b)。当在被积函数作用区中，原始取样仅给出一个点或少数几个点时，"极大减极小"估计值证明了它的价值。很多现实情况，取样点处在更重要的作用区附近，这使"极大减极小"提供的较大取样加权更有益。

包含在程序中的第二项改进是引入"颤动参数"dith，它的非零值使子体积不在中间精确划分，而是在小数 $0.5 \pm$ dith 处划分，其中符号 $\pm$ 是指引入随机数程序内的随机选择。通常可令 dith 为零。然而，如果被积函数有某些特殊的对称性，使作用区正好处在区域的中点，或区域二次幂因数的中心，则使 dith 不为零将大有好处。要避免这样的极端情形，作用区被划分为 d 维空间紧密相邻的 2^d 个小块。在这种情况 dith 非零的典型值取 0.1 是有益的。当然，在颤动参数为非零时，必须考虑子体积的不同大小，程序通过变量 fracl 来实现此操作。

该程序还有最后一个特性要阐明，RSS 算法使用单组取样点在所有 d 个方向计算(7.8.23)式，在递归底层，取样点数可能非常少。虽然稀少，但有时也会出现在某一方向的所有取样点只在体积的某一半中。在这种情况下，此方向宁可被忽略。更少见的情况是，在所有方向上，所有取样点都只处在总体积的一半中，在这种情况下，可选择一个随机的方向。如果这种情况在工作中经常发生，就应该增加 MNPT(见程序中的 if(jh = = −1)… 行)。

正如已给出的 miser 返回 ave 作为平均函数值 $\langle f\rangle$ 的一个估计值，不是 f 在整个范围内的积分。选用其

他约定,程序 vegas 返回 tgral 作为这个积分。由于长方体体积 V 是已知的,这两个约定自然用(7.8.8)式表示成一般关系式。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::miser(DP func(Vec_I_DP &), Vec_I_DP &regn, const int npts,
    const DP dith, DP &ave, DP &var)
```

此程序实现在以 regn[0..2*ndim-1]指定的长方体内,对用户提供的 ndim 维函数 func 进行蒙特卡罗取样,这个 regn[0..2*ndim-1]向量由 ndim 的"较低左边"坐标,其后跟随 ndim 的"较高右边"坐标组成。函数全部取样 npts 次,其位置由递归分层取样确定。在这范围内函数的均值是以 ave 返回,ave 的统计不确定性的估计值(标准偏离的平方)以 var 返回。输入参数 dith 按常规设置为零,但若函数的作用区域落入以 2 的幂次方划分此区域的边界,则应将 dith 设置为 0.1。

```
{
const int MNPT=15, MNBS=60;
const DP PFAC=0.1, TINY=1.0e-30, BIG=1.0e30;
```

其中 PFAC 是剩余函数计算的一部分,用在每一步来确定函数的方差。至少 MNPT 函数的计算在任何子区域末端完成,仅当 MNBS 函数的计算是可行的,子区域才将进一步被二分。我们取MNBS = 4*MNPT。

```
static int iran=0;
int j,jb,n,ndim,npre,nptl,nptr;
DP aval,varl,fracl,fval,rgl,rgm,rgr,s,sigl,siglb,sigr,sigrb;
DP sum,sumb,summ,summ2;

ndim=regn.size()/2;
Vec_DP pt(ndim);
if (npts < MNBS) {                          太少的点进行二分、直接进行蒙特卡罗。
    summ=summ2=0.0;
    for (n=0;n<npts;n++) {
        ranpt(pt,regn);
        fval=func(pt);
        summ += fval;
        summ2 += fval * fval;
    }
    ave=summ/npts;
    var=MAX(TINY,(summ2-summ*summ/npts)/(npts*npts));
} else {                                    进行原(均匀)取样.
    Vec_DP rmid(ndim);
    npre=MAX(int(npts*PFAC),int(MNPT));
    Vec_DP fmaxl(ndim),fmaxr(ndim),fminl(ndim),fminr(ndim);
    for (j=0;j<ndim;j++) {                  对每维初始化左、右边界。
        iran=(iran*2661+36979) % 175000;
        s=SIGN(dith,DP(iran-87500));
        rmid[j]=(0.5+s)*regn[j]+(0.5-s)*regn[ndim+j];
        fminl[j]=fminr[j]=BIG;
        fmaxl[j]=fmaxr[j]=(-BIG);
    }
    for (n=0;n<npre;n++) {                  在取样点上循环。
        ranpt(pt,regn);
        fval=func(pt);
        for (j=0;j<ndim;j++) {              找出每维的左、右边界。
            if (pt[j]<=rmid[j]) {
                fminl[j]=MIN(fminl[j],fval);
                fmaxl[j]=MAX(fmaxl[j],fval);
            } else {
                fminr[j]=MIN(fminr[j],fval);
                fmaxr[j]=MAX(fmaxr[j],fval);
            }
        }
    }
    sumb=BIG;                               选取哪维 jb 进行二分。
    jb= -1;
    siglb=sigrb=1.0;
    for (j=0;j<ndim;j++) {
        if (fmaxl[j] > fminl[j] && fmaxr[j] > fminr[j]) {
            sigl=MAX(TINY,pow(fmaxl[j]-fminl[j],2.0/3.0));
            sigr=MAX(TINY,pow(fmaxr[j]-fminr[j],2.0/3.0));
            sum=sigl+sigr;                  (7.8.24)式,见正文。
            if (sum<=sumb) {
                sumb=sum;
```

```
                jb=j;
                siglb=sigl;
                sigrb=sigr;
            }
        }
    }
    if (jb == -1) jb=(ndim*iran)/175000;   MNPT可以很小。
        rgl=regn[jb];                          在左和右之间按比例分配其余取样点。
        rgm=rmid[jb];
        rgr=regn[ndim+jb];
        fracl=fabs((rgm-rgl)/(rgr-rgl));
        nptl=int(MNPT+(npts-npre-2*MNPT)*fracl*siglb
            /(fracl*siglb+(1.0-fracl)*sigrb));          (7.8.23)式。
        nptr=npts-npre-nptl;
        Vec_DP regn_temp(2*ndim);              现在分配和积分这两个子区域。
        for (j=0;j<ndim;j++) {
            regn_temp[j]=regn[j];
            regn_temp[ndim+j]=regn[ndim+j];
        }
        regn_temp[ndim+jb]=rmid[jb];
        miser(func,regn_temp,nptl,dith,avel,varl);
        regn_temp[jb]=rmid[jb];                递归调用，这里将最终返回。
        regn_temp[ndim+jb]=regn[ndim+jb];
        miser(func,regn_temp,nptr,dith,ave,var);
        ave=fracl*avel+(1-fracl)*ave;
        var=fracl*fracl*varl+(1-fracl)*(1-fracl)*var;
        Combine left and right regions by equation (7.8.11) (1st line).
    }
}
```

程序 miser 调用一个短函数 ranpt，用在一特定的 d 维区域内以得到随机点。下述的改进程序 ranpt，采用连续调用得到一致随机数生成程序，并做出明确的注释。在这里，通过准随机程序 sobseq（参见 7.7 节），可以很容易地修改 ranpt 产生随机点。其实，具有 sobseq 的 miser 程序比具有一致随机偏差的 miser 更精确。然而，由于 RSS 的使用与准随机数的使用是完全独立的，所以没有使在这里给出的程序依赖 sobseq。对重要取样也可给出类似的说明，它在原则上能与 RSS 混合使用（尽管编程可能极为复杂，但原则上重要取样也可与 vegas 及 miser 混合使用）。

```
#include "nr.h"

extern int idum;

void NR::ranpt(Vec_O_DP &pt, Vec_I_DP &regn)
在n维长方形范围中返回均匀随机点pt。它用于miser，为了一致偏离调用程序ran1。用户的主程序中应该初始化全程
变量idum，使其为负的种子整数。
{
    int j;

    int n=pt.size();
    for (j=0;j<n;j++)
        pt[j]=regn[j]+(regn[n+j]-regn[j])*ran1(idum);
}
```

参考文献和深入阅读

Hammersley, J.M. and Handscomb, D.C. 1964, *Monte Carlo Methods* (London: Methuen).

Kalos, M.H. and Whitlock, P.A. 1986, *Monte Carlo Methods* (New York: Wiley).

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation*, 2nd ed. (New York: Springer-Verlag).

Lepage, G.P. 1978, *Journal of Computational Physics*, vol. 27, pp. 192–203. [1]

Lepage, G.P. 1980, "VEGAS: An Adaptive Multidimensional Integration Program," Publication CLNS-80/447, Cornell University. [2]

Press, W.H., and Farrar, G.R. 1990, *Computers in Physics*, vol. 4, pp. 190–195. [3]

第8章 排　　序

8.0 引言

本章的内容在其他关于数值算法的书中讨论的较少，然而对于任何优秀的程序员来说，排序技术是应具有的必不可少的实用知识。因此，不要忽略了排序这一类基础课题。

在处理数据(实验数据或数值生成的数据)时，常常需要进行排序。需要有数据表格或列表，来表示一个或多个独立(或者"控制")的变量，以及一个或多个应变量(或者"所测量"的量)。在各种不同情况下，可能希望按这些变量中一个或另一个的顺序来排列这些数据。或者，有时可能只希望知道这些数值表中的"中间值"或"四分位"值，后者的任务称为**挑选**，它是与排序紧密相关的。

更明确地说，本章将处理以下一些内容：

- 排序，即按数值的顺序重排数组。
- 按数值顺序重排一个数组的同时，对另外一个或几个数组执行相应的重排，而使所有数组中元素之间的对应关系保持不变。
- 给定一数组，准备一个**索引表**，即指针表，它指明哪一个数组元素按数序排第一，哪一个排第二，等等。
- 给定一数组，准备一个**秩表**，它能指明第一个数组元素的数值秩是什么，第二个数组元素的数值秩是什么，等等。
- 从数组中选取第 M 大的元素。

对于 N 个元素排序的基本任务来说，最佳算法所需的运算量为 $N \log_2 N$ 的几倍。算法发明者试图将这一估计或前面的常数变得尽可能地小。两个最佳算法是由 C.A.R. Hoare 提出的**快速排序法**(参见 8.2 节)，以及由 J.W.J.Williams 提出的**堆积排序法**(参见 8.3 节)。

对于大数 N(如 $N > 1000$)，在大多数机器上，快速排序法要快 1.5 或 2 倍。然而，它需要增加一点额外的存储量，而且是一个较复杂的程序。堆积排序法是一个真正的"同址排序"，因而它编程更加紧凑，更容易略加修改而适应特殊的目的。均衡考虑，推荐使用快速排序法，但仍将给出这两种算法的实施程序。

对于小数 N，如果前面的常数足够小的话，最好选用运算量是 N 的较高阶幂次的算法。对于 $N < 20$，**直接插入法**(参见 8.1 节)是简洁和足够快的。我们将它包括进来有一些担心：它是一个 N^2 的算法，其潜在的误用可能性(将它用于太大的 N)很大。它所耗用的机时是可观的，以至于我们不打算将任何 N^2 算法包括进来。然而，将把界线划在"气泡排序法"这一低效的 N^2 算法处，虽然它是基本计算机科学书中所喜爱的方法。

对于 $N < 50$，Shell 方法(参见 8.1 节)只比直接插入法稍复杂，与更复杂的快速排序法相比更有竞争力。这种方法在最坏情况下是 $N^{3/2}$ 运算量，但通常更快。

想要了解排序领域的进一步知识和详细内容，参见本节参考文献[1,2]。

参考文献和深入阅读：

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley). [1]

Sedgewick, R. 1998, *Algorithms in C*, 3rd ed. (Reading, MA: Addison-Wesley), Chapters 8–13. [2]

8.1　直接插入法和 Shell 方法

直接插入法是一种 N^2 运算量的方法，因而只能用在 N 较小时，比如 $N<20$。

这种技巧完全是有经验的玩扑克者洗牌时对牌排序所用的方法：挑出第二张牌将它按与第一张的顺序插入，然后挑出第三张将它按大小顺序插到前两张中，如此下去直到将最后一张拿起并插入。

```
#include "nr.h"

void NR::piksrt(Vec_IO_DP &arr)
用直接插入法将数组 arr[0..n-1]按递增顺序排列，输出时 arr 被排序好的序列替换。
{
    int i,j;
    DP a;

    int n=arr.size();
    for (j=1;j<n;j++) {                     依次挑出每个元素。
        a=arr[j];
        i=j;
        while (i > 0 && arr[i-1] > a) {     寻找插入的元素。
            arr[i]=arr[i-1];
            i--;
        }
        arr[i]=a;                           将其插入。
    }
}
```

如果想在排序 arr 的同时重排一个数组 brr，可简单地在移动 arr 的一个元素的同时，移动一个 brr 的元素。

```
#include "nr.h"

void NR::piksr2(Vec_IO_DP &arr, Vec_IO_DP &brr)
用直接插入法将序列 arr[0..n-1]按递增顺序排列，同时对序列的 brr[0..n-1]进行相应的调整。
{
    int i,j;
    DP a,b;

    int n=arr.size();
    for (j=1;j<n;j++) {                     依次挑出每个元素。
        a=arr[j];
        b=brr[j];
        i=j;
        while (i > 0 && arr[i-1] > a) {     寻找插入的位置。
            arr[i]=arr[i-1];
            brr[i]=brr[i-1];
            i--;
        }
        arr[i]=a;                           将其插入。
        brr[i]=b;
    }
}
```

对于在排序一个数组的同时有很多数组需要重排的情况，参见 8.4 节。

8.1.1　Shell 方法

这实际上是直接插入法的一个变形，但确实是一种非常强有力的变形。大致的思想是，例如对排序16个数 $n_0,\ldots,n_{15}$ 的情况，是如下进行的：第一次排序，用直接插入法依次排序 8 组数 $(n_0,n_8),(n_1,n_9),\ldots,(n_7,n_{15})$。然后依此排序 4 组数 $(n_0,n_4,n_8,n_{12}),\ldots,(n_3,n_7,n_{11},n_{15})$。之后依次排序两个各有 8 个数的数组，前一个是 $(n_0,n_2,n_4,n_6,n_8,n_{10},n_{12},n_{14})$。最后，将全部 16 个数排序。

当然，只有**最后一次**排序是将数按顺序排列所**必需**的。那么，前面的部分排序是什么目的呢？答案是，前面的排序使数字有效地上下移动，以接近它们的最终位置。因而，最后一次排序实施直接插入法时，实际只需移动少数几个元素。

每次数据排序中待排数组元素之间的间隔(上例中为 8,4,2,1)称为**增量**，所以有时 Shell 方法又称为**减**

小增量排序法。对于如何选择一组好的增量已有了大量的研究,但最佳选择还未找到。实际上,...,8,4,2,1这组并非是好的选择,特别是对于 N 是2的幂次时。一个较好的选择是序列

$$(3^k-1)/2,\ldots,40,13,4,1 \tag{8.1.1}$$

它由递推产生:

$$i_0=1,\qquad i_{k+1}=3i_k+1,\quad k=0,1,\ldots \tag{8.1.2}$$

可以证明(参见本节参考文献[1]),采用这一序列的增量,对于最坏可能顺序的原始数据,上述过程所需的总运算量是 $N^{3/2}$ 量级。因此,对于"随机"顺序的数据,至少对于 $N<60\,000$,其运算量近似为 $N^{1.25}$。对于 $N>50$,快速排序法一般是更快的。程序如下:

```
#include "nr.h"

void NR::shell(const int m, Vec_IO_DP &a)
用 Shell 方法(减小增量排序)将数组 a[0..n-1]按递增顺序排列。输出时 a 被排序好的数组替换。通常,变量 m 应保存
着数组 a 的长度 n,但如果 m < a.size(),那说明 a 中的前 m 个元素已排序。此特性在 selip 中使用。
{
    int i,j,inc;
    DP v;

    inc=1;                          确定初始增量。
    do {
        inc *= 3;
        inc++;
    } while (inc <= m);
    do {                            部分排序循环。
        inc /= 3;
        for (i=inc;i<m;i++) {       直接插入的外循环。
            v=a[i];
            j=i;
            while (a[j-inc] > v) {  直接插入的内循环。
                a[j]=a[j-inc];
                j -= inc;
                if (j < inc) break;
            }
            a[j]=v;
        }
    } while (inc > 1);
}
```

参考文献和深入阅读

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.1. [1]

Sedgewick, R. 1998, *Algorithms in C*, 3rd ed. (Reading, MA: Addison-Wesley), Chapter 8.

8.2 快速排序法

在大多数机器上,平均来说,对于大 N,快速排序法是一种最快的排序算法。这是一种"划分-交换"的排序方法:从待排序列中选取一个"划分元素"a,然后经过成对地交换元素,原先的序列被划分成两个子数组。在一次划分结束之后,元素 a 已经处于它在的数组中的最终位置。在它左面的子数组的元素全都 $\leq$ a,而右边子数组的元素全都 $\geq$ a。进面,在左右子序列中分别继续进行这种划分,如此下去。

划分的过程首先得选出一个元素,比如说最左边的元素,将其作为划分元素 a。用一个指针从上向下扫过数组直到发现一个 > a 的元素,然后用另一个指针从数组尾部向上扫过数组直到发现一个 < a 的元素。这两个元素显然不在最后划分完之后它们应该处的位置上,于是交换它们。继续这一过程直到两指针交汇。交汇处便是应插入 a 的位置,这一轮划分便结束了。当一个元素与划分元素相等时应该怎么办?这个问题是很微妙的,建议参阅 Sedgewick[1]中的讨论(答案:应该停下并进行一次交换)。

为提高运行速度,在实施快速排序法时不使用递归。因而此算法需要有辅助的存储数组,其长度为 $2\log_2 N$,它被用来当做一个下压堆栈来保存那些暂时搁置的子数组。当一个子数组已经被缩减到只有 M 个元素时,用直接插入法会变得更快些(参见8.1节),因而我们也是这样做的。M 值的最佳设置与机器有关,

但$M=7$总不会太差。有些人喜欢将短的子数组暂不排序，留到最后进行一次大的插入排序。因为每个元素最多移动 7 次，这与立刻进行排序的效率一样，并且节约总开销。但是，在采用页面存储的现代机器上，一次整个处理一个大数组会增大开销。而且最后进行插入排序也没有什么好处。

如前所述，快速排序法的平均运行时间是很快的，但它的**最坏情况**的运行时间可以很慢；实际上对于最坏情况它是一个 N^2 方法。对于最直接地应用快速排序法来说，最坏情况实际上就是输入数组已经基本有序的情况。这种输入数组的有序性很容易在实际情况中出现。避免这种情况的方法是，采用一个小的随机数生成程序来挑选一个随机元素作为划分元素。另一种方法是，采用当前子数组的第一、中间和最后元素的中间值来替换划分元素。

快速排序法的高速来源于内循环的简单和有效。仅加上一项不必要的检测（例如，一项保证指针不移出数组的检测）就会使运行时间加倍。为了避免这种不必要的检测，在子数组的两端设置"限制"。最左边的限制≤a，最右边的限制≥a。若采用"三数中值法"来挑选划分元素，则可以运用那两个不是中值的元素作为这一子序列的限制。

下面是我们的实现程序（参见本节参考文献[1]）：

```
#include "nr.h"

void NR::sort(Vec_IO_DP &arr)
用快速排序法使数组 arr[0..n-1]按递增排列，arr 在输出时由已排序的数组替换。
其中 M 是使用直接插入法排序的子数组的大小，NSTACK 是所需的辅助存储器。
{
    const int M=7,NSTACK=50;
    其中 M 是使用直接插入法排序的子数组的大小，NSTACK 是所需的辅助存储器。
    int i,ir,j,k,jstack=-1,l=0;
    DP a;
    Vec_INT istack(NSTACK);

    int n=arr.size();
    ir=n-1;
    for (;;) {                          当子数组足够小时插入排序。
        if (ir-l < M) {
            for (j=l+1;j<=ir;j++) {
                a=arr[j];
                for (i=j-1;i>=l;i--) {
                    if (arr[i] <= a) break;
                    arr[i+1]=arr[i];
                }
                arr[i+1]=a;
            }
            if (jstack < 0) break;
            ir=istack[jstack--];        弹出堆栈并开始新一轮划分。
            l=istack[jstack--];
        } else {
            k=(l+ir) >> 1;              挑选左、中、右三元素的中值作为划分元素a，
            SWAP(arr[k],arr[l+1]);      同时重新排列以使a[l]≤a[l+1]≤a[ir]。
            if (arr[l] > arr[ir]) {
                SWAP(arr[l],arr[ir]);
            }
            if (arr[l+1] > arr[ir]) {
                SWAP(arr[l+1],arr[ir]);
            }
            if (arr[l] > arr[l+1]) {
                SWAP(arr[l],arr[l+1]);
            }
            i=l+1;                      为划分初始化指针。
            j=ir;
            a=arr[l+1];                 划分元素。
            for (;;) {                  开始内循环。
                do i++; while (arr[i] < a);     从数组首部向下扫描寻找元素>a。
                do j--; while (arr[j] > a);     从数组尾部向上扫描寻找元素<a。
                if (j < i) break;       指针交汇，划分结束。
                SWAP(arr[i],arr[j]);    交换元素。
            }                           结束内循环。
            arr[l+1]=arr[j];            插入划分元素。
            arr[j]=a;
            jstack += 2;
```

```
                将指针压入堆栈中更大的子数组,立刻处理较小的子数组。
                if (jstack >= NSTACK) nrerror("NSTACK too small in sort.");
                if (ir-i+1 >= j-l) {
                    istack[jstack]=ir;
                    istack[jstack-1]=i;
                    ir=j-1;
                } else {
                    istack[jstack]=j-1;
                    istack[jstack-1]=l;
                    l=i;
                }
            }
        }
    }
```

可以在排序 arr 的同时,重排其他任何数组,程序基本是前一程序的重复。

```
#include "nr.h"

void NR::sort2(Vec_IO_DP &arr, Vec_IO_DP &brr)
用快速排序法使数组 arr[0..n-1]按递增排列,而同时相应地重排数组 brr[0..n-1]。
{
    const int M=7,NSTACK=50;
    int i,ir,j,k,jstack=-1,l=0;
    DP a,b;
    Vec_INT istack(NSTACK);

    int n=arr.size();
    ir=n-1;
    for (;;) {                          当子数组足够小时插入排序。
        if (ir-l < M) {
            for (j=l+1;j<=ir;j++) {
                a=arr[j];
                b=brr[j];
                for (i=j-1;i>=l;i--) {
                    if (arr[i] <= a) break;
                    arr[i+1]=arr[i];
                    brr[i+1]=brr[i];
                }
                arr[i+1]=a;
                brr[i+1]=b;
            }
            if (jstack < 0) break;
            ir=istack[jstack--];        弹出堆栈并开始新一轮划分。
            l=istack[jstack--];
        } else {
            k=(l+ir) >> 1;              挑选左、中、右三元素的中值作为划分元素a,
            SWAP(arr[k],arr[l+1]);      同时重新排列以使a[l]≤a[l+1]≤a[ir]。
            SWAP(brr[k],brr[l+1]);
            if (arr[l] > arr[ir]) {
                SWAP(arr[l],arr[ir]);
                SWAP(brr[l],brr[ir]);
            }
            if (arr[l+1] > arr[ir]) {
                SWAP(arr[l+1],arr[ir]);
                SWAP(brr[l+1],brr[ir]);
            }
            if (arr[l] > arr[l+1]) {
                SWAP(arr[l],arr[l+1]);
                SWAP(brr[l],brr[l+1]);
            }
            i=l+1;                      为划分初始化指针。
            j=ir;
            a=arr[l+1];                 划分元素。
            b=brr[l+1];
            for (;;) {                  开始内循环。
                do i++; while (arr[i] < a);     从数组首部向下扫描寻找元素>a。
                do j--; while (arr[j] > a);     从数组尾部向上扫描寻找元素<a。
                if (j < i) break;       指针交汇,划分结束。
                SWAP(arr[i],arr[j]);    交换两个数组的元素。
                SWAP(brr[i],brr[j]);
            }                           结束内循环。
```

```
                arr[l+1]=arr[j];                    在两个数组中插入划分元素。
                arr[j]=a;
                brr[l+1]=brr[j];
                brr[j]=b;
                jstack += 2;
                将指针压入堆栈中更大的子数组,立刻处理较小的子数组。
                if (jstack >= NSTACK) nrerror("NSTACK too small in sort2.");
                if (ir-i+1 >= j-l) {
                    istack[jstack]=ir;
                    istack[jstack-1]=i;
                    ir=j-1;
                } else {
                    istack[jstack]=j-1;
                    istack[jstack-1]=l;
                    l=i;
                }
            }
        }
    }
```

从原则上说,可以同 brr 一起重排任何数目的附加数组,但在数组的数目增多时,这样做是很浪费的。有效的方法是,使用 8.4 节中讨论的索引表的方法。

参考文献和深入阅读

Sedgewick, R. 1978, *Communications of the ACM*, vol. 21, pp. 847–857. [1]

8.3　堆排序法

通常,堆排序法不如快速排序法快,但它是被广泛使用的一种排序法。它是真正的"同址"排序,不要求附加的存储单元。它是一个 $N \log_2 N$ 次的过程,这不仅从平均上来说,而且对于输入数据是最坏情况的次序,它也是如此。事实上,最坏情况下只比平均运行时间多 20% 左右。

对于堆排序法给出完整的理论说明已超出了本书讨论的范围。现在,只介绍它的基本原理,如果读者想了解细节可参见本节参考文献[1,2],或者自己分析程序。

一系列 N 个数 a_j, $j=0,\ldots,N-1$,如果满足下列关系就说它形成一个"堆":

$$a_{(j-1)/2} \geqslant a_j \quad 对于 \quad 0 \leqslant (j-1)/2 < j < N \tag{8.3.1}$$

其中的除法 $j/2$ 是指"整数除法",也就是说是一个确切的整数或是略去小数位的最近整数。定义(8.3.1)式是有意义的,如果将数字 a_i 看成是按一个二叉树排列的,在最上面的是"上级"结点 a_0,两个"下属"结点是 a_1 和 a_2,它们的四个下属结点是 a_3 到 a_6,等等(参见图 8.3.1)。按这种形式,这一等级制度从上而下,每个"管理者"大于或等于它的两个"被管理者"。

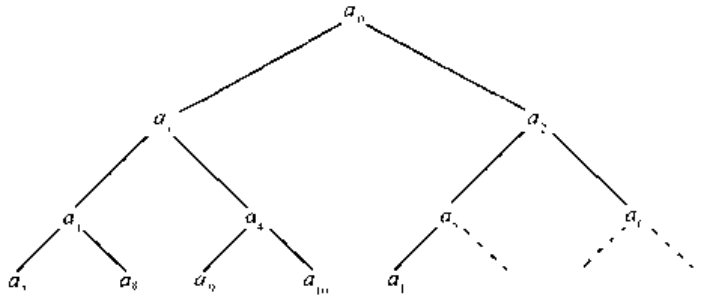


图 8.3.1　12 个元素的"堆"所隐含的顺序。由向上的通路所连接的元素是相对排序的,但"横向"相邻的元素却没有必然的大小关系

如果已经将数组设法重整成堆的形式,那么将它排序就很简单了:将"堆顶"除去,即使不排序也知道它是最大的。然后将它的最大下属元素"提升"到堆顶。然后再提升它的最大下属元素,如此继续。这一过程

类似在大公司中总裁退休时会发生的情况,然后可以使新的总裁退休而重复上述过程。最终整个事件是一个$N \log_2 N$过程,因为每个总裁退休将导至$\log_2 N$次下属的提升。

但开始时如何将数组整理成一个堆呢?答案仍然类似公司中提升的"上移"过程。假想公司开始时有$N/2$个在生产线上的雇员,但没有管理人员。于是一个管理人员被雇来管理两个工人。如果他没有那两个工人中的任何一个有能力,则工人中的一个被提升为管理员,而原管理员降级到生产线上。雇完管理员之后,就开始雇管理人员的管理者,依次类推,直到公司阶梯的最高层。每个雇员从这棵树的顶部被带进来,然后立刻向下筛选,更有能力的工人得到提升直到停在他合适的阶层上。

在应用堆排序法时,同样的"上移"代码段可以应用在开始产生堆时,也可以用在随后的退休和提升过程中。堆排序法的一个完整实现,代表了一个大公司的整个生存循环:雇用$N/2$个工人;雇用$N/2$个潜在的管理者;有一个职位的升迁过程,它是一种超级Peter原理的排序;最后,按注定的过程,每个原来的雇员被依次提升为总裁。

```
#include "nr.h"

namespace {
    void sift_down(Vec_IO_DP &ra, const int l, const int r)
    Carry out the sift-down on element ra(l) to maintain the heap structure. l and r determine the
    "left" and "right" range of the sift-down.
    {
        int j,jold;
        DP a;

        a=ra[l];
        jold=l;
        j=l+1;
        while (j <= r) {
           if (j < r && ra[j] < ra[j+1]) j++      与下属中较大的比较。
           if (a >= ra[j]) break;                 找到a所在的层, 则停止筛选. 否
           ra[jold]=ra[j];                        则, 更新a, 继续。
           jold=j;
           j=2*j+1;
        }
        ra[jold]=a;                               将a归位。
    }
}

void NR::hpsort(Vec_IO_DP &ra)
用堆排序法将数组ra[0..n-1]按递增顺序排列。输出时ra被排序好的数组替换。
{
    int i;

    int n=ra.size();
    for (i=n/2-1; i>=0; i--)
        Here the "right" range of the sift-down is decrmented from n-2 down to 0 during the "retire-
        ment-and-promotion"(heap selection)phase.
        sift_down(ra,i,n-1);
    for (i=n-1; i>0; i--) {
        在"退休-和-提升"(堆选择)的过程中,"适当"的筛选范围从n-2减少到0.
        SWAP(ra[0],ra[i]);              在数组尾部清出一个空间, 使堆顶退休并存入
        sift_down(ra,0,i-1);            其中.
    }
}
```

参考文献和深入阅读

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.3. [1]

Sedgewick, R. 1998, *Algorithms in C*, 3rd ed. (Reading, MA: Addison-Wesley), Chapter 11. [2]

8.4 索引和分秩

在数据文件管理中,**关键**这个概念起着重要的作用。在数据文件中一条**数据记录**可包含若干项,或称为**域**。

例如，天气观察文件中的一条记录可以有记录的时间、温度和风速的域。当排序记录时，必须决定哪个域要顺序排列，记录中其他域只是跟着移动。这些域通常在最后不形成某种特殊的顺序。这个执行排序的域称为**关键域**。

对于一个有很多记录和很多域的数据文件，要将 N 条记录按照它们的关键数组 K_i，$i=0,\ldots,N-1$ 排列成序，其工作量很大，实现很困难。但是，可以代之构造**索引表** I_j，$j=0,\ldots,N-1$，使得最小的 K_i 有 $i=I_0$，第二小的有 $i=I_1$，依次类推，直至对于最大的 K_i 有 $i=I_{N-1}$。换言之，数组

$$K_{I_j} \quad j=0,1,\ldots,N-1 \tag{8.4.1}$$

是以 j 为索引的顺序排列。当有一索引表后，就不需要将记录从它原来的顺序中移动。进一步地，同样的一组记录可以产生不同的索引表，这是根据不同的关键域产生的。

产生一个索引表的算法是很直接的：将索引数组初始化成整数 0 到 $N-1$，之后用快速排序法，像排序关键数组一样移动其元素。于是初始时最小的关键数组所对应的序号最终停在第一位，依次类推。

我们用两种专门的实现重载了 indexx 函数，一种适合 double 值，另一种适合整数值。

```
#include "nr.h"

void NR::indexx(Vec_I_DP &arr, Vec_O_INT &indx)
索引一个数组 arr[0..n-1]，也就是输出一个数组 indx[0..n-1]使得 arr[indx[j]]，对于 j=0,1,...,N-1 是递增的。输入
量 arr 不改变。
{
    const int M=7,NSTACK=50;
    int i,indxt,ir,j,k,jstack=-1,l=0;
    DP a;
    Vec_INT istack(NSTACK);

    int n=arr.size();
    ir=n-1;
    for (j=0;j<n;j++) indx[j]=j;
    for (;;) {
        if (ir-l < M) {
            for (j=l+1;j<=ir;j++) {
                indxt=indx[j];
                a=arr[indxt];
                for (i=j-1;i>=l;i--) {
                    if (arr[indx[i]] <= a) break;
                    indx[i+1]=indx[i];
                }
                indx[i+1]=indxt;
            }
            if (jstack < 0) break;
            ir=istack[jstack--];
            l=istack[jstack--];
        } else {
            k=(l+ir) >> 1;
            SWAP(indx[k],indx[l+1]);
            if (arr[indx[l]] > arr[indx[ir]]) {
                SWAP(indx[l],indx[ir]);
            }
            if (arr[indx[l+1]] > arr[indx[ir]]) {
                SWAP(indx[l+1],indx[ir]);
            }
            if (arr[indx[l]] > arr[indx[l+1]]) {
                SWAP(indx[l],indx[l+1]);
            }
            i=l+1;
            j=ir;
            indxt=indx[l+1];
            a=arr[indxt];
            for (;;) {
                do i++; while (arr[indx[i]] < a);
                do j--; while (arr[indx[j]] > a);
                if (j < i) break;
                SWAP(indx[i],indx[j]);
            }
            indx[l+1]=indx[j];
            indx[j]=indxt;
            jstack += 2;
            if (jstack >= NSTACK) nrerror("NSTACK too small in indexx.");
```

```
            if (ir-i+1 >= j-1) {
                istack[jstack]=ir;
                istack[jstack-1]=i;
                ir=j-1;
            } else {
                istack[jstack]=j-1;
                istack[jstack-1]=l;
                l=i;
            }
        }
    }
}

void NR::indexx(Vec_I_INT &arr, Vec_O_INT &indx)
针对整数数组的重载版本。
{
    const int M=7,NSTACK=50;
    int i,indxt,ir,j,k,jstack=-1,l=0;
    int a;
    余下的代码与 double 版本相同,在此省略。
```

如果想要在排序一个序列的同时,将几个或很多其他数组进行相应的调整,则应该首先产生一个索引表,然后利用它依次重整每个数组。这就需要两个工作空间:一个放置索引,一个用来进行暂时的移动,然后将排好顺序的数组存回原数组中。对于三个数组的情况,程序如下:

```
#include "nr.h"

void NR::sort3(Vec_IO_DP &ra, Vec_IO_DP &rb, Vec_IO_DP &rc)
将数组 ra[0..n-1] 排序成递增顺序,同时将数组 rb[0..n-1]和 rc[0..n-1]进行相应调整。通过程序 indexx 构成一个
索引表
{
    int j;

    int n=ra.size();
    Vec_INT iwksp(n);
    Vec_DP wksp(n);
    indexx(ra,iwksp);                          制作索引表。
    for (j=0;j<n;j++) wksp[j]=ra[j];           存储数组ra。
    for (j=0;j<n;j++) ra[j]=wksp[iwksp[j]];    复制它并重排返回。
    for (j=0;j<n;j++) wksp[j]=rb[j];           依次处理rb。
    for (j=0;j<n;j++) rb[j]=wksp[iwksp[j]];
    for (j=0;j<n;j++) wksp[j]=rc[j];           依次处理rc。
    for (j=0;j<n;j++) rc[j]=wksp[iwksp[j]];
}
```

数组数更多的情况显然可直接类推。

一个**秩表**不同于一个索引表。一个秩表的第 j 个入口给出原先的关键数组中第 j 个元素的秩,其范围是从 0(如果该元素是最小的)到 $N-1$(如果该元素最大)。能够很容易地从一个索引表产生一个秩表:

```
#include "nr.h"

void NR::rank(Vec_I_INT &indx, Vec_O_INT &irank)
给定程序 indexx 输出的索引表数组 indx[0..n-1],本程序返回相应的秩表数组 irank[0..n-1]。
{
    int j;

    int n=indx.size();
    for (j=0;j<n;j++) irank[indx[j]]=j;
}
```

图 8.4.1 总结了本节中所讨论的概念。

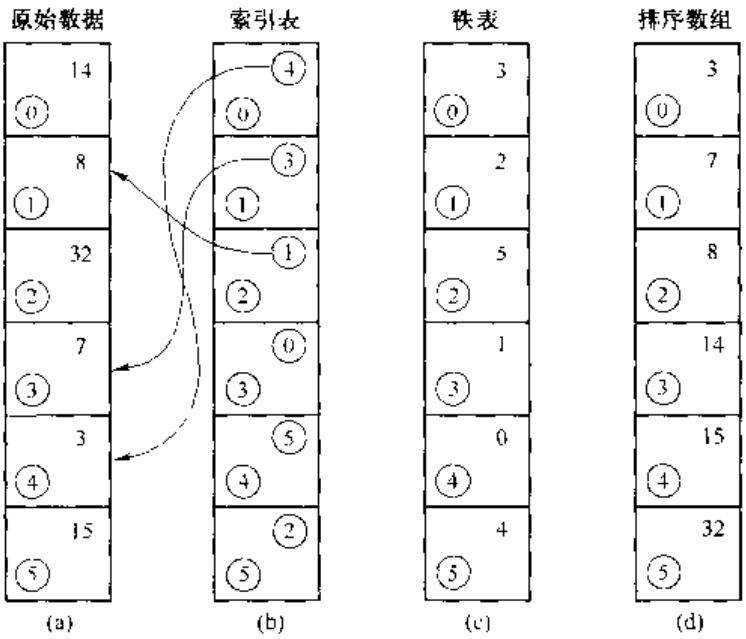


图 8.4.1　(a)：一个未排序的六个数的数组；(b)：索引表，其内容是指出(a)中元素按递增顺序排列的指引；(c)：秩表，其内容是对应(a)中元素的秩序号；(d)：(a)中元素的排序数组

8.5　挑选第 M 大的元素

挑选和排序是相关的。排序要求将整个数据数组重排，而挑选则只求选出一个返回值：在 N 个元素中哪一个是第 k 个小的元素(或等效地，第 $m=N-1-k$ 大的元素)？(注意，本节中 $k=0,1\cdots,N-1$，所以 $k=0$ 对应数组中的最小元素。)遗憾的是，为了达到自身的计算目的，最快的挑选方法需重整数组，典型的是将所有较小的元素置于第 k 个元素之左，所有较大元素置于右，而在每一个子集内顺序是杂乱的。这种副作用轻则是无害的，重则就十分的不方便。当序列很长时，进行打印就会很费时。即使挑选所带来的计算负担，只是一个更大计算量中可忽略的一部分，人们还是喜欢选用没有副作用的挑选方法，这种方法不打乱原数组的顺序。这种**定点**挑选法的速度是更快方法的 1/10。下面给出两种程序。

挑选通常用于对一组数据进行统计定性。通常想知道一列数据的中间值，或上四分之一或下四分之一值。当 N 是奇数时，中间值就是第 k 个元素，其中 $k=(N-1)/2$。当 N 是偶数时，统计书上定义中间值是 $k=N/2-1$ 和 $k=N/2$ 两元素(也就是从上数第 $N/2$ 和从下数第 $N/2$ 个元素)的算术平均值。如果接受这种规定，就必须分别进行两次挑选，来找到这些值。对于 $N>100$，通常定义 $k=N/2$ 为中间元素，不按这种规定。

若允许重排数组，最快的挑选方法则就是"划分法"，与快速排序法(参见 8.2 节)中操作完全相同。挑选一个"随机"分割元素，用它来核对整个序列，迫使较小的元素排列左边，较大的排到右边。正如在快速排序法中那样，优化内循环是非常重要的，使用"限制"(参见 8.2 节)来减少比较的数量。但在排序中，还需对分割出的两个子数组分别继续进行操作。然而在挑选中，可以略去一个子数组，而只注意包含了要找的第 k 个元素的那一个子集。用划分法来挑选就不需要用于操作期间的堆栈，并且它的运算量是 N 量级而不是 $N\log N$ 量级的[1]。对照 8.2 节中的排序程序 sort，下列程序就很容易明白：

```
#include "nr.h"

DP NR::select(const int k, Vec_IO_DP &arr)
```

返回数组 arr[0..n-1]中第 k 小的元素。输入的数组将被重排，所挑选的值返回在 arr[k]中，所有小的元素在 arr[0..k-1]中(以任意顺序)，而所有大的元素都在 arr[k+1..n-1]中(以任意顺序)。

```
{
    int i,ir,j,l,mid;
    DP a;

    int n=arr.size();
    l=0;
    ir=n-1;
    for (;;) {
        if (ir <= l+1) {                    激活包含1个或者2个元素的划分。
            if (ir == l+1 && arr[ir] < arr[l])        两个元素的情况。
                SWAP(arr[l],arr[ir]);
            return arr[k];
        } else {
            mid=(l+ir) >> 1;                挑选左、中、右三元素的中间值作为元素a,
            SWAP(arr[mid],arr[l+1]);        同时重整以使arr[l]≤arr[l+1],arr[ir]≥arr[l+1]。
            if (arr[l] > arr[ir])
                SWAP(arr[l],arr[ir]);
            if (arr[l+1] > arr[ir])
                SWAP(arr[l+1],arr[ir]);
            if (arr[l] > arr[l+1])
                SWAP(arr[l],arr[l+1]);
            i=l+1;                          为划分初始化指针。
            j=ir;
            a=arr[l+1];                     划分元素。
            for (;;) {                      开始内循环。
                do i++; while (arr[i] < a);         从数组首部向下扫描寻找元素>a。
                do j--; while (arr[j] > a);         从数组尾部向上扫描寻找元素<a。
                if (j < i) break;           指针交汇，划分结束。
                SWAP(arr[i],arr[j]);
            }                               结束内循环。
            arr[l+1]=arr[j];                插入划分元素。
            arr[j]=a;
            if (j >= k) ir=j-1;             使包含第k元素的划分继续进行。
            if (j <= k) l=i;
        }
    }
}
```

定点挑选或称非破坏挑选在概念上很简单，但它需要很多记录，因而相对较慢。基本的想法是，随机地选出 M 个元素，将其排序，然后将整个数组通过一遍，计算在 $M+1$ 个空隙中分别有多少元素。包含第 k 个大的元素的空隙称为"活"空隙。然后再进行第二轮，首先在活空隙中随机地挑选 M 个元素，再确定当前活元素落入新的、更细的 $M+1$ 个空隙中的哪一个，依次类推，直到第 k 个元素定位在一个大小为 M 的单个数组中，这时即可进行直接挑选。

如何挑选 M 呢？操作的循环次数为 $\log_M N = \log_2 N/\log_2 M$ 随着 M 的增大而减小，但将每一元素定位于 $M+1$ 个子空隙的工作量则增大。例如，对于二分法来说是 $\log_2 M$ 量级的大小。在每一轮中需要查看所有 N 个元素，若仅找那些活元素，那二分法主要由发生在第一轮中的 N 所支配。使 $O(N\log_M N)+O(N\log_2 M)$ 取极小值，便得结果

$$M \sim 2\sqrt{\log_2 N} \tag{8.5.1}$$

对数值的平方根变化是很缓慢的，因而对机器时间的考虑变得重要了。通常选用 $M=64$。

在下列程序 selip 中有两个附加的小技巧，(i)将 M 个随机值的集增扩到 $M+1$，其第 $M+1$ 个元素是它们的算术平均值，(ii)用一种使靠前的元素更容易被选中的方法，在一次"飞过"数据过程中挑选这 M 个随机值(这种基本想法是，使 $m>M$ 的元素有 M/m 的机会被选中。它能用归纳法证明，将会产生所需要的结果)。

```
#include "nr.h"

DP NR::selip(const int k, Vec_I_DP &arr)
返回数组 arr[0,n-1]中第 k 小的元素，输入数组不变。
{
    const int M=64;
    const DP BIG=1.0e30;
    int i,j,jl,jm,ju,kk,mm,nlo,nxtmm;
    DP ahi,alo,sum;
```

```
    Vec_INT isel(M+2);
    Vec_DP sel(M+2);

    int n=arr.size();
    if (k < 0 || k > n-1) nrerror("bad input to selip");
    kk=k;
    ahi=BIG;
    alo = -BIG;
    for (;;) {                                  主反复循环直到找到所需元素。
        mm=nlo=0;
        sum=0.0;
        nxtmm=M+1;
        for (i=0;i<n;i++) {                     将整个数组通过一遍。
            if (arr[i] >= alo && arr[i] <= ahi) {
                只考虑括号内的元素。
                mm++;
                if (arr[i] == alo) nlo++;       在括号相等的情况中。
                现在用统计方法在范围以内等概率地选 m 个元素，即使事先不知道它们有多少。
                if (mm <= M) sel[mm-1]=arr[i];
                else if (mm == nxtmm) {
                    nxtmm=mm+mm/M;
                    sel[(i+2+mm+kk) % M]=arr[i];      %操作提供某种随机数。
                }
                sum += arr[i];
            }

        }
        if (kk < nlo) {                         所需元素与下界相等，返回它。
            return alo;
        }
        else if (mm < M+1) {                    所有范围内的元素被保留，于是用直接
            shell(mm,sel);                      方法返回答案。
            ahi = sel[kk];
            return ahi;
        }
        sel[M]=sum/mm;                          用平均值扩充（固定退化）所选集合
        shell(M+1,sel);                         并排序。
        sel[M+1]=ahi;
        for (j=0;j<M+2;j++) isel[j]=0;          将计数数组置零。
        for (i=0;i<n;i++) {                     将整数数组再过一遍。
            if (arr[i] >= alo && arr[i] <= ahi) {        对于范围内的元素 ……
                jl=0;
                ju=M+2;
                while (ju-jl > 1) {             ……用二分方法找寻它的位置……
                    jm=(ju+jl)/2;
                    if (arr[i] >= sel[jm-1]) jl=jm;
                    else ju=jm;
                }
                isel[ju-1]++;                   …… 并计数器加1。
            }
        }
        j=0;                                    现在可以将范围缩小一级，即缩小m
        while (kk >= isel[j]) {                 级因子。
            alo=sel[j];
            kk -= isel[j++];
        }
        ahi=sel[j];
    }
}
```

大概的时间估算：selip 大约是 select 的 1/10。实际上，当 N 在 $\sim 10^5$ 范围内，selip 的时间是用 sort 进行一次全排序的 2/3，而 select 比 sort 快 6 倍。用户应在时间、存储和方便性之间仔细权衡。

当然，要寻找数组中最大或最小元素，不应该采用上述任一种程序，读者可以自己简单地用 for 循环编程。当与 N 比较 k 是较适中的情况，还有一种好的编程方法，使附加 k 级存储量的负担不是太重。例如，采用堆排序法（参见 8.3 节），将整个 N 长的数组通过一遍，而保存 m 个**最大**的元素。堆积结构的优点是，每个候选人被入选只需比较 $\log m$ 一次，而不是 m 次。当 $m > O(\sqrt{N})$ 时，此方法确定需占用相当的内存，但这并不碍事而且容易编程。下列程序展示了这一思想。

```
#include "nr.h"

void NR::hpsel(Vec_I_DP &arr, Vec_O_DP &heap)
在 heap[0..m-1]中返回数组 arr[0..n-1]的前 m 个最大的元素,并保证 heap[0]是第 m 大的元素。数组 arr 不变。为保证效率,本程序必须仅当 m≪n 时使用。
{
    int i,j,k;

    int m=heap.size();
    int n=arr.size();
    if (m > n/2 || m < 1) nrerror("probable misuse of hpsel");
    for (i=0;i<m;i++) heap[i]=arr[i];
    sort(heap);                     通过淘汰创建初始堆。假定m<<n。
    for (i=m;i<n;i++) {             对每一个所剩元素……
        if (arr[i] > heap[0]) {     置于堆上?
            heap[0]=arr[i];
            for (j=0;;) {           下移。
                k=(j << 1)+1;
                if (k > m-1) break;
                if (k != (m-1) && heap[k] > heap[k+1]) k++;
                if (heap[j] <= heap[k]) break;
                SWAP(heap[k],heap[j]);
                j=k;
            }
        }
    }
}
```

参考文献和深入阅读

Sedgewick, R. 1998, *Algorithms in C*, 3rd ed. (Reading, MA: Addison-Wesley), pp. 126ff. [1]

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley).

8.6 深入讨论:等价类的确定

一些排序和查找的技巧涉及到数据结构方面的知识,例如树和链接表等,其详细内容已超出了本书的讨论范围。这些结构及其操作是计算机科学的精彩部分,它与数字分析内容截然不同,关于这一主题的书籍并不缺乏。

在处理实验数据的过程中,我们发现一种,即特殊操作确定等价类,它在应用中常常出现,所以我们将它包括起来。

问题是:有 N 个元素(或者数据点),编号为 $0,\ldots,N-1$。给出一些表明元素是否属于"同一性"的相同等价类的配对信息,这些信息可以是根据任何恰好感兴趣的标准得来的。例如,可能得到一系列事实,如:"元素 3 和元素 7 是同一类,元素 19 和元素 4 是同一类,元素 7 和元素 12 是同一类,等等"。或者有一个过程,可以用来判断任何给定元素 j 和 k 是属于同一类还是属于不同类(回忆一下,一个等价关系可以是满足 **RST 属性**的任何关系,即满足自反性、对称性和传递性,这是和同一性的任何直观定义相兼容的)。

希望得到的输出是,将 N 个元素中的每一个赋予一个等价类序号。当且仅当两个元素被赋予同一类序号时它们属于同一类。

有效的算法是这样进行的:令 $F(j)$ 是元素 j 的类序号或"家族"序号。从每个元素在它自己的家族开始,所以 $F(j)=j$。数组 $F(j)$ 能被解释成一个树结构,其中 $F(j)$ 代表 j 的双亲。如果我们将每个家族排成它自己的树,与所有其他"家族树"不相交,那么,就可以用它最老的祖先来标志这一家族(等价类)。这一树的具体拓扑结构并不重要,只要在某处将有关的元素嫁接上去即可。

因此,按如下方式来处理每个基本的论据"j 等价于 k":(i)追溯 j 到它的最老祖先;(ii)追溯 k 到它的最老祖先;(iii)令 j 为 k 的新双亲,或反之亦然(这没有什么区别)。当处理所有的关系之后,检查所有的元素 j,并将它们的 $F(j)$ 重新置成其可能的最老祖先,从而它标示了等价类。

下面是基于 Knuth[1]的程序,假定有 m 个基本信息。它们存储于两个长度为 m 的数组中 lista 和 listb。它的解释是:lista[j]和 listb[j](j=0...m-1)是两个相关元素的序号。

```
#include "nr.h"

void NR::eclass(Vec_O_INT &nf, Vec_I_INT &lista, Vec_I_INT &listb)
在输入数组 lista[0...m-1]和 listb[0...m-1]中,给出 n 个单个元素对之间 m 个两两等价信息,本程序返回 nf[0...n-1]标明每
个元素的等价类序号,该序号是从 0 到 n-1 的整数(不是都要用到)。
{
    int l,k,j;

    int n=nf.size();
    int m=lista.size();
    for (k=0;k<n;k++) nf[k]=k;              将每个元素初始化成它自身的一类。
    for (l=0;l<m;l++) {                     对每一条输入信息……
        j=lista[l];
        while (nf[j] != j) j=nf[j];         追溯第一个元素到它的祖先。
        k=listb[l];
        while (nf[k] != k) k=nf[k];         追溯第一个元素到它的祖先。
        if (j != k) nf[j]=k;                如果它们还没有相关，则使它们相关。
    }
    for (j=0;j<n;j++)                       最后扫到最高祖先。
        while (nf[j] != nf[nf[j]]) nf[j]=nf[nf[j]];
}
```

或有可能,可以构造一个布尔值函数 equiv(j,k),它在 j 和 k 相关时返回一个 true 值,或在不相关时返回 false。于是,循环所有的元素对,就可得到一个完整的图片。D. Eardley 发明了一种方法能在完成上述要求的同时,以一种流畅且能消除大多数最终扫视的方法,对树结构扫视到老祖先。

```
#include "nr.h"

void NR::eclazz(Vec_O_INT &nf, bool equiv(const int, const int))
给出一个用户提供的布尔值函数 equiv,它告之一对元素是否相关(每个元素都在范围 0...n-1 中)。以 nf[0..n-1]返
回每个元素的等价分类号。
{
    int kk,jj;

    int n=nf.size();
    nf[0]=0;
    for (jj=1;jj<n;jj++) {                  对所有对的第一个元素循环。
        nf[jj]=jj;
        for (kk=0;kk<jj;kk++) {             对所有对的第二个元素循环。
            nf[kk]=nf[nf[kk]];              上扫到此。
            if (equiv(jj+1,kk+1)) nf[nf[nf[kk]]]=jj;
            清楚为什么需要这样多的祖先是一个很好的练习。
        }
    }
    for (jj=0;jj<n;jj++) nf[jj]=nf[nf[jj]];        最终只需要这么多的扫视。
}
```

参考文献和深入阅读

Knuth, D.E. 1997, *Fundamental Algorithms*, 3rd ed., vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §2.3.3. [1]

Sedgewick, R. 1998, *Algorithms in C*, 3rd ed. (Reading, MA: Addison-Wesley), Chapter 30.

第9章　求根与非线性方程组

9.0　引言

现在，来考虑一个最基本的问题，即用数值方法求解方程的根。绝大多数方程既有左端项又有右端项，通常人们将所有项移至左端，化为

$$f(x) = 0 \tag{9.0.1}$$

的形式以求其解。当仅有一个独立变量时，问题称为**一维**的，即求一元函数的一个或多个根。

如果独立变量不只一个，则有不只一个方程可以同时被满足。读者可能学过**隐函数理论**，这个理论告诉我们，具有 N 个变量的 N 个方程有可能被同时满足。注意，这里仅仅是可能而不是肯定。一个方程的非线性集合可能根本没有(实)解；相反，它也可能有多个解。隐函数理论告诉我们，一般地，这若干个解之间应是互不相同、呈点状分布且相互分离的。但是，有时可能会遇到非一般的情况，即退化的情况，这时又可能得到连续的一族解。如果将问题用向量记号表示，即求一个或多个 N 维向量 $\mathbf{x}$，使得

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{9.0.2}$$

其中 $\mathbf{f}$ 是 N 维向量值的函数，由同时被满足的 N 个独立方程组成。

不要被(9.0.1)式与(9.0.2)式形式上的相似所迷惑。求同时满足多个方程的 N 维解，比一维情况下的求根要复杂得多。它们的主要区别在于：一维情况下可以确定解的区间，然后在解区间内"搜索"方程的根；而在多维情况下，始终不能确定根在何处，直到确实找到它为止。

除线性情况外，求根过程总要借助迭代来完成，在一维和多维情况下都是如此。常用的算法总是从某个近似的初始解出发，不断修改这个解，直至满足某种预先确定的收敛准则为止。对于光滑变化的函数，如果初始解足够好，则好的算法总是可以收敛的。实际上，绝大多数算法的收敛速度甚至都可以预先确定。

但是，不能过分强调求解成功与否是如何依赖于初始预测解，特别是对于多维问题。这个具有决定意义的开始步骤通常依赖于分析，而不是单纯数值。精心选择的初始预测不仅可以减少计算量，而且可以增强理解。汉明(Hamming)曾指出，"计算的目的在于洞察，而非得到数字"，这个观点特别适于求根过程。当读者的程序以十位数字的精确度收敛于一个问题的错解，或者由于根本无解，或虽有解但因预测初值没有足够靠近该值而使程序不能收敛时，请仔细思考汉明的这句名言。

读者可能会问，"汉明的观点是正确的，但是到底应该怎样做呢？"对一维求根问题，答案很直截了当：在试着开始求解之前，应该首先对方程的特点有一个大致的轮廓。如果要成批地求解许多不同方程的根，那么至少应对其中一些典型方程做到心中有数。其次，在迭代求根前应确定好求解区间，即确定函数在什么区间内改变其正负号。

最后(对这一点某些读者可能有不同看法，但我们仍在此提出来)，在迭代的每一步中，都不要让迭代算法跨越已划定的最佳搜索区间之外，下面将会看到一些在教学上很重要的算法，如**弦截法**或称 **Newton-Raphson 算法**等都可能会违背这条限制。因此，如果不对这些算法进行某些改进，建议最好不要使用它们。

出现多重根或根相近的情况，的确是个令人头痛的问题，特别是当重数为偶数时。在这种情况下，函数可能不会有明显的符号变化，因而确定和保持解区间就变得很困难了。但是可以采取强硬措施，即硬行确定一个解区间，哪怕需要使用第10章中的极小值搜索技术，来决定函数的某个低峰是否确实通过零点(读者可以将10.1节中的简单黄金分割法程序做一点小的修改，使之一旦检查出函数的符号变化，就立即返回调用处，而且可以判断如果函数的极小值恰好是零，则我们找到了一个二重根)。

我们照例不主张读者把程序作为"黑箱"程序使用，而一概对其内部结构不求理解。但是，对初学者来说，以下几条建议是很可取的：

- 在函数的导数不易计算时，可以选用 9.3 节中的 Brent 算法，它适于求解一般一元函数的根，但这个根应位于搜索区间内。在 9.2 节中 Ridders 方法简明扼要，也是比较理想的选择。
- 如果函数的导数是可以计算的，建议读者利用 9.4 节中程序 rtsafe，该程序把对边界信息所做的某些记录与 Newton-Raphson 方法结合起来，但它同样也需要首先确定解区间。
- 多项式求根是一种特殊情况，可以用 9.5 节中的拉盖尔方法作为起点。应注意的是，有些多项式是病态的。
- 最后，对于多维问题，除 Newton-Raphson 方法（参见 9.6 节）外，本书将不再讨论其他算法。如果能给出一个很合理的关于解的初始预测，将是一个相当不错的算法，读者不妨试一试。此外 9.7 节中还有几种更复杂但全局收敛性更好的算法，可供读者进一步阅读参考。

为适于不同的计算机实现，本书一般都避开了交互式的或与图形相关的程序。但此处仍有一个例外。在下面的程序中，我们用交互式的方法确定坐标轴的刻度，从而产生一幅粗略的函数图形。对于初涉方程求根领域的读者，采用这种方法可以避免很多麻烦

```
#include <string>
#include <iostream>
#include <iomanip>
#include "nr.h"
using namespace std;

void NR::scrsho(DP fx(const DP))
适用于交互式 CRT 终端，在区间 x1 至 x2 内绘出函数 fx 的大致图形。等待输入另一图形区间，直到用户输入结束标志，
以示满意为止。
{
    const int ISCR=60, JSCR=21;                 屏上所显示的水平和垂直位置的数目。
    const char BLANK=' ', ZERO='-', YY='l', XX='-', FF='x';     display.
    int jz,j,i;
    DP ysml,ybig,x2,x1,x,dyj,dx;
    Vec_DP y(ISCR);
    string scr[JSCR];

    for (;;) {
        cout << endl << "Enter x1 x2 (x1=x2 to stop):" << endl;
        等待输入另一个图形，当 x1=x2 时退出。
        cin >> x1 >> x2;
        if (x1 == x2) break;
        scr[0]=YY;                              左上角填入字符 'l'。
        for (i=1;i<(ISCR-1);i++)                首行其余位置填入字符 '-'。
            scr[0] += XX;
        scr[0] += YY;                           首行末尾处填入字符 'l'。
        for (j=1;j<(JSCR-1);j++) {
            scr[j]=YY;                          左边填 'l'。
            for (i=1;i<(ISCR-1);i++)            内部填空格。
                scr[j] += BLANK;
            scr[j] += YY;                       右边填 'l'。
        }
        scr[JSCR-1]=scr[0];                     底边与首行相同。
        dx=(x2-x1)/(ISCR-1);
        x=x1;
        ysml=ybig=0.0;                          边界包括 0。
        for (i=0;i<ISCR;i++) {                  计算函数在等间隔点的值。找出最大与
            y[i]=fx(x);                         最小值。
            if (y[i] < ysml) ysml=y[i];
            if (y[i] > ybig) ybig=y[i];
            x += dx;
        }
        if (ybig == ysml) ybig=ysml+1.0;        确保顶端和底端分开。
        dyj=(JSCR-1)/(ybig-ysml);
        jz=int(-ysml*dyj);                      指明哪一行对应零。
        for (i=0;i<ISCR;i++) {                  在函数高度和零位处设置标记。
            scr[jz][i]=ZERO;
            j=int((y[i]-ysml)*dyj);
            scr[j][i]=FF;
        }
        cout << fixed << setprecision(3);
        cout << setw(11) << ybig << " " << scr[JSCR-1] << endl;
        for (j=JSCR-2;j>=1;j--)                 显示。
            cout << " " << scr[j] << endl;
```

```
            cout << setw(11) << yaml << " " << scr[0] << endl;
            cout << setw(19) << x1 << setw(55) << x2;
        }
    }
```

参考文献与深入阅读

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), Chapter 5.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapters 2, 7, and 14.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), Chapter 8.

Householder, A.S. 1970, *The Numerical Treatment of a Single Nonlinear Equation* (New York: McGraw-Hill).

9.1 划界与二分

如果 $f(a)$ 与 $f(b)$ 具有相反的符号，则我们认为将有一根被**划界**在区间 (a,b) 内；又若函数是连续的，则至少有一个根必落在该区间内(**中值定理**)。对非连续但有界的函数，则可能有一阶跃(而不是一个根)间断跨过零点(参见图 9.1.1)。从数值上解释，这一阶跃也可能是一个根，因为这种情况与某种连续函数很难区分，这种连续函数在机器的有限精度表示下，其零点可能会出现在两个"相邻"的浮点数之间。只有带奇点的函数才有可能在划定的区间内没有任何根存在。

例如，对于函数

$$f(x)=\frac{1}{x-c} \tag{9.1.1}$$

某些求根算法(如本节的二分法)可能会收敛到 c。幸运的是，大多数情况下几乎都不可能得到 c 或任何接近于 c 的错误解，因为只需稍微计算一下 $|f(x)|$ 就会发现这个值很大而不是很小。

如果函数是在黑箱中给出的，则没有一种确定的方法可以给根划界，甚至不能确定函数是否有根。例如(3.0.1)式，它的两个实根仅仅落在一个极小的区间范围 $x=\pi\pm10^{-667}$ 内。

在下一章中，将讨论与本章相关的函数极小值的划界问题。对于这个问题，可以找到一种在任何情况下都能成功的求解方法，从本质上讲，即"向下搜索，同时逐渐加大步长，直到函数值又开始回升为止"。但在求根过程中，则没有类似的方法可寻。"向下搜索，直到函数改变符号"的方法对于具有简单极值的函数并不适用。但是，如果对"失败"的结果有所准备，这种方法却通常是一个良好的开端：如果函数在 $x\to\pm\infty$ 的范围内具有相反的符号，则可以很有把握地得到成功的结果。

```
#include <cmath>
#include "nr.h"
using namespace std;

bool NR::zbrac(DP func(const DP), DP &x1, DP &x2)
给定一个函数 func 及一个初始的预测范围 x1 到 x2，本程序将这处范围进行几何扩展，直到方程有某个根被划界在返回
的值 x1 和 x2 之间(此时，zbrac 返回 true)，或者直至这个范围被扩展到大得不能接受的程度为止(这时 zbrac 返回 false)
{
    const int NTRY=50;
    const DP FACTOR=1.6;
    int j;
    DP f1,f2;

    if (x1 == x2) nrerror("Bad initial range in zbrac");
    f1=func(x1);
    f2=func(x2);
    for (j=0;j<NTRY;j++) {
        if (f1*f2 < 0.0) return true;
        if (fabs(f1) < fabs(f2))
            f1=func(x1 += FACTOR*(x1-x2));
        else
            f2=func(x2 += FACTOR*(x2-x1));
    }
    return false;
}
```

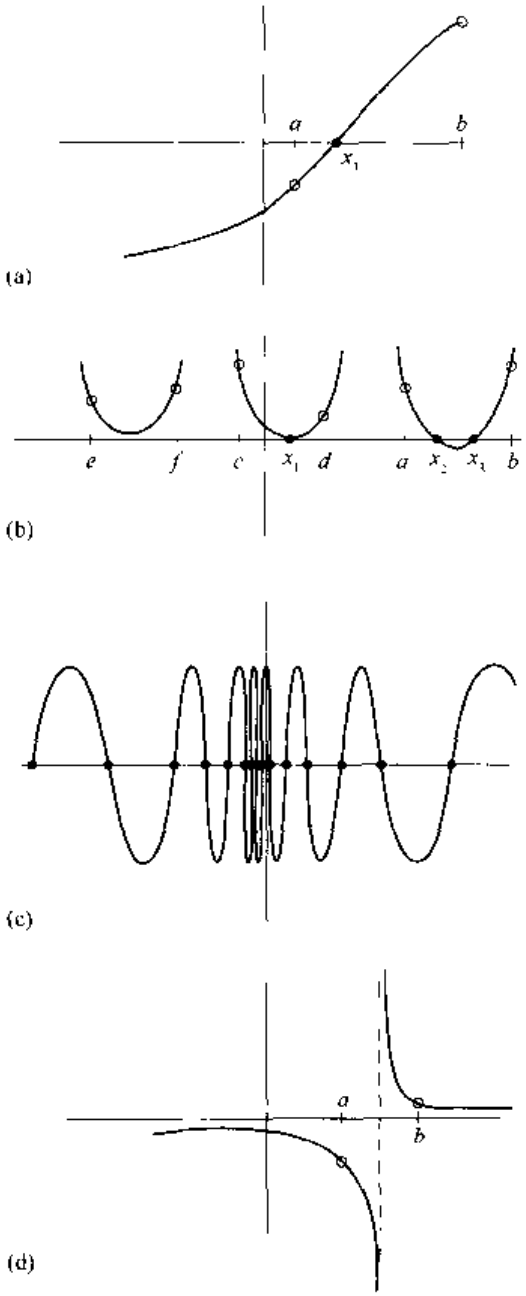


图 9.1.1　在方程式求根过程中可能会遇到的几种情况。(a)表明由 a 和 b 两点划界的一个孤立的根 x_1，其中 a 和 b 两点函数值具有相反的符号；(b)说明在一个二重根附近，函数值不一定必须有符号变化(实际上也不一定必须有一个根)；(c)说明一个病态函数可能有多个根；在(d)中，函数在 a 和 b 两点符号相反，但 a 和 b 两点之间包含的是一个奇异点，而不是一个根

读者可能期望"向内"考察初始区间，而不希望将这个区间"向外"扩展，并且提出，如果将区间 x_1 到 x_2 划分为 n 个等间隔的小区间来进行搜索，函数 $f(x)$ 在这些小区间内是否有根存在。下面的程序可以计算出 nb 个不同的小区间的边界，其中每个小区间内包含一个或多个根。

```
#include "nr.h"

void NR::zbrak(DP fx(const DP), const DP x1, const DP x2, const int n,
    Vec_O_DP &xb1, Vec_O_DP &xb2, int &nroot)
```

给定一个定义在区间(x1,x2)内的函数 fx，将这个区间划分为 n 个等间隔的小段，并求函数过零点的次数。将划定的区间的两端成对顺序存入数组 xb1[0..nb-1]，xb2[0..nb-1]，其中需保证 nb 足够大。而最终划定的区间的实际数目存入 nroot。

```
{
    int i;
    DP x,fp,fc,dx;

    int nb=xb1.size();
    nroot=0;
    dx=(x2-x1)/n;                    确定适当的网格划分。
    fp=fx(x=x1);
    for (i=0;i<n;i++) {              对所有区间进行循环。
        fc=fx(x += dx);
        if (fc*fp <= 0.0) {          若有符号变化，则记下边界值。
            xb1[nroot]=x-dx;
            xb2[nroot++]=x;
            if(nroot == nb) return;
        }
        fp=fc;
    }
}
```

9.1.1 二分法

一旦知道了某个区间内包含有方程的一个根，则有好几种经典方法可将该区间进一步细分。这些方法以不同的收敛速度和确定性程度趋于答案。但遗憾的是，能够保证收敛的方法求解速度却很慢，而那些收敛速度很快的方法却极有可能以极快的速度趋向无穷，而且如果没有采取措施避免这种情况的发生，算法本身不会给出任何警告信息。

二分法是一种常胜的方法，故它不是针对性质很差的问题的方法。其设计思想相当简单：如果在某一区间内函数的符号有变化，则在此区间内函数必过零。计算函数在该区间中点的值，并考察其符号。用中点替代与它具有相同函数符号的端点。每经过一次迭代，包含根的区间就减小了一半，假设 n 次迭代后，根位于长度为ϵ_n 的区间内，则在下一轮迭代结束后，这个根将被划界在长度恰好为

$$\epsilon_{n+1} = \epsilon_n/2 \tag{9.1.2}$$

的区间内，因此，我们便可以事先计算出要达到给定容限的解所需的迭代次数，为

$$n = \log_2 \frac{\epsilon_0}{\epsilon} \tag{9.1.3}$$

其中ϵ_0 是初始划分区间的长度，ϵ为迭代结束时所期望达到的容限。

二分法必然求解成功。假如区间碰巧包含两个或多个根，则用二分法可以找到其中之一。如果区间内不包含方程的根而仅有一奇点，则二分法将收敛到该奇点。

如果一种方法，是以前次迭代不确定性的一阶幂乘以一个因子(小于 1)的速度收敛，则称这种方法为**线性收敛**(如二分法就属于这种情况)。而以高阶幂收敛的方法，即

$$\epsilon_{n+1} = \text{constant} \times (\epsilon_n)^m \qquad m > 1 \tag{9.1.4}$$

称为超线性收敛。在其他书中可能会把“线性”收敛称为“指数”收敛或“几何”收敛，但名称是什么无所谓。线性收敛性就是指，有效数字随着计算而线性地逐次增加。

剩下的工作就是讨论如何确定一种实用收敛准则。机器中总是以固定位数的二进制数来表示浮点的，记住这一点极为关键。如果一个函数经分析可知它必过零点，但在计算中可能对任何浮点数变量，函数根本不取零值。因此，必须确定出方程的根可以达到的精度值。例如，当方程根的值大小接近 1，那么取收敛到绝对值小于 10^{-6}就是比较合理的，但这个精度对于接近 10^{26}的根当然是不可能达到的。由此，可能会想到确定一个相对的(分数的)收敛准则，但对接近于零的根来说，这个准则就又不适用了。为了最大限度地确保通用性，下面的程序要求用户指定一个绝对容限，使算法在区间的绝对长度小于该容限时终止迭代。通常，我们可能倾向于将该容限值取为$\epsilon(|x_1| + |x_2|)/2$，其中$\epsilon$为机器精度，$x_1$ 和 x_2 为初始区间的两个端点。但当根位于零值附近时，必须注意仔细考虑什么样的容限值对我们的函数来说是合理的。下面程序在任何情况下最多允许执行 40 次二分，因为 $2^{-40} \approx 10^{-12}$。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::rtbis(DP func(const DP), const DP x1, const DP x2, const DP xacc)
```

```
已知位于 x1 和 x2 之间的函数 func,用二分法求其根。在精度达到 ±xacc 之前,这个根将被不断修正,并最终以 rtbis 返回。
{
    const int JMAX=40;                                  允许迭代的最大次数。
    int j;
    DP dx,f,fmid,xmid,rtb;

    f=func(x1);
    fmid=func(x2);
    if (f*fmid >= 0.0) nrerror("Root must be bracketed for bisection in rtbis");
    rtb = f < 0.0 ? (dx=x2-x1,x1) : (dx=x1-x2,x2);      确定搜索方向以使 f>0 位于
    for (j=0;j<JMAX;j++) {                              x+dx 处。
        fmid=func(xmid=rtb+(dx *= 0.5));                二分法循环。
        if (fmid <= 0.0) rtb=xmid;
        if (fabs(dx) < xacc || fmid == 0.0) return rtb;
    }
    nrerror("Too many bisections in rtbis");
    return 0.0;                                         程序执行不到此处。
}
```

9.2　弦截法、试位法和Ridders方法

对于在根附近光滑的函数,**试位法**和**弦截法**的收敛速度通常要比二分法快。在这两种方法中,函数在所讨论的局部范围内均假设为近似线性,而且根的下一估计值总是取在接近直线与坐标轴的交点处。每一轮迭代结束后,上一轮迭代的边界点中将有一个被舍弃,以有利于根的最新估计。

上述两种方法的**惟一**区别在于,弦截法总是取两个端点的最新估计值(参见图 9.2.1,该方法要求在第一轮迭代开始任意选择一个区间);而试位法需要保留上一轮迭代的一个端点,条件是该端点的函数值,与当前求出的根的最佳近似的函数值,具有相反的符号,从而可以使这两点能继续划界根(参见图 9.2.2)。精确地说,对于足够连续的函数,弦截法在根附近收敛速度加快,其收敛阶可达到"黄金比率"1.618...,即

$$\lim_{k\to\infty} |\epsilon_{k+1}| \approx \text{const} \times |\epsilon_k|^{1.618} \tag{9.2.1}$$

但是弦截法也有一大缺陷,即方程的根不一定总能落在迭代求出的划界之内。这样对于不是足够连续的函数,该算法就不能保证收敛,因为局部迭代过程可能会导致趋于无穷。

由于试位法有时需保留旧端点,而不取新计算出来的估计值,因而其收敛阶要稍低一些。正因为新的函数值只是**有时**被保留下来,所以这种方法常常是超线性收敛的,但是精确的收敛阶数却不易估算。

这两种相关方法的程序实现将在下面给出。尽管它们都为教科书所采用,但程序后面给出的 Ridders 方法以及下节中将要介绍的 Brent 方法常常更受青睐。图 9.2.3 展示的是弦截法和试位法对一个特殊函数的求根过程。

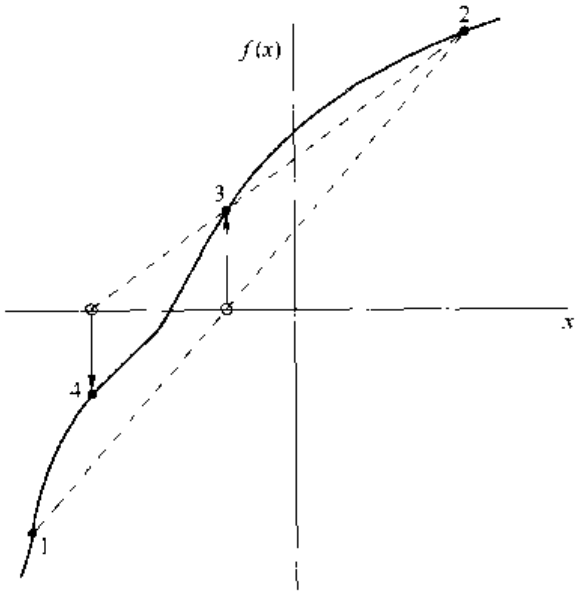


图 9.2.1　弦截法图示。过最新计算出来的两点做内插线或外推线(虚线所示)而不论这两点是否划界这函数。图中的点是按被使用的顺序进行编号的

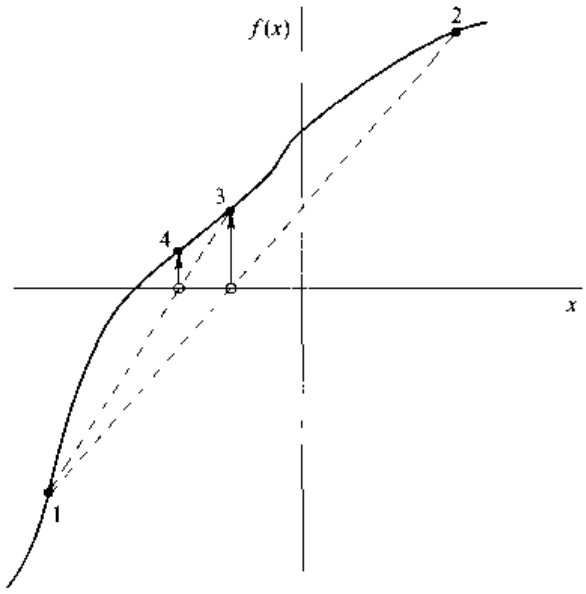


图 9.2.2 试位法图示。过划界根的最新端点做内插线(虚线所示),这两个端点之间必须包含有方程的根。在本例中,点1在很多步迭代中都将保持"有效"而不被其他点取代。试位法比弦截法收敛速度慢,但比弦截法确保收敛的程度要高

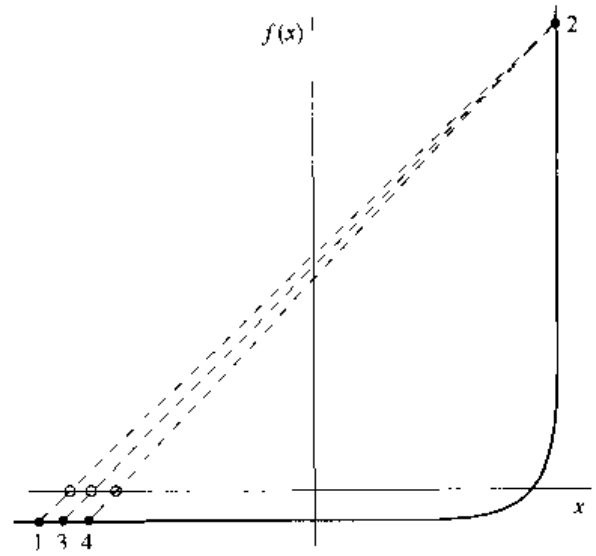


图 9.2.3 对本图所示的函数,弦截法和试位法都需要进行很多步迭代才能求出真解,而用许多其他算法都不易求解

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::rtflsp(DP func(const DP), const DP x1, const DP x2, const DP xacc)
已知位于 x1 和 x2 之间的函数 func,用试位法求其根,在精度达到 ± xacc 之前,修正不断地这个根,并最终以 rtflsp 返回。
{
    const int MAXIT=30;                 允许迭代的最多次数。
    int j;
    DP fl,fh,xl,xh,dx,del,f,rtf;

    fl=func(x1);
    fh=func(x2);                        确保区间内包含一个根。
    if (fl*fh > 0.0) nrerror("Root must be bracketed in rtflsp");
    if (fl < 0.0) {                     调整端点标号,使 x1 对应于函数值的下限边。
```

```
        xl=x1;
        xh=x2;
    } else {
        xl=x2;
        xh=x1;
        SWAP(fl,fh);
    }
    dx=xh-xl;
    for (j=0;j<MAXIT;j++) {          试位法循环.
        rtf=xl+dx*fl/(fl-fh);        根据最新值调整增量。
        f=func(rtf);
        if (f < 0.0) {               替代满足条件的端点。
            del=xl-rtf;
            xl=rtf;
            fl=f;
        } else {
            del=xh-rtf;
            xh=rtf;
            fh=f;
        }
        dx=xh-xl;
        if (fabs(del) < xacc || f == 0.0) return rtf;       收敛。
    }
    nrerror("Maximum number of iterations exceeded in rtflsp");
    return 0.0;                      程序执行不到此处。

}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::rtsec(DP func(const DP), const DP x1, const DP x2, const DP xacc)
若函数 func 位于 x1 和 x2 之间,用弦截法求它的根。在精度达到 ±xacc 之前,不断地修正这个根,并最终以 rtsec 返回.
{
    const int MAXIT=30;              允许迭代的最多次数.
    int j;
    DP fl,f,dx,xl,rts;

    fl=func(x1);
    f=func(x2);
    if (fabs(fl) < fabs(f)) {        Pick the bound with the smaller function value as
        rts=x1;                          the most recent guess.
        xl=x2;
        SWAP(fl,f);
    } else {
        xl=x1;
        rts=x2;
    }
    for (j=0;j<MAXIT;j++) {          Secant loop.
        dx=(xl-rts)*f/(f-fl);        Increment with respect to latest value.
        xl=rts;
        fl=f;
        rts += dx;
        f=func(rts);
        if (fabs(dx) < xacc || f == 0.0) return rts;     Convergence.
    }
    nrerror("Maximum number of iterations exceeded in rtsec");
    return 0.0;                      Never get here.
}
```

9.2.1　Ridders 方法

Ridders[1]对试位法进行了极有益的改进。当某个根位于 x_1 与 x_2 之间时，Ridders 方法首先计算函数在中点 $x_3=(x_1+x_2)/2$ 的值，然后对下述指数函数进行因式分解，这个独特的函数将残差函数变为一条直线：

$$f(x_1)-2f(x_3)e^{Q}+f(x_2)e^{2Q}=0 \tag{9.2.2}$$

解这个关于 e^{Q} 的二次方程，可得

$$e^Q = \frac{f(x_3) + \mathrm{sign}[f(x_2)]\sqrt{f(x_3)^2 - f(x_1)f(x_2)}}{f(x_2)} \tag{9.2.3}$$

接下来将试位法应用于 $f(x_1), f(x_3)e^Q, f(x_2)e^{2Q}$(注意不是 $f(x_1), f(x_3), f(x_2)$)三个值,得到关于根的一个新估计值 x_4。总的更新公式利用了(9.2.3)式,为

$$x_4 = x_3 + (x_3 - x_1)\frac{\mathrm{sign}[f(x_1) - f(x_2)]f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}} \tag{9.2.4}$$

(9.2.4)式具有一些非常好的性质:首先,x_4 肯定落在区间(x_1, x_2)内,因此 Ridders 方法在迭代过程中不会越出划界;其次,连续利用(9.2.4)式可得收敛阶为二次,也即(9.1.4)式中的 m 为 2。由于每用一次(9.2.4)式需进行两次函数值的计算,因此 Ridders 方法实际的收敛为$\sqrt{2}$,而不是 2;但是每经过两次的函数值计算以后,迭代结果的有效数字位数就几乎翻倍,这也可以算是超线性收敛;再次,由于 Ridders 方法采用了指数(或称比率)因子,而不是通过多项式技术(如拟合一条抛物线)来消除函数的"转弯",因此它可称得上是一种特别有用的算法,在可靠性和收敛速度两个方面,Ridders 方法均可与下面将要讨论的 Van Wijngaarden, Dekker 及 Brent 方法相媲美

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::zriddr(DP func(const DP), const DP x1, const DP x2, const DP xacc)
```

已知位于 x_1 和 x_2 之间的函数 func,用 Ridders 方法求其根。这个根在精度达到 xacc 之前将不断地被修正,并最终以 zriddr 返回调用处。

```
{
    const int MAXIT=60;
    const DP UNUSED=-1.11e30;
    int j;
    DP ans,fh,fl,fm,fnew,s,xh,xl,xm,xnew;

    fl=func(x1);
    fh=func(x2);
    if ((fl > 0.0 && fh < 0.0) || (fl < 0.0 && fh > 0.0)) {
        xl=x1;
        xh=x2;
        ans=UNUSED;                          设置一个极不可能出现的值以简化下面程
        for (j=0;j<MAXIT;j++) {              序中的逻辑判断。
            xm=0.5*(xl+xh);
            fm=func(xm);                     每次迭代中计算的两个函数值中的第一个。
            s=sqrt(fm*fm-fl*fh);
            if (s == 0.0) return ans;
            xnew=xm+(xm-xl)*((fl >= fh ? 1.0 : -1.0)*fm/s);    修正公式。
            if (fabs(xnew-ans) <= xacc) return ans;
            ans=xnew;
            fnew=func(ans);                  每次迭代中计算的两个函数值中的第二个。
            if (fnew == 0.0) return ans;
            if (SIGN(fm,fnew) != fm) {       在下一次迭代中保证根被划界。
                xl=xm;
                fl=fm;
                xh=ans;
                fh=fnew;
            } else if (SIGN(fl,fnew) != fl) {
                xh=ans;
                fh=fnew;
            } else if (SIGN(fh,fnew) != fh) {
                xl=ans;
                fl=fnew;
            } else nrerror("never get here.");
            if (fabs(xh-xl) <= xacc) return ans;
        }
        nrerror("zriddr exceed maximum iterations");
    }
```

```
        else {
            if (fl == 0.0) return x1;
            if (fh == 0.0) return x2;
            nrerror("root must be bracketed in zriddr.");
        }
        return 0.0;                                  永远执行不到此处。
    }
```

参考文献和深入阅读

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §8.3.

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press), Chapter 12.

Ridders, C.J.F. 1979, *IEEE Transactions on Circuits and Systems*, vol. CAS-26, pp. 979–980. [1]

9.3　Van Wijngaarden-Dekker-Brent 方法

虽然弦截法和试位法一般来讲比二分法收敛速度快，但是在实践中人们发现，对于病态函数却是二分法的收敛速度要快些，如锯齿形函数、不连续函数，甚至是二阶导数在根附近有突变的光滑函数等。二分法总是将解区间对分一半，而弦截法和试位法有时要进行很多次迭代才能把两个相距较远的端点向根所在的位置拉近。Ridders 方法虽然很好，但有时反被复杂化。究竟有没有一种方法，既能超线性收敛，又不失二分法的确定性呢？

答案是肯定的。可以对一个假设为超线性收敛的方法进行跟踪和记录，看它是否真正地按假定的方式收敛。如果不是的话，可以在迭代过程中穿插一些二分法的步骤，以保证这种方法至少具有线性收敛性。这类堪称上策的方法，需要注意记录迭代过程中的细节，并需仔细考虑舍入误差对指导策略的影响，而且还必须清楚知道算法何时收敛。

20 世纪 60 年代，阿姆斯特丹数学中心的 Van Wijngaarden 和 Dekker 等学者成功地研究出一种能够实现上述要求的算法，而后该算法又由 Brent 进行了改进[1]。为简单起见，我们称该算法的最后版本为 **Brent 方法**。只要函数在包含根的初始区间内是可计算的，Brent 方法即可保证收敛（这个结论已由 Brent 证明）。

Brent 方法将根的划界、二分法以及**二次反插值**方法相结合，从过零附近的邻域进行迭代收敛。前面介绍的试位法和弦截法是在两个根的估值之间进行近似的线性运算，而二次反插值则用三个估值点拟合一个二次反函数（x 作为 y 的二次函数），该函数在 $y=0$ 处的值将作为根 x 的下一估值。当然，对于根落在解区间之外的情形应该也有应急措施。Brent 方法考虑了上述各种情况出现的可能性。假设三个点对为 $[a,f(a)]$，$[b,f(b)]$，$[c,f(c)]$，则内插公式（参见(3.1.1)式）为

$$x=\frac{[y-f(a)][y-f(b)]c}{[f(c)-f(a)][f(c)-f(b)]}+\frac{[y-f(b)][y-f(c)]a}{[f(a)-f(b)][f(a)-f(c)]}+\frac{[y-f(c)][y-f(a)]b}{[f(b)-f(c)][f(b)-f(a)]} \tag{9.3.1}$$

置 $y=0$，则可求得根的下一估值为

$$x=b+P/Q \tag{9.3.2}$$

其中记

$$R\equiv f(b)/f(c),\qquad S\equiv f(b)/f(a),\qquad T\equiv f(a)/f(c) \tag{9.3.3}$$

则有

$$P=S[T(R-T)(c-b)-(1-R)(b-a)] \tag{9.3.4}$$

$$Q=(T-1)(R-1)(S-1) \tag{9.3.5}$$

实际计算时，b 将作为当前迭代中根的最佳估计，P/Q 应为一个“小”的修正项。二次函数方法仅适合于比较光滑的函数，它们在运行过程中，极有可能迭代出与根相差很远的估计值，而且当 Q 很小而接近零时，在机器中有被零除以致溢出的危险。Brent 方法在设计时，通过保持对方程根的划界，并在做除法之前检查应在何处进行内插，因而避免了这些问题的发生。当修正项 P/Q 使根的估计值落在解区间之外，或者解区间长度的减小速度不够快时，Brent 方法将采用二分法的步骤。于是，在适当的步骤中，Brent 方法将二分法的收敛确定性与高阶方法收敛速度快的优点结合起来。对只知函数值（而不可求其导数或函数的形式）的一般一元方程的求根，建议读者选用这种方法。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

DP NR::zbrent(DP func(const DP), const DP x1, const DP x2, const DP tol)
已知位于x1和x2之间的函数func,用Brent方法求它的根,在其精度达到tol以前,这个根将不断地被修正,并最终以zbrent返回。
{
    const int ITMAX=100;                    允许迭代的最大次数。
    const DP EPS=numeric_limits<DP>::epsilon();
    机器的浮点数精度。
    int iter;
    DP a=x1,b=x2,c=x2,d,e,min1,min2;
    DP fa=func(a),fb=func(b),fc,p,q,r,s,tol1,xm;

    if ((fa > 0.0 && fb > 0.0) || (fa < 0.0 && fb < 0.0))
        nrerror("Root must be bracketed in zbrent");
    fc=fb;
    for (iter=0;iter<ITMAX;iter++) {
        if ((fb > 0.0 && fc > 0.0) || (fb < 0.0 && fc < 0.0)) {
            c=a;                            对a,b,c更名并调整解区间d。
            fc=fa;
            e=d=b-a;
        }
        if (fabs(fc) < fabs(fb)) {
            a=b;
            b=c;
            c=a;
            fa=fb;
            fb=fc;
            fc=fa;
        }
        tol1=2.0*EPS*fabs(b)+0.5*tol;       收敛性检验。
        xm=0.5*(c-b);
        if (fabs(xm) <= tol1 || fb == 0.0) return b;
        if (fabs(e) >= tol1 && fabs(fa) > fabs(fb)) {
            s=fb/fa;                        欲进行二次反内插。
            if (a == c) {
                p=2.0*xm*s;
                q=1.0-s;
            } else {
                q=fa/fc;
                r=fb/fc;
                p=s*(2.0*xm*q*(q-r)-(b-a)*(r-1.0));
                q=(q-1.0)*(r-1.0)*(s-1.0);
            }
            if (p > 0.0) q = -q;            检查是否在解区间内。
            p=fabs(p);
            min1=3.0*xm*q-fabs(tol1*q);
            min2=fabs(e*q);
            if (2.0*p < (min1 < min2 ? min1 : min2)) {
                e=d;                        可以进行内插。
                d=p/q;
            } else {
                d=xm;                       不能进行内插，改用二分法。
                e=d;
            }
        } else {                            界下降速度太慢，改用二分法。
            d=xm;
            e=d;
        }
        a=b;                                将最新估值赋给a。
        fa=fb;
        if (fabs(d) > tol1)                 计算新的试验解。
            b += d;
        else
            b += SIGN(tol1,xm);
            fb=func(b);
    }
    nrerror("Maximum number of iterations exceeded in zbrent");
    return 0.0;                             永远不能执行到此。
}
```

参考文献和深入阅读

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 3, 4. [1]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §7.2.

9.4　利用导数的 Newton-Raphson 方法

在所有一维的求根方法中，最著名的是**牛顿方法**，也称为 Newton-Raphson（牛顿－拉斐森）方法。这种方法与前面几节所述方法的区别在于，它既要利用任意点 x 处的函数值 $f(x)$，还需计算其导数值 $f'(x)$。从几何上解释，Newton-Raphson 方法是将当前点 x_i 处的切线延长，使之与横轴相交，然后置下一估计值 x_{i+1} 为交点处的横坐标值（参见图 9.4.1）

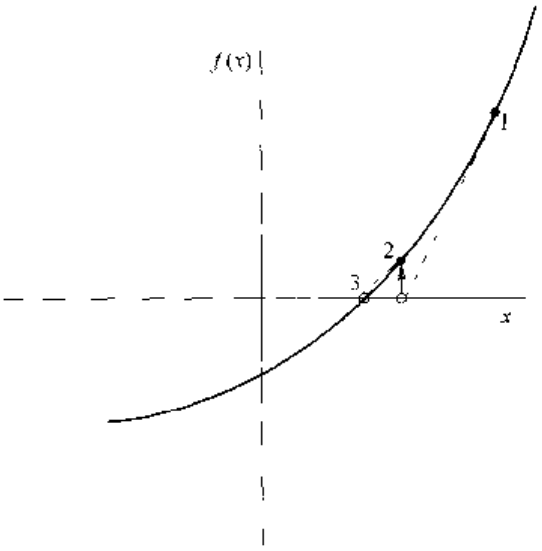


图 9.4.1　用牛顿方法将局部导数外推以寻求根的下一估计值。对本图所示的函数，牛顿方法的运行状态良好，结果达到了二次收敛

从代数上解释，该方法的推导，利用了我们所熟悉的在某点附近函数的泰勒级数展开，即

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \dots . \tag{9.4.1}$$

对足够小的 δ 以及性能良好的函数，上式中二次及二次以上的项均可忽略不计，因此由 $f(x+\delta)=0$ 可导出

$$\delta = -\frac{f(x)}{f'(x)} \tag{9.4.2}$$

Newton-Raphson 方法并不限于一维问题，正如在 9.6 节和 9.7 节中将要讨论的，它可推广至多维情形。在离根较远的地方，展开式中的高阶项不能忽略，这时，Newton-Raphson 方法可以粗略地给出一些非精确的而且没有什么特殊意义的修正项。例如，为了让搜索区间包含函数的某个局部极大值或极小值，根的初始预测可能需置在离真正的根很远的地方。这种情况有可能会导致求根失败（参见图 9.4.2）。如果某次迭代求出的估计值位于这样的局部极值附近，这时一阶导数几乎变为零，方程的根将"丢失"而且很难再找到。像绝大多数功能强大的算法工具一样，在不适当的场合，Newton-Raphson 方法也可以是破坏性的。图 9.4.3 所示的是另外一种可能的病态情况。

为什么还要说 Newton-Raphson 是一种很有效的算法呢？原因在于其收敛的速率：在点 x 的一个很小的邻域范围 ϵ 内，函数及其导数可近似地表示为

$$\begin{aligned} f(x+\epsilon) &= f(x) + \epsilon f'(x) + \epsilon^2 \frac{f''(x)}{2} + \cdots , \\ f'(x+\epsilon) &= f'(x) + \epsilon f''(x) + \cdots \end{aligned} \tag{9.4.3}$$

由 Newton-Raphson 公式

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \tag{9.4.4}$$

有

$$\epsilon_{i+1} = \epsilon_i - \frac{f(x_i)}{f'(x_i)} \tag{9.4.5}$$

如果估计值 x_i 与真值相差ϵ_i，则可利用(9.4.3)式，用ϵ_i 和根本身的导数来表示(9.4.4)式中的 $f(x_i)$ 及 $f'(x_i)$，这个结果是一个关于估计值导数的递推关系式，即

$$\epsilon_{i+1} = -\epsilon_i^2 \frac{f''(x)}{2f'(x)} \tag{9.4.6}$$

(9.4.6)式说明，Newton-Raphson 方法是**二次收敛算法**，参考(9.2.3)式. 在根附近，每迭代一步，有效数字的位数几乎**翻倍**。这种很强的收敛性质，使得 Newton-Raphson 方法非常适合于那些易计算，并且在根附近有连续且非零的导数的函数。

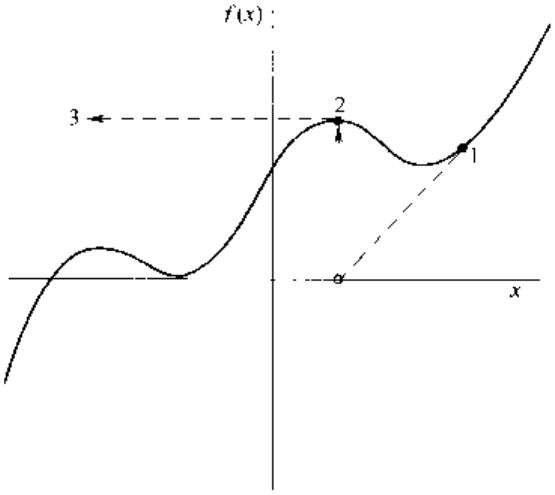


图 9.4.2 应用牛顿方法导致求根失败的示例。在迭代过程中遇到了局部极值，这时该点的切线将射向外部空间，永远无法与横轴相交。这里若用划界方法确定解区间(如 *rtsafe* 那样)，就可以避免这种情况的发生

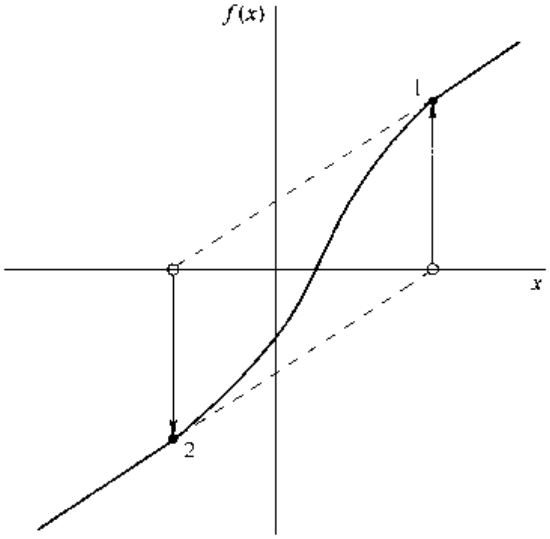


图 9.4.3 应用牛顿方法导致死循环的示例。当函数 f 全部或部分地由表插值得到时，常常会遇到这种情况，但若对初值预测得稍好一些，该例仍能求出真解

即使在收敛的最初几步中，Newton-Raphson 方法就被否定了（由于其较差的整体收敛性质），但是人们却经常在根的修正过程中，还穿插一两步这种方法，因为它可以使有效数字的位数两倍或四倍地增加。

实现 Newton-Raphson 方法时需由用户提供一个程序，用以计算函数及其一阶导数在任意点 x 处的值 $f(x)$ 和 $f'(x)$。Newton-Raphson 公式还有一个应用，即以一个数值差分来近似真值的局部导数：

$$f'(x) \approx \frac{f(x+dx)-f(x)}{dx} \tag{9.4.7}$$

但是，并不提倡这种方法，因为：(i)每步要做两次函数值的计算，因此，超线性收敛的阶最多只能达到$\sqrt{2}$；(ii)如果 dx 取得太小就会因四舍五入而被略去，而如果取得太大，收敛的阶又只能达到线性量级，这样还不如在随后的所有步骤中都采用最初计算出来的 $f'(x_0)$。因此，利用数值导数的 Newton-Raphson 方法（在一维情况）通常要让位于 9.2 节中的弦截法（在多维情况，由于目前提供的方法很少，对待利用导数的 Newton-Raphson 方法需要更加慎重，详见 9.6 节至 9.7 节）。

下面的程序中调用一个由用户提供的函数 funcd(x,fn,df)，其中 fn 为函数值，df 为导数值。这里我们包含一个对根的输入区间，目的只是为了与前面的求根程序相一致。牛顿方法不是对这个区间进行调整，而只用到点 x 附近的局部信息。这个输入区间只有两个用途，一是选它的中点作为迭代第一步的预测值；二是用于对跨到区间外的解做出放弃的决定。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::rtnewt(void funcd(const DP, DP &, DP &), const DP x1, const DP x2,
    const DP xacc)
已知位于区间[x1,x2]内的一个函数，用 Newton-Raphson 方法求它的根。这个根 rtnewt 将不断地被修正，直到其精度达到
±xacc 内为止。funcd 是一个用户提供的程序，求在任意点 x 处的函数值及其一阶导数值。
{
    const int JMAX=20;                          设置最大迭代次数。
    int j;
    DP df,dx,f,rtn;
    rtn=0.5*(x1+x2);                            初始预测。
    for (j=0;j<JMAX;j++) {
        funcd(rtn,f,df);
        dx=f/df;
        rtn -= dx;
        if ((x1-rtn)*(rtn-x2) < 0.0)
            nrerror("Jumped out of brackets in rtnewt");
        if (fabs(dx) < xacc) return rtn;        收敛。
    }
    nrerror("Maximum number of iterations exceeded in rtnewt");
    return 0.0;                                 永远不能执行到此处。
}
```

虽然 Newton-Raphson 方法的整体收敛性不够好，但是将它与二分法结合，设计一个失效保护程序却相对较容易。这种混合算法的基本思想是，一旦 Newton-Raphson 方法在运行过程中出现解跨出解区间或者解区间长度的减小速度不够快的情况，就即刻采取一步二分法

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::rtsafe(void funcd(const DP, DP &, DP &), const DP x1, const DP x2,
    const DP xacc)
用 Newton-Raphson 方法和二分法相结合的方法求一个方程的根，这个根位于解区间[x1,x2]内，在其精度达到 ±xacc 之前，
这个根将不断地被修正，并最终以 rtsafe 返回。funcd 是一个由用户提供的程序，返回函数值及函数的一阶导数值。
{
    const int MAXIT=100;                        允许最大迭代次数。
    int j;
    DP df,dx,dxold,f,fh,fl,temp,xh,xl,rts;

    funcd(x1,fl,df);
    funcd(x2,fh,df);
    if ((fl > 0.0 && fh > 0.0) || (fl < 0.0 && fh < 0.0))
        nrerror("Root must be bracketed in rtsafe");
```

```
    if (fl == 0.0) return x1;
    if (fh == 0.0) return x2;
    if (fl < 0.0) {                          确定搜索方向以使f(x1)<0。
        xl=x1;
        xh=x2;
    } else {
        xh=x1;
        xl=x2;
    }
    rts=0.5*(x1+x2);                         初始化根的估值、倒数第二步的步长以及
    dxold=fabs(x2-x1);                       最后一步的步长。
    dx=dxold;
    funcd(rts,f,df);
    for (j=0;j<MAXIT;j++) {                  在允许迭代数内循环。
        if ((((rts-xh)*df-f)*((rts-xl)*df-f) > 0.0)     若牛顿方法超出解区间或收敛
            || (fabs(2.0*f) > fabs(dxold*df))) {        速度不够快，则采用二分法。
            dxold=dx;
            dx=0.5*(xh-xl);
            rts=xl+dx;
            if (xl == rts) return rts;       根的变化很小可忽略。
        } else {                             可以采用牛顿方法，取牛顿步长
            dxold=dx;
            dx=f/df;
            temp=rts;
            rts -= dx;
            if (temp == rts) return rts;
        }
        if (fabs(dx) < xacc) return rts;     收敛准则。
        funcd(rts,f,df);
        每迭代一次都要重新计算函数及其导数值。
        if (f < 0.0)                         使根保持在解区内。
            xl=rts;
        else
            xh=rts;
    }
    nrerror("Maximum number of iterations exceeded in rtsafe");
    return 0.0;                              永远不能执行到此。
}
```

对很多函数来说，导数 $f'(x)$ 常常先于函数 $f(x)$ 收敛到机器精度，在这种情况下不需要连续更新 $f'(x)$。建议读者在熟悉了函数的一般性质后再采用这个小技巧。但如果导数计算很繁琐，这种方法倒是可以加快计算速度(形式上，这只能使收敛达到线性量级，但是若导数无论如何都不变化的话，收敛性也仅如此)。

9.4.1 Newton-Raphson 方法和分形

提醒读者，在 Newton-Raphson 方法的全局收敛性无法预知的同时(尽管其局部收敛非常迅速)，探讨一下某些特殊方程迭代的初值集合也是有益的，从这个初值集合出发，算法既可能收敛到方程的根，也有可能收敛不到方程的根。

考虑如下这个简单的方程：

$$z^3 - 1 = 0 \tag{9.4.8}$$

它的实数根只有一个，即 $z=1$。其另外两个复数根是单位立方根，分别为 $\exp(\pm 2\pi i/3)$。由牛顿方法导出的迭代公式为

$$z_{j+1} = z_j - \frac{z_j^3 - 1}{3z_j^2} \tag{9.4.9}$$

至今为止，仅仅是将(9.4.9)式这样的迭代方程作用于实的初始值 z_0，但实际上本节中所有方程都可应用于复平面。这样，就可以将复平面划分为一些区域，从这些区域出发，对某个初值 z_0 用(9.4.9)式进行迭代，其结果或不收敛，或收敛到 $z=1$。可能会很自然地希望找到一个“收敛域”，该收敛域以某种方式包围 $z=1$ 这个根。当然不希望收敛域填满整个平面，因为平面内还必须包含收敛到两个复根的区域。实际上根据对称性，这三个区域应该具有相同的形状(可能是三个对称的 120° 楔形，每个楔形的中心有一个根)。

图 9.4.4 展示的是经次数值外推的结果。收敛域的确覆盖了整个复平面的 1/3 区域，但它的边界可以说是极其不规则的，更确切地讲是呈分形(fractal)，所谓分形就是它具有自身相似的结构，而且这种结构又以各种大小比例连续排列成形。如此高度复杂的分形是如何从牛顿方法以及(9.4.8)式这样简单的方程中演变而来的呢？其实问题的答案早已隐含在图 9.4.2 中了，因为该图展示了某个局部极小点是如何导致牛顿方法趋向于无穷远处的。设想从这个极小点处稍微挪开一段极小的距离，那么等待我们的将不会再是无穷远处的外部空间。如果幸运的话，也许能够恰好进入包含所求根的收敛域。但这意味着在这个极值的领域内，必须存在一个"收敛域"的微型"复制品"，这个"复制品"可能产生了部分变形，具有某种"一级反射"的性质。同理可以推断，后面的迭代过程还会产生"二级反射"、"三级反射"等诸如此类的近似收敛域的微型复制品。这就是为什么会出现分形的原因。

注意到对(9.4.8)式来说，几乎整个实轴都处在根 $z=1$ 的收敛域内。之所以对"整个实轴"冠以"几乎"二字，是因为位于实负半轴上的一些特殊离散点，它们的收敛性是不确定的(参见图 9.4.4)。如果从它们当中的某点开始执行牛顿方法，结果会如何呢？读者不妨试试看。

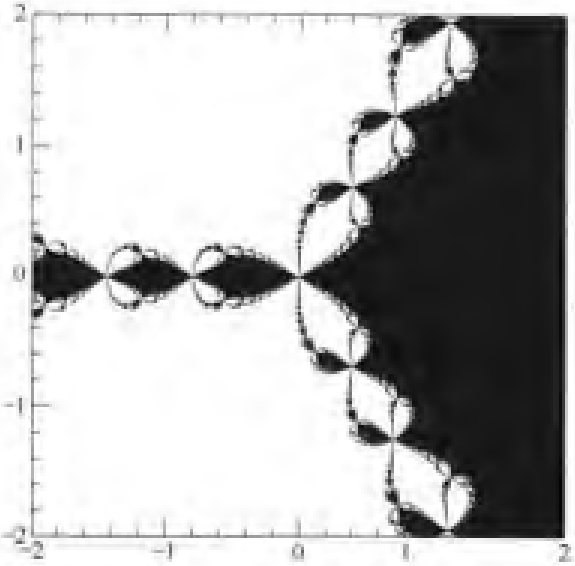


图 9.4.4 图中显示的是实轴和虚轴均在$(-2,2)$范围内的复平面 z，其中黑色区域为某些具有特殊性质的点的集合，从这些点出发，牛顿方法将收敛到方程$z^3-1=0$的根$z=1$。黑色区域的形状就是所谓的分形

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §8.4.

Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press).

Mandelbrot, B.B. 1983, *The Fractal Geometry of Nature* (San Francisco: W.H. Freeman).

Peitgen, H.-O., and Saupe, D. (eds.) 1988, *The Science of Fractal Images* (New York: Springer-Verlag).

9.5 多项式的根

本节将介绍几种多项式求根的方法。这些方法适用于求解绝大多数实际问题，包括阶数不太高的多项式以及性能良好的较高阶多项式的求根问题等。但遗憾的是，仍有一些多项式的性质极差，而且在最坏情况下，甚至多项式系数的最小扰动都可能导致多项式的各个根零乱地散布到整个复平面上。Wilkinson 构造了一个这样的例子，详阅 Acton[1]。

我们知道，n 阶多项式有 n 个根，这些根可以是实的也可以是复的，而且它们当中可能有重根。如果多

项式系数是实数,则它的复数根以共轭对的形式出现,也就是说,若 $x_1 = a + bi$ 是该多项式一个根,则 $x_2 = a - bi$ 必然也是其根。对系数是复数的多项式,其复根之间就不一定有关联了。

多重根或根相近的情况给数值算法带来了极大的不便(参见图 9.5.1)。例如,$P(x) = (x - a)^2$ 在 $x = a$ 处有一个二重实根,但我们既不能用通常的方法确定解区间,即判断函数在哪些邻域内改变符号,也不能用像 Newton-Raphson 那样的斜率跟踪法,因为在重根处函数及其导数都将变为零。由于可能会出现大的舍入误差,Newton-Raphson 方法即使是可行的,其速度也相当慢。如果预先能知道某个根是重根,则可以毫不费力地设计出特殊的处理方法。但在通常情况下,我们并不能预先判断出根将会出现什么样的病态情况,这样便产生了各种各样的问题。

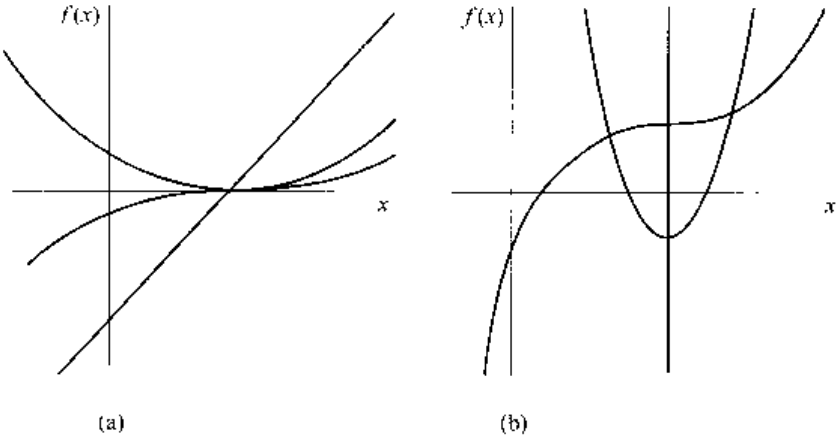


图 9.5.1 多项式的根。(a) 线性、二次及三次多项式在其根处的情况;(b) 只有将(a)高倍率放大才可以清楚地看到,二次多项式只有一个根而不是三个根,三次多项式有两个根,而不是没有根

9.5.1 多项式的降阶

在求多项式的某几个根或所有根时,使用**降阶**方法可大大减少计算量。每求得一个多项式的根 r,便可以将该多项式因式分解为一个含根 r 的项乘以一个比原多项式低一阶的多项式,即 $P(x) = (x - r)Q(x)$。由于 $Q(x)$ 的根是 $P(x)$ 根的子集,因此求根的计算量减小了。在相继求出 $P(x)$ 各根的同时,多项式的阶数也越来越低。更重要的是,用降阶方法可以避免一个严重错误,即可避免迭代算法两次收敛到同一个根,而这个根并不是重根,因为这种情况下正确的结果应是分别收敛到两个不同的根。

相当于综合除法的降阶方法可以看成是对多项式系数阵列的简单操作。在前面的 5.3 节中曾给出了通过单项式因子进行综合除法的源程序代码,对复数根的降价,可以把该程序改写为适合于复数的类型。对实系数多项式可能有复根的情况,也可以通过一个二次因子来降阶,即:

$$[x - (a + ib)][x - (a - ib)] = x^2 - 2ax + (a^2 + b^2) \tag{9.5.1}$$

在 5.3 节中的程序 poldiv 可用于多项式除以上述因子。

但是,对降阶方法必须谨慎使用。因为新求出的每个根都只有有限的精确度。在确定逐次降阶的多项式的系数时就不知不觉地产生了累计的误差,结果会造成求出的根越来越不精确。无论这种不精确度是稳定地增长(即每步增加或减少机器精度的若干倍),还是不稳定地增长(即有效数字不断减少,直到结果变得毫无意义)都是关系重大的。究竟取哪一种情况仅取决于根被约去的方式。向前降阶在 5.3 节中已经介绍过,这种方式中新多项式系数的计算,是从 x 的最高幂项开始顺序下去直到常数项。可以证明,如果在每一步中总是绝对值最小的根被约去,则这种方法是稳定的。与之相反的**向后降阶**在计算新多项式的系数时,是从常数项上升至 x 的最高次幂项。如果每步中总是绝对值**最大**的根被约去,则向后降阶的方法也是稳定的。

如果把一个多项式的系数"首—尾"交换(即把常数项变成最高幂项系数等),则该多项式的根将被映射为其倒数(证明:整个多项式被其最高阶 x^n 除,然后以 $1/x$ 为变量将结果重写为一个新的多项式形式)。因此,向后降阶算法与向前降阶算法实际上是等同的,只是原多项式的系数本身就是逆序的,并且使用了降阶根的倒数。因为下面将要使用向前降阶,所以向后降阶算法将留给读者作为练习(和 5.3 节中一样)。有关

降阶方法稳定性的进一步探讨，请参见本节参考文献[2]。

使用降阶方法时，为了使累积误差(包括稳定的误差)的影响达到最小，不妨把逐次降阶后的多项式的根看做是原多项式的**试验根**，然后以这些试验根作为初始预测值并进行修正，其中这些初始值将利用**没有降阶**的原多项式 P 重新求解。这里同样需要特别小心，以免用降阶法求出的两个根因精度过低而导致被修正后都收敛到同一个根(这个根可不通过降阶求得)。这时会出现伪重根和丢根的情况，但幸运的是，我们可以对这种情况进行监测，不妨对该多项式只进行一次降阶(而且只对这个根)，然后再修正这个试探根。也可以通过 Maehly 方法(参见下面的(9.5.29)式)来解决。

下面将进一步讨论有关修正实根和共轭复根的技巧。首先，我们来看看一般策略。

实系数多项式的求根方法可分为两大类。一类是，设法获得最容易获得的目标，即相异的实根，在求根过程中采用与一般函数求根法基本相同的方法(前节已讨论过)，也就是先用试位方式确定解区间，然后，再利用比较保险的 Newton-Raphson 方法(如程序 rtsafe)。有些情况下，可能只对多项式的实数根感兴趣，这时第一类方法就足以应付自如了。实系数多项式求根方法的第二类方法是，利用形如(9.5.1)式的二次因子。Bairstow 方法就属于这一类，将在下文有关根修正的小节中进行讨论。另外还有一种由 Muller 提出的方法，下面简要介绍。

9.5.2　Muller 方法

Muller 方法是弦截法的推广，但它采用的是三点之间的二次内插，而不是两点间的线性内插。此外二次式求根允许用此算法求解复根对。若给定根的三个估计值 x_{i-2}, x_{i-1}, x_i 及多项式 $P(x)$ 在这些点处的值，则根的下一估计值 x_{i+1} 可由下列公式计算得到：

$$
\begin{aligned}
q &\equiv \frac{x_i - x_{i-1}}{x_{i-1} - x_{i-2}} \\
A &\equiv qP(x_i) - q(1+q)P(x_{i-1}) + q^2P(x_{i-2}) \\
B &\equiv (2q+1)P(x_i) - (1+q)^2P(x_{i-1}) + q^2P(x_{i-2}) \\
C &\equiv (1+q)P(x_i)
\end{aligned}
\tag{9.5.2}
$$

则有

$$
x_{i+1} = x_i - (x_i - x_{i-1})\left[\frac{2C}{B \pm \sqrt{B^2 - 4AC}}\right] \tag{9.5.3}
$$

上式中分母内正负号的选取原则是使分母的绝对值或模尽可能大。迭代时初始值可以随意选取三个数，例如取实轴上等间隔的三个点。值得注意的是，Muller 方法在实现过程中，必须允许复数分母的出现以及随之带来的复数运算。

有时候 Muller 方法也被用于在复平面上求解解析函数(不仅限于多项式)的复根，如在 IMSL 程序 ZANLY[3]中。

9.5.3　拉盖尔方法

本书所采用的方法属于实系数多项式求根方法的第二类。这类算法的数目极少，而且收敛速度有快有慢，但每种算法都可以收敛到各种类型的根：实根、复根、单根及重根。可以先用这类算法求得 n 阶多项式所有 n 个根的试验值，然后再回来一一修正这些值。

在这些一般的复数算法中，**拉盖尔(Laguerre)方法**是至今为止最直接的方法。尽管它甚至在收敛到实根的情况下也需要进行复数运算，但对于所有根都是实数的多项式来讲，它能确保从任意初始点开始收敛到某个根。然而，对于具有复根的多项式，有关拉盖尔方法的收敛性证明在理论上几乎没有。但不少经验表明，不收敛的情况出现的次数极小，而且几乎总是可以通过冲破某个有限的非收敛圈来解决(这种解决方法在下面的程序中有具体的实现步骤)。例如，对于一个高阶(≥ 20)多项式的示例，它的全部根恰好位于复数单位圆之外，并且基本上呈等距离分布。当算法收敛到零这个简单的复根时，可知其收敛阶为三次。

在某些情况下，由于多项式本身可能具有复数系数，因此拉盖尔方法对复数运算并不会带来不便。

为推导拉盖尔公式(虽然不是严格的推导)，可记多项式与其根及导数的关系如下所示：

$$P_n(x) = (x - x_0)(x - x_1)\ldots(x - x_{n-1}) \tag{9.5.4}$$

$$\ln|P_n(x)| = \ln|x - x_0| + \ln|x - x_1| + \ldots + \ln|x - x_{n-1}| \tag{9.5.5}$$

$$\frac{d\ln|P_n(x)|}{dx} = +\frac{1}{x - x_0} + \frac{1}{x - x_1} + \ldots + \frac{1}{x - x_{n-1}} = \frac{P_n'}{P_n} \equiv G \tag{9.5.6}$$

$$\begin{aligned} -\frac{d^2\ln|P_n(x)|}{dx^2} &= +\frac{1}{(x - x_0)^2} + \frac{1}{(x - x_1)^2} + \ldots + \frac{1}{(x - x_{n-1})^2} \\ &= \left[\frac{P_n'}{P_n}\right]^2 - \frac{P_n''}{P_n} \equiv H \end{aligned} \tag{9.5.7}$$

由这些关系式开始,拉盖尔公式构造了 Acton[1]称为的"一个非常有力的假设集":假设所求的根 x_0 与当前估计值 x 的距离为 a,而其他根与 x 的距离为 b,即

$$x - x_0 = a \quad ; \quad x - x_i = b \quad i = 1, 2, \ldots, n-1 \tag{9.5.8}$$

则(9.5.6)式及(9.5.7)式可分别表示为

$$\frac{1}{a} + \frac{n-1}{b} = G \tag{9.5.9}$$

$$\frac{1}{a^2} + \frac{n-1}{b^2} = H \tag{9.5.10}$$

从而解得 a 的表达式为

$$a = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}} \tag{9.5.11}$$

其中正负号的选取要使分母的数值达到最大。由于上式中平方根一项有可能是负的,因此 a 可以为复数。有关(9.5.11)式的严格推导,参见本节参考文献[4]。

拉盖尔方法是以迭代方式进行的,即对某个试验值 x,用(9.5.11)式计算 a,以 $x - a$ 作为下一个试验值,重复以上步骤直到 a 足够小为止。

下面是拉盖尔方法的程序实现,该程序用来求解某个给定的 m 阶复系数多项式的一个根。这里和前面一样,第一个 a[0]将用于存储常数项,而 a[m]则存放 x 的最高阶项系数。拉盖尔算法采用简化的 Adams[5]终止准则,该准则在以下两方面进行了巧妙的权衡,即一方面要保证足够的机器精度,另一方面又要考虑由舍入误差引起的无穷迭代的危险性。

```
#include <cmath>
#include <complex>
#include <limits>
#include "nr.h"
using namespace std;

void NR::laguer(Vec_I_CPLX_DP &a, complex<DP> &x, int &its)
```

给定多项式 $\sum_{i=0}^{m} a[i]x^i$ 的 m+1 个复系数 a[0..m],同时给定一个复数 x,本程序将用拉盖尔方法不断修正这个 x,直到它在可以达到的舍入极限范围内,收敛到多项式的某个根为止。所迭代的次数将以 its 返回。

```
{
    const int MR=8,MT=10,MAXIT=MT*MR;
    const DP EPS=numeric_limits<DP>::epsilon();
```

EPS 为预先给定的舍入误差。在 MT 步的每一步中以 MR 个不同的分数值冲破限制圈,这里允许迭代的最多次数为 MAXIT。

```
    static const DP frac[MR+1]=
        {0.0,0.5,0.25,0.75,0.13,0.38,0.62,0.88,1.0};
```

分数用于冲破限制圈。

```
    int iter,j;
    DP abx,abp,abm,err;
    complex<DP> dx,x1,b,d,f,g,h,sq,gp,gm,g2;

    int m=a.size()-1;
    for (iter=1;iter<=MAXIT;iter++) {
```

在允许的迭代次数范围内做循环迭代。

```
        its=iter;
        b=a[m];
        err=abs(b);
        d=f=0.0;
        abx=abs(x);
        for (j=m-1;j>=0;j--) {              多项式及其两个一阶导数的快速计算，f存
            f=x*f+d;                        储P''/2。
            d=x*d+b;
            b=x*b+a[j];
            err=abs(b)+abx*err;
        }
        err *= EPS;
        计算多项式值进估计舍入误差。
        if (abs(b) <= err) return;          已找到根，返回。
        g=d/b;                              一般情形：利用Laguerre公式。
        g2=g*g;
        h=g2-2.0*f/b;
        sq=sqrt(DP(m-1)*(DP(m)*h-g2));
        gp=g+sq;
        gm=g-sq;
        abp=abs(gp);
        abm=abs(gm);
        if (abp < abm) gp=gm;
        dx=MAX(abp,abm) > 0.0 ? DP(m)/gp : polar(1+abx,DP(iter));
        x1=x-dx;
        if (x == x1) return;        已收敛。
        if (iter % MT != 0) x=x1;
        else x -= frac[iter/MT]*dx;
        偶尔采取一次冲破限制圈的步骤(这种情况极少发生)。
    }
    nrerror("too many iterations in laguer");
    极不常见——只有复根才可能发生。不妨改变一下初值。
    return;
}
```

下面的驱动程序通过连续调用函数 laguer 求解多项式的每一个根。该程序首先对多项式进行降阶，然后以同样的拉盖尔方法(如果不打算用其他方法的话)随意地对根进行修正，最后将它们按其实部排序(在第 13 章中将利用这个驱动程序)。

```
#include <cmath>
#include <complex>
#include "nr.h"
using namespace std;

void NR::zroots(Vec_I_CPLX_DP &a, Vec_O_CPLX_DP &roots, const bool &polish)
给定多项式∑_{i=0}^{m} a(i)x^i 的 m+1 个复系数 a[0..m]。本程序通过连续调用 laguer 以求解多项式的所有 m 个复根，并将它
们存储在数组 roots[0..m-1]中，如果根需要修正(仍用拉盖尔方法)，则逻辑变量 polish 的输入值应为 true；如果用其他方
法，则输入值应为 false。
{
    const DP EPS=1.0e-14;                   一个很小的数。
    int i,its,j,jj;
    complex<DP> x,b,c;

    int m=a.size()-1;
    Vec_CPLX_DP ad(m+1);
    for (j=0;j<=m;j++) ad[j]=a[j];          对系数进行复制，以备逐次降价用。
    for (j=m-1;j>=0;j--) {                  对每一个欲求的根循环。
        x=0.0;                              从零点开始以利于收敛到最小的一个剩余根，
        Vec_CPLX_DP ad_v(j+2);              并找这个根。
        for (jj=0;jj<j+2;jj++) ad_v[jj]=ad[jj];
        laguer(ad_v,x,its);
        if (fabs(imag(x)) <= 2.0*EPS*fabs(real(x)))
            x=complex<DP>(real(x),0.0);
        roots[j]=x;
        b=ad[j+1];                          向前降价。
        for (jj=j;jj>=0;jj--) {
```

```
            c=ad[jj];
            ad[jj]=b;
            b=x*b+c;
        }
    }
    if (polish)
        for (j=0;j<m;j++)                    用原来未降阶的系数修正各根。
            laguer(a,roots[j],its);
    for (j=1;j<m;j++) {                      通过直接插入法对根的实部排序。
        x=roots[j];
        for (i=j-1;i>=0;i--) {
            if (real(roots[i]) <= real(x)) break;
            roots[i+1]=roots[i];
        }
        roots[i+1]=x;
    }
}
```

9.5.4 本征值方法

一个矩阵 **A** 的本征值是"特征多项式"$P(x)=\det[\mathbf{A}-x\mathbf{I}]$的根，但正如在第 11 章中将要看到的，通过求根方法求本征值，一般来讲并不是一种有效的手段。换一个角度来考虑，可用更有效的本征值方法(第 11 章中讨论的方法)求任意多项式的根。极易证明(参见本节参考文献[6])特殊 $m\times m$ 阶**伴随矩阵**

$$\mathbf{A}=\begin{pmatrix} -\frac{a_{m-1}}{a_m} & -\frac{a_{m-2}}{a_m} & \cdots & -\frac{a_1}{a_m} & -\frac{a_0}{a_m} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \tag{9.5.12}$$

的特征多项式与一般形式的多项式

$$P(x)=\sum_{i=0}^{m} a_i x^i \tag{9.5.13}$$

完全等价。如果系数 a_i 是实数而不是复数，则 **A** 的本征值可以通过 11.5 节～11.6 节中的程序 balanc 和 hqr 求得(详见第 11 章)。尽管本征值法的程序 zrhqr(实现见下)的实现通常是 zroots(见上)的 1/2，但对某些类型的多项式来说，本征值法更能解决问题，这主要是因为 hqr 中的收敛算法太繁琐。在多项式系数为实数的情况下，如果用 zroots 求根不太顺利，建议读者不妨试试下面的 zrhqr。

```
#include <complex>
#include "nr.h"
using namespace std;

void NR::zrhqr(Vec_I_DP &a, Vec_O_CPLX_DP &rt)
在给定实系数多项式∑_{i=0}^{m} a(i)x^i 的系数 a[0..m]的条件下，求解它的所有根。采用的方法是先构造一个较高阶的 Hessenberg
矩阵(其本征值即为所求根)。然后再利用程序 balanc 和 hqr。根的实部和虚部将分别以rtr[0..m-1]和 rti[0..m-1]返回。
{
    int j,k;
    complex<DP> x;

    int m=a.size()-1;
    Mat_DP hess(m,m);
    for (k=0;k<m;k++) {          构造矩阵。
        hess[0][k] = -a[m-k-1]/a[m];
        for (j=1;j<m;j++) hess[j][k]=0.0;
        if (k != m-1) hess[k+1][k]=1.0;
    }
    balanc(hess);                求本征值。
    hqr(hess,rt);
    for (j=1;j<m;j++) {          用直接插入法对根按其内部进行排序。
        x=rt[j];
        for (k=j-1;k>=0;k--) {
            if (real(rt[k]) <= real(x)) break;
            rt[k+1]=rt[k];
        }
        rt[k+1]=x;
    }
}
```

9.5.5　其他可靠的求根方法

在多项式求根的黑箱方法中，如 IMSL 库[3]中的求根算法，**Jenkins-Traub 方法**已成为一种实用标准，但由于这种方法过于复杂，这里不做讨论。可参见本节参考文献[4]。

Lehmer-Schur 算法属于一类求根方法，这类方法将一维的划界概念加以推广，即把根孤立在复平面内，也就是在一个圆心和半径都给定的圆内，有效地确定是否含有多项式的根，然后，再对于何处放置试验圆做出一系列判断，以便分门别类地求出多项式的全部根，参见本节参考文献[1]中的论述。

9.5.6　根修正的技巧

一旦确定出根所在的区间范围，那么用 Newton-Raphson 方法求实数根是很行之有效的。如 5.3 节所述，多项式及其导数值可以很快地同时计算出来。对一个阶数为 n，系数为 c[0]..c[n]的多项式，下面这段小程序是 Newton-Raphson 方法的一个循环。

```
p=c[n]*x+c[n-1];
p1=c[n];
for(i=n-2;i>=0;i--) {
    p1=p+p1*x;
    p=c[i]+p*x;
}
if (p1 == 0.0) NR::nrerror("derivative should not vanish");
x -= p/p1;
```

修正了一个多项式的所有实根之后，就需要对所有复根进行修正，这时既可以用直接方法，也可通过二次因子。

若将上面那段小程序改写为复数类型的数据，则 Newton-Raphson 方法即为修正复根的直接方法。对实系数的多项式，在设置初始预测值(试验解)时，应注意使其离开实轴，否则迭代时将永远无法脱离开实轴，而且可能会在多项式的极大或极小值处射向无穷远。

对实多项式，修正复根(这时也可称为重实根)的方法是 **Bairstow 方法**，这种方法要求寻找二次因子，其好处在于避免了所有的复数运算。具体地说，也就是找一个二次因子，使其包含两个根 $x = a \pm ib$，即

$$x^2 - 2ax + (a^2 + b^2) \equiv x^2 + Bx + C \tag{9.5.14}$$

一般地，如果以一个二次因子去除某个多项式，则剩下的将是一个线性项

$$P(x) = (x^2 + Bx + C)Q(x) + Rx + S \tag{9.5.15}$$

如果给定 B 和 C，则由多项式除法(参见 5.3 节)容易求得 R 及 S。我们可视 R 和 S 为 B 和 C 的可调整函数，如果这个二次因子为零则 R 和 S 均为零，在某个领域内，因 B 和 C 的微小变化而引起的 R 和 S 的改变可以用一阶泰勒级数展开来近似，即

$$R(B + \delta B, C + \delta C) \approx R(B, C) + \frac{\partial R}{\partial B}\delta B + \frac{\partial R}{\partial C}\delta C \tag{9.5.16}$$

$$S(B + \delta B, C + \delta C) \approx S(B, C) + \frac{\partial S}{\partial B}\delta B + \frac{\partial S}{\partial C}\delta C \tag{9.5.17}$$

为了计算偏导数，考虑(9.5.15)式对 C 的导数。因为 $P(x)$ 为一固定的多项式，故与 C 无关，从而有

$$0 = (x^2 + Bx + C)\frac{\partial Q}{\partial C} + Q(x) + \frac{\partial R}{\partial C}x + \frac{\partial S}{\partial C} \tag{9.5.18}$$

移项可得

$$-Q(x) = (x^2 + Bx + C)\frac{\partial Q}{\partial C} + \frac{\partial R}{\partial C}x + \frac{\partial S}{\partial C} \tag{9.5.19}$$

同理，由于 $P(x)$ 与 B 无关，因而将(9.5.15)式对 B 进行微分可以得到

$$-xQ(x) = (x^2 + Bx + C)\frac{\partial Q}{\partial B} + \frac{\partial R}{\partial B}x + \frac{\partial S}{\partial B} \tag{9.5.20}$$

比较(9.5.19)式和(9.5.15)式可以发现，它们在形式上完全匹配。因此，如果对 $P(x)$ 进行二次综合除法，即除以 $Q(x)$ 后得到 $R_1x + S_1$，则有

$$\frac{\partial R}{\partial C} = -R_1 \qquad \frac{\partial S}{\partial C} = -S_1 \tag{9.5.21}$$

为了求出偏导数的表达式,在二次因子的两个根 x_+ 和 x_- 处计算(9.5.20)式的值。由于

$$Q(x_\pm) = R_1 x_\pm + S_1 \tag{9.5.22}$$

我们有

$$\frac{\partial R}{\partial B} x_+ + \frac{\partial S}{\partial B} = -x_+(R_1 x_+ + S_1) \tag{9.5.23}$$

$$\frac{\partial R}{\partial B} x_- + \frac{\partial S}{\partial B} = -x_-(R_1 x_- + S_1) \tag{9.5.24}$$

求解这两个关于偏导数的方程,并利用

$$x_+ + x_- = -B \qquad x_+ x_- = C \tag{9.5.25}$$

可以得到

$$\frac{\partial R}{\partial B} = BR_1 - S_1 \qquad \frac{\partial S}{\partial B} = CR_1 \tag{9.5.26}$$

现在的 Bairstow 方法包括在二维情况下,用 Newton-Raphson 方法(这正是下节中将要讨论的主要内容),求使 R 和 S 同时为零的解。在每次循环中,需使用两次综合除法来算 R, S 及其对 B, C 的偏导数,和一维的 Newton-Raphson 方法一样,Bairstow 方法在某一对根(或实或复)附近收敛效果不错,但若从任一随机点开始迭代则可能导致失败。因此建议读者只有在对试验复根进行修正时才使用这一方法。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::qroot(Vec_I_DP &p, DP &b, DP &c, const DP eps)
给定一个 n 阶多项式的 n+1 个系数 p[0..n],以及一个二次因子 x*x+b*x+c 各系数的试验值,本程序将不断地修正
问题的解,直到系数 b,c 的变化幅度小于 eps 为止。程序中调用了 5.3 节中的 poldiv。
{
    const int ITMAX=20;                     最大迭代次数。
    const DP TINY=1.0e-14;
    int iter;
    DP sc,sb,s,rc,rb,r,dv,delc,delb;
    Vec_DP d(3);

    int n=p.size()-1;
    Vec_DP q(n+1),qq(n+1),rem(n+1);
    d[2]=1.0;
    for (iter=0;iter<ITMAX;iter++) {
        d[1]=b;
        d[0]=c;
        poldiv(p,d,q,rem);
        s=rem[0];                           第一次除法算 r,s。
        r=rem[1];
        poldiv(q,d,qq,rem);
        sb = -c*(rc = -rem[1]);             第二次除法求 r,s 对 c 的偏导数。
        rb = -b*rc+(sc = -rem[0]);
        dv=1.0/(sb*rc-sc*rb);               解 2×2 的方程。
        delb=(r*sc-s*rc)*dv;
        delc=(-r*sb+s*rb)*dv;
        b += (delb=(r*sc-s*rc)*dv);
        c += (delc=(-r*sb+s*rb)*dv);
        if ((fabs(delb) <= eps*fabs(b) || fabs(b) < TINY)
            && (fabs(delc) <= eps*fabs(c) || fabs(c) < TINY)) {
            return;                         系数收敛。
        }
    }
    nrerror("Too many iterations in routine qroot");
}
```

前面曾经提到过,如果两个试探解经修正后收敛到一个根,这将是非常麻烦的,因为这时并不知道在修正过程中是否丢失了一个根,也不知道是否真有一重根,而这个重根在前面的降阶步骤中仅仅由于舍入误差才被区分开。有一种解决办法是降阶并重新修正,但在修正阶段,需要避免的恰恰是降阶。这时,可以转而使用 **Maehly 方程**。Maehly 指出,对降了阶的多项式

$$P_j(x) \equiv \frac{P(x)}{(x - x_0)\cdots(x - x_{j-1})} \tag{9.5.27}$$

其导数可以写成

$$P_j'(x) = \frac{P'(x)}{(x-x_0)\cdots(x-x_{j-1})} - \frac{P(x)}{(x-x_0)\cdots(x-x_{j-1})}\sum_{i=0}^{j-1}(x-x_i)^{-1} \tag{9.5.28}$$

因而 Newton-Raphson 方法中，由估计值 x_k 推导下一个估计值 x_{k+1} 的公式可写为

$$x_{k+1} = x_k - \frac{P(x_k)}{P'(x_k) - P(x_k)\sum_{i=0}^{j-1}(x_k - x_i)^{-1}} \tag{9.5.29}$$

使用上述方程时，如果以 i 标记已修正过的根，则可以避免某个试验解收敛到另一试验解所对应的真根上，这是一个以所谓消零替代真降阶的例子。

此外，前面提到过的 Muller 方法也是一种值得使用的根修正法。

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 7. [1]

Peters G., and Wilkinson, J.H. 1971, *Journal of the Institute of Mathematics and its Applications*, vol. 8, pp. 16–35. [2]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [3]

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover), §8.9–8.13. [4]

Adams, D.A. 1967, *Communications of the ACM*, vol. 10, pp. 655–658. [5]

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §4.4.3. [6]

Henrici, P. 1974, *Applied and Computational Complex Analysis*, vol. 1 (New York: Wiley)

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §§5.5–5.9.

9.6　非线性方程组的 Newton-Raphson 方法

对于含一个以上线性方程的方程组，目前还没有一种好的一般性解法。这一断言虽然极端但却很实际，而且不难理解为什么将来也(很可能)不会有好的通用方法。考虑二维情况，即求同时满足

$$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0 \end{aligned} \tag{9.6.1}$$

的解，其中 f 和 g 是两个任意函数，每个函数都具有一些零等高线，这些零等高线把 (x,y) 平面分成若干区域，在这些区域内每个函数分别相应地取正值或负值。这里我们所感兴趣的仅仅是这些零等高线的边界，因为所要求的解(如果有的话)就是 f 和 g 之零等高线的交点(参见图 9.6.1)。但是一般情况下，f 和 g 相互之间根本没有任何联系。无论从 f 的角度还是从 g 的角度来分析，它们的公共点都没有什么特殊性质。为了找到所有的公共点，即求非线性方程组的解，(一般)需给出两个函数的全部零等高线。而且由于零等高线(一般)是由一些数目不定的不相交的闭曲线组成，那么，我们怎样才能知道全部闭曲线已经被找出来了呢？

对于两维以上的问题，需要找到同时属于 N 个不相关的零等值超平面的点，其中每个超平面的维数是 $N-1$ 维。如果没有相当出色的洞察力的话，求根简直是不可能的事。因为几乎总是需要利用一些附加的关于特定问题的特定信息，才能搞清楚诸如"该问题是不是能得到一个惟一解"，以及"解大约在什么位置"之类的疑问。Acton[1]对这些特殊技巧进行了一些尝试性的有益探讨。

本节中将介绍多维求根方法中最简单的一种，即 Newton-Raphson 方法，该方法对于足够好的初值，收敛效果很明显。在收敛的情况下能指出(虽然不是证明)假想的根不存在。在 9.7 节中将讨论复杂一些的 Newton-Raphson 方法，它有望对其全局收敛性质进行一番改进。在 9.7 节中，还将介绍一种称为 Broyden 方法，它是对弦截法的多维推广。

一般地，给定含有 N 个变量 $x_i\,(i=0,1,\ldots,N-1)$ 的 N 个函数方程：

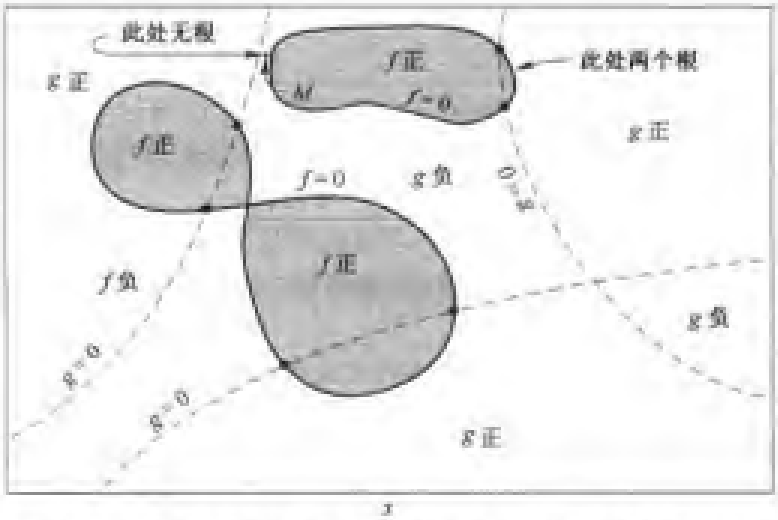


图9.6.1 两个二元非线性方程的解。其中实线代表$f(x,y)$,虚线为$g(x,y)$,每个方程将(x,y)平面分成若干个正的和负的区域,这些区域的边界即为零等高线,所要求的解即为这些不相关的曲线的交点,但解的个数事前是未知的

$$F_i(x_0,x_1,\ldots,x_{N-1})=0 \qquad i=0,1,\ldots,N-1 \tag{9.6.2}$$

记$\mathbf{x}$为以x_i为分量的向量,$\mathbf{F}$为以F_i为分量的向量,那么在$\mathbf{x}$的领域内,每个函数F_i可用泰勒级数展开为

$$F_i(\mathbf{x}+\delta\mathbf{x})=F_i(\mathbf{x})+\sum_{j=0}^{N-1}\frac{\partial F_i}{\partial x_j}\delta x_j+O(\delta\mathbf{x}^2) \tag{9.6.3}$$

出现在(9.6.3)式中的偏导数构成的矩阵是雅可比矩阵$\mathbf{J}$:

$$J_{ij}\equiv\frac{\partial F_i}{\partial x_j} \tag{9.6.4}$$

(9.6.3)式以矩阵形式表示,即为

$$\mathbf{F}(\mathbf{x}+\delta\mathbf{x})=\mathbf{F}(\mathbf{x})+\mathbf{J}\cdot\delta\mathbf{x}+O(\delta\mathbf{x}^2) \tag{9.6.5}$$

忽略$\delta\mathbf{x}^2$项及其更高阶项,并置$\mathbf{F}(\mathbf{x}+\delta\mathbf{x})=0$,可以得到一个关于修正项$\delta\mathbf{x}$的线性方程组,即

$$\mathbf{J}\cdot\delta\mathbf{x}=-\mathbf{F} \tag{9.5.6}$$

其中$\delta\mathbf{x}$可使每个函数都同时接近于零。

对(9.6.6)式所示的矩阵方程,用2.3节中介绍的LU分解方法来求解,并将求出的修正项添加到解向量中,

$$\mathbf{x}_{\text{new}}=\mathbf{x}_{\text{old}}+\delta\mathbf{x} \tag{9.6.7}$$

这一过程不断迭代直到收敛为止。一般情况下,最好检查一下函数和变量已收敛的程度,当其中一个达到机器精度时,另一个就不会改变了。

下面的程序mnewt将从初始解向量x[0..n-1]开始,执行ntrial次迭代。当函数F_i的大小之和小于某一容差限tolf,或者修正项δx_i的绝对值之和小于容差限tolx时,迭代过程将终止。mnewt中调用了一个由用户定义的函数usrfun,它提供函数值$\mathbf{F}$及雅可比矩阵$\mathbf{J}$。若$\mathbf{J}$的解析计算太复杂,不妨在usrfun中调用9.7节中介绍的fdjac,利用有限差分来计算偏导数,还应注意迭代次数ntrial不能取得太大,而且在进入下一轮迭代以前,最好观察一下当前的运行结果。

```
#include <cmath>
#include "nr.h"
using namespace std;

void usrfun(Vec_I_DP &x, Vec_O_DP &fvec, Mat_O_DP &fjac);

void NR::mnewt(const int ntrial, Vec_IO_DP &x, const DP tolx, const DP tolf)
```

给定一个n维根的初始猜测值x[0..n-1],通过ntrial步Newton-Raphson方法修正这个根。当各变量的绝对增量之和小于tolx或函数的绝对值之和小于tolf时,迭代终止。

```
{
    int k,i;
    DP errx,errf,d;
    int n=x.size();
    Vec_INT indx(n);
    Vec_DP p(n),fvec(n);
    Mat_DP fjac(n,n);
    for (k=0;k<ntrial;k++) {
        usrfun(x,fvec,fjac);                    用户定义函数，fvec表示函数在x处的值，
        errf=0.0;                               fjac代表雅可比矩阵。
        for (i=0;i<n;i++) errf += fabs(fvec[i]); 检查函数收敛。
        if (errf <= tolf) return;
        for (i=0;i<n;i++) p[i] = -fvec[i];      线性议程的右端项。
        ludcmp(fjac,indx,d);                    用LU分解法解线性方程。
        lubksb(fjac,indx,p);
        errx=0.0;                               检查根收敛。
        for (i=0;i<n;i++) {                     更新解。
            errx += fabs(p[i]);
            x[i] += p[i];
        }
        if (errx <= tolx) return;
    }
    return;
}
```

9.6.1　牛顿法与极小化

在下一章中，将会看到对于多元函数求极小值的问题，目前已有一些很有效的一般性技巧。读者也许会问：为什么这类问题往往(相对)比较容易，而多维情况下的求根常常很难？极小化问题难道不是等价于求使 N 维梯度向量为零的点，也就是说与 N 元函数求根没有多大区别吗？不是的，因为一个梯度向量的各个分量不是独立的、任意的函数，相反，它们服从一些极其严格的所谓可积条件。大概地说，沿一个单独平面下降，总能找到一个极小值，因此，"下降"检验是一维的，而对多维求根则没有一种类似的概念上的方法可寻。在这种情况下，"下降"意味着从 N 个独立的函数空间同时下降，因此，决定某一个维上要进展多少值需与另一维的进展情况进行比较，这样在这一问题上必然允许有大量的折中方案。

对多维求根问题，读者可能会想到把所有维数拆成一维，即把每个函数 F_i 的平方和相加，得一个主函数 F。这个主函数满足：(i)具有正定性；(ii)在原非线性方程组的所有解处恰好有一个全局极小值零点。但是，正如在下一章中将要看到的，那些求极小值的有效算法对全局和局部极小值根本不加区分，而且非常遗憾的是，函数 F 常常有很多个局部极小值。例如在图 9.6.1 中，只要 f 和 g 的零等高线靠得很近，就有可能存在一个局部极小值。图中标号为 M 的点即是如此，但可以看出在 M 点附近并没有根。

不管怎样，多维求根实际上还是可以利用极小化的思想的，方式是将主函数 F 与牛顿法相结合，并将牛顿法应用于函数 F_i 的全体集合。尽管这种方法有时只能收敛到 F 的某个局部极小点，但对单独用牛顿法不能解决的问题一般它都能求解。下节中将详细讨论这些方法。

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 14. [1]

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press).

Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press).

9.7　非线性方程组的全局收敛法

已经看到在求解非线方程时，如果初始预测值没有足够地接近根，则牛顿法有一个不好的趋势，即会偏离到无规则的远处，而**全局法**对于几乎任意初始值都能收敛到解。本节将发展一种算法，它将快速局部收敛的牛顿法和一个全局收敛策略结合起来，这个全局收敛策略在每一步迭代中，将保证某些进程趋于解。这个算法和 10.7 节中将要详细讨论的拟牛顿法的极小化是密切相关的。

回顾9.6节中的讨论，下列方程组的牛顿法步骤是

$$\mathbf{F}(\mathbf{x}) = 0 \tag{9.7.1}$$

令

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \delta\mathbf{x} \tag{9.7.2}$$

其中

$$\delta\mathbf{x} = -\mathbf{J}^{-1} \cdot \mathbf{F} \tag{9.7.3}$$

$\mathbf{J}$是雅可比矩阵。怎样决定是否接收牛顿步长$\delta\mathbf{x}$呢？一个合理的策略是，要求这个步长使$|\mathbf{F}|^2 = \mathbf{F}\cdot\mathbf{F}$下降。这就相同于我们强加的要求，使下式$f$极小化：

$$f = \frac{1}{2}\mathbf{F} \cdot \mathbf{F} \tag{9.7.4}$$

(为了后面的方便使用才用$\frac{1}{2}$)。每次求解(9.7.1)式，同时极小化(9.7.4)式，但是可能(9.7.4)式的局部最小值使(9.7.1)式无解。因此，如前所述，简单地从第10章找一个求极小值算法运用于(9.7.4)式，这不是一种很好的思想。

为了推出一种较好的策略，注意到牛顿法步长(9.7.3)式对f是下降方向：

$$\nabla f \cdot \delta\mathbf{x} = (\mathbf{F} \cdot \mathbf{J}) \cdot (-\mathbf{J}^{-1} \cdot \mathbf{F}) = -\mathbf{F} \cdot \mathbf{F} < 0 \tag{9.7.5}$$

所以我们的策略非常简单：首先，常常试着采用牛顿全步长，因为一旦我们取的初始点离解足够近，那么将得到二次收敛。然而，在每一次迭代过程中，要检查这个所设定的步长是否减少f值。若不，就应回过头来沿牛顿方向追踪，直到有一个可接受的步长为止。牛顿步长对f而言是下降方向，以保证用回溯方法可以找到一个可接受的步长。下面将详细讨论这种回溯算法。

注意这种方法本质上是，采用使$\mathbf{F}$等于零的牛顿步长来求f极小化。而不是等价于，采用使∇f等于零的牛顿步长来直接求f极小化。当着落在f的局部极少值时，此方法仍然可能发生失败，但这种情况在实践中是非常罕见的。在下面的程序newt中，若这种情况发生它就会告诫用户。而补救的方法是用一个新的初始点重新试验。

9.7.1 深入讨论：线性搜索和回溯

当没有足够接近f的极小值时，就无需取全牛顿步长$\mathbf{p} = \delta\mathbf{x}$来下降函数，为了使二次收敛有效，我们可以移动很远。当沿着牛顿方向移动，已保证了f一开始就是下降的。所以，移动的目标是沿牛顿步长$\mathbf{p}$的方向移动到一个新的点$\mathbf{x}_{\text{new}}$，但不需要移动步长的全部路程：

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \lambda\mathbf{p}, \qquad 0 < \lambda \leqslant 1 \tag{9.7.6}$$

目的是要寻找λ值以使$f(\mathbf{x}_{\text{old}} + \lambda\mathbf{p})$充分下降。早在20世纪70年代早期，标准的方法是，选择λ以使$\mathbf{x}_{\text{new}}$在$\mathbf{p}$方向上确实使f达到极小化。但是，现在我们知道这样做就要计算函数值，这是极大的浪费。一个比较好的策略是：因为在我们的算法中，$\mathbf{p}$方向常常是牛顿方向，所以首先选$\lambda = 1$，全牛顿步长。当$\mathbf{x}$是充分接近解时，这就将导致二次收敛。但是，若$f(\mathbf{x}_{\text{new}})$不能满足我们可接受的准则，则沿着牛顿方向回溯，试着使用较小的λ值，直到找到一个合适点。因为牛顿方向是下降方向，所以保证了对于足够小的λ可使f递减。

对于可接受的步长采用什么准则呢？若只要求$f(\mathbf{x}_{\text{new}}) < f(\mathbf{x}_{\text{old}})$是不充分的。这个准则在下面两个情况之一中，会使收敛到f的最小值发生失败。第一个情况，构造的满足准则的步长序列使f下降速度相对于步长的长度太慢。第二个情况，有一组步长序列，其步长长度相对f下降的初始速率太小(这些序列的例子见本节参考文献[1])。

处理第一种问题的简单方法是，要求f下降的平均速率至少是$\nabla f\cdot\mathbf{p}$初始下降率的α部分：

$$f(\mathbf{x}_{\text{new}}) \leqslant f(\mathbf{x}_{\text{old}}) + \alpha\nabla f \cdot (\mathbf{x}_{\text{new}} - \mathbf{x}_{\text{old}}) \tag{9.7.7}$$

其中参数α满足$0 < \alpha < 1$。能够避免得到非常小的α值，较好的选择是$\alpha = 10^{-4}$。

第二种问题可以用这种方法来解决，即要求在$\mathbf{x}_{\text{new}}$处f下降率大于在$\mathbf{x}_{\text{old}}$处f下降率的β部分。实践中，常常不需要改进此第二种约束，因为回溯算法将有一个内在中止，以避免步长取得太小。

对于实际的回溯算法程序有一个策略定义：

$$g(\lambda) \equiv f(\mathbf{x}_{\text{old}} + \lambda \mathbf{p}) \tag{9.7.8}$$

以使

$$g'(\lambda) = \nabla f \cdot \mathbf{p} \tag{9.7.9}$$

若需要回溯的话，则用已有的最近信息构造模型g，并且选择 λ 使模型极小化。可以用有效的$g(0)$和$g'(0)$开始。通常，第一步是牛顿步长，$\lambda=1$。若这一步没有接收，我们有了有效的$g(1)$。所以构造模型$g(\lambda)$为二次型：

$$g(\lambda) \approx [g(1) - g(0) - g'(0)]\lambda^2 + g'(0)\lambda + g(0) \tag{9.7.10}$$

求这二次型的微分，当

$$\lambda = -\frac{g'(0)}{2[g(1) - g(0) - g'(0)]} \tag{9.7.11}$$

二次型取极小。因为牛顿步长失败，所以能够证明对于小的α，$\lambda \lesssim \frac{1}{2}$。但是，需要警惕防止 λ 太小的值，一般设 $\lambda_{\min} = 0.1$。

在第二次和以后的回溯过程中，取模型g为 λ 的三次型，利用原先的$g(\lambda_1)$值和当前第二个$g(\lambda_2)$的值来构造：

$$g(\lambda) = a\lambda^3 + b\lambda^2 + g'(0)\lambda + g(0) \tag{9.7.12}$$

在 λ_1 和 λ_2 处给出上述表达式g的当前值，构成二个方程，由此求出系数 a 和 b：

$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{\lambda_1 - \lambda_2} \begin{bmatrix} 1/\lambda_1^2 & -1/\lambda_2^2 \\ -\lambda_2/\lambda_1^2 & \lambda_1/\lambda_2^2 \end{bmatrix} \cdot \begin{bmatrix} g(\lambda_1) - g'(0)\lambda_1 - g(0) \\ g(\lambda_2) - g'(0)\lambda_2 - g(0) \end{bmatrix} \tag{9.7.13}$$

(9.7.12)式三次型的极小值是在

$$\lambda = \frac{-b + \sqrt{b^2 - 3ag'(0)}}{3a} \tag{9.7.14}$$

强制使 λ 处于 $\lambda_{\max} = 0.5\lambda_1$ 和 $\lambda_{\min} = 0.1\lambda_1$ 之间。

程序有二个附加特性值，最小步长长度 alamin 和最大步长长度 stpmax。lnsrch 也可用于下节拟牛顿极小化程序 dfpmin 中。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

void NR::lnsrch(Vec_I_DP &xold, const DP fold, Vec_I_DP &g, Vec_IO_DP &p,
    Vec_O_DP &x, DP &f, const DP stpmax, bool &check, DP func(Vec_I_DP &))
```

给定一个 n 维的点 xold[0..n-1]、函数值 fold 和斜率 g[0..n-1]，以及方向 p[0..n-1]，从 xold 沿着方向 p 寻找一个新的点 x[0..n-1]，那里使函数 func"充分地"下降。新的函数返回到 f。stpmax 是限制步长长度的输入量，以便在函数没有定义的范围或者溢出的情况下都不计算函数。p 通常是牛顿方向。输出量 check 是 false，正常退出；是 true，表示 x 非常接近 xold。在极小化算法中，这通常是单调收敛和可被忽略的。但是，在零点寻找算法中，由在调用的程序应该检查收敛是否是伪的。

```
{
    const DP ALF=1.0e-4, TOLX=numeric_limits<DP>::epsilon();
    ALF 保证函数值充分下降；TOLX 是对Δx的收敛性准则。
    DP a,alam,alam2=0.0,alamin,b,disc,f2=0.0;
    DP rhs1,rhs2,slope,sum,temp,test,tmplam;

    int n=xold.size();
    check=false;
    sum=0.0;
    for (i=0;i<n;i++) sum += p[i]*p[i];
    sum=sqrt(sum);
    if (sum > stpmax)
        for (i=0;i<n;i++) p[i] *= stpmax/sum;     测量所尝试的步长是否太长。
    slope=0.0;
    for (i=0;i<n;i++)
        slope += g[i]*p[i];
    if (slope >= 0.0) nrerror("Roundoff problem in lnsrch.");
    test=0.0;                                     计算 λmin。
    for (i=0;i<n;i++) {
```

```
            temp=fabs(p[i])/MAX(fabs(xold[i]),1.0);
            if (temp > test) test=temp;
        }
        alamin=TOLX/test;
        alam=1.0;                                   通常首先尝试全牛顿步长。
        for (;;) {                                  迭代循环开始。
            for (i=0;i<n;i++) x[i]=xold[i]+alam*p[i];
            f=func(x);
            if (alam < alamin) {                    在Δx上收敛。对于零寻找法，所谓用
                for (i=0;i<n;i++) x[i]=xold[i];     的程序将证明收敛性。
                check=true;
                return;
            } else if (f <= fold+ALF*alam*slope) return;        函数充分下降。
            else {                                              回溯。
                if (alam == 1.0)
                    tmplam = -slope/(2.0*(f-fold-slope));       第一次。
                else {                                          继续回溯。
                    rhs1=f-fold-alam*slope;
                    rhs2=f2-fold-alam2*slope;
                    a=(rhs1/(alam*alam)-rhs2/(alam2*alam2))/(alam-alam2);
                    b=(-alam2*rhs1/(alam*alam)+alam*rhs2/(alam2*alam2))/(alam-alam2);
                    if (a == 0.0) tmplam = -slope/(2.0*b);
                    else {
                        disc=b*b-3.0*a*slope;
                        if (disc < 0.0) tmplam=0.5*alam;
                        else if (b <= 0.0) tmplam=(-b+sqrt(disc))/(3.0*a);
                        else tmplam=-slope/(b+sqrt(disc));
                    }
                    if (tmplam>0.5*alam)
                        tmplam=0.5*alam;                λ≤0.5λ1。
                }
            }
            alam2=alam;
            f2 = f;
            alam=MAX(tmplam,0.1*alam);                  λ≥0.1λ1。
        }                                               重新再试。
    }
```

现在，下面是全局收敛牛顿法程序 newt，它使用 lnsrch。程序 newt 的特点是，用户不需提供解析的雅可比矩阵；此程序将使用程序 fdjac 中的有限差分，试着计算必需的 F 的偏导。此程序为了计算数值导数，使用了 5.7 节中讨论的某些技巧。当然，如果对用户来说是容易的话，通常就能用一个数值计算雅可比行列式的程序来代替 fdjac。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

Vec_DP *fvec_p;                                     Global variables to communicate with fmin.
void (*nrfuncv)(Vec_I_DP &v, Vec_O_DP &f);

void NR::newt(Vec_IO_DP &x, bool &check, void vecfunc(Vec_I_DP &, Vec_O_DP &))
给定一个n维根的初始猜测值x[0..n-1]，用全局收敛牛顿法找根。使用用户提供的程序vecfunc(x,fvec)返回函数等于
零的向量，在程序中称为fvec[0..n-1]。输出量check为false是正常返回；若收敛到一个函数的局部极小值，定义为
fmin，则check为true。在这种情况下，试着从一个不同的初始猜测值重新开始。
{
    const int MAXITS=200;
const DP TOLF=1.0e-8,TOLMIN=1.0e-12,STPMX=100.0;
const DP TOLX=numeric_limits<DP>::epsilon();
其中MAXITS是迭代的最大次数；TOLF设置了函数值收敛的判别准则；TOLMIN设置了判定是否有假收敛于极小值fmin发
生的判别准则；TOLX是对δx收敛的判别准则；STPMX是所允许的在线性搜索中所测量的最大步长长度。
int i,j,its;
DP d,den,f,fold,stpmax,sum,temp,test;

int n=x.size();
Vec_INT indx(n);
Vec_DP g(n),p(n),xold(n);
Mat_DP fjac(n,n);
fvec_p=new Vec_DP(n);                               定义全局变量。
nrfuncv=vecfunc;
Vec_DP &fvec=*fvec_p;                               使用别名来简化编码。
```

```
    f=fmin(x);                                  由此调用计算 fvec。
    test=0.0;                                   测试初始猜测根，用比 TOLF 更直接的
    for (i=0;i<n;i++)                           测试。
        if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
    if (test < 0.01*TOLF) {
        check=false;
        delete fvec_p;
        return;
    }
    sum=0.0;
    for (i=0;i<n;i++) sum += SQR(x[i]);         为线性搜索计算 stpmax。
    stpmax=STPMX*MAX(sqrt(sum),DP(n));
    for (its=0;its<MAXITS;its++) {              迭代循环开始。
        fdjac(x,fvec,fjac,vecfunc);
        若解析的雅可比行列式是有效的，用户可以用自己程序替换程序 fdjac。
        for (i=0;i<n;i++) {                     对线性搜索计算∇f。
            sum=0.0;
            for (j=0;j<n;j++) sum += fjac[j][i]*fvec[j];
            g[i]=sum;
        }
        for (i=0;i<n;i++) xold[i]=x[i];         存储 x 和 f。
        fold=f;
        for (i=0;i<n;i++) p[i] = -fvec[i];      线性方程组的右端。
        ludcmp(fjac,indx,d);                    用 LU 分解法求解线性方程。
        lubksb(fjac,indx,p);
        lnsrch(xold,fold,g,p,x,f,stpmax,check,fmin);
        返回新的 x 和 f，当调用 fmin 时在新的 x 处也计算 fvec。
        test=0.0;                               为函数值收敛而测试。
        for (i=0;i<n;i++)
            if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
        if (test < TOLF) {
            check=false;
            delete fvec_p;
            return;
        }
        if (check) {                            检测 f 的梯度为零，即伪收敛。
            test=0.0;
            den=MAX(f,0.5*n);
            for (i=0;i<n;i++) {
                temp=fabs(g[i])*MAX(fabs(x[i]),1.0)/den;
                if (temp > test) test=temp;
            }
            check=(test < TOLMIN);
            delete fvec_p;
            return;
        }
        test=0.0;                               为 δx 收敛而测试。
          for (i=0;i<n;i++) {
              temp=(fabs(x[i]-xold[i]))/MAX(fabs(x[i]),1.0);
              if (temp > test) test=temp;
          }
          if (test < TOLX) {
              delete fvec_p;
              return;
          }
    }
    nrerror("MAXITS exceeded in newt");
}


#include <cmath>
#include "nr.h"
using namespace std;

void NR::fdjac(Vec_IO_DP &x, Vec_I_DP &fvec, Mat_O_DP &df,
    void vecfunc(Vec_I_DP &, Vec_O_DP &))
```

计算正向差分以逼近雅可比行列式。输入 x[0..n-1]是雅可比行列式被计算的点，fvec[0..n-1]是在那点的函数值向量，以及 vecfunc(x,f)是一个用户所提供的程序，它返回 x 点处的函数向量。输出 df[0..n-1][0..n-1]是雅可比矩阵。

```
{
    const DP EPS=1.0e-8;            逼近机器精度的平方根。
    int i,j;
    DP h,temp;

    int n=x.size();
    Vec_DP f(n);
    for (j=0;j<n;j++) {
        temp=x[j];
        h=EPS*fabs(temp);
        if (h == 0.0) h=EPS;
        x[j]=temp+h;                减少有限精度误差的技巧。
        h=x[j]-temp;
        vecfunc(x,f);
        x[j]=temp;
        for (i=0;i<n;i++)           正向差分公式。
            df[i][j]=(f[i]-fvec[i])/h;
    }
}


#include "nr.h"

extern Vec_DP *fvec_p;
extern void (*nrfuncv)(Vec_I_DP &v, Vec_O_DP &f);

DP NR::fmin(Vec_I_DP &x)
返回 x 处的 f=1/2 F·F。全局指针 *nrfuncv 指向那个返回 x 处函数向量的程序。在调用程序中它设置为指向用户所提供的
程序。全局变量也传输函数值返回到所调用的程序中去。
{
    int i;
    DP sum;

    Vec_DP &fvec=*fvec_p;
    nrfuncv(x,fvec);
    int n=x.size();
    for (sum=0.0,i=0;i<n;i++) sum += SQR(fvec[i]);
    return 0.5*sum;
}
```

程序 newt 假设 **x** 和 **F** 的所有分量的典型值是顺序单位,若这假设被极坏地违反了,程序就会失败。若这种情况发生,在调用 newt 以前,应该用它们的典型值重新换算这些变量。

9.7.2 深入讨论:多维弦截法——Broyden 方法

上述实现的牛顿法的功能十分强大,但它仍然有几个缺点。它需要有雅可比行列式矩阵。在许多问题中,解析的导数也是得不到的。若函数的估算非常费时,则由于雅可比行列式的有限差分测定的代价使此方法的使用受到了禁止。

正如在 10.7 节中讨论的拟牛顿法,它提供了在极小化算法中,黑塞(Hessian)矩阵的容易计算的近似值。拟牛顿法也提供了在寻找零值中,雅可比矩阵的容易计算的近似值。这些方法通常也称为弦截法,因为在一维情况下,它们退化为弦截法(参见 9.2 节)(参见本书参考文献[1])。下面首先介绍的方法是这些方法中最好的一种,即 Broyden 方法(参见本书参考文献[2])。

用 **B** 表示近似的雅可比矩阵。那么,第 i 个拟牛顿步长 $\delta\mathbf{x}_i$ 是下列方程的解:

$$\mathbf{B}_i \cdot \delta\mathbf{x}_i = -\mathbf{F}_i \tag{9.7.15}$$

其中 $\delta\mathbf{x}_i = \mathbf{x}_{i+1} - \mathbf{x}_i$(参见(9.7.3)式)。拟牛顿条件或弦截条件是 $\mathbf{B}_{i+1}$ 满足

$$\mathbf{B}_{i+1} \cdot \delta\mathbf{x}_i = \delta\mathbf{F}_i \tag{9.7.16}$$

其中 $\delta\mathbf{F}_i = \mathbf{F}_{i+1} - \mathbf{F}_i$。这是一维的弦截法逼近导数 $\delta F/\delta x$ 的推广。但是在大于一维情况下,(9.7.16)式不能惟一确定 $\mathbf{B}_{i+1}$。

许多约束 $\mathbf{B}_{i+1}$ 的辅助条件已经被研究,而在实践中最好的执行算法是由 Broyden 公式得到。这个公式是基于这样的思路,即它使 $\mathbf{B}_i$ 的最小变化与弦截方程式(9.7.16)一致从而得到 $\mathbf{B}_{i+1}$。Broyden 指出这个最终公式是

$$\mathbf{B}_{i+1} = \mathbf{B}_i + \frac{(\delta\mathbf{F}_i - \mathbf{B}_i \cdot \delta\mathbf{x}_i) \otimes \delta\mathbf{x}_i}{\delta\mathbf{x}_i \cdot \delta\mathbf{x}_i} \tag{9.7.17}$$

很容易检验 $\mathbf{B}_{i+1}$ 满足(9.7.16)式。

Broyden 方法的早期实现采用 Sherman-Morrison 公式，即(2.7.2)式，解析地求(9.7.17)式的逆，

$$\mathbf{B}_{i+1}^{-1} = \mathbf{B}_i^{-1} + \frac{(\delta\mathbf{x}_i - \mathbf{B}_i^{-1} \cdot \delta\mathbf{F}_i) \otimes \delta\mathbf{x}_i \cdot \mathbf{B}_i^{-1}}{\delta\mathbf{x}_i \cdot \mathbf{B}_i^{-1} \cdot \delta\mathbf{F}_i} \tag{9.7.18}$$

那么，就可通过 $O(N^2)$ 次矩阵乘法操作来确定

$$\delta\mathbf{x}_i = -\mathbf{B}_i^{-1} \cdot \mathbf{F}_i \tag{9.7.19}$$

由此替换用 LU 分解(9.7.3)式。此方法的缺点是它不能很容易地嵌入到全局收敛策略中去。因为在全局收敛策略中，求(9.7.4)式的梯度需要用 $\mathbf{B}$，而不是 $\mathbf{B}^{-1}$，

$$\nabla(\tfrac{1}{2}\mathbf{F} \cdot \mathbf{F}) \simeq \mathbf{B}^T \cdot \mathbf{F} \tag{9.7.20}$$

相应地，在(9.7.17)式中实行更改的公式。

然而，仍然保留了 $O(N^2)$ 次操作求解(9.7.3)式，但用 QR 分解(参见 2.10 节)替换了 LU 分解，其理由是因为(9.7.17)式的特殊形式，通过 $O(N^2)$ 次操作 $\mathbf{B}_i$ 的 QR 分解，可以被更新成 $\mathbf{B}_{i+1}$ 的 QR 分解。只需用一个初始近似值 $\mathbf{B}_0$ 起始。通常起始时，用一个简单的单位矩阵是可行的，然后允许 $O(N)$ 次更新，以便产生一个合理的雅可比行列式的近似值。但是，更愿意通过调用 fdjac，使用有限差分近似方法，计算前第 N 个函数值来初始化 $\mathbf{B}$。

因为 $\mathbf{B}$ 不是精确的雅可比行列式，所以不能保证 $\delta\mathbf{x}$ 对于 $f = \frac{1}{2}\mathbf{F}\cdot\mathbf{F}$(即(9.7.5)式)是在下降的方向。因此，如果 $\mathbf{B}$ 偏离真实的雅可比行列式很远，那么线性搜索算法不能返回一个合适步长。在这种情况下，需要另外调用 fdjac 重新初始化 $\mathbf{B}$。

和一维情况中的弦截法相同，一旦足以接近根值，Broyden 方法将超线性地收敛。将此方法嵌入到全局策略中，它几乎也和牛顿法一样稳健，并且通常需要极少几个函数值的计算来确定零点。注意，即便方法收敛时，$\mathbf{B}$ 的最后值常常也不是接近根处真正的雅可比行列式。

下面给出的程序 broydn 在结构上非常类似于程序 newt。基本差别是用 QR 分解替换 LU 分解，并且用更新的公式替换直接地确定雅可比行列式。在程序 newt 末尾关于变量测度的标记同样可用于 broydn 中。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

Vec_DP *fvec_p;                          与 fmin进行全局变量通信。
void (*nrfuncv)(Vec_I_DP &v, Vec_O_DP &f);

void NR::broydn(Vec_IO_DP &x, bool &check, void vecfunc(Vec_I_DP &, Vec_O_DP &))
给定一个 n 维 x[0..n-1]的根的初始猜测值，用 Broyden 方法嵌入全局收敛策略中的方法来寻找根。使用用户提供的程
序 vecfunc(x,fvec)返回函数等于零的向量，在下面的程序中称为向量 fvec[0..n-1]。程序 fdjac 和程序 newt 中的函数
fmin 都被使用。输出量 check 为 false 是正常返回；若程序收敛到一个函数的局部极小值 fmin，或者 Broyden 方法没有进一
步进展，则 check 为 true。在这种情况下，试着从一个不同的初始猜测值重新开始。
{
    const int MAXITS=200;
    const DP EPS=numeric_limits<DP>::epsilon();
    const DP TOLF=1.0e-8, TOLX=EPS, STPMX=100.0, TOLMIN=1.0e-12;
    其中 MAXITS 是迭代的最大次数；EPS 是接近机器的精度数；TOLF 是函数值收敛的判别准则；TOLX 是 δx 的收敛判
    别准则；STPMX 是在线性搜索中所允许的、测量的最大步长长度；TOLMIN 是用来判决是否有假收敛于极小值 fmin
    的事件发生。
    bool restrt,sing,skip;
    int i,its,j,k;
    DP den,f,fold,stpmax,sum,temp,test;

    int n=x.size();
    Mat_DP qt(n,n),r(n,n);
    Vec_DP c(n),d(n),fvcold(n),g(n),p(n),s(n),t(n),w(n),xold(n);
    fvec_p=new Vec_DP(n);                定义全局变量。
    nrfuncv=vecfunc;
    Vec_DP &fvec=*fvec_p;                使用引用简化编码。
```

```
f=fmin(x);                                      在此调用时也计算向量 fvec。
test=0.0;
for (i=0;i<n;i++)                               测试初始猜测是否是根,
    if (fabs(fvec[i])>test) test=fabs(fvec[i]); 采用此 TOLF 更直接的测试。
if (test < 0.01*TOLF) {
    check=false;
    delete fvec_p;
    return;
}
for (sum=0.0,i=0;i<n;i++) sum += SQR(x[i]);     为线性搜索计算 stpmax。
stpmax=STPMX*MAX(sqrt(sum),DP(n));
restrt=true;                                    保证初始雅可比行列式得到计算。
for (its=1;its<=MAXITS;its++) {                 迭代循环开始。
    if (restrt) {
        fdjac(x,fvec,r,vecfunc);                雅可比行列式 QR 分解。
          qrdcmp(r,c,d,sing);
          if (sing) nrerror("singular Jacobian in broydn");
          for (i=0;i<n;i++) {                       从 Q^T 显见。
              for (j=0;j<n;j++) qt[i][j]=0.0;
              qt[i][i]=1.0;
          }
          for (k=0;k<n-1;k++) {
              if (c[k] != 0.0) {
                  for (j=0;j<n;j++) {
                      sum=0.0;
                      for (i=k;i<n;i++)
                          sum += r[i][k]*qt[i][j];
                      sum /= c[k];
                      for (i=k;i<n;i++)
                          qt[i][j] -= sum*r[i][k];
                  }
              }
          }
          for (i=0;i<n;i++) {                       从 R 显见。
              r[i][i]=d[i];
              for (j=0;j<i;j++) r[i][j]=0.0;
          }
      } else {                                      执行 Broyden 更新。
          for (i=0;i<n;i++) s[i]=x[i]-xold[i];      s = δx。
          for (i=0;i<n;i++) {                       t = R·s。
              for (sum=0.0,j=i;j<n;j++) sum += r[i][j]*s[j];
              t[i]=sum;
          }
          skip=true;
          for (i=0;i<n;i++) {                       w = δF − B·s。
              for (sum=0.0,j=0;j<n;j++) sum += qt[j][i]*t[j];
              w[i]=fvec[i]-fvcold[i]-sum;
              if (fabs(w[i]) >= EPS*(fabs(fvec[i])+fabs(fvcold[i]))) skip=false;
              不要更新 w 中无噪的分量。
              else w[i]=0.0;
          }
          if (!skip) {
              for (i=0;i<n;i++) {                   t = Q^T·w。
                  for (sum=0.0,j=0;j<n;j++) sum += qt[i][j]*w[j];
                  t[i]=sum;
              }
              for (den=0.0,i=0;i<n;i++) den += SQR(s[i]);
              for (i=0;i<n;i++) s[i] /= den;        在 s 中存储 s/(s·s)。
              qrupdt(r,qt,t,s);                     更新 R 和 Q^T。
              for (i=0;i<n;i++) {
                  if (r[i][i] == 0.0) nrerror("r singular in broydn");
                  d[i]=r[i][i];                     R 中对角元存入 d。
              }
          }
      }
      for (i=0;i<n;i++) {                           线性方程的右端是 −Q^T·F。
          for (sum=0.0,j=0;j<n;j++) sum += qt[i][j]*fvec[j];
          p[i] = -sum;
      }
      for (i=n-1;i>=0;i--) {                        对于线性搜索计算 ∇f≈(Q·R)^T·F。
          for (sum=0.0,j=0;j<=i;j++) sum -= r[j][i]*p[j];
```

```
            g[i]=sum;
        }
        for (i=0;i<n;i++) {                存储 x 和 F。
            xold[i]=x[i];
            fvcold[i]=fvec[i];
        }
        fold=f;                            存储 f。
        rsolv(r,d,p);                      解线性方程。
          lnsrch(xold,fold,g,p,x,f,stpmax,check,fmin);
          返回新的 x 和 f。当调用 fmin 时，也在新的 x 处计算 fvec。
          test=0.0;                          测试函数值的收敛性。
          for (i=0;i<n;i++)
              if (fabs(fvec[i]) > test) test=fabs(fvec[i]);
          if (test < TOLF) {
              check=false;
              delete fvec_p;
              return;
          }
          if (check) {                       若线性搜索寻找新的 x 失败，则为真。
              if (restrt) {                  失败，已经试着重新初始化雅可比行列式。
                  delete fvec_p;
                  return;
              } else {
                  test=0.0;                  检查 f 的梯度为零，即伪收敛。
                  den=MAX(f,0.5*n);
                  for (i=0;i<n;i++) {
                      temp=fabs(g[i])*MAX(fabs(x[i]),1.0)/den;
                      if (temp > test) test=temp;
                  }
                  if (test < TOLMIN) {
                      delete fvec_p;
                      return;
                  }
                  else restrt=true;          试着重新初始化雅可比行列式。
              }
          } else {                           成功的步骤，对下一步将采用 Broyden 更新。
              restrt=false;
              test=0.0;                      为 δx 收敛而测试。
              for (i=0;i<n;i++) {
                  temp=(fabs(x[i]-xold[i]))/MAX(fabs(x[i]),1.0);
                  if (temp > test) test=temp;
              }
              if (test < TOLX) {
                  delete fvec_p;
                  return;
              }
          }
      }
      nrerror("MAXITS exceeded in broydn");
      return;
}
```

9.7.3　深入讨论：更先进的实现

假如 **J**(在牛顿法中)或 **B**(在 Broyden 方法中)成为奇异的或接近奇异的，则至今所叙述的方法都可能失败，以使 $\delta\mathbf{x}$ 不能被确定。倘若幸运的话，这种情况在实际情况中常常不会发生。解决这些问题的方法已有进展，所处理的问题包含若检测出奇异性或接近奇异性，则调节 **J** 的制约数及扰动 **J**。最容易实现的是，在牛顿法中用 QR 分解替代 LU 分解(参见本节参考文献[1])。个人的经验是，当某一种方法，它能够解决 **J** 确实为奇异的问题以及标准牛顿法失效的问题，但是此方法应用于其他 LU 分解法成功的问题中，却可能减少稳定性。显然，这里重要的是要包含舍入误差、下溢等问题。

求极小化和寻找零值的全局策略都是基于线性搜索。其他一些全局算法，如**钩形步长**(hook step)法和**曲折形步长**(dogleg step)法，都是基于**模型可信赖域方法**，它和非线性最小二乘的 Levenberg-Marquardt 算法有关(参见 15.5 节)。尽管这些方法比线性搜索法要复杂得多，但是这些方法以性能稳健而盛名，甚至当起始时远离所期望的零值或最小值时，它仍是稳健的[1]。

参考文献和深入阅读

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*; reprinted 1996 (Philadelphia: S.I.A.M.). [1]

Broyden, C.G. 1965, *Mathematics of Computation*, vol. 19, pp. 577–593. [2]

第 10 章　函数的极值

10.0　引言

本章要研究的问题概括来说，即给定一个函数 f，它依赖于一个或多个独立变量，求 f 达到某一极大值或极小值时自变量的取值，并计算出在这个极大值点或极小值点处函数 f 的取值。显然求极大值问题与求极小值问题有关，因为 f 的极大值恰好是 $-f$ 的极小值。对计算方法的要求，求极值与其他问题一样，都希望速度快、计算量小且占用内存少。一般来说，计算量的大小主要取决于函数值的计算量（根据算法的不同，可能还包括 f 对其所有变量的偏导的数值计算量）。因此，对计算方法的诸多要求常被简化为一条简单要求，即使计算 f 值的次数尽可能地少。

函数的极值（极大值或极小值）可以是**整体的**（即函数真正的最大值或最小值），也可能是**局部的**（在有限邻域内的最大值或最小值，不包括这个领域的边界），参见图 10.0.1。一般来说，求整体极值是一个很难解决的问题，目前广泛采用的是两种标准的启发式方法：其一，从自变量的各种不同初值（如 7.7 节中那样拟随机选取）开始，求出所有局部极值，然后从中选出最大的或最小的（如果这些局部极值不是完全一样的话）；其二，对局部极值以一有限步长的扰动，然后分析由此计算出来的结果是稍有改进，还是"一直"保持不变。值得一提的是，近年来有一类称为"模拟退火"（simulated annealing）的方法，在各种全局极小化问题上取得了一些重要的和成功的进展（详见 10.9 节）。

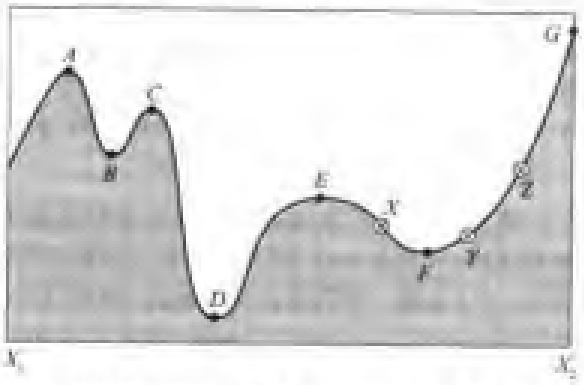


图 10.0.1　在某一区间内的函数极值。点 A，C 和 E 为局部（非整体）极大值，点 B 和 F 为局部（不是整体的）极小值。整体极大值点为 G 点，它位于区间的边界上，因此函数在该点的导数不必为零。整体极小值位于 D 点。在点 E 处导数值高于一阶的都为零，这种情况对某些算法来说，可能会带来处理上的不便。我们称点 X，Y，Z "划界"了极小值点 F，这是因为 Y 点的函数值比 X 点和 Z 点的函数值都小

本章的标题也可以称为**最优化**，它是这一广泛的数值研究领域中通常采用的名词。最优化方法之所以重要，是因为它所涉及的问题形形色色、种类繁多，而这一重要性在很大程度上也依赖于特殊问题的特殊要求。比如说，经济学家和工程师们着重关心的是**约束最优化问题**，这类问题对自变量的允许值预先有一定的限制。例如，美国的小麦产量必须是一个非负的数等。约束最优化问题中，研究得颇为透彻的一个领域称为**线性规划**，即目标函数及约束条件正好是自变量的线性组合。本章 10.8 节将对线性规划中的所谓"单纯形法"进行讨论，但这一节的内容与本章其他各节有些不太连贯。

10.9 节的内容也属于本章讨论的主题之外的内容，其原因在于：所谓的"退火法"目前还很新，尚不能确定它适合于问题求解过程中的哪些步骤。但是，退火法确实解决了一些以前被认为在实际应用当中不能解决的问题。它直接提出了在大量局部极值中求整体极值这类问题，其中这些局部极值的数目常常是不可预期的。

本章的其他各节，精选了几种无约束极小化问题的最优算法（为了明确起见，在后面的讨论中，我们认为最优化问题就是求极小值问题）。各节是连贯的，后面的内容依赖于前面的讨论。如果读者只想寻找一种"完善"的算法来解决某个特殊问题，那么可能会感到书中所讲述的内容比你想要了解的东西多。遗憾的是，目前还没有一种完善的最优化算法。因此建议读者多试验几种算法，多进行比较。对于最初算法的选择可以从如下几方面来考虑：

- 目前有两种方法可供选择,一种只需计算目标函数的值,另一种还需计算函数的导数值。在多维情况下,函数的导数即梯度,它是一个向量。使用导数的算法在某些功能上优于只使用函数值的算法,但功能上的优点通常不足以补偿所增加的导数的计算量。可以很容易地构造出这两种方法的例子。但是,如果函数的导数可以计算出来,则不妨试着使用它们。
- 对于不需计算导数的一维极小值问题(即求一个自变量的函数的极小值),可以先用10.1节介绍的方法确定极小值点所在的区间,然后再用10.2节的Brent方法求解。如果目标函数二阶(或更低阶)导数不连续,则Brent方法的抛物线内插不适合使用,这时可使用**黄金分割搜索方法**的最简单形式(详见10.1节)。
- 对于需计算导数的一维极值问题,10.3节提供了Brent方法的一种变化形式,这种方法只是有限地利用了一阶导数的信息。一般情况下,回避用导数信息去构造高阶的内插多项式,因为凭我们的经验,这种做法在收敛性方面的改进(即收敛到一个光滑的解析的极小值附近)并不能弥补多项式在迭代初期做出错误内插的趋向,特别对于那些具有极明显"指数"特征的函数。

现在再来看看多维情况,讨论一下使用和不使用一阶导数的算法需注意的问题。

- 同样必须在两类方法之间做选择,一种是存储数量级为 N^2 的算法,另一种是数量级仅为 N 的算法(这里 N 为维数)。虽然对于 N 的适中值以及合理的内存大小这一约束并不算苛刻,但偶尔也有一些问题对存储量的要求很严格。
- 在10.4节中,将给出**下降的单纯形法**,这种不太受重视的算法是由Nelder和Mead提出的(注意,这里所用的"单纯形"一词不能与线性规划中的单纯形方法相混淆)。它对目标函数几乎不需进行任何特殊的假设,是一种直接的下降算法。其下降速度虽然很慢,但在某些情况下,也算是一种功能很强的算法。特别值得一提的是,这种算法的源程序相当简明,而且完全不需其他任何调用。也许会觉得不可思议,这个求解一般 N 维极小化问题的程序还不到100行语句。如果在某个整体项目中,求极小值问题仅仅是其中的一个枝节部分,那么这种方法将是最实用的,其存储数量级为 N^2,并且不需计算导数。
- 在10.5节中将介绍以**Powell方法**为原型的**方向集法**,在函数的导数不易计算时,可选择这类方法,即使在导数可以计算时,也不妨一试。方向集法虽然不需要计算导数,但却要用到一维极小化子算法,如Brent方法等(见上文)。这种方法的存储数量级也达到 N^2。

在求多维极小化问题的算法中,需要一阶导数参加运算的方法主要有两类,它们都要利用一维的子算法,而子算法本身用不用导数信息都无妨,只要认为合适即可(依赖于函数及其梯度向量的相对计算量)。需要说明的是,对于各种应用问题来说,并不是某一类算法占主导地位,两者均可供选择:

- 第一类通称为**共轭梯度法**,典型的有**Fletcher-Reeves算法**,以及与之密切相关且比它有所改进的**Polak-Ribiere算法**。共轭梯度法的存储量只是 N 的几倍数量级,但需要计算导数及一维子算法。详细的讨论及实现请见10.6节。
- 第二类算法统称为**拟牛顿法**或**变尺度法**,典型的有Davidon-Fletcher-Powell(DFP)算法(有时也称Fletcher-Powell算法),以及与之密切相关的Broyden-Fletcher-Goldfarb-Shanno(BFGS)算法。这些算法存储量都达到 N^2,需要导数信息及一维子算法,详见10.7节。

至此,就可以进入本章正题了,了解一下求实际的最优化问题的难易程度。

参考文献和深入阅读

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*; reprinted 1996 (Philadelphia: S.I.A M.).

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press).

Gill, P.E., Murray, W., and Wright, M.H. 1981, *Practical Optimization* (New York: Academic Press).

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 17.

Jacobs, D.A.H. (ed) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III 1.

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall).

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 10.

10.1　一维黄金分割搜索

先回忆一下 9.1 节中的二分法在一维情况下如何求解方程根：首先，假设根落在某一区间(a,b)内，然后，计算函数在区间中间某点 x 的值，得到一个较小的划界区间(a,x)和(x,b)。对上述过程进行迭代，直到区间的长度足够小为止。显然将 x 选为(a,b)区间的中点是最优的，这样即使在函数"不合作"的情况下(即下一解区间总是被迫选为较长的区间段)，也能使区间的减小速度达到最快。

二分法的这些思想完全可以平移到求极小值问题上，但有一个问题是：如何对极小值点划界呢？如果函数在 a,b 两点(满足 $a<b$)的值具有相反的符号，则该函数必有一个根落在点对 a 与 b 之间。与之对照，如果当有三点组 $a<b<c$(或 $c<b<a$)，且 $f(b)$比 $f(a)$和 $f(c)$都小，则称为有一极小值被划界。对上面这种情况，函数(如果是非奇异的)极小值显然落在区间(a,c)内。

与二分法类似，需要在 a 和 b 之间或 b 和 c 之间选择一个新的点 x。假定在 b 和 c 之间选择了某点 x，并计算出 $f(x)$的值。如果 $f(b)<f(x)$，则取新的三点组为(a,b,x)；相反地，如果 $f(b)>f(x)$，则新的三点组将取为(b,x,c)。不论哪一种情况，三点组的中间点的纵坐标总是当前得到的函数值中最小的，见图 10.1.1。继续这一过程，直至三点组中的两外点距离小到可容许的程度为止。

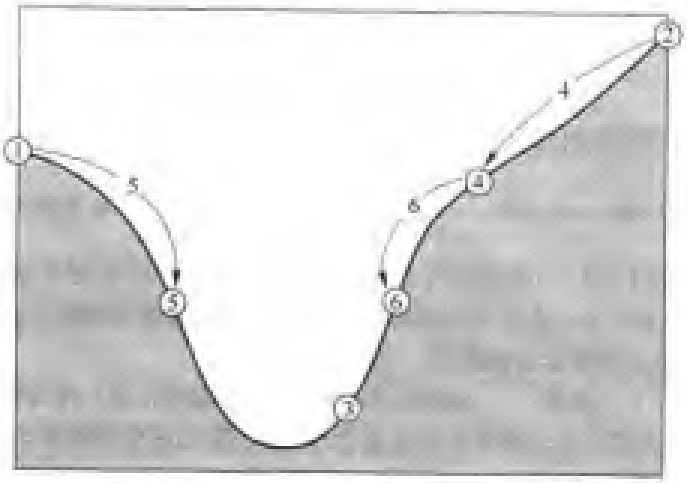


图 10.1.1　求极小值的逐步划界过程。包含极小点的划界区间开始时取为点 1,3,2。计算函数在点 4 的值，由点4代替点2，然后由点5代替点1，接着又由点6代替点4。每步迭代的规则是保持三点组的内点函数值始终比两外点的函数值小。经过图中所示的几步搜索之后，极小值点被划界在点5,3,6之间

"可容许小"到底有多小呢？若函数在 b 点取极小值，读者可能会简单地认为，极小值点的划界区间范围最终可以小到$(1-\epsilon)b<b<(1+\epsilon)b$，这里 ϵ 为所使用的计算机的浮点精度，如 3×10^{-8}(对浮点数)或 10^{-15}(对双精度数)。事实上不是这样的。一般地，在 b 点附近$f(x)$的形式可由泰勒定理给出：

$$f(x)\approx f(b)+\frac{1}{2}f''(b)(x-b)^2 \tag{10.1.1}$$

当

$$|x-b|<\sqrt{\epsilon}|b|\sqrt{\frac{2|f(b)|}{b^2f''(b)}} \tag{10.1.2}$$

时，(10.1.1)式中第二项较第一项小ϵ倍，若将它加到和式中相当于添零，因此可忽略。我们将(10.1.2)式中的右端项写成所示形式是因为，对绝大多数函数来说，最后的平方根项是一个单位阶的数。因此，根据粗略估计，要想划界一个解区间，使其长度小于其中心值的$\sqrt{\epsilon}$倍，即只有大约 10^{-4}(单精度)或 3×10^{-8}(双精度)的宽度，这几乎是不可能的。只要清楚这一事实，就可以避免做许多无用功。

本章给出的求极小值的程序通常需要由用户提供一个参数 tol，并要求程序返回横坐标的相对精度达到 ± tol(划界区间的长度为 2 × tol)。除非能对(10.1.2)式的右端项给出更好的估计值，否则必须置 tol 等于机器

浮点精度的平方根(不能小得太多),因为比之更小的值将得不到任何结果。

剩下的问题就是要在给定(a,b,c)的条件下,如何确定新的点x。设b分a和c间的距离为比例w,即

$$\frac{b-a}{c-a}=w \qquad \frac{c-b}{c-a}=1-w \tag{10.1.3}$$

设下一试验点x与b,c,a的关系如下式所示:

$$\frac{x-b}{c-a}=z \tag{10.1.4}$$

则下一划界段的长度将是当前区间长度的$w+z$倍或$1-w$倍。为了使最坏情况出现的可能性达到最小,选择一个z值,使得它们相等,即

$$z=1-2w \tag{10.1.5}$$

立即可以看出,新点x是在原区间内b点的对称点,也就是说$|b-a|$等于$|x-c|$。这说明点x位于两小段区间中较长的那一段中(仅当$w<\frac{1}{2}$时,z为正)。

但是x究竟位于较长区间段的什么具体位置上呢?w值本身又从哪里得到呢?假设w是从前面的步骤中得到的。因此,如果z的选择是最佳的,那么在此以前选择的w也是最佳的。这一比例相似性说明x分b,c(假设从b到c为较长区间段)的比例与b分a,c的比例应相同,也就是说:

$$\frac{z}{1-w}=w \tag{10.1.6}$$

由(10.1.5)式及(10.1.6)式可以得到二次方程

$$w^2-3w+1=0 \qquad \text{导出} \qquad w=\frac{3-\sqrt{5}}{2}\approx 0.38197 \tag{10.1.7}$$

换句话说,最佳划界区间(a,b,c)应满足内点b与其中一个外点(如a)的距离比例为0.381 97,而与另一外点(如b)的距离比例为0.618 03。这些比例即所谓的**黄金中项**或**黄金分割数**,这种与二分法求根类似的函数极小化最优方法也由此得名为**黄金分割搜索**。将其归纳如下。

在每一步划界中给定一个三点组,下一试探点取在较长区间段的0.381 97分割点(从三点组的中间点量起)。如果开始时划界的三点组的各段没有按照黄金比率分割,那么逐次选择黄金分割点的程序将会很快地收敛到合适的、自行重复的比率。

黄金分割搜索可以保证每计算一次函数值(获得自行重复的比率之后),划界区间长度变为原来的0.618 03倍,但这与二分法求根中的0.500 00倍相比,显然要差一些。黄金分割法的收敛是线性的(按第9章的语言来说),即随着附加函数值的计算,有效数字逐步线性增长。下一节中还将介绍一种超线性方法,这种方法每做一次函数计算,有效数字的增长速度都有所提高。

10.1.1 确定初始划界为极小的程序

在前面的讨论中,已假定初始划界区间是能够确定的,并认为这是所有一维极小值问题的一个基本步骤。虽然有某些一维算法不需要确定严格的初始划界的区间,但是即使这些非划界算法程序在运行过程中能保证函数值不断减小,我们也**始终不能**确信极小值的存在。因此在求极小值(或零解)之前,一定要确定包含它们的区间。

确定初始划界的步骤和方法没有太多的理论依据。虽然向下搜索这一方向是很明确的,但究竟搜索多远呢?从初始预测值开始,希望采取越来越大的步长,加大步长的方法既可以通过一常数因子,也可以利用前面的点进行抛物线外推的结果,这种外推的设计就是为了达到外推拐点。即使步长增加到很大也是没有多大关系的,因为毕竟在进行向下搜索,所以已经具备了三点划界点的左边点和中间点,再用足够大的一步以停止下降的趋势,就可以得到函数值略高的第三个点。

确定初始搜索区间的标准程序如下所示:

```
#include <cmath>
#include "nr.h"
using namespace std;

namespace {
    inline void shft3(DP &a, DP &b, DP &c, const DP d)
    {
        a=b;
        b=c;
        c=d;
    }
}

void NR::mnbrak(DP &ax, DP &bx, DP &cx, DP &fa, DP &fb, DP &fc,
    DP func(const DP))
给定一个函数 func 及两个相异初始点 ax 和 bx,本程序沿向下方向进行搜索(方向是由函数在初始点处的值确定),并返回
一个新的点 ax,bx,cx,由它们确定的区间能划界函数的一个极小值;本程序同时返回的还有函数在这三个点的值 fa,fb 及
fc。
{
    const DP GOLD=1.618034,GLIMIT=100.0,TINY=1.0e-20;
    其中 GOLD 是连续区间放大的默认倍率;GLIMIT 是抛物线拟合过程中允许的最大放大倍数.
    DP ulim,u,r,q,fu;

    fa=func(ax);
    fb=func(bx);
    if (fb > fa) {                          交换 a 和 b 的位置，使能够沿着由 a 到 b 方向
        SWAP(ax,bx);                        向下搜索。
        SWAP(fb,fa);
    }
    cx=bx+GOLD*(bx-ax);                     c 的第一个估值。
    fc=func(cx);
    while (fb > fc) {                       在界定搜索区间之前，使程序在此继续返回
        r=(bx-ax)*(fb-fc);                  通过 a,b,c 的抛物外推计算 u。 TINY 在此处
        q=(bx-cx)*(fb-fa);                  的作用是防止被零除的情况发生。
        u=bx-((bx-cx)*q-(bx-ax)*r)/
            (2.0*SIGN(MAX(fabs(q-r),TINY),q-r));
        ulim=bx+GLIMIT*(cx-bx);
        至此程序不能再往下进行了。现在来考虑一下各种可能的情况。
        if ((bx-u)*(u-cx) > 0.0) {          u 位于 b 和 c 之间。
            fu=func(u);
            if (fu < fc) {                  在 b 和 c 之间找到了一个极小点。
                ax=bx;
                bx=u;
                fa=fb;
                fb=fu;
                return;
            } else if (fu > fb) {           在 a 和 u 之间找到了一个极小点。
                cx=u;
                fc=fu;
                return;
            }
            u=cx+GOLD*(cx-bx);              抛物线拟合不发挥作用。利用默认的放大
            fu=func(u);                     倍数。
        } else if ((cx-u)*(u-ulim) > 0.0) {         抛物线拟合在 c 及其允许限
            fu=func(u);                             之间进行。
            if (fu < fc) {
                shft3(bx,cx,u,cx+GOLD*(cx-bx));
                shft3(fb,fc,fu,func(u));
            }
        } else if ((u-ulim)*(ulim-cx) >= 0.0) {     将抛物线 u 界定在最大允
            u=ulim;                                 许值内。
            fu=func(u);
        } else {                            放弃 u，利用默认的放大倍数。
            u=cx+GOLD*(cx-bx);
            fu=func(u);
        }
        shft3(ax,bx,cx,u);                  去掉最开始的点，继续执行搜索。
        shft3(fa,fb,fc,fu);
    }
}
```

(因为上述程序要对三四个点及其函数值进行分析和讨论,所以其结尾看起来似乎有些麻烦。本章中还

有几个程序也是如此,但它们中包含的思想却相当简单。)

10.1.2 黄金分割搜索方法的程序

```
#include <cmath>
#include "nr.h"
using namespace std;

namespace {
    inline void shft2(DP &a, DP &b, const DP c)
    {
        a=b;
        b=c;
    }

    inline void shft3(DP &a, DP &b, DP &c, const DP d)
    {
        a=b;
        b=c;
        c=d;
    }
}

DP NR::golden(const DP ax, const DP bx, const DP cx, DP f(const DP),
    const DP tol, DP &xmin)
给定一个函数 f 及一个包含极小值的三点组 ax,bx,cx(其中 bx 位于 ax 与 cx 之间,其函数值 f(bx)比 f(ax)和f(cx)都小)。本
程序将利用黄金分割搜索方法求 f 的极小值,结果的相对精度可达到 tol,其中极小值点的横坐标将以 xmin 返回,而极小点
的函数值则以 golden 返回(golden 也是本程序的返回值)。
{
    const DP R=0.61803399,C=1.0-R;          黄金分割率。
    DP f1,f2,x0,x1,x2,x3;

    x0=ax;                                  每个时刻都要跟踪四个点 x0,x1,x2,x3。
    x3=cx;
    if (fabs(cx-bx) > fabs(bx-ax)) {        以 x0,x1 作为较短区间段的端点,
        x1=bx;
        x2=bx+C*(cx-bx);                    并填入新的测探点。
    } else {
        x2=bx;
        x1=bx-C*(bx-ax);
    }
    f1=f(x1);                               函数值的计算。注意函数在原始端点的值总
    f2=f(x2);                               是不需计算。
    while (fabs(x3-x0) > tol*(fabs(x1)+fabs(x2))) {
        if (f2 < f1) {                      一个可能的结果,
            shft3(x0,x1,x2,R*x2+C*x3);      作记录,
            shft2(f1,f2,f(x2));             计算新的函数值。
        } else {                            另一种结果,
            shft3(x3,x2,x1,R*x1+C*x0);
            shft2(f2,f1,f(x1));             计算新的函数值。
        }
    }                                       若进行完毕则回头查看。
    if (f1 < f2) {                          进行完毕。输出两个当前值中最佳的
        xmin=x1;                            那一个。
        return f1;
    } else {
        xmin=x2;
        return f2;
    }
}
```

10.2 抛物线内插和一维 Brent 方法

在上一节中给出的程序 mnbrak 中,曾经提到过抛物线内插用途,现在进行详细探讨。假设很不容易才发现某个极小点,而一直未计算出来。实际上黄金分割搜索方法的设计正是为了有效地处理这种最坏的可能。但我们为什么要假设这种最坏情况呢?如果函数在极小值附近具有很好的抛物线性质(足够光滑的函数一般都具备这一性质),那么利用过任意三点拟合出来的抛物线,应能够很容易地求出极小值,或者至少能

够非常接近极小值(参见图 10.2.1)。由于目标是求极小值点的横坐标,而不是其纵坐标,因此上述方法在技术上被称为**逆抛物线内插**。

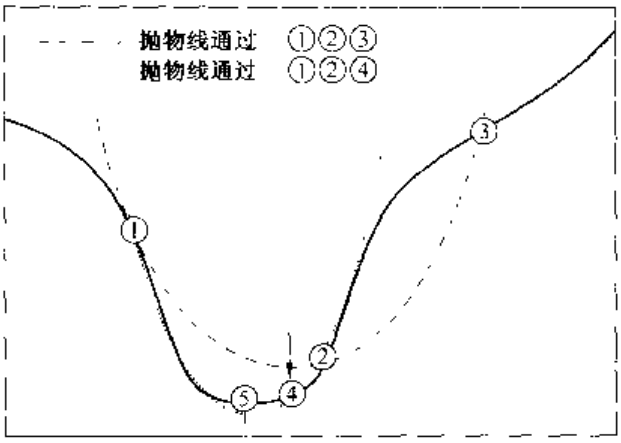


图 10.2.1　用逆抛物线内插法收敛到极小点的过程图。过给定函数(图中以实线表示)上的三点 1,2,3 做一条抛物线(图中虚线所示),计算在抛物线的极小值点处的函数值4,并以此点代替点3。再过点1,4,2做一条新的抛物线(图中点线所示),这条抛物线的极小值点5已经很接近目标函数的极小值点

对于过 $f(a)$,$f(b)$和 $f(c)$ 三点的抛物线,其极小值点的横坐标公式为

$$x = b - \frac{1}{2}\frac{(b-a)^2[f(b)-f(c)]-(b-c)^2[f(b)-f(a)]}{(b-a)[f(b)-f(c)]-(b-c)[f(b)-f(a)]} \tag{10.2.1}$$

显然 x 能很容易得到。只有当三点共线,即分母为零时(此时抛物线的极小值在无穷远处),这个公式才是无意义的。但是,还应注意到由(10.2.1)式求出的点既可以是抛物线的极小值,也可能是极大值点。在实际情况中,没有一种求极小值的方案仅仅依赖于(10.2.1)式就能得到成功的结果。

我们的任务就在于设计一种方案,当函数"不配合"时能依赖于一种收敛速度慢但很可靠的技巧(比如黄金分割搜索方法)来求解,而在目标函数允许的情况下,又能转而利用(10.2.1)式来求解。看来这个任务并不轻松,原因是:(i)在两种方法间转换时,为避免不必要的函数计算而所需的记录工作可能会很复杂;(ii)在计算结果的末尾小数时必须谨慎小心,因为此时函数值的计算已经非常接近(10.1.2)式的舍入极限;(iii)检验目标函数配合与否的方案设计必须要考虑周全。

Brent 方法[1]能在各种特殊情况下完成上述任务。它在每一特定阶段跟踪记录总共 6 个函数点(不必全不相同)a,b,u,v,w 和 x,这些点的定义如下:a 和 b 之间包含极小值点;x 是当前所能找到的最小函数值对应的点(或在函数值相等的情况下,x 是最新求出的点);w 点的函数值为第二小的函数值;v 是 w 的上一次迭代值;而最近一次计算函数值是在 u 点。Brent 方法中,还将出现点 x_m,它是 a 和 b 的中间点,但不在该点计算函数值。

阅读下文的程序可以了解算法的逻辑结构,但这里还要提几条有益的一般性原则:假设用抛物线内插拟合点 x,v 及 w。为了达到求解的目的,抛物线必须满足:(i)在限定区间(a,b)内下降;(ii)从当前最佳值 x 移动的步长能够小于前面第二步移动步长的一半。后一条准则保证了内插步骤可以有效地收敛到某点,而不会在某一非收敛的有限范围内跳跃。可能最坏的情况是,抛物线内插可以进行但不起作用,这时 Brent 方法将交替使用内插和黄金分割两种方法,并由后者保证其收敛性。在算法中要与前面第二步的结果进行比较的理由是:经验表明,如果某一步迭代结果不佳,但却能在下一步中得到补偿,这时最好不要对其进行"惩罚",这一点从本质上来看是很有启发性的。

程序中还有一条限制是,函数在与已计算过的点(或已知的端点)相距不足 tol 的点处不需计算其值,其中的原因正如在(10.1.2)式中所看到的,这样做没有任何信息包含在里面:这些点的函数值之间的差距仅仅在舍入误差的数量级上。因此在下面的程序中读者可以看到,有几处对新的可能试验点的检验和调整反映了这条限制。此外,这条限制随着"已做点"的测试会有细微的相互影响。因此这点也同样应考虑在列。

Brent 方法的运行结果,通常是 a 与 b 之间相距 $2\times x\times$ tol,其中 x(极小值点的横坐标)为 a 与 b 的中点,因此它的结果精度将达到 $\pm$ tol。

最后应提醒大家注意的是tol一般不能小于机器浮点精度的平方根。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

namespace {
    inline void shft3(DP &a, DP &b, DP &c, const DP d)
    {
        a=b;
        b=c;
        c=d;
    }
}

DP NR::brent(const DP ax, const DP bx, const DP cx, DP f(const DP),
    const DP tol, DP &xmin)
给定一个函数f并给定划界一点的横坐标ax,bx,cx(满足bx位于ax与cx之间,其函数值f(bx)比f(ax)和f(cx)都小)。本
程序将用Brent方法搜索f的极小值点,结果的相对精度可达tol,其中横坐标以xmin返回,函数值以brent返回(这也是本
程序的返回值)
{
    const int ITMAX=100;
    const DP CGOLD=0.3819660;
    const DP ZEPS=numeric_limits<DP>::epsilon()*1.0e-3;
    其中ITMAX是允许迭代的最大次数;CGOLD是黄金分割率;ZEPS是对恰巧为零的极小值情况而设置的一个相对精确
    度的小数。
    int iter;
    DP a,b,d=0.0,etemp,fu,fv,fw,fx;
    DP p,q,r,tol1,tol2,u,v,w,x,xm;
    DP e=0.0;                                   前第二步移动的距离。

    a=(ax < cx ? ax : cx);                      不管a和b的输入情况如何，其横
    b=(ax > cx ? ax : cx);                      坐标需按上升顺序。
    x=w=v=bx;                                   初始化……
    fw=fv=fx=f(x);
    for (iter=0;iter<ITMAX;iter++) {            主循环。
        xm=0.5*(a+b);
        tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
        if (fabs(x-xm) <= (tol2-0.5*(b-a))) {   这里做测试。
            xmin=x;
            return fx;
        }
        if (fabs(e) > tol1) {                   构造一个试探用的抛物线拟合。
            r=(x-w)*(fx-fv);
            q=(x-v)*(fx-fw);
            p=(x-v)*q-(x-w)*r;
            q=2.0*(q-r);
            if (q > 0.0) p = -p;
            q=fabs(q);
            etemp=e;
            e=d;
            if (fabs(p) >= fabs(0.5*q*etemp) || p <= q*(a-x) || p >= q*(b-x))
                d=CGOLD*(e=(x >= xm ? a-x : b-x));
            上述条件决定抛物线内插是否合适。在较长区间段采用黄金分割法。
            else {
                d=p/q;                          采用抛物线内插。
                u=x+d;
                if (u-a < tol2 || b-u < tol2)
                    d=SIGN(tol1,xm-x);
            }
        } else {
            d=CGOLD*(e=(x >= xm ? a-x : b-x));
        }
        u=(fabs(d) >= tol1 ? x+d : x+SIGN(tol1,d));
        fu=f(u);
        每次迭代在此计算一次函数值。
        if (fu <= fx) {                         现在判断下一步该做什么。
            if (u >= x) a=x; else b=x;
            shft3(v,w,x,u);                     整理:
            shft3(fv,fw,fx,fu);
        } else {
```

```
            if (u < x) a=u; else b=u;
            if (fu <= fw || w == x) {
                v=w;
                w=u;
                fv=fw;
                fw=fu;
            } else if (fu <= fv || v == x || v == w) {
                v=u;
                fv=fu;
            }
        }                                          整理完毕。返回进入下轮迭代。
    }
    nrerror("Too many iterations in brent");
    xmin=x;                                        永远不能到达这里。
    return fx;
}
```

参考文献和深入阅读

Brent, R.P 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 5 [1]

Forsythe, G.E., Malcolm, M.A., and Moler, C.B 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §8.2.

10.3　使用一阶导数的一维搜索方法

在本节中详细讨论与上节同样的问题，即在具有横坐标(a,b,c)划界的三点中搜索函数的极小值，所不同的是，除利用函数本身的值之外，本节所介绍的方法还要利用函数的一阶导数信息。

一般情况下，可能只是简单地想到利用像程序 rtflsp 或 zbrent(参见 9.2 节和 9.3 节)中的求根方法，来搜索导数的零点，而将函数值的信息撇开。但很快就会放弃这一想法：因为这样做根本无法区分是极大值，还是极小值；而且在初始条件下，如果界点的导数信息说明"下降"应沿划界区间之外的方向，那么从这样的初始条件来分析将根本无从下手。

但在任何时候都不想放弃将极小值严格划界在某一区间内这一策略。保持这个区间的惟一途径就是用函数(而不是导数)信息不断更新它，使其中心点的函数值保持最低。因此，导数起的角色仅仅是帮助我们在区间内选择新的试验点而已。

有一种观点认为"应利用已经得到的一切信息"，即求一个相对高阶的多项式(三次或更高次)，使其满足先前得到的一些函数值和导数值。例如，满足两点的函数值和导数值的三次多项式是惟一的，因此可以转而求这个三次多项式的内插极小值(如果在划界的区间内存在极小值的话)。上述方法的计算公式已由Davidon等人导出，详见本节参考文献[1]。

与上述观点相比，本书中的方法略保守。一旦一种算法确定为超线性收敛，其收敛阶数低一些或高一些都无关紧要。在所遇到的一些实际问题当中，绝大多数函数计算都花在力图使算法足够接近极小值，以确保超线性收敛上。所以我们更关心的是，高阶多项式可能引起各种麻烦(参见图 3.0.1b)以及它们对舍入误差的敏感程度。

因此，只能如下利用导数信息：根据在三个划界(a,b,c)之中心点的导数符号惟一地决定下一试验点是取在(a,b)内，还是在(b,c)内。将这个导数值以及当前仅次于极小值点的导数值用弦截法(逆线性插值)外推到零(注意弦截法本身是超线性收敛的，其收敛阶为 1.618，这又是一黄金分割比率，详见本节参考文献[1]中第 57 页)。对新的试验点，也像在 Brent 方法中那样加上同样的限制：如果该试验点必须被放弃，则对区间进行精细二分。

在一维极小化问题中利用导数信息时我们的确是很保守的。但是，由于某些函数计算出来"导数"并不能与函数值相统一，也不能精确地指向求极小值的路径，它频繁出现的原因常常是因为舍入误差的影响，有时也是由于在计算导数的方法中出现截断误差。

将会看到，下面的程序与上节的程序 brent 在形式上极为相似。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

namespace {
    inline void mov3(DP &a, DP &b, DP &c, const DP d, const DP e,
        const DP f)
    {
        a=d; b=e; c=f;
    }
}

DP NR::dbrent(const DP ax, const DP bx, const DP cx, DP f(const DP),
    DP df(const DP), const DP tol, DP &xmin)
给定一个函数f及其导函数df,并给定一个三个划界点的横坐标ax,bx,cx(满足bx位于ax与cx之间,且f(bx)小于f(ax)及
f(cx))。本程序将用Brent方法的修改形式,即利用导数的Brent方法搜索出f的极小值,结果的相对精度可达tol。极小值
横坐标以xmin返回,最小函数值以dbrent返回(dbrent也是本程序的返回值)。
{
    const int ITMAX=100;
    const DP ZEPS=numeric_limits<DP>::epsilon()*1.0e-3;
    bool ok1,ok2;                                   作为标志,用于记录提议的方法是否
    int iter;                                       可接受。
    DP a,b,d=0.0,d1,d2,du,dv,dw,dx,e=0.0;
    DP fu,fv,fw,fx,olde,tol1,tol2,u,u1,u2,v,w,x,xm;

    下面的注释将指出与程序brent的不同之处,读者应首先阅读该程序。
    a=(ax < cx ? ax : cx);
    b=(ax > cx ? ax : cx);
    x=w=v=bx;
    fw=fv=fx=f(x);
    dw=dv=dx=df(x);                                 整理的信息既包括数值也包括函数值。
    for (iter=0;iter<ITMAX;iter++) {
        xm=0.5*(a+b);
        tol1=tol*fabs(x)+ZEPS;
        tol2=2.0*tol1;
        if (fabs(x-xm) <= (tol2-0.5*(b-a))) {
            xmin=x;
            return fx;
        }
        if (fabs(e) > tol1) {
            d1=2.0*(b-a);                           将d设置为搜索区间之外的值。
            d2=d1;
            if (dw != dx) d1=(w-x)*dx/(dx-dw);      Secant method with one point.
            if (dv != dx) d2=(v-x)*dx/(dx-dv);      And the other.
            到底取哪个d估值?我们的原则是它们位于搜索区间内,且在x的导数所指的一侧。
            u1=x+d1;
            u2=x+d2;
            ok1 = (a-u1)*(u1-b) > 0.0 && dx*d1 <= 0.0;
            ok2 = (a-u2)*(u2-b) > 0.0 && dx*d2 <= 0.0;
            olde=e;                                 前第二步的移动步长。
            e=d;
            if (ok1 || ok2) {                       只取一个d值,若两个都可以,取
                if (ok1 && ok2)                     最小的那个。
                    d=(fabs(d1) < fabs(d2) ? d1 : d2);
                else if (ok1)
                    d=d1;
                else
                    d=d2;
                if (fabs(d) <= fabs(0.5*olde)) {
                    u=x+d;
                    if (u-a < tol2 || b-u < tol2)
                        d=SIGN(tol1,xm-x);
                } else {                            二分法而不是黄金分割法。
                    d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
                    由导数的符号决定取哪个区间段。
                }
            } else {
                d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
            }
        } else {
            d=0.5*(e=(dx >= 0.0 ? a-x : b-x));
        }
```

```
        if (fabs(d) >= tol1) {
            u=x+d;
            fu=f(u);
        } else {
            u=x+SIGN(tol1,d);
            fu=f(u);
            if (fu > fx) {                          若在下降过程中反而出现上升则已
                xmin=x;                             找到极小值点。
                return fx;
            }
        }
        du=df(u);                                   现在对全部信息进行整理。
        if (fu <= fx) {
            if (u >= x) a=x; else b=x;
            mov3(v,fv,dv,w,fw,dw);
            mov3(w,fw,dw,x,fx,dx);
            mov3(x,fx,dx,u,fu,du);
        } else {
            if (u < x) a=u; else b=u;
            if (fu <= fw || w == x) {
                mov3(v,fv,dv,w,fw,dw);
                mov3(w,fw,dw,u,fu,du);
            } else if (fu < fv || v == x || v == w) {
                mov3(v,fv,dv,u,fu,du);
            }
        }
    }
    nrerror("Too many iterations in routine dbrent");
    return 0.0;                                     永远不能到达这里。
}
```

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 55; 454–458. [1]

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), p. 78.

10.4　多维下降单纯形法

从本节开始讨论多维极小化问题，即求有多个自变量的函数之极小值。但本节的内容与后面各节的内容相差较远，本节之后介绍的所有算法都将利用 10.1 节、10.2 节、10.3 节的一维极小化算法作为其求解策略的一部分，而本节的算法完全是封闭式的，不需使用一维算法。

下降的单纯形法由 Nelder 和 Mead[1]提出，这种方法只需计算函数值，不需导数参加运算。就其所需的函数值计算的次数而言，该方法的效率不算很高。而 Powell 算法（参见 10.5 节）对所有可能的应用问题，其求解速度几乎肯定快得多。但是，对于一个计算量小的问题，并且函数图形"可使计算速度变快些"，这时下降单纯法常常是最佳的方法。

下降单纯形法所具有的几何特性，能使这方法描述起来或读起来更易理解，介绍如下：

一个**单纯形**是一个几何形体，它在 N 维情况下，是由 $N+1$ 个点（或顶点）、所有相互连接的线段以及多边形面等组成的几何图形。二维情况单纯形即为三角形；在三维情况，则为一个四面体，但不一定必须是规则的四面体（线性规划中的单纯形方法也利用单纯形的几何概念，但是它与本节讨论的算法没有任何的关联（参见 10.8 节））。一般情况下，只研究非退化的单纯形，它包含一个有限的 N 维内体积。若取其上任意一点作为原点，则剩下的 N 个点定义了向量方向，它们张成 N 维向量空间。

在一维情况可以划界极小值的区间，从而能保证迭代求解的成功。可惜在多维空间中没有类似的方法可寻。在多维情形，我们所能做到的就是给出一个初始预测值，即由自变量组成的一个 N 维向量作为迭代起点，然后设计一种算法不断地进行下降搜索，直到遇到一个（至少是局部）极小值为止，而在搜索期间 N 维形体操作的复杂程度将是难以想像的。

下降的单纯形法必须从 $N+1$ 个点而不仅仅是从单个点开始迭代，这 $N+1$ 个点定义了一个初始单纯形，如果把其中一个点（哪一个无关紧要）作为初始起点 $\mathbf{P}_0$，则其他 N 个点可取为：

$$\mathbf{P}_i = \mathbf{P}_0 + \lambda \mathbf{e}_i \tag{10.4.1}$$

其中 $\mathbf{e}_i$ 为 N 个单位向量，λ 为一常数，它是对问题的特征长度大小的估计值(也可以对每一个向量方向取不同的 λ_i 值)

然后下降的单纯形法将采取一系列步骤。其中绝大多数步骤是，将函数值达到最大的单纯形点(即"最高点")通过单纯形的背向面移到一个较低点。这些步骤称为反射，构造它们的目的是为了保持单纯形的体积不变(从而保持其非退化性)。之后，下降的单纯形法将对单纯形在某个方向上进行扩展以加大步长。当到达"谷底"处时，单纯形将自行做横向收缩，并试图流入谷底。如遇到单纯形需"穿过针眼"的情况，它会从各个方向自行收缩，且自行拉向最低点(即最佳点)附近。程序 amoeba 的取名即描述这一过程。图 10.4.1 概括了几种基本的拉伸动作。

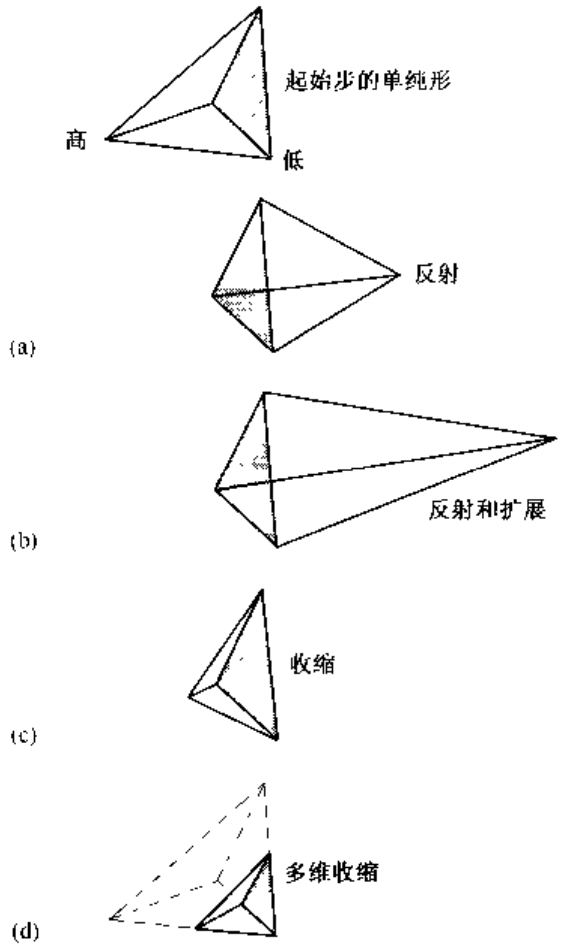


图 10.4.1 下降单纯形法中某步的几种可能结果。在这一步开始时，单纯形(这里是四面体)如顶图所示，在这一步结束之后，单纯形可为一个：(a)从高点的背向反射；(b)从高点的背向反射并扩展；(c)从高点沿一维的收缩；(d)沿各个方向向低点的收缩，算法经过许多步这样的迭代之后，一般总能收敛到函数的极小值

在任何多维极小化程序中，终止准则的确定可能是个很棘手的问题。由于没有划界区间，加上自变量又不只一个，我们不可能像一维那样只要对一个自变量给出某一容差值就可以了。多维情况下一般要在算法中确定一个"循环"或者一个"步骤"，这样在那一步中，当向量移动的距离大小上小于某一容差值 tol 时，算法才有可能终止。当然也可以将终止准则定义为函数值下降的幅度在比值上小于某一容差限 ftol。因为 tol 通常不能小于机器精度的平方根值，所以将 ftol 置为机器精度的数量级是最合适的(也可以稍取大一些以免受舍入误差的影响)。

但是在因为某种原因迭代无法终止的情况下，上述终止准则中却没有一种能够"奏效"。这时不妨从某个已被认为是找到的极小点处开始，重新运行多维极小化程序，但这种做法需要对所有辅助输入参数重新初始化，例如在下降的单纯形法中，需用(10.4.1)式对单纯形 $N+1$ 个顶点中的 N 个进行重新设置，其中 $\mathbf{P}_0$ 为已确认的极小值顶点之一。

“重新开始”不应该很费事，因为毕竟算法已经有了一次收敛的“经历”，它所收敛到的点即是“重新开始”的起点，而“重新开始”也不过是重新使用原算法。

接下来，看一下 N 维情况下的 amoeba 是如何运行的：

```
#include <cmath>
#include "nr.h"
using namespace std;

namespace {
    inline void get_psum(Mat_I_DP &p, Vec_O_DP &psum)
    {
        int i,j;
        DP sum;

        int mpts=p.nrows();
        int ndim=p.ncols();
        for (j=0;j<ndim;j++) {
            for (sum=0.0,i=0;i<mpts;i++)
                sum += p[i][j];
            psum[j]=sum;
        }
    }
}

void NR::amoeba(Mat_IO_DP &p, Vec_IO_DP &y, const DP ftol, DP funk(Vec_I_DP &),
    int &nfunk)
用 Nelder 和 Mead 提出的下降单纯形法求多元函数 funk(x)的极小值，其中 x[0..ndim-1]为 ndim 维向量。矩阵 p[0..ndim]
[0..ndim-1]为输入参数，它的 ndim+1 个 ndim 维的行向量组成初始单纯形的 ndim+1 个顶点。输入项中还包括向量
y[0..ndim]及 ftol，其中 y[0..ndim]的每一分量都需预先初始化为函数 funk 在单纯形的 ndim+1 个顶点处的值，ftol 需预置
为收敛时函数值(注意!)的相对容差限。输出时，p 和 y 又将作为输出参数被赋值为 ndim+1 个新的点，在这些点处函数
值在极小函数值的容差限 ftol 范围内。另一输出参数 nfunk 将给出程序中函数值的求值次数。
{
    const int NMAX=5000;                允许计算函数值的最多次数。
    const DP TINY=1.0e-10;
    int i,ihi,ilo,inhi,j;
    DP rtol,ysave,ytry;

    int mpts=p.nrows();
    int ndim=p.ncols();
    Vec_DP psum(ndim);
    nfunk=0;
    get_psum(p,psum);
    for (;;) {
        ilo=0;
        首先确定最高点(结果最差的点)、次高点和最低点(结果最好的点)。确定的方法是对单纯形中的点进行循环。
        ihi = y[0]>y[1] ? (inhi=1,0) : (inhi=0,1);
        for (i=0;i<mpts;i++) {
            if (y[i] <= y[ilo]) ilo=i;
            if (y[i] > y[ihi]) {
                inhi=ihi;
                ihi=i;
            } else if (y[i] > y[inhi] && i != ihi) inhi=i;
        }
        rtol=2.0*fabs(y[ihi]-y[ilo])/(fabs(y[ihi])+fabs(y[ilo])+TINY);
        计算从最高点到最低点的比率范围，如果合适则返回。
        if (rtol < ftol) {              若返回，则在第一槽内设最佳点和值。
            SWAP(y[0],y[ilo]);
            for (i=0;i<ndim;i++) SWAP(p[0][i],p[ilo][i]);
            break;
        }
        if (nfunk >= NMAX) nrerror("NMAX exceeded");
        nfunk += 2;
        新一轮迭代的开始。首先从高点通过单纯形的表面，以因子 -1 做外推，即从高点将单纯形反射。
        ytry=amotry(p,y,psum,funk,ihi,-1.0);
        if (ytry <= y[ilo])
            得到一个比最佳点稍好的结果，因此用因子 2 再进行一次外推。
            ytry=amotry(p,y,psum,funk,ihi,2.0);
        else if (ytry >= y[inhi]) {
            反射点如果不如次高点，取一中等低的点，即进行一次一维收缩。
```

```
            ysave=y[ihi];
            ytry=amotry(p,y,psum,funk,ihi,0.5);
            if (ytry >= ysave) {        不能去掉高点。最好围绕最低点收缩。
                for (i=0;i<mpts;i++) {
                    if (i != ilo) {
                        for (j=0;j<ndim;j++)
                            p[i][j]=psum[j]=0.5*(p[i][j]+p[ilo][j]);
                        y[i]=funk(psum);
                    }
                }
                nfunk += ndim;          跟踪记录函数值。
                get_psum(p,psum);       重新计算 psum。
            }
        } else --nfunk;                 修正估算值。
    }                                   返回测试处，并进行下一轮迭代。
}
```

```
#include "nr.h"

DP NR::amotry(Mat_IO_DP &p, Vec_O_DP &y, Vec_IO_DP &psum, DP funk(Vec_I_DP &),
    const int ihi, const DP fac)
从高点通过单纯形的表面以因子 fac 试做外推，若新求出的点结果优于高点，则以前者代后者。
{
    int j;
    DP fac1,fac2,ytry;

    int ndim=p.ncols();
    Vec_DP ptry(ndim);
    fac1=(1.0-fac)/ndim;
    fac2=fac1-fac;
    for (j=0;j<ndim;j++)
        ptry[j]=psum[j]*fac1-p[ihi][j]*fac2;
    ytry=funk(ptry);            计算函数在试探点处的值。
    if (ytry < y[ihi]) {        如果试探点的结果优于最高点，则以前者替代后者。
        y[ihi]=ytry;
        for (j=0;j<ndim;j++) {
            psum[j] += ptry[j]-p[ihi][j];
            p[ihi][j]=ptry[j];
        }
    }
    return ytry;
}
```

参考文献和深入阅读

Nelder, J.A., and Mead, R. 1965, *Computer Journal*, vol. 7, pp. 308–313. [1]

Yarbro, L.A., and Deming, S.N. 1974, *Analytica Chimica Acta*, vol. 73, pp. 391–398.

Jacoby, S.L.S, Kowalik, J.S., and Pizzo, J.T. 1972, *Iterative Methods for Nonlinear Optimization Problems* (Englewood Cliffs, NJ: Prentice-Hall).

10.5 多维情况下的方向集(Powell)方法

从 10.1 节到 10.3 节中我们已经知道如何求一元函数的极小值。若从 N 维空间中的某点 $\mathbf{P}$ 出发，沿某一向量的方向 $\mathbf{n}$ 进行搜索，那么任何一个 N 元函数 $f(\mathbf{P})$ 都可沿方向 $\mathbf{n}$ 用一维方法求出极小值。因此可以设想，各种各样的多维极小化方法，都可以用一系列这样的沿某一方向的一维方法组合而成，各种方法之间仅仅由于每一步中所选择的下一搜索方向 $\mathbf{n}$ 的不同而不同。所有这样的方法都要假定存在一个"黑箱"子算法，我们称之为 linmin(本节最后将给出详细的程序)，其定义如下：

> linmin：给定输入参数向量 $\mathbf{P}$ 和 $\mathbf{n}$ 及函数 f，求出使 $f(\mathbf{P}+\lambda\mathbf{n})$ 达到最小的标量 λ，然后以 $\mathbf{P}+\lambda\mathbf{n}$ 替代 $\mathbf{P}$，用 $\lambda\mathbf{n}$ 替代 $\mathbf{n}$，结束。

本节及后面两节中所有的求极小值方法，都将基于这种逐步一维搜索的一般思想(在 10.7 节中的算法不需要精确的一维搜索，它有自己专门的近似一维搜索程序 lnsrch)。本节中将讨论的方法在选择搜索方向

时，不包含函数梯度。读者将注意到，在 linmin 中不需特别说明是否要利用梯度信息，读者可以自己选择，优劣程度由目标函数本身决定。但是大多数读者可能会倾向于在 linmin 中利用梯度信息，而在选择搜索方向时忽略这些信息，因为这样可以大大地减少整个过程的计算量。

但是，如果目标函数的梯度不能计算又该怎么办呢？读者可能首先会想到这样一种简单的方法：取单位向量 $\mathbf{e}_0, \mathbf{e}_1, \ldots, \mathbf{e}_{N-1}$ 作为一个**方向集合**，利用 linmin 算法，沿 $\mathbf{e}_0$ 搜索达到该方向上的极小值，然后再从这个极小值处开始，沿第二个方向 $\mathbf{e}_1$ 搜索，求得函数在 $\mathbf{e}_1$ 方向上的极小值，这样一直下去，根据所需绕整个方向集进行尽可能多次数的循环，直到函数值不再减小。

实际上，这种策略方法对许多函数来说应用的效果并不坏。但它对其他一些函数求解的效率就不很高，这其中的原因非常有趣。让我们来考虑，某个二元函数，其等高线图（水平线）恰好是一个与坐标轴成某一角度的狭长山谷（参见图 10.5.1）。对于这种函数，在每一阶段平行于坐标轴"沿山谷的高度下降方向"搜索的路径，只能是一系列极小的步长。在更一般的 N 维情况，如果函数二阶导数的大小，在某些方向上比在其他方向上大得多，那么为收敛到某点就需要绕全部 N 个基本向量转很多圈才能达到目的。这种情况并不罕见，因此我们不能回避它。

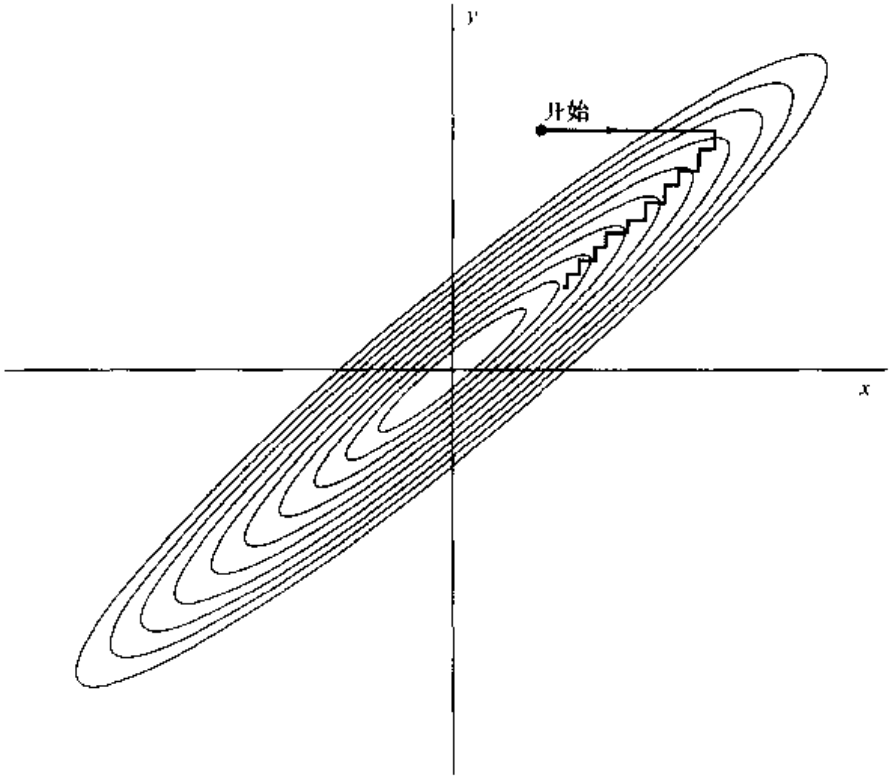


图 10.5.1　在一狭长"山谷"中（图中以轮廓线表示）沿坐标轴方向搜索极小值。除非山谷的方位是最佳的，否则这种逐步一维搜索的方法效率将极低，需要搜索很多步才能达到极小值，而且每步的步长都很小，还要来回穿过主轴多次

由此看来，显然需要比 $\mathbf{e}_i$ 更好的方向集。所有的**方向集方法**都要求在运行当中能给出更新方向集的规则，并使新的方向集满足：(i)在其中某几个相当好的方向上，可使搜索路径沿窄谷前进很长一段距离；(ii)包含某些"互不干扰"的方向，它们具有一种特殊性质，即沿某一个方向的一维搜索，与沿另一方向进行的下一轮一维搜索不会相互干扰。这个比(i)更精细的约束条件可避免绕方向集的循环无休止地进行下去。

10.5.1　共轭方向

这种"互不相关"的方向，更规范一些称为**共轭方向**。在此对这一概念进行数学上的详细说明。

首先，如果沿某一方向 $\mathbf{u}$ 求一函数的极小值，那么在 $\mathbf{u}$ 方向的极小值处，该函数的梯度必然与 $\mathbf{u}$ 垂直；否则，沿 $\mathbf{u}$ 方向还将会有一个非零的方向导数。

下一步,取某一特殊点 $\mathbf{P}$ 作为坐标系的原点,若坐标系的坐标以 $\mathbf{x}$ 标记,则任意函数 f 可用泰勒级数近似为

$$\begin{aligned} f(\mathbf{x}) &= f(\mathbf{P}) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \cdots \\ &\approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2} \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x} \end{aligned} \tag{10.5.1}$$

其中

$$c \equiv f(\mathbf{P}) \qquad \mathbf{b} \equiv -\nabla f|_{\mathbf{P}} \qquad [\mathbf{A}]_{ij} \equiv \left.\frac{\partial^2 f}{\partial x_i \partial x_j}\right|_{\mathbf{P}} \tag{10.5.2}$$

我们称矩阵 $\mathbf{A}$ 为目标函数在 $\mathbf{P}$ 点的黑塞矩阵,$\mathbf{A}$ 中各元素是函数在 $\mathbf{P}$ 点的二阶偏导数。

由(10.5.1)近似式容易求得 f 的梯度为

$$\nabla f = \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \tag{10.5.3}$$

上式说明,满足 $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ 的点可使梯度变为零,即函数在该处将取得极值,在 10.7 节中还要用到这一结论。

当搜索路径沿某一方向移动时,梯度 ∇f 将会如何变化呢?很显然

$$\delta(\nabla f) = \mathbf{A} \cdot (\delta \mathbf{x}) \tag{10.5.4}$$

假设已沿某方向 $\mathbf{u}$ 达到某个极小值,现在打算沿一个新的方向 $\mathbf{v}$ 进行搜索,前提是沿 $\mathbf{v}$ 的移动不破坏沿 $\mathbf{u}$ 的一维搜索,这就是说梯度始终与 $\mathbf{u}$ 保持垂直,即梯度的变化与 $\mathbf{u}$ 垂直,以公式表示,就是(由(10.5.4)式得到)

$$0 = \mathbf{u} \cdot \delta(\nabla f) = \mathbf{u} \cdot \mathbf{A} \cdot \mathbf{v} \tag{10.5.5}$$

如果上式对两个向量 $\mathbf{u}$ 和 $\mathbf{v}$ 成立,则称它们是**共轭的**。若一个向量集的所有向量两两满足这一关系式,那么称这个向量集为共轭集。若沿某共轭方向集的各个方向顺序做一维搜索求极小值,则在任何一个方向上都不需做重复搜索(当然,除非破坏了共轭集的性质,即在一个与方向集不共轭的方向上做一维搜索)。

一个方向集方法成功的关键在于能够构造出一个方向集,其 N 个方向线性无关,且两两共轭。那么对于形如(10.5.1)式的二次式,只需沿 N 个方向做一遍一维搜索,就可以精确地求出其极小值,对于不具有精确二次形式的函数 f,一轮迭代不能精确地求出极小值,但经过几轮一维搜索之后,也能在适当的时候二次收敛到它的极小值。

10.5.2 Powell 二次收敛方法

Powell 首先提出一种方向集方法,该方法可以构造出 N 个两两共轭的方向。其构造过程如下:首先将方向集 $\mathbf{u}_i$ 初始化为坐标轴向量,

$$\mathbf{u}_i = \mathbf{e}_i \qquad i = 0, \ldots, N-1 \tag{10.5.6}$$

然后重复下列各步骤(称为"基本步骤"),直至函数值不再减小:

- 记起始位置为 $\mathbf{P}_0$。
- 对 $i = 0, \ldots, N-1$,将 $\mathbf{P}_i$ 移至目标函数沿 $\mathbf{u}_i$ 方向的极小值点,并记此点为 $\mathbf{P}_{i+1}$。
- 对 $i = 0, \ldots, N-2$,置 $\mathbf{u}_i \leftarrow \mathbf{u}_{i+1}$。
- 置 $\mathbf{u}_{N-1} \leftarrow \mathbf{P}_N - \mathbf{P}_0$。
- 将 $\mathbf{P}_N$ 移至函数在 $\mathbf{u}_{N-1}$ 方向上的极小点,并记该点为 $\mathbf{P}_0$。

Powell 曾于 1964 年,对于形如(10.5.1)式的二次型,上述基本步骤经过 k 轮迭代以后,可以生成一个方向集 $\mathbf{u}_i$,它的最后 k 个方向是两两共轭的。因此,在 N 轮迭代后,总共经过 $N(N+1)$ 次一维搜索,便可以精确地求出一个二次型的极小值。Brent[1]以一种易于理解的形式给出了上述结论的证明。

但是,Powell 的二次收敛算法还存在一个问题,即在每一阶段用 $\mathbf{P}_N - \mathbf{P}_0$ 取代 $\mathbf{u}_0$ 时可能会导致各方向集"互相重叠"而变成线性相关。一旦这样的情况发生,则只能求出全 N 维空间中一个子空间内的函数极小

值，也就是说，得到的结果是错误的。因此，该算法不能用于上述情况。

解决 Powell 算法中的线性相关问题有几种途径，其中包括：

1. 在每 N 轮或 $N+1$ 轮迭代以后，将方向集 $\mathbf{u}_i$ 重新初始化为基向量 $\mathbf{e}_i$。这种方法方便实用。特别是当二次收敛性对于问题本身来说，是一个需着重考虑的因素时（即目标函数非常接近二次型，而且结果的精度要求较高），建议读者不妨试试这种方法。
2. Brent 曾指出，将方向集重置为任意正交矩阵的列向量同样也是很好的办法。他提出将方向集重置为已计算出来的矩阵的 $\mathbf{A}$ 的主方向（他给出了确定它的步骤），而不是放弃已建立的共轭方向上的信息。计算主方向的过程从本质上来看，就是一个奇异值分解的算法（参见 2.6 节）。Brent 还提出另外一些很好的算法技巧，而且他对 Powell 算法所提出的改进也可能是至今为止最佳的一种。具体算法及程序清单请读者参阅本书参考文献[1]，这些内容超出了本书的讨论范围。
3. 还可以放弃二次收敛的性质，而采用一种更有启发性的方案（由 Powell 提出），即沿窄谷找出几个好的方向向量，以代替 N 个必须共轭的方向。下面将要讨论的就是这种方法（这一方法也是 Acton 在本节参考文献[2]中所给出的 Powell 方法的翻版，下文中有几部分内容就是从该书中截取的）

10.5.3　舍弃函数值下降最多的方向

现在打算放弃二次收敛性，这依赖于目标函数本身。有些目标函数呈长且迂回的山谷状。如果一个程序的运行必须绕来绕去向下搜索，那么二次收敛性就显得没有什么特殊的价值了。沿着这一长方向，一种具有二次收敛性的算法总是试图外推到某条抛物线的极小值，而这条抛物线根本不能拟合出来；同时，$N-1$ 个横截方向的共轭性逐渐被这种曲折性破坏。

但是，算法或迟或早总能收敛于一个近似椭圆的极小值（参见(10.5.1)式中当梯度 $\mathbf{b}$ 为零时的情况）。那么，根据精度要求，具有二次收敛性的算法可以节省几倍 N^2 的附加一维搜索计算量，因为二次收敛性在每轮迭代中都使有效位数翻倍。

对 Powell 算法所做的改进，在基本思想上仍是置 $\mathbf{P}_N-\mathbf{P}_0$ 为一新的搜索方向，因为它毕竟是经过对全部 N 个可能的方向试探以后得到的平均方向。对于长方向迂回缓慢的山谷，这个方向可能是沿新长方向的好路径。与原方法比较，这里的改变之处就在于，放弃函数值下降最多的原方向，这看来似乎有些不合理，因此该方向是上一轮迭代中的最好方向。但是，这也可能是我们所添加的这个新方向的一个主要成分。因此，放弃它反而给我们提供了一个绝好机会，可以避免线性相关情况的出现。

但也有几种情况例外。有时候不增加任何新方向反而好些，定义

$$f_0 \equiv f(\mathbf{P}_0) \qquad f_N \equiv f(\mathbf{P}_N) \qquad f_E \equiv f(2\mathbf{P}_N-\mathbf{P}_0) \tag{10.5.7}$$

其中 f_E 为函数在某"外推"点处的值，该点沿所设置的新方向较远。同时定义 Δf 为本轮迭代中沿某特定方向函数值的最大下降量（Δf 为一正值），则有：

1. 若 $f_E \geqslant f_0$，则保持原方向集不变进入下一轮迭代，因为这种情况下平均方向 $\mathbf{P}_N-\mathbf{P}_0$ 已经毫无用处。
2. 若 $2(f_0-2f_N+f_E)[(f_0-f_N)-\Delta f]^2 \geqslant (f_0-f_E)^2\Delta f$，则仍以原方向集进入下一轮迭代。这是因为：(i)沿平均方向，函数值的下降在很大程度上并不是由于某一单个方向上的下降；(ii)沿平均方向有一相当大的二阶导数存在，这说明已经很接近极小值的底部。

上面这种 Powell 方法的程序实现将在下面给出。在该程序中，需说明的是，xi 为一矩阵，其各列为方向集中的元素 $\mathbf{n}_i$。至于其他记号，它们对应的内容是明确的。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::powell(Vec_IO_DP &p, Mat_IO_DP &xi, const DP ftol, int &iter,
    DP &fret, DP func(Vec_I_DP &))
```

求 n 个变量的函数 func 的极小值。输入参数包括初始点 p[0..n-1]、初始矩阵 xi[0..n-1][0..n-1]（其列向量给出初始方向集（通常是 n 个单位向量）），以及函数的容差限 ftol（当某轮迭代中，函数值的下降幅度小于该值时，迭代终止）。输出时，p 被置为所求得的极小点，xi 为迭代终止时的方向集，fret 为返回的 p 点之函数值，iter 为迭代次数。在本程序中调用程序linmin。

```
{
    const int ITMAX=200;                          允许迭代的最大次数。
    const DP TINY=1.0e-25;                        一个小数。
    int i,j,ibig;
    DP del,fp,fptt,t;

    int n=p.size();
    Vec_DP pt(n),ptt(n),xit(n);
    fret=func(p);
    for (j=0;j<n;j++) pt[j]=p[j];                 记录初始点。
    for (iter=0;;++iter) {
        fp=fret;
        ibig=0;
        del=0.0;                                  该值将作为函数值最大的下降量。
        for (i=0;i<n;i++) {                       在每轮迭代中，对方向集中所有方向
            for (j=0;j<n;j++) xit[j]=xi[j][i];进行循环。将该方向复制下来。
            fptt=fret;
            linmin(p,xit,fret,func);              沿该方向求极小。
            if (fptt-fret > del) {                如果它是迄今为止函数值下降量中最大的，
                del=fptt-fret;                    则将其记录下来。
                ibig=i+1;
            }
        }                                         终止准则。
        if (2.0*(fp-fret) <= ftol*(fabs(fp)+fabs(fret))+TINY) {
            return;
        }
        if (iter == ITMAX) nrerror("powell exceeding maximum iterations.");
        for (j=0;j<n;j++) {                       构造外推点和移动的平均方向。将旧的起
            ptt[j]=2.0*p[j]-pt[j];                始点记录下来。
            xit[j]=p[j]-pt[j];
            pt[j]=p[j];
        }
        fptt=func(ptt);                           外推点处的函数值。
        if (fptt < fp) {
            t=2.0*(fp-2.0*fret+fptt)*SQR(fp-fret-del)-del*SQR(fp-fptt);
            if (t < 0.0) {
                linmin(p,xit,fret,func);          找出新方向的极小点，并将这个新方向
                for (j=0;j<n;j++) {               记录下来。
                    xi[j][ibig-1]=xi[j][n-1];
                    xi[j][n-1]=xit[j];
                }
            }
        }
    }
}
```

10.5.4 线性极小化的程序实现

用10.1节～10.3节中所述的一维搜索方法可以正确无误地实现linmin，所不同的是，这里要以向量点**P**(这些点都位于某一给定的方向**n**上)的形式重写那些算法，而不能再用标量x。尽管这项工作很明确，但程序实现却相当繁琐，其间要有许多"for(k=0;k<n;k++)"循环。

由于篇幅所限，本书中不能给出详细的程序清单，这里的linmin只是一种运行良好的框架，它构造了一个一元"模拟"函数fldim，这个函数是目标函数func沿通过p点的直线方向xi上的值。linmin调用我们很熟悉的一维算法mnbrak(参见10.1节)以及brent(参见10.3节)来求fldim的极小值，它同fldim的信息传递是通过全局(外部)变量，而"跨入"mnbrak和brent。linmin在将指针变量传递给fldim时也采用了这种方式，其中这个指针变量指向用户提供的函数。

但linmin作为多维极小化与一维极小化程序之间的接口产生一些不必要的向量复制，这是它惟一的缺陷。一般来讲程序不会因此增加很多的计算量，但这一缺陷却是不能忽视的。

```
#include "nr.h"

int ncom;                                         与fldim通信的全局变量。
DP (*nrfunc)(Vec_I_DP &);
Vec_DP *pcom_p,*xicom_p;

void NR::linmin(Vec_IO_DP &p, Vec_IO_DP &xi, DP &fret, DP func(Vec_I_DP &))
```

给定一个 n 维点 p[0..n-1]和一个 n 维方向 xi[0..n-1]，从 p 点开始沿 xi 方向移动，不断重置 p 点，直至 func(p)取得 xi 方向上的极小值，并置 xi 为 p 所移动的实际向量位移；同时返回的还有 fret，它是函数 func 返回在 p 点的值。实际上该程序完全是通过调用 mnbrak 和 brent 来实现的。

```
{
    int j;
    const DP TOL=1.0e-8;                          传送到 brent 的容差限。
    DP xx,xmin,fx,fb,fa,bx,ax;

    int n=p.size();
    ncom=n;                                       定义全局变量。
    pcom_p=new Vec_DP(n);
    xicom_p=new Vec_DP(n);
    nrfunc=func;
    Vec_DP &pcom=*pcom_p,&xicom=*xicom_p;         解区间的初始预测。
    for (j=0;j<n;j++) {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    ax=0.0;                                       解区间的初始预测。
    xx=1.0;
    mnbrak(ax,xx,bx,fa,fx,fb,fidim);
    fret=brent(ax,xx,bx,fidim,TOL,xmin);
    for (j=0;j<n;j++) {                           构造要返回的向量结果。
        xi[j] *= xmin;
        p[j] += xi[j];
    }
    delete xicom_p;
    delete pcom_p;
}


#include "nr.h"

extern int ncom;                                  在 linmin 中定义。
extern DP (*nrfunc)(Vec_I_DP &);
extern Vec_DP *pcom_p,*xicom_p;

DP NR::fidim(const DP x)
Must accompany linmin                             必须伴随例程 linmin。
{
    int j;

    Vec_DP xt(ncom);
    Vec_DP &pcom=*pcom_p,&xicom=*xicom_p;         使用引用简化编码。
    for (j=0;j<ncom;j++)
        xt[j]=pcom[j]+x*xicom[j];
    return nrfunc(xt);
}
```

参考文献和深入阅读

Brent, R.P 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 7. [1]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 464–467. [2]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), pp. 259–262.

10.6　多维共轭梯度法

现在来考察这样一种情形，即在某一给定的 N 维空间中的点 $\mathbf{P}$ 处，不仅函数值 $f(\mathbf{P})$ 可计算，而且其梯度（即一阶偏导数向量）$\nabla f(\mathbf{P})$ 也是可计算的。

只要经过一些简单的计算论证就可发现使用梯度信息的好处所在：设 f 为一近似二次型，如(10.5.1)式所示：

$$f(\mathbf{x}) \approx c - \mathbf{b}\cdot\mathbf{x} + \frac{1}{2}\mathbf{x}\cdot\mathbf{A}\cdot\mathbf{x} \tag{10.6.1}$$

则 f 中未知参数的个数与 $\mathbf{A}$ 和 $\mathbf{b}$ 中自由参数的个数相同,都为 $\frac{1}{2}N(N+1)$ 个,具有 N^2 数量级。改变其中任意一个参数,都能使极小点的位置发生变化。因此,在收集到与 N^2 等量级的信息量以前,无法找出目标函数的极小值。

在 10.5 节介绍的方向集方法中,利用次数达 N^2 数量级的独立一维搜索,来搜集到必要的信息,而每次一维搜索都需要"几次"(有时是很多次)函数计算。因为每计算一次梯度可以得到 N 个新的信息,因此如果运用得当,做独立一维搜索的次数只需 N 数量级。本节和下节中的算法实际上都是如此。

利用导数信息还可使计算速度改进 N 倍,这是很明显的。粗略估算一下,可以想像,计算梯度的每一分量的时间与计算函数值本身的时间大约是一样的。因此,利用导数和不利用导数的算法都需花费 N^2 数量级的计算量,与计算函数值等价。但是,即使优越性不在于数量级为 N 的计算量,它仍是相当重要的:(i)每计算一次梯度分量,一般可节约的函数值计算不只一次,而是几次,一般可以节约与一次完整的线性极小化的计算等价量;(ii)在函数梯度各分量的计算公式中,常有很高阶的冗余量,如遇这种情况,特别是当函数值的计算中也有很多冗余时,则梯度的计算量能比 N 次函数计算的量少得多。

初学者常认为只要合理地引入梯度信息,就能获得和其他算法差不多好的效果。由这种思路可推出如下的并**不算很好**的算法,即"**最速下降法**":

> 最速下降:以 $\mathbf{P}_0$ 为起始点,沿过 $\mathbf{P}_i$ 的路径,在局部下降梯度 $-\nabla f(\mathbf{P}_i)$ 方向上做一维极小化搜索求得 $\mathbf{P}_{i+1}$,并将 $\mathbf{P}_i$ 移至 $\mathbf{P}_{i+1}$ 点,重复这一过程直到所需的次数。

最速下降法(顺便提一句,它可以完全归结于柯西(Cauchy)算法)所遇到的问题与图 10.5.1 中所示的问题一样。即使目标函数为一严格的二次型,该方法在沿某一狭长山谷下降的过程中,仍执行很多次步长很小的步骤。例如在二维场合,可能会希望迭代第一步就到达谷底,第二步便能直接沿长轴下降,但需记住,新的梯度方向总是与刚刚经过的方向垂直。因此,使用最速下降法时,必须进行适当角度转动,而这种角度转动一般并不能找到极小值(参见图 10.6.1)。

与(10.5.5)推导式所进行的讨论一样,特别希望有一种办法,不是沿新的梯度方向下降,而是构造一个与原梯度方向**共轭**的方向,而且该方向还要尽可能地与在此以前所有已经过的方向共轭。能够实现这一构造过程的方法称为**共轭梯度方法**。

在 2.7 节中也曾讨论过共轭梯度法,那节的内容是通过极小化二次型来求解线性代数方程。虽然共轭梯度法当时是作为一种求解技巧来介绍的,但那套公式对于本节求一个以二次型近似式(10.6.1)的函数的极小值这类问题也同样适用。以任一向量 $\mathbf{g}_0$ 为起点,令 $\mathbf{h}_0=\mathbf{g}_0$,并用下面的递推公式构造两个向量序列:

$$\mathbf{g}_{i+1}=\mathbf{g}_i-\lambda_i\mathbf{A}\cdot\mathbf{h}_i \qquad \mathbf{h}_{i+1}=\mathbf{g}_{i+1}+\gamma_i\mathbf{h}_i \qquad i=0,1,2,\ldots \tag{10.6.2}$$

按这种方式构造出来的向量可同时满足正交性和共轭性:

$$\mathbf{g}_i\cdot\mathbf{g}_j=0 \qquad \mathbf{h}_i\cdot\mathbf{A}\cdot\mathbf{h}_j=0 \qquad \mathbf{g}_i\cdot\mathbf{h}_j=0 \qquad j<i \tag{10.6.3}$$

其中标量 λ_i 和 γ_i 的计算公式是

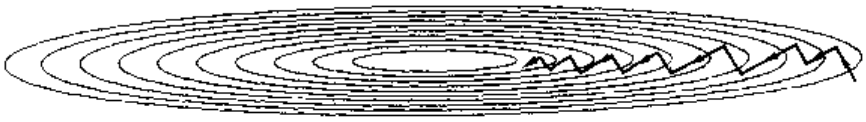
$$\lambda_i=\frac{\mathbf{g}_i\cdot\mathbf{g}_i}{\mathbf{h}_i\cdot\mathbf{A}\cdot\mathbf{h}_i}=\frac{\mathbf{g}_i\cdot\mathbf{h}_i}{\mathbf{h}_i\cdot\mathbf{A}\cdot\mathbf{h}_i} \tag{10.6.4}$$

$$\gamma_i=\frac{\mathbf{g}_{i+1}\cdot\mathbf{g}_{i+1}}{\mathbf{g}_i\cdot\mathbf{g}_i} \tag{10.6.5}$$

(10.6.2)式~(10.6.5)式这组公式与(2.7.32)式~(2.7.35)式的不同之处除了在前者中 $\mathbf{A}$ 是对称的之外,其他仅仅是记号上的不同(针对求函数极小值问题推导这些结论的全部过程在 Polak[1]的书中有详细介绍)。

现在假设(10.6.1)式中的黑塞矩阵 $\mathbf{A}$ 是已知的,那么就可以利用(10.6.2)式来构造一系列的共轭方向 $\mathbf{h}_i$,并沿这些方向进行一维搜索。这样进行 N 次以后,就能求出二次型的极小值。但是,事实上 $\mathbf{A}$ 是未知的。

下面给出一个著名的定理来解决上面的问题。假设在某点 $\mathbf{P}_i$ 恰有 $\mathbf{g}_i=-\nabla f(\mathbf{P}_i)$,其中 f 如(10.6.1)式所示。再假设从 $\mathbf{P}_i$ 开始沿 $\mathbf{h}_i$ 方向到达 f 的某个局部极小值点 $\mathbf{P}_{i+1}$,并置 $\mathbf{g}_{i+1}=-\nabla f(\mathbf{P}_{i+1})$,那么这个 $\mathbf{g}_{i+1}$ 应该与由(10.6.2)式构造的向量完全相同(至此,并没有利用 $\mathbf{A}$ 构造出 $\mathbf{g}_i$)。



(a)

(b)

图 10.6.1　(a) 对一狭长函数使用最速下降法，虽然该方法比图 10.5.1 所示的方法快一些，但仍不能作为一种好的算法，在到达谷底之前也要经过很多步；(b)将最速下降法执行过程中的某一步放大来看，从局部梯度方向出发(这一方向与轮廓线的切线垂直)，经过一维搜索到达局部极小值，此时一维搜索路径与那里的轮廓线的切线平行

证明：由(10.5.3)式得到 $\mathbf{g}_i = -\mathbf{A}\cdot\mathbf{P}_i + \mathbf{b}$ 及

$$\mathbf{g}_{i+1} = -\mathbf{A}\cdot(\mathbf{P}_i + \lambda\mathbf{h}_i) + \mathbf{b} = \mathbf{g}_i - \lambda\mathbf{A}\cdot\mathbf{h}_i \tag{10.6.6}$$

其中λ的取值达到该方向的极小值。但在这个极小值点，$\mathbf{h}_i\cdot\nabla f = -\mathbf{h}_i\cdot\mathbf{g}_{i+1} = 0$。将后面的这个等式与(10.6.6)式联立可以很容易求 λ，而结果恰好是(10.6.4)式所示的表达式。再以这个 λ 值代入(10.6.6)式中，所得表达式与(10.6.2)式完全相同。证毕。

这样，对于既不需要已知黑塞矩阵 **A**，甚至也不需对其花费存储空间的算法，我们已找到了它的理论基础。只需利用一维搜索和梯度向量的计算，并设置一个辅助向量存放序列 **g** 的当前向量，方向序列 $\mathbf{h}_i$ 就可以被构造出来。

以上所讨论的算法是共轭梯度法的 Fletcher-Reeves 版本的原形，后来 Polak 和 Ribiere 对这一版本做了一些小改动，虽然改动不大，但有些情况下其意义却非同一般。他们提出用下列表达式

$$\gamma_i = \frac{(\mathbf{g}_{i+1} - \mathbf{g}_i)\cdot\mathbf{g}_{i+1}}{\mathbf{g}_i\cdot\mathbf{g}_i} \tag{10.6.7}$$

来替代(10.6.5)式。介绍到这里，读者肯定会问，根据正交条件(10.6.3)，这两个表达式不是等价的吗？不错，对精确二次型它们是等价的，但实际上，目标函数常常不是精确的二次型，因此在求出假设的极小值后，还需另一轮迭代。实践证明[2]，Polak 和 Ribiere 提出的方法恰恰可以非常漂亮地实现向深入迭代的过渡：一旦算法偏离路径，则重置 **h** 为沿局部梯度下降的方向，这就相当于重新开始执行共轭梯度法。

下面这段程序是 Polak-Ribiere 方法的一个变形版本，只要改变程序的某一行，它就变成了 Fletcher-Reeves 方法。该程序中利用了函数 func(p)(其中 p[0..n-1]是一个 n 维向量)和函数 dfunc(p,df)，其中 dfunc(p,df)用来计算输入点 p 处的向量梯度 df[0..n-1]。

程序中还调用了 linmin 来执行一维搜索。正如以前讨论过的一样，需把 linmin 中的 brent 改为 dbrent，也就是说在一维搜索中需要利用梯度信息。请读者参见下面程序中的注释说明。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::frprmn(Vec_IO_DP &p, const DP ftol, int &iter, DP &fret,
    DP func(Vec_I_DP &), void dfunc(Vec_I_DP &, Vec_O_DP &))
```

给定起始点 p[0..n-1]，利用程序 dfunc 提供的梯度信息，对函数 func 用 Fletcher-Reeves-Polak-Ribiere 方法求极小值。函数值的收敛容差限以输入参数 ftol 给出。返回项包括 p(极小值点的坐标)，iter(总共执行的迭代次数)以及 fret(函数的极小值)。本程序还调用了 linmin 执行一维搜索。

```
{
    const int ITMAX=200;
    const DP EPS=1.0e-18;
    ITMAX 为允许迭代的最大次数. EPS 为对收敛到零函数值的特殊情况进行调整的最小数.
    int j,its;
    DP gg,gam,fp,dgg;

    int n=p.size();
    Vec_DP g(n),h(n),xi(n);
    fp=func(p);                                 初始化。
    dfunc(p,xi);
    for (j=0;j<n;j++) {
        g[j] = -xi[j];
        xi[j]=h[j]=g[j];
    }
    for (its=0;its<ITMAX;its++) {               迭代循环。
        iter=its;
        linmin(p,xi,fret,func);                 下一个语句是正常返回;
        if (2.0*fabs(fret-fp) <= ftol*(fabs(fret)+fabs(fp)+EPS))
            return;
        fp=fret;
        dfunc(p,xi);
        dgg=gg=0.0;
        for (j=0;j<n;j++) {
            gg += g[j]*g[j];
//          dgg += xi[j]*xi[j];                 Fletcher-Reeves 方法。
            dgg += (xi[j]+g[j])*xi[j];          Polak-Ribiere 方法。
        }
        if (gg == 0.0)                          不大可能出现的情况: 如果梯度恰为 0,
            return;                             则可以结束返回。
        gam=dgg/gg;
        for (j=0;j<n;j++) {
            g[j] = -xi[j];
            xi[j]=h[j]=g[j]+gam*h[j];
        }
    }
    nrerror("Too many iterations in frprmn");
}
```

10.6.1 有关利用导数的线性极小化之说明

请重新阅读一下本章 10.5 节最后一部分的内容,下面我们所要做的事情与它完全相同,只是在执行线性极小化过程中利用了导数信息。

把修改后的 linmin 取名为 dlinmin,下面给出在 dlinmin 中所调用的程序dfldim。

```
#include "nr.h"

int ncom;                                       与 dfldim 全局变量通信。
DP (*nrfunc)(Vec_I_DP &);
void (*nrdfun)(Vec_I_DP &, Vec_O_DP &);
Vec_DP *pcom_p,*xicom_p;

void NR::dlinmin(Vec_IO_DP &p, Vec_IO_DP &xi, DP &fret, DP func(Vec_I_DP &),
    void dfunc(Vec_I_DP &, Vec_O_DP &))
给定一n 维点 p[0..n-1]和一n 维方向 xi[0..n-1],沿 xi 方向不断移动 p 点,直至函数 func(p)取得 xi 方向上的极小值。
返回时置 p 为极小值点,并置 xi 为 p 所移动的实际向量位移。同时返回的还有 fret,它是函数 func 在程序返回的 p 点所取
的值。实际上该程序完全是通过调用 mnbrak 和 dbrent 来实现的。
{
    const DP TOL=2.0e-8;                        这是传送到 dbrent 的容差限.
    int j;
    DP xx,xmin,fx,fb,fa,bx,ax;

    int n=p.size();
    ncom=n;                                     定义全局变量。
    pcom_p=new Vec_DP(n);
    xicom_p=new Vec_DP(n);
    nrfunc=func;
    nrdfun=dfunc;
```

```
    Vec_DP &pcom=*pcom_p,&xicom=*xicom_p;      定义全局变量。
    for (j=0;j<n;j++) {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    ax=0.0;                                    划界的初始预测。
    xx=1.0;
    mnbrak(ax,xx,bx,fa,fx,fb,f1dim);
    fret=dbrent(ax,xx,bx,f1dim,df1dim,TOL,xmin);
    for (j=0;j<n;j++) {                        构造要返回的向量结果。
        xi[j] *= xmin;
        p[j] += xi[j];
    }
    delete xicom_p;
    delete pcom_p;
}
```

```
#include "nr.h"

extern int ncom;                               在 dlinmin 中定义。
extern DP (*nrfunc)(Vec_I_DP &);
extern void (*nrdfun)(Vec_I_DP &, Vec_O_DP &);
extern Vec_DP *pcom_p,*xicom_p;

DP NR::df1dim(const DP x)
{
    int j;
    DP df1=0.0;
    Vec_DP xt(ncom),df(ncom);

    Vec_DP &pcom=*pcom_p,&xicom=*xicom_p;      使用引用简化编码。
    for (j=0;j<ncom;j++) xt[j]=pcom[j]+x*xicom[j];
    nrdfun(xt,df);
    for (j=0;j<ncom;j++) df1 += df[j]*xicom[j];
    return df1;
}
```

参考文献和深入阅读

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), §2.3. [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1.7 (by K.W. Brodlie). [2]

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §8.7.

10.7　多维变尺度法

变尺度法(有时也称为**拟牛顿法**)的目标是从逐步的线性极小化中不断积累信息,使得算法在经过 N 次线性极小化以后,便可精确地收敛于一个 N 元二次型的极小值,这与前面讨论的共轭梯度法是完全相同的。由此看来,对于更一般的光滑函数,变尺度法也能达到二次收敛。

无论是变尺度法还是共轭梯度法都要求能计算出目标函数在任意点的梯度(即一阶导数向量)。与共轭梯度法所不同的是,变尺度法需要不断存储和更新已有的信息。这种方法所需的不是数量级为 N(这里 N 为维数)的中间存储量,而是一个 $N \times N$ 大小的矩阵。一般情况下,只要 N 不太大,这样大小的矩阵是根本不成问题的。

另一方面,根据我们所掌握的情况,除了历史原因以外,变尺度法与共轭梯度法相比较并不拥有任何明显优势。由于变尺度法建立得较早而且推广得较为广泛,因此它现在已经能满足众多使用者的要求。除此之外,一些用变尺度法开发的优秀软件实施被用于解决舍入误差的极小化、特殊条件的处理等高难度问题。虽然本书倾向于使用变尺度法,而不是共轭梯度法,但没有理由把这一习惯强加给读者。

变尺度法主要分为两大类,一类是 Davidon-Fletcher-Powell(DFP)算法(有时简称 Fletcher-Powell 算法);另一

类是 Broyden-Fletcher-Goldfarb-Shanno(BFGS)算法。BFGS 算法与 DFP 算法的不同之处仅仅在于舍入误差、收敛容差限以及一些类似的细节问题上。它们都超出了本书的讨论范围,参阅本节参考文献[1,2]。但是人们已经普遍认识到,从经验上来看,BFGS 方法在这些细节问题上比 DFP 方法要好,因此本节将着重研究 BFGS 方法。

和前面一样,假设任意函数 $f(\mathbf{x})$ 可用(10.6.1)式那样的二次型进行局部的近似,但是对二次型的参数 $\mathbf{A}$ 和 $\mathbf{b}$ 的值仍一无所知,仅仅只能从函数值的计算和线性极小化中找到一些有关的信息。

变尺度法的基本思想就是,用逐步迭代的方式求出逆黑塞矩阵 $\mathbf{A}^{-1}$ 的一个足够好的近似矩阵,即构造出一个矩阵序列 $\mathbf{H}_i$,使其具有如下性质:

$$\lim_{i\to\infty}\mathbf{H}_i=\mathbf{A}^{-1} \tag{10.7.1}$$

如果这个极限值经过 N 次而不是 ∞ 次迭代以后就能得到,则更好。

现在来解释一下为什么变尺度法有时也被称为"拟牛顿法"。考虑一种求函数极小值的方法,即用牛顿法搜索该函数导数的零点。在当前点 $\mathbf{x}_i$ 的某邻域附近,有

$$f(\mathbf{x})=f(\mathbf{x}_i)+(\mathbf{x}-\mathbf{x}_i)\cdot\nabla f(\mathbf{x}_i)+\frac{1}{2}(\mathbf{x}-\mathbf{x}_i)\cdot\mathbf{A}\cdot(\mathbf{x}-\mathbf{x}_i) \tag{10.7.2}$$

进而有下式成立:

$$\nabla f(\mathbf{x})=\nabla f(\mathbf{x}_i)+\mathbf{A}\cdot(\mathbf{x}-\mathbf{x}_i) \tag{10.7.3}$$

在牛顿法中是通过置 $\nabla f(\mathbf{x})=0$ 来确定下一个迭代点的,即

$$\mathbf{x}-\mathbf{x}_i=-\mathbf{A}^{-1}\cdot\nabla f(\mathbf{x}_i) \tag{10.7.4}$$

上式中左端项是在达到精确极小以前需要经过的有限步长,而右端项在求出精确的 $\mathbf{H}\approx\mathbf{A}^{-1}$ 之后便可成为已知项。

之所以在"牛顿法"之前冠以"拟"字,是因为算法中使用的不是函数 f 的真正黑塞矩阵,而是该矩阵的当前近似矩阵。这样做的效果常常比利用真正的黑塞矩阵效果要好。这个结论并不难理解。让我们考虑一下函数 f 在 $\mathbf{x}_i$ 处的**下降方向**,沿这些方向 $\mathbf{p}$,函数 f 的值是减小的,即 $\nabla f\cdot\mathbf{p}<0$。为了让牛顿方向(10.7.4)成为下降方向,必须满足

$$\nabla f(\mathbf{x}_i)\cdot(\mathbf{x}-\mathbf{x}_i)=-(\mathbf{x}-\mathbf{x}_i)\cdot\mathbf{A}\cdot(\mathbf{x}-\mathbf{x}_i)<0 \tag{10.7.5}$$

也就是说 $\mathbf{A}$ 必须是正定矩阵。一般来说,在远离极小值处,并不能保证黑塞矩阵是正定的,对真正的黑塞矩阵,用实际的牛顿步长只能找到函数值增加的点,拟牛顿法的基本思想是,先找一个正定对称的矩阵作为 $\mathbf{A}$ 的初始近似矩阵,然后从这个近似矩阵出发,构造一个矩阵序列 $\mathbf{H}_i$,构造的原则是保证每个 $\mathbf{H}_i$ 都是正定和对称的。在离极小值较远处,这种构造方法能够保证算法沿下降方向进行迭代,而在接近极小值的地方,更新后的公式渐近真正的黑塞矩阵。这样可以获得牛顿法所具备的二次收敛性。

当我们没有充分接近极小值的时候,即使是对正定矩阵 $\mathbf{A}$ 采用完全的牛顿步长 $\mathbf{p}$ 也不一定能使函数值下降,原因是移动的距离可能过远,因而使二次近似变得毫无意义。惟一可以确定的是,在沿牛顿方向移动过程中,函数 f 最初是下降的。这种情况下,可以使用 9.7 节中的划界策略,可以利用它沿牛顿**步长** $\mathbf{p}$ 的方向选择一轮迭代步长,但并不能总这样做。

本书将省略 DFP 算法的严格推导,有兴趣的读者请参见本节参考文献[3]中的详细推导过程。根据 Brodlie[2],将在下文中给出该方法的启发式机理。

以 $\mathbf{x}_{i+1}$ 替代(10.7.4)式中的 $\mathbf{x}_i$,再用所得方程减去原来的(10.7.4)式,有

$$\mathbf{x}_{i+1}-\mathbf{x}_i=\mathbf{A}^{-1}\cdot(\nabla f_{i+1}-\nabla f_i) \tag{10.7.6}$$

其中 $\nabla f_i\equiv\nabla f(\mathbf{x}_i)$。从 $\mathbf{x}_i$ 得到 $\mathbf{x}_{i+1}$ 之后,很自然地希望新求出的估计值 $\mathbf{H}_{i+1}$ 能够像 $\mathbf{A}^{-1}$ 一样满足(10.7.6)式,即

$$\mathbf{x}_{i+1}-\mathbf{x}_i=\mathbf{H}_{i+1}\cdot(\nabla f_{i+1}-\nabla f_i) \tag{10.7.7}$$

不仅如此,它的修正公式还应具有"$\mathbf{H}_{i+1}=\mathbf{H}_i+$ 修正值"这种形式。

它周围有什么"对象"可以用来构造修正项呢?最明显的是 $\mathbf{x}_{i+1}-\mathbf{x}_i$ 和 $\nabla f_{i+1}-\nabla f_i$ 这两个向量,还有外

加矩阵序列 $\mathbf{H}_i$。由这些项来构造矩阵不可能有无穷多种方式，特别是在还要满足(10.7.7)式的情况下。

DFP 方法的修正公式给出了其中一种构造方法，它具有如下形式：

$$\mathbf{H}_{i+1} = \mathbf{H}_i + \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i) \otimes (\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} - \frac{[\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)] \otimes [\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)} \tag{10.7.8}$$

上式中$\otimes$表示两个向量的"外"积或"直接"积，其结果为一矩阵：$\mathbf{u} \otimes \mathbf{v}$ 的第 i 行第 j 列元素为 $u_i v_j$(读者可以证明(10.7.8)式确实满足(10.7.7)式)。

BFGS 的修正公式与 DFP 值的形式相同，只是多增加了一项，

$$\cdots + [(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)]\, \mathbf{u} \otimes \mathbf{u} \tag{10.7.9}$$

其中 $\mathbf{u}$ 定义为如下向量：

$$\mathbf{u} \equiv \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i) \cdot (\nabla f_{i+1} - \nabla f_i)} - \frac{\mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)}{(\nabla f_{i+1} - \nabla f_i) \cdot \mathbf{H}_i \cdot (\nabla f_{i+1} - \nabla f_i)} \tag{10.7.10}$$

同样可以证明(10.7.10)式也满足(10.7.7)式。

只要 f 是一个二次型，无论在(10.7.8)式中加或不加(10.7.9)式这一项，都能得到"$\{\mathbf{H}_i\}$ 在 N 步之内必收敛于 $\mathbf{A}^{-1}$"的结论。对于这个结论，读者可以尽管放心或可参阅本节参考文献[3]。

下面的这段源代码 dfpmin 是拟牛顿法的程序实现，dfpmin 中需调用 9.7 节中的 lnsrch。在 9.7 节中 newt 的结尾处我们曾经提到，如果变量的比例选取不当有可能导致算法失败，下面的程序也是如此。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

void NR::dfpmin(Vec_IO_DP &p, const DP gtol, int &iter, DP &fret,
    DP func(Vec_I_DP &), void dfunc(Vec_I_DP &, Vec_O_DP &))
给定起始点 p[0..n-1](n 维向量)，利用程序 dfunc 提供的梯度信息，用修改后的 Davidon-Fletcher-Powell 方法(DFP)，即
Broyden-Fletcher-Goldfarb-Shanno(BFGS)方法对函数 func 求极小值。梯度零点的收敛容差限以输入参数 gtol 给出。程序的返
回值包括 p[0..n-1](极小值点的位置坐标)，iter(总共执行的迭代次数)以及 fret(函数的极小值)。本程序中调用了程序
lnsrch 执行近似线性极小化。
{
    const int ITMAX=200;
    const DP EPS=numeric_limits<DP>::epsilon();
    const DP TOLX=4*EPS,STPMX=100.0;
    其中 ITMAX 是允许迭代的最大次数；EPS 是机器精度；TOLX 是 x 值的收敛准则；STPMX 是线性搜索中所允许的最大
    步长。
    bool check;
    int i,its,j;
    DP den,fac,fad,fae,fp,stpmax,sum=0.0,sumdg,sumxi,temp,test;

    int n=p.size();
    Vec_DP dg(n),g(n),hdg(n),pnew(n),xi(n);
    Mat_DP hessin(n,n);
    fp=func(p);                                 计算起始点函数和梯度，
    dfunc(p,g);
    for (i=0;i<n;i++) {                         并将逆 Hesse 矩阵初始化为单位矩阵。
        for (j=0;j<n;j++) hessin[i][j]=0.0;
        hessin[i][i]=1.0;
        xi[i] = -g[i];                          初始化一维搜索方向。
        sum += p[i]*p[i];
    }
    stpmax=STPMX*MAX(sqrt(sum),DP(n));
    for (its=0;its<ITMAX;its++) {               迭代循环。
        iter=its;
        lnsrch(p,fp,g,xi,pnew,fret,stpmax,check,func);
        在 lnsrch 中计算新的函数值并存在 fp 中以备下一轮线性搜索时使用。
```

```
        fp=fret;
        for (i=0;i<n;i++) {
            xi[i]=pnew[i]-p[i];                 更新搜索方向和当前点。
            p[i]=pnew[i];
        }
        test=0.0;                               检验 Δx 的收敛性。
        for (i=0;i<n;i++) {
            temp=fabs(xi[i])/MAX(fabs(p[i]),1.0);
            if (temp > test) test=temp;
        }
        if (test < TOLX)
            return;
        for (i=0;i<n;i++) dg[i]=g[i];           将原梯度存放起来。
        dfunc(p,g);                             计算新的梯度。
        test=0.0;                               检验零梯度的收敛性。
        den=MAX(fret,1.0);
        for (i=0;i<n;i++) {
            temp=fabs(g[i])*MAX(fabs(p[i]),1.0)/den;
            if (temp > test) test=temp;
        }
        if (test < gtol)
            return;
        for (i=0;i<n;i++) dg[i]=g[i]-dg[i];     计算梯度的差值。
        for (i=0;i<n;i++) {                     并用该差值乘以当前矩阵。
            hdg[i]=0.0;
            for (j=0;j<n;j++) hdg[i] += hessin[i][j]*dg[j];
        }
        fac=fae=sumdg=sumxi=0.0;                计算分母的点数。
        for (i=0;i<n;i++) {
            fac += dg[i]*xi[i];
            fae += dg[i]*hdg[i];
            sumdg += SQR(dg[i]);
            sumxi += SQR(xi[i]);
        }
        if (fac > sqrt(EPS*sumdg*sumxi)) {      若 fac 没有足够的正数特性，这一步
            fac=1.0/fac;                        跳过不执行。
            fad=1.0/fae;
            BFGS 比 DFP 多出的那一项。
            for (i=0;i<n;i++) dg[i]=fac*xi[i]-fad*hdg[i];
            for (i=0;i<n;i++) {                 BFGS 修正公式。
                for (j=i;j<n;j++) {
                    hessin[i][j] += fac*xi[i]*xi[j]
                        -fad*hdg[i]*hdg[j]+fae*dg[i]*dg[j];
                    hessin[j][i]=hessin[i][j];
                }
            }
        }
        for (i=0;i<n;i++) {                     计算下一搜索方向。
            xi[i]=0.0;
                for (j=0;j<n;j++) xi[i] -= hessin[i][j]*g[j];
            }
        }                                       并返回执行下一轮迭代。
        nrerror("too many iterations in dfpmin");
    }
```

这里说明一点，由 dfpmin 实现的拟牛顿法只需近似的线性极小化（由程序 lnsrch 执行）就能得到很好的结论；而 10.5 节中的 powell 以及 10.6 节中的 frpmin 则需要比较精确的 linmin 求执行线性最小化。

10.7.1 深入讨论:变尺度法的进一步实现

由于舍入误差的影响有时会导致矩阵 $\mathbf{H}_i$ 接近奇异或非正定，这种情况虽属少见，但却无法避免其发生的可能。这时，所推测的搜索方向可能并不是函数值下降的方向，而且当前 $\mathbf{H}_i$ 的近似奇异性也会“移传”给定它后面的 $\mathbf{H}_i$，因此其后果将是很严重的。

这类问题的补救办法非常简单，在 10.4 节也曾提到过：一旦遇到上述情况发生的可能，应从已求出的极小点处**重新开始**执行该算法，看看极小点会不会移到别处去。这种处理方法虽然简单易行，但却不够精细。现代变尺度法在处理这一问题的方式上要比它复杂得多。

让我们从另一角度分析算法。如果不去构造 $\mathbf{A}^{-1}$ 的近似矩阵，而去构造矩阵 $\mathbf{A}$ 本身的近似也是可能的。

而且这样一来，不需直接计算(10.7.4)式的左边项，只要求解如下的线性方程组即可：

$$\mathbf{A}\cdot(\mathbf{x}-\mathbf{x}_i)=-\nabla f(\mathbf{x}_i) \tag{10.7.11}$$

这种想法初看很差，因为求解(10.7.11)式的计算量已达到 N^3 数量级，而且也看不出它对舍入误差问题的解决有何帮助。其实其中的技巧在于，存储的是矩阵 **A** 三角分解(即 Cholesky 分解，参见 2.9 节)后的矩阵，而不是存储 **A** 本身。**A** 的 Cholesky 分解修正公式是 N^2 数量级，即使在有限舍入误差存在的情况下，它也能确保矩阵正定性和非退化性质。该修正公式是由 Gill 和 Murray 提出的，有兴趣的读者请参阅本节参考文献[1,2]。

参考文献和深入阅读

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*; reprinted 1996 (Philadelphia: S.I.A.M.). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.1, §§3–6 (by K. W. Brodlie). [2]

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press), pp. 56ff. [3]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 467–468.

10.8　线性规划和单纯形法

线性规划（有时也称为**线性优化**）所讨论的问题可归纳为如下形式：对 N 个自变量 $x_0,\ldots,x_{N-1}$ 求函数

$$z=a_{00}x_0+a_{01}x_1+\cdots+a_{0,N-1}x_{N-1} \tag{10.8.1}$$

的极大值。条件是满足 N 个基本约束条件

$$x_0\geqslant 0,\quad x_1\geqslant 0,\quad \ldots\quad x_{N-1}\geqslant 0 \tag{10.8.2}$$

及 $M=m_1+m_2+m_3$ 个附加约束条件。在附加约束中，有 m_1 个具有形式

$$a_{i0}x_0+a_{i1}x_1+\cdots+a_{i,N-1}x_{N-1}\leqslant b_i \qquad (b_i\geqslant 0) \qquad i=1,\ldots,m_1 \tag{10.8.3}$$

m_2 个具有形式

$$a_{j0}x_0+a_{j1}x_1+\cdots+a_{j,N-1}x_{N-1}\geqslant b_j\geqslant 0 \qquad j=m_1+1,\ldots,m_1+m_2 \tag{10.8.4}$$

其余 m_3 个具有形式

$$\begin{gathered}a_{k0}x_0+a_{k1}x_1+\cdots+a_{k,N-1}x_{N-1}=b_k\geqslant 0\\ k=m_1+m_2+1,\ldots,m_1+m_2+m_3\end{gathered} \tag{10.8.5}$$

上面各式中，参数 a_{ij} 可正可负，也可以是零，而所有参数 b 则规定为非负数(如上述三式中最末尾的不等式所示)，但这只是一种惯例而已，因为可以利用 -1 去乘任何一个不等式的两端。此外，线性规划问题对约束条件的个数 M 没有特殊限制，无论它是小于、等于还是大于变量个数 N。

满足约束条件(10.8.2)式～(10.8.5)式的任一组 $x_0,\ldots,x_{N-1}$ 值组成的向量称为**可行向量**；欲求极大值的函数称为**目标函数**；使目标函数取得极大值的可行向量为**最优可行向量**。但是，有些情况下最优可行向量可能不存在，原因有两个：(i)根本没有可行向量存在，即给定的约束条件之间是相互排斥的；(ii)目标函数没有极大值，也就是说在 N 维空间中的某个方向上，一个或多个变量可无限增大而约束条件仍旧能被满足，此时目标函数的值是无界的。

如大家所见，线性规划的研究对象中包含了许许多多的记号和术语。实际上，线性规划中的基本思想相当简单。为简明起见，先通过几个特殊的例子来阐明线性规划的基本问题，然后再将它们推广至一般。

为什么说线性规划如此重要呢？首先，因为"非负性"是对某类变量 x_i 的一种非常普遍的约束。这些 x_i 代表的是一些实际物品的明确数量，这种"非负性"得到的便是约束条件(10.8.2)。其次，因为人们常常对一些由人类或自然界施加的加法(线性)限制或界限感兴趣，如最低营养需求、最大可支付开销、现有人力或资金的最高使用限度等，这些都可以抽象成为(10.8.3)式～(10.8.5)式的形式。再次，由于目标函数极有可能

是线性的,或至少可有线性函数来近似(因为这才是线性规划可以解决的问题),我们才有如(10.8.1)式形式的目标函数。总之,有关线性规划应用情况的综述,可以参见本节参考文献[1]。

下面给出一个线性规划问题的具体例子。在这个例子中 $N=4$, $m_1=2$, $m_2=m_3=1$,从而有 $M=4$:

$$\text{极大化} \qquad z = x_0 + x_1 + 3x_2 - \tfrac{1}{2}x_3 \tag{10.8.6}$$

其中所有变量 x 非负,且

$$\begin{aligned} x_0 + 2x_2 &\leqslant 740 \\ 2x_1 - 7x_3 &\leqslant 0 \\ x_1 - x_2 + 2x_3 &\geqslant \tfrac{1}{2} \\ x_0 + x_1 + x_2 + x_3 &= 9 \end{aligned} \tag{10.8.7}$$

这个问题的解为(取两位有效数字)$x_0=0$, $x_1=3.33$, $x_2=4.73$, $x_3=0.95$。在本节余下的篇幅中,将详细介绍这个解是如何求得的。但在开始之前,先用图 10.8.1 总结一下前面提到的所有术语。

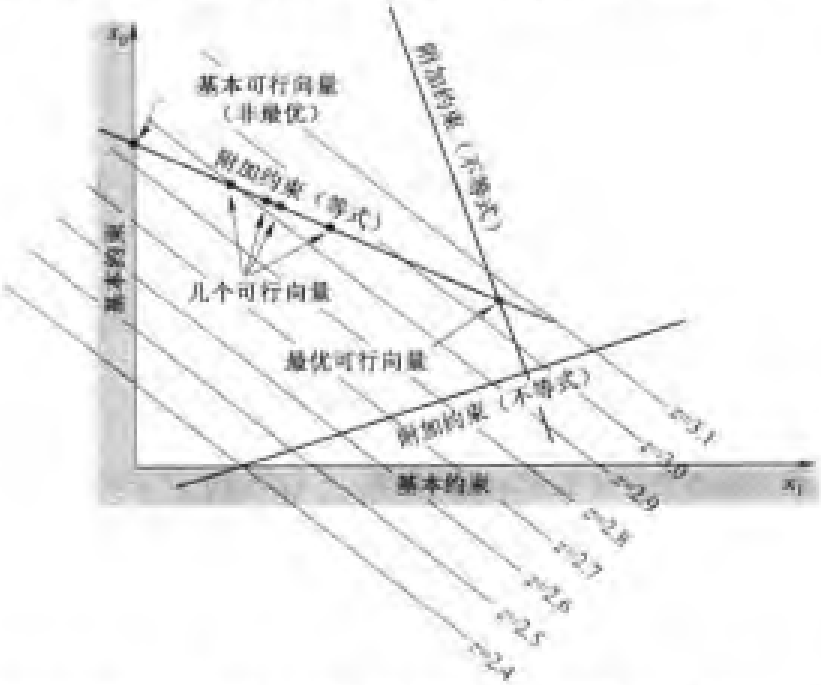


图 10.8.1 线性规划的基本概念。图中所示为两个自变量的情况,目标函数 z 由等高线给出。基本约束是要求 x_0 和 x_1 为正数,而附加约束则将问题的解划定在某一区域内(指不等式约束)或局限在降低了维数的面上(指等式约束)。可行向量要求满足所有的约束条件,基本可行向量也需位于允许区域的边界上。本节将介绍的单纯形法是在所有基本可行向量中间进行转移,直到找到最优可行向量为止

10.8.1 线性规划基本定理

假设从待选向量的全 N 维空间出发,然后(至少在想像中)把由约束条件依次淘汰的区域"挖掉"。由于约束条件是线性的,因而由这一过程产生出来的边界将是一个平面或者一个超平面。形如(10.8.5)式的等式约束条件迫使可行区域落在降低了维数的超平面上,而不等式约束条件仅仅把可行区域划分成一些满足条件和不满足条件的小块。

当把所有约束条件都施加在 N 维空间之后,只有两种可能发生,一种是剩下了某个可行域,另一种是根本没有可行向量存在。因为在第一种情况下,可行域是由许多超平面围成的,它在几何上就形成了一种凸多面体或单纯形(参考 10.4 节)。如果确实有某个可行域存在的话,那么最优可行向量不会完全位于其内部某处而脱离开边界。因为目标函数是线性的,也就是说它总会有一非零的梯度存在,而这反过来又说明,沿梯度向上总能使目标函数的值增加,直到碰到某一边界为止。

由于几何区域边界的维数比其内部的维数少一维，因此，可以沿投影在边界的梯度一直向上，直至到达界面的边缘，然后再沿这个边缘向上，依次类推，不管穿过多少维数的数目，但最后都能到达某一点，这一点是原单纯形的某个**顶点**。因为它所有 N 个坐标都是已定义过的，因此这点必为同时满足某 N 个**等式**的解，其中这 N 个等式是从原来所有等式和不等式的约束集合（10.8.2）式～(10.8.5）式中抽取出来的。

使原始约束中的某 N 个约束能以等号满足的可行向量称为**基本可行向量**。若 $N>M$，则基本可行向量中至少应有 $N-M$ 个分量为零，因为至少还需要这么多个约束条件才能补足(10.8.2)式中的变元总数 N，也就是说，在基本可行解中最多有 M 个分量是非零的。不妨用(10.8.6)式～(10.8.7)式给出的例子进行验证，它的解能以等号满足的约束条件，包括(10.8.7)式中的后三个约束条件和基本约束条件 $x_0\geqslant 0$，总数达到了所要求的 4 个。

把前面两段的内容综合起来，可以得到**线性优化的基本定理**：若最优化可行向量存在，则必有一基本可行向量是最优的。

上述定理的重要意义在于，它把优化问题化简为一个"组合"问题，即在(10.8.2)式～(10.8.5)式所示的所有 $M+N$ 个约束条件中，决定最优化可行解到底应满足其中的哪 N 个约束条件的问题，而我们所要做的就是不断对各种不同的组合进行试探，并计算每种情况下的目标函数值，直到找到最优的为止。

但是盲目地使用组合方法会使问题复杂化，并增加大量的计算量。Dantzig 于 1948 年(参见本节参考文献[2])首先提出了针对这一问题的单纯形方法，这种方法的特点是：(i)只对某一系列组合进行试验，在这些组合中目标函数每一步都增加；(ii)一般经过几乎总是不大于 M 或 N 次数的迭代(不论 M 和 N 阶哪一个大)，就求出最优可行向量。对于第二个性质，虽然自从单纯形法建立之后人们就早已从经验中得到了结论，但是其正确性的证明直到 1982 年才由 Stephen Smale 得出(参见本节参考文献[3])。

10.8.2　关于约束标准形式的单纯形法

若在某个线性规划问题中没有(10.8.3)式或(10.8.4)式形式的约束，而只有形如(10.8.5)式的等式约束和形如(10.8.2)式的非负约束，我们称该线性规划问题具有**标准形式**。

为了研究上的需要，不妨考察一类约束条件更严格的问题，它们具有如下的附加性质：每一个形如(10.8.5)式的等式约束中，必须有至少一个变量的系数为正，且这个变量只在该约束条件中出现。在每一约束方程中选择一个这样的变量，并以它作为变元求解该约束方程。这样选出来的变量称为**左端变量**或**基本变量**，其总数为 $M(=m_3)$ 个。而剩下的 $N-M$ 变量则称为**右端变量**或**非基本变量**。虽然只有在 $M\leqslant N$ 时才能得到这种**约束标准形式**，但在下文中我们仍将以这类线性规划问题为出发点来考虑。

可能会认为这类约束标准形式有些过于特殊化，它不可能包括我们想要解决的所有线性规划问题，但事实并非如此。下面将要介绍的就是如何将任意一个线性规划问题转化为约束标准形式，以及如何将单纯形法应用于这种约束标准形式。

让我们来看一个约束标准形式问题的例子：

$$\text{极大化}\qquad z=2x_1-4x_2 \tag{10.8.8}$$

其中 x_0,x_1,x_2,x_3 非负且满足

$$\begin{aligned}x_0&=2-6x_1+x_2\\x_3&=8+3x_1-4x_2\end{aligned} \tag{10.8.9}$$

此例中 $N=4$，$M=2$。左端变量为 x_0 和 x_3；右端变量为 x_1 和 x_2。这里我们将目标函数(10.8.8)式写成了仅包含右端变量的形式。但应注意的是，这实际上并不是对具有约束标准形式的目标函数的一种约束要求，因为出现在目标函数中的任何一个左端变量，都可以用(10.8.9)式或类似的式子以代数方式被右端变量替换掉。

对于任何一个具有约束标准形式的线性规划问题，都能立刻求出它的一个基本可行解(虽然这个基本解不必非是**最优**的)，方法很简单，只要将所有右端变量都置为零，满足约束的所有左端变量的值从(10.8.9)式中即可求得。单纯形法的基本思想就是进行一系列的基本可行解的变换。在每次变换过程中，将右端变量与左端变量互调位置，而每一步中都保持当前的线性规划问题具有约束标准形，并且与原来的问题完全等价。

为了在符号表示上的方便,将(10.8.8)式和(10.8.9)式所包含的信息记录在如下的表格中:

		x_1	x_2
z	0	2	−4
x_0	2	−6	1
x_3	8	3	−4

(10.8.10)

应仔细研究(10.8.10)式,要非常清楚表中各项对应的是哪个值,还要知道对约束标准形式问题,如何在其表格形式与其表达式形式之间来回地进行转换。

单纯形法的第一步是查看表的头一行(我们称之为"z 行"),以及标有右端变量的各栏(称为"右列")中各项的值。依次让每一右端变量从某当前的零值开始增加,同时保持其余右端变量仍为零不变,然后考察这么做的结果:目标函数值是增加了还是减小了?答案由 z 行中各项元的符号给出。既然目的是要增加函数值,因此,我们只对那些 z 行项中为正值的右列感兴趣。在(10.8.10)式中只有一列满足上述条件,它的 z 行项为 2。

下一步要做的是,考察位于由第一步选出的每个 z 行项下面的那些列表值。我们的问题是,在某一个左端变量变为负值(当然这是不允许的)之前,右端变量可以增加到多少呢?如果右端变量所在的列与左端变量所在行之交叉处的表中元素为正数,那么该元素将不受任何限制:相应的左端变量只会越变越大。如果任何右端变量所在列的所有表项都是正值,则目标函数无界,说明问题已经得到了答案。

若在某个值为正的 z 行项下面,有一个或多个表中元素为负数,那么我们就需清楚,到底是哪一项首先限制了那一列的右端变量值的增加。显然,这一受限的增加量可以用右端变量所在列的元素(称为主元)来除主元素所在行的"常数列"(最左边的列)中元素而得到。若得到的数值绝对值越小说明受到限制越多。于是,由于选择了这一主元而使目标函数增加的量,应由这个数值乘以那一行对应的 z 行表项值得到。对所有可能的右列重复这一过程,并求出具有最大增加量的主元,这样就完成了"选主元"的工作。

在上面的例子中,惟一的一个值为正的 z 行表项是 2,它下面只有一个负值项,即 −6,因此,−6 即为主元,其常数列元素为 2。由于该主元允许 x_1 增加的量为 $2 \div |6|$,继而目标函数可因此增加 $(2 \times 2) \div |6|$ 的绝对数量。

单纯形法的第三步是,给选定的右端变量一个增值,使之成为一个左端变量;同时修改各左端变量,让主元所在行的元素减为零而使之成为一个右端变量。对上面的例子,首先解主元所在行的方程,以旧的左端变量 x_0 表示新左端变量 x_1,即

$$x_0 = 2 - 6x_1 + x_2 \qquad \rightarrow \qquad x_1 = \tfrac{1}{3} - \tfrac{1}{6}x_0 + \tfrac{1}{6}x_2 \tag{10.8.11}$$

然后将上式代入原来的 z 行中,得到

$$z = 2x_1 - 4x_2 = 2\left[\tfrac{1}{3} - \tfrac{1}{6}x_0 + \tfrac{1}{6}x_2\right] - 4x_2 = \tfrac{2}{3} - \tfrac{1}{3}x_0 - \tfrac{11}{3}x_2 \tag{10.8.12}$$

再将其代入其他所有左端变量所在的行中,这里只有 x_3,即

$$x_3 = 8 + 3\left[\tfrac{1}{3} - \tfrac{1}{6}x_0 + \tfrac{1}{6}x_2\right] - 4x_2 = 9 - \tfrac{1}{2}x_0 - \tfrac{7}{2}x_2 \tag{10.8.13}$$

至此,由(10.8.11)式~(10.8.13)式可以形成如下所示的新表格:

		x_0	x_2
z	$\frac{2}{3}$	$-\frac{1}{3}$	$-\frac{11}{3}$
x_1	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$
x_3	9	$-\frac{1}{2}$	$-\frac{7}{2}$

(10.8.14)

第四步是转回并重复第一步,寻找目标函数的下一可能增值。不断重复上述过程,直到 z 行的所有右端变量项都变成负值为止。这表明目标函数不可能再增加了。在上面给出的例子中,(10.8.14)式已经满足了

这个条件，因此求解过程至此可以结束。

整个问题的解可以从最后一张表的常数列中读出。在(10.8.14)表中可以看到，目标函数的最大值为 2/3，解向量为 $x_1 = 1/3, x_3 = 9, x_0 = x_2 = 0$。

现在让我们回看一下从(10.8.10)表导出(10.8.14)表的过程。可以看出，整个过程可以完全用表格的形式总结成一系列规定的基本矩阵运算：

- 选出主元并存储。
- 存储整个主元所在的列。
- 将表中各行(主元所在的行除外)用其自身与主元行的线性组合来替代，使其主元所在列的那一项变为零。
- 用主元的负值去除主元所在的行
- 以存储单元内的主元值的倒数替代主元素。
- 用存储单元内的主元值去除主元所在列各剩余项的存储值，并用所得结果替代主元所在列中的各剩余项。

上述一系列的操作实际上可由一个线性规划程序来完成，这个程序将在下文中给出。

至此，我们已能解决任何一个以约束标准形式给出的线性规划问题了。只有一种特殊情况在求解时可能会遇到一些麻烦，即在某一步中，常数列中的某个项元变成了零，以至于某个左端变量也为零，进而所有右端变量都成了零。这种情况下得到的解我们称之为**退化可行解**。为了使求解过程继续下去，可能需要将这个退化的左端变量与某一个右端变量进行交换，有时候这样交换可能要进行好几次。

10.8.3　将一般问题转化为约束标准形式

关于从一般线性规划问题到约束标准形式的转化，有几种很巧妙的办法可以使转化过程变得轻而易举。

首先，需要把(10.8.3)式或(10.8.4)式形式的不等式约束条件例如(10.8.7)式中的前三个约束条件去除。具体做法是，引入一些所谓的**松弛变量**，利用它们的非负性，将不等式转化为等式。将松弛变量记为 y_i，它们的总数有 $m_1 + m_2$ 个。应注意的是，一旦将这些松弛变量引入，就要把它们与原来变量 x_i 同样对待。但在求解过程结束时，将它们忽略即可。

例如，在引入松弛变量之后，目标函数(10.8.6)式并未发生变化，而(10.8.7)式却变成如下形式：

$$\begin{aligned} x_0 + 2x_2 + y_0 &= 740 \\ 2x_1 - 7x_3 + y_1 &= 0 \\ x_1 - x_2 + 2x_3 - y_2 &= \tfrac{1}{2} \\ x_0 + x_1 + x_2 + x_3 &= 9 \end{aligned} \tag{10.8.15}$$

请注意松弛变量前的系数的符号由相应的原不等式的方向决定。

其次，需确保存在一个由 M 个左端变量组成的集合，这样就可以用约束标准形式建立一个初始表格(换言之，需要找一个“可行的基本初始向量”)。我们采用的方法还是引入一些新的变量，这些变量称为**人工变量**，以 z_i 标记，其总数为 M 个。例如，在(10.8.15)式所示的每一约束等式中都引入一个人工变量，使其变为如下形式：

$$\begin{aligned} z_0 &= 740 - x_0 - 2x_2 - y_0 \\ z_1 &= -2x_1 + 7x_3 - y_1 \\ z_2 &= \tfrac{1}{2} - x_1 + x_2 - 2x_3 + y_2 \\ z_3 &= 9 - x_0 - x_1 - x_2 - x_3 \end{aligned} \tag{10.8.16}$$

至此，原问题已经化为约束标准形。

读者可能会提出这样一个问题:**除非所有 z_i 都是零**,否则(10.8.16)式与(10.8.15)式根本不等价。事实的确如此,但这里颇有一些玄妙之处。在求解问题时必须分两个阶段进行,第一阶段:用一个所谓的**辅助目标函数**替代原来的目标函数(10.8.16)式,即

$$z' \equiv -z_0 - z_1 - z_2 - z_3 = -(749\tfrac{1}{2} - 2x_0 - 4x_1 - 2x_2 + 4x_3 - y_0 - y_1 + y_2) \qquad (10.8.17)$$

其中,后一个等式是将(10.8.16)式代入得到的。现在,再对(10.8.17)式所示的辅助目标函数使用单纯形法显然,如果所有 z_i 都为零,则辅助目标函数对于非负的 z_i 将取得最大值。因此,我们希望单纯形法能够在第一阶段构造出一个左端变量的集合,这些左端变量仅仅从 x_i 和 y_i 中选取,而所有 z_i 都取为右端变量。这样,我们就可以划掉所有的 z_i,剩下的就是只含 x_i 和 y_i 的约束标准形式的问题了。因此,换言之,第一阶段的任务就是构造一个初始基本可行向量。在第二阶段,我们的目标是求解由第一阶段导出的问题,但在这时要利用原来的目标函数,而不能再用辅助目标函数了。

如果在第一阶段不能对所有 z_i 都取零值,又该如何处理呢?实际上,这种情况恰恰说明,该线性规划问题根本没有初始基本可行向量存在。也就是说,约束条件本身是自相矛盾的,只要指出这一点,就算解答了此问题。

下面将介绍,如何将第一阶段和第二阶段所需的信息转化成表格形式。仍以目标函数及约束条件由(10.8.6)式~(10.8.7)式给出的问题为例。

		x_0	x_1	x_2	x_3	y_0	y_1	y_2
z	0	1	1	3	$-\frac{1}{2}$	0	0	0
z_0	740	−1	0	−2	0	−1	0	0
z_1	0	0	−2	0	7	0	−1	0
z_2	$\frac{1}{2}$	0	−1	1	−2	0	0	1
z_3	9	−1	−1	−1	−1	0	0	0
z'	$-749\frac{1}{2}$	2	4	2	−4	1	1	−1

(10.8.18)

在双线框内的表项只不过是将原问题的系数组织成表格形式而已。实际上,这些项加上 N, M, m_1, m_2, m_3 的值就是下面的单纯形法程序所需的全部输入参数。松弛变量 y_i 下面的各列简单地记录了 M 条约束条件中的每一个是否具有 $\leqslant$, $\geqslant$ 或 $=$ 的形式。事实上只要我们在造表时是按正确的顺序填写各行内容的,这些信息及 m_1, m_2, m_3 的值都不过是冗余信息而已。由于辅助目标的系数(最末一行)恰好是其对应列各行之和的负数,因此它们的计算非常容易。

一个单纯形法运行程序的输出需包括以下几项:(i)一个标志项,指出运行结果的类型,是有解、无解、还是有无界解;(ii)一张更新后的表格。由(10.8.18)表得到输出表格为(两位有效数字)

		x_0	y_1	y_2	···
z	17.03	−.95	−.05	−1.05	···
x_1	3.33	−.35	−.15	.35	···
x_2	4.73	−.55	.05	−.45	···
x_3	.95	−.10	.10	.10	···
y_0	730.55	.10	−.10	.90	···

(10.8.19)

只需稍稍数一下 x_i 和 y_i 的个数就会发现,在输入表和输出表中都有 $M+1$ 行(包括 z 行),但在输出表中只有 $N+1-m_1$ 列(包括常数列)包含有用信息,其他列都属于可以忽略的人工变量。在输出表中,数字列的第一列包括解向量以及目标函数最大值的信息。如果某个松弛变量(y_i)出现在表的左边,则相应的数值就表示其对应的不等式在得到充分满足时所需的量。在输出表中,非左端的变量具有零值。而具有零值的松弛变量则对应于以等式形式被满足的约束条件。

10.8.4　单纯形法的程序实现

下面程序是以 Kuenzi,Tzschach 和 Zehnder[4]的程序为基础而实现的。例程中除要输入 M、N、m_1、m_2、m_3 的值外，主要的输入项是一个二维矩阵 a，a 的存储内容是在(10.8.18)表中以双线框出的部分。这些输入值只占用 a 的 $M+1$ 个行和 $N+1$ 个列，但在程序内部却要用到矩阵 a 的附加行(用于存储辅助目标函数，这和(10.8.18)表的表示完全一样)，因此程序需把 a 说明为 Mat_IO_DP &a，且指针必须指向允许范围为

$$\mathrm{a}[i][k],\quad i=0\ldots\mathrm{m+1},\ k=0\ldots\mathrm{n} \tag{10.8.20}$$

的分配空间。此外，还有一点不能忽略的是 a 的各行必须完全按照(10.8.1)式、(10.8.3)式、(10.8.4)式、(10.8.5)式的顺序输入，即先输入目标函数，然后是“≤”约束条件和“≥”约束条件，最后输入“=”约束条件。

在输出项中，矩阵 a 的下标由两个返回的整数阵列确定。iposv[j](j = 0…$M-1$)包含序数 i，i 对应的原始变量 x_i 的当前状态由 a 的第 j + 1 行表示，它们即为解向量中的右端变量(a 的第一行当然存储的是 z 行)若 i 大于 N 则表示这个 i 对应的变量是松弛变量 y_i 而不是 x_i，也即 $x_{N+j}\equiv y_j$。同样，对 j = 0…$N-1$，izrov[j]给出了序数 i，它所对应的原始变量 x_i 在当前状态是一个右端变量，存储在 a 的第 j + 1 列中。$i>N-1$ 的意思与上面相同，所不同的是当 $i\geqslant N+m_1+m_2$ 时，它对应的变量表示某个人工变量或松弛变量，这个变量原先仅仅作为内部使用，而现在应完全忽略。

当程序运行结果为找到了一个有限解时，标志项 icase 被置为 0；当目标函数无界时，它被置为 + 1；如果没有解满足给定的约束条件，则 icase 被置为 - 1。

下面的程序对退化可行解的情况也进行了处理，因此读者不必担心这种情况的出现。此外，该程序不需存储(10.8.18)表中双线右边的各列内容，这可能会令使用者感到满意。不仅如此，我们还在程序中对松弛变量的跟踪记录采取了一种更加有效的方式。

还有一点值得注意的是，程序中收敛性的检验并不如想像的那么复杂。尽管程序中对带零不等式的检验是通过对某个很小的参数 EPS 的检验来实现的，但它并没有对这个参数进行调整，以适应不同量级输入数据的变化。尽管如此，程序已经足以适合问题的需要了，因为许多问题的输入数据与单位“1”在大小的数量级上并没有相差很多。但是，如果遇到死循环的情况，则必须修改程序 simplx 和 simp2 中的 EPS。此外，用置换变量的办法也可以达到解决问题的目的。最后需要说明的是，与下述介绍有关的内容，读者可以参见本节参考文献[5]。

```
#include "nr.h"

void NR::simplx(Mat_IO_DP &a, const int m1, const int m2, const int m3,
    int &icase, Vec_O_INT &izrov, Vec_O_INT &iposv)
线性规划中的单纯形法。输入参数为 a、m1、m2 和 m3，输出参数 a、icase、izrov 和 iposv，它们的定义见正文。
{
    const DP EPS=1.0e-14;                          其中 EPS 是绝对精度，它应该根据
    int i,k,ip,is,kh,kp,nl1;                       用户变量的尺度进行调整。
    DP q1,bmax;

    int m=a.nrows()-2;
    int n=a.ncols()-1;
    if (m != (m1+m2+m3)) nrerror("Bad input constraint counts in simplx");
    Vec_INT l1(n+1),l3(m);
    nl1=n;
    for (k=0;k<n;k++) {                            初始化表示可交换列的索引表，将所
        l1[k]=k+1;                                 有变量初始化为右端变量。
        izrov[k]=k;
    }
    for (i=1;i<=m;i++) {
        if (a[i][0] < 0.0) nrerror("Bad input tableau in simplx");
        常数 b_i 项为非负数。
        iposv[i-1]=n+i-1;
        初始化左端变量。m1 型约束条件通过将它们的松弛变量初始化为左端变量来表现，其
        中没有人工变量。m2 型约束条件通过将它们的松弛变量初始化为左端变量并带有负号
        来体现，它们的人工变量在做第一次交换时被隐含地处理了。m3 型约束条件中的松弛变
        量初始化为左端变量。
```

```
        }
        if (m2+m3 != 0) {                          初始点不是一个可行的解;
            for (i=0;i<m2;i++) l3[i]=1;            必须执行段落 1。
            初始化那些包含了从未被交换过的松弛变量的 m2 约束条件的列表。
            for (k=0;k<(n+1);k++) {                计算机辅助目标函数。
                q1=0.0;
                for (i=m1+1;i<m+1;i++) q1 += a[i][k];
                a[m+1][k] = -q1;
            }
            for (;;) {
                simp1(a,m+1,l1,nl1,0,kp,bmax);     求辅助目标函数的最大系数。
                if (bmax <= EPS && a[m+1][0] < -EPS) {
                    icase = -1;
                    辅助目标函数仍为负，而且不能进一步改进，因此无可行解存在。
                    return;
                } else if (bmax <= EPS && a[m+1][0] <= EPS) {
                辅助目标函数为 0,并不能再改进。这表明我们得到了一个可行的起始解通过"goto one"将所有人工变
                量清除,然后进入第二阶段继续求解。
                    for (ip=m1+m2+1;ip<m+1;ip++) {
                        if (iposv[ip-1] == (ip+n-1)) {   为等式约束条件求人工变量。
                            simp1(a,ip,l1,nl1,1,kp,bmax);
                            if (bmax > EPS)        与最大行主元所在列交换。
                                goto one;
                        }
                    }
                    for (i=m1+1;i<=m1+m2;i++)      改变所有对应一初始值没有被改变
                        if (l3[i-m1-1] == 1)       过的 m2 约束条件的行的符号。
                            for (k=0;k<n+1;k++)
                                a[i][k]= -a[i][k];
                    break;                         转到段落 2。
                }
                simp2(a,m,n,ip,kp);                选主元(段落 1)。
                if (ip == 0) {                     由于辅助目标函数的最大值无界，因
                    icase = -1;                    此不存在可行解。
                    return;
                }
        one:    simp3(a,m+1,n,ip,kp);
                将一个左端变量和一个右端变量互换(段落 1)，更新列表。
                if (iposv[ip-1] >= (n+m1+m2)) {    将等式约束条件中的人工变量换出。
                    for (k=0;k<nl1;k++)            确保将它从 l1 列表中清除。
                        if (l1[k] == kp) break;
                    --nl1;
                for (is=k;is<nl1;is++) l1[is]=l1[is+1];
            } else {
                kh=iposv[ip-1]-m1-n+1;
                if (kh >= 1 && l3[kh-1]) {         首次将 m2 约束条件挽出。在带负号
                    l3[kh-1]=0;                    的项与隐式人工变量中修正主元。
                    ++a[m+1][kp];
                    for (i=0;i<m+2;i++)
                        a[i][kp]= -a[i][kp];
                }
            }
            SWAP(izrov[kp-1],iposv[ip-1]);
            更新左、右端变量表。
        }                                          仍在段落 1 中，返回到for(;;)。
    }
    第一阶段结束,即求一个初始可行解的工作告一段落;下面将进入第二阶段,即将该可行解进行优化。
    for (;;) {
        simp1(a,0,l1,nl1,0,kp,bmax);               测试 z-行。
        if (bmax <= EPS) {                         最优解找到，程序带值返回。
            icase=0;
            return;
        }
        simp2(a,m,n,ip,kp);                        选主元(段落 2)。
        if (ip == 0) {                             目标函数无界。报错，返回。
            icase=1;
            return;
        }
        simp3(a,m,n,ip,kp);                        将一个左端变量与一右端变量进行交
        SWAP(izrov[kp-1],iposv[ip-1]);             换(段落 2)，返回做下一轮迭代。
    }
}
```

在前面的程序中，利用了下面几个实用函数

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::simp1(Mat_I_DP &a, const int mm, Vec_I_INT &ll, const int nll,
    const int iabf, int &kp, DP &bmax)
确定一些元素的最大值，这些元素的索引存储在已知的列表 ll 中，标志 iabf 指示所求出的最大值是否为所有元素的绝对
值的最大值。
{
    int k;
    DP test;

    if (nll <= 0)                    没有符合条件的列。
        bmax=0.0;
    else {
        kp=ll[0];
        bmax=a[mm][kp];
        for (k=1;k<nll;k++) {
            if (iabf == 0)
                test=a[mm][ll[k]]-bmax;
            else
                test=fabs(a[mm][ll[k]])-fabs(bmax);
            if (test > 0.0) {
                bmax=a[mm][ll[k]];
                kp=ll[k];
            }
        }
    }
}
#include "nr.h"

void NR::simp2(Mat_I_DP &a, const int m, const int n, int &ip, const int kp)
选主元，包括退化情况在内。
{
    const DP EPS=1.0e-14;
    int k,i;
    DP qp,q0,q,q1;

    ip=0;
    for (i=0;i<m;i++)
        if (a[i+1][kp] < -EPS) break;          任何可能的主元？
    if (i+1>m) return;
    q1 = -a[i+1][0]/a[i+1][kp];
    ip=i+1;
    for (i=ip;i<m;i++) {
        if (a[i+1][kp] < -EPS) {
            q = -a[i+i][0]/a[i+1][kp];
            if (q < q1) {
                ip=i+1;
                q1=q;
            } else if (q == q1) {              退化情况。
                for (k=0;k<n;k++) {
                    qp = -a[ip][k+1]/a[ip][kp];
                    q0 = -a[i][k+1]/a[i][kp];
                    if (q0 != qp) break;
                }
                if (q0 < qp) ip=i+1;
            }
        }
    }
}



#include "nr.h"

void NR::simp3(Mat_IO_DP &a, const int i1, const int k1, const int ip,
    const int kp)
用于交换左端变量和右端变量的一些矩阵操作(见正文中说明)。
```

```
{
    int ii,kk;
    DP piv;

    piv=1.0/a[ip][kp];
    for (ii=0;ii<i1+1;ii++)
        if (ii != ip) {
            a[ii][kp] *= piv;
            for (kk=0;kk<k1+1;kk++)
                if (kk != kp)
                    a[ii][kk] -= a[ip][kk]*a[ii][kp];
        }
    for (kk=0;kk<k1+1;kk++)
        if (kk != kp) a[ip][kk] *= -piv;
    a[ip][kp]=piv;
}
```

10.8.5 其他线性规划方法简述

每个包含 N 个变量和 M 个约束条件、且具有标准形式的线性规划问题,都有一个与其对应的对偶问题,该对偶问题含有 M 个变量和 N 个约束条件,它的表格表示本质上就是其原问题(有时称为原始规划)的表格表示的转置。从对偶问题的解导出原始规划的解是完全可以做到的,而且这种求解方法在计算上有时也很有用,但它在一般情况下并不常用。

修改的单纯形法中有一个步骤与单纯形法完全等价,即选择哪几个左端变量和右端变量用做交换。尽管前者的计算量与后者相比并没有很大的改进,但它在对存储的组织方面的确与后者有不少不同之处。在计算的中间步骤,修改的单纯形法仅需一个大小为 $M \times M$ 矩阵,而不是 $M \times N$ 大小的矩阵,如果规划问题的限制条件很多,特别是对存储量的限制也包括在其中,那么就可以考虑用这种修改的单纯形法。

原始-对偶算法以及**复合单纯法**是两种不同的方法,它们都是为了避免一般单纯形法的两个阶段而设计的;两者在同时寻找可行解和最优解方面有一定的改进。但目前来看似乎还没有什么明显的迹象表明这两种方法能很大地优于通常情况下所使用的单纯形法。

如果目标函数和/或一个或多个约束条件,具有自变量的非线性表达式的形式,则称该问题为非线性规划问题。有关这类问题的文献很多,它们超出了本书的讨论范围。非线性规划问题有一种特殊情况,即表达式都具有二次型式,因此称为二次规划。此外变量只能取整数的最优化问题称为整数规划问题,它是离散优化问题的一个特例。在下一节中,将讨论一类特殊的离散优化问题。

参考文献和深入阅读

Bland, R.G. 1981, *Scientific American*, vol. 244 (June), pp. 126–144. [1]

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press). [2]

Kolata, G. 1982, *Science*, vol. 217, p. 39. [3]

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), Chapters 7–8.

Cooper, L., and Steinberg, D. 1970, *Introduction to Methods of Optimization* (Philadelphia: Saunders).

Gass, S.I. 1985, *Linear Programming*, 5th ed. (New York: McGraw-Hill).

Murty, K.G. 1976, *Linear and Combinatorial Programming* (New York: Wiley).

Land, A.H., and Powell, S. 1973, *Fortran Codes for Mathematical Programming* (London: Wiley-Interscience).

Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press). [4]

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §4.10.

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [5]

10.9　模拟退火法

模拟退火法(参见本节参考文献[1,2])作为一种适合于求解大规模优化问题的技术,近来已引起极大的关注。特别是当优化问题有很多局部极值而全局极值又很难求出时,模拟退火法尤其有效。在实用上,它有效地"解决了"著名的**旅行推销员问题**,即在必须依次访问每一个城市(共有 N 个城市)的前提下,为旅行推销员设计一条能够返回起点的最短旅程(其他一些实用的方法也可以解决这个问题)。模拟退火法还被成功地用于设计复杂的集成电路,也就是说如何最佳地安排几十万个电路元件,使它们全部集成在一个很小的硅片上,而相互连接的线路之间的缠绕能够达到最小(参见本节参考文献[3,4]),尽管退火法很有效,但它的算法实现却相对地简单,这一点似乎有些不可思议。

请注意,上面提到的两个例子都属于**组合极小化问题**。本类问题通常也有一个目标函数,但是函数的定义域并不是简单地由 N 个连续参变量组成的 N 维空间,而是一个离散的巨大构形空间,例如,由所有可能的城市旅行路线组成的集合,或者硅片电路元件的所有可能的分配方式的集合。构形空间中元素的数量相当巨大,根本不可能完全列举,而且因为集合是离散的,也不可能"沿合适的方向连续下降"。因此在构形空间中,"方向"概念就没有什么意义了。

后面还将介绍如何在有连续控制参数的空间(就如 10.4 节至 10.7 节中的空间)中利用模拟退火法。这种应用实际上要比组合问题复杂一些,因为其中又要出现"狭长山谷"的情况。正如在下文中将看到的,模拟退火法的试探步骤是"随机"的,但在一个狭窄且漫长的等高线山谷中,几乎所有的随机步骤都呈向上趋势。因此,算法中需要增加一些技巧。

模拟退火的核心思想与热力学的原理颇为相似,而且尤其类似于液体流动和结晶以及金属冷却和退火的方式。在高温下,一种液体的大量分子彼此之间进行着相对自由移动。如果该流体慢慢地冷却下来,热能可动性便会消失。大量原子常常能够自行排列成行,形成一个纯净的晶体,该晶体在各个方向上都被完全有序地排列在几百万倍于单个原子大小的距离之内。对于这个系统来说,晶体状态是能量最低状态,而所有缓慢冷却的系统都可以自然达到这个最低能量状态,这可以说是一个令人惊奇的事实。实际上,如果某种液体金属被迅速冷却或被"猝熄",那么它不会达到这一状态,而只能达到一种具有较高能量的多晶状态或非结晶状态。

因此,这一过程的本质在于**缓缓地**制冷,以争取充足的时间,让大量原子在丧失可动性之前进行重新分布。这就是所谓**退火**在技术上的定义,同时也是确保达到低能量状态所必需的条件。

尽管我们的比喻并不算贴切,但是迄今为止所讨论的所有极小化算法,确实与快速冷却猝熄有某些相关联之处。以往我们处理问题的方式都是:从初始点开始,立即沿下降方向前进,走得越远越好,似乎这样才能迅速求得问题的解。但是,正如前面常常提到的,这种方法往往只能求得局部极小点,却求不到整体小点。自然界本身的极小化算法则基于一种截然不同的方式。所谓的 Boltzmann 概率分布,

$$\mathrm{Prob}(E) \sim \exp(-E/kT) \tag{10.9.1}$$

表达了这样一种思想,即一个处于热平衡状态且具有温度 T 的系统,其能量按照概率分布于所有不同的能量状态 E 之中。即使在很低的温度下,系统也有可能(虽然这种可能性很小)处于一个较高的能量状态。因此相应地,系统也能够获得摆脱局部能量极小点的机会,并找到一个更好的、更接近于整体的极小点。(10.9.1)式中的参数 k(称为 Boltzmann 常数)是一个自然常数,它的作用是将温度与能量联系起来。换句话说,在有些情况下系统的能量可上升也可下降,但是温度越低,显著上升的可能性就越小。

1953 年,Metropolis 及其合作者(参见本节参考文献[5])首次将这种原理渗透到数值计算中。他们对一个模拟热力学系统提供了一系列选择项,并假设:系统构形从能量 E_1 变化到能量 E_2 的概率为 $p = \exp[-(E_2 - E_1)/kT]$。很显然,如果 $E_2 < E_1$,p 将大于一;在这类情况下,对构形的能量变化任意指定一个概率值 $p = 1$,也就是说,该系统总是取这个选择项。这种格式总是采取下降过程,偶尔采取上升步骤。目前已被公认为 Metropolis 算法。

为了将 Metropolis 算法应用于热力学以外的系统,必须提供以下几项基本要素:

1. 对可能的系统构形的一种描述。
2. 一个有关构形内部随机变化的生成函数,这些变化将作为"选择项"提交给该系统。
3. 一个目标函数 E(类似于能量)求解 E 的极小值,即为算法所要完成的工作。
4. 一个控制参数 T(类似于温度)和一个**退火进程**,该进程用来说明系统是如何从高值向低值降低的,例如在 T 时每次下降步骤中要经过多少次随机的构形变化以及该步长是多大等。应说明的是,这里"高"和"低"的含义,还有进程表的确定,都需要一定的物理知识和/或一些摸索的实验。

10.9.1 组合极小化:旅行推销员问题

下面用"旅行推销员问题"为具体实例说明模拟退火法的应用。假设一个推销员,要去 N 个分别位于 (x_i, y_i) 的城市进行推销,并于最后返回他原来所在的城市,要求每个城市只能去一次,而且所经过的路程要尽可能地短。这个问题属于一类所谓"NP 完全问题"。这类问题求出一个**精确解**所需的计算时间是随 N 的增加以指数 exp(常数 × N)增长的。当 N 不断增大时,运行时间将迅速增加,进而导致费用高到令人难以接受的程序。旅行推销员问题也是众多极小化问题中的一种,它的目标函数 E 具有多个局部极小值。在实际应用当中,常常有足够多的条件可以从多个极小值中选出一个最小值,这个最小值即使不是绝对最小,也相当接近绝对最小了。退火法的目的就是要获得这个最小值,同时又要将计算量限制在 N 的低阶次的数量级上。

旅行推销员问题作为模拟退火问题时可以用如下的方式进行处理:

1. **构形** 将 N 个城市分别标记为 $i=0\ldots N-1$ 中的数,其中每个城市具有坐标 (x_i, y_i)。一个构形就是数字 $0\ldots N-1$ 的一个排列,可以解释为推销员途径的城市顺序。
2. **调整** Lin[6]曾经提出过一种所谓"转移有效集"。这里的"转移"包括两种类型:(a)移走路径的某一段,然后对这段路径上的城市用相反次序重新进行排列,并用后者来代替前者;(b)移走某段路径,并用位于城市间的随机选取的另一段路径来取代被移走的路径。
3. **目标函数** 在旅行推销员问题的一种最简单的形式中,目标函数 E 被定义为旅途的总长度,

$$E = L \equiv \sum_{i=0}^{N-1} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \tag{10.9.2}$$

这里认为点 N 与点 0 是重合的。但是为了表明模拟退火法的灵活性,我们还可以加上以下的额外限制:假设推销员不愿飞越密西西比河,在这种情况下,对每个城市给定一个参数 μ_i,如果该城市位于密西西比河以东,μ_i 取 1,若在密西西比河以西,μ_i 取 -1。对于目标函数 E,将其改写为

$$E = \sum_{i=0}^{N-1} \left[\sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} + \lambda(\mu_i - \mu_{i+1})^2\right] \tag{10.9.3}$$

由于每过一次河都将以 4λ 作为惩罚,因而现在我们设计的算法,就变成了寻找尽可能回避过河的最短路径。路径长度对过河次数的相对重要性将由我们选择的 λ 来确定。图 10.9.1 表明了所得的结果。显然,这种技巧可以推广到包含许多相互冲突目的要求的极小化问题当中。

4. **退火进程** 这一步需要借助试验来确定。首先要进行一些随机调整,然后利用它们来确定从转移到转移过程中将会遇到的 ΔE 值之范围。对参数 T 取一初始值(这个初始值要远远大于通常所能遇到的 ΔE 的最大值),并以倍增的步长下减,每次使 T 总共减少 10%。用每个新的常数 T 值去试各种 $100N$ 重构形,或 $10N$ 成功的重构形,无论哪个在前出现就取哪个。当实在不能再进一步减小 E 时,则停止。

下面的旅行推销员程序利用了 Metropolis 算法,并展示了模拟退火技术应用于组合问题的几个主要方面。

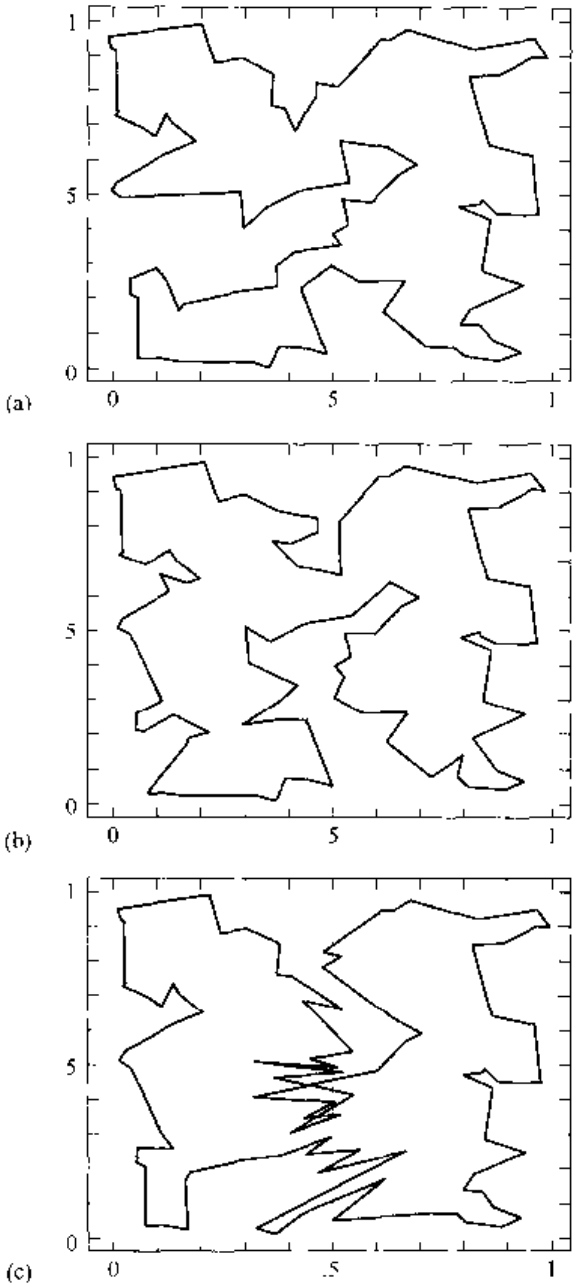


图 10.9.1　用模拟退火法解决旅行推销员问题. 图(a)表示的是从 100 个随机分布的城市中间找到的一条(接近)最短路径,点线标识的是一条河流,但这里对过河没有附加惩罚项,在图(b)中对过河施加的惩罚项很大,而图中所示的解本身的过河次数也相应地只有少得不能再少的两次,在图(c)中惩罚项为负,这就是说,推销员实际上成了恣意偷渡的走私者

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <cstdlib>
#include "nr.h"
using namespace std;

namespace {
    inline DP alen(const DP a, const DP b, const DP c, const DP d)
    {
        return sqrt((b-a)*(b-a)+(d-c)*(d-c));
    }
}

void NR::anneal(Vec_I_DP &x, Vec_I_DP &y, Vec_IO_INT &iorder)
```

本算法用于求解在 ncity 个城市之间进行往返旅行的最短路径，其中这 ncity 个城市的位置坐标存储在数组 x[0..ncity-1] 和 y[0..ncity-1] 中，数组 iorder[0..ncity-1] 表示途经城市的顺序。在输出项中，iorder 中的元素将被置为数字 0 到 ncity 的某排列，本程序将返回它所能求出的最佳选择路径。

```
{
    const DP TFACTR=0.9;          退火进程: 每步中t的下降值由该因子决定。
    bool ans;
    int i,i1,i2,idec,idum,j,k,nn,nover,nlimit,nsucc;
    static Vec_INT n(6);
    unsigned long iseed;
    DP path,de,t;

    int ncity=x.size();
    nover=100*ncity;              在任何温度下，所试路径的最大次数。
    nlimit=10*ncity;              在继续进行之前，路径成功改变的最大次数。
    path=0.0;
    t=0.5;
    for (i=0;i<ncity-1;i++) {     计算初始路径的长度。
        i1=iorder[i];
        i2=iorder[i+1];
        path += alen(x[i1],x[i2],y[i1],y[i2]);
    }
    i1=iorder[ncity-1];
    i2=iorder[0];
    path += alen(x[i1],x[i2],y[i1],y[i2]);
    idum = -1;
    iseed=111;
    cout << fixed << setprecision(6);
    for (j=0;j<100;j++) {         试验100个温度值。
        nsucc=0;
        for (k=0;k<nover;k++) {
            do {
                n[0]=int(ncity*ran3(idum));          选择段的起始点……
                n[1]=int((ncity-1)*ran3(idum));      ……和段的结尾。
                if (n[1] >= n[0]) ++n[1];
                nn=(n[0]-n[1]+ncity-1) % ncity;      nn为不位于当前段上的城市数。
            } while (nn<2);
            idec=irbit1(iseed);
            确定是否做段反转或段输送。
            if (idec == 0) {                         进行输送。
                n[2]=n[1]+int(abs(nn-1)*ran3(idum))+1;
                n[2] %= ncity;                       输送到一个不在当前路径上的某处。
                de=trncst(x,y,iorder,n);             计算代价。
                ans=metrop(de,t);                    进行预测。
                if (ans) {
                    ++nsucc;
                    path += de;
                    trnspt(iorder,n);                输送工作结束。
                }
            } else {                                 进行段反转。
                de=revcst(x,y,iorder,n);             计算代价。
                ans=metrop(de,t);                    进行预测。
                if (ans) {
                    ++nsucc;
                    path += de;
                    reverse(iorder,n);               完成段反转。
                }
            }
            if (nsucc >= nlimit) break;              如果成功的转移超过次数则提前结束。

        }
        cout << endl << "T = " << setw(12) << t;
        cout << "     Path Length = " << setw(12) << path << endl;
        cout << "Successful Moves: " << nsucc << endl;
        t *= TFACTR;                                 退火进程。
        if (nsucc == 0) return;                      如果不成功则返回。
    }
}


#include <cmath>
```

```
#include "nr.h"
using namespace std;

namespace {
    inline DP alen(const DP a, const DP b, const DP c, const DP d)
    {
        return sqrt((b-a)*(b-a)+(d-c)*(d-c));
    }
}

DP NR::revcst(Vec_I_DP &x, Vec_I_DP &y, Vec_I_INT &iorder, Vec_IO_INT &n)
该程序返回的是反转某给定路径所需的代价函数值。参数中 ncity 为城市数；数组 x[0..ncity-1],y[0..ncity-1]为这些城
市的位置坐标；iorder[0..ncity-1]为当前路线；数组 n 的头两个元素 n[0]和 n[1]分别代表将要被反转的路径段的起点城
市和终点城市。函数的返回值为反转所需的代价值，但真正的反转过程并不是由该程序执行。
{
    int j,ii;
    DP de;
    Vec_DP xx(4),yy(4);

    int ncity=x.size();
    n[2]=(n[0]+ncity-1) % ncity;                找出 n[0]以前的城市……
    n[3]=(n[1]+1) % ncity;                      ……找出 n[1]以后的城市。
    for (j=0;j<4;j++) {
        ii=iorder[n[j]];                        求四个所涉及到的城市的坐标。
        xx[j]=x[ii];
        yy[j]=y[ii];
    }
    de = -alen(xx[0],xx[2],yy[0],yy[2]);        计算断开路径段两端所需的代价。
    de -= alen(xx[1],xx[3],yy[1],yy[3]);        以及用相反顺序重新连接所需的代价。
    de += alen(xx[0],xx[3],yy[0],yy[3]);
    de += alen(xx[1],xx[2],yy[1],yy[2]);
    return de;
}


#include "nr.h"

void NR::reverse(Vec_IO_INT &iorder, Vec_I_INT &n)
该程序的作用是执行一段路径的反转过程。输入参数 iorder[0..ncity-1]给出当前的路线顺序；向量 n 的前 4 个元素中，n[0]
和 n[1]分别为将要被反转的路径的起点和终点城市，n[2]和 n[3]则分别为紧靠 n[0]之前和紧随 n[1]之后的两个城市的标
号，其中 n[2]和 n[3]由函数 revcst 给出。在输出端，iorder[0..ncity-1]将被返回，它存放 n[0]至 n[1]路段被反转后的元素值。
{
    int nn,j,k,l,itmp;

    int ncity=iorder.size();
    nn=(1+((n[1]-n[0]+ncity) % ncity))/2;       为实现反转操作，必须交换这么多城市。
    for (j=0;j<nn;j++) {
        k=(n[0]+j) % ncity;                     从段的端部开始，顺序交换城市对，
        l=(n[1]-j+ncity) % ncity;               直至到达路径的中心。
        itmp=iorder[k];
        iorder[k]=iorder[l];
        iorder[l]=itmp;
    }
}


#include <cmath>
#include "nr.h"
using namespace std;

namespace {
    inline DP alen(const DP a, const DP b, const DP c, const DP d)
    {
        return sqrt((b-a)*(b-a)+(d-c)*(d-c));
    }
}

DP NR::trncst(Vec_I_DP &x, Vec_I_DP &y, Vec_I_INT &iorder, Vec_IO_INT &n)
```

该程序返回的是输送某段指定路径所需的代价函数值。输入参数 ncity 为城市的个数;x[0..ncity-1]和 y[0..ncity-1]为这些城市的位置坐标;iorder[0..ncity-1]为当前路线;数组 n 的前 7 个元素分别为:将要被输送的路径段的起、止点城市以及这段路径将要被插入处的标志城市(插在该城市之后)。该函数的返回值为计算出来的输送代价值,但实际的输送操作并不由该程序执行。

```
{
    int j,ii;
    DP de;
    Vec_DP xx(6),yy(6);

    int ncity=x.size();
    n[3]=(n[2]+1) % ncity;                    找出位于 n[2]之后的城市……
    n[4]=(n[0]+ncity-1) % ncity;              ……位于 n[0]之前的一个城市……
    n[5]=(n[1]+1) % ncity;                    ……位于 n[1]之后的一个城市。
    for (j=0;j<6;j++) {
        ii=iorder[n[j]];                      求出六个城市的有关坐标。
        xx[j]=x[ii];
        yy[j]=y[ii];
    }
    de = -alen(xx[1],xx[5],yy[1],yy[5]);      计算下列操作所需代价,断开 n[1]到 n[2]
    de -= alen(xx[0],xx[4],yy[0],yy[4]);      间的路径,打开 n[3]和 n[4]之间的空间,
    de -= alen(xx[2],xx[3],yy[2],yy[3]);      连接这个空间中的路径段,以及连接 n[5]
    de += alen(xx[0],xx[2],yy[0],yy[2]);      到 n[6]。
    de += alen(xx[1],xx[3],yy[1],yy[3]);
    de += alen(xx[4],xx[5],yy[4],yy[5]);
    return de;
}
```

```
#include "nr.h"

void NR::trnspt(Vec_IO_INT &iorder, Vec_I_INT &n)
```

该程序的作用是执行真正的段输送操作,输入参数 iorder[0..ncity-1]给出当前的路径顺序;数组 n 共有 6 个元素,其意义分别为:n[0]和 n[1]分别代表将要被输送的路径段的起点城市和终点城市;n[2]和 n[3]为两个相邻的城市,n[0]至 n[1]路径段即将放入它们中间;n[4]和 n[5]分别为 n[0]之前和 n[1]之后的两个城市。在这 6 个元素中 n[3]、n[4]和 n[5]由函数 trnsf 给出。在输出端,iorder 将根据路径段的移动和变化作出相应的修改。

```
{
    int m1,m2,m3,nn,j,jj;

    int ncity=iorder.size();
    Vec_INT jorder(ncity);
    m1=(n[1]-n[0]+ncity) % ncity;             找出位于 n[0]和 n[1]间的城市个数……
    m2=(n[4]-n[3]+ncity) % ncity;             …… n[3]到 n[4]间的城市个数……
    m3=(n[2]-n[5]+ncity) % ncity;             …… n[5]到 n[2]间的城市个数。
    nn=0;
    for (j=0;j<=m1;j++) {
        jj=(j+n[0]) % ncity;                  复制所选路径段。
        jorder[nn++]=iorder[jj];
    }
    for (j=0;j<=m2;j++) {                     复制 n[3]到 n[4]间的路径段。
        jj=(j+n[3]) % ncity;
        jorder[nn++]=iorder[jj];
    }
    for (j=0;j<=m3;j++) {                     最后,复制 n[5]到 n[2]间的路径段。
        jj=(j+n[5]) % ncity;
        jorder[nn++]=iorder[jj];
    }
    for (j=0;j<ncity;j++)                     将 jorder 复制到 iorder。
        iorder[j]=jorder[j];
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

bool NR::metrop(const DP de, const DP t)
```

该程序为 Metropolis 算法的程序实现。metrop 返回的是一个布尔型变量,由该变量决定是否接受一个使目标函数 e 产生改变量 de 的重构形。如果 de<0 则 metrop=true;而当 de>0 时,metrop 为 true 的概率是 exp(-de/t),其中 t 是一个由退火进程决定的温度值。

```
{
    static int gljdum=1;

    return de < 0.0 || ran3(gljdum) < exp(-de/t);
}
```

10.9.2　模拟退火法在连续极小化问题中的应用

模拟退火法的基本思想也可以应用于具有连续 N 维控制参数的极小化问题当中，例如，在某个函数 $f(\mathbf{x})$（这里 $\mathbf{x}$ 为一个 N 维向量）有许多局部极小点的情况下，求解它的（理想的、全局的）极小值。这时 Metropolis 算法所需的四要素可以具体化为：函数 f 就是目标函数；系统状态就是 N 维空间中的点 $\mathbf{x}$；类似于温度的控制参数 T 以及一个使 T 逐渐降低的退火进程仍为原先的定义；描述构形内部随机变化的发生器即为一个从 $\mathbf{x}$ 到 $\mathbf{x}+\Delta\mathbf{x}$ 采取随机步骤的方法。

在上述四要素中问题最大的是最后一条。目前已发表的文献[7～10]，介绍了几种不同的选择 $\Delta\mathbf{x}$ 的方法。但我们认为，这些方法都不算成功。问题在于"效率"二字：**当局部的向下运动存在时**，如果某个随机变化发生器几乎总是做出向上运动的决策，那么它的效率就很低。我们认为，一个好的发生器在等高线的"窄谷"中仍应保持高效性；当算法在接近收敛到极小点处时，它的效率也不应越变越低。在上面提到的几篇文献中，除[7]中介绍的方法之外，其他所有方法都将表现不同程度的低效性。

下面将要介绍的这种方法，利用了下降单纯形法（参见 10.4 节）的一种修改后的形式。首先将 Metropolis 算法四要素中的系统状态描述，由单个点 $\mathbf{x}$ 改为一个具有 $N+1$ 个点的单纯形；单纯形的"操作"和 10.4 节中介绍的相同，分为反射、扩张和收缩三种。然后将一个正的、呈对数分布的随机变量（与温度 T 成比例）**添加**到存储的函数值中（该函数值与单纯形的每个顶点都有关联），再从每个被当做替代的新点的函数值中**减去**一个类似的随机变量。和普通的 Metropolis 方法一样，这种方法总是接受一个真正的下降步骤，但有时也接受一次上升步骤。在极限过程 $T\to 0$ 中，该算法恰好变成了下降的单纯形法，并收敛到一个局部极小点。

在 T 的某有限值处，单纯形将扩展到一定的规模，其大小接近于在这个温度值所能达到的区域，然后单纯形在这个区域内部做随机的滚动翻转式布朗运动，并在该过程中抽取一些新的、近似随机的点作为样本。一个区域被利用的效率与其狭窄度（对椭球状山谷，指它的主轴比率）及方向性均无关。如果温度降低得足够缓慢，那么单纯形将极有可能收缩到那个区域内，而那个区域内包含已遇到的最低相对极小。

由于在所有模拟退火法的应用场合中，"足够缓慢"一词的含意根据问题的不同可以有相当大的区别，因而成功与否往往取决于退火进程选择。下面几种可能性值得一试：

- 每经过 m 步移动之后，将 T 减到 $(1-\epsilon)T$，这里 ϵ/m 的具体值要通过实验确定。
- 设总的移动步数为 K，每经过 m 步移动之后将 T 减到 $T=T_0(1-k/K)^\alpha$，其中 k 为到目前为止所经过的步数的累加值。α 为一常数，可取为 1，2 或 4 等。α 的最佳值取决于各种深度的相对于极小的统计分布，稍大一些的 α 值在较低温度时，需花费的迭代次数将更多。
- 每经过 m 步骤移动之后，置 T 为 β 乘 f_1-f_b，其中 β 为一个阶数为 1 的常数，其具体值由试验确定；f_1 为目前单纯形中最小的函数值；f_b 为从未遇到的最佳函数。但应当注意的是，T 的降低幅度一次不要超过某个分数值 γ。

另一个策略方法上的问题是，当单纯形的某个顶点被放弃并让位于"永远的最佳点"时，是否需要**重新开始**（但必须保证在进行这项工作时，这个"永远的最佳点"当前并不在单纯形中）。对于有些问题重新开始（例如，只要温度降低了因子 3 即执行重新开始步骤）的效果极佳。而对于另外一些问题，重新开始不仅没有任何效果反而会产生负作用。上述两种截然相反的情况，我们都找到了例子可以作为例证。

将下面的程序 amebsa 同 10.4 节中与之相应的程序 amoeba 进行比较，会发现参数 iter 的使用方式在两个程序中略有不同。

```
#include <cmath>
#include "nr.h"
using namespace std;

namespace {
```

```
    inline void get_psum(Mat_I_DP &p, Vec_O_DP &psum)
    {
        int n,m;
        DP sum;

        int mpts=p.nrows();
        int ndim=p.ncols();
        for (n=0;n<ndim;n++) {
            for (sum=0.0,m=0;m<mpts;m++) sum += p[m][n];
            psum[n]=sum;
        }
    }
}

extern int idum;                                    应在main中定义并初始化。
DP tt;                                              与amotsa通信。

void NR::amebsa(Mat_IO_DP &p, Vec_IO_DP &y, Vec_O_DP &pb, DP &yb, const DP ftol,
    DP funk(Vec_I_DP &), int &iter, const DP temptr)
```

用模拟退火与Nelder和Mead的下降单纯形法相结合的方法求多元函数funk(x)的极小值，其中x[0..ndim-1]为一个ndim维向量。作为输入参数的矩阵p[0..ndim][0..ndim-1]共有ndim+1行，每行均为一个ndim维向量，分别代表初始单纯形的各个顶点。amebsa的输入项还包括矢量y[0..ndim]、浮点数ftol以及iter和temptr，其中y的各个元素将被初始化为函数funk在p的ndim+1个顶点(即行)的值；ftol为函数值所要达到的相对收敛容许限。一旦满足要求，程序将尽早返回；iter和temptr的意义分别是函数值的计算次数和温度。程序在某退火温度temptr处进行iter次函数值计算后返回。而接下来要做的事就是根据退火进程调低温度temptr，重置iter并再次调用该程序(每次调用时其他参数保持不变)。如果iter以正值返回，则说明程序正常收敛。如果第一次调用时yb被初始化为一个很大的数，则yb和pb[0..ndim-1]将依次返回已遇到的最佳函数值和最佳点(这个最佳点可以永远不是单纯形中的点)。

```
{
    int i,ihi,ilo,j,n;
    DP rtol,yhi,ylo,ynhi,ysave,yt,ytry;

    int mpts=p.nrows();
    int ndim=p.ncols();
    Vec_DP psum(ndim);
    tt = -temptr;
    get_psum(p,psum);
    for (;;) {
        ilo=0;                                      确定最高点(即差点)、次高点和最低点
        ihi=1;                                      (即最佳点)。
        ynhi=ylo=y[0]+tt*log(ran1(idum));           我们所"看到"的顶点总是处于随机的热
        yhi=y[1]+tt*log(ran1(idum));                起伏状态。
        if (ylo > yhi) {
            ihi=0;
            ilo=1;
            ynhi=yhi;
            yhi=ylo;
            ylo=ynhi;
        }
        for (i=3;i<=mpts;i++) {                     对单纯形中的点进行循环。
            yt=y[i-1]+tt*log(ran1(idum));           更多的热起伏运动。
            if (yt <= ylo) {
                ilo=i-1;
                ylo=yt;
            }
            if (yt > yhi) {
                ynhi=yhi;
                ihi=i-1;
                yhi=yt;
            } else if (yt > ynhi) {
                ynhi=yt;
            }
        }
        rtol=2.0*fabs(yhi-ylo)/(fabs(yhi)+fabs(ylo));
        计算从最高点到最低点的范围，若合乎要求，则返回。
        if (rtol < ftol || iter < 0) {              若返回，将最佳点和最佳值放入槽0中。
            SWAP(y[0],y[ilo]);
            for (n=0;n<ndim;n++)
                SWAP(p[0][n],p[ilo][n]);
            break;
        }
        iter -= 2;
```

```
        开始新的一轮迭代,首先从高点通过单纯形的表面,以因子 -1 做外插,即从高点对单纯形进行反射。
        ytry=amotsa(p,y,psum,pb,yb,funk,ihi,yhi,-1.0);
        if (ytry <= ylo) {
            得到了比最佳点还要好的结果, 因此用因子 2 再做一次外推。
            ytry=amotsa(p,y,psum,pb,yb,funk,ihi,yhi,2.0);
        } else if (ytry >= ynhi) {

            反射后的点不如次高点,因此需要找一个中间的较低点,即做一次一维收缩。
            ysave=yhi;
            ytry=amotsa(p,y,psum,pb,yb,funk,ihi,yhi,0.5);
            if (ytry >= ysave) {                    似乎无法摆脱高点, 最好围绕最低点也
                for (i=0;i<mpts;i++) {              即最佳点进行收缩。
                    if (i != ilo) {
                        for (j=0;j<ndim;j++) {
                            psum[j]=0.5*(p[i][j]+p[ilo][j]);
                            p[i][j]=psum[j];
                        }
                        y[i]=funk(psum);
                    }
                }
                iter -= ndim;
                get_psum(p,psum);                   重新计算 psum。
            }
        } else ++iter;                              纠正计数器。
    }
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

extern int idum;                                    在主程序中定义和初始赋值。
extern DP tt;                                       在 amebsa 中定义。

DP NR::amotsa(Mat_IO_DP &p, Vec_O_DP &y, Vec_IO_DP &psum, Vec_O_DP &pb, DP &yb,
    DP funk(Vec_I_DP &), const int ihi, DP &yhi, const DP fac)
从高点通过单纯形的表面,以因子 fac 做外推,如果得到的新点较好,则用它替代高点。
{
    int j;
    DP fac1,fac2,yflu,ytry;

    int ndim=p.ncols();
    Vec_DP ptry(ndim);
    fac1=(1.0-fac)/ndim;
    fac2=fac1-fac;
    for (j=0;j<ndim;j++)
        ptry[j]=psum[j]*fac1-p[ihi][j]*fac2;
    ytry=funk(ptry);
    if (ytry <= yb) {                       存储至今最好的。
        for (j=0;j<ndim;j++) pb[j]=ptry[j];
        yb=ytry;
    }
    yflu=ytry-tt*log(ran1(idum));           我们曾经对所有的当前顶点添加热起伏运动,
    if (yflu < yhi) {                       但这里我们要减少热起伏, 目的是给单纯形
        y[ihi]=ytry;                        一个热布朗运动, 就像接受任何提议的变化
        yhi=yflu;                           一样。
        for (j=0;j<ndim;j++) {
            psum[j] += ptry[j]-p[ihi][j];
            p[ihi][j]=ptry[j];
        }
    }
    return yflu;
}
```

模拟退火法在优化方法中将处于什么样的地位,扮演什么样的角色? 对于这个问题目前还没有足够的实践经验来明确回答。但是这种方法的确有几个极为吸引人的特点,与其他优化技术相比显得尤其突出。

首先,这种方法能迅速求出某些局部极小值,但那些实际上是不利的极小值。假如给定一些足够一般的

重构形,这种方法会在深度小于 T 的局部极小值之间自由地游动。当 T 值降低,这些具备频繁访问条件的极小值的数目也将逐渐减少。

其次,这种方法中有关构形的判定是趋向于以一种合乎逻辑的顺序进行的。当控制参数 T 很大时,那些引起最大能量差异的诸多变化将被筛去。T 减小时,这些判定就变得更恒定,相应地,算法将注意力更多地转向解的较精细的改进方面。例如,在旅行推销员问题中,如果密西西比河迂回于旅途之中,而且 λ 值很大,那么,在很高的 T 值下,只能做出跨河两次的判定,而位于跨河两岸的一些特定路径只能在稍后的步骤中才能确定。

以热力学原理为借鉴的方法远不止我们所讨论的这些内容,它们还可以推广到更广泛的应用领域,可以定义一些与特定热量和熵类似的量,这些量对于监测算法向某个可接受解逐步接近的过程是很有帮助的。关于这些内容的讨论读者可参见本节参考文献[1]中的介绍。

参考文献和深入阅读

Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. 1983, *Science*, vol. 220, pp. 671–680. [1]

Kirkpatrick, S. 1984, *Journal of Statistical Physics*, vol. 34, pp. 975–986. [2]

Vecchi, M.P. and Kirkpatrick, S. 1983, *IEEE Transactions on Computer Aided Design*, vol. CAD-2, pp. 215–222. [3]

Otten, R.H.J.M., and van Ginneken, L.P.P.P. 1989, *The Annealing Algorithm* (Boston: Kluwer) [contains many references to the literature]. [4]

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller A., and Teller, E. 1953, *Journal of Chemical Physics*, vol. 21, pp. 1087–1092. [5]

Lin, S. 1965, *Bell System Technical Journal*, vol. 44, pp. 2245–2269. [6]

Vanderbilt, D., and Louie, S.G. 1984, *Journal of Computational Physics*, vol. 56, pp. 259–271. [7]

Bohachevsky, I.O., Johnson, M.E., and Stein, M.L. 1986, *Technometrics*, vol. 28, pp. 209–217. [8]

Corana, A., Marchesi, M., Martini, C., and Ridella, S. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 262–280. [9]

Bélisle, C.J.P., Romeijn, H.E., and Smith, R.L. 1990, Technical Report 90-25, Department of Industrial and Operations Engineering, University of Michigan, submitted to *Mathematical Programming*. [10]

Christofides, N., Mingozzi, A., Toth, P., and Sandi, C. (eds.) 1979, *Combinatorial Optimization* (London and New York: Wiley-Interscience) [not simulated annealing, but other topics and algorithms].

第11章 特征系统

11.0 引言

一个 $N \times N$ 阶矩阵 $\mathbf{A}$ 如果满足下式,则它具有一个特征向量 $\mathbf{x}$ 和相应的特征值 λ:

$$\mathbf{A} \cdot \mathbf{x} = \lambda \mathbf{x} \tag{11.0.1}$$

显然,特征向量 $\mathbf{x}$ 的任意倍数也是特征向量,但我们不认为这样的倍数是不同的特征向量(零向量也不认为是特征向量)。显而易见,仅当在满足下式时,(11.0.1)式才成立:

$$\det|\mathbf{A} - \lambda \mathbf{1}| = 0 \tag{11.0.2}$$

如果展开上式,就可得到一个 λ 的 N 阶多项式,其根就是特征值。这也证明了总存在 N 个特征值(不一定是不同的)。由于重根的相同特征值被称为**退化**。用特征方程求根的方法来求特征值,不是好的计算方法。本章将学习一些更好的方法,以及寻找对应的特征向量的有效方法。

上面两式同样说明在 N 个特征值中,任意一个特征值都有一个(不一定是不同的)对应的特征向量:如果 λ 确定为某特征值,则矩阵 $\mathbf{A} - \lambda \mathbf{1}$ 是奇异的,而且我们知道,每个奇异矩阵在它的零空间里至少有一个非零向量(参见2.6节关于奇异值分解的讨论)。

如果将(11.0.1)式两边加上 $\tau \mathbf{x}$,则很容易发现,任意矩阵加上一个常数 τ 乘以单位矩阵后得到的矩阵,就将相应的特征值改变或平移了一个常数 τ。在这种平移下,特征向量不变。将会看到,这种平移是很多计算特征值算法中的重要内容。同样看到,零特征值没有什么特殊的意义。任何特征值都可通过平移成为零特征值,或者零特征值平移成为非零特征值。

11.0.1 定义和基本事实

一个矩阵如果等于它自身的转置,则称为是**对称的**:

$$\mathbf{A} = \mathbf{A}^T \qquad \text{或} \qquad a_{ij} = a_{ji} \tag{11.0.3}$$

如果一个矩阵等于自身转置的复共轭(它的埃尔米特共轭,用"†"表示),则称它为埃尔米特(Hermite)的或自共轭的:

$$\mathbf{A} = \mathbf{A}^\dagger \qquad \text{或} \qquad a_{ij} = a_{ji}^* \tag{11.0.4}$$

如果矩阵的转置等于它的逆,则称它为**正交的**:

$$\mathbf{A}^T \cdot \mathbf{A} = \mathbf{A} \cdot \mathbf{A}^T = \mathbf{1} \tag{11.0.5}$$

如果矩阵的埃尔米特共轭等于它的逆,则称它为酉矩阵。最后,如果一个矩阵与它的埃尔米特共轭可以交换,则称它为**正规的**:

$$\mathbf{A} \cdot \mathbf{A}^\dagger = \mathbf{A}^\dagger \cdot \mathbf{A} \tag{11.0.6}$$

对于实矩阵来说,埃尔米特性和对称性是同一的。酉正性和正交性也是同一的,而且这两类不同的矩阵都是正规的。

矩阵的"埃尔米特性"在矩阵特征值问题中的重要性,在于埃尔米特矩阵的特征值都是实的。相反地,实非对称矩阵的特征值可能包含实值,也可能包含成对的复共轭值。而对于非埃尔米特的复矩阵来说,其特征值一般都是复的。

"正规的"这一概念在特征值问题中是很重要的,因为具有非退化特征值(不同的)的正规矩阵的特征向量是完备正交的,它们可张成一个 N 维的向量空间。对于具有退化特征值的正规矩阵,可以将对应于退化特征值的特征向量进行线性组合来代替它的特征向量,并且这种替换给出了额外的自由度。利用这一自由

度,总可以从 Gram-Schmidt 正交化方法(参阅任意一本线性代数书)找到一组类似非退化情形的完备正交的特征向量。若矩阵各列是正交的特征向量集,则显然该矩阵是酉矩阵。一种特殊的情形是,由实对称矩阵的特征向量构成的矩阵是正交的,因为该矩阵的特征向量都是实的。

如果一个矩阵是非正规的,如任何随机非对称的实矩阵,则一般不能找到任何一组归一化正交的特征向量,甚至不能找到任何一对正交的特征向量(除非出于偶然)。尽管这 N 个非归一化正交的特征向量"常常"能张成一个 N 维向量空间,但不是总能这样做,也就是说,这些特征向量不总是完备的,这样的矩阵称为是亏损的。

11.0.2 左特征向量和右特征向量

非正规矩阵的特征向量不能保证它们之间是正交的,但它们却与另一组我们将要定义的向量有正交的关系。迄今为止,我们提到的特征向量都是满足(11.0.1)式与矩阵 $\mathbf{A}$ 右乘的列向量。更准确地讲,应称这些向量为**右特征向量**。同样可以发现,满足下式的左乘以矩阵 $\mathbf{A}$ 的行向量:

$$\mathbf{x}\cdot\mathbf{A}=\lambda\mathbf{x} \tag{11.0.7}$$

这样的向量称为**左特征向量**。将(11.0.7)式两边取转置,可得左特征向量就是 $\mathbf{A}$ 转置的右特征向量的转置。现在与(11.0.2)式比较,并且利用矩阵的行列式等于其转置的行列式这一事实,显然左右的特征值是恒等的。

如果矩阵 $\mathbf{A}$ 是对称的,则左、右特征向量恰好是互为转置,也就是说,分量的数值是相同的。相似地,如果矩阵是自共轭的,则左、右特征向量是互为埃尔米特共轭的。对于一般的非正规情形,有如下的计算:令 $\mathbf{X}_R$ 表示为由右特征向量按列排列形成的矩阵,$\mathbf{X}_L$ 表示为由左特征向量按行排列形成的矩阵,则(11.0.1)式和(11.0.7)式可重写成

$$\mathbf{A}\cdot\mathbf{X}_R=\mathbf{X}_R\cdot\mathrm{diag}(\lambda_0\ldots\lambda_{N-1})\qquad \mathbf{X}_L\cdot\mathbf{A}=\mathrm{diag}(\lambda_0\ldots\lambda_{N-1})\cdot\mathbf{X}_L \tag{11.0.8}$$

将前一式两边左乘以 $\mathbf{X}_L$,将后一式两边右乘 $\mathbf{X}_R$,两式相减得

$$(\mathbf{X}_L\cdot\mathbf{X}_R)\cdot\mathrm{diag}(\lambda_0\ldots\lambda_{N-1})=\mathrm{diag}(\lambda_0\ldots\lambda_{N-1})\cdot(\mathbf{X}_L\cdot\mathbf{X}_R) \tag{11.0.9}$$

这说明左右特征向量之点积构成的矩阵与特征值的对角矩阵可以交换。但是,只有对角矩阵本身和具有相异元素的对角矩阵才可以交换。因而,如果特征值是非退化的,则左特征向量除去所对应的右特征向量以外,应是与所有右特征向量相互正交的,反之亦然。通过归一化,对于任何非退化特征值的矩阵来说,相对应的左、右特征向量点积总可以成为单位 1。

如果特征值是退化的,则与退化特征值相对应的左、右特征向量都必须进行线性组合,才能得到分别与左或右特征向量的正交性。这总可以通过一种类似于 Gram-Schmidt 正交化方法的过程来实现。于是可以调整归一化的方法,就可以使对应的左右特征向量的非零点积成为单位 1。如果对应的左右特征向量的点积为零,则说明这样的特征向量是不完备的。注意,不完备的特征向量只有在退化特征值的情况下出现,但在这种情况下,也并不总是如此(事实上,对于"正规的"矩阵就决不会发生)。详细讨论参见本节参考文献[1]。

不管是退化或非退化的情形,最后对于非零点积的归一化产生如下结果:如果逆矩阵存在的话,左特征向量按行构成的矩阵是右特征向量按列构成的矩阵的逆。

11.0.3 矩阵的对角化

将(11.0.8)式的第一式左乘 $\mathbf{X}_L$,并利用 $\mathbf{X}_L$ 和 $\mathbf{X}_R$ 互为逆矩阵,得到

$$\mathbf{X}_R^{-1}\cdot\mathbf{A}\cdot\mathbf{X}_R=\mathrm{diag}(\lambda_0\ldots\lambda_{N-1}) \tag{11.0.10}$$

这是对矩阵 $\mathbf{A}$ 的**相似变换**的一种特殊情形:

$$\mathbf{A}\quad\rightarrow\quad\mathbf{Z}^{-1}\cdot\mathbf{A}\cdot\mathbf{Z} \tag{11.0.11}$$

其中 $\mathbf{Z}$ 是某个变换矩阵。相似变换在特征计算中起着决定性的作用,因为相似变换不改变一个矩阵的特征值。这很容易从以下推导看出:

$$\begin{aligned}\det\left|\mathbf{Z}^{-1}\cdot\mathbf{A}\cdot\mathbf{Z}-\lambda\mathbf{1}\right|&=\det\left|\mathbf{Z}^{-1}\cdot(\mathbf{A}-\lambda\mathbf{1})\cdot\mathbf{Z}\right|\\&=\det|\mathbf{Z}|\ \det|\mathbf{A}-\lambda\mathbf{1}|\ \det\left|\mathbf{Z}^{-1}\right|\\&=\det|\mathbf{A}-\lambda\mathbf{1}|\end{aligned} \tag{11.0.12}$$

(11.0.10)式说明,任何具有完备特征向量的矩阵(包括所有正规矩阵和"大多数"随机非正规矩阵),可以通过相似变换而对角化,实现对角化的变换矩阵之列就是右特征向量,而它的逆矩阵之行就是左特征向量。

对实对称矩阵来说,特征向量是实的并且是归一正交的,因而变换矩阵是正交的。所以这一相似变换也是一种下列形式的**正交变换**:

$$\mathbf{A} \quad \rightarrow \quad \mathbf{Z}^T \cdot \mathbf{A} \cdot \mathbf{Z} \tag{11.0.13}$$

实的非对称性矩阵可以通过它们的完备特征向量进行对角化,但变换矩阵不一定是实的。实际情况是,一个实的相似变换可以"基本上"实现对角化,它可以将矩阵化简成沿主对角线只有一些两行两列的小块,而其他部分元素均为零。每一个两行两列的小块对应一对复共轭的复特征值。在本章的后面将会看到利用这一思想的一些程序。

几乎所有现代特征系统的计算程序的"重大策略"都是通过一系列相似变换,将矩阵 $\mathbf{A}$ 接近对角形式:

$$\begin{aligned} \mathbf{A} \quad &\rightarrow \quad \mathbf{P}_1^{-1} \cdot \mathbf{A} \cdot \mathbf{P}_1 \quad \rightarrow \quad \mathbf{P}_2^{-1} \cdot \mathbf{P}_1^{-1} \cdot \mathbf{A} \cdot \mathbf{P}_1 \cdot \mathbf{P}_2 \\ &\rightarrow \quad \mathbf{P}_3^{-1} \cdot \mathbf{P}_2^{-1} \cdot \mathbf{P}_1^{-1} \cdot \mathbf{A} \cdot \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \quad \rightarrow \quad \text{等} \end{aligned} \tag{11.0.14}$$

如果最终得到了对角形式,则特征向量是所有变换乘积的列向量:

$$\mathbf{X}_R = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \cdot \ldots \tag{11.0.15}$$

有时,不需要一直变换成对角形式。例如,只关心特征值而不是特征向量,那么将矩阵 $\mathbf{A}$ 变成三角形式已经足够了,即对角线以下(或以上)的元素都为零。在这种情况下,对角元素已经是特征值了,这可以通过将(11.0.2)式用于行列式展开来验证。

有两组不同的技术可以用来实施上述重大策略(11.0.14)。实际上,它们组合起来使用效果较好,所以大多数现代特征系统程序都使用了这两组技巧。第一组技巧是,由一系列用来完成不同特殊功能的"原子"变换 $\mathbf{P}_i$ 构成,例如将某一特定非对角元素零化(雅可比(Jacobi)变换,参见11.1节);或将某一特定的整行或整列零化(Householder变换,参见11.2节;消去法,参见11.5节)。一般说来,有限次这样的简单变换不能将整个矩阵对角化。于是有两种选择:或者用有限次该变换完成大部分约化工作(例如,转化成一些诸如**三对角形**或**Hessenberg**的特殊形式,参见11.2节和11.5节),然后再使用将要提到的第二组技术;或者反复应用这一系列有限次简单的变换,直到矩阵与对角形式之间的差别可忽略。后一种方法在概念上最简单,所以将在下一节中讨论它。尽管如此,对于 N 大小 ~10,这种方法在计算上的非有效性大约有常数因子 ~5,所以效率很低。

第二种更加精巧的技术称为**因式分解法**。假设矩阵 $\mathbf{A}$ 能被分解成一个左因子 $\mathbf{F}_L$ 和一个右因子 $\mathbf{F}_R$,则

$$\mathbf{A} = \mathbf{F}_L \cdot \mathbf{F}_R \qquad \text{或等价地} \qquad \mathbf{F}_L^{-1} \cdot \mathbf{A} = \mathbf{F}_R \tag{11.0.16}$$

如果将这两因子交换相乘,并利用(11.0.16)式中的第二式,得到

$$\mathbf{F}_R \cdot \mathbf{F}_L = \mathbf{F}_L^{-1} \cdot \mathbf{A} \cdot \mathbf{F}_L \tag{11.0.17}$$

上式可以看做是对 $\mathbf{A}$ 进行了一个相似变换,其变换矩阵是 $\mathbf{F}_L$。在11.3节和11.6节中将讨论体现这种思想的**QR方法**。

因式分解法同样地不能经过有限次变换就准确地收敛。但某一些较好的因式分解法确实能快速地而且稳定地收敛,并且当在一个使用简单相似变换的适当的初始简化时,它们则是应选择的方法。

11.0.4 成品化特征系统程序的特征系统软件包

现在可能已经意识到,特征系统的求解是个很复杂的问题。确实如此。这正是本书所涉及的少数几个主题中的一个。关于这个主题,建议使用已成品化的程序。本章的目的是,要正确地告诉读者这些成品化程序的内部工作情况,以便在使用它们时能审慎地进行选择,并且当出现错误时能明智地予以判断。

当今几乎所有的成品化程序都可以追溯到它们的前身,即Wilkinson和Reinsch发表的"Handbook for Automatic Computation, Vol. Ⅱ, Linear Algebra"[2](后简称为Handbook)中的程序。该书包含了许多位作者的优秀论文,它们是这一领域的权威之作。Handbook中公开的部分FORTRAN程序是EISPACK程序集[3]。本章中程序都是从Handbook或EISPACK中的程序转译过来的,所以理解了这些程序将会有助于增进对那些典型软件包的理解。

IMSL[4]和NAG[5]都提供了用FORTRAN和C编写的专用程序的实施,这些程序基本上是Handbook中的程序。

一个好的"特征系统软件包"将提供进行如下所需计算的不同的程序或系列程序的不同途径:

- 所有的特征值但不求特征向量
- 所有的特征值和某些对应的特征向量
- 所有的特征值和所有对应的特征向量

区别这些的目的是为了节省计算时间和存储量,计算那些并不需要的特征向量是很浪费工时的。人们经常只需知道几个最大的或几个绝对值最大的、或几个负的特征值所对应的特征向量。如果只需要不到总数的四分之一的特征向量,那么采用计算这些部分特征向量的方法要比计算所有特征向量的方法更为有效。

一个好的特征系统软件包以应该对每个如下特殊形式的矩阵提供上面的每种计算的不同途径:

- 实、对称、三对角型矩阵
- 实、对称、带状(只有少数次对角或超对角元素是非零的)矩阵
- 实、对称矩阵
- 实、非对称矩阵
- 复、埃尔米特矩阵
- 复、非埃尔米特矩阵

区分的目的同样是为了节省时间和存储量。它是通过采用在任何特定应用中都适用的最少通用程序来达到这个目的的。

下面,我们给出实施有关下述问题的优秀程序所处的章节:

- 计算实对称三对角型矩阵的所有特征值和特征向量(参见11.3节)
- 计算实对称矩阵的所有特征值和特征向量(参见11.1节~11.3节)
- 计算复埃尔米特矩阵的所有特征值和特征向量(参见11.4节)
- 计算实非对称矩阵的所有特征值,不计算特征向量(参见11.5节~11.6节)

还将在11.7节中讨论如何用逆迭代的方法求得非对称矩阵的某些特征向量。

11.0.5 广义的和非线性特征值问题

许多特征系统程序包还可以处理所谓的广义特征问题[6]:

$$\mathbf{A}\cdot\mathbf{x}=\lambda\mathbf{B}\cdot\mathbf{x} \tag{11.0.18}$$

其中**A**和**B**都是矩阵。在很多这类问题中,**B**是非奇异的,它能用下面的等式处理:

$$(\mathbf{B}^{-1}\cdot\mathbf{A})\cdot\mathbf{x}=\lambda\mathbf{x} \tag{11.0.19}$$

通常**A**和**B**是对称的,并且**B**是正定矩阵。在(11.0.19)式中矩阵$\mathbf{B}^{-1}\cdot\mathbf{A}$是非对称矩阵,但可以通过采用2.9节中介绍的Cholesky(楚列斯基)分解$\mathbf{B}=\mathbf{L}\cdot\mathbf{L}^T$方法,将它转换成对称特征值问题。(11.0.18)式乘以$\mathbf{L}^{-1}$,得

$$\mathbf{C}\cdot(\mathbf{L}^T\cdot\mathbf{x})=\lambda(\mathbf{L}^T\cdot\mathbf{x}) \tag{11.0.20}$$

其中

$$\mathbf{C}=\mathbf{L}^{-1}\cdot\mathbf{A}\cdot(\mathbf{L}^{-1})^T \tag{11.0.21}$$

矩阵**C**是对称的并且与原问题(11.0.18)式有相同的特征值。它的特征函数是$\mathbf{L}^T\cdot\mathbf{x}$。构成**C**矩阵的有效方法是,首先求解方程

$$\mathbf{Y}\cdot\mathbf{L}^T=\mathbf{A} \tag{11.0.22}$$

得矩阵**Y**的下三角元素。然后,对于对称矩阵**C**的下三角元素,可通过求解

$$\mathbf{L}\cdot\mathbf{C}=\mathbf{Y} \tag{11.0.23}$$

得到。

另一类广义的标准特征值问题是特征值 λ 中的非线性问题，例如

$$(\mathbf{A}\lambda^2 + \mathbf{B}\lambda + \mathbf{C}) \cdot \mathbf{x} = 0 \tag{11.0.24}$$

这类问题可以通过引入一个附加的未知特征向量 $\mathbf{y}$，并求解 $2N \times 2N$ 阶特征系统，而转换成线性问题。这个特征系统是

$$\begin{pmatrix} 0 & \mathbf{1} \\ -\mathbf{A}^{-1} \cdot \mathbf{C} & -\mathbf{A}^{-1} \cdot \mathbf{B} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \lambda \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \tag{11.0.25}$$

这类技术归纳为 λ 的高阶多项式问题。M 阶多项式产生线性的 $MN \times MN$ 阶特征系统(参见本节参考文献[7])。

参考文献和深入阅读

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), Chapter 6. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [3]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [4]

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.), Chapter F02. [5]

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §7.7. [6]

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press). [7]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington. Mathematical Association of America), Chapter 13.

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press).

11.1 对称矩阵的雅可比变换

雅可比方法是由一系列形如(11.0.14)式的正交相似变换组成的。每一步变换(**雅可比旋转**)是用来将某一非对角元素零化的平面旋转。相继的变换会把以前已经零化的元素再赋值，尽管如此，非对角元素会变得越来越小，直到在机器的精度内矩阵对角化。将所有变换累乘起来，就得到特征向量的矩阵(如(11.0.15)式的形式)，而最终对角化矩阵的对角元素就是特征值。

对所有实对称矩阵来说，雅可比方法是非常简单易行的。但当矩阵维数大于 10 以后，这种算法将会以一个显著的常数因子慢下来(与 11.3 节中将要介绍的 QR 方法相比较)。不管怎样，雅可比方法比大多数更有效的方法要简单得多。对于计算代价不是考虑的主要问题的时候，并且维数适度的矩阵，推荐使用这种方法。

基本的雅可比旋转 $\mathbf{P}_{pq}$ 是如下形式的矩阵：

$$\mathbf{P}_{pq} = \begin{bmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & c & \cdots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \cdots & c & & \\ & & & & & \cdots & \\ & & & & & & 1 \end{bmatrix} \tag{11.1.1}$$

其实除了在行列数都为 p 和 q 的对角元素为两个元素 c 外，其他对角元素均为 1。除去两个元素 s 和 $-s$ 外，所有其他非对角元素都为零。c 和 s 分别是旋转角 ϕ 的余弦和正弦，故 $c^2 + s^2 = 1$。

一个如(11.1.1)式的平面旋转按如下方式变换矩阵 $\mathbf{A}$：

$$\mathbf{A}' = \mathbf{P}_{pq}^T \cdot \mathbf{A} \cdot \mathbf{P}_{pq} \tag{11.1.2}$$

$\mathbf{P}_{pq}^T \cdot \mathbf{A}$ 只改变 $\mathbf{A}$ 的 p 行和 q 行，而 $\mathbf{A} \cdot \mathbf{P}_{pq}$ 只改变 $\mathbf{A}$ 的 p 列和 q 列。注意，这里用的下标 p 和 q 并不是指 $\mathbf{P}_{pq}$ 中的元素，而是用以表明矩阵是哪一种类型的旋转，即标明它所作用哪些行和列。因此，(11.1.2)式中被改变的 $\mathbf{A}$ 中的元素只是 p、q 的行和列，写出如下形式：

$$\mathbf{A}' = \begin{bmatrix} & & a'_{0p} & & a'_{0q} & & \\ & & \vdots & & \vdots & & \\ a'_{p0} & \cdots & a'_{pp} & \cdots & a'_{pq} & \cdots & a'_{p,n-1} \\ & & \vdots & & \vdots & & \\ a'_{q0} & \cdots & a'_{qp} & \cdots & a'_{qq} & \cdots & a'_{q,n-1} \\ & & \vdots & & \vdots & & \\ & & a'_{n-1,p} & & a'_{n-1,q} & & \end{bmatrix} \tag{11.1.3}$$

计算(11.1.2)式，并利用 $\mathbf{A}$ 的对称性，得到如下显式的表达式：

$$\begin{aligned} a'_{rp} &= ca_{rp} - sa_{rq} \\ a'_{rq} &= ca_{rq} + sa_{rp} \end{aligned} \qquad r \neq p,\ r \neq q \tag{11.1.4}$$

$$a'_{pp} = c^2 a_{pp} + s^2 a_{qq} - 2sca_{pq} \tag{11.1.5}$$

$$a'_{qq} = s^2 a_{pp} + c^2 a_{qq} + 2sca_{pq} \tag{11.1.6}$$

$$a'_{pq} = (c^2 - s^2)a_{pq} + sc(a_{pp} - a_{qq}) \tag{11.1.7}$$

雅可比方法试图通过一系列平面旋转将非对角元素零化。相应地，令 $a'_{pq} = 0$，从(11.1.7)式给出如下的旋转角ϕ表达式：

$$\theta \equiv \cot 2\phi \equiv \frac{c^2 - s^2}{2sc} = \frac{a_{qq} - a_{pp}}{2a_{pq}} \tag{11.1.8}$$

如果令 $t \equiv s/c$，θ 的定义可以重写成

$$t^2 + 2t\theta - 1 = 0 \tag{11.1.9}$$

这一方程较小的根对应着一个数量上小于 $\pi/4$ 的旋转角。每一步做这样的选择就可给出最稳定的化简。利用二次求根公式并对分母进行区分，可以写出这一较小的根为

$$t = \frac{\operatorname{sgn}(\theta)}{|\theta| + \sqrt{\theta^2 + 1}} \tag{11.1.10}$$

如果 θ 很大，以至于 θ^2 将会在计算机上溢出，令 $t = 1/(2\theta)$。于是就有

$$c = \frac{1}{\sqrt{t^2 + 1}} \tag{11.1.11}$$

$$s = tc \tag{11.1.12}$$

当数值地应用(11.1.4)式～(11.1.7)式时，将它们重写，以使舍入误差达到最小。(11.1.7)式由下式代替：

$$a'_{pq} = 0 \tag{11.1.13}$$

剩下等式的指导思想就是，将它们的新值写成旧值加上一个小的修正。于是利用(11.1.7)式和(11.1.13)式将(11.1.5)式中的 a_{pq} 消去，得到

$$a'_{pp} = a_{pp} - ta_{pq} \tag{11.1.14}$$

类似地有

$$a'_{qq} = a_{qq} + ta_{pq} \tag{11.1.15}$$

$$a'_{rp} = a_{rp} - s(a_{rq} + \tau a_{rp}) \tag{11.1.16}$$

$$a'_{rq} = a_{rq} + s(a_{rp} - \tau a_{rq}) \tag{11.1.17}$$

其中$\tau(=\tan\phi/2)$定义为

$$\tau \equiv \frac{s}{1+c} \tag{11.1.18}$$

可以通过所有非对角元素的平方和看出雅可比方法的收敛性

$$S = \sum_{r \neq s} |a_{rs}|^2 \tag{11.1.19}$$

由(11.1.4)式至(11.1.7)式可得

$$S' = S - 2|a_{pq}|^2 \tag{11.1.20}$$

(因为转换是正交的,对角元素的平方和相应地增加 $2|a_{pq}|^2$.)于是 S 的序列单调递减。因为序列的下界为零,而且可选择 a_{pq} 为我们所需要的值,故这一序列可以收敛到零。

最终可以得到在机器精度内的对角矩阵 **D**。它的对角元素给出原来矩阵 **A** 的特征值,由于

$$\mathbf{D} = \mathbf{V}^T \cdot \mathbf{A} \cdot \mathbf{V} \tag{11.1.21}$$

其中

$$\mathbf{V} = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \cdots \tag{11.1.22}$$

$\mathbf{P}_i$ 是一系列相继的雅可比旋转矩阵。**V** 的列是特征向量(由于 $\mathbf{A}\cdot\mathbf{V}=\mathbf{V}\cdot\mathbf{D}$)。它们可以通过每一步计算

$$\mathbf{V}' = \mathbf{V} \cdot \mathbf{P}_i \tag{11.1.23}$$

而得出,其中最开始的 **V** 是原来的矩阵 **A**。具体地,(11.1.23)式可写成

$$\begin{aligned} v'_{rs} &= v_{rs} \qquad (s \neq p,\ s \neq q) \\ v'_{rp} &= cv_{rp} - sv_{rq} \\ v'_{rq} &= sv_{rp} + cv_{rq} \end{aligned} \tag{11.1.24}$$

也可像(11.1.16)式和(11.1.17)式那样,用τ将这些公式重写而减小舍入误差。

剩下的惟一问题就是消元的次序问题。1846 年雅可比的最初算法是,每一步搜索出矩阵的上三角部分中的最大元素,并将其置为零。这对于手算来说,是一种合理的方法;但在计算机中,仅只寻找最大元素这一项就使每一步雅可比旋转的运算是 N^2 而不是 N。

一种更好的方法叫**循环雅可比法**,就是按严格的顺序来零化各元素。例如,可以简单地按行进行消去:$\mathbf{P}_{01}, \mathbf{P}_{02}, \ldots, \mathbf{P}_{0,n-1}$,然后 $\mathbf{P}_{12}, \mathbf{P}_{13}$ 等。可以证明,对于非退化特征值的情况,原来的方法和这种循环雅可比法一般都是二次收敛的。这样一组 $n(n-1)/2$ 个雅可比旋转称为一次**清扫**。

下面是基于本节参考文献[1,2]中的程序,并使用了两项改进:

- 在前三次清扫中,只对 $|a_{pq}| > \epsilon$的元素进行 pq 旋转,其中ϵ是某一阈值

$$\epsilon = \frac{1}{5}\frac{S_0}{n^2} \tag{11.1.25}$$

其中 S_0 是非对角元的模数之和:

$$S_0 = \sum_{r<s} |a_{rs}| \tag{11.1.26}$$

- 四次清扫之后,如果 $|a_{pq}| \ll |a_{pp}|$,并且 $|a_{pq}| \ll |a_{qq}|$,则令 $|a_{pq}| = 0$,并跳过这次旋转。比较的标准是 $|a_{pq}| < 10^{-(D+2)}|a_{pp}|$,其中 D 是机器的十进制有效位数,对 $|a_{qq}|$ 也同样处理。

在下列程序中,n×n 的对称矩阵 a 存储在数组 a[0..n-1][0..n-1]中。输出时 a 的超对角元素都已被破坏,但对角和次对角元素还是原来矩阵 a 的值。向量 d[0..n-1]返回 a 的特征值,在计算过程中它存有当前 a 的对角元素值。矩阵 v[0..n-1][0..n-1]输出归一化的特征向量,其中第 k 列对应特征值 d[k]。参数 nrot 是达到收敛所需的雅可比旋转次数。

典型的矩阵需要 6 到 10 次清扫,也就是 $3n^2$ 到 $5n^2$ 次雅可比旋转。每次旋转需要 $4n$ 次操作,每次操作

包括乘和加,因而总运算量是 $12n^3$ 到 $20n^3$ 量级的。计算特征值的同时若还需计算特征向量,则每次旋转的操作数从 $4n$ 变成 $6n$ 次,仅增加了 50% 的运算量。

```
#include <cmath>
#include "nr.h"
using namespace std;

namespace {
    inline void rot(Mat_IO_DP &a, const DP s, const DP tau, const int i,
        const int j, const int k, const int l)
    {
        DP g,h;

        g=a[i][j];
        h=a[k][l];
        a[i][j]=g-s*(h+g*tau);
        a[k][l]=h+s*(g-h*tau);
    }
}

void NR::jacobi(Mat_IO_DP &a, Vec_O_DP &d, Mat_O_DP &v, int &nrot)
计算实对称矩阵 a[0..n-1][0..n-1]的所有特征值和特征向量。输出时 a 的对角线以上的元素被破坏。d[0..n-1]
返回 a 的特征值。输出的 v[0..n-1][0..n-1]的列是 a 的规范特征向量。nrot 返回所需的雅可比旋转次数。
{
    int i,j,ip,iq;
    DP tresh,theta,tau,t,sm,s,h,g,c;

    int n=d.size();
    Vec_DP b(n),z(n);
    for (ip=0;ip<n;ip++) {                          对单位矩阵初始赋值。
        for (iq=0;iq<n;iq++) v[ip][iq]=0.0;
        v[ip][ip]=1.0;
    }
    for (ip=0;ip<n;ip++) {                          将 b 和 d 初始化赋值成 a 的对角
        b[ip]=d[ip]=a[ip][ip];                      元素。
        z[ip]=0.0;                                  该向量将按(11.1.14)式中的 $ta_{pq}$
    }                                               项积累。
    nrot=0;
    for (i=1;i<=50;i++) {
        sm=0.0;
        for (ip=0;ip<n-1;ip++) {                    将非对角元素求和。
            for (iq=ip+1;iq<n;iq++)
                sm += fabs(a[ip][iq]);
        }
        if (sm == 0.0)                              正常返回,依赖于四次收敛到机器
            return;                                 下溢出。
        if (i < 4)
            tresh=0.2*sm/(n*n);                     …… 在前三次扫描中。
        else
            tresh=0.0;                              …… 这以后。
        for (ip=0;ip<n-1;ip++) {
            for (iq=ip+1;iq<n;iq++) {
                g=100.0*fabs(a[ip][iq]);
                四次扫描后,如果非对角元素很小,则跳过旋转。
                if (i > 4 && (fabs(d[ip])+g) == fabs(d[ip])
                    && (fabs(d[iq])+g) == fabs(d[iq]))
                        a[ip][iq]=0.0;
                else if (fabs(a[ip][iq]) > tresh) {
                    h=d[iq]-d[ip];
                    if ((fabs(h)+g) == fabs(h))
                        t=(a[ip][iq])/h;            t = 1/(2θ)
                    else {
                        theta=0.5*h/(a[ip][iq]);    (11.1.10)式。
                        t=1.0/(fabs(theta)+sqrt(1.0+theta*theta));
                        if (theta < 0.0) t = -t;
                    }
                    c=1.0/sqrt(1+t*t);
                    s=t*c;
                    tau=s/(1.0+c);
                    h=t*a[ip][iq];
                    z[ip] -= h;
```

```
                    z[iq] += h;
                    d[ip] -= h;
                    d[iq] += h;
                    a[ip][iq]=0.0;
                    for (j=0;j<ip;j++)                  0 ≤ j < p 旋转的情形。
                        rot(a,s,tau,j,ip,j,iq);
                    for (j=ip+1;j<iq;j++)               p < j < q 旋转的情形。
                        rot(a,s,tau,ip,j,j,iq);
                    for (j=iq+1;j<n;j++)                q < j < n 旋转的情形。
                        rot(a,s,tau,ip,j,iq,j);
                    for (j=0;j<n;j++)
                        rot(v,s,tau,j,ip,j,iq);
                    ++nrot;
                }
            }
        }
        for (ip=0;ip<n;ip++) {
            b[ip] += z[ip];
            d[ip]=b[ip];                                将 d 修改成 ta_pq 的和，
            z[ip]=0.0;                                  并重新初始化 z。
        }
    }
    nrerror("Too many iterations in routine jacobi");
}
```

注意，上述程序是假设下溢被置为零。在不满足此条件的机器上，必须修改此程序。

输出时的特征值不是按顺序的。如果需要排序，可以调用下列程序将程序 jacobi 或本章其他程序的输出重新排序(所用的方法是直接插入法，其运算量是 N^2 而不是 $N \log N$ 量级。但由于原程序中刚刚使用了一个 N^3 量级的过程来寻找特征值，所以开销不算大)。

```
#include "nr.h"

void NR::eigsrt(Vec_IO_DP &d, Mat_IO_DP &v)
给定 jacobi(参见 11.1 节)或 tqli(参见 11.3 节)的输出的特征值 d[0..n-1]和特征向量 v[0..n-1][0..n-1]，本程序将
特征值按降序排列，并对特征向量 v 进行相应的调整。所用方法是直接插入法。
{
    int i,j,k;
    DP p;

    int n=d.size();
    for (i=0;i<n-1;i++) {
        p=d[k=i];
        for (j=i;j<n;j++)
            if (d[j] >= p) p=d[k=j];
        if (k != i) {
            d[k]=d[i];
            d[i]=p;
            for (j=0;j<n;j++) {
                p=v[j][i];
                v[j][i]=v[j][k];
                v[j][k]=p;
            }
        }
    }
}
```

参考文献和深入阅读

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §8.4.

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

11.2 将对称矩阵约化为三对角形式:Givens 约化和 Householder 约化

正如已经提到的,找寻特征值和特征向量的最优策略是,首先将矩阵化成一个简单的形式,再开始一个迭代过程。对于对称矩阵来说,最好的简单形式是三对角形式。Givens(吉文斯)约化是对雅可比方法的一个改进。当把矩阵化简为三对角形式时就停止运算,而不是把它一直化到为对角形式。这就可以使整个过程用有限步完成,而不是像雅可比方法那样一直迭代到收敛

11.2.1 Givens 方法

对于 Givens 方法,在(11.1.1)式中选择旋转角来将上的元素化为零不在四个"角"上(即不是(11.1.3)式中的 a_{pp}、a_{pq}或 a_{qq})。我们特别地选择 $\mathbf{P}_{12}$来将 a_{21}(对称地还有 a_{12})化为零。然后选择 $\mathbf{P}_{13}$将 a_{31}化为零一般地,选择序列

$$\mathbf{P}_{12},\mathbf{P}_{13},\ldots,\mathbf{P}_{1,n-1};\mathbf{P}_{23},\ldots,\mathbf{P}_{2,n-1};\ldots;\mathbf{P}_{n-2,n-1}$$

其中 $\mathbf{P}_{jk}$用来将 $a_{k,j-1}$化为零。这种方法能够奏效是由于按照(11.1.4)式,a'_{rp}和 a'_{rq}($r\neq p$、$r\neq q$)是旧值 a_{rp}和 a_{rq}的线性组合。因而如果 a_{rp}和 a_{rq}已被零化,当约化过程继续进行时它们仍保持为零。明显地最终需要 $n^2/2$ 次旋转,而这种方法的直接实施需要 $4n^3/3$ 量级的乘法次数,这不包括为求得特征向量而进行的矩阵乘法。

随后将要讨论的 Householder(豪斯霍尔德)方法像 Givens 约化方法一样稳定而且双倍地有效,所以通常不使用 Givens 方法。最近的工作(参见本节参考文献[1])表明,通过将 Givens 方法重新表述整理,可以将操作次数减少一倍,并且可以避免取平方根的运算。这使得这种算法可以和 Householder 约化方法相竞争。尽管这样,我们还是没有足够的经验来认为这种方法比 Householder 方法更可取。

11.2.2 Householder 方法

Householder 算法用 $n-2$ 个正交变换将 $n\times n$ 的对称矩阵 $\mathbf{A}$ 约化成三对角形式。每个变换将一整行和一整列中的相应元素化为零。基本的成分是如下形式的 Householder 矩阵 $\mathbf{P}$:

$$\mathbf{P}=\mathbf{1}-2\mathbf{w}\cdot\mathbf{w}^T \tag{11.2.1}$$

其中 $\mathbf{w}$ 是满足 $|\mathbf{w}|^2=1$ 的实向量(在当前的表示方法中,两个向量 $\mathbf{a}$ 和 $\mathbf{b}$ 的外积或矩阵积写成 $\mathbf{a}\cdot\mathbf{b}^T$,而它们的内积或标量积写成 $\mathbf{a}^T\cdot\mathbf{b}$)。矩阵 $\mathbf{P}$ 是正交的,因为

$$\begin{aligned}\mathbf{P}^2&=(\mathbf{1}-2\mathbf{w}\cdot\mathbf{w}^T)\cdot(\mathbf{1}-2\mathbf{w}\cdot\mathbf{w}^T)\\&=\mathbf{1}-4\mathbf{w}\cdot\mathbf{w}^T+4\mathbf{w}\cdot(\mathbf{w}^T\cdot\mathbf{w})\cdot\mathbf{w}^T\\&=\mathbf{1}\end{aligned} \tag{11.2.2}$$

所以 $\mathbf{P}=\mathbf{P}^{-1}$。但因 $\mathbf{P}^T=\mathbf{P}$,于是 $\mathbf{P}^T=\mathbf{P}^{-1}$,这就证明了 $\mathbf{P}$ 的正交性。

将 $\mathbf{P}$ 重写成

$$\mathbf{P}=\mathbf{1}-\frac{\mathbf{u}\cdot\mathbf{u}^T}{H} \tag{11.2.3}$$

其中的标量 H 是

$$H\equiv\frac{1}{2}|\mathbf{u}|^2 \tag{11.2.4}$$

现在 $\mathbf{u}$ 可以是任何向量。假设 $\mathbf{x}$ 是矩阵 $\mathbf{A}$ 的第一列构成的向量。选择

$$\mathbf{u}=\mathbf{x}\mp|\mathbf{x}|\mathbf{e}_0 \tag{11.2.5}$$

其中 $\mathbf{e}_0$ 是单位向量$[1,0,\ldots,0]^T$,有关符号的选择后面将讨论。于是有

$$
\begin{aligned}
\mathbf{P}\cdot\mathbf{x} &= \mathbf{x} - \frac{\mathbf{u}}{H}\cdot(\mathbf{x}\mp|\mathbf{x}|\mathbf{e}_0)^T\cdot\mathbf{x} \\
&= \mathbf{x} - \frac{2\mathbf{u}\cdot(|\mathbf{x}|^2\mp|\mathbf{x}|x_0)}{2|\mathbf{x}|^2\mp 2|\mathbf{x}|x_0} \\
&= \mathbf{x} - \mathbf{u} \\
&= \pm|\mathbf{x}|\mathbf{e}_0
\end{aligned}
\tag{11.2.6}
$$

这表明 Householder 矩阵 $\mathbf{P}$ 作用在给定向量 $\mathbf{x}$ 上就将零化除第一个元素外的所有元素。

为了将对称矩阵 $\mathbf{A}$ 化为三对角形式，我们选择第一个 Householder 矩阵的向量 $\mathbf{x}$ 为矩阵 $\mathbf{A}$ 第一列下面的 $n-1$ 个元素。于是以下 $n-2$ 个元素将被零化：

$$
\mathbf{P}_1\cdot\mathbf{A} = \left[\begin{array}{c|ccc} 1 & 0 & 0 \quad\cdots & 0 \\ \hline 0 & & & \\ 0 & & & \\ \vdots & & {}^{(n-1)}\mathbf{P}_1 & \\ 0 & & & \end{array}\right]\cdot\left[\begin{array}{c|ccc} a_{00} & a_{01} & a_{02} \quad\cdots & a_{0,n-1} \\ \hline a_{10} & & & \\ a_{20} & & & \\ \vdots & & \text{不相关的} & \\ a_{n-1,0} & & & \end{array}\right]
$$

$$
= \left[\begin{array}{c|ccc} a_{00} & a_{01} & a_{02} \quad\cdots & a_{0,n-1} \\ \hline k & & & \\ 0 & & & \\ \vdots & & \text{不相关的} & \\ 0 & & & \end{array}\right]
\tag{11.2.7}
$$

这里我们将矩阵写成分块的形式，${}^{(n-1)}\mathbf{P}$ 表示一个 $(n-1)\times(n-1)$ 维的 Householder 矩阵。k 就是向量 $[a_{10},\ldots,a_{n-1,0}]^T$ 的模的正值或负值。

完整的正交变换是

$$
\mathbf{A}' = \mathbf{P}\cdot\mathbf{A}\cdot\mathbf{P} = \left[\begin{array}{c|ccc} a_{00} & k & 0 \quad\cdots & 0 \\ \hline k & & & \\ 0 & & & \\ \vdots & & \text{不相关的} & \\ 0 & & & \end{array}\right]
\tag{11.2.8}
$$

这里已经用到了 $\mathbf{P}^T=\mathbf{P}$。

现在选择第二个 Householder 矩阵的 $\mathbf{x}$ 为第二列下面的 $n-2$ 个元素，从而构成矩阵

$$
\mathbf{P}_2 \equiv \left[\begin{array}{cc|ccc} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \hline 0 & 0 & & & \\ \vdots & \vdots & & {}^{(n-2)}\mathbf{P}_2 & \\ 0 & 0 & & & \end{array}\right]
\tag{11.2.9}
$$

左上角的单位块保证了第一步已经三对角化了的部分不会被破坏，而 $(n-2)$ 维的 Householder 矩阵 ${}^{(n-2)}\mathbf{P}_2$ 将新的一行和一列三对角化。显然，一系列 $n-2$ 个这样的变换就可将矩阵 $\mathbf{A}$ 化为三对角形式。

为了避免直接进行矩阵乘法 $\mathbf{P}\cdot\mathbf{A}\cdot\mathbf{P}$，我们计算一个向量

$$
\mathbf{p} \equiv \frac{\mathbf{A}\cdot\mathbf{u}}{H}
\tag{11.2.10}
$$

则

$$\mathbf{A}\cdot\mathbf{P}=\mathbf{A}\cdot(1-\frac{\mathbf{u}\cdot\mathbf{u}^T}{H})=\mathbf{A}-\mathbf{p}\cdot\mathbf{u}^T$$

$$\mathbf{A}'=\mathbf{P}\cdot\mathbf{A}\cdot\mathbf{P}=\mathbf{A}-\mathbf{p}\cdot\mathbf{u}^T-\mathbf{u}\cdot\mathbf{p}^T+2K\mathbf{u}\cdot\mathbf{u}^T$$

其中 K 定义为

$$K=\frac{\mathbf{u}^T\cdot\mathbf{p}}{2H} \tag{11.2.11}$$

如果写

$$\mathbf{q}\equiv\mathbf{p}-K\mathbf{u} \tag{11.2.12}$$

就得到

$$\mathbf{A}'=\mathbf{A}-\mathbf{q}\cdot\mathbf{u}^T-\mathbf{u}\cdot\mathbf{q}^T \tag{11.2.13}$$

这是一个计算上很有用的公式。

按照本节参考文献[2]，下面给出的 Householder 约化程序是从 A 的第 $n-1$ 列开始，而不是如上面介绍的从第 0 列开始。有关的公式如下：在第 m 步($m=1,2,\ldots,n-2$)向量 $\mathbf{u}$ 具有形式

$$\mathbf{u}^T=[a_{i0},a_{i1},\ldots,a_{i,i-2},a_{i,i-1}\pm\sqrt{\sigma},0,\ldots,0] \tag{11.2.14}$$

其中

$$i\equiv n-m=n-1,n-2,\ldots,2 \tag{11.2.15}$$

σ(在以前的记号为 $|x|^2$)是

$$\sigma=(a_{i0})^2+\cdots+(a_{i,i-1})^2 \tag{11.2.16}$$

为了减小舍入误差，选择(11.2.14)式中的σ的符号与 $a_{i,i-1}$的符号相同。

各变量的计算按下列顺序进行：$\sigma,\mathbf{u},H,\mathbf{p},K,\mathbf{q},\mathbf{A}'$。在进行第 m 步时，A 的最后 $m-1$ 行和列已被三对角化。

如果最后三对角矩阵的特征向量已被找到(例如，通过下一节的程序)，则 A 的特征向量就是将下列联合变换作用到那些特征向量上：

$$\mathbf{Q}=\mathbf{P}_1\cdot\mathbf{P}_2\cdots\mathbf{P}_{n-2} \tag{11.2.17}$$

在所有 P 确定之后可通过递归来得到 Q：

$$\begin{aligned}\mathbf{Q}_{n-2}&=\mathbf{P}_{n-2}\\ \mathbf{Q}_j&=\mathbf{P}_j\cdot\mathbf{Q}_{j+1},\qquad j=n-3,\ldots,1\\ \mathbf{Q}&=\mathbf{Q}_1\end{aligned} \tag{11.2.18}$$

下面程序的输入是实对称矩阵 a[0..n-1][0..n-1]。输出时，a 中包含有正交矩阵 q 的元素。向量 d[0..n-1]被置成三对角矩阵 A′的对角元素，而向量 e[0..n-1]从 e[1]到 e[n-1]存有 A′非对角元素，且设 e[0]=0。注意由于计算中 a 被重新写入数值，如果后续计算中需要的话，应该在程序调用前将 a 复制并保存起来。

不需要对中间结果设置特殊的存储数组。在第 m 步，向量 $\mathbf{p}$ 和 $\mathbf{q}$ 只有 $0,\ldots,i$ 的元素是非零的(回忆一下 $i=n-m$)，而 $\mathbf{u}$ 只有第 $0,\ldots,i-1$ 元素是非零的。向量 e 的元素是按照第 $n-1,n-2,\ldots$ 的顺序确定的，所以可将向量 $\mathbf{p}$ 的非零元素存储在向量 e 的尚未确定部分，向量 $\mathbf{q}$ 可在 $\mathbf{p}$ 不再需要时写在 $\mathbf{p}$ 的位置上。将 $\mathbf{u}$ 存储在 a 的第 i 行，并把 $\mathbf{u}/H$ 存储在 a 的第 i 列。当约化结束之后，可用存储在 a 中的 $\mathbf{u}$ 和 $\mathbf{u}/H$ 来计算矩阵 $\mathbf{Q}_j$。因为 $\mathbf{Q}_j$ 的后 $n-j$ 行和列是单位矩阵，故只需计算前 $n-j-1$ 行和前 $n-j-1$ 列。这就可以重新存入 a 中 $\mathbf{u}$ 和 $\mathbf{u}/H$ 的对应行和列，因为后续 $\mathbf{Q}$ 不再需要这些位置的元素。

下面给出的程序 tred2 还包括一项改进。如果在任何一步中，数量 σ 是零或“小量”，就可以跳过对应变换。一个简单的标准，例如下式：

$$\sigma<\frac{\text{在机器上可表示的最小正数}}{\text{机器精度}}$$

在大多数情况下是合适的。实际上，所应用的是一种更详细的标准。在第 i 步，定义量

$$\epsilon = \sum_{k=0}^{i-1} |a_{ik}| \tag{11.2.19}$$

如果在机器精度下 $\epsilon = 0$，就跳过这次变换。否则重新定义，将

$$a_{ik} \quad 变为 \quad a_{ik}/\epsilon \tag{11.2.20}$$

并用重新定义了的变量进行变换(一个 Householder 变换只取决于矩阵元素之间的比例)。

注意，当所处理的矩阵的元素数值大小跨跃很大的数量级时，需将矩阵进行置换，要尽量置换成小的元素在矩阵的左上角，这一点是很重要的，因为约化是从右下角开始，而大小元素的混杂会带来很大的舍入误差。

程序 tred2 被设计成与下一节中的程序 tqli 联合使用。tqli 是用来计算对称三对角矩阵的特征值和特征向量的。在计算实对称矩阵特征值和特征向量的技术中，tred2 和 tqli 的结合是已知最有效的方法。

在下列程序清单中，由注释标出的语句只有在计算特征向量时才有用。如果只需计算特征值，则在大 n 值情况下，忽略这些标出的语句可以使 tred2 的运行速度提高一倍。在较大 n 值的限制下，只计算特征值的 Householder 约化的运算量是 $2n^3/3$ 阶，而特征值和特征向量都计算时需要 $4n^3/3$ 阶的运算量。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::tred2(Mat_IO_DP &a, Vec_O_DP &d, Vec_O_DP &e)
实对称矩阵 a[0..n-1][0..n-1]的 Householder 约化。输出时，a 被产生变换的正交矩阵 Q 取代，d[0..n-1]返回三
对角矩阵的对角元素，e[0..n-1]返回非对角元素，且有 e[0]=0。正像注释中注明的，有些程序语句在只需要求
特征值的情况下才可略去，这时 a 中不返回有用的信息。在其他情况下，这些语句应包含在程序中。
{
    int l,k,j,i;
    DP scale,hh,h,g,f;

    int n=d.size();
    for (i=n-1;i>0;i--) {
        l=i-1;
        h=scale=0.0;
        if (l > 0) {
            for (k=0;k<l+1;k++)
                scale += fabs(a[i][k]);          跳过变换。
            if (scale == 0.0)
                e[i]=a[i][l];
            else {
                for (k=0;k<l+1;k++) {
                    a[i][k] /= scale;            将标定的 u 用于变换。
                    h += a[i][k]*a[i][k];        在 h 中形成 σ。
                }
                f=a[i][l];
                g=(f >= 0.0 ? -sqrt(h) : sqrt(h));
                e[i]=scale*g;
                h -= f*g;                        现在 h 是 (11.2.4) 式。
                a[i][l]=f-g;                     将 u 存入 a 的第 i 行。
                f=0.0;
                for (j=0;j<l+1;j++) {
                // Next statement can be omitted if eigenvectors not wanted
                    a[j][i]=a[i][j]/h;           将 u/H 存入 a 的第 i 列。
                    g=0.0;                       在 g 中形成 A·u 的一个元素。
                    for (k=0;k<j+1;k++)
                        g += a[j][k]*a[i][k];
                    for (k=j+1;k<l+1;k++)
                        g += a[k][j]*a[i][k];
                    e[j]=g/h;                    在 e 的暂时不用的元素中形成 p 的元素。
                    f += e[j]*a[i][j];
                }
                hh=f/(h+h);                      形成 K，(11.2.11) 式。
                for (j=0;j<l+1;j++) {            形成 q 并存入 e 中 p 的位置上。
                    f=a[i][j];
                    e[j]=g=e[j]-hh*f;
                    for (k=0;k<j+1;k++)          约化 a，(11.2.13) 式。
```

```
                        a[j][k] -= (f*e[k]+g*a[i][k]);
                    }
                }
            } else
                e[l]=a[i][l];
            d[i]=h;
        }
        // Next statement can be omitted if eigenvectors not wanted
        d[0]=0.0;
        e[0]=0.0;
        // Contents of this loop can be omitted if eigenvectors not
        //    wanted except for statement d[i]=a[i][i];
        for (i=0;i<n;i++) {                          开始矩阵变换的积累。
            l=i;
            if (d[i] != 0.0) {                       当 i=0时跳过这一块。
                for (j=0;j<l;j++) {
                    g=0.0;
                    for (k=0;k<l;k++)                利用a中存储的u和u/H来形成P·Q。
                        g += a[i][k]*a[k][j];
                    for (k=0;k<l;k++)
                        a[k][j] -= g*a[k][i];
                }
            }
            d[i]=a[i][i];                            保留这一语句。
            a[i][i]=1.0;                             为下一次迭代将a重新置成单位矩阵。
            for (j=0;j<l;j++) a[j][i]=a[i][j]=0.0;
        }
    }
```

参考文献和深入阅读

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §5.1. [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag).

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

11.3 三对角矩阵的特征值和特征向量

11.3.1 特征多项式的赋值

一旦把原先的实对称矩阵约化为三对角矩阵,确定它的特征值的一种可能途径就是直接寻找特征值的多项式 $p_n(\lambda)$的根。对于任何试验值 λ,三对角矩阵的特征多项式都可以通过一种有效的递归关系进行求值(例如,参见本节参考文献[1])。递归过程中产生的较低阶多项式形成一个 Sturm 序列,该序列可用来将特征值定位于实轴上的某些区间。然后可以进一步使用像二分法或牛顿法那样的求根方法来更精确地确定区间。相应的特征向量可以通过逆迭代找到(参见 11.7 节)。

基于上述思想的程序可以在本节参考文献[2,3]中找到。然而,如果只需计算一小部分特征值和特征向量,则下面讨论的因式分解的方法更加有效。

11.3.2 QR 和 QL 算法

QR 算法的基本思想是,任何实矩阵可以分解成下列形式:

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{11.3.1}$$

其中 **Q** 是正交的,而 **R** 是上三角的。对于一般矩阵,这种分解可以通过采用 Householder 变换,逐列地将矩阵 **A** 的对角线以下部分零化来构成。

现在,考虑(11.3.1)式中因子交换所得到的矩阵:

$$\mathbf{A}' = \mathbf{R} \cdot \mathbf{Q} \tag{11.3.2}$$

因为 $\mathbf{Q}$ 是正交的，(11.3.1)式给出 $\mathbf{R}=\mathbf{Q}^T\cdot\mathbf{A}$。所以(11.3.2)式变成

$$\mathbf{A}'=\mathbf{Q}^T\cdot\mathbf{A}\cdot\mathbf{Q} \tag{11.3.3}$$

可以看到 $\mathbf{A}'$ 是 $\mathbf{A}$ 的一个正交变换。

可以验证，QR 变换保持了矩阵的下列性质：对称性、三对角形式和 Hessenberg 形式(将在 11.5 节中定义)。

将 $\mathbf{A}$ 的因子之一选为上三角矩阵没有特别的意义。也可以等价地使它是下三角式，这就叫做 QL 算法，因为

$$\mathbf{A}=\mathbf{Q}\cdot\mathbf{L} \tag{11.3.4}$$

其中 $\mathbf{L}$ 是下三角的(这种标准的，但又使人迷惑的 R 和 L 代表矩阵的右边或左边为非零)。

回忆一下，在 11.2 节中 Householder 方法将矩阵三对角化时，我们是从原来矩阵的第 $n-1$ 列开始的。为减小舍入误差，那时建议尽量将矩阵的最大元素放到右下角。如果现在希望将所得到的三对角矩阵对角化，则 QL 算法比 QR 算法的舍入误差小，所以今后我们使用 QL 算法。

QL 算法包括一系列正交变换：

$$\begin{aligned}\mathbf{A}_s&=\mathbf{Q}_s\cdot\mathbf{L}_s\\ \mathbf{A}_{s+1}&=\mathbf{L}_s\cdot\mathbf{Q}_s\qquad(=\mathbf{Q}_s^T\cdot\mathbf{A}_s\cdot\mathbf{Q}_s)\end{aligned} \tag{11.3.5}$$

下列定理(不是那么显然)是关于一般矩阵 $\mathbf{A}$ 的算法的基础：(i)如果 $\mathbf{A}$ 的特征值具有不同的绝对值 $|\lambda_i|$，则当 $s\to\infty$ 时 $\mathbf{A}_s\to$[下三角形式]，对角线上出现的特征值按某绝对值递增的顺序排列。(ii)如果 $\mathbf{A}$ 有 p 重特征值 $\{\lambda_i\}$，则当 $s\to\infty$ 时 $\mathbf{A}_s\to$[下三角形式]，但除去一个特征值 $\to\lambda_i$ 的 p 维对角子块阵。这一定理的证明篇幅很长，可参见本节参考文献[4]。

QL 算法对于一般矩阵的运算量是每次迭代 $O(n^3)$，这对于一般矩阵来说运算量太大了。但它对于三对角矩阵每次迭代的运算量只有 $O(n)$，而对于 Hessenberg 矩阵是 $O(n^2)$，故使该算法用于这两种形式的矩阵是很有效的。

在本节中，我们只考虑 $\mathbf{A}$ 是实、对称的三对角矩阵，因而所有的特征值 λ_i 是实的。根据上述定理，如果任一 λ_i 是 p 重的，则在次对角和超对角上至少有 $p-1$ 个零元。于是，矩阵可以分裂成能够单独对角化的子矩阵，而且在一般情况下，由对角块带来的复杂性就无关紧要了。

在上述定理的证明过程中，一般超对角元素按下列方式收敛于零：

$$a_{ij}^{(s)}\sim\left(\frac{\lambda_i}{\lambda_j}\right)^s \tag{11.3.6}$$

虽然 $\lambda_i<\lambda_j$，但如果 λ_i 与 λ_j 很接近的话，收敛的速度可能很慢。收敛可以通过**位移**的技巧来加速：如果 k 是任意常数，则 $\mathbf{A}-k\mathbf{1}$ 具有特征值 λ_i-k。如果分解

$$\mathbf{A}_s-k_s\mathbf{1}=\mathbf{Q}_s\cdot\mathbf{L}_s \tag{11.3.7}$$

于是

$$\begin{aligned}\mathbf{A}_{s+1}&=\mathbf{L}_s\cdot\mathbf{Q}_s+k_s\mathbf{1}\\ &=\mathbf{Q}_s^T\cdot\mathbf{A}_s\cdot\mathbf{Q}_s\end{aligned} \tag{11.3.8}$$

则收敛由以下比例确定：

$$\frac{\lambda_i-k_s}{\lambda_j-k_s} \tag{11.3.9}$$

这种思想是要在每一步选择位移量 k_s，以使收敛的速度最快。对于位移量的一种好的初始选择是使 k_s 接近最小的特征值 λ_0。于是第一列的非对角元素将很快趋于零。可是 λ_0 通常**事先**并非是已知的。在实践中，一个很有效的方法(虽然没有证明它是最优的)是，计算 $\mathbf{A}$ 的首个 2×2 对角子矩阵的特征值，然后令 k_s 等于离 a_{00} 较近的那个特征值。

更一般地，假设已经找到 $\mathbf{A}$ 的 r 个特征值，则可以删去前 r 个行和列而将矩阵化简成

$$\mathbf{A}=\begin{bmatrix} 0 & & & \cdots & \cdots & & & 0 \\ & \ddots & & & & & & \\ & & 0 & & & & & \\ \vdots & & & d_r & e_r & & & \vdots \\ \vdots & & & e_r & d_{r+1} & & & \\ & & & & & \cdots & & 0 \\ & & & & & & d_{n-2} & e_{n-2} \\ 0 & & & \cdots & & 0 & e_{n-2} & d_{n-1} \end{bmatrix} \tag{11.3.10}$$

选择 k_s 等于前面 2×2 子矩阵的特征值中较接近 d_r 的那一个。可以证明采用这种方法后,算法的收敛性一般是三次方的(在退化特征值的情况下,最差的情况也是二次收敛的)。这样快的收敛速度使这种算法很具有吸引力。

注意经过位移后,特征值不再按绝对值递增的顺序出现在对角线上。若需要的话,可调用程序 eigsrt(参见 11.1 节)。

正如前面提到的,一般矩阵的 QL 分解由一系列 Householder 变换来完成。对于三对角矩阵来说,用平面旋转 $\mathbf{P}_{pq}$ 更加有效。使用变换序列 $\mathbf{P}_{01},\mathbf{P}_{12},\ldots,\mathbf{P}_{n-2,n-1}$ 将元素 $a_{01},a_{12},\ldots,a_{n-2,n-1}$ 零化。对称地,次对角元素 $a_{10},a_{21},\ldots,a_{n-1,n-2}$ 也被零化。因此,$\mathbf{Q}_s$ 是这些平面旋转的乘积:

$$\mathbf{Q}_s^T=\mathbf{P}_1^{(s)}\cdot\mathbf{P}_2^{(s)}\cdots\mathbf{P}_{n-1}^{(s)} \tag{11.3.11}$$

其中 $\mathbf{P}_i$ 将 $a_{i-1,i}$ 零化。注意(11.3.11)式中不是 $\mathbf{Q}$ 而是 $\mathbf{Q}^T$,因为曾定义 $\mathbf{L}=\mathbf{Q}^T\cdot\mathbf{A}$。

11.3.3 具有隐含位移的 QL 算法

迄今为止,所描述的算法都是很成功的。然而,如果矩阵 $\mathbf{A}$ 的元素在数量级上相差很多,则从对角元中减去一个很大的 k_s 会导致小的特征值损失精度。但是,**隐含位移的 QL 算法**避免了这一问题。隐含位移的 QL 算法在数学上等价于原来的 QL 算法,只是计算中不需从 $\mathbf{A}$ 中将 $k_s\mathbf{1}$ 减去。

这种算法基于下述引理:如果 $\mathbf{A}$ 是一个对称非奇异矩阵,并且 $\mathbf{B}=\mathbf{Q}^T\cdot\mathbf{A}\cdot\mathbf{Q}$,其中 $\mathbf{Q}$ 是正交的,而 $\mathbf{B}$ 是具有正的非对角元素的三对角矩阵,则只要 $\mathbf{Q}^T$ 的最后一行给出后,$\mathbf{Q}$ 和 $\mathbf{B}$ 就能完全确定。证明:令 $\mathbf{q}_i^T$ 代表矩阵 $\mathbf{Q}^T$ 的第 i 个行向量,则 $\mathbf{q}_i$ 就是矩阵 $\mathbf{Q}$ 的第 i 个列向量。关系式 $\mathbf{B}\cdot\mathbf{Q}^T=\mathbf{Q}^T\cdot\mathbf{A}$ 可表示成

$$\begin{bmatrix} \beta_0 & \gamma_0 & & & & \\ \alpha_1 & \beta_1 & \gamma_1 & & & \\ & & \vdots & & & \\ & & & \alpha_{n-2} & \beta_{n-2} & \gamma_{n-2} \\ & & & & \alpha_{n-1} & \beta_{n-1} \end{bmatrix}\cdot\begin{bmatrix} \mathbf{q}_0^T \\ \mathbf{q}_1^T \\ \vdots \\ \mathbf{q}_{n-2}^T \\ \mathbf{q}_{n-1}^T \end{bmatrix}=\begin{bmatrix} \mathbf{q}_0^T \\ \mathbf{q}_1^T \\ \vdots \\ \mathbf{q}_{n-2}^T \\ \mathbf{q}_{n-1}^T \end{bmatrix}\cdot\mathbf{A} \tag{11.3.12}$$

上述矩阵方程的第 $n-1$ 行是

$$\alpha_{n-1}\mathbf{q}_{n-2}^T+\beta_{n-1}\mathbf{q}_{n-1}^T=\mathbf{q}_{n-1}^T\cdot\mathbf{A} \tag{11.3.13}$$

因为 $\mathbf{Q}$ 是正交的,

$$\mathbf{q}_{n-1}^T\cdot\mathbf{q}_m=\delta_{n-1,m} \tag{11.3.14}$$

如果将(11.3.13)式两边右乘 $\mathbf{q}_{n-1}$,则发现

$$\beta_{n-1}=\mathbf{q}_{n-1}^T\cdot\mathbf{A}\cdot\mathbf{q}_{n-1} \tag{11.3.15}$$

由于 $\mathbf{q}_{n-1}$ 已知,故上式是已知的。再由(11.3.13)式给出

$$\alpha_{n-1}\mathbf{q}_{n-2}^T=\mathbf{z}_{n-2}^T \tag{11.3.16}$$

其中

$$\mathbf{z}_{n-2}^T\equiv\mathbf{q}_{n-1}^T\cdot\mathbf{A}-\beta_{n-1}\mathbf{q}_{n-1}^T \tag{11.3.17}$$

是已知的。所以

$$\alpha_{n-1}^2 = \mathbf{z}_{n-2}^T \mathbf{z}_{n-2}, \tag{11.3.18}$$

或者

$$\alpha_{n-1} = |\mathbf{z}_{n-2}| \tag{11.3.19}$$

并且

$$\mathbf{q}_{n-2}^T = \mathbf{z}_{n-2}^T / \alpha_{n-1} \tag{11.3.20}$$

(其中假设α_{n-1}不为零。)类似地,可以推证出如果已知 $\mathbf{q}_{n-1}, \mathbf{q}_{n-2}, \ldots, \mathbf{q}_{n-j}$ 和脚标大于 $n-j$ 的α,β 和γ,就可以知道脚标为 $n-(j+1)$的有关量。

将上述引理用于实际计算,假设通过某种方法找到了一个三对角矩阵 $\overline{\mathbf{A}}_{s+1}$ 满足

$$\overline{\mathbf{A}}_{s+1} = \overline{\mathbf{Q}}_s^T \cdot \overline{\mathbf{A}}_s \cdot \overline{\mathbf{Q}}_s \tag{11.3.21}$$

其中 $\overline{\mathbf{Q}}_s^T$ 是正交的并与原来 QL 算法中的 $\mathbf{Q}_s^T$ 有相同的最后一行,则有 $\overline{\mathbf{Q}}_s = \mathbf{Q}_s$ 和 $\overline{\mathbf{A}}_{s+1} = \mathbf{A}_{s+1}$。

在原先的算法中,从(11.3.11)式我们看到 $\mathbf{Q}_s^T$ 的最后一行与 $\mathbf{P}_{n-1}^{(s)}$ 的最后一行相同。回忆一下,$\mathbf{P}_{n-1}^{(s)}$ 是一个用来将 $\mathbf{A}_s - k_s\mathbf{1}$ 的第$(n-2, n-1)$元素零化的平面旋转矩阵。从(11.1.1)式出发进行简单的计算给出它的参数

$$c = \frac{d_{n-1} - k_s}{\sqrt{e_{n-1}^2 + (d_{n-1} - k_s)^2}}, \quad s = \frac{-e_{n-2}}{\sqrt{e_{n-1}^2 + (d_{n-1} - k_s)^2}} \tag{11.3.22}$$

矩阵 $\mathbf{P}_{n-1}^{(s)} \cdot \mathbf{A}_s \cdot \mathbf{P}_{n-1}^{(s)T}$ 除去两个多余元素外是三对角的:

$$\begin{bmatrix} \cdots & & & & & \\ \cdot & \times & \times & \times & & \\ & & \times & \times & \times & \mathbf{x} \\ & & & \times & \times & \times \\ & & & \mathbf{x} & \times & \times \end{bmatrix} \tag{11.3.23}$$

现在必须使用一个最后一行为[0,0,...,0,1]的正交矩阵将上述矩阵化为三对角形式,这样 $\overline{\mathbf{Q}}_s^T$ 的最后一行就和 $\mathbf{P}_{n-1}^{(s)}$ 的最后一行相同。这可以通过一系列 Householder 或 Givens 变换来完成。对于(11.3.23)式这样特殊形式的矩阵来说,Givens 方法更好。在平面$(n-3, n-2)$中进行旋转使元素$(n-3, n-1)$零化(对称地,元素$(n1, n-3)$也将被代为零)。这样便得到一个除去多余元素$(n-4, n-2)$和$(n-2, n-4)$外都已三对角化了的矩阵,再由在平面$(n-4, n-3)$上的旋转零化这两个元素,依次类推。于是需要 $n-2$ 个 Givens 旋转的序列。结果得到

$$\mathbf{Q}_s^T = \overline{\mathbf{Q}}_s^T = \overline{\mathbf{P}}_1^{(s)} \cdot \overline{\mathbf{P}}_2^{(s)} \cdots \overline{\mathbf{P}}_{n-2}^{(s)} \cdot \mathbf{P}_{n-1}^{(s)} \tag{11.3.24}$$

其中 $\mathbf{P}$ 是 Givens 旋转,而 $\mathbf{P}_{n-1}$ 是原先算法中那种平面旋转。(11.3.21)式给出了下一迭代的 $\mathbf{A}_s$。注意,位移量 k_s 已经隐含在(11.3.22)式中了。

下面基于本节参考文献[2,3]的程序 tqli("三对角形 QL 隐含位移方法"),在实践中运行良好,对于前几个特征值的迭代次数可能是 4 或 5 次,但同时右下角的非对角元素也会相应地减少。后面的特征值就只需很少的计算量就可求出。每个特征值的平均迭代次数一般为 1.3 ~ 1.6 次。每次迭代的操作数为 $O(n)$,并有很高的有效系数($\sim 20n$)。因此,对角化所需的操作总数为 $\sim 20n \times (1.3 - 1.6)n \sim 30n^2$。如果同时需计算特征向量,则注释中标注出的语句需包含在程序中,而且将增加高达 $3n^3$ 的运算量。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::tqli(Vec_IO_DP &d, Vec_IO_DP &e, Mat_IO_DP &z)
```

为确定实对称三对角矩阵,或者事先由 11.2 节中程序 tred2 约化后的实对称矩阵的特征值和特征向量,本程序采用的是隐含位移的 QL 算法。输入变量 d[0..n-1]包括三对角矩阵的对角元素,输出时它返回特征值。向量e[0..n-1]给出三对角矩阵的次对角元素,e[0]任意值。输出时,e 的值已被破坏。如果只需求特征值,程序中有几行由注释标明的语句可以略去,如果需要求一个三对角矩阵的特征向量,则矩阵 z[0..n-1][0..n-1]作为单位矩阵输入;如果需要求一个由程序 tred2 已约化的矩阵的特征向量,则 z 由 tred2 输出的矩阵作为输入。不管是在哪种情况下,z 的第 k 列返回与 d[k]相对应的规范特征向量。

```
{
    int m,l,iter,i,k;
    DP s,r,p,g,f,dd,c,b;

    int n=d.size();
    for (i=1;i<n;i++) e[i-1]=e[i];                  便于重编号e中元素。
    e[n-1]=0.0;
    for (l=0;l<n;l++) {
        iter=0;
        do {
            for (m=l;m<n-1;m++) {                   寻找一个单一的小次对角元素
                dd=fabs(d[m])+fabs(d[m+1]);         用来分裂矩阵。
                if (fabs(e[m])+dd == dd) break;
            }
            if (m != l) {
                if (iter++ == 30) nrerror("Too many iterations in tqli");
                g=(d[l+1]-d[l])/(2.0*e[l]);         形成位移。
                r=pythag(g,1.0);
                g=d[m]-d[l]+e[l]/(g+SIGN(r,g));     这是 d_m-k_s。
                s=c=1.0;
                p=0.0;
                for (i=m-1;i>=l;i--) {              如同原来QL方法一样的平面转动,
                    f=s*e[i];                       随后Givens旋转，用以恢复三对角
                    b=c*e[i];                       形式。
                    e[i+1]=(r=pythag(f,g));
                    if (r == 0.0) {                 从下溢处恢复。
                        d[i+1] -= p;
                        e[m]=0.0;
                        break;
                    }
                    s=f/r;
                    c=g/r;
                    g=d[i+1]-p;
                    r=(d[i]-g)*s+2.0*c*b;
                    d[i+1]=g+(p=s*r);
                    g=c*r-b;
                    // Next loop can be omitted if eigenvectors not wanted
                    for (k=0;k<n;k++) {             形成特征向量.
                        f=z[k][i+1];
                        z[k][i+1]=s*z[k][i]+c*f;
                        z[k][i]=c*z[k][i]-s*f;
                    }
                }
                if (r == 0.0 && i >= 1) continue;
                d[l] -= p;
                e[l]=g;
                e[m]=0.0;
            }
        } while (m != l);
    }
}
```

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), pp. 331–335. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [3]

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §6.6.4. [4]

11.4 埃尔米特矩阵

在复数域中,实对称矩阵所对应的是埃尔米特(Hermite)矩阵,满足(11.0.4)式。雅可比变换可以用来寻找它的特征值和特征向量,如同用Householder变换将它约化为三对角形式以后,使用QL迭代一样。以前的程序

jacobi, tred2 和 tqli 在复数域中的形式与在实数域中极为类似，具体的工作程序，参见本节参考文献[1,2]。

一种替代的方法是应用本书中的程序，将埃尔米特问题化成实对称问题：如果 $\mathbf{C}=\mathbf{A}+i\mathbf{B}$ 是一个埃尔米特矩阵，则 $n\times n$ 复特征值问题为

$$(\mathbf{A}+i\mathbf{B})\cdot(\mathbf{u}+i\mathbf{v})=\lambda(\mathbf{u}+i\mathbf{v}) \tag{11.4.1}$$

等价于一个 $2n\times 2n$ 的实问题

$$\begin{bmatrix}\mathbf{A} & -\mathbf{B}\\ \mathbf{B} & \mathbf{A}\end{bmatrix}\cdot\begin{bmatrix}\mathbf{u}\\ \mathbf{v}\end{bmatrix}=\lambda\begin{bmatrix}\mathbf{u}\\ \mathbf{v}\end{bmatrix} \tag{11.4.2}$$

注意(11.4.2)式中的 $2n\times 2n$ 矩阵是对称的；如果 $\mathbf{C}$ 是埃尔米特距阵，则 $\mathbf{A}^T=\mathbf{A}$，并且 $\mathbf{B}^T=-\mathbf{B}$。

对应于一个给定的特征值 λ，向量

$$\begin{bmatrix}-\mathbf{v}\\ \mathbf{u}\end{bmatrix} \tag{11.4.3}$$

也是一个特征向量，它可以通过将(11.4.2)式中两个蕴含的矩阵方程写开来验证这一点。因而，如果 λ_0，$\lambda_1,\ldots,\lambda_{n-1}$ 是 $\mathbf{C}$ 的特征值，则扩展问题(11.4.2)的 $2n$ 个特征值是 $\lambda_0,\lambda_0,\lambda_1,\lambda_1,\ldots,\lambda_{n-1},\lambda_{n-1}$，也就是说，每个重复两次。特征向量是 $\mathbf{u}+i\mathbf{v}$ 和 $i(\mathbf{u}+i\mathbf{v})$ 的成对形式，即它们只相差一个非实质性的相位。因而求解扩展问题(11.4.2)，并从每一对中挑选一个特征值和一对特征向量，就得到了原来矩阵 $\mathbf{C}$ 的特征值和特征向量。

计算这个扩展的矩阵比原来的复矩阵运算需要多一倍的存储量。从原则上说，在计算时间上用复数算法比用扩展问题求解的效率要高一倍。实际上，除非是完全的实数计算，多数复数计算不会达到如此高的效率。

参考文献和深入阅读

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

11.5　将一般矩阵化为 Hessenberg 形式

在前几节中，所给出的对称矩阵的算法在实践中很令人满意。相反地，对于非对称矩阵就不可能找到同样令人满意的算法，这有两方面原因。首先，非对称矩阵的特征值可能对矩阵元素的微小变化很敏感；其次，矩阵本身可能是亏损的，因而找不到一套完备的特征向量。我们强调的这些困难是任何数值计算方法都不能解决的，它是某些非对称矩阵的内在性质。最大的希望就是，算法过程不要使这类问题的症状加剧。

舍入误差的出现只能使情况变得更糟。在有限的精度下，人们甚至于不能找到一种简便的方法来判定一个给定矩阵究竟是否亏损。所以当前的算法一般都是试着去发现一组完备的特征向量，至于结果怎样，得由用户自己判断。如果有两个特征向量近乎平行，则该矩阵有可能亏损。

除去介绍有关文献和搜集于本节参考文献[1,2]中的程序外，我们将把特征向量的问题放在一边，而只给出特征值的算法。如果只需要一小部分特征向量，则可参阅 11.7 节并考虑用逆迭代的方法来找寻它们。找寻非对称矩阵的全部特征向量的问题已超出了本书的讨论范围。

11.5.1　平衡

在进行某种算法时，特征值对舍入误差的敏感程度可以通过平衡的过程来减小。由数值过程导致的特征系统的误差一般是与矩阵的欧几里得(Euclid)范数成正比，也就是说，与所有元素的平方和之平方根成正比。平衡的思想就是，用相似变换将矩阵对应的行和列的范数变得相接近，从而在不改变特征值的前提下使矩阵的总范数减小。对称矩阵是一个已经平衡了的矩阵。

平衡是一个 N^2 次运算量的过程。所以下述程序 balanc 的运行时间永远不会多于找特征值所需时间百分之几。因而建议读者总要先将非对称矩阵配平。这样只会大大提高那些极度不平衡矩阵的特征值计算的精度。

正如本节参考文献[1]中讨论的,实际的算法基于 Osborne。它由对角矩阵 **D** 完成的一系列相似变换组成。为了避免在平衡过程中引入舍入误差,**D** 的元素限制为用于浮点算术的基底的适当幂次(例如,对大多数机器基数是 2,但对于 IBM 大型机体系结构的基数是 16)。输出是一个对于某种范数已经平衡的矩阵。这种范数是取所有矩阵元素的绝对值之和,它比用欧几里得范数更方便,而且同样有效:一种范数能有大的约化蕴含着另一种范数也有大的约化

注意,如果矩阵的一行或一列的非对角元素全部为零,则它的对角元素就是一个特征值。如果这个特征值恰巧是病态的(对于矩阵元素的微小变化很敏感),那么当用程序 hqr(参见 11.6 节)确定它时,将会带来相当大的误差。如果事先检查整个矩阵,就能确定这些孤立的特征值,并将相应的行和列从矩阵中去掉。读者应当考虑这样一种事先检查是否在应用中有用(对于对称矩阵,我们给出的程序在任何情况下都能准确地确定孤立的特征值)

程序 balanc 并不记录对原来矩阵所进行的相似变换本身,因为我们只关心寻找非对称矩阵的特征值而不是特征向量。如果想要跟踪这些变换的话,请参见本节参考文献[1~3]。

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

void NR::balanc(Mat_IO_DP &a)
给定一个矩阵 a[0..n-1][0..n-1],本程序用一个具有相同特征值的平衡矩阵取代它。而对称矩阵是已经平衡的,故
本程序对它不起作用。常数 RADIX 是机器的浮点基数
{
    const DP RADIX = numeric_limits<DP>::radix;
    int i,j,last=0;
    DP s,r,g,f,c,sqrdx;

    int n=a.nrows();
    sqrdx=RADIX*RADIX;
    while (last == 0) {
        last=1;
        for (i=0;i<n;i++) {                    计算行和列的范数。
            r=c=0.0;
            for (j=0;j<n;j++)
                if (j != i) {
                    c += fabs(a[j][i]);
                    r += fabs(a[i][j]);
                }
            if (c != 0.0 && r != 0.0) {        若两者都为非零。
                g=r/RADIX;
                f=1.0;
                s=c+r;
                while (c<g) {                  寻找机器基数的整数幂,使它接近配平
                    f *= RADIX;                的矩阵。
                    c += sqrdx;
                }
                g=r*RADIX;
                while (c>g) {
                    f /= RADIX;
                    c /= sqrdx;
                }
                if ((c+r)/f < 0.95*s) {
                    last=0;
                    g=1.0/f;
                    for (j=0;j<n;j++) a[i][j] *= g;      采用类似的变换。
                    for (j=0;j<n;j++) a[j][i] *= f;
                }
            }
        }
    }
}
```

11.5.2 约化成 Hessenberg 形式

找寻一般矩阵的特征系统的方法与对称情形是平行的。首先,将矩阵约化成一种简单的形式,然后对这种简单的形式进行某种迭代。这里所用的简单形式为 **Hessenberg 形式**。一个上 Hessenberg 矩阵在其对角线

下，除去第一个次对角线上元素之外全部为零。例如，对于 6×6 的情形，非零元素是

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \end{bmatrix}$$

至此，应该能够一眼看出，这样一种形式可以通过一系列 Householder 变换得到，每一变换将矩阵一列中的相应元素化为零。Householder 方法的确是一种可接受的方法。然而，一种替换的方法是，类似高斯选主元的消去法的过程。选用这种消去法，是因为它比 Householder 方法效率高一倍，而且我们也想向读者介绍这种方法。可以构造出这样一种矩阵，它对于 Householder 约化的正交变换是稳定的，而对于消去法是不稳定的，但这种矩阵在实践中是极为罕见的。

直接高斯消去法不是一种矩阵的相似变换。因此，实际所用的消去过程有点不同，其步骤记为 $r=1,2,\cdots\cdots,N-2$。在第 r 步开始之前，原来的矩阵 $\mathbf{A}\equiv\mathbf{A}_1$ 已经变成 $\mathbf{A}_r$，它的前 $r-1$ 行和 $r-1$ 列已经化为上 Hessenberg 形式。第 r 步由下列操作构成：

- 找出第 $r-1$ 列中对角元以下的元素中最大的一个。如果它为零则跳过以下两项操作，这一步就完成了。否则就假设最大元素在第 r' 行。
- 将第 r' 行和第 r 行相交换。这是一个选主元过程。为了使这种置换成为相似变换，同样交换第 r' 列和第 r 列。
- 对 $i=r+1,r+2,\ldots,N-1$，计算乘数

$$n_{ir}\equiv\frac{a_{i,r-1}}{a_{r,r-1}}$$

从第 i 行中减去 n_{ir} 乘以第 r 行。为了使这种消去成为相似变换，同样将 n_{ir} 乘以第 i 列加到第 r 列上去

一共需 $N-2$ 个这样的步骤。

如果矩阵元素的大小数量级相差很多，则应试着重新安排矩阵，以使最大元素位于左上角。这就可以减小舍入误差，因为整个过程是从左向右进行的。

因为我们只关心特征值，程序 elmhes 不跟踪记录所进行的相似变换。对于较大的 N，运算量大约为 $5N^3/6$。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::elmhes(Mat_IO_DP &a)
用消去法将矩阵约化成 Hessenberg 形式。实的非对称矩阵 a[0..n-1][0..n-1]由具有相同特征值的上 Hessenberg 矩阵
所代替。建议(但不必须)运行本程序前先调用 balanc。输出时，Hessenberg 矩阵在 a[i][j]中(i≤j+1)。具有 i>j+1 的
元素应看做为零，但返回一些随机数。
{
    int i,j,m;
    DP y,x;

    int n=a.nrows();
    for (m=1;m<n-1;m++) {                m 在正文中称为 r。
        x=0.0;
        i=m;
        for (j=m;j<n;j++) {              寻找主元。
            if (fabs(a[j][m-1]) > fabs(x)) {
                x=a[j][m-1];
                i=j;
            }
        }
        if (i != m) {                    交换行和列。
            for (j=m-1;j<n;j++) SWAP(a[i][j],a[m][j]);
            for (j=0;j<n;j++) SWAP(a[j][i],a[j][m]);
```

```
            }
            if (x != 0.0) {                    执行消去法。
                for (i=m+1;i<n;i++) {
                    if ((y=a[i][m-1]) != 0.0) {
                        y /= x;
                        a[i][m-1]=y;
                        for (j=m;j<n;j++) a[i][j] -= y*a[m][j];
                        for (j=0;j<n;j++) a[j][m] += y*a[j][i];
                    }
                }
            }
        }
    }
```

参考文献和深入阅读

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §6.5.4. [3]

11.6 实 Hessenberg 矩阵的 QR 算法

回忆一下带有位移的 QR 算法的下列关系式:

$$\mathbf{Q}_s \cdot (\mathbf{A}_s - k_s\mathbf{1}) = \mathbf{R}_s \tag{11.6.1}$$

其中 $\mathbf{Q}$ 是正交矩阵,而 $\mathbf{R}$ 是上三角矩阵,并且有

$$\begin{aligned}\mathbf{A}_{s+1} &= \mathbf{R}_s \cdot \mathbf{Q}_s^T + k_s\mathbf{1} \\ &= \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T\end{aligned} \tag{11.6.2}$$

QR 变换保持原来矩阵 $\mathbf{A} \equiv \mathbf{A}_1$ 的上 Hessenberg 形式,并且每次迭代的运算量是 $O(n^2)$,而对于一般矩阵来说,却是 $O(n^3)$。当 $s \to \infty$,$\mathbf{A}_s$ 收敛于这样一种形式,在这种形式中特征值或者在对角线上被分离出来,或者是对角线上一个 2×2 子矩阵的特征值。

正如在 11.3 节中指出的,位移是快速收敛的关键。这里一个关键的区别是非对称实矩阵可能有复特征值。这就意味着位移量 k_s 可能是复数,显然需要使用复数运算。

尽管如此,可以通过一个技巧来避免复数运算。这一技巧依赖于一个类似于 11.3 节中用于隐含位移那样的引理。现在需要的引理可陈述为:如果 $\mathbf{B}$ 是一个非奇异矩阵,满足

$$\mathbf{B} \cdot \mathbf{Q} = \mathbf{Q} \cdot \mathbf{H} \tag{11.6.3}$$

其中 $\mathbf{Q}$ 是正交矩阵,而 $\mathbf{H}$ 是上 Hessenberg 形式的,则 $\mathbf{Q}$ 和 $\mathbf{H}$ 由 $\mathbf{Q}$ 的第零列完全确定(如果 $\mathbf{H}$ 具有正的下次对角元素,则这种确定是惟一的)。这一引理可以用类似于 11.3 节中对三对角矩阵所做的推证来证明。

这一引理在实践中的应用是采用两步 QR 算法,或者取两个实的位移量 k_s 和 k_{s+1},或者是一对复共轭的值 k_s 和 $k_{s+1} = k_s^*$。这就给出了一个实矩阵 $\mathbf{A}_{s+2}$:

$$\mathbf{A}_{s+2} = \mathbf{Q}_{s+1} \cdot \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \cdot \mathbf{Q}_{s+1}^T \tag{11.6.4}$$

其中的 $\mathbf{Q}$ 由下列关系确定:

$$\mathbf{A}_s - k_s\mathbf{1} = \mathbf{Q}_s^T \cdot \mathbf{R}_s \tag{11.6.5}$$

$$\mathbf{A}_{s+1} = \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \tag{11.6.6}$$

$$\mathbf{A}_{s+1} - k_{s+1}\mathbf{1} = \mathbf{Q}_{s+1}^T \cdot \mathbf{R}_{s+1} \tag{11.6.7}$$

应用(11.6.6)式、(11.6.7)式可重写成

$$\mathbf{A}_s - k_{s+1}\mathbf{1} = \mathbf{Q}_s^T \cdot \mathbf{Q}_{s+1}^T \cdot \mathbf{R}_{s+1} \cdot \mathbf{Q}_s \tag{11.6.8}$$

于是，如果我们定义

$$\mathbf{M} = (\mathbf{A}_s - k_{s+1}\mathbf{1}) \cdot (\mathbf{A}_s - k_s\mathbf{1}) \tag{11.6.9}$$

则(11.6.5)式和(11.6.8)式给出

$$\mathbf{R} = \mathbf{Q} \cdot \mathbf{M} \tag{11.6.10}$$

其中

$$\mathbf{Q} = \mathbf{Q}_{s+1} \cdot \mathbf{Q}_s \tag{11.6.11}$$

$$\mathbf{R} = \mathbf{R}_{s+1} \cdot \mathbf{R}_s \tag{11.6.12}$$

(11.6.4)式可重写成

$$\mathbf{A}_s \cdot \mathbf{Q}^T = \mathbf{Q}^T \cdot \mathbf{A}_{s+2} \tag{11.6.13}$$

于是，假设我们能通过某种方法找到一个上 Hessenberg 矩阵 $\mathbf{H}$ 使得

$$\mathbf{A}_s \cdot \overline{\mathbf{Q}}^T = \overline{\mathbf{Q}}^T \cdot \mathbf{H} \tag{11.6.14}$$

成立，其中 $\overline{\mathbf{Q}}$ 是正交矩阵。如果 $\overline{\mathbf{Q}}^T$ 与 $\mathbf{Q}^T$ 有相同的第零列(也就是 $\overline{\mathbf{Q}}$ 与 $\mathbf{Q}$ 有相同的第零行)，则 $\overline{\mathbf{Q}} = \mathbf{Q}$ 并且 $\mathbf{A}_{s+2} = \mathbf{H}$。

$\mathbf{Q}$ 的第零行可以按如下方法找到。(11.6.10)式表明 $\mathbf{Q}$ 是将实矩阵 $\mathbf{M}$ 三角化的正交矩阵。任何一个矩阵可以通过左乘一系列 Householder 矩阵而三角化。这一系列 Householder 矩阵是 $\mathbf{P}_1$(作用在第零列上)，$\mathbf{P}_2$(作用在第一列上)，...，$\mathbf{P}_{n-1}$。于是 $\mathbf{Q} = \mathbf{P}_{n-1} \cdots \mathbf{P}_2 \cdot \mathbf{P}_1$，而 $\mathbf{Q}$ 的第零行就是 $\mathbf{P}_1$ 的第零行，因为 $\mathbf{P}_i$ 的左上角是 $(i-1) \times (i-1)$ 的单位矩阵。我们现在必须去找满足(11.6.14)式并且与 $\mathbf{P}_1$ 有同样第零行的 $\overline{\mathbf{Q}}$。

Householder 矩阵 $\mathbf{P}_1$ 是由 $\mathbf{M}$ 的第零列决定的。因为 $\mathbf{A}_s$ 是上 Hessenberg 形式的，(11.6.9)式表明 $\mathbf{M}$ 的第零列有形式 $[p_1, q_1, r_1, 0, \ldots, 0]^T$，其中

$$\begin{aligned} p_1 &= a_{00}^2 - a_{00}(k_s + k_{s+1}) + k_s k_{s+1} + a_{01}a_{10} \\ q_1 &= a_{10}(a_{00} + a_{11} - k_s - k_{s+1}) \\ r_1 &= a_{10}a_{21} \end{aligned} \tag{11.6.15}$$

因此

$$\mathbf{P}_1 = 1 - 2\mathbf{w}_1 \cdot \mathbf{w}_1^T \tag{11.6.16}$$

其中 $\mathbf{w}_1$ 只有前三个元素非零，参看(11.2.5)式。于是矩阵 $\mathbf{P}_1 \cdot \mathbf{A}_s \cdot \mathbf{P}_1^T$ 除去三个多余元素之外是上 Hessenberg 形式：

$$\mathbf{P}_1 \cdot \mathbf{A}_1 \cdot \mathbf{P}_1^T = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ \mathbf{x} & \times & \times & \times & \times & \times & \times \\ \mathbf{x} & \mathbf{x} & \times & \times & \times & \times & \times \\ & & & \times & \times & \times & \times \\ & & & & \times & \times & \times \\ & & & & & \times & \times \end{bmatrix} \tag{11.6.17}$$

这个矩阵可以通过一系列 Householder 相似变换重新变成上 Hessenberg 形式，而不影响它的第零行。这样的 Householder 矩阵中的第一个 $\mathbf{P}_2$，作于在第 0 列第 1、第 2 和第 3 元素上，并将第 2 和第 3 元素零化。这就产生了一个和(11.6.17)式形式相同的矩阵，只是三个多余元素向后推了一列：

$$\begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times & \times \\ & \mathbf{x} & \times & \times & \times & \times & \times \\ & \mathbf{x} & \mathbf{x} & \times & \times & \times & \times \\ & & & & \times & \times & \times \\ & & & & & \times & \times \end{bmatrix} \tag{11.6.18}$$

这样继续直到 $\mathbf{P}_{n-1}$,我们看到每一步的 Householder 矩阵 $\mathbf{P}_r$ 有一个只有第 $r-1$, r 和 $r+1$ 元素为非零的向量 $\mathbf{w}_r$。这些元素是由当前矩阵的第 $(r-2)$ 列的第 $r-1$, r 和 $r+1$ 元素决定的。注意 $\mathbf{P}_1$ 与 $\mathbf{P}_2,\ldots,\mathbf{P}_{n-1}$ 有相同的结构。

结果是

$$\mathbf{P}_{n-1}\cdots\mathbf{P}_2\cdot\mathbf{P}_1\cdot\mathbf{A}_s\cdot\mathbf{P}_1^T\cdot\mathbf{P}_2^T\cdots\mathbf{P}_{n-1}^T=\mathbf{H} \tag{11.6.19}$$

其中 $\mathbf{H}$ 是上 Hessenberg 形式。于是

$$\overline{\mathbf{Q}}=\mathbf{Q}=\mathbf{P}_{n-1}\cdots\mathbf{P}_2\cdot\mathbf{P}_1 \tag{11.6.20}$$

和

$$\mathbf{A}_{s+2}=\mathbf{H} \tag{11.6.21}$$

每一步的原点位移都取为当前矩阵 $\mathbf{A}_s$ 的右下角处 2×2 矩阵的特征值。由此给出

$$\begin{aligned}k_s+k_{s+2}&=a_{n-2,n-2}+a_{n-1,n-1}\\ k_sk_{s+1}&=a_{n-2,n-2}a_{n-1,n-1}-a_{n-2,n-1}a_{n-1,n-2}\end{aligned} \tag{11.6.22}$$

将(11.6.22)式代入(11.6.15)式,得到

$$\begin{aligned}p_1&=a_{10}\{[(a_{n-1,n-1}-a_{00})(a_{n-2,n-2}-a_{00})-a_{n-2,n-1}a_{n-1,n-2}]/a_{10}+a_{01}\}\\ q_1&=a_{10}[a_{11}-a_{00}-(a_{n-1,n-1}-a_{00})-(a_{n-2,n-2}-a_{00})]\\ r_1&=a_{10}a_{21}\end{aligned} \tag{11.6.23}$$

我们已经明智地归类某些项,以减小当存在小的非对角元时可能带来的舍入误差。由于 Householder 矩阵只和元素间的比例有关,可以略去(11.6.23)式中的因子 a_{10}。

总之,为完成两步 QR 迭代,我们构造一系列的 Householder 矩阵 $\mathbf{P}_r$, $r=1,\ldots,n-1$。对于 $\mathbf{P}_1$ 我们使用(11.6.23)式给出的 p_1, q_1 和 r_1。对于剩下的矩阵,p_r, q_r 和 r_r 由当前矩阵的 $(r-1,r-2)$, $(r,r-2)$ 和 $(r+1,r-2)$ 元素确定。可以通过将 Householder 矩阵 $2\mathbf{w}\cdot\mathbf{w}^T$ 部分的非零元素写成如下形式,来减少算术操作的次数:

$$2\mathbf{w}\cdot\mathbf{w}^T=\begin{bmatrix}(p\pm s)/(\pm s)\\ q/(\pm s)\\ r/(\pm s)\end{bmatrix}\cdot[1\quad q/(p\pm s)\quad r/(p\pm s)] \tag{11.6.24}$$

其中

$$s^2=p^2+q^2+r^2 \tag{11.6.25}$$

简单地将每一元素除以一个归一化因子,参见 11.2 节中的公式。

按这种方法进行,收敛通常是很快的。对于一个特征值有两种可能的方式可以结束所进行的迭代。首先,如果 $a_{n-1,n-2}$ 变成是"可忽略的",则 $a_{n-1,n-1}$ 是一个特征值。于是可以删去矩阵的第 $n-1$ 行和第 $n-1$ 列并继续寻找下一个特征值。第二种情况,$a_{n-2,n-3}$ 可能变得可忽略。这时右下角的 2×2 矩阵的特征值就是原来矩阵的特征值。删去第 $n-1$ 和第 $n-2$ 行和列,并继续进行下面的计算。

对特征值收敛性的检验与可忽略的次对角元素的检验相结合。这种可忽略的次对角元素使矩阵能分裂成子矩阵。寻找最大的 i 值使得 $a_{i,i-1}$ 可忽略,如果 $i=n-1$,我们就找到了一个特征值;如果 $i=n-2$,就找到了两个特征值。否则,就继续进行对于子矩阵从第 i 行到第 $n-1$ 行的迭代(如果没有小的次对角元素,i 就设为 0)

确定 i 值之后,检查子矩阵从第 i 行到第 $n-1$ 行,是否有两个连续次对角元素的乘积足够小,以至于能针对一个更小的子矩阵进行计算。例如,在第 m 行,从 $m=n-3$ 开始一直下降到 $i+1$,用(11.6.23)式计算 p、q 和 r,但要将公式中的 0 用 m 代替,1 用 $m+1$ 代替。如果这些是一个双重步 QR 中特殊的"第一"Householder 矩阵元素,使用这一 Householder 矩阵将会导致 $(m+1,m-1)$, $(m+2,m-1)$ 和 $(m+2,m)$ 位置上的元素不为零。要求这些元素中的前两个元素与相邻的对角元素 $a_{m-1,m-1}$、a_{mm} 和 $a_{m+1,m+1}$ 比较起来较小。一个令人满意的近似标准是

$$|a_{m,m-1}|(|q|+|r|) \ll |p|(|a_{m+1,m+1}|+|a_{mm}|+|a_{m-1,m-1}|) \tag{11.6.26}$$

上述过程很少是不收敛的。经验告诉我们，如果一个双重步 QR 迭代中的位移与矩阵范数是同一量级的，那么通常收敛很快。相应地，如果 10 次迭代还没有确定出一个特征值，下次迭代通常的位移要变换成由下式定义的新的位移：

$$\begin{aligned} k_s + k_{s+1} &= 1.5 \times (|a_{n-1,n-2}| + |a_{n-2,n-3}|) \\ k_s k_{s+1} &= (|a_{n-1,n-2}| + |a_{n-2,n-3}|)^2 \end{aligned} \tag{11.6.27}$$

其中因子 1.5 可以任意挑选，以减少无效的移动。在 20 次不成功迭代后，这一策略重复使用。如果 30 次迭代不成功，则程序宣告失败。

这里描述的 QR 算法的运算量合计为每次迭代 $\sim 5k^2$，其中 k 是当前矩阵的阶数。每个特征值的平均迭代数为 ~ 1.8，故所有特征值的总运算量为 $\sim 3n^3$。这一估计忽略了矩阵分裂或稀疏性可能带来的效率。

下列程序 hqr 基于上述算法，源于本节参考文献[1,2]。

```
#include <cmath>
#include <complex>
#include "nr.h"
using namespace std;

void NR::hqr(Mat_IO_DP &a, Vec_O_CPLX_DP &wri)
求上 Hessenberg 矩阵 a[0..n-1][0..n-1]的所有特征值。输入 a 可以是 11.5 节中 elmhes 的输出值，输出时 a 的原值已
被破坏。复特征值返回到 wri[0..n-1]
{
    int nn,m,l,k,j,its,i,mmin;
    DP z,y,x,w,v,u,t,s,r,q,p,anorm;

    int n=a.nrows();
    anorm=0.0;                          计算矩阵范数，有可能将它用来定位小的、
    for (i=0;i<n;i++)                   单一的次对角元素。
        for (j=MAX(i-1,0);j<n;j++)
            anorm += fabs(a[i][j]);
    nn=n-1;
    t=0.0;                              只有一个额外的位移才发生变化。
    while (nn >= 0) {                   开始寻找下一个特征值。
        its=0;
        do {
            for (l=nn;l>0;l--) {        开始迭代：寻找小的、单一的次对角元素。
                s=fabs(a[l-1][l-1])+fabs(a[l][l]);
                if (s == 0.0) s=anorm;
                if (fabs(a[l][l-1]) + s == s) {
                    a[l][l-1] = 0.0;
                    break;
                }
            }
            x=a[nn][nn];
            if (l == nn) {              找到一个根。
                wri[nn--]=x+t;
            } else {
                y=a[nn-1][nn-1];
                w=a[nn][nn-1]*a[nn-1][nn];
                if (l == nn-1) {        找到两个根……
                    p=0.5*(y-x);
                    q=p*p+w;
                    z=sqrt(fabs(q));
                    x += t;
                    if (q >= 0.0) {     ……一对实数根。
                        z=p+SIGN(z,p);
                        wri[nn-1]=wri[nn]=x+z;
                        if (z != 0.0) wri[nn]=x-w/z;
                    } else {            ……一对复数根。
                        wri[nn]=complex<DP>(x+p,z);
                        wri[nn-1]=conj(wri[nn]);
                    }
                    nn -= 2;
                } else {                没有找到根，继续迭代。
                    if (its == 30) nrerror("Too many iterations in hqr");
                    if (its == 10 || its == 20) {       形成特殊的位移。
```

```
        t += x;
        for (i=0;i<nn+1;i++) a[i][i] -= x;
        s=fabs(a[nn][nn-1])+fabs(a[nn-1][nn-2]);
        y=x=0.75*s;
        w = -0.4375*s*s;
    }
    ++its;
    for (m=nn-2;m>=l;m--) {                 形成位移并寻找2个连续的、
        z=a[m][m];                          小的次对角元素。
        r=x-z;
        s=y-z;
        p=(r*s-w)/a[m+1][m]+a[m][m+1];      (11.6.23)式。
        q=a[m+1][m+1]-z-r-s;
        r=a[m+2][m+1];
        s=fabs(p)+fabs(q)+fabs(r);          为防止上溢出或下溢出而改变尺度。
        p /= s;
        q /= s;
        r /= s;
        if (m == l) break;
        u=fabs(a[m][m-1])*(fabs(q)+fabs(r));
        v=fabs(p)*(fabs(a[m-1][m-1])+fabs(z)+fabs(a[m+1][m+1]));
        if (u+v == v) break;                (11.6.26)式。
    }
    for (i=m;i<nn-1;i++) {
        a[i+2][i]=0.0;
        if (i != m) a[i+2][i-1]=0.0;
    }
    for (k=m;k<nn;k++) {
    对从第l行到第nn行及从第m列到第nn列施行两步QR。
        if (k != m) {
            p=a[k][k-1];                    开始设置Householder向量。
            q=a[k+1][k-1];
            r=0.0;
            if (k+1 != nn) r=a[k+2][k-1];
            if ((x=fabs(p)+fabs(q)+fabs(r)) != 0.0) {
                p /= x;                     为防止上溢出或下溢而
                q /= x;                     改变尺度。
                r /= x;
            }
        }
        if ((s=SIGN(sqrt(p*p+q*q+r*r),p)) != 0.0) {
            if (k == m) {
                if (l != m)
                a[k][k-1] = -a[k][k-1];
            } else
                a[k][k-1] = -s*x;
            p += s;                         (11.6.24)式。
            x=p/s;
            y=q/s;
            z=r/s;
            q /= p;
            r /= p;
            for (j=k;j<nn+1;j++) {          行调整。
                p=a[k][j]+q*a[k+1][j];
                if (k+1 != nn) {
                    p += r*a[k+2][j];
                    a[k+2][j] -= p*z;
                }
                a[k+1][j] -= p*y;
                a[k][j] -= p*x;
            }
            mmin = nn < k+3 ? nn : k+3;
            for (i=l;i<mmin+1;i++) {        列调整。
                p=x*a[i][k]+y*a[i][k+1];
                if (k != (nn)) {
                    p += z*a[i][k+2];
                    a[i][k+2] -= p*r;
                }
                a[i][k+1] -= p*q;
                a[i][k] -= p;
```

```
                    }
                }
            }
        }
    }
    } while (l+1 < nn);
  }
}
```

参考文献和深入阅读

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [1]

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §7.5.

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

11.7 用逆迭代法改进特征值并求解特征向量

逆迭代的基本思想很简单。令 $\mathbf{y}$ 是如下线性系统的解：

$$(\mathbf{A}-\tau\mathbf{1})\cdot\mathbf{y}=\mathbf{b} \tag{11.7.1}$$

其中 $\mathbf{b}$ 是一个随机向量，而 τ 是靠近 $\mathbf{A}$ 的某一特征值 λ，则 $\mathbf{y}$ 接近 λ 所对应的特征向量。这一过程可迭代进行：用 $\mathbf{y}$ 代替 $\mathbf{b}$ 并解出新的 $\mathbf{y}$，新的 $\mathbf{y}$ 更接近真实的特征向量。

可以通过将 $\mathbf{y}$ 和 $\mathbf{b}$ 展开成 $\mathbf{A}$ 的特征向量 $\mathbf{x}_j$ 的线性组合，来查看逆迭代法的工作原理：

$$\mathbf{y}=\sum_j \alpha_j\mathbf{x}_j \qquad \mathbf{b}=\sum_j \beta_j\mathbf{x}_j \tag{11.7.2}$$

于是(11.7.1)式给出

$$\sum_j \alpha_j(\lambda_j-\tau)\mathbf{x}_j=\sum_j \beta_j\mathbf{x}_j \tag{11.7.3}$$

从而有

$$\alpha_j=\frac{\beta_j}{\lambda_j-\tau} \tag{11.7.4}$$

和

$$\mathbf{y}=\sum_j \frac{\beta_j\mathbf{x}_j}{\lambda_j-\tau} \tag{11.7.5}$$

如果 τ 接近 λ_n，则只要 β_n 不是意外地太小，$\mathbf{y}$ 将近似于某种归一化的 $\mathbf{x}_n$。进而，这一过程的每一步迭代都给出(11.7.5)式中的分母 $\lambda_j-\tau$ 的一个幂次。所以，对于分得很开的特征值其收敛很快。

假设在第 k 步迭代，解等式

$$(\mathbf{A}-\tau_k\mathbf{1})\cdot\mathbf{y}=\mathbf{b}_k \tag{11.7.6}$$

其中 $\mathbf{b}_k$ 和 τ_k 是我们感兴趣的特征向量和特征值的当前估计(例如，$\mathbf{x}_n$ 和 λ_n)。归一化 $\mathbf{b}_k$，使 $\mathbf{b}_k\cdot\mathbf{b}_k=1$。精确的特征向量和特征值满足

$$\mathbf{A}\cdot\mathbf{x}_n=\lambda_n\mathbf{x}_n \tag{11.7.7}$$

于是

$$(\mathbf{A}-\tau_k\mathbf{1})\cdot\mathbf{x}_n=(\lambda_n-\tau_k)\mathbf{x}_n \tag{11.7.8}$$

因为(11.7.6)式中的 $\mathbf{y}$ 是 $\mathbf{x}_n$ 的进一步近似，所以将其归一化并令

$$\mathbf{b}_{k+1} = \frac{\mathbf{y}}{|\mathbf{y}|} \tag{11.7.9}$$

通过将改进的猜测值 $\mathbf{y}$ 代入(11.7.8)式,得到特征值的一个改进的估计值。根据(11.7.6)式,左边是 $\mathbf{b}_k$,将 λ_n 称为新值 τ_{k+1},发现

$$\tau_{k+1} = \tau_k + \frac{1}{\mathbf{b}_k \cdot \mathbf{y}} \tag{11.7.10}$$

上述公式看起来很简单,但实际计算中可能很麻烦。首先,一个需要重新考虑的问题就是何时使用逆迭代。大多数工作量是解线性系统(11.7.6)。所以一个可能的策略是,首先将矩阵 $\mathbf{A}$ 约化成一个简单的形式,使得(11.7.6)式更容易求解。显然,这样的形式对于对称矩阵来说是三对角形式,而对于非对称矩阵来说是 Hessenberg 形式。然后,使用逆迭代来产生所有的特征向量。这种方法对于对称矩阵的运算量为 $O(N^3)$,它远不及前面提到的 QL 方法有效。实际上,一旦要求计算四分之一以上数量的特征向量,即使是最好的逆迭代程序也不及 QL 方法有效。所以只在当已知有较好的特征值,并只需求其中几个选出的特征向量时,才使用逆迭代方法。

读者可以自己写一个简单的逆代迭代程序,使用 LU 分解来解(11.7.6)式。可以自己决定是使用在第 2 章给出的一般 LU 算法,还是利用三对角或 Hessenberg 矩阵的特殊形式。注意,因为(11.7.6)式的线性系统是近乎奇异的,必须小心使用 2.3 节中的 LU 分解,它用一个非常小的数来代替零主元。

在本书中,没有给出一个一般的逆代程序。因为要考虑所有可能出现的情况是相当复杂的。在本节参考文献[1,2]中给出了程序。如果使用这些程序或者自己写程序,下面的说明是很有用的:

在开始的时候,给出一个代表特征值 λ_n 的初值 τ_0,选择一个归一化的随机向量 $\mathbf{b}_0$ 作为对特征向量 $\mathbf{x}_n$ 的初始猜测,并求解(11.7.6)式。新的向量 $\mathbf{y}$ 比 $\mathbf{b}_0$ 大一个"增长因子" $|\mathbf{y}|$,它理所当然应该大些。同样,特征值的变化,根据(11.7.10)式本质上应是 $1/|\mathbf{y}|$,它应该小一些。下列情况可能出现:

- 如果开始时增长因子太小,则可以肯定选择了一个"坏"的随机向量。这不仅可能是由于(11.7.5)式中的 β_n 很小,同样,也可能由于是亏损矩阵情形,这时(11.7.5)式根本不能使用(详细情况参见本节参考文献[1,3])。应返回到起始位置并选择一个新的初始向量。
- 变化量 $|\mathbf{b}_1 - \mathbf{b}_0|$ 应小于某容差限 ϵ。我们能用它作为停止迭代的标准。一直迭代到满足此标准,最多要进行 5 ~ 10 次迭代。
- 经过几次迭代后,如果 $|\mathbf{b}_{k+1} - \mathbf{b}_k|$ 减小得不够快,应根据(11.7.10)式或试着修正特征值。如果在机器精度下 $\tau_{k+1} = \tau_k$,就不要再去修正特征向量而应放弃迭代。否则就用新的特征值开始另一轮迭代。

不是每一步都修正特征值,其原因是用 LU 分解线性系统(11.7.6)时,如果 τ_k 是固定的则可以节省分解。在每次更新 $\mathbf{b}_k$ 时,只需做回代。对于具有固定 τ_k 的迭代次数是以下两者的折中,或者以二次收敛但由于每次改进 τ_k 带来 $O(N^3)$ 的运算量,或者以线性收敛但由于保持固定的 τ_k 所需运算量为 $O(N^2)$。如果已经应用本章前面给出的某个程序确定了特征值,则它可能修正到机器的精度,于是就不必更新它。

有两种病态情况可能会在逆迭代过程中出现。一种是重根或离得很近的根。这是对称矩阵情况中常见的问题。逆迭代对于一个给定的初始猜测 τ_0 只能找到一个特征向量。一个好的策略是,扰动 τ_0 的最后几个有效位并再次迭代。这样,常常能得到一个线性无关的特征向量。一般来说,必须采取特殊的步骤来保证线性无关特征向量的正交性,鉴于雅可比和 QL 算法,即使在多重特征值的情况下,也会自动产生正交的特征向量。

第二个问题,特别对于非对称矩阵来说,是关于亏损的问题。除非做出一个"好"的初始估计,不然增长因子会很小。更有甚者,迭代对问题没有改进。在这种情况下,解决的办法是选择随机的初始向量,解一次(11.7.6)式,并只要任何向量给出一个可接受大小的增长因子,就停止迭代。通常只需进行几次试验。

对于非对称矩阵的一个进一步的复杂情况是实矩阵可能有一对复共轭的特征值。这时将不得不使用复数运算从(11.7.6)式解出复向量。对于任何中型(或大型)非对称矩阵,建议避免使用逆迭代法,而选用包括复数运算的 QR 方法。读者可以从本节参考文献[1,2]中找到这类程序。

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America).

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), p. 418. [1]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [2]

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), p. 375. [3]

第 12 章　快速傅里叶变换

12.0　引言

有一大类重要的计算问题归入一般的"傅里叶(Fourier)变换法"或"谱方法"范畴之内。对于其中某些计算问题来说,傅里叶变换只是一种完成数据的某种普通操作的有效计算工具。而对于其他一些情况,人们关心的问题却是傅里叶变换(或有关的"功率谱")其自身内在的含意是什么。这两种问题具有一种共同的方法论。

主要由于历史原因,使傅里叶方法和谱方法的文献一直与"古典的"数值分析的文献没有什么联系。但如今,应当将它们联系起来。傅里叶方法在研究中是极为平凡的方法,其并非高深莫测。同时,许多计算机用户在此领域内的经验比其他领域,例如微分方程或数值积分领域的经验相对地要少些。所以,对解析结果的分析要做得更完整些。数值算法的内容将在 12.2 节开始讨论,而傅里叶变换方法的各种应用将在第 13 章中讨论。

一个物理过程既可以在**时域**内,通过把物理量 h 作为时间 t 的函数 $h(t)$ 来描述,也可以在**频域**内,通过将振幅 H(通常是一个包含相位的复数)作为频率 f 的函数 $H(f)$ 来描述,其中 $-\infty<f<\infty$。在许多情况下,把 $h(t)$ 和 $H(f)$ 考虑为同一函数的两种不同的表达方式是很有用的。根据傅里叶变换方程便可实现这两个表达之间的相互转变,

$$
\begin{aligned}
H(f) &= \int_{-\infty}^{\infty} h(t)e^{2\pi i f t}dt \\
h(t) &= \int_{-\infty}^{\infty} H(f)e^{-2\pi i f t}df
\end{aligned}
\tag{12.0.1}
$$

如果 t 以秒为单位,则(12.0.1)式中 f 是以周期每秒或赫兹(Hertz)为单位(赫兹是频率单位)。当然,方程也可采用其他单位。如果 h 是位移 x(以米为单位)的函数,H 则为反波长(周期数/米)的函数,等等。若读者是一位训练有素的物理学家或数学家,就可能更习惯于使用**角频率** ω(单位为弧度/秒),ω 和 f、$H(\omega)$ 和 $H(f)$ 之间的关系为

$$
\omega \equiv 2\pi f \qquad H(\omega) \equiv [H(f)]_{f=\omega/2\pi} \tag{12.0.2}
$$

则(12.0.1)式可写成如下形式:

$$
\begin{aligned}
H(\omega) &= \int_{-\infty}^{\infty} h(t)e^{i\omega t}dt \\
h(t) &= \frac{1}{2\pi}\int_{-\infty}^{\infty} H(\omega)e^{-i\omega t}d\omega
\end{aligned}
\tag{12.0.3}
$$

我们开始时习惯使用单位 ω,可后来有所改变。因为若使用 f 单位,则有关 2π 的因子便可不用记住了,特别是在 12.1 节中当处理离散样本数据时更是如此。

从(12.0.1)式中可明显看出傅里叶变换是**线性**运算。两个函数之和的变换应等于函数变换之和。一个常数乘以一个函数之积的变换,应等于同一常数乘以函数变换的积。

在时域中,函数 $h(t)$ 可能具有一个或多个特殊对称性。它可能是**纯实函数**或者**纯虚函数**,也可能是偶函数 $h(t)=h(-t)$,或者奇函数 $h(t)=-h(-t)$。在频域中,这些对称性导出了 $H(f)$ 和 $H(-f)$ 之间的关系。下表列出了在时域和频域中对称性的对应关系:

假如……	则……
$h(t)$是实函数	$H(-f)=[H(f)]^*$
$h(t)$是虚函数	$H(-f)=-[H(f)]^*$
$h(t)$是偶函数	$H(-f)=H(f)$[即 $H(f)$是偶函数]
$h(t)$是奇函数	$H(-f)=-H(f)$[即 $H(f)$是奇函数]
$h(t)$是实偶函数	$H(f)$是实偶函数
$h(t)$是实奇函数	$H(f)$是虚奇函数
$h(t)$是虚偶函数	$H(f)$是虚偶函数
$h(t)$是虚奇函数	$H(f)$是实奇函数

在以后的章节中，将会讨论如何利用这些对称性来提高计算效率。

下面还有傅里叶变换的一些其他基本性质(用符号"⇔"表示一个变换对)。如果

$$h(t) \Longleftrightarrow H(f)$$

是一对变换对，则其他变换对为

$$h(at) \Longleftrightarrow \frac{1}{|a|}H(\frac{f}{a}) \qquad \text{"时间标度变换"} \tag{12.0.4}$$

$$\frac{1}{|b|}h(\frac{t}{b}) \Longleftrightarrow H(bf) \qquad \text{"频率标度变换"} \tag{12.0.5}$$

$$h(t-t_0) \Longleftrightarrow H(f)\,e^{2\pi i f t_0} \qquad \text{"时间平移"} \tag{12.0.6}$$

$$h(t)\,e^{-2\pi i f_0 t} \Longleftrightarrow H(f-f_0) \qquad \text{"频率平移"} \tag{12.0.7}$$

利用两个函数 $h(t)$和$g(t)$、以及它们相应的傅里叶变换 $H(f)$和 $G(f)$，可以组成两种别有意义的组合。这两个函数的**卷积**记做$g*h$，其定义为

$$g*h \equiv \int_{-\infty}^{\infty} g(\tau)h(t-\tau)\,d\tau \tag{12.0.8}$$

注意$g*h$ 是一个时域函数，而且$g*h=h*g$。由此推导出函数$g*h$ 是以下简单变换对中的一员：

$$g*h \Longleftrightarrow G(f)H(f) \qquad \text{"卷积定理"} \tag{12.0.9}$$

换言之，卷积的傅里叶变换恰好等于傅里叶变换的乘积。

这两个函数的**相关函数**，记做 $\mathrm{Corr}(g,h)$，其定义为

$$\mathrm{Corr}(g,h) \equiv \int_{-\infty}^{\infty} g(\tau+t)h(\tau)\,d\tau \tag{12.0.10}$$

该相关函数是 t 的函数，称 t 为滞后时间。所以，它依赖于时域，而且可以证明它是以下变换对中的一员，

$$\mathrm{Corr}(g,h) \Longleftrightarrow G(f)H^*(f) \qquad \text{"相关定理"} \tag{12.0.11}$$

(更一般而言，变换对中的另一员是 $G(f)H(-f)$，但通常情况下限制g 与 h 都为实函数，所以特设 $H(-f)=H^*(f)$。)这个结果证明，一个函数的傅里叶变换乘以另一函数傅里叶变换的复共轭就得到它们的相关函数的傅里叶变换。一个函数与其自身的相关称为它的**自相关**。在这种情况下，(12.0.11)式成为以下变换对

$$\mathrm{Corr}(g,g) \Longleftrightarrow |G(f)|^2 \qquad \text{"Wiener-Khinchin 定理"} \tag{12.0.12}$$

无论是在时域还是在频域里计算，一个信号的**总功率**总是相同的。这一结论就是 Parseval 定理：

$$\text{Total Power} \equiv \int_{-\infty}^{\infty} |h(t)|^2\,dt = \int_{-\infty}^{\infty} |H(f)|^2\,df \tag{12.0.13}$$

通常，人们想知道包含在 f 和$f+df$ 频率区间内的"功率大小"。在这种情况下，通常不区分正、负频率 f，而认为 f 是从 0("零频率"或直流)变化到 $+\infty$。这时，定义函数 h 的**单侧功率谱密度(PSD)**为

$$P_h(f) \equiv |H(f)|^2 + |H(-f)|^2 \qquad 0 \leqslant f < \infty \tag{12.0.14}$$

因此,总功率正好是 $P_h(f)$ 从 $f=0$ 到 $f=\infty$ 的积分。若函数 $h(t)$ 为实函数,则(12.0.14)式中两项相等,故 $P_h(f)=2|H(f)|^2$。要注意的是,会偶尔看到所定义的 PSD 没有这个因子 2。严格地说,这应称为**双侧功率谱密度**,可是有些参考书并未明确地指出所假定的是单侧还是双侧谱密度,我们总是假定由(12.0.14)式所定义的为单侧功率谱密度。图 12.0.1 将这两种定义进行了对比。

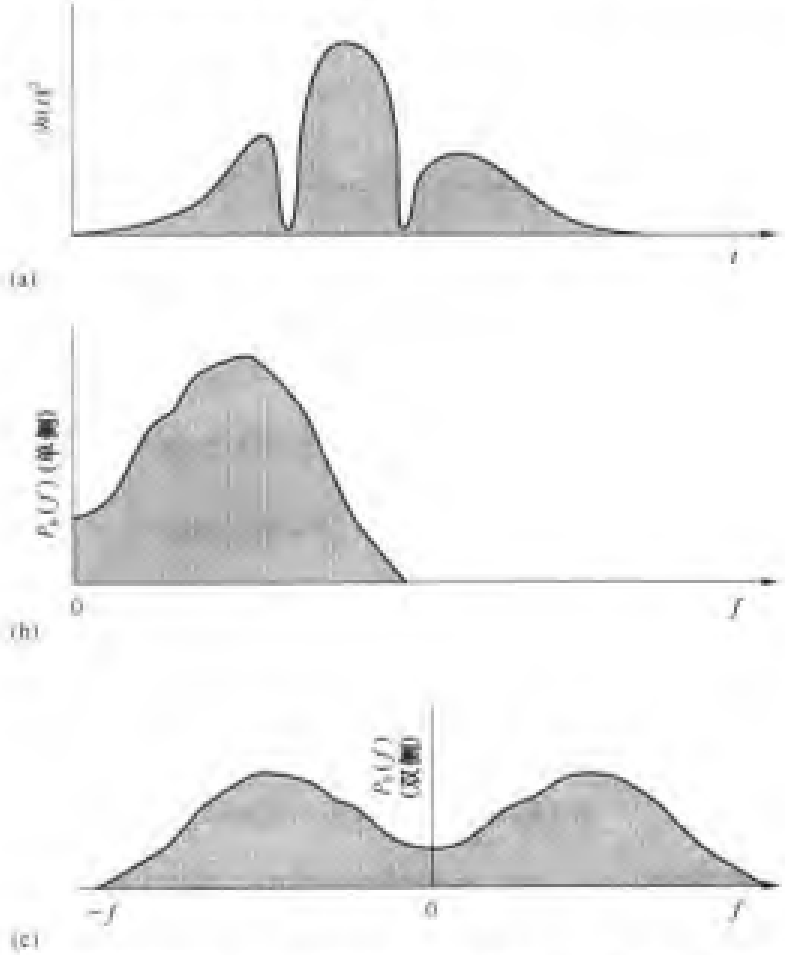


图 12.0.1 单侧和双侧功率谱的规范图。(a)中函数平方的曲线下的面积,等于如(b)所示的在正频率域上单侧功率谱密度曲线下的面积,也等于如(c)所示的正、负频率域上双侧功率密度曲线下的面积

如果函数 $h(t)$ 在 $-\infty<t<\infty$ 中变化是无穷的,则一般而言总功率和功率谱密度将是无穷的。于是人们感兴趣的就是**单位时间的(单侧或双侧)功率谱密度**。通过选取函数 $h(t)$ 的一个长而有限的截段来计算它的 PSD(即,计算在有限长度截段内等于 $h(t)$,而其他区域等于零的一个函数的 PSD),然后,用所取的截段长度除以所得的 PSD。在这种情况下,Parseval 定理表明,在正频率域上单位时间的单侧 PSD 的积分值等于信号 $h(t)$ 的均方振幅值。

可能会担心,单位时间的 PSD 作为频率 f 的函数,当用越来越长的数据截段来求它的值时,它会不会收敛。这个问题正是"功率谱估计"的专题内容,将在后面 13.4 节 - 13.7 节中讨论。现在只给出一个简单回答:除了那些 $h(t)$ 具有有限振幅的离散正弦波(或余弦波)分量的频率以外,在所有频率上,单位时间的 PSD 收敛到有限值。在其他频率上,它成为一个 δ 函数,即一个陡直尖峰,它的宽度越来越窄,但它的面积却收敛到该频率离散正弦分量的均方振幅值。

迄今为止,已阐述了本章将用到的所有解析公式,只有一个例外:即在计算工作中,特别是在带实验数据的计算工作中,几乎从来不给出一个连续函数 $h(t)$ 来计算,而是给出离散集 t_i 上的一系列测量值 $h(t_i)$,下一节将深入讨论。

参考文献和深入阅读

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.1　离散样本数据的傅里叶变换

在最普通情况下，函数 $h(t)$ 在等距时间间隔上被取样（即在这些时间间隔上函数值被记录）。设 Δ 表示相邻样本间的时间间隔，于是样本值的序列为

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \tag{12.1.1}$$

时间间隔 Δ 的倒数称为**取样率**。例如，如果 Δ 用秒度量，则取样率就是每秒记录的样本的个数。

12.1.1　取样定理与混叠现象

对于任何取样间隔 Δ，都有一个特殊的频率 f_c，称为 **Nyquist 临界频率**，定义为

$$f_c \equiv \frac{1}{2\Delta} \tag{12.1.2}$$

如果具有 Nyquist 临界频率的一个正弦波在它的正波峰值处被取样，则下一个样本值将取在负波谷处，然后样本值再一次取在正波峰值上，依次类推。也可以说是：一个正弦波的临界取样是每个周期有两个样本点。人们常常选用取样间隔 Δ 作为单位来测量时间，在这种情况下，Nyquist 临界频率恰好是常数 1/2。

Nyquist 临界频率之所以重要，是因为有两个彼此相关又互不相同的理由。一个理由是正面的，另一个理由是负面的。首先谈谈正面的理由，这就是众所周知的取样定理：如果一个连续函数 $h(t)$ 以间隔 Δ 取样，并且函数的振幅谱带宽受限，最高频率小于 f_c，即如果对于所有 $|f| \geq f_c$，$H(f) = 0$，则函数 $h(t)$ 可由它的样本值 h_n **完全确定**。实际上 $h(t)$ 可用以下公式表示：

$$h(t) = \Delta \sum_{n=-\infty}^{+\infty} h_n \frac{\sin[2\pi f_c(t - n\Delta)]}{\pi(t - n\Delta)} \tag{12.1.3}$$

有很多理由可说明这是一个引人注目的定理，理由之一是定理表明，在某种意义上，一个带宽有限的函数，其"信息量"比普通连续函数的"信息量"要无限地小。通常，人们所处理的这些具有物理背景的信号，其带宽总是有限的（或至少近似有限带宽）。例如，信号通过一个具有已知有限频率响应的放大器。在这种情况下，取样定理告诉我们，此信号的全部信息量可以通过取样被记录下来，其取样率 Δ^{-1} 等于所通过的放大器的最高频率的两倍，参看(12.1.2)式。

接下来谈谈负面的理由。它关系到对一个带宽频率不是限制为小于 Nyquist 临界频率的连续函数进行取样的效果。在这种情况下，将出现位于频率范围 $-f_c < f < f_c$ 以外的所有功率谱密度都错误地移入了该频率范围。这一现象称为**混叠效应**。正是通过离散取样的作用，频率范围 $(-f_c, f_c)$ 以外的任何频率分量都被混淆（错误地转移）进了该区域。读者可以很容易地确定，当且仅当 f_1 和 f_2 间距差是 $1/\Delta$ 的倍数，即这倍数恰好为 $(-f_c, f_c)$ 的频带宽度，则两个波 $\exp(2\pi i f_1 t)$ 和 $\exp(2\pi i f_2 t)$ 在间隔 Δ 上会给出同样的样本值。这时，一旦对信号进行了离散取样，便无法再消去所混叠的功率。克服混叠效应的方法有：(i) 知道信号的自然限制带宽，或者也可将连续信号通过模拟滤波，强制实行已知带宽的限制；(ii) 以足够高的取样率进行取样，以使在最高频率的一个周期内，能至少得到两个取样点。图 12.1.1 阐明了这些思想。

为了更好地理解这一点，可以持这样的观点：如果一个连续函数已被完全取样，则当从离散样本序列来估算它的傅里叶变换时，可以假设（或者还是这样假设好）在 $-f_c$ 至 f_c 频率范围之外，其傅里叶变换等于零。于是，可根据这一假设观察某函数的傅里叶变换，从而得知这个连续函数是否被完全取样（即混叠效应达到最小）。要做到这一点，注意观察，当频率从左边趋向 f_c 或者频率从右边趋向 $-f_c$ 时，傅里叶变换是否已经趋向于零。相反，若傅里叶变换趋于某个有限值，则说明频率范围以外的分量可能已经混叠到临界频率范围之内了。

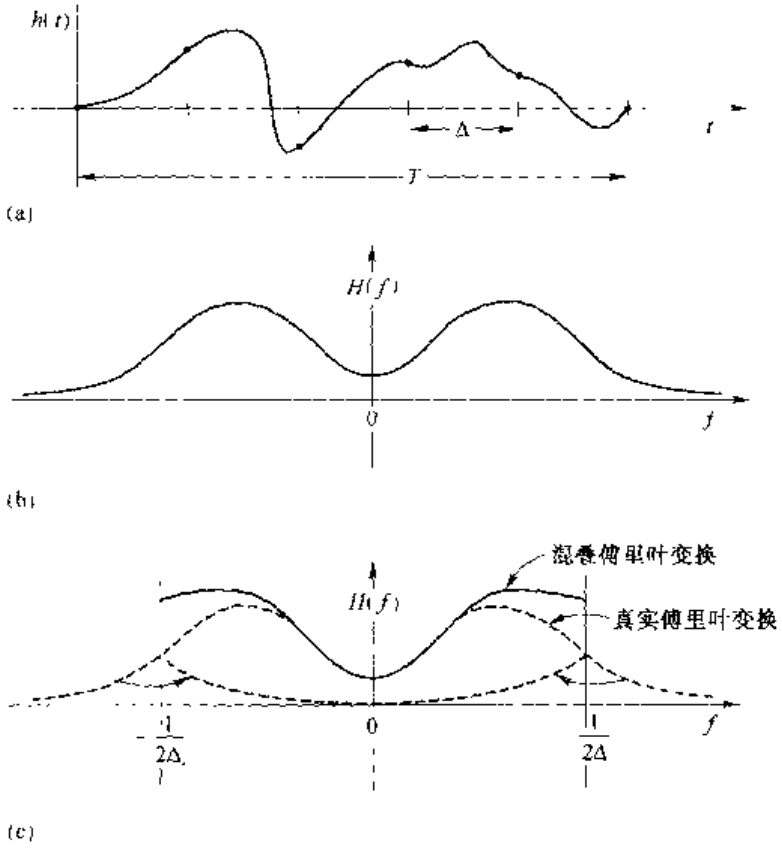


图 12.1.1 克服混叠效应的方法。(a)中所示的连续函数只在有限时间间隔 T 内不为零;(b)中所示的是(a)的傅里叶变换,其不是带宽受限的,而是对所有频率都有有限振幅 若原来的函数以取样间隔Δ进行取样,如(a)中所示,则其样本序列的傅里叶变换如(c)所示,它只定义在正负Nyquist临界频率之间,此范围以外的功率被折回并混叠到该范围上。只有在取样以前,将原来的函数通过低通滤波后,这种效应才能消除

12.1.2 离散傅里叶变换

现在,从函数的有限个样本点来估算这个函数的傅里叶变换。假如有 N 个连续相邻的样本值

$$h_k \equiv h(t_k), \qquad t_k \equiv k\Delta, \qquad k = 0, 1, 2, \ldots, N-1 \tag{12.1.4}$$

其中 Δ 为取样间隔。为了使问题简化,还假设 N 是偶数。如果函数 $h(t)$ 只在有限时间间隔内不为零,那么应假设整个有限时间间隔应包容在给出的这 N 个点的区间内。另外,如果函数 $h(t)$ 永远延伸下去,那么应假设这 N 个样本点至少应是 $h(t)$ 的一些典型的样本点,它与其他时间的 $h(t)$ 形式相似。

有了 N 个输入数,显然可以得出仅仅是 N 个独立的输出数。因此,不必去估算在 $-f_c$ 到 f_c 范围内所有 f 值的傅里叶变换 $H(f)$,而只需在离散值

$$f_n \equiv \frac{n}{N\Delta}, \qquad n = -\frac{N}{2}, \ldots, \frac{N}{2} \tag{12.1.5}$$

处求 $H(f)$ 估计值。(12.1.5)式中 n 的两个端点值正好对应于 Nyquist 临界频率范围的下极限和上极限。如果读者有经验,就会注意到(12.1.5)式中,n 的值有 $N+1$ 个,而不是 N 个。由此得出,n 的两个端点值不是无关的(实际上它们是相等的),而其他所有的值是无关的,这便将总数减少到 N 个。

余下的步骤是,有一个离散求和式近似(12.0.1)式中的积分:

$$H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t} dt \approx \sum_{k=0}^{N-1} h_k \, e^{2\pi i f_n t_k} \Delta = \Delta \sum_{k=0}^{N-1} h_k \, e^{2\pi i k n/N} \tag{12.1.6}$$

此处,(12.1.4)式和(12.1.5)式已用于最末的等式中。称(12.1.6)式的最末和式为 N 个点 h_k 的**离散傅里叶变换**,记为 H_n。

$$H_n \equiv \sum_{k=0}^{N-1} h_k\, e^{2\pi ikn/N} \tag{12.1.7}$$

离散傅里叶变换将 N 个复数(h_k)映射成了 N 个复数(H_n)。它不依赖于任何维参数,例如时标 Δ。当将一组数看成是以间隔 Δ 取样的连续函数的样本值时,(12.1.6)式给出了这组数的离散傅里叶变换和函数的连续傅里叶变换之间的关系,(12.1.6)式可以重新写成

$$H(f_n) \approx \Delta H_n \tag{12.1.8}$$

其中 f_n 由(12.1.5)式给出。

迄今为止,一直认为(12.1.7)式中的指数 n 由 $-N/2$ 变化到 $N/2$(参看(12.1.5)式)。然而不难看出,在(12.1.7)式中,n 是以周期 N 循环。因此,$H_{-n} = H_{N-n}$($n=1,2,\ldots$)。当记住了这种变换,一般可设 H_n 中的 n 是从 0 到 $N-1$ 间变化(一个完全周期)。n 和 k(h_k 中的)也在同一范围内变化,所以 N 个数到 N 个数的映射是很明显的。一旦遵循这个约定,就必须记住零频率对应于 $n=0$,正频率 $0<f<f_c$ 对应于值 $1\leqslant n\leqslant N/2-1$,负频率 $-f_c<f<0$ 对应于 $N/2+1\leqslant n\leqslant N-1$。值 $n=N/2$ 既对应于 $f=f_c$,也对应于 $f=-f_c$。

类似于连续傅里叶变换,离散傅里叶变换也具有对称性。例如,在(12.0.3)式之后的表中,只要将 $h(t)$ 改成 h_k,$H(f)$ 改成 H_n,$H(-f)$ 改为 H_{N-n},则表中所有的对称性都成立(同样,时间的"偶"、"奇"性涉及到 h_k 在 k 和 $N-k$ 处的值究竟是恒等还是互为相反数)。

离散傅里叶逆变换就是由 H_n 准确地恢复成这组数组 h_k,其公式是

$$h_k = \frac{1}{N}\sum_{n=0}^{N-1} H_n\, e^{-2\pi ikn/N} \tag{12.1.9}$$

注意(12.1.9)式和(12.1.7)式之间仅有的差别是:(i)改变了指数的符号;(ii)将结果除以 N。这意味着,计算离散傅里叶变换的程序在稍加修改后同样也适用于计算逆变换。

Parseval 定理的离散形式是

$$\sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N}\sum_{n=0}^{N-1} |H_n|^2 \tag{12.1.10}$$

卷积定理和相关定理((12.0.9)式和(12.0.11)式)也有类似的离散形式,将在 13.1 节和 13.2 节中分别进行讨论。

参考文献和深入阅读

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

12.2　快速傅里叶变换(FFT)

计算 N 个点的傅里叶变换,如(12.1.7)式,需要多少计算量?许多年来,直到 20 世纪 60 年代中期,标准答案是:定义 W 为复数

$$W \equiv e^{2\pi i/N} \tag{12.2.1}$$

则(12.1.7)式可写成

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k \tag{12.2.2}$$

换言之,一个矩阵与向量 h_k 相乘,该矩阵在(n,k)处的元素为常数 W 的 $n\times k$ 次幂。这样,矩阵相乘的结果得到一个向量,其分量就是 H_n 的分量。这个矩阵乘法显然需要 N^2 个复数乘法运算和加上生成 W 的幂次所

需的少量运算。因此,离散傅里叶变换呈现为一个 $O(N^2)$过程。可是,这种表面现象是虚假的。事实上,离散傅里叶变换可以用一种称为**快速傅里叶变换**的算法,即 FFT 来计算,只需 $O(N\log_2 N)$次运算。$N\log_2 N$ 和 N^2间的差别是巨大的。例如,当 $N=10^6$,在一个每秒运算百万次的计算机上,粗略地说,它们之间就是占用 30 秒 CPU 时间和两星期 CPU 时间的差别。直到 20 世纪 60 年代中期,从 J.W.Cooley 和 J.W.Tukey 的研究中,人们才逐渐熟悉 FFT 算法。回顾历史,现在我们知道,从 1805 年高斯起,已有十几个人分别独立地发现了计算 DFT 的有效算法,并在某些情况下得以实现(参见本节参考文献[1])。

FFT 的"再发现者"是 Danielson 和 Lanczos。在 1942 年,他们提供了算法的一种最清晰的推导。Danielson 和 Lanczos 证明了一个长度为 N 的离散傅里叶变换可以重新写成两个长度各为 $N/2$ 的离散傅里叶变换之和,其中一个变换由原来 N 个点中的偶数点构成,另一个变换由奇数点构成。简单证明如下:

$$
\begin{aligned}
F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \\
&= F_k^e + W^k F_k^o
\end{aligned}
\tag{12.2.3}
$$

最后一行中,W 是(12.2.1)式中的同一复常数。F_k^e 是由原 f_j 的偶分量得到的,长度为 $N/2$ 的傅里叶变换的第 k 个分量;而 F_k^o 是由奇分量得到的长度为 $N/2$ 的对应变换的分量。还需注意,(12.2.3)式中最后一行的 k 是从 0 到 N 间变化,而不是仅变到 $N/2$。然而,变换 F_k^e 和 F_k^o 对 k 而言其循环周期为 $N/2$,所以 F_k^e 和 F_k^o 要重复两次循环才能获得所有F_k。

Danielson-Lanczos 引理的巧妙之处就是在于它可以递归地使用。在将计算 F_k 的问题简化为计算 F_k^e 和 F_k^o 以后,可以再做同样的简化,将 F_k^e 简化成计算它的 $N/4$ 偶数项输入数据的变换和 $N/4$ 奇数项数据的变换。换言之,可以将 F_k^{ee} 和 F_k^{eo} 分别定义为在数据逐次细分后,数据的偶-偶点和偶-奇点的离散傅里叶变换。

虽然有一些其他情况的处理方法,但最容易的情况还是,原始 N 为 2 的整数次幂的 FFT。事实上,当 N 为 2 的整数次幂时,极力推荐只使用 FFT 方法。如果数据集的长度不是 2 的幂次时,则可添上一些零值,直到长度为 2 的下一幂次(在随后的章节中,将给一些更复杂的建议)。有了这种对 N 的限制,显然我们可以继续应用 Danielson-Lanczos 引理,直到将全部数据细分成长度为 1 的变换。什么是长度为 1 的傅里叶变换呢?它正是把一个输入数复制成它的一个输出值的恒等运算。换言之,对于所有 $\log_2 N$ 个 e 和 o 的每个模式,都有一个一点变换,它正是输入数字中的一个 f_n:

$$
F_k^{eoeeoeo\cdots oee} = f_n \qquad \text{对于某个 } n \tag{12.2.4}
$$

(当然,这个一点变换事实上不依赖于 k,因为它对 k 的周期为 1。)

下一步策略是,断定哪个 n 值对应于(12.2.4)式中哪一个 e 和 o 的模式。答案是:将 e 和 o 模式的次序颠倒,然后令 $e=0$,$o=1$,则得到 n 值的**二进制数**。这样做的原因是,将数据逐次细分偶数和奇数就是逐次检验 n 中的低位(最低有效位)。这个位序颠倒思想与 Danielson-Lanczos 引理一起可以开拓成一种更灵巧的方法,使 FFT 更为实用:假如将数据 f_j 的初始向量,重排成位序颠倒的顺序(参见图 12.2.1),以使每个数字不再按原 j 的顺序排列,而是按 j 的二进制数颠倒后得出的数排列。于是,Danielson-Lanczos 引理的递归应用的内部操作就变得十分简单,所给的点是一点变换。组合相邻的一对点就得到两点变换,再组合这些两点变换对的相邻对,又得到 4 点变换,依次类推,直到整个数据集的一半和另一半组合成最终的变换。每组合一次需 N 阶运算,并且明显地有 $\log_2 N$ 次组合,因此整个算法是 $N\log_2 N$ 阶运算量(假设分类成位序颠倒顺序的过程,其运算量不大于 $N\log_2 N$ 阶,通常情况也正是如此)。

于是,FFT 算法的结构分为两个部分。第一部分,将数据整理成位序颠倒的次序。幸运的是,这并不占据更多的存储空间,因为它只包含元素的交换(若 k_1 和 k_2 的位序颠倒,则 k_2 亦是 k_1 的位序颠倒)。第二部

分，有一个执行 $\log_2 N$ 次的外循环，它依次计算长度为 2,4,8,...,N 的变换。对于这一过程的每一步来说，为了履行 Danielson-Lanczos 引理，有两个嵌套的内循环，其涉及到已计算的子变换和每个变换的元素。通过限制外部调用正弦和余弦到外层循环，可以使运算更有效，在外层循环中只调用它们 $\log_2 N$ 次。倍角的正弦和余弦的计算是通过内循环中简单的递归关系进行的，参考(5.5.6)式。

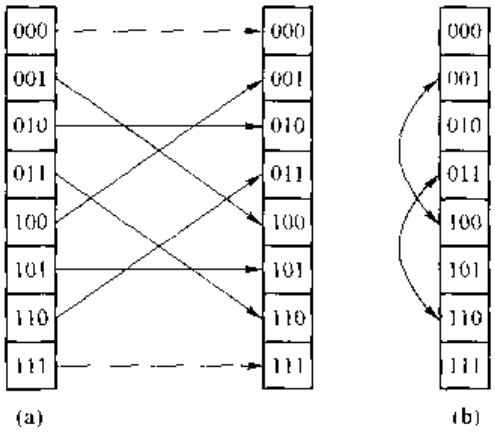


图 12.2.1　用位序颠倒重新排序一数组(长度为 8)。(a)两个数组之间的对应关系，它相当于(b)数组中适当位置上重新排列。位序颠倒重新排序是快速傅里叶变换算法(FFT)的必要部分

下面给出的 FFT 算法程序是以 N.M.Brenner 原来写的一个程序为基础。输入量是复数据点的个数(nn)、数据数组(data[0..2*nn-1])和 isign，isign 必须置成 +1 或 -1，它是(12.1.7)式中指数上 i 的符号。当 isign 置为 -1，则程序是计算逆变换式(12.1.9)——只是程序中没有乘上等式中出现的归一化因子 $1/N$，用户可以自己补加上。

注意，变量 nn 是复数据点的个数。但在许多计算机上，复数运算的实施效率是很低的，应避免使用它。实型数组(data[0..2*nn-1])的实际长度是 2 倍的 nn，而每个复数值占据了两个相继的存储单元。换言之，data[0]是 f_0 的实部，data[1]是 f_0 的虚部，依次类推，直到 data[2*nn-2]是 f_{N-1} 的实部，data[2*nn-1]是 f_{N-1} 的虚部。FFT 程序返回如 nn 个复数同样方式压缩的 F_n。

零频率分量 F_0 的实部和虚部在 data[0]和 data[1]中；最小的非零正频率的实部和虚部在 data[2]和 data[3]中；(数量上)最小的非零负频率的实部和虚部在 data[2*nn-2]和 data[2*nn-1]中。数量上增长的正频率存储在实-虚对 data[4]和 data[5]，直到 data[nn-2]和 data[nn-1]中。数量上增长的负频率存储在 data[2*nn-4]和 data[2*nn-3]直到 data[nn+2]和 data[nn+3]之中。最后，data[nn]和 data[nn+1]对存储一个混叠点的实部和虚部，这混叠点包含最大正频率和最小负频率。应该试着熟悉复频谱的这种内存排列方式，如图 12.2.2 所示，因为这是一种实用标准排列。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::four1(Vec_IO_DP &data, const int isign)
假如 isign 输入为 +1，数据 data[0..2*nn-1]由其离散傅里叶变换替换；假如 isign 输入为 -1，数据 data[0..2*nn-1]输
入的是 nn 个离散傅里叶的逆变换。data 是长度为 nn 的复数组或者等价于长度为 2*nn 的实数组。nn 必须是 2 的整数
次幂(这点未检验)。
{
    int n,mmax,m,j,istep,i;
    DP wtemp,wr,wpr,wpi,wi,theta,tempr,tempi;

    int nn=data.size()/2;
    n=nn << 1;
    j=1;
    for (i=1;i<n;i+=2) {                    这是程序中位序颠倒的部分。
        if (j > i) {                        交换两个复数。
            SWAP(data[j-1],data[i-1]);
            SWAP(data[j],data[i]);
        }
```

```
        m=nn;
        while (m >= 2 && j > m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    }
    下面开始是程序的Danielson-Lanczos部分。
    mmax=2;
    while (n > mmax) {                                  执行 log2 nn 次外循环。
        istep=mmax << 1;
        theta=isign*(6.2831853071969/mmax);             三角递归的初始赋值。
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (m=1;m<mmax;m+=2) {                         这是两个嵌套内循环。
            for (i=m;i<=n;i+=istep) {
                j=i+mmax;                               这是Danielson-Lanczos公式。
                tempr=wr*data[j-1]-wi*data[j];
                tempi=wr*data[j]+wi*data[j-1];
                data[j-1]=data[i-1]-tempr;
                data[j]=data[i]-tempi;
                data[i-1] += tempr;
                data[i] += tempi;
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;                三角递归。
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}
```

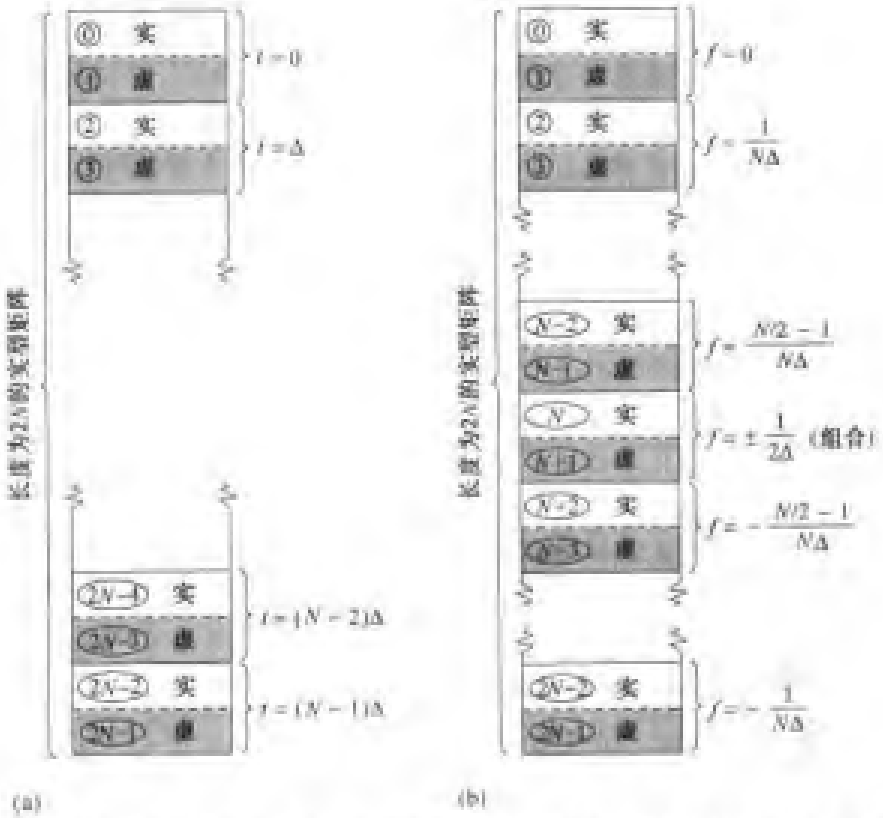


图 12.2.2 FFT的输入和输出数组。(a)输入数组包含 N(2的幂次)个时间上的复样本值,形成长度为 $2N$的实型数组,其实部和虚部交替相隔;(b)输出数组是包含N个频率值上的复傅里叶频谱,实部和虚部也交替相隔,数组从零频率开始,逐渐排列到最大正频率(这一处也可以作为最小的负频率),然后紧接着是负频率,从次最小的负频率一直升到恰好位于零以下的负频率

12.2.1　其他 FFT 算法

我们应该提到，对于上述给出的基本 FFT 算法还有一些变形算法。正如我们已经看到，上述算法首先将输入元素重排成位序颠倒的次序，然后用 $\log_2 N$ 个迭代得到输出变换。在文献中，这种算法称为**时间抽选 FFT 算法**或 **Cooley-Tukey FFT 算法**。现还可以导出另一种 FFT 算法，它首先对输入数据经过一系列 $\log_2 N$ 次迭代，然后将输出值重排成位序颠倒的次序。这种算法称为**频率抽选 FFT 算法**或 **Sande-Tukey FFT 算法**。在一些应用中，例如卷积(参见 13.1 节)来说，先将数据集放入傅里叶域内，然后经过某些操作后再返回来。在这种情况下，可以避免所有的位序颠倒。使用频率抽选算法(不用位序颠倒)进入"不规则的"傅里叶域内，在那里进行操作，然后用逆算法(也不用位序颠倒)再返回到时域。虽然，这样处理在理论上看是很完美的，但实际上节省不了很多的运算时间，因为位序颠倒只呈现 FFT 运算量的一小部分，并且在频率域中，大多数有用的运算都要求知道哪些点对应于哪些频率。

另一类 FFT 算法并不把长度为 N 的初始数据集细分成长度为 1 的基本变换，而是细分成某些 2 的较小幂次，例如 $N=4$，基数为 4 的 FFT；或 $N=8$，基数为 8 的 FFT。于是，这些小的变换可用最优化编码的小段来完成，这些最优化编码利用了这种特别小的 N 之独特的对称性。例如，当 $N=4$ 时，所参与的三角正弦和余弦全部是 ± 1 或 0，所以许多乘法可省略，只剩下大量的加法和减法。这种方法能比简单的 FFT 算法快 20% 或 30%，这是很有意义的。

对于长度为 N 而 N 不是 2 的幂次的数据集，也有一些 FFT 算法。它们是通过使用类似 Danielson-Lanczos 引理的方法，将初始问题逐次细分成一些较小的问题，这时不是每次一半一半地细分，而是每次用可除尽 N 的某个最小的素数因子去细分。N 的最大素数因子越大，这种方法的效果越差。如果 N 本身是素数，则不可能再细分，那么用户(不管他是否知道)正采用较慢的傅里叶变换，它是 N^2 阶，而不是 $N\log_2 N$ 阶运算量。然而也许有一些例外，即 **Winograd Fourier 变换算法**。Winograd 算法在某些方面是类似于基数为 4 和基数为 8 的 FFT 算法。Winograd 已推导出高度优化编码可用于小的 N，例如 $N=2,3,4,5,7,8,11,13,16$ 等进行离散傅里叶变换。这种算法还采用了新颖而巧妙的方法组合子因子。在分层次处理之前和之后，这一方法都包含数据的重新排序，但是它允许在算法内对乘法次数进行有效归并。对于一些特别的 N 值，Winograd 算法可以比最靠近 2 的整数次幂的简单 FFT 算法要有效地快(例如，快一倍)。然而，速度上的优势，必须与在这些变换中包含相当复杂的数据变址以及 Winograd 变换不能够"同址"计算的事实进行权衡比较。

最后，一类有趣的快速求卷积的变换是数论变换，这些格式是用某个大素数 $N+1$ 为模的整数算法来代替浮点算法，或用等价的模数算法代替 1 的 N 次方根。严格地说，这已根本不是傅里叶变换，但是它们性能十分相似，而且运算速度快得多。另一方面，它们的用途像相关的卷积一样受到一些限制，这是因为变换本身已不容易解释为"频率"谱了。

参考文献和深入阅读

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley).

Van Loan, C. 1992, *Computational Frameworks for the Fast Fourier Transform* (Philadelphia: S.I.A.M.).

Beauchamp, K.G. 1984, *Applications of Walsh Functions and Related Functions* (New York: Academic Press) [non-Fourier transforms].

Heideman, M.T., Johnson, D.H., and Burris, C.S. 1984, *IEEE ASSP Magazine*, pp. 14–21 (October).

12.3 实函数的FFT、正弦变换和余弦变换

常见的情况是,要求FFT的数据由实值样本 $f_j(j=0...N-1)$ 组成。为了使用four1,将这些样本数据存入一个复型数组中,并令其所有虚部为零。计算所得的变换 $F_n(n=0...N-1)$ 满足 $F_{N-n}{}^*=F_n$。因为对于 F_0 和 $F_{N/2}$,以及其他 $(N/2)-1$ 个独立值 $F_1...F_{N/2-1}$ 来说,这个复值数组都是实值,故它与原始的实数据集有相同的 $2(N/2-1)+2=N$ 个"自由度"。然而,从执行时间和对存储需求两方面来看,对实型数据采用完全复型的FFT算法,其效率是很低的。

有两种更好的方法。第一种是"成批生产法",将两个单独的实函数组合成输入数组,其组合方式是,能使各自单独的变换从计算结果中分离出来。这种方法在以下程序twofft中实现。这类似于搭配销售,购货者只需购买一件物品时却被强迫购买了两件。可是请记住,对于相关和卷积来说,这样可包括两种函数的傅里叶变换,故这是一举两得的便利方法。第二种方法是,除去多余的零值,巧妙地将实型输入数组组合成长度只有一半的复型数组。然后,可对这一组较短的长度数组执行复数型FFT,这里有个技巧就是如何从结果中得到所需要的答案。这在后面的程序realft中实现。

12.3.1 两个实函数同时变换

首先,说明如何利用变换 F_n 的对称性来一次处理两个实函数。因为输入数据 f_j 是实型的,故离散傅里叶变换的分量满足

$$F_{N-n}=(F_n)^* \tag{12.3.1}$$

其中星号表示复共轭。同理,纯虚型数据集合 g_j 的离散傅里叶变换有相反的对称性:

$$G_{N-n}=-(G_n)^* \tag{12.3.2}$$

因此通过把两个数据数组分别填充为four1中复型输入数组的实部与虚部,可以同时求得长度均为 N 的两个实函数的离散傅里叶变换。然后,借助于两种对称性,可以将计算结果的变换数组分离成两个复型数组。程序twofft实现了这些想法。

```
#include "nr.h"

void NR::twofft(Vec_I_DP &data1, Vec_I_DP &data2, Vec_O_DP &fft1,
    Vec_O_DP &fft2)
给定两个输入实数组data1[0..n-1]和data2[0..n-1],本程序调用函数four1并返回两个复型输出数组fft1[0..2n-1]
和fft2[0..2n-1],每个数组的复长度为n(即实长度为2×n),它们包含各自data数组的离散傅里叶变换。n必须是2的
整数次幂。
{
    int nn3,nn2,jj,j;
    DP rep,rem,aip,aim;

    int n=data1.size();
    nn3=1+(nn2=n+n);
    for (j=0,jj=0;j<n;j++,jj+=2) {          将两个实型数组组合成一个复型数组。
        fft1[jj]=data1[j];
        fft1[jj+1]=data2[j];
    }
    four1(fft1,1);                          求复型数组的变换。
    fft2[0]=fft1[1];
    fft1[1]=fft2[1]=0.0;
    for (j=2;j<n+1;j+=2) {
        rep=0.5*(fft1[j]+fft1[nn2-j]);      用对称性分离两个变换。
        rem=0.5*(fft1[j]-fft1[nn2-j]);
        aip=0.5*(fft1[j+1]+fft1[nn3-j]);
        aim=0.5*(fft1[j+1]-fft1[nn3-j]);    将它们按两个复型数组输送出来。
        fft1[j]=rep;
        fft1[j+1]=aim;
        fft1[nn2-j]=rep;
```

```
            fft1[nn3-j]= -aim;
            fft2[j]=aip;
            fft2[j+1]= -rem;
            fft2[nn2-j]=aip;
            fft2[nn3-j]=rem;
        }
    }
```

逆过程又是怎样的呢？假设有两个复变换数组，每一个都具有(12.3.1)式的对称性，那么就可知道这两个变换的逆变换都是实函数。能否在单个的 FFT 中对两个变换求逆呢？这种求逆变换甚至比变换更容易。可利用 FFT 是线性的这一事实，形成第一次变换加上 i 倍第二次变换的总和。可用 four1 并设 isign = −1 来求逆。计算得到的复型数组的实部和虚部就是两个所求的实函数

12.3.2　单个实函数的 FFT

第二种方法允许我们不用冗余位便可执行单个函数的 FFT，为了实施这种方法，将数据集一分为二，形成两个大小只有一半的实型数组。可以把上述程序用于这两组数组，当然结果将不是原始数据的变换，它将是两个变换的错乱集合，但其每个变换都有我们所需的一半信息，幸运的是，这种错乱是可以处理的，做法如下所述。

正确划分原始数据的方法是，把偶数项的 f_j 作为一个数据集，奇数项的 f_j 作为另一数据集。这样做的优点是，可以选取原始实型数组，并把它当做一个长度只有一半的复型数组 h_j 来处理。第一个数据集是这个复型数组的实部，第二个是它的虚部，如同 twofft 中规定的一样，而不需要重新组合排列。换言之，$h_j = f_{2j} + if_{2j+1}$ $(j = 0, \ldots, N/2 - 1)$。将这个数组提交给 four1，它将返回一个复型数组 $H_n = F_n^e + iF_n^o (n = 0, \ldots, N/2 - 1)$，并且

$$
\begin{aligned}
F_n^e &= \sum_{k=0}^{N/2-1} f_{2k}\, e^{2\pi ikn/(N/2)} \\
F_n^o &= \sum_{k=0}^{N/2-1} f_{2k+1}\, e^{2\pi ikn/(N/2)}
\end{aligned}
\tag{12.3.3}
$$

程序 twofft 的讨论是告诉读者如何从 H_n 中分出两个变换 F_n^e 和 F_n^o。现在，如何做才能将它们变成原始数据集 f_j 的变换 F_n 呢？建议回顾一下(12.2.3)式：

$$
F_n = F_n^e + e^{2\pi in/N} F_n^o \qquad n = 0, \ldots, N-1 \tag{12.3.4}
$$

根据实型数据集(冒充复型数组)的变换 H_n 直接来表达，其结果是

$$
F_n = \frac{1}{2}(H_n + H_{N/2-n}{}^*) - \frac{i}{2}(H_n - H_{N/2-n}{}^*)e^{2\pi in/N} \qquad n = 0, \ldots, N-1 \tag{12.3.5}
$$

下面是几点注意事项：

- 由于 $F_{N-n}{}^* = F_n$，所以没有存储整个频谱的点。正半频已足够，并能如同原始数据一样存入同一数组中，事实上，这一运算可以同址完成。
- 即使如此，仍需要数值 H_n，$n = 0, \ldots, N/2$，而 four1 只给出数值 $n = 0, \ldots, N/2 - 1$。利用对称性可补救，并有 $H_{N/2} = H_0$。
- 数值 F_0 和 $F_{N/2}$ 都是实数并且是无关的。为了真正地把整个 F_n 插入原始数组空间，把 $F_{N/2}$ 置入 F_0 的虚部是比较合适的。
- 尽管形式复杂，但以上过程是可逆的，首先从 F_0 中分离出 $F_{N/2}$，然后构成

$$
\begin{aligned}
F_n^e &= \frac{1}{2}(F_n + F_{N/2-n}^*) \\
F_n^o &= \frac{1}{2}e^{-2\pi in/N}(F_n - F_{N/2-n}^*)
\end{aligned}
\qquad n = 0, \ldots, N/2-1 \tag{12.3.6}
$$

再用 four1 来寻求 $H_n = F_n^{(1)} + iF_n^{(2)}$ 的逆变换。令人惊奇的是，具体的代数步骤实际上与正向变换的那些步骤完全相同。

下面是相应的典型程序:

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::realft(Vec_IO_DP &data, const int isign)
计算一组n个实值数据点的傅里叶变换。用复傅里叶变换的正半频率替换这些数据(它存储在数组data[0..n-1]中)。复
变换的第一个和最后一个分量的实数值分别返回单元data[0]和data[1]中。n必须是2的整数次幂。这个程序也能计算复
型数据数组的逆变换,只要该数组是实值数据的变换(在这种情况下,其结果必须乘以2/n)即可。
{
    int i,i1,i2,i3,i4;
    DP c1=0.5,c2,h1r,h1i,h2r,h2i,wr,wi,wpr,wpi,wtemp,theta;

    int n=data.size();
    theta=3.141592653589793238/DP(n>>1);          递归的初始赋值。
    if (isign == 1) {
        c2 = -0.5;
        fouri(data,1);                            此处是正向变换。
    } else {
        c2=0.5;                                   否则设置逆变换。
        theta = -theta;
    }
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0+wpr;
    wi=wpi;
    for (i=1;i<(n>>2);i++) {                      情况i=0时以下分别完成。
        i2=i+(i1=i+i);
        i4=i+(i3=n-i1);
        h1r=c1*(data[i1]+data[i3]);               两个分离变换从data中分离出来。
        h1i=c1*(data[i2]-data[i4]);
        h2r= -c2*(data[i2]+data[i4]);
        h2i=c2*(data[i1]-data[i3]);
        data[i1]=h1r+wr*h2r-wi*h2i;               此处重新组合以形成原始实型数据
        data[i2]=h1i+wr*h2i+wi*h2r;               的真实变换。
        data[i3]=h1r-wr*h2r+wi*h2i;
        data[i4]= -h1i+wr*h2i+wi*h2r;
        wr=(wtemp=wr)*wpr-wi*wpi+wr;              递归式。
        wi=wi*wpr+wtemp*wpi+wi;
    }
    if (isign == 1) {
        data[0] = (h1r=data[0])+data[1];          同时挤压第一个和最后一个数据使
        data[1] = h1r-data[1];                    它们都在原始数组中。
    } else {
        data[0]=c1*((h1r=data[0])+data[1]);
        data[1]=c1*(h1r-data[1]);
        fouri(data,-1);                           此处是isign=-1时的逆变换。
    }
}
```

12.3.3 快速正弦和余弦变换

在函数的傅里叶变换的其他用途中,还可用来解微分方程(参见19.4节)。求解的最常用边界条件是它们在边界上取零值,或者在边界上它们的导数为零。在第一种情况下,常用的变换是正弦变换,如下所示:

$$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N) \qquad \text{正弦变换} \tag{12.3.7}$$

其中$f_j(j=0,\ldots,N-1)$是数据数组,且$f_0=0$。

上式初看好像就是离散傅里叶变换的虚部。可是,正弦的变量与用离散傅里叶变换来计算的变量值相差2倍。正弦变换只使用在0至2π区间内一个完全函数集的一些**正弦函数**,并如同我们将看到的,余弦变换也只使用一些**余弦函数**。相比之下,标准的FFT却使用了正弦和余弦两个函数,并只使用各函数集中的一半(参见图12.3.1)

可以使(12.3.7)式"强制适合"于一种可用 FFT 来计算的形式,这种形式可允许. 这一思想是通过函数的最后列表值来扩展给定函数的右端。将数据的长度扩大一倍,使它们成为关于 $j=N$ 的奇函数,且 $f_N=0$.

$$f_{2N-j} \equiv -f_j \qquad j=0,\ldots,N-1 \tag{12.3.8}$$

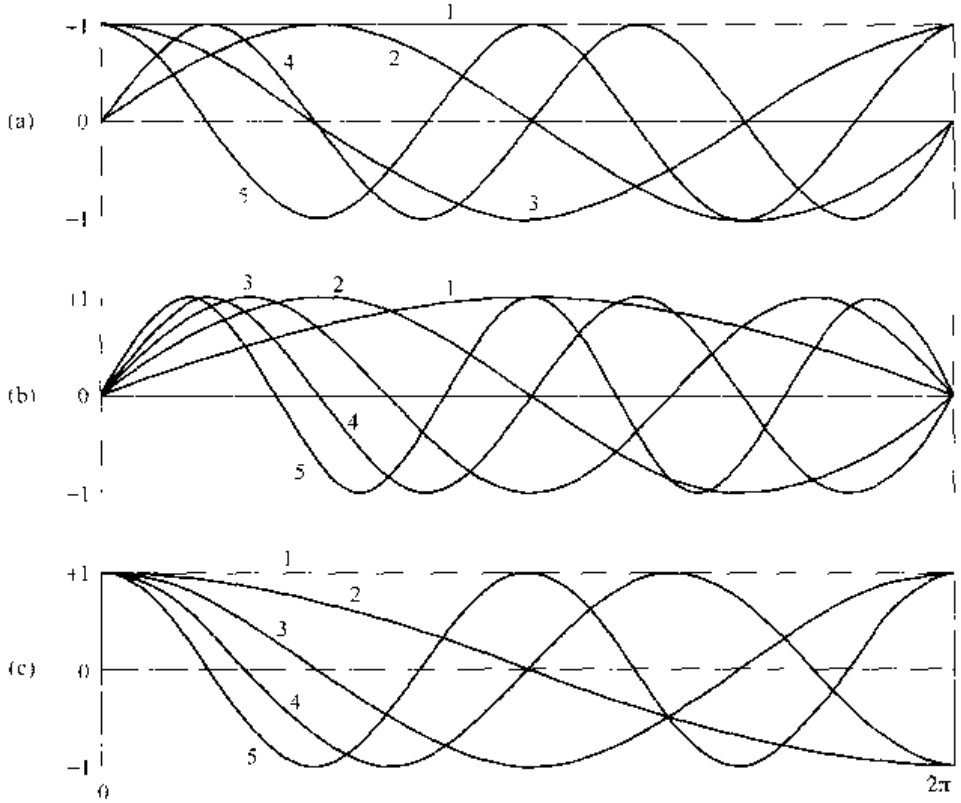


图 12.3.1　快速正弦和余弦变换. 绘制了傅里叶变换(a)、正弦变换(b)和余弦变换(c)所使用的基函数. 每种情况展示了前5个基函数(对于傅里叶变换,基函数的实部和虚部都予以展示). 虽然某些基函数出现在不止一个变换中,但基函数集是不同的. 例如,正弦变换中标号为(1)、(3)、(5)的基函数不出现在傅里叶变换中. 在表明的区间内,任意函数可由3个基函数集的任一函数集展开. 但是,若用正弦变换和余弦变换展开函数,最好函数与各自基函数的边界条件相匹配,即零函数值为正弦变换,零导数值为余弦变换

考虑这个扩展函数的 FFT.

$$F_k = \sum_{j=0}^{2N-1} f_j e^{2\pi ijk/(2N)} \tag{12.3.9}$$

该和式中从 $j=N$ 到 $j=2N-1$ 的这一半,可用 $j'=2N-j$ 重写为

$$\begin{aligned}\sum_{j=N}^{2N-1} f_j e^{2\pi ijk/(2N)} &= \sum_{j'=1}^{N} f_{2N-j'} e^{2\pi i(2N-j')k/(2N)} \\ &= -\sum_{j'=0}^{N-1} f_{j'} e^{-2\pi ij'k/(2N)}\end{aligned} \tag{12.3.10}$$

所以

$$\begin{aligned}F_k &= \sum_{j=0}^{N-1} f_j \left[e^{2\pi ijk/(2N)} - e^{-2\pi ijk/(2N)}\right] \\ &= 2i \sum_{j=0}^{N-1} f_j \sin(\pi jk/N)\end{aligned} \tag{12.3.11}$$

因此,除了因子 $2i$ 外,从扩展函数的 FFT 中得到正弦变换.

这种方法由于扩展了数据,使得计算效率降低了一半。这种低效率在FFT的输出时体现,因为变换的每个元素的实部都是零。对于一维问题,效率降低一半尚能接受,特别是考虑到该方法的简单性。但当计算二维或三维的偏微分方程时,效率将降低到原来的1/4或1/8,所以应努力消除这种低效性。

从原始实型数据数组 f_j 中,可以构造一个辅助数组 y_j,并将它运用于程序realft中。然后,可用输出的数据来构成我们所求的变换。对于数据 $f_j(j=1,\ldots,N-1)$ 的正弦变换,辅助数组为

$$\begin{aligned} y_0 &= 0 \\ y_j &= \sin(j\pi/N)(f_j + f_{N-j}) + \frac{1}{2}(f_j - f_{N-j}) \qquad j = 1,\ldots,N-1 \end{aligned} \tag{12.3.12}$$

此数组与原始数组的维数相同。注意,第一项关于 $j=N/2$ 是对称的,且第二项是反对称的。因而当realft用于 y_j 时,其结果具有下式给出的实部 R_k 和虚部 I_k:

$$\begin{aligned} R_k &= \sum_{j=0}^{N-1} y_j \cos(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j + f_{N-j}) \sin(j\pi/N) \cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} 2f_j \sin(j\pi/N) \cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} f_j \left[\sin\frac{(2k+1)j\pi}{N} - \sin\frac{(2k-1)j\pi}{N}\right] \\ &= F_{2k+1} - F_{2k-1} \end{aligned} \tag{12.3.13}$$

$$\begin{aligned} I_k &= \sum_{j=0}^{N-1} y_j \sin(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j - f_{N-j}) \frac{1}{2} \sin(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} f_j \sin(2\pi jk/N) \\ &= F_{2k} \end{aligned} \tag{12.3.14}$$

所以,F_k 可以确定如下:

$$F_{2k} = I_k \qquad F_{2k+1} = F_{2k-1} + R_k \qquad k = 0,\ldots,(N/2-1) \tag{12.3.15}$$

F_k 的偶数项可以直接确定。而奇数项需要通过递归运算得到,在(12.3.15)式中设 $k=0$,利用 $F_1 = -F_{-1}$ 求得初始点为

$$F_1 = \frac{1}{2}R_0 \tag{12.3.16}$$

实现程序为:

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::sinft(Vec_IO_DP &y)
计算存储在数组 y[0..n-1]中一组 n 个实值数据点的正弦变换。n 必须是 2 的幂次。输出时,y 被它的变换所代替。这
个程序,不加任何改变也可计算正弦变换的逆变换,但在这种情况下,输出数组应乘以 2/n。
{
    int j;
    DP sum,y1,y2,theta,wi=0.0,wr=1.0,wpi,wpr,wtemp;
```

```
        int n=y.size();
        theta=3.141592653589793238/DP(n);  递归的初始赋值。
        wtemp=sin(0.5*theta);
        wpr= -2.0*wtemp*wtemp;
        wpi=sin(theta);
        y[0]=0.0;
        for (j=1;j<(n>>1)+1;j++) {
            wr=(wtemp=wr)*wpr-wi*wpi+wr;   计算辅助数据的正弦。
            wi=wi*wpr+wtemp*wpi+wi;        为继续递归需要余弦。
            y1=wi*(y[j]+y[n-j]);           构造辅助数组。
            y2=0.5*(y[j]-y[n-j]);
            y[j]=y1+y2;                    j和N - j项相关联。
            y[n-j]=y1-y2;
        }
        realft(y,1);                       求辅助数组的变换。
        y[0]*=0.5;                         对以下奇数项的和式初始赋值。
        sum=y[1]=0.0;
        for (j=0;j<n-1;j+=2) {
            sum += y[j];
            y[j]=y[j+1];                   变换中偶数项直接确定。
            y[j+1]=sum;                    通过运行和式奇数项被确定。
        }
    }
```

有趣的是，正弦变换亦是其自身的逆变换。如果应用正弦变换两次，便可得到原始数组，但只是乘上了一个因子 $N/2$。

微分方程另一种常见的边界条件是，边界上函数的导数为零。在这种情况下，正常的变换是余弦变换。定义这种变换有几种可能的方式。每种定义都可认为是一种结果。该结果可以通过不同的扩展方法，从给定的数组构造一个双倍长度的偶数组来获得，或者也可以用所扩展的数组中是否包括 $2N-1$，$2N$ 或其他一些点的数来求得。实际上，许多可能的形式中只有两种是有用的，所以只讨论这两种形式。

余弦变换的第一种形式用 $N+1$ 个数据点：

$$F_k = \frac{1}{2}[f_0 + (-1)^k f_N] + \sum_{j=1}^{N-1} f_j \cos(\pi jk/N) \tag{12.3.17}$$

该结果是将给定数组扩展成为关于 $j=N$ 的偶数组而获得的，并有

$$f_{2N-j} = f_j, \qquad j = 0, \ldots, N-1 \tag{12.3.18}$$

若将扩展数组代入(12.3.9)式，接着类似地按推导式(12.3.11)的步骤，就将发现傅里叶变换是余弦变换(12.3.17)式的两倍。关于(12.3.17)式另一种方法正是前面已提到的 Chelyshev-Gauss-Lobatto 积分公式(参见 4.5 节)，它常应用于 Clenshaw-Curtis 自适应积分中(参见 5.9 节(5.9.4)式)。

另外一种方法，可以计算这个变换而不降低一半效率。此时，辅助函数是

$$y_j = \frac{1}{2}(f_j + f_{N-j}) - \sin(j\pi/N)(f_j - f_{N-j}) \qquad j = 0, \ldots, N-1 \tag{12.3.19}$$

替换(12.3.15)式，现在程序 realft 给出

$$F_{2k} = R_k \qquad F_{2k+1} = F_{2k-1} + I_k \qquad k = 0, \ldots, (N/2-1) \tag{12.3.20}$$

在此情况下，对于奇数 k 递归的起始值是

$$F_1 = \frac{1}{2}(f_0 - f_N) + \sum_{j=1}^{N-1} f_j \cos(j\pi/N) \tag{12.3.21}$$

该和式中没有出现 R_k 和 I_k，所以可以在数组 y_j 生成过程中累加该和式。

这个变换也是其自身的逆变换，所以下述程序能用于求变换及逆变换。请注意，虽然该余弦变换的形式有 $N+1$ 个输入和输出值，但它仅仅传送长度为 N 的数组到程序 realft 中。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::cosft1(Vec_IO_DP &y)
```

计算实值数据点集合y[0..n-1]的余弦变换。所得变换数据替代了数组y中的原始数据。n必须是2的幂次。该程序不用改变就可以计算余弦变换,但此时输出数组必须乘上2/n。

```
{
    const DP PI=3.141592653589793238;
    int j;
    DP sum,y1,y2,theta,wi=0.0,wpi,wpr,wr=1.0,wtemp;

    int n=y.size()-1;
    Vec_DP yy(n);                          程序realft, 需要长度为n的数组。
    theta=PI/n;                            递归的初始赋值。
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    sum=0.5*(y[0]-y[n]);
    yy[0]=0.5*(y[0]+y[n]);
    for (j=1;j<n/2;j++) {
        wr=(wtemp=wr)*wpr-wi*wpi+wr;       执行递归。
        wi=wi*wpr+wtemp*wpi+wi;
        y1=0.5*(y[j]+y[n-j]);              计算辅助函数。
        y2=(y[j]-y[n-j]);
        yy[j]=y1-wi*y2;                    j和N-j项相关联。
        yy[n-j]=y1+wi*y2;
        sum += wr*y2;                      为了用于后面所呈现的变换而求和。
    }
    yy[n/2]=y[n/2];                        y[n/2]未改变
    realft(yy,1);                          计算辅助函数的变换。
    for (j=0;j<n;j++) y[j]=yy[j];
    y[n]=y[1];
    y[1]=sum;                              sum是(12.3.21)式中F1的值。
    for (j=3;j<n;j+=2) {
        sum += y[j];                       (12.3.20)式。
        y[j]=sum;
    }
}
```

余弦变换的第二种重要形式定义为

$$F_k = \sum_{j=0}^{N-1} f_j \cos \frac{\pi k(j+\frac{1}{2})}{N} \tag{12.3.22}$$

其逆变换

$$f_j = \frac{2}{N} \sum_{k=0}^{N-1}{}' F_k \cos \frac{\pi k(j+\frac{1}{2})}{N} \tag{12.3.23}$$

其中,和式上的符号(′)表示对 $k=0$ 的那项有一系数 1/2。这一形式可以由扩展给定的数据来体现,定义扩展的数据从 $j=0,\ldots,N-1$ 到 $j=N,\ldots,2N-1$,在这种情况下,它对点 $N-\frac{1}{2}$ 是偶的和周期的(因此关于 $j=-\frac{1}{2}$ 也是偶的)。(12.3.23)式与高斯-切比雪夫求积公式有关(参见(4.5.19)式),也与切比雪夫近似(参见5.8节,(5.8.7)式)和 Clenshaw-Curtis 积分(参见5.9节)有关。

在"交错"网格上解微分方程时,余弦变换的这种形式是有效的,其中,变量置于网络点间的中心位置上。它也是数据压缩和图像处理领域中的标准形式。

在这种情况下,使用的辅助函数类似于(12.3.19)式:

$$y_j = \frac{1}{2}(f_j + f_{N-j-1}) + \sin\frac{\pi(j+\frac{1}{2})}{N}(f_j - f_{N-j-1}) \qquad j=0,\ldots,N-1 \tag{12.3.24}$$

执行类似于(12.3.12)式至(12.3.15)式所采用的步骤,可推导得

$$F_{2k} = \cos\frac{\pi k}{N} R_k - \sin\frac{\pi k}{N} I_k \tag{12.3.25}$$

$$F_{2k-1} = \sin\frac{\pi k}{N} R_k + \cos\frac{\pi k}{N} I_k + F_{2k+1} \tag{12.3.26}$$

注意由(12.3.26)式得

$$F_{N-1} = \frac{1}{2} R_{N/2} \tag{12.3.27}$$

所以偶分量可直接从(12.3.25)式求得，而奇分量可由(12.3.27)式起始，从 $k = N/2 - 1$ 逐渐递推(12.3.26)式求得。

因为此变换不是其自身的逆变换，所以必须颠倒上述步骤以求出逆变换。程序如下：

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::cosft2(Vec_IO_DP &y, const int isign)
计算实值数据 y[0..n-1]的"交错"余弦变换。所得变换数据替换了数组 y 中的原始数据。n 必须是 2 的幂次。令 isign
= +1 是余弦变换，而 isign = -1 是逆余弦变换。在逆变换运算时，输出数组要乘上 2/n。
{
    const DP PI=3.141592653589793238;
    int i;
    DP sum,sum1,y1,y2,ytemp,theta,wi=0.0,wi1,wpi,wpr,wr=1.0,wr1,wtemp;

    int n=y.size();
    theta=0.5*PI/n;                                  递归的初始赋值。
    wr1=cos(theta);
    wi1=sin(theta);
    wpr = -2.0*wi1*wi1;
    wpi=sin(2.0*theta);
    if (isign == 1) {                                正向变换。
        for (i=0;i<n/2;i++) {
            y1=0.5*(y[i]+y[n-1-i]);                  计算辅助函数。
            y2=wi1*(y[i]-y[n-1-i]);
            y[i]=y1+y2;
            y[n-1-i]=y1-y2;
            wr1=(wtemp=wr1)*wpr-wi1*wpi+wr1;         执行递归。
            wi1=wi1*wpr+wtemp*wpi+wi1;
        }
        realft(y,1);                                 交换辅助函数。
        for (i=2;i<n;i+=2) {                         偶数项。
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
            y1=y[i]*wr-y[i+1]*wi;
            y2=y[i+1]*wr+y[i]*wi;
            y[i]=y1;
            y[i+1]=y2;
        }
        sum=0.5*y[1];                                用 1/2 R_{N/2} 对奇数项递归进行初始赋值。
        for (i=n-1;i>0;i-=2) {
            sum1=sum;                                奇数项执行递归。
            sum += y[i];
            y[i]=sum1;
        }
    } else if (isign == -1) {                        逆变换。
        ytemp=y[n-1];
        for (i=n-1;i>2;i-=2)                         奇数项的差。
            y[i]=y[i-2]-y[i];
        y[1]=2.0*ytemp;
        for (i=2;i<n;i+=2) {                         计算 R_k 和 I_k。
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
            y1=y[i]*wr+y[i+1]*wi;
            y2=y[i+1]*wr-y[i]*wi;
            y[i]=y1;
            y[i+1]=y2;
        }
        realft(y,-1);
        for (i=0;i<n/2;i++) {                        反转辅助数组。
            y1=y[i]+y[n-1-i];
            y2=(0.5/wi1)*(y[i]-y[n-1-i]);
            y[i]=0.5*(y1+y2);
            y[n-1-i]=0.5*(y1-y2);
            wr1=(wtemp=wr1)*wpr-wi1*wpi+wr1;
            wi1=wi1*wpr+wtemp*wpi+wi1;
        }
    }
}
```

实施该算法的一种替代方法是,将 f_j 的偶元素复制到前 $N/2$ 个存储单元,而将 f_j 的奇元素以倒序复制后 $N/2$ 个存储单元,从而形成辅助函数。可是,如果没有暂存数组,实施此替代算法是不容易的,因此建议使用上述的同址算法。

最后,对小的整数 N 存在着快速余弦变换,它不依赖辅助函数或不使用 FFT 程序,而是常常对一些固定的小维数 N 硬件编码后直接计算余弦变换[1]

参考文献和深入阅读

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §10–10.

Sorensen, H.V., Jones, D.L., Heideman, M.T., and Burris, C.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 849–863.

Hou, H.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 1455–1461 [see for additional references].

Hockney, R.W. 1971, in *Methods in Computational Physics*, vol. 9 (New York: Academic Press).

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp 314–329.

Clarke, R.J. 1985, *Transform Coding of Images*, (Reading, MA: Addison-Wesley).

Gonzalez, R.C., and Wintz, P. 1987, *Digital Image Processing*, (Reading, MA: Addison-Wesley).

Chen, W., Smith, C.H., and Fralick, S.C. 1977, *IEEE Transactions on Communications*, vol. COM-25, pp. 1004–1009. [1]

12.4 二维或多维的 FFT

给定一个定义在二维网格 $0 \leq k_1 \leq N_1 - 1, 0 \leq k_2 \leq N_2 - 1$ 的复函数 $h(k_1, k_2)$。可以在同样的网格上定义它的二维离散傅里叶变换为复函数 $H(n_1, n_2)$。

$$H(n_1, n_2) \equiv \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_2 n_2/N_2)\ \exp(2\pi i k_1 n_1/N_1)\ h(k_1, k_2) \tag{12.4.1}$$

将"下标为2"的指数放到关于 k_1 的和式之外;或者颠倒求和的次序,将"下标为1"的指数放到关于 k_2 的和式之外。可以发现,连续地在每个指标上计算原函数的一维 FFT 就能求得二维 FFT。用符号表示为

$$\begin{aligned} H(n_1, n_2) &= \text{FFT-on-index-1}\,(\text{FFT-on-index-2}\,[h(k_1, k_2)]) \\ &= \text{FFT-on-index-2}\,(\text{FFT-on-index-1}\,[h(k_1, k_2)]) \end{aligned} \tag{12.4.2}$$

当然为了实用,N_1 和 N_2 两者都应该是 FFT 的某个有效长度,通常均为 2 的幂次。用一维 FFT 的程序按照(12.4.2)式编制的二维 FFT 程序比最初表面上看起来要略差一点。因为一维的程序要求它输入的数据在一维复型数组中是连续排列的,因此可发现,这样将从多维输入数组中无止境地复制数据出来,然后又将数据复制回去。因此这是不值得推荐的技术。当然,应该采用多维的 FFT 程序,下面给出一个程序。

显而易见,可将(12.4.1)式推广到二维以上,设 L 维的一般表达式为

$$\begin{aligned} H(n_1, \ldots, n_L) \equiv \sum_{k_L=0}^{N_L-1} \cdots \sum_{k_1=0}^{N_1-1} &\exp(2\pi i k_L n_L/N_L) \times \cdots \\ &\times \exp(2\pi i k_1 n_1/N_1)\ h(k_1, \ldots, k_L) \end{aligned} \tag{12.4.3}$$

其中 n_1 和 k_1 的范围从 0 至 $N_1 - 1, \ldots, n_L$ 和 k_L 的范围从 0 至 $N_L - 1$。在(12.4.3)式中要调用多次一维 FFT。首先对 $k_1, k_2, \ldots, k_{L-1}$ 的每个值做 FFT,求出 L 下标中的变换。然后对 $k_1, k_2, \ldots, k_{L-2}$ 和 n_L 的每个值做 FFT,求出 $L-1$ 下标中的变换,依次类推。如果采用上述这种做法,就只有依靠某个经验丰富的操作人员了。

(12.4.1)式或(12.4.3)式的逆变换正如期望的那样:将指数 i 变成 $-i$,并在整个式子前乘上一个整体因子 $1/(N_1 \times \cdots \times N_L)$。多维 FFT 也具有类似一维情况中已讨论过的许多其他特性:

- 在变换中频率按环绕顺序排列，但现在是对每个单独的维数。
- 输入数据也按环绕顺序处理。假如它们经过此周期的识别(在任何维数)是间断的，则因为间断性，在高频上频谱将会有某些过量的功率。如果介意的话，处理方法就是移去多维的线性趋势。
- 若需要进行空间滤波，并且为环绕效应而烦恼，则需要围绕多维数组边界进行零元填充。但是，请注意在多维变换中零元填充的代价很大。如果使用了过多的零元填充，将占用大量的存储空间，尤其是三维和更高维的情况。
- 如果一维或多维的变换不存在足够的带宽限制的话，混叠现象常常会发生。

这里提供的程序 fourn 是根据 N.M. Brenner 编写的一个程序派生出来的。作为输入，它要求：(i)一个向量，说明在每一维上数组的长度，例如(32,64)。请注意这些长度必须都是 2 的幂次，并且是在每个方向上复值的个数；(ii)一个通常等于 ±1 的标量，它表示是求变换还是逆变换；(iii)数据数组。(i)中向量的长度决定问题的维数。

关于数据数组，程序 fourn 是以一个一维实数数组来存取数组，即 data[0..(2$N_1N_2...N_L$)−1]，其长度是 L 维长度乘积的两倍。假设数组表示为 L 维复数组，其单个分量的顺序如下：(i)每一复数值使用两个连续的存储单元，实部随后跟虚部；(ii)当搜索数组时第一个下标变化最慢而最末一个下标变化最迅速(即在 C++ 标准语言中"以行存储")。几乎所有使 fourn 工作失败的原因都是由于对数据数组的上述顺序不正确选择的结果(图 12.4.1 说明了输出数据的格式)。

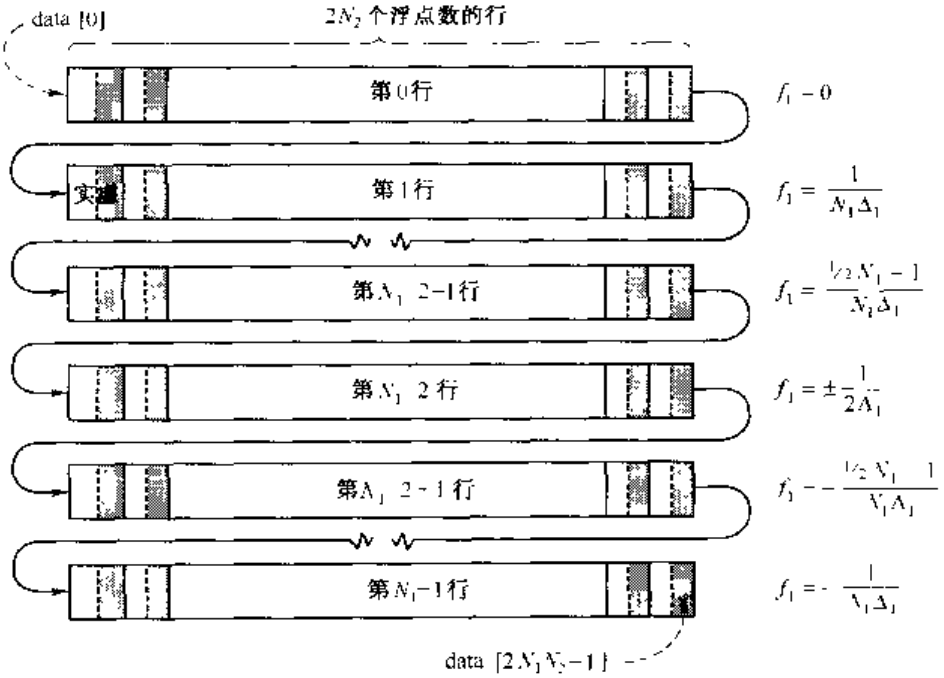


图 12.4.1　二维 FFT 的输出 $H(f_1,f_2)$ 中频率的存储排列。输入数据是二维 $N_1\times N_2$ 数组 $h(t_1,t_2)$(以复数的行存储)，输出也以复数行存储，每行对应于特定的 f_1 值，如图所示。每一行中频率 f_2 的排列完全如图 12.2.2 所示。Δ_1 和 Δ_2 分别是沿 1 和 2 方向上的取样间隔。(实)数组元素的总数为 $2N_1N_2$。程序 fourn 也能用于高于二维的情况，并用显而易见的方法推广存储的排列

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::fourn(Vec_IO_DP &data, Vec_I_INT &nn, const int isign)
```

若 isign 的输入值为 1，用 data 的 ndim 维离散傅里叶变换替换它自身的 data。nn[0..ndim−1]是包含每维长度(复值个数)的整型数组，它们必须全都是 2 的幂次。data 是实型数组，其长度为这些长度乘积的两倍，在这数组中数据以多维复型数组存储：每个元素的实部和虚部是连续地放置，并且当沿 data 进行时，数组的最右边序号增加最迅速。对于二维数组，即等于以行存储数组。若 isign 的输入值为 −1，则 data 被它们的逆变换乘以所有维数的长度之积所替代。

```
{
    int idim,i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
    int ibit,k1,k2,n,nprev,nrem,ntot;
    DP tempi,tempr,theta,wi,wpi,wpr,wr,wtemp;

    int ndim=nn.size();
    ntot=data.size()/2;                                  计算复值的总个数。
    nprev=1;
    for (idim=ndim-1;idim>=0;idim--) {                   维数上的总循环。
        n=nn[idim];
        nrem=ntot/(n*nprev);
        ip1=nprev << 1;
        ip2=ip1*n;
        ip3=ip2*nrem;
        i2rev=0;
        for (i2=0;i2<ip2;i2+=ip1) {                      这是程序的位序颠倒部分。
            if (i2 < i2rev) {
                for (i1=i2;i1<i2+ip1-1;i1+=2) {
                    for (i3=i1;i3<ip3;i3+=ip2) {
                        i3rev=i2rev+i3-i2;
                        SWAP(data[i3],data[i3rev]);
                        SWAP(data[i3+1],data[i3rev+1]);
                    }
                }
            }
            ibit=ip2 >> 1;
            while (ibit >= ip1 && i2rev+1 > ibit) {
                i2rev -= ibit;
                ibit >>= 1;
            }
            i2rev += ibit;
        }
        ifp1=ip1;                                        这里开始程序的Danielson-Lanczos部分。
        while (ifp1 < ip2) {
            ifp2=ifp1 << 1;
            theta=isign*6.28318530717959/(ifp2/ip1);     对三角递归初始赋值。
            wtemp=sin(0.5*theta);
            wpr= -2.0*wtemp*wtemp;
            wpi=sin(theta);
            wr=1.0;
            wi=0.0;
            for (i3=0;i3<ifp1;i3+=ip1) {
                for (i1=i3;i1<i3+ip1-1;i1+=2) {
                    for (i2=i1;i2<ip3;i2+=ifp2) {
                        k1=i2;                           Danielson-Lanczos 公式。
                        k2=k1+ifp1;
                        tempr=wr*data[k2]-wi*data[k2+1];
                        tempi=wr*data[k2+1]+wi*data[k2];
                        data[k2]=data[k1]-tempr;
                        data[k2+1]=data[k1+1]-tempi;
                        data[k1] += tempr;
                        data[k1+1] += tempi;
                    }
                }
                wr=(wtemp=wr)*wpr-wi*wpi+wr;             三角递归。
                wi=wi*wpr+wtemp*wpi+wi;
            }
            ifp1=ifp2;
        }
        nprev *= n;
    }
}
```

参考文献和深入阅读

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

12.5　二维和三维实数据的傅里叶变换

在图像处理领域中，二维 FFT 特别重要。一幅图像通常表示为像素强度值的一个二维数组，并且是实数（通常为正数）。通常要求从图像中滤波出高频或低频的空间分量，或者将图像与某些仪器的点扩展函数进行卷积或解卷积。所以，FFT 的运用是至今最有效的技术。

在三维情况下，FFT 的一种通常用法是，在代表三维离散空间中的三维网格上求解泊松（Poisson）方程，以得到位能（即电磁或万有引力）。其中源项（物质分布或电荷分布）和所求位能也都是实数。在二维和三维中具有大型的数组，通常存储量很受重视。所以，尽可能以"同址"来实施 FFT 是很重要的。我们希望有一个程序，其作用与多维 FFT 程序 fourn 相似（参见 12.4 节），但它是对实型输入数据操作而不是对复型数据操作。本节将给出这一程序。程序的导出与 12.3 节中推导出的一维程序 realft 相似（建议复习那节内容，特别是(12.3.5)式）。

可以很方便地将(12.4.3)式中的自变量 $n_1, \ldots, n_L$ 视为在波数空间的一个 L 维向量 $\vec{n}$，并且有整数格点上的值。然后变换 $H(n_1, \ldots, n_L)$ 即可写成 $H(\vec{n})$。

很容易知道，变换 $H(\vec{n})$ 在它 L 维的每一维上是周期性的。特别是，若 $\vec{P}_1, \vec{P}_2, \vec{P}_3, \ldots$ 表示成向量 $(N_1, 0, 0, \ldots), (0, N_2, 0, \ldots), (0, 0, N_3, \ldots)$ 等，则

$$H(\vec{n} \pm \vec{P}_j) = H(\vec{n}) \qquad j = 1, \ldots, L \tag{12.5.1}$$

对于任意输入数据——实型或复型，(12.5.1)式都成立。当数据是实数，有附加对称性

$$H(-\vec{n}) = H(\vec{n})^* \tag{12.5.2}$$

(12.5.1)式和(12.5.2)式表明，该全变换能够容易地从格点值 $\vec{n}$ 的子集中获得，该子集有

$$\begin{aligned} 0 &\leqslant n_1 \leqslant N_1 - 1 \\ 0 &\leqslant n_2 \leqslant N_2 - 1 \\ &\cdots \\ 0 &\leqslant n_L \leqslant \frac{N_L}{2} \end{aligned} \tag{12.5.3}$$

事实上，这个子集的数值是过于完全了，因为在 $n_L = 0$ 和 $n_L = N_L/2$ 处变换数值之间存在附加的对称关系。但是，这些对称性是很复杂的，并且它们运用后会变得相当混乱。所以，仍在(12.5.3)式的网格子集上计算 FFT，尽管这样会要求少量额外的存储单元，即变换不是完全"同址"（事实上，同址变换是可以实现的。但是我们发现，实际上不可能向用户讲解明白如何弄清楚变换的输出，即在某些特殊频率处如何寻找变换的实部和虚部分量）。

将实施三维情况 $L = 3$ 时的多维实傅里叶变换，并且输入数据存储为一组实型三维数组 data[0..nn1 − 1][0..nn2 − 1][0..nn3 − 1]。这个实施允许用来处理二维数据，它同样有效而且只需简单地选取 nn1 = 1（注意，必须设第一维等于 1），至少逻辑上而言，输出谱将以复型三维数组成包地返回，其中我们能够调用 SPEC[0..nn1 − 1][0..nn2 − 1][0..nn3/2]，参见(12.5.3)式。在变换的三维数据中，前两维的频率值分别为 f_1 和 f_2，以环绕顺序存储，即零频率值存在第一个下标值中，最小的正频率存储在第二个下标值中，最小负频率存储在最末下标值中，等等（参见关于程序 four1 和 fourn 的讨论）。三维数据中第三维仅仅返回频谱的正半部分。图 12.5.1 表明了这种逻辑存储结构。复输出谱的返回部分是图中较下方的非阴影部分。

相当对于逻辑而言，输出谱的物理结构与逻辑结构有少许不同，因为对于相同的实型和复型数组，C++ 语言没有方便的、易于移动的结构。下标范围 SPEC[0..nn1 − 1][0..nn2 − 1][0..nn3/2 − 1] 返回到输入的数组 data[0..nn1 − 1][0..nn2 − 1][0..nn3 − 1] 中，具有对应关系为

$$\begin{aligned} \mathrm{Re}(\texttt{SPEC[i1][i2][i3]}) &= \texttt{data[i1][i2][2*i3]} \\ \mathrm{Im}(\texttt{SPEC[i1][i2][i3]}) &= \texttt{data[i1][i2][2*i3+1]} \end{aligned} \tag{12.5.4}$$

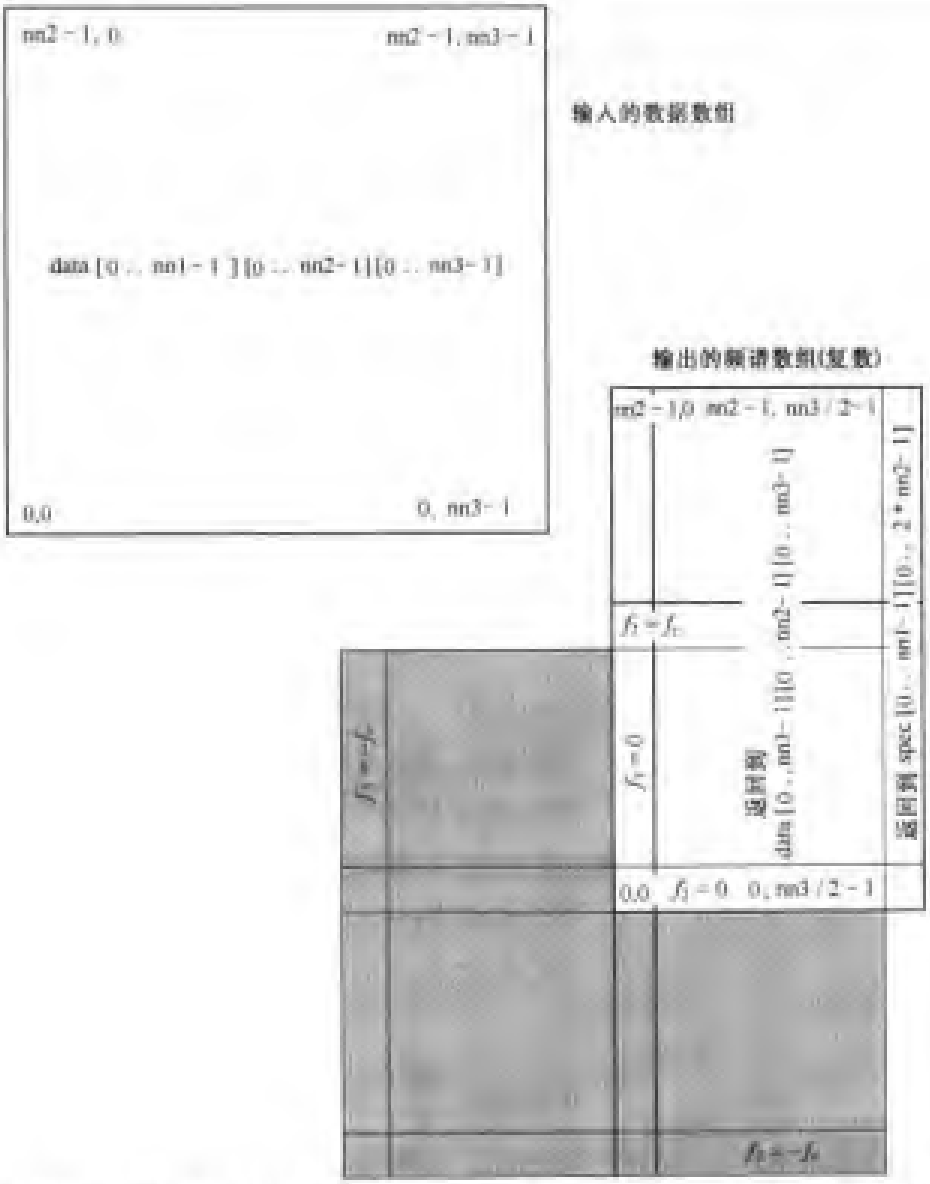


图 12.5.1 程序 dft3 中输入和输出数据的排列。全部数组被假设为具有范围[0..nn1-1]的第一个(最右边)的维,它伸出纸面。输入数组是一组三维实数组data[0..nn1-1][0..nn2-1][0..nn3-1](对于二维数据,可设为nn1=1)。输出数组可视为具有[0..nn1-1][0..nn2-1][0..nn3/2]维的一个单精度复型数组,参见(12.5.3)式,对应的频率 f_1 和 f_2 的分量以环绕次序存储,且只存储 f_3 的正频率部分(其余可通过对称性求得)。实际上,输出数据几乎都返回到输入数组data中,但部分存储在实数组speq[0..nn1-1][0..2*nn2-1]中,详见正文

其余"同址"的值 SPEC[0..nn1-1][0..nn2-1][nn3/2]是返回到二维双精度型数组 speq[0..nn1-1][0..2*nn2-1],其具有对应关系为

$$\begin{aligned}\mathtt{Re(SPEC[i1][i2][nn3/2])} &= \mathtt{speq[i1][2*i2]}\\ \mathtt{Im(SPEC[i1][i2][nn3/2])} &= \mathtt{speq[i1][2*i2+1]}\end{aligned} \tag{12.5.5}$$

注意,speq 中包含频率分量,其中第三个分量 f_3 在 Nyquist 临界频率 $\pm f_c$ 上。事实上,在某些应用中,此值将被忽略或设成零,因为此处的值在正、负频率之间是被混叠的。

根据以上这些介绍,称为 dft3 的实现程序可能有些令人失望。查看程序中最内层循环,它对最后面的变换下标实现(12.3.5)式。对 i3=0 单独地编码是为了将数组 speq 填满以便替换输入数据的数组被覆盖。三个 for 所包围的循环(对下标 i2,i3 和 i1,从内向外)事实上可以按任意顺序完成,因为它们的作用是可以互换的。我们选择了如程序中所示的顺序有以下几点考虑:(i)i3 绝对不能是最内层循环。因为若它是最内层循环,则 wr 和 wi 的递归关系就变得累赘了;(ii)在虚拟内存的计算机上,i1 必须是外循环,因为数组 data 中的结

果(以 C++ 语言的数组存储顺序)非常大,并且是以块序列的次序进行存取的。

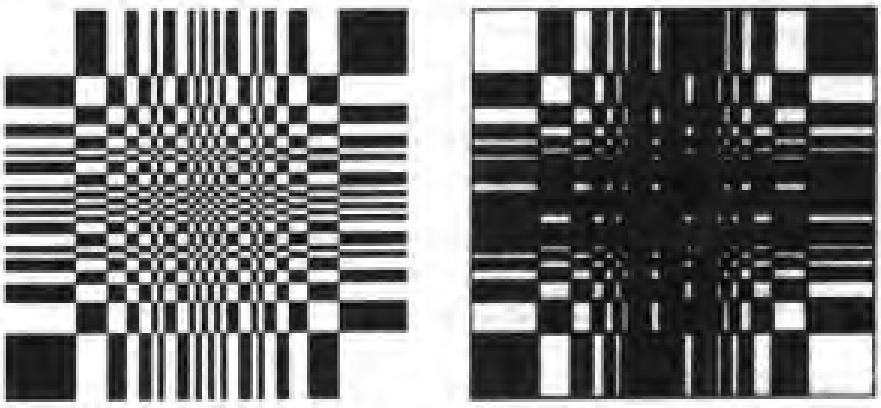
注意到,与程序 fourn 中所完成的计算相关的复型 FFT 的实际工作相比较,程序 rlft3 中所有的计算都是对数量级的,可以忽略不计。因为 C++ 语言中复型计算是低效的,所以以下的复数操作完全是根据实部和虚部分别执行的。下面的程序 rlft3 是基于 G.B. Rybicki 提出的较早的程序。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::rlft3(Mat3D_IO_DP &data, Mat_IO_DP &speq, const int isign)
已知三维实型数组 data[0..nn1-1][0..nn2-1][0..nn3-1](当 nn1=1 时对应于二维数组的情况),该程序(对 isign=1)以
两个复数组返回复 FFT;输出数组 data 中包含第二个频率分量的零频率和正频率值,而 speq[0..nn1-1][0..2*nn2-1]中
包含第一频率分量的 Nyquist 临界频率值。第一(和第二)频率分量是以零频率、正频率和负频率的标准环绕次序存储
的。详见正文中所述复数值如何排列的。对于 isign=-1,执行逆变换(要乘上一个因子 nn1*nn2*nn3/2),其输出数组
data(看成实型数组)由输入数组 data(看做复型)和 speq 得到。维数 nn1,nn2,nn3 通常必须是 2 的幂次。
{
    int i1,i2,i3,j1,j2,j3,ii3,k1,k2,k3,k4;
    DP theta,wi,wpi,wpr,wr,wtemp;
    DP c1,c2,h1r,h1i,h2r,h2i;
    Vec_INT nn(3);

    int nn1=data.dim1();
    int nn2=data.dim2();
    int nn3=data.dim3();
    c1=0.5;
    c2= -0.5*isign;
    theta=isign*(6.28318530717959/nn3);
    wtemp=sin(0.5*theta);
    wpr= -2.0*wtemp*wtemp;
    wpi=sin(theta);
    nn[0]=nn1;
    nn[1]=nn2;
    nn[2]=nn3 >> 1;
    Vec_DP data_v(&data[0][0][0],nn1*nn2*nn3);
    if (isign == 1) {                                   正变换的情况。
        fourn(data_v,nn,isign);                         绝大部分的计算时间花在此处。
        k1=0;
        for (i1=0;i1<nn1;i1++)                          周期地将data数组扩展成speq。
            for (i2=0,j2=0;i2<nn2;i2++,k1+=nn3) {
                speq[i1][j2++]=data_v[k1];
                speq[i1][j2++]=data_v[k1+1];
            }
    }
    for (i1=0;i1<nn1;i1++) {
        j1=(i1 != 0 ? nn1-i1 : 0);
        零频率是其自身的映射,其他按照负频率以环绕次序定位。
        wr=1.0;                                         三角递归初始赋值。
        wi=0.0;
        for (ii3=0;ii3<=(nn3>>1);ii3+=2) {
            k1=i1*nn2*nn3;
            k3=j1*nn2*nn3;
            for (i2=0;i2<nn2;i2++,k1+=nn3) {
                if (ii3 == 0) {                         (12.3.5)式。
                    j2=(i2 != 0 ? ((nn2-i2)<<1) : 0);
                    h1r=c1*(data_v[k1]+speq[j1][j2]);
                    h1i=c1*(data_v[k1+1]-speq[j1][j2+1]);
                    h2i=c2*(data_v[k1]-speq[j1][j2]);
                    h2r= -c2*(data_v[k1+1]+speq[j1][j2+1]);
                    data_v[k1]=h1r+h2r;
                    data_v[k1+1]=h1i+h2i;
                    speq[j1][j2]=h1r-h2r;
                    speq[j1][j2+1]=h2i-h1i;
                } else {
                    j2=(i2 != 0 ? nn2-i2 : 0);
                    j3=nn3-ii3;
                    k2=k1+ii3;
                    k4=k3+j2*nn3+j3;
                    h1r=c1*(data_v[k2]+data_v[k4]);
                    h1i=c1*(data_v[k2+1]-data_v[k4+1]);
```

```
                    h2i=c2*(data_v[k2]-data_v[k4]);
                    h2r= -c2*(data_v[k2+1]+data_v[k4+1]);
                    data_v[k2]=h1r+wr*h2r-wi*h2i;
                    data_v[k2+1]=h1i+wr*h2i+wi*h2r;
                    data_v[k4]=h1r-wr*h2r+wi*h2i;
                    data_v[k4+1]= -h1i+wr*h2i+wi*h2r;
                }
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;          实现递归。
            wi=wi*wpr+wtemp*wpi+wi;
        }
    }
    if (isign == -1) fourn(data_v,nn,isign);     逆变换的情况。
    k1=0;
    for (i1=0;i1<nn1;i1++)
        for (i2=0;i2<nn2;i2++)
            for (i3=0;i3<nn3;i3++) data[i1][i2][i3]=data_v[k1++];
}
```

下面给出几个程序段,说明如何对二维和三维数据调用 rlft3 程序。注意,这个程序实际上并不能区分二维和三维数据,二维同三维一样处理,只是第一维数组的长度设为 1。因为第一维是在最外层循环,所以实际上没有引起任何低效率。

第一个程序段是二维数据的 FFT,允许对它进行某种处理,如滤波处理,然后取其逆变换。图 12.5.2 显示了这段代码使用的例子:当一幅轮廓鲜明的图像的高频空间分量被因子(此处)$\max(1-6f^2/f_c^2,0)$抑制后,轮廓变得模糊起来。第二个程序段举例说明一个三维 FFT,其中三维方向具有不同的长度。第三个程序是一个卷积的例子,它可以用来计算三维信源分布所产生的位能。



(a) (b)

图 12.5.2 (a):一幅只有纯黑或纯白灰度的二维图像。(b):图像(a)用程序 rlft3 经过低通过滤波后的图像。其具有明显标尺特性的区域变成了灰色

```
#include "nr.h"

int main(void)          // example1
这段程序说明如何对一幅 256 × 256 的数字图像进行滤波。
{
    const int N2=256,N3=256;
    NRMat3d<DP> data(1,N2,N3);       注意第一个分量必须设为1。
    Mat_DP speq(1,2*N2);

//    ...                            这里图像必须装入data数组。
    NR::rlft3(data,speq,1);          这里数组data和speq必须乘以适当的(频率的)
//    ...                            滤波因子。
    NR::rlft3(data,speq,-1);         这里已被滤波的图像应该从data数组中卸下。
//    ...
    return 0;
}

int main(void)          // example2
这段程序说明如何对一个 32 × 64 × 16 三维实型数组作 FFT。
```

```
{
    const int N1=32,N2=64,N3=16;
    NRMat3d<DP> data(N1,N2,N3);
    Mat_DP speq(N1,2*N2);
//    ...                          这里装载data数组。
    NR::rlft3(data,speq,1);
//    ...                          这里卸下data和speq。
    return 0;
}

int main(void)        // example3
这段程序说明如何对两个 32×32×32 实型三维数组实行卷积,结果替换了原来的第一个数组。
{
    int j;
    DP fac,r,i,*sp1,*sp2;
    const int N=32;
    NRMat3d<DP> data1(N,N,N),data2(N,N,N);
    Mat_DP speq1(N,2*N),speq2(N,2*N);
//    ...
    NR::rlft3(data1,speq1,1);                 二个输入数组的FFT。
    NR::rlft3(data2,speq2,1);
    fac=2.0/(N*N*N);                          为了得到标准逆变换所需的因子。
    sp1 = &data1[0][0][0];
    sp2 = &data1[0][0][0];
    for (j=0;j<N*N*N/2;j++) {                 注意怎样使用指针sp1和sp2组成单个for
        r = sp1[0]*sp2[0] - sp1[1]*sp2[1];    循环而替换三个嵌套的for循环。
        i = sp1[0]*sp2[1] + sp1[1]*sp2[0];
        sp1[0] = fac*r;
        sp1[1] = fac*i;
        sp1 += 2;
        sp2 += 2;
    }
    sp1 = &speq1[0][0];                       如果在实施NRMat时数组元素不是连续
    sp2 = &speq2[0][0];                       存放的，需要把此循环写成两个以speq1
    for (j=0;j<N*N;j++) {                     和spec2为下标的显式循环，且不再使用
        r = sp1[0]*sp2[0] - sp1[1]*sp2[1];    指针。
        i = sp1[0]*sp2[1] + sp1[1]*sp2[0];
        sp1[0] = fac*r;
        sp1[1] = fac*i;
        sp1 += 2;
        sp2 += 2;
    }
    NR::rlft3(data1,speq1,-1);                一个FFT乘积的逆FFT。
//    ...
    return 0;
}
```

为了将程序 rlft3 扩展成四维，可以简单地增加一个附加的(最外层)对 i0 嵌套的 for 循环，类似于原先的 i1(也可将程序修改成任意维，如同程序 fourn，这是一个很好的编程练习)。

参考文献和深入阅读

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall).

Swartztrauber, P. N. 1986, *Mathematics of Computation*, vol. 47, pp. 323–346.

12.6　深入讨论：外部存储和局部内存的 FFT

有时在工作中，需要计算非常大的数据集的傅里叶变换，它大于计算机物理内存的大小。在这种情况下，数据必须存在某些外部的介质中，如磁盘或光盘。因此，需要一种算法，用于管理外部数据的连续读取，中间结果输入到外部介质以及之后的读取。

事实上，FFT 发明不久，Singleton[1]就已提出了这种算法。这种算法要求有四个顺序的存储设备，每个设备的容量是输入数据的一半。通常，输入数据前一半在一个设备内，另一半在另一个设备内。

Singleton 算法是基于通过下述的操作步骤，2^M 个值的位序颠倒是可能的：第一步，从两个输入设备交替地读入数值，并且写入到一个输出设备中(直到它存满数据的一半)，然后写入到另一个输出设备。第二步，将输

出设备变成输入设备,并将原输入设备变成输出设备。现在,从第一个设备复制两个数值,然后从第二个设备中复制两个数值,把它们写入(如前)到第一个输出设备中,写满后就写入第二个输出设备。依次进行,同时读4,8等输入数据的操作过程。直到完成了 $M-1$ 次操作后,数据便按位序颠倒次序排列。

其次,根据 Singleton 算法,本质上位序颠倒技术的过程和实现 Danielson-Lanczos 组合公式(12.2.3)的每一步过程可以交替地进行。其过程大致如下:起始和前述相同,在每一设备内各读入一半的输入数据。第一步,从每个输入设备中读入一个复型数值,形成两个组合,并且将各自写入两个输出设备之一。在这些"计算"过程以后,这些设备被改写,并且实现了"置换"过程,其中一组数值从第一个输入设备读入,又交换地被写入第一个和第二个输出设备;当第一个输入设备被耗尽,第二个输入设备同样地处理。这种计算和置换过程重复 $M-K-1$ 次,此处 2^K 是程序有效的内部缓冲器的大小。第二个相位的计算由最后 K 次计算过程组成。现在,第二个相位与第一个相位的区别是在计算过程中,由于置换是局部的,可以用同址实现,所以在第二个相位中不存在分离的置换步骤。处理数据总共需 $2M-K-2$ 步。

以下是 Singleton 算法的实现程序,基于本节参考文献[1]。

```
#include <iostream>
#include <fstream>
#include <cmath>
#include "nr.h"
using namespace std;

void NR::fourfs(Vec_FSTREAM_p &file, Vec_I_INT &nn, const int isign)
存储在外部介质的大数据集的一维或多维傅里叶变换。对于输入,nn[0..ndim-1]包括每一维的长度(实部和虚部数值
对的个数),并且必须都是2的幂次,这里ndim是维数。file[0..3]是指向4个临时文件的流式指针,每个大小要足以放
下一半数据。这4个流式指针在系统的"二进制"模式(相对为"文本"模式)下必须是开放的。输入数据按C++语言标
准顺序排列,以自然浮点数形式将数据的前一半存入file[0],其后一半存入file[1],每次缓冲读或写所处理的数据个数
为KBF,isign为1是FFT,为-1是逆变换。输出时,数组file中的值可能被置换,结果的前半部分存储file[2],后半部分
存储到file[3]。注意:当ndim>1时,输出不按C++语言的标准形式的顺序排列,而是最左端的数组下标循环最快,然
后是次右端,以此下去。ndim=2,意味着输出是以列存储,而非行存储。换言之,现在的输出是程序fourn产生的输出结
果的转置。
{
    const int KBF=128;
    static int mate[4]={1,0,3,2};
    int cc,cc0,j,j12,jk,k,kk,n=1,mm,kc=0,kd,ks,kr,na,nb,nc,nd,nr,ns,nv;
    DP tempr,tempi,wr,wi,wpr,wpi,wtemp,theta;
    Vec_DP afa(KBF),afb(KBF),afc(KBF);

    int ndim=nn.size();
    for (j=0;j<ndim;j++) {
        n *= nn[j];
        if (nn[j] <= 1) nrerror("invalid DP or wrong ndim in fourfs");
    }
    nv=0;
    jk=nn[nv];
    mm=n;
    ns=n/KBF;
    nr=ns >> 1;
    kd=KBF >> 1;
    ks=n;
    fourew(file,na,nb,nc,nd);
    此处开始第一个相位的变换。
    for (;;) {                                  计算过程开始。
        theta=isign*3.14159265358979 3/(n/mm);
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        mm >>= 1;
        for (j12=0;j12<2;j12++) {
            kr=0;
            do {
                cc0=(*file[na]).tellg()/sizeof(DP);
                (*file[na]).read((char *) &afa[0],KBF*sizeof(DP));
                cc=(*file[na]).tellg()/sizeof(DP);
                if ((cc-cc0) != KBF) nrerror("read error 1 in fourfs");
                cc0=(*file[nb]).tellg()/sizeof(DP);
```

```
        (*file[nb]).read((char *) &afb[0],KBF*sizeof(DP));
        cc=(*file[nb]).tellg()/sizeof(DP);
        if ((cc-cc0) != KBF) nrerror("read error 2 in fourfs");
        for (j=0;j<KBF;j+=2) {
            tempr=wr*afb[j]-wi*afb[j+1];
            tempi=wi*afb[j]+wr*afb[j+1];
            afb[j]=afa[j]-tempr;
            afa[j] += tempr;
            afb[j+1]=afa[j+1]-tempi;
            afa[j+1] += tempi;
        }
        kc += kd;
        if (kc == mm) {
            kc=0;
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
        }
        cc0=(*file[nc]).tellp()/sizeof(DP);
        (*file[nc]).write((char *) &afa[0],KBF*sizeof(DP));
        cc=(*file[nc]).tellp()/sizeof(DP);
        if ((cc-cc0) != KBF) nrerror("write error 1 in fourfs");
        cc0=(*file[nd]).tellp()/sizeof(DP);
        (*file[nd]).write((char *) &afb[0],KBF*sizeof(DP));
        cc=(*file[nd]).tellp()/sizeof(DP);
        if ((cc-cc0) != KBF) nrerror("write error 2 in fourfs");
    } while (++kr < nr);
    if (j12 == 0 && ks != n && ks == KBF) {
        na=mate[na];
        nb=na;
    }
    if (nr == 0) break;
}
fourew(file,na,nb,nc,nd);          置换过程开始。
jk >>= 1;
while (jk == 1) {
    mm=n;
    jk=nn[++nv];
}
ks >>= 1;
if (ks > KBF) {
    for (j12=0;j12<2;j12++) {
        for (kr=0;kr<ns;kr+=ks/KBF) {
            for (k=0;k<ks;k+=KBF) {
                cc0=(*file[na]).tellg()/sizeof(DP);
                (*file[na]).read((char *) &afa[0],KBF*sizeof(DP));
                cc=(*file[na]).tellg()/sizeof(DP);
                if ((cc-cc0) != KBF) nrerror("read error 3 in fourfs");
                cc0=(*file[nc]).tellp()/sizeof(DP);
                (*file[nc]).write((char *) &afa[0],KBF*sizeof(DP));
                cc=(*file[nc]).tellp()/sizeof(DP);
                    if ((cc-cc0) != KBF) nrerror("write error 3 in fourfs");
                }
                nc=mate[nc];
            }
            na=mate[na];
        }
        fourew(file,na,nb,nc,nd);
    } else if (ks == KBF) nb=na;
    else break;
}
j=0;
此处开始第二个相位的变换　现在,余留下的置换足以同址完成。
for (;;) {
    theta=isign*3.141592653589793/(n/mm);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    mm >>= 1;
```

```
    ks=kd;
    kd >>= 1;
    for (j12=0;j12<2;j12++) {
        for (kr=0;kr<ns;kr++) {
            cc0=(*file[na]).tellg()/sizeof(DP);
            (*file[na]).read((char *) &afc[0],KBF*sizeof(DP));
            cc=(*file[na]).tellg()/sizeof(DP);
            if ((cc-cc0) != KBF) nrerror("read error 4 in fourfs");
            kk=0;
            k=ks;
            for (;;) {
                tempr=wr*afc[kk+ks]-wi*afc[kk+ks+1];
                tempi=wi*afc[kk+ks]+wr*afc[kk+ks+1];
                afa[j]=afc[kk]+tempr;
                afb[j]=afc[kk]-tempr;
                afa[++j]=afc[++kk]+tempi;
                afb[j++]=afc[kk++]-tempi;
                if (kk < k) continue;
                kc += kd;
                if (kc == mm) {
                    kc=0;
                    wr=(wtemp=wr)*wpr-wi*wpi+wr;
                    wi=wi*wpr+wtemp*wpi+wi;
                }
                kk += ks;
                if (kk > KBF-1) break;
                else k=kk+ks;
            }
            if (j > KBF-1) {
                cc0=(*file[nc]).tellp()/sizeof(DP);
                (*file[nc]).write((char *) &afa[0],KBF*sizeof(DP));
                cc=(*file[nc]).tellp()/sizeof(DP);
                if ((cc-cc0) != KBF) nrerror("write error 4 in fourfs");
                cc0=(*file[nd]).tellp()/sizeof(DP);
                (*file[nd]).write((char *) &afb[0],KBF*sizeof(DP));
                cc=(*file[nd]).tellp()/sizeof(DP);
                if ((cc-cc0) != KBF) nrerror("write error 5 in fourfs");
                j=0;
            }
        }
        na=mate[na];
    }
    fourew(file,na,nb,nc,nd);
        jk >>= 1;
        if (jk > 1) continue;
        mm=n;
        do {
            if (nv < ndim-1) jk=nn[++nv];
            else return;
        } while (jk == 1);
    }
}


#include <fstream>
#include "nr.h"
using namespace std;
用于 fourfs 的实用程序、重环绕和币记数四个文件、
{
    int i;

    for (i=0;i<4;i++) (*file[i]).seekp(0);
    for (i=0;i<4;i++) (*file[i]).seekg(0);
    SWAP(file[1],file[3]);
    SWAP(file[0],file[2]);
    na=2;
    nb=3;
    nc=0;
    nd=1;
}
```

对于一维数据，Singleton 算法的输出结果，基本上和标准的 FFT(即 four1 程序)有相同的排列顺序。而对于多维数据，算法输出是常规排列(即程序 fourn 的输出)的转置，而不是在 fourn 中输出的 C++ 语言的数组标准顺序——即在扫描数据时，数组的最左边下标周期变化最快，然后是右边第二下标，等等。一般地，这种独特性是这个算法固有的特性，只是不太方便。对于卷积，计算非标准排列的两个变换的部分对部分乘积，然后对结果计算逆变换，这有些不方便。注意，若不同维数的长度不全相同，则必须在实行逆变换之前，颠倒 nn[0..ndim-1]中的数值次序(因此给出输出权组转置的维数)，也必须注意，和程序 fourn 相同，实现变换然后实现逆变换，逆变换的结果必须乘上原数据所有维数的长度之积。

下面给出一个习题，请读者将程序 fourfs 的输出重新排成标准序形式，需通过外部存储数据的过程。但我们怀疑这种重排是否真正必要。

有时，需要修改 fourfs 以适合自己的特殊应用。例如，程序中所写 $KBF=2^k$，它起了双重角色，它是内部缓冲器的大小，也是无格式化读和写记录大小。后者由于机器的 I/O 设备所允许的大小而受限。修改的简单方法是对于大的 KBF 数，采用双倍读取，就可减少一些过程。

修改 fourfs 的另一情况是用常规的算法完成高效率的 FFT(常规算法中，内存引用是极其非本地的)，这时虚拟内存有足够的地址空间，而没有足够的物理内存。在这种情况下，必须用将数组 afa，afb 和 afc 映射到地址空间的方法来代替读、写和重绕。换言之，这些数组被引用单个数据的数组替代，并且当 fourfs 程序提供一个 I/O 操作后，其填补的内容也得到修改。这种算法其内存引用的局部块是 KBF 的大小。尽管它要求花费二倍于同址 FFT 所需的虚拟内存，但有时其执行速度却大大地提高。

参考文献和深入阅读

Singleton, R.C. 1967, *IEEE Transactions on Audio and Electroacoustics*, vol. AU-15, pp. 91–97. [1]

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9.

第 13 章　傅里叶和谱的应用

13.0　引言

傅里叶方法已经使科学和工程领域，从射电天文到医学映像，从地震学到光谱学，都发生了重大的变革。本章将讨论一些傅里叶和谱方法的基本应用，是它们使这些变革成为可能。

“傅里叶”对数值工作者来说第一反应是“FFT”(快速傅里叶变换)。的确，傅里叶方法的广泛应用原则上应该归功于快速傅里叶变换的存在。暂且离开主题来说：假如有人将任意一个非平凡的算法加快一个百万因子或更多，则在世界上将开辟一条寻找这种算法有效应用的途径。FFT 最直接的应用就是数据的卷积和解卷积(参见 13.1 节)、相关和自相关(参见 13.2 节)、最优滤波(参见 13.3 节)、功率谱估计(参见 13.4 节)以及傅里叶积分的计算(参见 13.9 节)。

虽然 FFT 方法很重要，但并非是谱分析的全部内容。在 13.5 节将简要介绍时域数字滤波器。在谱域中，FFT 的一个限制因素是，常常将函数的傅里叶变换表示成 z 的多项式，即 $z=\exp(2\pi if\Delta)$(参见(12.1.7)式)。有时候，过程的谱形状不能很好地由这种形式来表示。而另外一种形式，它允许谱在 z 处有极点，并且常应用于线性预测技术(参见 13.6 节)和最大熵谱估计(参见 13.7 节)。

所有 FFT 方法的另一个重要的限制因素是，要求输入数据在均匀间隔内取样。对于非规则的或者非完整的取样数据，其他方法(尽管速度慢)是有效的，将在 13.8 节中讨论。

所谓小波方法就是函数空间的表示，它既不是在时域，也不是在谱域，而是在这两者之间，在 13.10 节将介绍这方面内容。最后 13.11 节的内容有点偏离主题，将讨论傅里叶取样定理的数值应用。

13.1　使用 FFT 做卷积和解卷积

已经在(12.0.8)式中对连续情形定义了两个函数的卷积，而且得出如(12.0.9)式的**卷积定理**。这个定理表明，两个函数卷积的傅里叶变换等于它们各自傅里叶变换的乘积。现在，我们来处理离散的情形。首先说明卷积是一个有用的过程，然后讨论如何运用 FFT 有效地计算卷积。

两个函数 $r(t)$ 和 $s(t)$ 的卷积，记做 $r*s$，从数学上来说，它等于这两函数反序的卷积 $s*r$。然而，在大多数应用中，这两个函数有十分不同的含义和特性。其中 s 函数，是一个典型信号或数据流，在时间上(或者在其他任何可能的适当的自变量上)无期限地进行下去。而另一函数 r 是一个“响应函数”，它是从最大值的两边都下降到零的有峰函数。卷积的结果是按照响应函数 $s(t)$ 提供的数据，在时间上抹去信号 $s(t)$，如图 13.1.1所示。特别是，假设在某一时刻 t_0 出现的单位面积的峰值函数或 δ 函数 s，其卷积结果将被涂抹成响应函数本身形状，只是时间从 0 移到 t_0，如同 $r(t-t_0)$。

在离散情况下，信号 $s(t)$ 用它在相等时间间隔上的样本值 s_j 所取代。响应函数同样也是离散数集 r_k，并且有以下解释：r_0 说明了将某一个通道中(j 的某一特定值)的多少重输入信号被复制到等同的输出通道(同样的 j 值)；r_1 说明在第 j 通道中有多少重输入信号又被复制到第 $j+1$ 输出通道；r_{-1} 说明有多少重信号被复制到第 $j-1$ 通道；对 r_k 的下标 k 取正负值即可以依次类推。图 13.1.2 描述了这种情况。

例如，一个响应函数 $r_0=1$，其他 r_k 都等于零，它正是一个恒等滤波器：一个信号与此响应函数的卷积给出同样的此信号。另一个例子是响应函数 $r_{14}=1.5$，而其余 r_k 都等于零。于此响应函数的卷积输出是输入信号乘上 1.5 并且延迟 14 个取样时间间隔而得到的信号。

显而易见，我们已经用文字描述了如下定义的有限持续时间 M 上的响应函数的离散卷积：

$$(r*s)_j \equiv \sum_{k=-M/2+1}^{M/2} s_{j-k}\, r_k \tag{13.1.1}$$

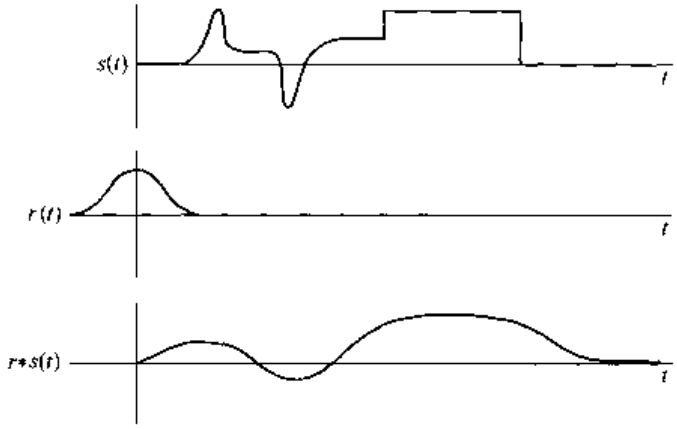


图 13.1.1　两个函数的卷积示例。信号 $s(t)$ 与响应函数 $r(t)$ 卷积。因为响应函数比原来信号的某些特征更为明显，因此在卷积中这些特征要被"洗掉"。在无附加噪声的情况下，这个过程可以被解卷积逆转过来

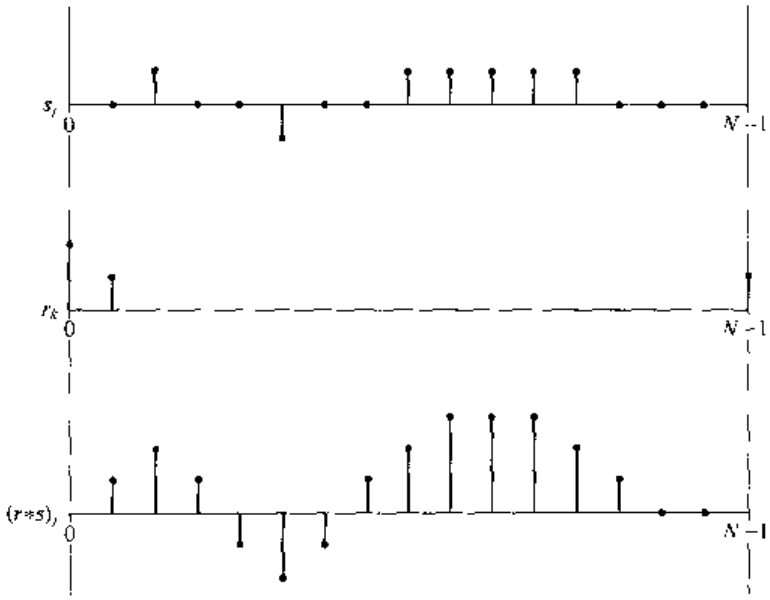


图 13.1.2　离散取样函数的卷积。注意，负时间响应函数是如何被环绕，并且被存储在 r_k 数组的最右端

如果一个离散响应函数只在某范围 $-M/2 < k \leq M/2$ 内为非零，其中 M 是足够大的偶整数，那么称响应函数为**有限脉冲响应(FIR)**，其**持续时间**为 M(注意，定义 M 为 r_k 的非零值的个数，这些值跨越了 $M-1$ 个取样时间间隔)。在大多数实际情况下，M 有限的情况是有意义的，这是因为响应确实有一个有限的持续时间，或者因为我们选择在某点将它截断，并用一个有限持续时间响应函数来逼近它。

离散卷积定理：如果信号 s_j 是以 N 为周期，所以它完全由 N 个值 $s_0, \ldots, s_{N-1}$ 所确定，则它与一个有限持续时间 N 之响应函数的卷积，就是一种离散傅里叶变换对，

$$\sum_{k=-N/2+1}^{N/2} s_{j-k}\, r_k \quad \Longleftrightarrow \quad S_n R_n \tag{13.1.2}$$

其中 $S_n\,(n=0,\ldots,N-1)$ 是数值 $s_j\,(j=0,\ldots,N-1)$ 的离散傅里叶变换，而 $R_n\,(n=0,\ldots,N-1)$ 是数值 r_k $(k=0,\ldots,N-1)$ 的离散傅里叶变换。r_k 的这些值与范围 $k=-N/2+1,\ldots,N/2$ 内的值相等，但按环绕顺序，如 12.2 节末尾所描述。

13.1.1 用零元填充的终端效应处理

离散卷积定理假定了两个非通用的事实。首先,它假定输入信号是周期的,但实际数据是不再重复地永远取下去,或是由一个有限长度的非周期信号伸展所组成;其次,卷积定理所取的将响应函数的持续时间与数据的周期相同,两者都是 N。需要首先围绕这两种约束进行工作。

第二种约束是非常简单明了的。人们对那种持续时间 M 远远小于数据集长度 N 的响应函数感兴趣。在这种情况下,简单地通过零元填充来扩展响应函数的长度到 N,即在 $M/2 \leqslant k \leqslant N/2$ 和 $-N/2+1 \leqslant k \leqslant -M/2+1$ 内,定义 $r_k=0$。而处理第一种约束,则需要更多要求。因为卷积定理轻率地假定数据是周期性的,这可能造成数据流 s_{N-1}, s_{N-2} 等远端的环绕数据对第一输出通道 $(r*s)_0$ 不真实地"污染"(如图 13.1.3 所示)。因此,必须在 s_j 向量的端部建立一个零元填充值的缓冲区,为的是使得这种污染为零。在这个缓冲区中需要的零值,确切地说应与响应函数非零(绝对值)的最大负索引一样多。例如,如果 r_{-3} 非零,而 $r_{-4}, r_{-5}, \ldots$ 均为零,则需要的数据端部填充三个零元:$s_{N-3}=s_{N-2}=s_{N-1}=0$。这些零元将保护第一输出通道 $(r*s)_0$ 不受环绕污染。显然,第二输出通道 $(r*s)_1$ 和随后的通道也都受这些零元的保护。设 K 表示填充零元的个数,则最后实际输入数据点的个数为 s_{N-K-1}。

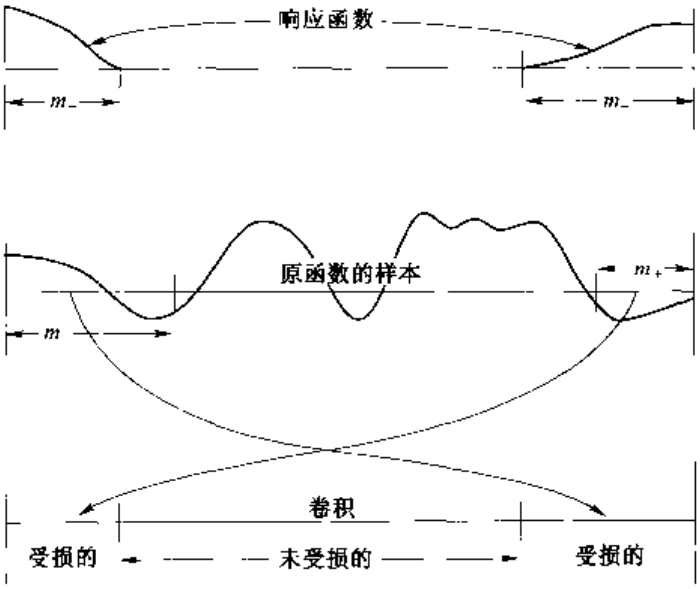


图 13.1.3 有限段函数卷积中的环绕问题。不仅响应函数的环绕必须看成是循环的,而且原样本函数也必须如此。所以,原函数每一端的一部分都被它与响应函数的卷积错误地环绕

那么,现在关于这个最终输出通道的污染情况如何呢?由于现在数据以 s_{N-K-1} 结尾,故有意义的最终输出通道是 $(r*s)_{N-K-1}$。这一通道可能被输入通道 s_0 的环绕而污染,除非数字 K 也足够大,足以照顾到最大正下标 k,而其响应函数 r_k 是非零的。例如,如果 r_0 直到 r_6 都不为零,而 $r_7, r_8, \ldots$ 都为零,则在数据末端至少需要 $K=6$ 个零元填充:$s_{N-6}=\ldots=s_{N-1}=0$。

总而言之,在数据一端用大量零元来填充,填充个数等于响应函数的最大正持续时间或者最大负持续时间,取决于**哪一段更大一些**(对于持续时间为 M 的对称响应函数,则只需 $M/2$ 个零元填充)。将这一操作和上述响应函数 r_k 的填充相结合,便可有效地形成周期性数据。图 13.1.4 阐述了这个问题。

13.1.2 FFT 对卷积的应用

用零元填充而完善的数据,现在是一组实数 $s_j(j=0,\ldots,N-1)$,响应函数被零元填充成持续时间为 N,并且以环绕顺序排列(一般来说,这意味着在数组的中部,r_k 的一大段邻近部分取零值,而非零值密集在数组的两个端部)。现在就可按以下步骤计算离散卷积:用 FFT 算法计算 s 和 r 的离散傅里叶变换。按逐个分量

将两个变换乘在一起，记住这两个变换是由复数组成的。然后用 FFT 算法计算乘积的离散傅里叶逆变换，其结果就是卷积 $r * s$

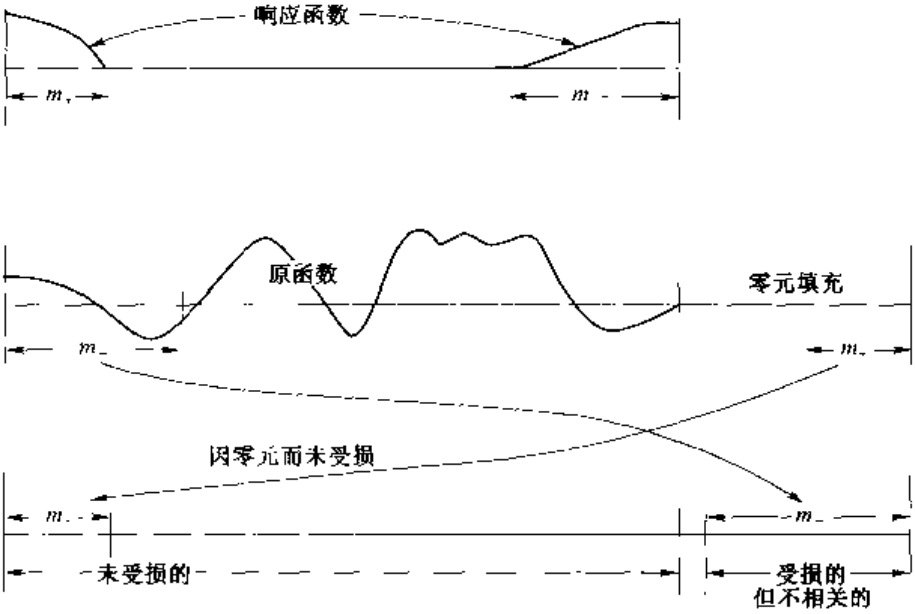


图 13.1.4　解决环绕问题的零元填充。原来函数被零元进行了扩展，它服务于两个目的：当零元环绕回来，它们并没有干扰真正的卷积，并且当原来函数环绕到零域时，这个区域可以丢弃

什么是**解卷积**呢？解卷积就是取消已被涂改数据集的过程，该涂改是在一个已知函数的影响下发生的，例如，由于不太完善的测量装置的已知影响。解卷积的定义类似于卷积式(13.1.1)，只是现在左端是已知的，并且(13.1.1)式可以被认为是以 s_j 为未知量的 N 个线性方程的方程组。在(13.1.1)式的时域中，求解这些联立线性方程，在大多数情况下是不现实的，但是 FFT 使问题简单化了。不再把信号的变换与响应函数的变换相乘以得到卷积的变换，而是用响应的变换去除已知卷积的变换，以得到解卷积信号的变换。

从数学上看，如果响应函数变换在某值 R_n 处恰好为零，因此不能用它做除数，则这个过程就要出错。这说明原来的卷积确实丧失了在那个频率上的所有信息，所以那个频率分量的重建是不可能的。然而，除数学问题以外，解卷积过程还有其他实践中的缺点。这个过程一般对输入数据中的噪声非常敏感，并且对于已知响应函数 r_k 的精确度也很敏感。由于这些原因，即使对解卷积是完全合理的尝试，有时也可能会产生荒谬的结果。在这种情况下，可能要利用附加的**最优滤波**过程，这将在 13.3 节中讨论。

下面是关于卷积和解卷积的程序，其使用 12.2 节中程序 four1 实现 FFT。因为数据函数和响应函数都是实型而非复型的，所以两者变换可用 twofft 同时进行。然而，如果 data 和 respns 数组总是具有不同的数据尺度，则应两次调用 realft 以使舍入误差达到最小。数据假定存储在一个双精度型数组 data[0..n-1]中，其中 n 是 2 的整数次幂。响应函数假定以环绕次序存储在数组 respns[0..m-1]中，m 值可以是小于或等于 n 的任意奇整数，因为程序要做的第一件事就是将响应函数以适当的环境顺序重新复制到长度为 n 的数组中。结果返回到 ans 中，ans 也作为工作空间。

```
#include "nr.h"

void NR::convlv(Vec_I_DP &data, Vec_I_DP &respns, const int isign,
    Vec_O_DP &ans)
```

实型数据集 data[0..n-1]与响应函数 respns[0..m-1]的卷积和解卷积(包括任何为用户提供的零元填充)，其中 m 为奇数且不大于 n。响应函数必须以环绕顺序存储：数组 respns 的前一半包含正时间上的脉冲响应函数，而数组的后一半包含时间上的脉冲响应函数，计算从最高元素 respns[m-1]开始。在输入时，isign 是 +1 做卷积，isign 是 -1 做解卷积。结果返回到 ans[0..n-1]。n 必须是 2 的整数次幂。

```
{
    int i,no2;
    DP mag2,tmp;

    int n=data.size();
    int m=respns.size();
    Vec_DP temp(n);
    temp[0]=respns[0];
    for (i=1;i<(m+1)/2;i++) {                    把respns取成长度为n的数组。
        temp[i]=respns[i];
        temp[n-i]=respns[m-i];
    }
    for (i=(m+1)/2;i<n-(m-1)/2;i++)              填充零元。
        temp[i]=0.0;
    for (i=0;i<n;i++)
        ans[i]=data[i];
    realft(ans,1);                               立刻进行两个函数的FFT。
    realft(temp,1);
    no2=n>>1;
    if (isign == 1) {
        for (i=2;i<n;i+=2) {                     FFT相乘以求卷积。
            tmp=ans[i];
            ans[i]=(ans[i]*temp[i]-ans[i+1]*temp[i+1])/no2;
            ans[i+1]=(ans[i+1]*temp[i]+tmp*temp[i+1])/no2;
        }
        ans[0]=ans[0]*temp[0]/no2;
        ans[1]=ans[1]*temp[1]/no2;
    } else if (isign == -1) {
        for (i=2;i<n;i+=2) {                     FFT相除以求卷积。
            if ((mag2=SQR(temp[i])+SQR(temp[i+1])) == 0.0)
                nrerror("Deconvolving at response zero in convlv");
            tmp=ans[i];
            ans[i]=(ans[i]*temp[i]+ans[i+1]*temp[i+1])/mag2/no2;
            ans[i+1]=(ans[i+1]*temp[i]-tmp*temp[i+1])/mag2/no2;
        }
        if (temp[0] == 0.0 || temp[1] == 0.0)
            nrerror("Deconvolving at response zero in convlv");
        ans[0]=ans[0]/temp[0]/no2;
        ans[1]=ans[1]/temp[1]/no2;
    } else nrerror("No meaning for isign in convlv");
    realft(ans,-1);                              求逆变换返回时域。
}
```

13.1.3 大型数据集的卷积和解卷积

如果数据集太长,不能一次全部装入内存,必须将它分成几段并且分别对每段做卷积。但是,这时终端效应的处理非常困难。不仅要担心有伪环绕效应,而且还要担心每段数据的终端肯定受到邻近的前面各段和后面各段的终端数据的影响。但是,因为每次机器内只有一段数据,所以这种影响不十分严重。

对于这个问题,有两个相关的、标准的解决方法。这两种方法都是显而易见的,因此这里不多叙述,读者应该能够自己去实现它们。第一种解决方法是**重叠存储法**。在这种技巧中,只需对数据刚开始部分填充足够的零元,以避免环绕污染。初始填充完毕后,应将所有的零元填充都忘掉。取入一段数据,并对它进行卷积或解卷积。然后丢弃两端上受环绕终端效应污染的点。只输出余下的中部的正常点。接着取入下一段数据,但不全是新数据。每一新段的前几个点与前一段数据的后面几个点重叠。各段必须充分地交叠,以便重新计算前一段的终端被污染的输出点,它是作为后一段的前几个未被污染的输出点。稍加考虑就可很容易地确定多少点要进行重叠和存储。

第二种解决方法称为**重叠相加法**,如图13.1.5所示。此方法不必重叠输入数据。数据的每一段彼此不相交,并且只使用一次。然而,应该仔细地在两端进行零元填充,才能使输出的卷积和解卷积中没有环绕引起的混乱性。然后再将各段的输出**叠加**起来。于是,靠近一段终端附近的输出点,将受到来自于适当地加在下一段起始端上的输入点的影响。同样,对靠近一段起始端附近的输出点也受到如此影响,只是在细节上要进行必要的修改。

即使在计算机内存够用时，将长数据集分段的方法也能使计算速度略有提高。这是因为 FFT 的 $N\log_2 N$ 次运算比线性 N 次运算稍微慢一些。然而，对数项的变化是如此缓慢，因此人们总是乐于避免重叠相加法和重叠存储法所引起的复杂操作。如果实际工作中需要计算长数据的卷积，还不如直接把全部数据集送入内存，再进行 FFT。这将在本章随后的各节中叙述。

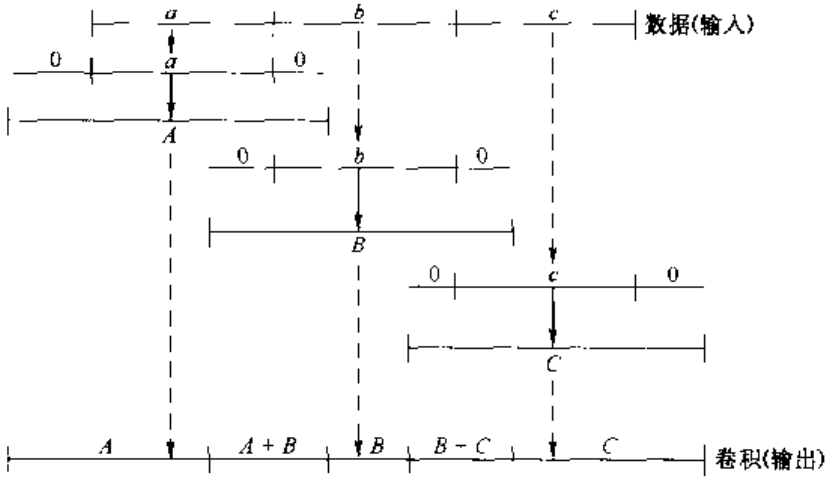


图 13.1.5　对非常长的信号与响应函数作卷积的重叠相加法。信号数据被分成若干小段，在每段的两端都填充零元，并作卷积(图中用粗线箭头表示)。最后，各段相加在一起，包括由零元填充而形成的重叠区域

参考文献和深入阅读

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 13.

13.2　使用 FFT 做相关和自相关

相关与卷积在数学上有着密切的关系。但是，在某些方面相关更简单一些，因为参与相关的两个函数，并不像参与卷积的数据和响应函数那样在概念上有相异之处。甚至在相关中，函数由不同但又通常是相似的数据集表示。通过直接叠加进行比较，来研究它们的“相关”，并且其中之一进行了左移或右移。

已经在(12.0.10)式中定义了两个连续函数 $g(t)$ 和 $h(t)$ 的相关，用 $\mathrm{Corr}(g,h)$ 表示，它是一个滞后 t 的函数。偶尔也用颇不方便的符号 $\mathrm{Corr}(g,h)(t)$ 来明确地表示这种函数的时间依赖性。如果第一个函数(g)是第二个函数(h)的近似的复制品，只是前者比后者延迟了某一时间 t，即第一个函数位移到第二个函数的右边，则在时间 t 上相关值很大。同样，在某些负值 t 上相关很大，只要第一个函数超前第二个函数 t，即位移到第二个函数的左边。当这两个函数交换后，则以下关系式成立：

$$\mathrm{Corr}(g,h)(t) = \mathrm{Corr}(h,g)(-t) \tag{13.2.1}$$

两个以 N 为周期的样本函数 g_k 和 h_k，其离散相关定义为

$$\mathrm{Corr}(g,h)_j \equiv \sum_{k=0}^{N-1} g_{j+k} h_k \tag{13.2.2}$$

离散相关定理表明两个实函数 g 和 h 的离散相关是离散傅里叶变换对中的一个：

$$\mathrm{Corr}(g,h)_j \Longleftrightarrow G_k H_k^* \tag{13.2.3}$$

其中 G_k 和 H_k 是 g_j 和 h_j 的离散傅里叶变换，星号($*$)表示复共轭。如同离散卷积定理一样，该定理中关于函数要做同样的假设。

能够用FFT计算相关,其方法如下:对两个数据集进行FFT,用一个变换结果乘以另一个变换的复共轭,然后再求乘积的逆变换。得出的结果(称为 r_k)形式上是长度为 N 的复向量。然而,因为原来两个数据集都是实型的,所以可以证明结果中所有的虚部都为零。r_k 的分量是不同滞后的相关值,正滞后和负滞后都按熟悉的环绕顺序存储:零滞后的相关是第一个分量 r_0;滞后1的相关是第二个分量 r_1;滞后-1的相关是最后一个分量 r_{N-1},如此等等。

如同卷积的情况一样,必须考虑终端效应,因为我们的数据通常不像相关定理所期望的那样是周期的。这样可以再次使用零元填充。如果对大滞后如 $\pm K$ 感兴趣,则必须要输入数据集的两端加上一个有 K 个零元的缓冲区。如果想要来自 N 个数据点的所有可能的滞后(不是件常见的事),则需要填充与数据相同个数的零元。这是一种极端情况,该程序如下:

```
#include "nr.h"

void NR::correl(Vec_I_DP &data1, Vec_I_DP &data2, Vec_O_DP &ans)
计算两个实型数据集 data1[0..n-1]和 data2[0..n-1]的相关(包括任何为用户提供的零元填充)。n 必须是 2 的整数次
幂。结果返回到 ans[0..n-1]中,并按环绕顺序存储,即逐次递增负滞后,其相关的存储地址从 ans[n-1]降到 ans[n/2];
而逐次递增正滞后,其相关的存储地址从 ans[0](零滞后)增加到 ans[n/2-1]。本程序的符号约定:如果 data1 滞后 da-
ta2,即 data1 位移到 data2 的右边,则 ans 将在正滞后呈现峰值。
{
    int no2,i;
    DP tmp;

    int n=data1.size();
    Vec_DP temp(n);
    for (i=0;i<n;i++) {
        ans[i]=data1[i];
        temp[i]=data2[i];
    }
    realft(ans,1);                          立刻变换两个data向量。
    realft(temp,1);
    no2=n>>1;                               逆FFT的归一化。
    for (i=2;i<n;i+=2) {                    相乘以求出它们相关的FFT。
        tmp=ans[i];
        ans[i]=(ans[i]*temp[i]+ans[i+1]*temp[i+1])/no2;
        ans[i+1]=(ans[i+1]*temp[i]-tmp*temp[i+1])/no2;
    }
    ans[0]=ans[0]*temp[0]/no2;
    ans[1]=ans[1]*temp[1]/no2;
    realft(ans,-1);                         求逆变换给出相关。
}
```

如同在程序 convlv 中那样,两次调用 realft 而不是一次调用 twofft,以使得当 data1 和 data2 的长度差别非常大时舍入误差最小。

一个样本函数 g_j 的**离散自相关**就是函数与其自身的离散相关。显然,就正、负滞后而言,它总是对称的。人们可以自由地应用以上程序 correl 以获得自相关,只要将两个自变量都用同一 data 向量简单地调用程序。如果效率低,则可用程序 realft 变换 data 向量。

参考文献和深入阅读

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §13-2.

13.3 具有FFT的最优(维纳)滤波

数值处理中有许多任务一般情况下都是采用傅里叶技巧处理的。其中之一就是对一个"讹误"信号中的噪声进行滤波。我们的情况是这样的:要测量一些潜在的、没有被讹误的信号 $u(t)$。但是,由于测量过程不完善,从测量设备得到的信号已是被讹误的信号 $c(t)$。信号 $c(t)$ 可能在两方面或其中一个方面不够完善。第一,仪器可能不具备完善的"δ 函数"响应,这样,真实信号被某已知函数 $r(t)$ 所卷积(被 $r(t)$ 涂改),从而得到一个涂改了的信号 $s(t)$,

$$s(t)=\int_{-\infty}^{\infty} r(t-\tau)u(\tau)\,d\tau \quad 或 \quad S(f)=R(f)U(f) \tag{13.3.1}$$

其中，S、R、U 分别是 s、r、u 的傅里叶变换。第二，所测的信号 $c(t)$ 可能包含了一个附加的噪声分量 $n(t)$，

$$c(t)=s(t)+n(t) \tag{13.3.2}$$

我们已经知道，在没有任何噪声的情况下如何对响应函数 r 的效果做解卷积（参见 13.1 节）。只需用 $R(f)$ 除 $C(f)$，就得到一个被解卷积的信号。而现在要在有噪声的情况下处理类似的问题。任务是寻找**最优滤波** $\phi(t)$ 或 $\Phi(f)$，将它们作用于已测信号 $c(t)$ 或 $C(f)$，然后用 $r(t)$ 或 $R(f)$ 与之做解卷积，产生一个信号 $\tilde{u}(t)$ 和 $\tilde{U}(f)$。换言之，将用下式来估计真实信号 U：

$$\tilde{U}(f)=\frac{C(f)\Phi(f)}{R(f)} \tag{13.3.3}$$

$\tilde{U}$ 怎么接近 U 呢？我们要求它以**最小二乘意义接近** U，即

$$\int_{-\infty}^{\infty}|\tilde{u}(t)-u(t)|^2\,dt=\int_{-\infty}^{\infty}\left|\tilde{U}(f)-U(f)\right|^2\,df \ 为最小 \tag{13.3.4}$$

代入(13.3.3)式和(13.3.2)式，(13.3.4)式右边变成

$$\begin{aligned}&\int_{-\infty}^{\infty}\left|\frac{[S(f)+N(f)]\Phi(f)}{R(f)}-\frac{S(f)}{R(f)}\right|^2\,df\\&=\int_{-\infty}^{\infty}|R(f)|^{-2}\left\{|S(f)|^2|1-\Phi(f)|^2+|N(f)|^2|\Phi(f)|^2\right\}\,df\end{aligned} \tag{13.3.5}$$

信号 S 和噪声 N 是不相关的，所以随频率 f 积分时，它们的叉积为零（这就是实际上对噪声的定义）。很显然，当且仅当在每个 f 值上被积函数对于 $\Phi(f)$ 为最小，则(13.3.5)式达到极小。让我们来寻找 $\Phi(f)$ 为实函数的这种解。对 Φ 求偏导，并设其结果为零，即得

$$\Phi(f)=\frac{|S(f)|^2}{|S(f)|^2+|N(f)|^2} \tag{13.3.6}$$

这就是最优滤波 $\Phi(f)$ 的公式。

注意，(13.3.6)式中包括已被涂改的信号 S 和噪声 N。这两者相加是已测信号 C。(13.3.6)式中不包含"真实"信号 U。由此得出一个最重要的简化：最优滤波的确定与 S 和 U 有关的解卷积函数的确定无关。

为了由(13.3.6)式确定最优滤波，需要一些分别估计 $|S|^2$ 和 $|N|^2$ 的方法。若想只用已测信号 C，而不用其他一些信息，或一些假设、猜想，是无法做到这点的。幸运的是，总能容易得到额外的信息。可以在长数据链 $c(t)$ 上取样，然后用(12.0.14)式、(12.1.8)式和(12.1.5)式绘出其功率谱密度的图形。这个量与 $|S|^2+|N|^2$ 成比例，即有

$$|S(f)|^2+|N(f)|^2\approx P_c(f)=|C(f)|^2 \qquad 0\leqslant f<f_c \tag{13.3.7}$$

（在 13.4 节和 13.7 节中，将讨论更多的估计功率谱密度的复杂方法，但是，上述估计方法总是足以适合解决最优过滤问题。）所得结果的图形（如图 13.3.1）经常立刻能显示出在连续噪声上伸展出的信号之谱特征。噪声谱可能是平坦的，或是倾斜的，或是光滑变化的。这都没有关系，只要我们能够根据它猜测出一个合理的假设。通过噪声谱画一条光滑曲线，将其外推到由信号支配的地方，再通过信号加噪声功率谱画一条光滑曲线。这两条曲线之差就是信号功率谱的光滑"模型"。信号功率谱模型与信号加功率谱模型之商就是最优滤波 $\Phi(f)$（用公式 $\Phi(-f)=\Phi(f)$ 将其扩展到 f 的负值）。注意，若噪声可忽略，$\Phi(f)$ 接近单位 1；在噪声占主导地位，$\Phi(f)$ 近似为零。这就是噪声如何起作用的。由(13.3.6)式给出的中间依赖关系，恰好就得出这两个极端之间的最优公式。

因为最优滤波是从求最小值问题中得出的，所以用最优滤波得出的结果与真实的最优值相差精度为二阶的数量级，最优滤波正是按这个精度确定的。换言之，即使是一个相当粗糙的被确定的最优滤波（例如，在 10% 的水平上），当应用于数据时，也能给出极好的结果。这就是为什么通常能从一个粗糙的功率谱密度图形中，用"肉眼"将已测得的信号 C 分离成信号分量 S 和噪声分量 N 的原因。这一切可能会想到对刚才描述

的过程进行迭代的问题。例如,用响应 $\Phi(f)$ 设计一个滤波器,并用它对信号进行合适的猜测 $\tilde{U}(f)=\Phi(f)C(f)/R(f)$ 以后,再回过来,把 $\tilde{U}(f)$ 看成是一个全新的信号,用同样的过滤技术对它进行估计,做进一步改进。不要将时间浪费在这种思路上。这种结构收敛到 $S(f)=0$ 的信号,的确存在有收敛的迭代方法,但上述方法不是其中的一种。

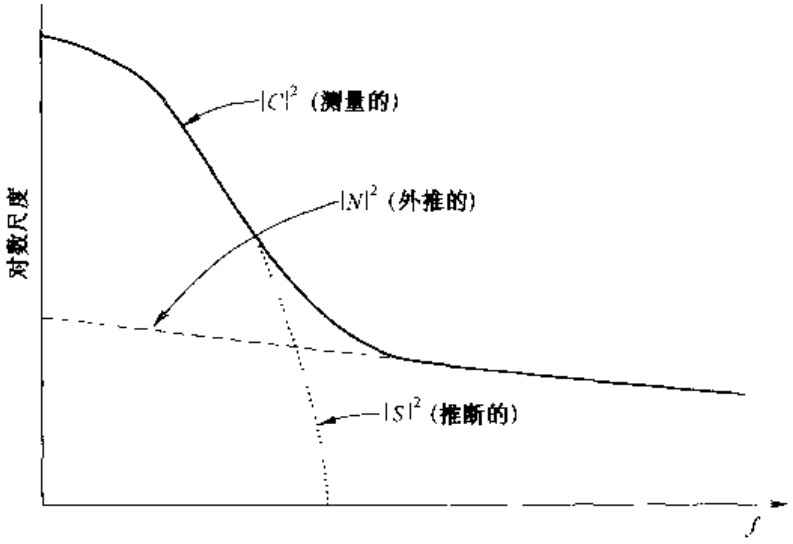


图 13.3.1 最优(维纳)滤波。信号加噪声的功率谱呈现了一个加上了噪声尾部的信号峰值。这个尾部作为"噪声模型"外推回到信号区域。做减法就得"信号模型"。对有用的方法模型不必是精确的。这些模型的简单代数组合就得到最优滤波(见正文)

当构造一个最优过滤器时,可以使用程序 four1(12.2 节)或 realft(12.3 节)对数据进行 FFT。可以使用 13.1 节中描述的方法对数据进行滤波。对于最优滤波,不需要专用程序 convlv,因为需要在频域开始处就构造滤波。但是,如果已经将数据与一已知响应函数做了解卷积,则可在其进行傅里叶逆变换之前,用最优滤波与它相乘,以此修改程序 convlv。

参考文献和深入阅读

Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

13.4 使用 FFT 做功率谱估计

在前一节,通过取函数 $c(t)$ 的某些有限取样范围的离散傅里叶变换的平方模,来"非正式地"估算了 $c(t)$ 的功率谱密度。在本节中,我们大致要做同样的事情,但是更加着重于细节,其结果将会出乎意料。

第一个细节是功率谱(亦称为功率谱密度或 PSD)的归一化。一般来说,函数的平方振幅的测量与 PSD 振幅的测量之间有某种比例关系。遗憾的是,在每一种域中,有几种不同描述归一化的约定,并且极易弄错这两种域之间的关系。假设函数 $c(t)$ 在 N 个点取样,产生值为 $c_0 \ldots c_{N-1}$,这些点横跨时间 T 的范围,即 $T=(N-1)\Delta$,其中 Δ 是取样间隔。于是,下面有几种关于总功率的描述:

$$\sum_{j=0}^{N-1}|c_j|^2 \equiv \text{"平方和振幅"} \tag{13.4.1}$$

$$\frac{1}{T}\int_0^T |c(t)|^2\ dt \approx \frac{1}{N}\sum_{j=0}^{N-1}|c_j|^2 \equiv \text{“均方振幅”} \tag{13.4.2}$$

$$\int_0^T |c(t)|^2\ dt \approx \Delta\sum_{j=0}^{N-1}|c_j|^2 \equiv \text{“时间积分平方振幅”} \tag{13.4.3}$$

正如我们将要看到的，PSD 估计器甚至有更多的变化。本节中考虑其中的一类，它对频率 f_i 的离散值进行估计，其中 i 取于整数值范围内。下一节中将学习不同的另一类估计器，即对频率 f 的连续函数进行估计。即便大家一致认为 PSD 的归一化与函数的归一化（例如(13.4.2)式）的特定描述是有关系的，但对 PSD 至少还有下述几种可能性：

- 定义在离散正频率、零频率和负频率上，并且在这些频率上的和是函数的均方振幅。
- 仅仅定义在零频率和正频率上，并且在此频率上之和是函数的均方振幅。
- 定义在从 $-f_c$ 到 f_c 的 Nyquist 区间上，并且在此范围内的积分是函数的均方振幅。
- 定义在 0 到 f_c 频率间，并且在此区域内积分是函数的均方振幅。

在 Nyquist 区间 $-f_c$ 到 f_c 以外对样本函数的 PSD 做积分是毫无意义的。因为根据取样定理，区间以外的功率已被混叠到 Nyquist 区间内。

定义足够多的符号用来区分所有可能的归一化组合是毫无希望的。因此，以后用记号 $P(f)$ 来代表上述 PSD 的任何一种定义，并在每一个情况中标出各个 $P(f)$ 是如何被归一化的。要注意文献中那些不统一的记号。

在前一节中使用的功率谱估计方法是一种简单的估计器方法，一般将这估计器称为**周期图**。如果在相等间隔上取函数 $c(t)$ 的 N 个样本点，并用 FFT 计算它的离散傅里叶变换

$$C_k = \sum_{j=0}^{N-1} c_j\ e^{2\pi ijk/N} \qquad k = 0,\ldots,N-1 \tag{13.4.4}$$

则定义在 $N/2+1$ 频率上的功率谱的周期图为

$$\begin{aligned} P(0) &= P(f_0) = \frac{1}{N^2}|C_0|^2 \\ P(f_k) &= \frac{1}{N^2}\left[|C_k|^2 + |C_{N-k}|^2\right] \qquad k = 1,2,\ldots,\left(\frac{N}{2}-1\right) \\ P(f_c) &= P(f_{N/2}) = \frac{1}{N^2}|C_{N/2}|^2 \end{aligned} \tag{13.4.5}$$

其中 f_k 只对零频率和正频率定义，

$$f_k \equiv \frac{k}{N\Delta} = 2f_c\frac{k}{N} \qquad k = 0,1,\ldots,\frac{N}{2} \tag{13.4.6}$$

根据 Parseval 定理，即(12.1.10)式，立即看出(13.4.5)式已被归一化，因此 P 的 $N/2+1$ 个值之总和等于函数 c_j 的均方振幅。

现在肯定要提这样一个问题，在什么意义下，周期图估计式(13.4.5)是潜在的函数 $c(t)$ 之功率谱的“真实”估计器。这可以在引用的文献中（参见本节参考文献[1]中的引言）找到详细的答案，摘要如下。

首先，周期图估计的**期望值**等于功率谱吗，也即估计器是正确的吗？既对又不对。我们确实不会期望 $P(f_k)$ 之一在 f_k 处确切地等于连续的 $P(f)$ 值，因为 f_k 被假设成是整个频率“箱”内的代表，它是从前一个离散频率的中途延伸到下一个离散频率的中途。但是，应该期望 $P(f_k)$ 是在以 f_k 为中心的狭窗函数内 $P(f)$ 的某种类型的平均值。对于周期图估计式(13.4.6)，作为箱中频率偏移 s 的函数的窗函数是

$$W(s) = \frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2 \tag{13.4.7}$$

注意 $W(s)$ 是振荡波瓣，除此以外，它大约以 $W(s)\approx(\pi s)^{-2}$ 衰减。这不是一个快速衰减，其导致周期图的估计从一个频率到另一个频率的重要**渗漏**（这是一个技术术语）。还要注意，当 s 为非零整数时，$W(s)$ 恰好为零。这意味着如果函数 $c(t)$ 正好等于 f_k 中某一个频率的纯正弦波，则相邻 f_k 之间就不会有渗漏。但这不是

典型的情况。如果频率在两个相邻 f_k 之间的 1/3 处,则渗漏将大大超过这两个相邻的箱。解决渗漏问题的方法称为**数据窗**,后面将对其进行讨论。

现在讨论关于周期图估计的另一个问题:随着 N 趋于无穷时估计的方差有多大?换言之,当随着原始函数取样点的增加(或者以相同的取样率取样一个更长的数据,或者以更快的取样率对相同的数据重新取样),估计 P_k 的正确性将增加多少?答案是令人失望的,周期图估计根本不会变得更精确。事实上,频率 f_k 处的周期图估计的方差始终等于在该频率处期望值的平方。换言之,标准差永远是该值的 100%,不依赖于 N。增加点后的所有信息都参与到更大量的离散频率 f_k 的估计过程。如果用同样的取样率对一个更长的数据取样,则 Nyquist 临界频率 f_c 是不变的,不过现在在 Nyquist 频率区内有更细的频率精度(更多的 f_k);另外,若用更细的取样间隔对同样长度的数据取样,则频率精度不变,但现在 Nyquist 区域扩展至更高的频率。在这两种情况下,增加样本都不会减少任一特定频率估计的 PSD 的方差。

不管怎样,不必承受百分之百的标准方差的 PSD 估计。只要知道几种减少估计方差的技巧就可以了。这里有两种技巧,它们在数学上几乎等同,但是执行过程不同。第一种技巧是,采用比实际所需还要更细一些的离散频率间隔进行取样并计算周期图估计,然后对 K 个相邻的离散频率上求周期图估计的和,以得到在这样 K 个频率的中间频率上的一个"更光滑"的估计。这个和估计的方差比其估计本身小一个 $1/K$ 因子,即标准差比100%小一个 $1/\sqrt{K}$ 因子。因此,为了对 0 和 f_c(包括 0 和 f_c)之间 $M+1$ 个离散频率上估计功率谱,开始时需要 $2MK$ 点的 FFT($2MK$ 最好是 2 的整数次幂)。然后取结果系数的模平方,将正的和负的频率对加起来,再除以 $(2MK)^2$,所有这些都是按照(13.4.5)式并取 $N=2MK$ 进行的。最后,将结果"装箱",使之成为 K 的求和组(不是平均组)。这个过程很容易编程,因此不必为没给出程序而烦恼。对 K 个相邻点求和而不是求平均,其原因是为了使最终 PSD 估算将保持其 $M+1$ 个值的和等于函数均方振幅的归一化特性。

估计 $M+1$ 个频率上的 PSD 的第二种技巧是,在区间 0 到 f_c 之间,将原始样本数据划分成 K 个小段,每段有 $2M$ 个连续的样本点。每段分别做 FFT,以得到一个周期图估计((13.4.5)式且取 $N=2M$)。最后,在每个频率上对 K 个周期图估计求平均。正是最后一步平均,使估计的方差降低了一个 K 因子(标准差降低了 $\sqrt{K}$ 因子)。从计算上看,第二种技巧比第一种技巧更为有效,因为对较短样本点做 FFT 与对较长样本点做 FFT 进行比较,其效率按对数量级提高。第二种技巧的主要优点还在于,在一段时间内只处理 $2M$ 个数据点,而不像第一种技巧中是处理 $2KM$ 个数据点。这意味着,对来自磁带或其他数据记录中的长数据的处理,第二种技巧是一种自然的选择。后面将给出实现第二种技巧的程序。但是,还需首先返回到渗漏和由(13.4.7)式引出的数据开窗问题。

13.4.1 数据开窗

数据开窗之目的在于修正(13.4.7)式,它表示一个离散频率的谱估计 P_k,与相近频率上实际的、潜在的连续谱 $P(f)$ 之间的关系。一般来说,一个"箱" k 内的功率谱实际上包括着其他 s 箱的频率分量所带来的渗漏,其中 s 是(13.4.7)式中的自变量。正如我们所指出的,甚至从适当大的 s 值,也有大量的渗漏。

当为周期谱估计选择一组 N 个样本点时,实际上已用了一个时间的窗函数乘上一个无穷长的样本数据 c_j,这个窗函数在整个取样时间 $N\Delta$ 内等于单位值,而其他时间取值为零。换言之,数据被一个矩形窗函数开了窗。根据卷积定理((12.0.9)式,但是互换 f 和 t 的位置),数据与这个矩形窗函数乘积的傅里叶变换,应该等于数据的傅里叶变换与窗函数的傅里叶变换的卷积。事实上,我们确定的(13.4.7)式,即为单位窗函数的离散傅里叶变换的平方:

$$W(s)=\frac{1}{N^2}\left[\frac{\sin(\pi s)}{\sin(\pi s/N)}\right]^2=\frac{1}{N^2}\left|\sum_{k=0}^{N-1}e^{2\pi isk/N}\right|^2 \tag{13.4.8}$$

在 s 值较大处,发生渗漏的原因在于,矩形窗函数接通和断开太迅速,它的傅里叶变换在高频处有大量的分量。为了改善这种情况,用窗函数乘以输入数据 $c_j(j=0,\ldots,N-1)$,使窗函数 w_j 随 j 从 0 至 N 变化而从零逐渐升至最大值,然后再降回到零。在这种情况下,周期图估计器((13.4.4)式~(13.4.5)式)成为

$$D_k\equiv\sum_{j=0}^{N-1}c_jw_j\,e^{2\pi ijk/N}\qquad k=0,\ldots,N-1 \tag{13.4.9}$$

$$
\begin{aligned}
P(0) &= P(f_0) = \frac{1}{W_{ss}}|D_0|^2 \\
P(f_k) &= \frac{1}{W_{ss}}\left[|D_k|^2 + |D_{N-k}|^2\right] \qquad k = 1, 2, \ldots, \left(\frac{N}{2} - 1\right) \\
P(f_c) &= P(f_{N/2}) = \frac{1}{W_{ss}}|D_{N/2}|^2
\end{aligned}
\tag{13.4.10}
$$

其中 W_{ss} 代表窗"平方求和"，

$$
W_{ss} \equiv N \sum_{j=0}^{N-1} w_j^2 \tag{13.4.11}
$$

其中 f_k 由(13.4.6)式给出。(13.4.7)式更一般的形式现在可以根据窗函数 w_j 写成

$$
\begin{aligned}
W(s) &= \frac{1}{W_{ss}}\left|\sum_{k=0}^{N-1} e^{2\pi i s k/N} w_k\right|^2 \\
&\approx \frac{1}{W_{ss}}\left|\int_{-N/2}^{N/2} \cos(2\pi s k/N) w(k - N/2)\, dk\right|^2
\end{aligned}
\tag{13.4.12}
$$

此处的近似式对实际估计是很有用的，而且适用于任何左右对称的窗(通常的情况)和对 $s \ll N$ 的情况(这是估计渗漏进入邻近箱的情况)。积分中的连续函数 $w(k - N/2)$ 是通过点 w_k 的某光滑函数。

关于选择窗函数，也许有许多不必要的经验，实践中，每个从零开始上升到顶峰然后又下降的函数都是以某个人的名字命名。常用的几个如下所示(参见图 13.4.1)：

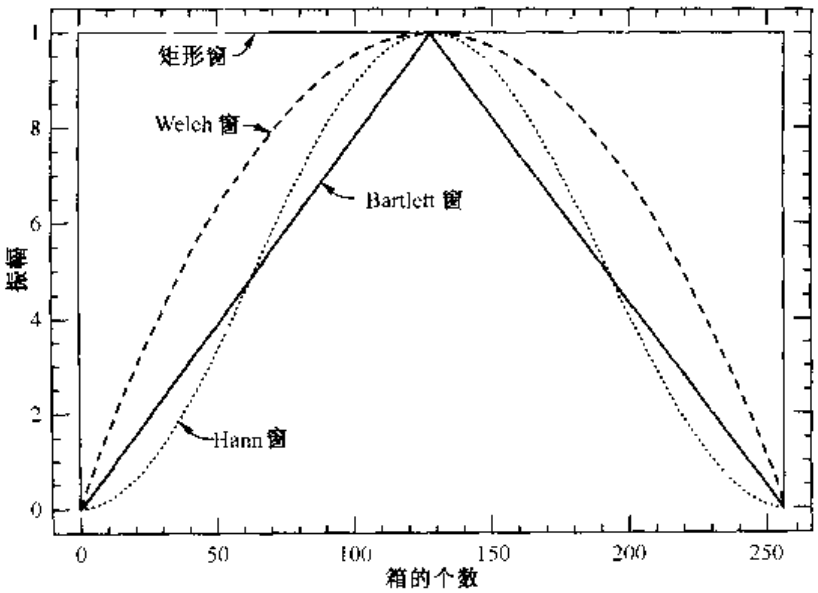


图 13.4.1　通常用于 FFT 功率谱的窗函数。数据段的长度为 256，在 FFT 计算之前，将数据段乘以窗函数(依此逐个箱)，矩形窗等价于没有开窗，不值得推荐。Welch窗和Bartlett窗是很好的选择

$$
w_j = 1 - \left|\frac{j - \frac{1}{2}N}{\frac{1}{2}N}\right| \equiv \text{“Bartlett 窗”} \tag{13.4.13}
$$

("Parzen 窗"与此非常类似。)

$$
w_j = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi j}{N}\right)\right] \equiv \text{“Hann 窗”} \tag{13.4.14}
$$

(汉明(Hamming)窗是类似的，但其在端部不会准确地到达零。)

$$
w_j = 1 - \left(\frac{j - \frac{1}{2}N}{\frac{1}{2}N}\right)^2 \equiv \text{“Welch 窗”} \tag{13.4.15}
$$

倾向于遵循 Welch,推荐在实际工作中使用(13.4.13)式或者(13.4.15)式。然而,以本书的水平,任何这些(或类似的)窗函数实际上没有区别。它们之间的差别在于各种有价值的标准图之间的微妙权衡,这些标准图形可以用来描述由(13.4.12)式计算出的谱渗漏函数的狭窄性和有峰性。这些标准图形还有一些名称:**最高旁瓣(dB)**、**旁瓣衰减(每个倍频程的 dB)**、**等价噪声带宽(箱)**、**3-dB 带宽(箱)**、**扇状损失(dB)**、**最坏情况过程损失(dB)**。大概地说,是在形成尽可能狭窄的中间峰值区和尽可能加快尾部的衰减两者之间进行主要的权衡。详情参阅本节参考文献[2]。图 13.4.2 中绘出了已讨论过的几种窗的渗漏振幅图。

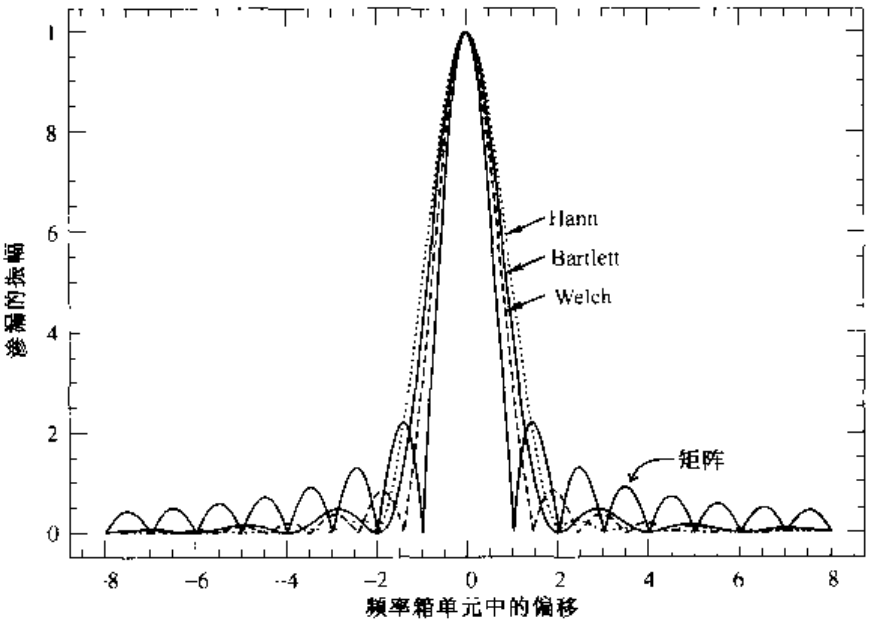


图 13.4.2 图 13.4.1 中窗函数的渗漏函数 实际上频率位于零偏移处的一个信号"渗漏"到具有图示振幅的邻近箱中。开窗的目的是为了减少在大频率偏移处的渗漏,而矩形窗(不开窗)具有大的旁瓣。偏移可以是分数值,因为实际信号频率可能位于FFT的两个频率箱之间

关于窗函数,还有一些经验知识,在数据的前一小部分处(设 10%)窗函数从零平滑地上升到单位值,然后窗函数停留在单位值,直到数据最后的一小部分(又假设为 10%),在这期间函数又从单位值平滑地下降到零。这个窗函数虽然使渗漏函数的主瓣变窄一点(永不会如 2 的倍数那么多),但将其渗漏尾部拓宽一个明显的因子(例如,10%的倒数,一个 10 的因子)。假如能将窗的宽度(在窗的极大值处的样本个数)和窗的上升(下降)时间(在其上升与下降过程中的样本数)区分开来;如果又能将渗漏函数主瓣的 FWHM(半最大值的全宽度)和渗透宽度(包含谱功率一半的全宽度,这些功率谱不包含在主瓣内)区分开来,则这些量大致地联系起来为

$$(\text{箱中的FWHM}) \approx \frac{N}{(\text{窗宽})} \tag{13.4.16}$$

$$(\text{箱中的渗漏宽度}) \approx \frac{N}{(\text{窗上升/下降时间})} \tag{13.4.17}$$

对于前面(13.4.13)式~(13.4.15)式给出的窗,有效窗宽和有效上升(下降)时间都是$\frac{1}{2}N$阶的。通常而言,当窗的上升和下降时间只是数据长度的很小部分,这种窗优点很小或没有时,我们避免使用它们。有时可能会听说,平顶的窗"损失较少数据",下面介绍一个更好的办法来解决这个问题,即用重叠数据段的方法。

假设选择了一个窗口函数,并准备把数据划分为有 $N=2M$ 个点的 K 段。对每段做 FFT,并将得到的 K 个周期图一起求平均,以获得 0 到 f_c 间的 $M+1$ 个频率值上的 PSD。现在必须区分两种可能的情况。可能想从固定的计算总数中得到最小方差,而不必须考虑所用数据点的个数。例如,通常当数据正在实时地聚集,并且数据的简化受计算机控制时,这种情况便成为目标。另一种可能情况是,可能想从有效的样本数据点中得到最小方差。通常,如果数据已经记录下来,并记录之后正在分析,则这便成为目标。

在第一种情况下(每次计算操作的最小谱方差),最好不交叠而将数据分段。前 $2M$ 数据点构成 1 号段,

其次 $2M$ 个数据点构成 2 号段，依次类推，直至 K 号段，共有 $2KM$ 个样本点。这种情况下的方差和单个段有关，方差减少到 $1/K$。

第二种情况下（每个数据点的最小谱方差），若将各段重叠其长度的一半，则所得的是最优的或非常接近最优的。第一组 M 个点和第二组 M 个点为 1 号段；第二组 M 个点和第三组 M 个点为 2 号段；依次类推，直至 K 号段，它由第 K 组 M 个点和 $K+1$ 组 M 个点组成。从而样本点总数是 $(K+1)M$，正好是没有用重叠段的总数的一半。方差的减少不是 $1/K$ 的整数倍，因为各段不是统计独立的。可以证明，方差大约减少 $9K/11$（见 Welch 的文章，参见本节参考文献[3]）。无论怎样，这比减少 $K/2$ 要好；如果没有重叠，用同样数量进行分段，这就降低 $K/2$。

现在将这些想法编成一个谱估计程序。尽管我们一般避免输入（输出）编码，但这里例外，将呈现如何通过一个数据文件顺序读出数据（参见参数 ifstream &fp）。在任一时刻，只有一小部分数据在内存中。注意，spctrm 返回在 M 个频率点（而不是 $M+1$ 个频率点）上的功率，因为省去了在临界估计下的分量 $P(f_c)$，它可以直接包括进来。

```
#include <fstream>
#include <cmath>
#include "nr.h"
using namespace std;

namespace {
    inline DP window(const int j, const DP a, const DP b)
    {
        return 1.0-fabs((j-a)*b); // Bartlett
        // return 1.0; // Square
        // return 1.0-SQR((j-a)*b); // Welch
    }
}

void NR::spctrm(ifstream &fp, Vec_O_DP &p, const int k, const bool ovrlap)
由文件指针 fp 指定的输入流中读出数据，并且将每个网格点频率 j/(2*m)周期处的数据之功率谱（均方振幅）返回到
p[j](j=0,1,...,m-1)；如果 vrlap 是 true，这些网格点是基于(2*k+1)*m 个数据点；或者如果 ovrlap 是 false，则基于
4*k*m个数据点。这两种情况数据段的个数都是 2*k；程序调用 four1 k 次，每次调用有两个部分，每部分有2*m 个实数
据点。
{
    int mm,m4,kk,joffn,joff,j2,j;
    DP w,facp,facm,sumw=0.0,den=0.0;

    int m=p.size();
    mm=m << 1;                                  有用的因式。
    m4=mm << 1;
    Vec_DP w1(m4),w2(m);
    facm=m;                                     窗函数要用到的因子。
    facp=1.0/m;
    for (j=0;j<mm;j++)                          求权重的平方和。
        sumw += SQR(window(j,facm,facp));
    for (j=0;j<m;j++) p[j]=0.0;                 将谱初始赋值为零。
    if (ovrlap)                                 对“保存”的半缓冲区初始赋值。
        for (j=0;j<m;j++)
            fp >> w2[j];
    for (kk=0;kk<k;kk++) {
    对两组数据集进行循环。
        for (joff=0;joff<2;joff++) {            两个完整段进入工作区。
            if (ovrlap) {
                for (j=0;j<m;j++) w1[joff+j+j]=w2[j];
                for (j=0;j<m;j++)
                    fp >> w2[j];
                joffn=joff+mm-1;
                for (j=0;j<m;j++) w1[joffn+j+j+1]=w2[j];
            } else {
                for (j=joff;j<m4;j+=2)
```

```
                    fp >> w1[j];
                }
            }
            for (j=0;j<mm;j++) {                    将窗应用于数据。
                j2=j+j;
                w=window(j,facm,facp);
                w1[j2+1] *= w;
                w1[j2] *= w;
            }
            four1(w1,1);                            对开窗的数据做FFT变换。
            p[0] += (SQR(w1[0])+SQR(w1[1]));        结果之和进入原先的段。
            for (j=1;j<m;j++) {
                j2=j+j;
                p[j] += (SQR(w1[j2+1])+SQR(w1[j2])
                    +SQR(w1[m4-j2+1])+SQR(w1[m4-j2]));
            }
            den += sumw;
        }
        den *= m4;                                  进行归一化。
        for (j=0;j<m;j++) p[j] /= den;              输出归一化。
    }
```

参考文献和深入阅读

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Harris, F.J. 1978, *Proceedings of the IEEE*, vol. 66, pp. 51–83. [2]

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), paper by P.D. Welch. [3]

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley).

Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

13.5 深入讨论:时域中的数字滤波

假设有一个信号要进行数字滤波。例如,也许会运用**高通**或**低通**滤波,分别滤去低频或高频噪声;也许只处理位于某个频带内的信号,则需要**带通滤波器**。或者如果测量中混杂有 60 Hz 电源线干扰,就可能需要一个**槽型滤波器**,只除去围绕该频率的狭窄带状。这一节特别阐述读者在时域中已经选择的这类滤波的情形。

继续讨论前,希望读者再考虑一下这种选择。回忆一下,在傅里叶域中滤波是非常方便的。只需记录全部数据,对其进行 FFT,将 FFT 的输出乘以一个滤波函数$\mathcal{H}(f)$,然后再做逆 FFT,得到一个返回时域中已滤波的数据集。下面是一些傅里叶方法的附加背景知识。

- 请记住,必须为正、负频率定义滤波函数$\mathcal{H}(f)$,而且频率端部的量值始终是 Nyquist 频率 $1/2\Delta$,其中 Δ 是取样区间。FFT 中最小非零频率的数量值应为 $\pm 1/(N\Delta)$,其中 N 是 FFT 中(复型)点的个数。这个滤波的正、负频率按环绕顺序排列。
- 如果已测数据是实数,而且希望滤波输出也是实数,则任意滤波函数都应遵循$\mathcal{H}(-f)=\mathcal{H}(f)^*$。可以极容易地实现这点,只需取一个关于 f 是实且为偶的$\mathcal{H}$。
- 如果选择的$\mathcal{H}(f)$有锐利垂直的边界,则滤波器的脉冲响应(输入一个短脉冲引起的输出)在边界对应频率处成"环形"衰减。这并没有出错,但如果不喜欢,那么就选择一个更平滑的$\mathcal{H}(f)$。为了直接得到关于滤波器脉冲响应的第一手材料,可以取$\mathcal{H}(f)$的逆 FFT。如果在某些 k 个点上对滤波函数所有的边界做了光滑处理,则滤波器的脉冲响应函数将有一个跨度为整个数据记录的 $1/k$ 的间距。
- 如果数据太长,不能一次进行 FFT,则需将它们分成适当长度的若干段,只要每段长度比滤波器的脉冲响应函数长得多就行。必要时,使用零元填充。

- 可能需要从数据中减去一条通过第一点和最后一点的一条直线(即,使第一点和最后一点为零),从而除掉来自数据的任何趋势。如果正在进行数据划分成段,则可以选取重叠的段,并只使用每段的中间部分,就能**较好地**远离边缘效应。
- 如果一个数字滤波器的某个特定时间步的输出仅仅依赖于这个时间步的输入或者更早时间步的输入,则称这个滤波器为因果性的或物理可实现的。如果它的输出既依赖于早先的输入,又依赖于以后的输入,则称它为无因果性的。一般情况,傅里叶域中的滤波器是无因果性的,因为数据是"成批"进行处理的,并不考虑时间顺序。通常无因果性的滤波器能带来高级的性能(例如,较少的相位色散、更尖锐的边缘、脉冲响应函数的不对称性减少)。人们使用因果性的滤波器不是因为它们更好些,而是因为在一些场合下不允许存取非时序的数据。原则上讲,时间滤波器既可为因果性的,也可为无因果性的,但是它们通常被用于物理可实现性受限的应用中。基于此,在以下内容中我们限定于有因果性的情况。

如果读者仍然偏爱时域滤波,这大概因为你有一个实时的应用问题。这个问题需要处理一串连续的数据流,而且希望按照接收原始数据的相同的速率输出滤波值。否则,可能是待处理的数据量太大,以至于只能在每个数据点上承受非常小量的浮点运算,而且甚至不能进行中型规模大小的 FFT(每个数据点的浮点运算次数是数据集或段中点的个数的对数值的几倍)。

13.5.1　线性滤波器

最常见的线性滤波器其输入点序列为 x_k,并由下述公式产生输出点序列 y_n:

$$y_n = \sum_{k=0}^{M} c_k\, x_{n-k} + \sum_{j=0}^{N-1} d_j\, y_{n-j-1} \tag{13.5.1}$$

其中 $M+1$ 个系数 c_k 和 N 个系数 d_j 都是固定的,并定义了滤波器的响应。滤波器(13.5.1)式是由当前输入值和先前 M 个输入值,以及它自己先前 N 个输出值产生每个新的输出值。如果 $N=0$,(13.5.1)式中第二项不存在,则滤波称为**非递推滤波或有限脉冲响应滤波(FIR)**。如果 $N\neq 0$,则滤波称为**递归的或无限脉冲响应的(IIR)**(术语"IIR"仅仅意味着这种滤波器能够具有无限长的脉冲响应,而不是说在特殊的应用中脉冲响应必定很长。典型的情况是,一个 IIR 滤波器的响应在以后时间里将成指数衰减,迅速衰减以至于可以忽略不计)。

c_k 和 d_j 与滤波的响应函数$\mathcal{H}(f)$的关系为

$$\mathcal{H}(f) = \frac{\sum_{k=0}^{M} c_k e^{-2\pi i k(f\Delta)}}{1 - \sum_{j=0}^{N-1} d_j e^{-2\pi i (j+1)(f\Delta)}} \tag{13.5.2}$$

其中 Δ 仍为取样间隔。Nyquist 区间对应于 $-1/2$ 至 $1/2$ 之间的 $f\Delta$。对 FIR 滤波来说,(13.5.2)式的分母恰好是单位 1。

(13.5.2)式告诉了我们如何由一系列 c 和 d 值确定$\mathcal{H}(f)$。然而,为了设计滤波器,需要一个进行逆变换的方法,即从期望的$\mathcal{H}(f)$中获得一个适当的 c 和 d 的集合,这个集要尽可能地小,以使计算量最小。有许多书以此为专题进行了讨论。像其他许多"逆问题"一样,它没有完美的解决方法。因为$\mathcal{H}(f)$是一个完全连续的函数,而小段的 c 和 d 只表示一些可调整的参数。因此必须折衷一下。数字滤波器设计的问题关系到进行这种折衷的各类方法。无法给出一种彻底的处理方法,但是我们可以概述两个基本技巧促使这一工作可以顺利开始。进一步详细讨论,请查阅某些专业书籍。

13.5.2　FIR(非递归)滤波

当(13.5.2)式中分母为单位 1 时,右端恰好是离散傅里叶变换。变换很容易求逆,根据在某些离散频率 f_i 上$\mathcal{H}(f_i)$的少量几个值,能求得同样少量几个欲求的 c_k 系数。然而,这一事实并非十分有用。因为,按这种方法计算出的 c_k 值,$\mathcal{H}(f)$将在离散频率间趋于剧烈地振荡,而在这些离散频率上取特定值。

作为文献中几种正规方法的基础的较好策略是:开始时,假设要极力找到相对大量的滤波系数,即相当大的 M 值。然后,$\mathcal{H}(f)$可以在相当细的网格上固定为所期望的值,并且 M 个系数 $c_k(k=0,\ldots,M-1)$可由 FFT 找到。下一步将大部分 c_k 截断(置为零),留下非零 c_k,例如前 K 个系数($c_0,c_1,\ldots,c_{K-1}$)和最后 $K-1$ 个系数($c_{M-K+1},\ldots,c_{M-1}$)。因为 FFT 的环绕性质,最后少数几个系数 c_k 是负滞后的滤波系数,但是我们并不要负滞后的系数,所以循环地位移 c_k 数组,以使得每个系数都成为正滞后(这相应于将时间延迟引入滤波器)。要做到这一点,可按以下顺序将 c_k 系数复制成一组长度为 M 的新数组:

$$(c_{M-K+1},\ldots,c_{M-1},\ c_0,\ c_1,\ldots,c_{K-1},\ 0,\ 0,\ldots,0) \tag{13.5.3}$$

为了观察所做的截断是否被接受,可以对(13.5.3)式的数组做 FFT,给出原来$\mathcal{H}(f)$的近似值。一般情况下,将模$|\mathcal{H}(f)|$与原来的函数进行比较,因为时间延迟已将复相位引入滤波响应中了。

如果新的滤波函数是可被接受的,则已进行完毕,并且有了一组 $2K-1$ 个滤波系数。如果它不可被接受,则既可以增大 K 再试一次,又可以对同样的 K 进行一些更仔细的工作以提高其可接受性。一个有关更仔细工作的例子就是修改不可接受的$\mathcal{H}(f)$的振幅(而不是相位),以使它与理想的情况更一致,然后进行 FFT 得到新的 c_k。再次全置零除了前 $2K-1$ 个值(不需要循环位移,因为已保护了时间延迟的相位),然后求逆变换以得到新的$\mathcal{H}(f)$,它往往是可被接受的。可以对此过程进行迭代。请注意,如果对接受性的要求过于苛刻,超过 $2K-1$ 个系数所能操纵的范围,则这个迭代过程不收敛。

换言之,关键思想是在于,在系数空间与函数$\mathcal{H}(f)$空间两者之间进行迭代,直到找到一对傅里叶共轭对,它们在两空间中都满足所强加的限制为止。这种迭代的一种更正规的技术是 **Remes 交换算法**,该算法对具有固定个数滤波系数的所期望的频率响应将产生最优的切比雪夫 Chebyshev 逼近(参见 5.13 节)。

13.5.3 IIR(递归)滤波器

递归滤波在某一给定的时间上,其输出既取决于当前输入和以前的输入,也取决于以前的输出,通常它比具有相同数量系数(或对每个输入点有相同数量的浮点操作)的非递归滤波更有优势。通过观察(13.5.2)式,理由是非常清楚的:非递归滤波具有一个以 $1/z$ 为变量的多项式的频率响应,其中

$$z \equiv e^{2\pi i(f\Delta)} \tag{13.5.4}$$

相比较,递归滤波的频率响应是变量 $1/z$ 的**有理函数**。而有理函数对于拟合具有锐利边缘或狭窄特征的函数非常好,并且大多数期望的滤波函数都是这种类型的。

非递归滤波永远是稳定的。如果将输入的新 x_i 序列断开,则不超过 M 步后,由(13.5.1)式产生的 y_j 序列也会断开。而递归滤波,如它们在进行输出时又反馈给自己那样,不必是稳定的。如果系数 d_j 选得很差,则递归滤波就可能成指数增长的方式——所谓**齐次的**,即使输入序列已经断开,输出序列还变得很大。这是很不好的。所以,设计递归滤波的问题并不仅仅是一个逆问题,这还是一个具有附加稳定性的限制的逆问题。

对一组给定的 c_k 和 d_j 系数,如何得知(13.5.1)式的滤波器是稳定的呢?稳定性仅仅取决于 d_j。当且仅当特征多项式

$$z^N - \sum_{j=0}^{N-1} d_j z^{(N-1)-j} = 0 \tag{13.5.5}$$

的所有 N 个复根都在单位圆内时,滤波器是稳定的,即满足

$$|z| \leq 1 \tag{13.5.6}$$

构造稳定递归滤波器的各种方法形成了一个学科领域,这需要参阅更多的专业书籍。然而,有一个非常有用的技术是**双线性变换方法**。关于这一专题,定义一个新变量 w,它将频率 f 重新参数化,

$$w \equiv \tan[\pi(f\Delta)] = i\left(\frac{1-e^{2\pi i(f\Delta)}}{1+e^{2\pi i(f\Delta)}}\right) = i\left(\frac{1-z}{1+z}\right) \tag{13.5.7}$$

不要被(13.5.7)式中的 i 所迷惑。这一等式把实频率 f 映射成实值 w。事实上,它把 Nyquist 区间 $-\frac{1}{2}<f\Delta<\frac{1}{2}$ 映射成实 w 轴 $-\infty<w<+\infty$。(13.5.7)式的逆等式为

$$z = e^{2\pi i(f\Delta)} = \frac{1+iw}{1-iw} \tag{13.5.8}$$

在重新参数化 f 时，当然 u 也再次参数化 z。所以，稳定的条件(13.5.5)式和(13.5.6)式可按 w 重新描述：如果滤波响应$\mathcal{H}(f)$写成 w 的函数，则滤波是稳定的，当且仅当滤波函数的极点(其分母的零点)都在上半复平面时，即

$$\mathrm{Im}(w) \geq 0 \tag{13.5.9}$$

双线性变换方法的思路是，只需指定所期望的滤波函数的模平方 $|\mathcal{H}(f)|^2 = \mathcal{H}(f)\ \mathcal{H}(f)^* = \mathcal{H}(f)\ \mathcal{H}(-f)$，来代替指定所期望的$\mathcal{H}(f)$。选取出一些 w^2 的有理函数逼近的函数，然后在 w 复平面上找出这个函数的所有极点。由于对称性，下半平面的每个极点都在上半平面上有一个对应的极点。其思路是得到只是以上半平面的较好极点为因子的乘积。这个乘积就是**稳定可实现**的$\mathcal{H}(f)$。现在，代入(13.5.7)式将函数写成为 z 的有理函数，并与(13.5.2)式比较，得到系数 c 和 d。

通过下面例子的讨论，这一过程会更清楚。假设要设计一个简单的带通滤波器，其低截止频率对应于值 $w = a$，而高截止频率对应于 $w = b$ 值，a 和 b 均为正数。能够完成这个要求的一个简单有理函数是

$$|\mathcal{H}(f)|^2 = \left(\frac{w^2}{w^2+a^2}\right)\left(\frac{b^2}{w^2+b^2}\right) \tag{13.5.10}$$

这个函数并没有非常突变的截止，但它阐明了更一般的情况。为得到更陡峭的边缘，可以取函数(13.5.10)式的某个正整数次幂，或等价地，通过从(13.5.10)式中得到的滤波器的某些复制，来连续地运行数据。

(13.5.10)式的极点明显地都在 $w = \pm ia$ 和 $w = \pm ib$ 上。所以稳定可实现的$\mathcal{H}(f)$是

$$\mathcal{H}(f) = \left(\frac{w}{w-ia}\right)\left(\frac{ib}{w-ib}\right) = \frac{\left(\frac{1-z}{1+z}\right)b}{\left[\left(\frac{1-z}{1+z}\right)-a\right]\left[\left(\frac{1-z}{1+z}\right)-b\right]} \tag{13.5.11}$$

把 i 放入第二个因式的分子中，为的是能得到实值的系数。如果将分母乘出来，(13.5.11)式可写成

$$\mathcal{H}(f) = \frac{-\frac{b}{(1+a)(1+b)} + \frac{b}{(1+a)(1+b)}z^{-2}}{1 - \frac{(1+a)(1-b)+(1-a)(1+b)}{(1+a)(1+b)}z^{-1} + \frac{(1-a)(1-b)}{(1+a)(1+b)}z^{-2}} \tag{13.5.12}$$

由此，可以得到(13.5.1)式的滤波系数，

$$\begin{aligned}
c_0 &= -\frac{b}{(1+a)(1+b)} \\
c_1 &= 0 \\
c_2 &= \frac{b}{(1+a)(1+b)} \\
d_0 &= \frac{(1+a)(1-b)+(1-a)(1+b)}{(1+a)(1+b)} \\
d_1 &= -\frac{(1-a)(1-b)}{(1+a)(1+b)}
\end{aligned} \tag{13.5.13}$$

这就完成了带通滤波器的设计。

有时，设计者能够想出怎样对$\mathcal{H}(f)$直接构造关于 w 的有理函数，而不必从它的模平方开始进行设计。但为了稳定性，构造的函数只能在上半平面上设有极点。若用 $-w$ 代替 w，则函数不应该具有变成其自身复共轭的特性，所以滤波系数将是实型的。

例如，下面是槽形滤波函数，它被设计成仅去掉围绕某个可靠频率 $w = w_0$ 的窄频率带，其中 w_0 是个正数，

$$\begin{aligned}
\mathcal{H}(f) &= \left(\frac{w-w_0}{w-w_0-i\epsilon w_0}\right)\left(\frac{w+w_0}{w+w_0-i\epsilon w_0}\right) \\
&= \frac{w^2-w_0^2}{(w-i\epsilon w_0)^2-w_0^2}
\end{aligned} \tag{13.5.14}$$

在(13.5.14)式中，参数ϵ作为 w_0 的分数，是一个较小正数，表示槽口的期望宽度。通过用 z 代替 w 的算术运算，给出滤波系数

$$
\begin{aligned}
c_0 &= \frac{1+w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\
c_1 &= -2\frac{1-w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\
c_2 &= \frac{1+w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\
d_0 &= 2\frac{1-\epsilon^2 w_0^2-w_0^2}{(1+\epsilon w_0)^2+w_0^2} \\
d_1 &= -\frac{(1-\epsilon w_0)^2+w_0^2}{(1+\epsilon w_0)^2+w_0^2}
\end{aligned}
\tag{13.5.15}
$$

图 13.5.1 显示了对"鸟鸣"输入信号使用(13.5.15)式形式的滤波后得出的结果。这种信号的频率向上渐变,一直跨过槽口频率。

双线性变换虽然看起来可能很一般,但它的应用由形成滤波的某些特征所限制。这种方法擅长于得到所期望的滤波器的普通形式,适用于"平坦"的理想情况。然而,w 和 f 之间的非线性映射,使得设计所期望的截止形状变得困难,并且可能把截止频率(用定量的 dB 定义)移离它们的期望位置。因而,数字滤波器的设计者将双线性变换留用于特殊的场合,并且给它们本身配备大量的其他方法。建议读者按设计方案的要求同样来做。

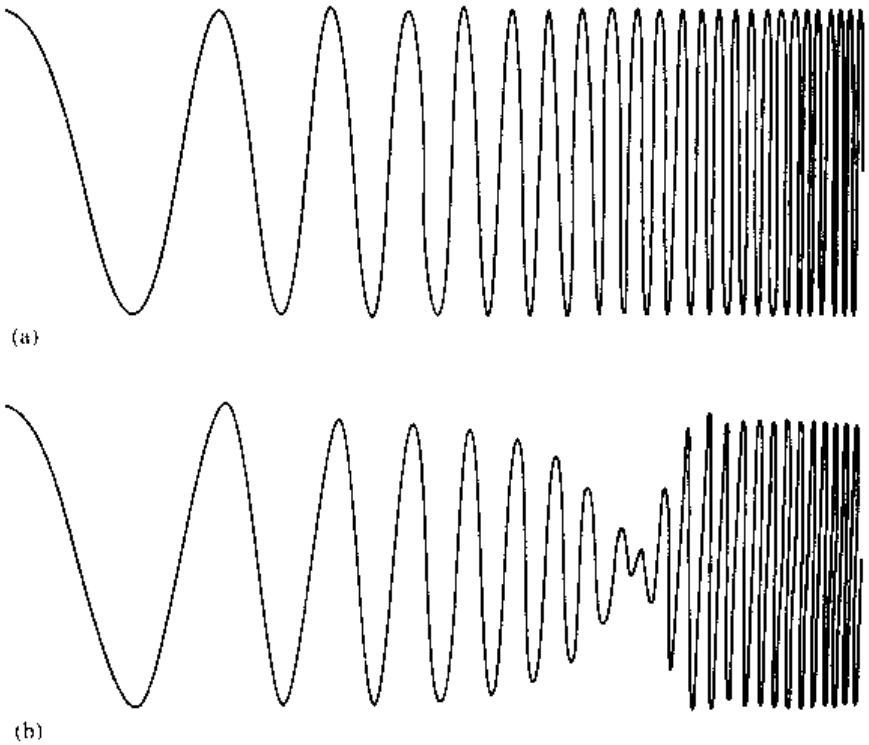


图 13.5.1 (a)一个"鸟鸣"信号,即频率随时间连续地增长;(b)通过槽形滤波器(13.5.15)后的同一信号,此处参数ϵ是0.2

参考文献和深入阅读

Hamming, R.W. 1983, *Digital Filters*, 2nd ed. (Englewood Cliffs, NJ: Prentice-Hall).

Antoniou, A. 1979, *Digital Filters: Analysis and Design* (New York: McGraw-Hill).

Parks, T.W., and Burrus, C.S. 1987, *Digital Filter Design* (New York: Wiley).

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

Rice, J.R. 1964, *The Approximation of Functions* (Reading, MA: Addison-Wesley); also 1969, *op. cit.*, Vol. 2.

Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

13.6　线性预测和线性预测编码

从一个非常一般的公式出发，然后与各种特殊情况联系起来。设$\{y'_\alpha\}$是变量 y 的一组测量数据，y 的真实值表示为$\{y_\alpha\}$，测量值由真实值增加了一个随机噪声

$$y'_\alpha = y_\alpha + n_\alpha \tag{13.6.1}$$

和(13.3.2)式相比，可以看出符号上的一些差别。采用希腊字母作为数据组的下标，是为了表明数据点不是在一条直线上均匀分布或顺序排列的。例如，它们可以是三维空间中"随机"分布的点。现在，假设要对一些特殊的点$y_\star$的真实值进行最优估计，$y_\star$是已知的带有噪声的数据点的线性组合，即

$$y_\star = \sum_\alpha d_{\star\alpha} y'_\alpha + x_\star \tag{13.6.2}$$

要找出系数$d_{\star\alpha}$以使差值 $x_\star$ 最小。系数$d_{\star\alpha}$有一个星号下标是为了表明它依赖于$y_\star$点的选择。后面，可能还要设$y_\star$作为y_α的一个存在值。在那种情况下，我们的问题是一个最优滤波或估计的问题，和 13.3 节中讨论的非常类似。而另一方面，可能希望$y_\star$成为全新的点，在这种情况下，我们的问题是一个**线性预测**的问题。

为使差值 $x_\star$ 最小，自然是在统计均方差意义下考虑。如果用尖括号代表统计均方差，则所求的$d_{\star\alpha}$值是使下式达到最小：

$$\begin{aligned}\langle x_\star^2\rangle &= \left\langle \left[\sum_\alpha d_{\star\alpha}(y_\alpha + n_\alpha) - y_\star\right]^2 \right\rangle \\ &= \sum_{\alpha\beta}(\langle y_\alpha y_\beta\rangle + \langle n_\alpha n_\beta\rangle) d_{\star\alpha} d_{\star\beta} - 2\sum_\alpha \langle y_\star y_\alpha\rangle d_{\star\alpha} + \langle y_\star^2\rangle\end{aligned} \tag{13.6.3}$$

其中，已经使用了噪声和信号不相关这个事实，即$\langle n_\alpha y_\beta\rangle = 0$。量$\langle y_\alpha y_\beta\rangle$和$\langle y_\star y_\alpha\rangle$描述了基本数据的自相关结构。在(13.2.2)式中对于一条直线上均匀分布的点，已经看到了类似的表达式。在第 14 章和第 15 章还会遇到统计意义上的相关性。量$\langle n_\alpha n_\beta\rangle$描写了噪声的自相关性质。通常对于点与点的不相关的噪声，存在关系式$\langle n_\alpha n_\beta\rangle = \langle n_\alpha^2\rangle \delta_{\alpha\beta}$。于是可以方便地得到组成矩阵和向量的各种相关量，

$$\phi_{\alpha\beta} \equiv \langle y_\alpha y_\beta\rangle \qquad \phi_{\star\alpha} \equiv \langle y_\star y_\alpha\rangle \qquad \eta_{\alpha\beta} \equiv \langle n_\alpha n_\beta\rangle \ \text{ or } \ \langle n_\alpha^2\rangle \delta_{\alpha\beta} \tag{13.6.4}$$

将(13.6.3)式对$d_{\star\alpha}$求导并令其为 0，得到线性方程组，

$$\sum_\beta [\phi_{\alpha\beta} + \eta_{\alpha\beta}] d_{\star\beta} = \phi_{\star\alpha} \tag{13.6.5}$$

如果采用逆矩阵的形式，那么在忽略最小差值 $x_\star$ 时估计(13.6.2)式可写成

$$y_\star \approx \sum_{\alpha\beta} \phi_{\star\alpha} [\phi_{\mu\nu} + \eta_{\mu\nu}]^{-1}_{\alpha\beta} y'_\beta \tag{13.6.6}$$

从(13.6.3)式和(13.6.5)式我们还可以计算出差值 $x_\star$ 的最小均方值，记为$\langle x_\star^2\rangle_0$，

$$\langle x_\star^2\rangle_0 = \langle y_\star^2\rangle - \sum_\beta d_{\star\beta}\phi_{\star\beta} = \langle y_\star^2\rangle - \sum_{\alpha\beta} \phi_{\star\alpha} [\phi_{\mu\nu} + \eta_{\mu\nu}]^{-1}_{\alpha\beta} \phi_{\star\beta} \tag{13.6.7}$$

最后，看一个一般的结果，它告知如果在(13.6.2)式中不用最优值 $d_{\star\beta}$，而用一些别的值 $\widehat{d}_{\star\beta}$，看均方差值$\langle x_\star^2\rangle$将增加多少。上面的等式隐含着

$$\langle x_\star^2\rangle = \langle x_\star^2\rangle_0 + \sum_{\alpha\beta}(\widehat{d}_{\star\alpha} - d_{\star\alpha})[\phi_{\alpha\beta} + \eta_{\alpha\beta}](\widehat{d}_{\star\beta} - d_{\star\beta}) \tag{13.6.8}$$

因为第二项是一个纯四次式，所以可以看到差值的增加仅仅是估计 $d_{\star\beta}$ 值时产生的误差的二次式。

13.6.1　与最优滤波的联系

如果将"星号"变为希腊下标，比如γ，则上面的等式描述的正是最优滤波，它是 13.3 节中讨论的推广，例如，如果噪声n_α的幅度变为零，那么噪声自相关$\eta_{\alpha\beta}$也将变为零，矩阵和它的逆矩阵相乘将互相抵消，于是，

(13.6.6)式就简化为$y_\gamma = y_\gamma'$。还有一种特殊情况是,矩阵$\phi_{\alpha\beta}$和$\eta_{\alpha\beta}$是对角矩阵,在这种情况下,(13.6.6)式成为

$$y_\gamma = \frac{\phi_{\gamma\gamma}}{\phi_{\gamma\gamma} + \eta_{\gamma\gamma}} y_\gamma' \tag{13.6.9}$$

和(13.3.6)式相比较可以看出 $S^2 \to \phi_{\gamma\gamma}$、$N^2 \to \eta_{\gamma\gamma}$。实际情况是:对于均匀间隔数据点的情况,在傅里叶变换域中,自相关就很简单地成为傅里叶振幅的平方(Wiener-Khinchin 定理,见(12.0.12)式),最优滤波可以从(13.6.9)式构造,而不需要对矩阵求逆。

更一般的情况是,在时域或其他域中,最优滤波(在测量噪声存在时,从基本的真实值中求差值平方极小化)可以通过估计自相关矩阵$\phi_{\alpha\beta}$和$\eta_{\alpha\beta}$以及运用(13.6.6)式(将$\star \to \gamma$)来构造(事实上,(13.6.8)式是 13.3 节中叙述情况的基础,在那里即使是粗糙的最优滤波也可能是十分有效的)。

13.6.2 线性预测

经典的线性预测讨论的是一种特殊情况,即数据点y_β在一条线上等间隔分布y_i($i = 0, 1, \ldots, N-1$),并且要求用 M 个连续点y_i来预测第 $M+1$ 点。假设其是平稳的,即$\langle y_j y_k \rangle$的自相关取决于差值$|j-k|$,而不单独地依赖于 j 或 k,因此自相关ϕ只有一个下标:

$$\phi_j \equiv \langle y_i y_{i+j} \rangle \approx \frac{1}{N-j} \sum_{i=0}^{N-j-1} y_i y_{i+j} \tag{13.6.10}$$

其中近似等式说明了如何用实际的数据组来估计自相关分量(实际上,还有一个更好的方法进行这种估计,参看下式)。在我们所描述的情况下,(13.6.2)式的估计式是

$$y_n = \sum_{j=0}^{M-1} d_j y_{n-j-1} + x_n \tag{13.6.11}$$

(和(13.5.1)式相比)(13.6.5)式成为含 M 个未知数 d_j 的 M 个方程的方程组,d_j 称为**线性预测系数(LP 系数)**,

$$\sum_{j=0}^{M-1} \phi_{|j-k-1|} d_j = \phi_k \qquad (k = 1, \ldots, M) \tag{13.6.12}$$

注意,尽管噪声没有明显地包含在上面的等式中,但如果点与点不相关,它还是被正确地考虑进去了:根据(13.6.10)式用测量值 y_i' 估计的ϕ_0,实际估计了上面的对角部分$\phi_{\alpha\alpha} + \eta_{\alpha\alpha}$,根据(13.6.7)式可以估计均方差值$\langle x_n^2 \rangle$为

$$\langle x_n^2 \rangle = \phi_0 - \phi_1 d_0 - \phi_2 d_1 - \cdots - \phi_M d_{M-1} \tag{13.6.13}$$

为了运用线性预测,首先利用(13.6.10)式和(13.6.12)式计算 d_j,接着计算(13.6.13)式。具体来说,就是将(13.6.11)式用于已知的记录数据中而求得差值 x_i 有多大。如果差值很小,则继续应用(13.6.11)式进行下一步的运算,估计下一步的差值 x_i,直到差值为 0。在这种运用情况下,(13.6.11)式是一种外推表达式。在很多情况下,可以发现这种外推比任何一种简单的多项式外推更加强有力(顺便提一句,请不要混淆了"线性预测"和"线性外推"这两个概念,线性预测的一般函数表达式比直线甚至低阶的多项式复杂得多)。

但是,为了充分利用线性预测,必须注意它所限制的一面:必须进行额外的测量以保证它的稳定性。(13.6.11)式是一般的线性滤波器(13.5.1)的特殊情况。(13.6.11)式作为线性预测器能够稳定的条件已精确地由(13.5.5)式和(13.5.6)式给出,也就是特征多项式

$$z^N - \sum_{j=0}^{N-1} d_j z^{(N-1)-j} = 0 \tag{13.6.14}$$

它使 N 个根都落在单位圆内,即

$$|z| \leq 1 \tag{13.6.15}$$

不能保证由(13.6.12)式确定的系数有这种性质。如果数据点包含许多振荡,而这些振荡没有任何振幅增加或减少的特定趋势,则(13.6.14)式中的复根都和单位圆靠得很近。有限长度的数据组也将使一些根在单位圆内,而其余的根在单位圆外。在某些应用中,结果的不稳定性是缓慢上升的,并且线性预测不会遍离得太

远，因此最好直接利用由(13.6.12)式得到的“没有加工”过的线性预测系数。例如，如果可以将一组数据外推一小步，那么可以从数据出发按此步长分别向前和向后外推。如果两个外推符合得非常好，则不稳定性便不是一个问题。

当不稳定性成为问题时，就必须对线性预测系数进行“加工处理”。可以按如下步骤进行：(i)求解(13.6.14)式得到 N 个复根(用数值方法)；(ii)将根移到认为应该是在单位圆内或在单位圆上；(iii)重新构造现在已经修正过的 LP 系数。读者可能会认为第(ii)步有一点模糊。确实如此，没有更好的过程了。如果认为信号确实是一组不衰减的正弦和余弦波之和(可能具有不相称的周期)，那么只需简单地将每个根 z_i 移到单位圆上，

$$z_i \rightarrow z_i/|z_i| \tag{13.6.16}$$

在另外的情况下，将一些病态的根映射到圆内可能比较合适，

$$z_i \rightarrow 1/z_i^* \tag{13.6.17}$$

这种替换方法有一个性质，即当(13.6.11)式输出的振幅由一组 x_i 的正弦曲线驱动时，其振幅保持不变。假设(13.6.12)式能够正确地识别共振的谱宽度，但是仅疏忽了识别它的时间含意，以使应随时间衰减的信号结束了振幅的增长。对(13.6.16)式和(13.6.17)式选择有一点随意性，但倾向于(13.6.17)式。

还有一个问题是 M 的选择，即使用的线性预测系数的个数。应该根据工作规模选择 M 尽可能小，也就是说，必须由实验来确定所选择的数据点。可以试取 $M=5,10,20,40$。如果需要的 M 比这个值还要大，要注意那些复根的“处理”过程对于舍入误差非常敏感，所以应采用双精度。

线性预测对于光滑和振荡信号的外推特别有用，尽管不要求信号是周期性的。在这种情况下，通过信号的很多次循环后，线性预测外推一般非常精确。相比之下，多项式外推一般在经过一两次循环之后就变得不精确。可以成功地进行线性预测的信号原型事例是海洋潮汐的高度，对于此信号，基本的 12 小时周期信号的相位和振幅随着年月进行调整，当地的水压效应会使一个周期的曲线的形状和正弦波相当不同。

我们已经注意到，从数据集来估算协方差 ϕ_k，(13.6.10)式并不是最好的方法。实际上，线性预测所得结果对 ϕ_k 值估计的精确程度非常敏感。Burg[1]中提出了一个较好的方法，该方法包括一个迭代过程，每次迭代 M 的阶数加 1。在每一步迭代中重新估计系数 $d_j(j=0,\ldots,M-1)$ 以使(13.6.13)式中的残差尽可小的。尽管Burg方法超出了本书的讨论范围，但下面的程序[1,2]给出了从一组数据估计 LP 系数 d_j 的方法。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::memcof(Vec_I_DP &data, DP &xms, Vec_O_DP &d)
给定实型向量 data[0..n-1]，此程序返回 m 个线性预测系数为 d[0..m-1]，同时均方差值以 xms 返回。
{
    int k,j,i;
    DP p=0.0;

    int n=data.size();
    int m=d.size();
    Vec_DP wk1(n),wk2(n),wkm(m);
    for (j=0;j<n;j++) p += SQR(data[j]);
    xms=p/n;
    wk1[0]=data[0];
    wk2[n-2]=data[n-1];
    for (j=1;j<n-1;j++) {
        wk1[j]=data[j];
        wk2[j-1]=data[j];
    }
    for (k=0;k<m;k++) {
        DP num=0.0,denom=0.0;
        for (j=0;j<(n-k-1);j++) {
            num += (wk1[j]*wk2[j]);
            denom += (SQR(wk1[j])+SQR(wk2[j]));
        }
        d[k]=2.0*num/denom;
        xms *= (1.0-SQR(d[k]));
        for (i=0;i<k;i++)
            d[i]=wkm[i]-d[k]*wkm[k-1-i];
```

```
            该算法随 m 值的增加进行迭代计算,直至达到所要求的值。在此算法中,此程序返回向量 d 和标量 xms,
            它们是由 k 项(而不是 m 项)组成的一组 LP 系数。
        if (k == m-1)
            return;
        for (i=0;i<=k;i++) wkm[i]=d[i];
        for (j=0;j<(n-k-2);j++) {
            wk1[j] -= (wkm[k]*wk2[j]);
            wk2[j]=wk2[j+1]-wkm[k]*wk1[j+1];
        }
    }
    nrerror("never get here in memcof.");
}
```

下面是使 LP 系数变得稳定的程序(如果需要这样做的话),用原始的或加工过的 LP 系数通过线性预测对一组数据进行外推。程序 zroots(参见 9.5 节)用来求一个多项式的所有复根。

```
#include <cmath>
#include <complex>
#include "nr.h"
using namespace std;

void NR::fixrts(Vec_IO_DP &d)
对于给定的 LP 系数 d[0..m-1],此程序找出特征多项式(13.6.14)的所有根,将单位圆外的根映射到单位圆内,接着返
回处理过的修正系数 d[0..m-1]。
{
    bool polish=true;
    int i,j;

    int m=d.size();
    Vec_CPLX_DP a(m+1),roots(m);
    a[m]=1.0;
    for (j=0;j<m;j++)                    设置所求根的多项式系数为复数值。
        a[j]= -d[m-1-j];
    zroots(a,roots,polish);              求出所有的根。
    for (j=0;j<m;j++)                    寻找单位圆外的根并将它映射到圆内。
        if (abs(roots[j]) > 1.0)
            roots[j]=1.0/conj(roots[j]);
    a[0]= -roots[0];                     重构造多项式系数。
    a[1]=1.0;
    for (j=1;j<m;j++) {                  通过对根循环和综合乘法。
        a[j+1]=1.0;
        for (i=j;i>=1;i--)
            a[i]=a[i-1]-roots[j]*a[i];
        a[0]= -roots[j]*a[0];
    }
    for (j=0;j<m;j++)                    多项式的系数是实数所以我们
        d[m-1-j] = -real(a[j]);          只需返回实部作为新的LP系数。
}

#include "nr.h"

void NR::predic(Vec_I_DP &data, Vec_I_DP &d, Vec_O_DP &future)
已知 data[0..ndata-1]和数据的 LP 系数 d[0..m-1],此程序运用(13.6.11)式预测后面 nfut 个数据点,这些数据点以数
组future[0..nfut-1]返回。注意程序只用 data 数组最后 m 个数据点作为预测的初始值。
{
    int k,j;
    DP sum,discrp;

    int ndata=data.size();
    int m=d.size();
    int nfut=future.size();
    Vec_DP reg(m);
    for (j=0;j<m;j++) reg[j]=data[ndata-1-j];
    for (j=0;j<nfut;j++) {
        discrp=0.0;
        如果是通过线性预测程序重新构造一个函数而不是通过线性预测外推一个函数,那么在此应该输入已知的差值。
```

```
            sum=discrp;
            for (k=0;k<m;k++) sum += d[k]*reg[k];
            for (k=m-1;k>=1;k--) reg[k]=reg[k-1];     [若要执行循环数组
            future[j]=reg[0]=sum;                      可以免去系数的移位。]
        }
    }
```

13.6.3　去除线性预测的偏差

可能会期望(13.6.11)式中 d_j 之和(或者更普遍一些,在(13.6.2)式中)应该是 1,以便对所有的数据点 y_i 增加一个常数后所产生的预测也增加一个相同的常数。但是 d_j 之和并不是 1,通常比 1 稍小。这个事实揭示了经典的线性预测不是**无偏**的,尽管它使均方差值最小。被测的自相关并不必然包含一个较好的估计,线性预测方程组倾向于预测趋向零的值。

有时,那正是人们所需的。如果产生 y_i 的过程确实有零平均值,则在缺少其他信息的情况下,零将是最优的估计值。然而,在其他的情况下,这是没有根据的。如果数据显示出围绕某正值有微小的变化,便不需要趋向零的线性预测。

通常还有一个非常有效的近似,那就是从数据组中减去平均值,再进行线性预测,接着再把平均值加上。这一过程是正确的,但简单的算术平均值并不是要减去的正确的常数。实际上,无偏估计式是通过各点减去由下式定义的自相关加权平均值而得到的(参见本节参考文献[3,4]):

$$\overline{y} \equiv \sum_{\beta} [\phi_{\mu\nu} + \eta_{\mu\nu}]^{-1}_{\alpha\beta}\, y_\beta \Big/ \sum_{\alpha\beta} [\phi_{\mu\nu} + \eta_{\mu\nu}]^{-1}_{\alpha\beta} \tag{13.6.18}$$

通过这种减法后,LP 系数之和应该是单位值。不论ϕ_k 如何估计,它都符合舍入误差和差值。

13.6.4　线性预测编码(LPC)

一个不同但是有关联的方法,即应用上面的等式对采样信号进行"压缩",以便使信号更紧凑地储存起来。原始的信号应该能从已压缩形式中精确地恢复出来。显然,仅当信号有冗余度时压缩才能实现。(13.6.11)式描述了一种冗余度:它说明除了一点小的偏差外,一个信号可以从它前面的值以及少量的 LP 系数预测出来。因此利用(13.6.11)式进行信号的压缩称为**线性预测编码**,或 **LPC**。

LPC 最基本的思想(用最简单的形式)是记录下述数据作为一个压缩的文件:(i)LPC 系数的数目 M;(ii)由于程序 memcof 得到的 M 个值;(iii)开始的 M 个数据点;(iv)以后每个点仅求它的残差 x_i((13.6.1)式)。当正在生成压缩文件时,将(13.6.1)式运用于前面的 M 个点,并从当前点的实际值减去和式而求得残差。当重新恢复原文件时,将残差再加到程序 predic 所显示的各点上。

可能还不是很明白为什么要用这种方法进行数据压缩。对每个数据点我们是存储残差值,为什么不存储原始数据值呢?答案取决于我们所涉及的数目的相对大小。残差是通过两个非常相近的数(数据和线性预测值)相减而得到的,因此,通常残差有数目非常少的非零位。这样,这些数据就能存储在一个被压缩的文件中。那么,怎样用高级语言做到这些呢?下面是一种方法:改变数据的尺寸,使它们成为整数值,比如在 +1 000 000 和 −1 000 000 之间(假设需要六位有效数字)。调整(13.6.1)式,使之包含上面"整数部分"操作的和。现在通过定义,差值将是整数值。对不同的 M 值实验,寻找 LP 系数使差值尽可能小,小到允许的程度。如果能达到 ± 127 的范围(在我们的实验中,这一点并不难),那么就可以在文件中用一个字节将差值写下来。和原来数据用 4 个字节的整数和浮点数相比,该压缩因子为 4

注意 LP 系数是用量化的数据计算的,差值也是量化了的,也就是说,在 LPC 循环的内外都是量化的数值。如果仔细地按上面的步骤进行,那么除了最初对数据量化外,在压缩-重建过程中不会引入任何,哪怕是单个位的舍入误差。尽管在对(13.6.11)式求和时会引入舍入误差,但所存储的残差在加回求和式时,给出了精确的原始(量化)数据值。还请注意,不必为了稳定性对 LP 系数进行加工处理,通过对各个点加上残差,将永远不会偏离原始数据,所以不会增加不稳定性。因此,没有必要像上面那样运行 fixrts。

在 20.4 节中将学习 **Huffman 编码**,该方法将利用小差值出现频率大于大差值出现频率这一事实,进一

步压缩残差。最初的 Huffman 编码采用下面形式,如果大多数的差值在 ±127 的范围内,只是偶尔一两个差值超出这范围,那么用数据值 127 表示为"超出范围",接着对超出范围的差值的全字值记录于文件中(紧随 127 之后)。在 20.4 节将解释怎样能做得更好。

有许多不同的处理过程都属于 LPC 的范畴。

- 如果数据的谱特征依赖于时间,则最好不要用整组数据来求一组 LP 系数,而是将数据分成几段,对每一段分别计算和存储不同的 LP 系数。
- 如果数据确能用它们的 LP 系数很好地表征,而且又能够容忍小量的误差的话,则不必麻烦地去存储所有的残差。可以直接进行线性预测直到误差超出容差限的范围,然后再重新初始化(且 M 个顺序存储的残差),并继续预测。
- 在某些应用情况下,最明显的应用是语言合成,人们关心的仅仅是重建信号的谱幅度,而不是相对相位。在这种情况下,人们不需要存储任何初始数据,仅仅是各个数据段的 LP 系数。输出的重建是通过用所有数据都为零(除一个非零的尖峰外)的初始条件驱动这些系数来完成的。一个语言合成芯片可有多达 10 阶的 LP 系数,它每秒可能变化 20 到 50 次。
- 有些人相信即使残差 x_i 不是很小,用 LPC 分析信号也非常有用。此时 x_i 被解释为真实的"输入信号",当通过由 LP 系数定义的多极点滤波器滤波后,它成为被观察的"输出信号"。据说,LPC 同时揭示了滤波器的本质以及驱动它的特定输入信号的性质。我们对这种应用表示怀疑,其有些过于夸张了。

参考文献和深入阅读

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), especially the paper by J. Makhoul (reprinted from *Proceedings of the IEEE*, vol. 63, p. 561, 1975).

Burg, J.P. 1968, reprinted in Childers, 1978. [1]

Anderson, N. 1974, reprinted in Childers, 1978. [2]

Cressie, N. 1991, in *Spatial Statistics and Digital Image Analysis* (Washington: National Academy Press). [3]

Press, W.H., and Rybicki, G.B. 1992, *Astrophysical Journal*, vol. 398, pp. 169–176. [4]

13.7 深入讨论:用最大熵(全极)方法做功率谱估计

FFT 并不是一个过程的功率谱估计的惟一途径,也不一定是达到目的的最好方法。为了弄清楚怎样构思另外一种方法,我们首先扩大一下范围,使之不仅仅包括 Nyquist 区间 $-f_c<f<f_c$ 内的实数频率,而且还包括整个复频率平面。从这一角度出发,通过下式将复 f 平面变换一个新的平面,称为 z 变换平面或 z 平面,由关系式

$$z \equiv e^{2\pi i f \Delta} \tag{13.7.1}$$

表示。其中 Δ 和通常情况一样,是时域的取样间隔。注意 f 平面上实轴 Nyquist 区间和复 z 平面上的单位圆有一一对应关系。

如果将(13.7.1)式与(13.4.4)式和(13.4.6)式比较,则可以看出,在不考虑归一化条件时,对任何实取样函数 $c_k \equiv c(t_k)$ 的 FFT 功率谱估计可写为

$$P(f)=\left|\sum_{k=-N/2}^{N/2-1} c_k z^k\right|^2 \tag{13.7.2}$$

当然(13.7.2)式不是基本函数 $c(t)$ 的真正功率谱,而仅仅是个估计。从下面两个有关方面可以看出为什么这个估计值不是精确的。首先在时域里,估计值仅仅是以函数 $c(t)$ 有限的一段为基础,而实际上我们知道 $c(t)$ 从 $t=-\infty$ 到 ∞ 是连续的。其次,在(13.7.2)式的 z 平面中,通常情况下,有限的 Laurent 级数仅仅对 z 的一般解析函数提供一个近似值。事实上,一个表示"真正"功率谱的正规表达式(归一化)是

$$P(f)=\left|\sum_{k=-\infty}^{\infty} c_k z^k\right|^2 \tag{13.7.3}$$

这是一个 Laurent 无穷级数，依赖于无穷个 c_k 值。(13.7.2)式仅仅是由(13.7.3)式确定的 z 的解析函数的一种解析近似。事实上，这种近似隐含在用 FFT 以周期图方法进行功率谱估计中。它有几种命名方法，包括**直接方法、全零点模型和移动平均（MA）模型**。"全零点"是特指模型的谱在 z 平面上有零点，但没有极点。

如果从更广的角度看(13.7.3)式的近似问题，就可以发现如果采用一个有理函数，在其分子和分母都采用(13.7.2)式型的级数，就可能做得更好。还有一个表达式，尽管不是很明显，但在近似时却有一些有利用处，这个近似值的自由参数都在分母上，也就是

$$P(f)\approx\frac{1}{\left|\sum_{k=-M/2}^{M/2} b_k z^k\right|^2}=\frac{a_0}{\left|1+\sum_{k=1}^{M} a_k z^k\right|^2} \tag{13.7.4}$$

其中，第二个等号引入了一组新的系数 a_k，它可以根据 z 都在单位圆上由 b_k 求得。而 b_k 可以根据（13.7.4）式的幂级数展开应和（13.7.3）式前面 $M+1$ 项一致的条件来确定。实际上，还有其他的方法确定 b_k 或 a_k。

近似式(13.7.2)和(13.7.4)的区别不仅仅是形式上的，它们的近似也具有不同的特征。最明显的是(13.7.4)式可以有极点，它对应于 z 单位圆上，也就是 Nyquist 区间的实频率上，无穷大的功率谱密度。这种极点精确地代表了具有离散的、陡峭的"直线"或 δ 函数的基本功率谱。相比之下，(13.7.2)式在 Nyquist 区间的实频率上只有零点，而没有极点。因此必须用多项式才能适合具有尖峰的谱的特征。近似(13.7.4)式有几个名字：**全极点模型、最大熵方法(MEM)或自回归模型(AR)**。只需知道怎样从数据组计算系数 a_0 和 a_k，就能用(13.7.4)式获得谱估计。

令人高兴的是，我们已经知道怎样计算。回到前面看看线性预测式(13.6.11)。将它和线性滤波(13.5.1)式和(13.5.2)式相比较，就可以发现从滤波器的角度将 x 作为输入信号，y 作为输出信号，线性预测有一个滤波函数

$$\mathcal{H}(f)=\frac{1}{1-\sum_{j=0}^{N-1} d_j z^{-(j+1)}} \tag{13.7.5}$$

因此，y 的功率谱等于 x 的功率谱乘以 $|\mathcal{H}(f)|^2$。现在，考虑当 x 是线性预测的残差时，这个输入信号 x 的功率谱是什么。虽然不进行正式证明，但从直观上我们可以相信，由于 x 是独立随机的，因此具有平坦的谱(白噪声)。粗略地说，x 中遗留的任何残差关系容许一个更精确的线性预测，并且这种残差关系也可以被去除。这种平坦谱的整体归一化就是 x 的均方振幅。这正是由(13.6.13)式计算的精确量，并且通过程序 memcof 以 xms 返回。因此(13.7.4)式中的系数 a_0 和 a_k 与由 memcof 返回的 LP 系数存在下面的简单关系：

$$a_0=\mathbf{xms}\qquad a_k=-\mathbf{d}(k-1),\quad k=1,\ldots,M \tag{13.7.6}$$

还有另外一种方法描述 a_k 和自相关分量 ϕ_k 的关系。Wiener-Khinchin 公理(12.0.12)说明自相关的傅里叶变换等于功率谱。用 z 变换的语言来说，这种傅里叶变换就是用 z 表示的 Laurent 级数。确定系数的(13.7.4)式就成为

$$\frac{a_0}{\left|1+\sum_{k=1}^{M} a_k z^k\right|^2}\approx\sum_{j=-M}^{M}\phi_j z^j \tag{13.7.7}$$

(13.7.7)式中的近似号具有一种特殊的解释。它意味着，左边幂级数展开式和右边的各项 z^{-M} 到 z^M 都是一致的。在此范围外，右边各项均为 0，而左边还有非零项。注意，M 近似等式左边的系数的个数，它可以最大到 N 的任一整数，N 是可用的自相关所有数目(实际中，M 可以选择得比 N 小得多)。M 被称为逼近的阶数或极点的个数。

对任何选择的 M 值，(13.7.7)式左边的级数展开定义了滞后大于 M 的自相关函数的某种外推，事实上其至可以超过 N，也就是说超出了实际所测量的数据范围。从信息论角度，可以证明在所有可能的外推中，

这种外推具有最大熵,因此称为**最大熵方法**或 **MEM**。MEM 的最大熵性质使它获得广泛应用。有人说它本身给出一个比其他方法"更好"的估计,不要相信这种说法。MEM 确实有让人信服的能适用于拟合带有陡峭的谱特征的性质,但对于功率谱估计它并没有其他任何特性。

在程序 memcof 中操作数标尺为 N(数据点的个数)和 M(MEM 近似期待的阶数)的乘积。如果 M 选择得和 N 一样,那么 MEM 方法比前一节 $N \log N$ 量级的 FFT 慢得多。然而实际中,人们通常限制 MEM 的阶数(或极点个数),把它近似为几倍于要估计的谱所具有的陡峭谱线的个数。由于对极点个数进行了这种限制,这种方法能在一定程度上使谱平滑,这通常是人们希望得到的性质。尽管精确值决定了实际应用,但人们通常取 $M = 10$ 或20 或 50 而 $N = 1000$ 或 10 000。在这种情况下,MEM 估计不比 FFT 估计慢多少。

有必要注意程序 memcof。如果极点的个数或数据的数目太大了,即使采用双精度,舍入误差也会成为一个问题。对于"尖峰"数据(也就是具有非常尖的谱线特征数据),甚至最低阶的算法也可能会将尖峰分开,并且尖峰还会随正弦波的相位而移动。另外,对于带有噪声的输入函数,如果选择的阶数太高,将会发现很多伪尖峰。一些专家建议将这种算法和一些守恒的算法如周期图法结合起来使用,以帮助选择正确的模型阶数,避免被假的谱线特征所迷惑。建议读者要根据自己的问题仔细地试用它。现在从对系数的讨论转到对 MEM 谱估计的计算。

MEM 估计式(13.7.4)是连续变化频率 f 的函数。如同 FFT 那样,对频率均匀划分没有任何特别的意义。事实上,因为 MEM 估计可能有非常尖的谱线特征,所以希望在这些特征线附近用非常精细的步骤来求值,而在远离这些特征线的地方粗糙一些。下面的函数,在系数已经计算出来后,估计(13.7.4)式,并且以 $f\Delta$(频率乘以取样间隔)为变量的函数形式返回估计的功率谱。$f\Delta$ 应位于 Nyquist 区间 $-1/2 \sim 1/2$ 中。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::evlmem(const DP fdt, Vec_I_DP &d, const DP xms)
利用 memcof 返回已知的 d[0..m-1],xms,此函数以 fdt = fΔ 为变量,返回功率谱 P(f)。
{
    int i;
    DP sumr=1.0,sumi=0.0,wr=1.0,wi=0.0,wpr,wpi,wtemp,theta;

    int m=d.size();
    theta=6.28318530717959*fdt;
    wpr=cos(theta);                    建立循环关系。
    wpi=sin(theta);
    for (i=0;i<m;i++) {                对求和中的各项循环。
        wr=(wtemp=wr)*wpr-wi*wpi;
        wi=wi*wpr+wtemp*wpi;
        sumr -= d[i]*wr;               此处的累积用于计算(13.7.4)式的分母。
        sumi -= d[i]*wi;
    }
    return xms/(sumr*sumr+sumi*sumi);
}
```

注意要确保用足够精细的网格估计 $P(f)$,以便寻找可能存在的非常窄的峰。这种窄峰如果出现,可能会包括数据中的所有功率。用户可能希望知道由程序 memcof 和 evlmem 产生 $P(f)$是怎样对输入数据向量的均方值归一化的,答案是:

$$\int_{-1/2}^{1/2} P(f\Delta)d(f\Delta) = 2\int_{0}^{1/2} P(f\Delta)d(f\Delta) = \text{数据的均方值} \tag{13.7.8}$$

由程序 memcof 和 evlmen 产生的谱的范例如图 13.7.1 所示。

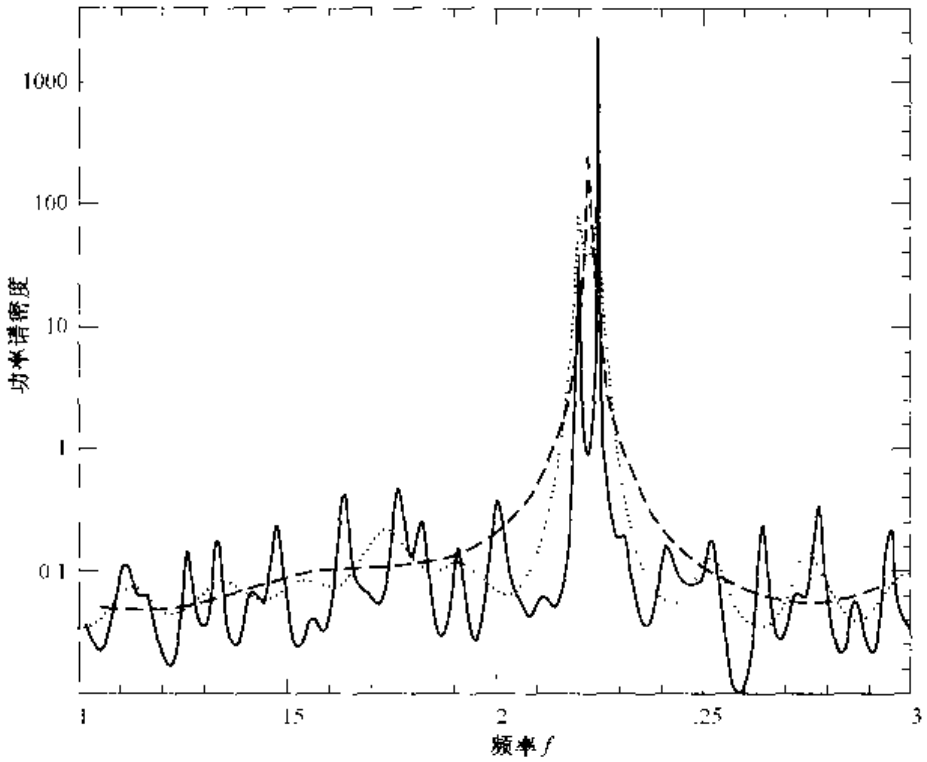


图 13.7.1　用最大熵谱估计的输出范例。输入信号由频率非常接近的两个正弦信号之和的512个采样点组成，再加上约为均匀功率谱的白噪声。图示的是整个Nyquist频率区间的扩展部分（它从0延伸到0.5）。图中短划线表示的谱估计使用了20个极点，点线表示的谱估计使用了40个极点，实线表示的谱估计使用了150个极点。这种方法可以明显地区分两个正弦信号，但是白噪声背景也开始显出伪实峰（注意图中的对数刻度）

参考文献和深入阅读

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), Chapter II.
Kay, S.M., and Marple, S.L. 1981, *Proceedings of the IEEE*, vol. 69, pp. 1380–1419.

13.8　深入讨论：非均匀取样数据的谱分析

迄今为止，我们全部都是处理均匀取样的数据点，

$$h_n = h(n\Delta) \qquad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \tag{13.8.1}$$

其中的 Δ 是取样间隔，它的倒数是取样率。在 12.1 节由取样定理说明 Nyquist 临界频率

$$f_c \equiv \frac{1}{2\Delta} \tag{13.8.2}$$

其意义是：由(13.8.1)式取样的数据集包含了信号 $h(t)$ 在 Nyquist 频率内所有谱分量的全部信息，而且混叠或杂混了频率大于 Nyquist 频率的任何信号分量的信息。因此取样定理说明了对均匀取样数据分析时的有利和不利之处。

但在有些情况下得不到均匀间隔的数据。一个普遍的情况是仪器的数据丢失，因此获得的数据只是(13.8.1)式的子集（非连续的整数），即所谓**数据丢失**的问题。还有一个普遍的情况存在于观察科学例如天文学中，此时观察者不能完全控制观察时间，而必须接受某个受限制的时间集 t_i。

从非均匀间隔的 t_i 到(13.8.1)式所示的均匀间隔，有某些显而易见的方法。插值是一种方法：设置一个时间均匀间隔的网，利用对数据插值得到网络点的值然后用 FFT 方法。对丢失数据的问题，只需插入丢失的数据点。如果很多连续数据点丢失了，最好令它们为 0，或者强制放到最后测量的点。但是，实践者的经验表明这种插入技巧并不可靠。一般来说，这种技巧的效果很差。例如，数据的大步跳跃会导致功率谱在低频成

分处(波长和步长相比)产生伪凸起。

对非均匀取样数据谱分析的一个完全不同的方法,能够克服上面的困难,并且具有一些让人满意的其他性质,它是由 Lomb[1]发展起来的,他的工作是建立在早期 Barning[2]和 Vaníček[3]工作的基础上的,并由 Scargle[4]进一步完善。Lomb 方法(我们以后都这样称呼)仅估计在实际测量的时刻 t_i 时的正弦的或余弦的数据。假设有 N 个数据点 $h_i \equiv h(t_i)(i=0,\ldots,N-1)$,则首先通过一般的关系求均值和方差:

$$\bar{h} \equiv \frac{1}{N}\sum_{i=0}^{N-1} h_i \qquad \sigma^2 \equiv \frac{1}{N-1}\sum_{i=0}^{N-1}(h_i-\bar{h})^2 \tag{13.8.3}$$

现在,按下式定义 **Lomb 归一化周期图**(功率谱作为角频率 $\omega \equiv 2\pi f > 0$ 的函数):

$$P_N(\omega) \equiv \frac{1}{2\sigma^2}\left\{\frac{\left[\sum_j (h_j-\bar{h})\cos\omega(t_j-\tau)\right]^2}{\sum_j \cos^2\omega(t_j-\tau)} + \frac{\left[\sum_j (h_j-\bar{h})\sin\omega(t_j-\tau)\right]^2}{\sum_j \sin^2\omega(t_j-\tau)}\right\} \tag{13.8.4}$$

其中 τ 由下面的关系式定义:

$$\tan(2\omega\tau) = \frac{\sum_j \sin 2\omega t_j}{\sum_j \cos 2\omega t_j} \tag{13.8.5}$$

常数 τ 是一种偏移量,它使得 $P_N(\omega)$ 在所有 t_i 移动一个常数时不变。Lomb 说明这种偏移量的选择还有一个更深远的效应:它使(13.8.4)式和通过最小二乘法对给定频率 ω 的谐振信号做估计所得的等式一致,其中谐振信号可以是如下的信号模型:

$$h(t) = A\cos\omega t + B\sin\omega t \tag{13.8.6}$$

此事实说明了为什么这种方法给出比 FFT 方法更好的结果:它对数据的估计建立在"各点"的基础上,而不是建立在"每一时间间隔"上,而后者在非均匀取样时会带来严重的误差。

一个非常普遍的情况是,被测量的数据点 h_i 是周期信号和独立(白的)高斯噪声之和。如果试图确定这种周期信号的出现和消失,则必须对下面的问题给予一个定量的回答,"在功率谱 $P_N(\omega)$ 中,一个峰应明显到怎样程度?"在这个问题中,虚的假设是假设数据也是独立的高斯分布的随机值。Lomb 归一化周期图的一个非常好的性质是,虚假设的可行性可以比较严格地验证,下面进行讨论。

"归一化"涉及(13.8.4)式分母中因子 σ^2。Scargle[4]证明在这种归一化下,在任何特定频率 ω 和虚假设的前提下,$P_N(\omega)$ 具有均值为 1 的指数概率分布。换言之,$P_N(\omega)$ 在 z 和 $z+dz$ 之间的概率是 $\exp(-z)\,dz$。还可以看出,如果对某 M 个独立的频率扫描,则不大于 z 值的概率是 $(1-e^{-z})^M$,因此

$$P(>z) \equiv 1-(1-e^{-z})^M \tag{13.8.7}$$

是虚假设的虚警概率,它也是我们所能看到的 $P_N(\omega)$ 中任意尖峰的**显著性水平**。虚警概率的一个小值指出了一个非常明显的周期信号的存在。

为了评估这种显著性,需要知道 M。事实上,观察的频率越多,谱中的尖峰越不明显(如果观察得足够长,可能什么也没有)。一个典型的过程是,在比较大的频率范围内把 $P_N(\omega)$ 绘成划分间隔很细的空间频率的函数。这些频率中有多少个量是独立的呢?

在回答这个问题之前,让我们首先看一看 M 需要达到多高的精度。我们感兴趣的区间是显著性为一个小数,即远离 1 的位置。所以(13.8.7)式可以通过级数展开为

$$P(>z) \approx Me^{-z} \tag{13.8.8}$$

我们看到显著性与 M 成线性比例关系。实际的显著性水平是 0.05,0.01 和 0.001 等数。在评估显著性中即使存在 ±50% 的误差也是可容忍的,因为所引用的显著性水平通常是根据因子 5 或 10 来划分的。因此对 M 的估计不必非常精确。

Horne 和 Baliunas[5]从大量的蒙特卡罗(Monte Carlo)实验中给出了不同情况下如何确定 M 的取值。通常,M 依赖于取样的频率数、数据点 N 的值以及它们的划分间隔。当数据点接近等间距划分时,并且当取样的频率"布满"了从零到 Nyquist 频率 f_c((13.8.2)式)间的所有频率范围时,可以发现 M 和 N 非常接近。还有,对于随机间隔的数据点或等距间隔的数据点,M 的值没有很大的区别。当在大于 Nyquist 频率的频率范围内取样时,M 成比例地增加。只有在这种情况下 M 的取值将明显地不同于等距间隔的情况,即数据点被紧密聚集在一起,例如数据聚集成 3 组,那么(可以想像)独立的频率数将减少大约 1/3。

下面的程序 period 基于上述的的规则来计算有效的 M 值，并且假设没有任何重要的数据聚集。在大多数情况下这种计算是足够的。但在一些特殊情况下，如果确定麻烦的话，用简单的蒙特卡罗方法计算更好的 M 值是不太困难的：用固定数目的数据点以及对应的位置 t_i，产生具有高斯（正态）方差分布的合成数据组，对这样的数据组找出 $P_N(\omega)$ 的最大值（用附带的程序），再根据（13.8.7）式对所得的数据分布拟合求得 M_c。

图 13.8.1 显示了应用上面讨论的方法所得的结果。在上图中，数据点是按时间轴绘出的。数据点的个数 $N=100$，其随时间 t 的分布是泊松分布。当然，看不出任何一点关于含有正弦信号的信息。而下图是按频率 $f=\omega/2\pi$ 绘制的 $P_N(\omega)$。如果数据点是按均匀间隔取样的话，那么获得的 Nyquist 临界频率将是 $f=f_c=0.5$。因为现在要搜索的频率范围是临界频率的两倍，又因为对频率 f 过量取样的点其 $P_N(\omega)$ 的连续值应是光滑地变化，所以取 $M=2N$。图中水平的虚线和点线从下至上依次表示显著性水平为 0.5，0.1，0.05，0.01，0.005和 0.001。可以看出，在频率 0.81 处有一个非常显著的尖峰。这正是出现在数据中的正弦波频率。

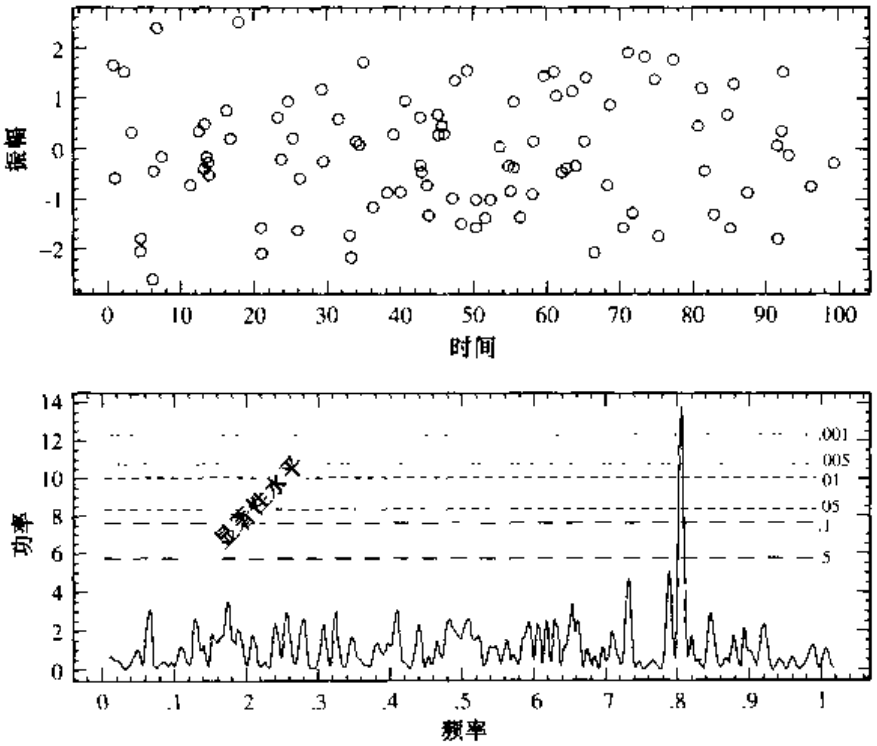


图 13.8.1　实施 Lomb 算法的实例。上图中在 0 到 100 的时间范围内 100 个数据是随机分布的。数据中的正弦分量在显著性水平优于0.001处被算法鉴别出来。如果100个数据点以单位时间间隔均匀地分隔，那么Nyquist临界频率将是0.5。注意，对非均匀时间间隔分隔的点，在Nyquist频率范围内也没有明显的混迭

注意图中还有另外两个峰，但都没有超过 50% 的显著性水平，它也可能是期望被选取的。还有一个值得注意的事实是：那个明显的峰是在 Nyquist 频率以上找到的（是正确的），而且在 Nyquist 频率之内不引起任何明显的混迭。这一点对均匀间隔取样的数据是不可能的。在这里成为可能因为一些数据点的随机时间取样间隔比"平均"取样率对应的时间间隔更紧密，而这就排除了混迭的模糊性。

用程序实施归一化的周期图是很直接的，但是有几点必须记住。正在进行的是一个很慢的算法。一般来说，对于 N 个数据点，可能希望实验 $2N$ 或 $4N$ 个频率点。在（13.8.4）式和（13.8.5）式中，每组频率和数据点的组合不仅仅是一些加法和乘法，而且是四次调用三角函数，运算很容易就达到 N^2 的几百倍。用递归代替这些三角函数的调用就非常必要，这将使运算速度提高 4 倍。但是这种做法仅在被检测的频率序列是线性序列时才可能，因为这种序列是大多数用户所需要的，所以将把这种算法付诸实施。

在本节的最后将叙述一种方法，它近似于估计等式（13.8.4）和（13.8.5），但是具有一定的近似精度，这种方法比较快，运算量达 $N\log N$[6]。这种快速方法适用于长数据组。

要检测的、独立的最低频率 f 是输入信号时间长度 $\max_i(t_i)-\min_i(t_i)\equiv T$ 的倒数。这个频率能使数据包括在一个完整周期内。再减去数据的平均值,(13.8.4)式已假设对数据的零频部分不感兴趣——零频对应的恰好是均值。在 FFT 方法中,较高的独立频率将是 $1/T$ 的整数倍。因为我们感兴趣的是尖峰出现的统计显著性,因此取样间隔最好比 $1/T$ 更精细一些,以便使取样点更接近任何峰的顶点。附带的程序中包含有一个过量取样参数,称为 ofac,实际应用中通常取 ofac ≥ 4。我们也必须标明最高频率 f_{hi},f_{hi} 的选择是将它和 Nyquist 频率 f_c 相比,如果在同样的时间间隔 T 内均匀选取 N 个数据点,那么 $f_c=N/(2T)$。附带的程序中还包括一个输入参数 hifac,它定义为 f_{hi}/f_c。由程序返回的不同频率数目 N_P 由下式给出:

$$N_P=\frac{\mathtt{ofac}\times\mathtt{hifac}}{2}N \tag{13.8.9}$$

(记住输出数组的长度至少为 N_P。)

程序对三角函数递归采用双精度,虽然余下的程序可以是单精度计算的,同时利用一些三角函数恒等关系以减小舍入误差。如果读者对此感兴趣的话可以仔细推敲。最后要注意的一个细节是,如果 z 太大的话,则由于舍入误差,等式(13.8.7)将无效,但(13.8.8)式在这方法中仍是有效的。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::period(Vec_I_DP &x, Vec_I_DP &y, const DP ofac, const DP hifac,
    Vec_O_DP &px, Vec_O_DP &py, int &nout, int &jmax, DP &prob)
给定 n 个数据点,横坐标为 x[0..n-1](这并不需要均匀间隔),纵坐标为 y[0..n-1],并且已知所需的过量取样因子为
ofac(典型值为 4 或更大)。程序返回数组 px[0..np-1],其频率(不是角频率)顺序增加到 hifac 乘以"平均"的 Nyquist 频
率,以及数组 py[0..np-1],其值为上面的各频率处的 Lomb 归一化周期图。数组 x 和 y 维持不变。np 是 px 和 py 的长
度,应大于等于(13.8.9)式得到的频率点数目 nout。程序还返回 jmax,以使 py[jmax] 成为数组 py 中的最大元素。程序返
回的 prob,是对于随机噪声假设的最大显著性估算值。较小的 prob 表明有一个明显的周期信号出现。
{
    const DP TWOPI=6.28318530717958647 6;
    int i,j;
    DP ave,c,cc,cwtau,effm,expy,pnow,pymax,s,ss,sumc,sumcy,sums,sumsh,
        sumsy,swtau,var,wtau,xave,xdif,xmax,xmin,yy,arg,wtemp;

    int n=x.size();
    int np=px.size();
    Vec_DP wi(n),wpi(n),wpr(n),wr(n);
    nout=0.5*ofac*hifac*n;
    if (nout > np) nrerror("output arrays too short in period");
    avevar(y,ave,var);                          求输入数据的均值和方差。
    if (var == 0.0) nrerror("zero variance in period");
    xmax=xmin=x[0];                             扫描数据,确定横坐标的范围。
    for (j=0;j<n;j++) {
        if (x[j] > xmax) xmax=x[j];
        if (x[j] < xmin) xmin=x[j];
    }
    xdif=xmax-xmin;
    xave=0.5*(xmax+xmin);
    pymax=0.0;
    pnow=1.0/(xdif*ofac);                       起始频率。
    for (j=0;j<n;j++) {                         在每个数据点对三角函数
        arg=TWOPI*((x[j]-xave)*pnow);           递归进行初始赋值。
        wpr[j]= -2.0*SQR(sin(0.5*arg));
        wpi[j]=sin(arg);
        wr[j]=cos(arg);
        wi[j]=wpi[j];
    }
    for (i=0;i<nout;i++) {                      对要估计的频率的主循环。
        px[i]=pnow;
        sumsh=sumc=0.0;
        for (j=0;j<n;j++) {                     首先,对数据循环得到和τ相关的量。
            c=wr[j];
            s=wi[j];
            sumsh += s*c;
            sumc += (c-s)*(c+s);
        }
        wtau=0.5*atan2(2.0*sumsh,sumc);
```

```
        swtau=sin(wtau);
        cwtau=cos(wtau);
        sums=sumc=sumsy=sumcy=0.0;                    然后，再对数据循环得到周期图
        for (j=0;j<n;j++) {                           的值。
            s=wi[j];
            c=wr[j];
            ss=s*cwtau-c*swtau;
            cc=c*cwtau+s*swtau;
            sums += ss*ss;
            sumc += cc*cc;
            yy=y[j]-ave;
            sumsy += yy*ss;
            sumcy += yy*cc;
            wr[j]=((wtemp=wr[j])*wpr[j]-wi[j]*wpi[j])+wr[j];
            wi[j]=(wi[j]*wpr[j]+wtemp*wpi[j])+wi[j];      对三角函数递归进行更新。
        }
        py[i]=0.5*(sumcy*sumcy/sumc+sumsy*sumsy/sums)/var;
        if (py[i] >= pymax) pymax=py[jmax=i];
        pnow += 1.0/(ofac*xdif);                      下一个频率。
    }
    expy=exp(-pymax);                                 求最大值的统计显著性。
    effm=2.0*nout/ofac;
    prob=effm*expy;
    if (prob > 0.01) prob=1.0-pow(1.0-expy,effm);
}
```

13.8.1　Lomb 周期图快速计算

下面说明怎样快速近似计算(13.8.4)式和(13.8.5)式，虽然是近似得到的，但这种近似能达到想要的精度，并且其运算量为 $N_P \log N_P$ 的数量级。此方法要用到 FFT，但决不是数据的 FFT 周期图。它实际上是对(13.8.4)式和(13.8.5)式，即 Lomb 归一化周期图的估计，它完全具有 FFT 周期图的优点和缺点。由于 Press 和 Rybicki[6]的贡献，使得将 Lomb 方法运用到 10^6 个数据点的计算成为可能，它比用(13.8.4)式和(13.8.5)式对数据点为 60 到 100 时的直接计算还要快。

注意，(13.8.4)式和(13.8.5)式中的三角函数求和可以化为 4 个简单的求和。如果定义

$$S_h \equiv \sum_{j=0}^{N-1}(h_j - \bar{h})\sin(\omega t_j) \qquad C_h \equiv \sum_{j=0}^{N-1}(h_j - \bar{h})\cos(\omega t_j) \tag{13.8.10}$$

和

$$S_2 \equiv \sum_{j=0}^{N-1}\sin(2\omega t_j) \qquad C_2 \equiv \sum_{j=0}^{N-1}\cos(2\omega t_j) \tag{13.8.11}$$

那么

$$\begin{aligned}
&\sum_{j=0}^{N-1}(h_j - \bar{h})\cos\omega(t_j - \tau) = C_h\cos\omega\tau + S_h\sin\omega\tau \\
&\sum_{j=0}^{N-1}(h_j - \bar{h})\sin\omega(t_j - \tau) = S_h\cos\omega\tau - C_h\sin\omega\tau \\
&\sum_{j=0}^{N-1}\cos^2\omega(t_j - \tau) = \frac{N}{2} + \frac{1}{2}C_2\cos(2\omega\tau) + \frac{1}{2}S_2\sin(2\omega\tau) \\
&\sum_{j=0}^{N-1}\sin^2\omega(t_j - \tau) = \frac{N}{2} - \frac{1}{2}C_2\cos(2\omega\tau) - \frac{1}{2}S_2\sin(2\omega\tau)
\end{aligned} \tag{13.8.12}$$

现在注意，如果时间 t_j 是均匀划分的，那么 S_h，C_h，S_2 和 C_2 可用两个复数的 FFT 计算，结果代入(13.8.12)式，再由(13.8.12)式可估算(13.8.4)式和(13.8.5)式。因此余下的问题是，在非均匀间隔的数据点处对(13.8.10)式和(13.8.11)式进行估算。

答案是采用内插或倒插值，我们称倒插值为**扩展插值**。如同经典描述的那样，插值使用在一规则网格上

的几个函数值来构造任意点处的精确近似值。扩展插值恰好相反,它是用规则网络上的几个函数值代替任意点上的函数值,并使网格上各点的和是原来任意点之和的精确近似。

不难看出,扩展插值的权函数等价于一般插值的权函数。假设要插值的函数 $h(t)$ 仅在离散点(非均匀间隔)上知道其值 $h(t_i)\equiv h_i$,而函数 $g(t)$(例如它可以是 $\cos\omega t$)在任意处都能估计其值。令 $\hat{t}_k$ 为一个规则网格上均匀间隔的点序列,那么拉格朗日(Lagrange)插值(参见 3.1 节)给出下面形式的一个近似:

$$g(t)\approx\sum_k w_k(t)g(\hat{t}_k) \tag{13.8.13}$$

其中,$w_k(t)$ 是插值的权重。通过下式来估算我们感兴趣的和式:

$$\sum_{j=0}^{N-1}h_j g(t_j)\approx\sum_{j=0}^{N-1}h_j\left[\sum_k w_k(t_j)g(\hat{t}_k)\right]=\sum_k\left[\sum_{j=0}^{N-1}h_j w_k(t_j)\right]g(\hat{t}_k)\equiv\sum_k\hat{h}_k\,g(\hat{t}_k) \tag{13.8.14}$$

其中,$\hat{h}_k\equiv\sum_j h_j w_k(t_j)$。注意(13.8.14)式用规则网格上的和值代替原先的和值。还请注意(13.8.13)式的精确度取决于函数 g 划分的网格精细程度,而和时间点 t_j 的间隔及函数 h 无关。(13.8.14)式的精确度也有这种性质。

因此这种快速估计方法的一般概括是:(i)选择一个足够大的网格能容纳想要的过量取样因子,还要能在感兴趣的最高频率的每半个波长处有几个外插点;(ii)在网格上外插值 h_i,并且进行 FFT,这就给出(13.8.10)式中的 S_h 和 C_h;(iii)将常数值 1 外插到另一个网格上,并进行 FFT;接着通过一些运算计算出(13.8.11)式中的 S_2 和 C_2;(iv)依次估算(13.8.12)式、(13.8.5)式和(13.8.4)式。

为了更有效地实施这种算法,还有一些其他的技巧。从程序中读者可以推测出大多数的技巧,但将说明以下几点:(a)用频率 2ω 处的变换值代替 ω 处的值的技巧是,将时间域的数据扩展为 2 倍,接着双倍覆盖初始的长度(这种技巧归功于 Tukey)。在程序中,这将以模函数的形式出现;(b)从(13.8.5)式的左边到各种所需的 $\omega\tau$ 的三角函数,都用到了三角函数的恒等式。C++ 语言中标识符如 cwt 和 hs2wt 代表了量 $\cos\omega\tau$ 和 $\frac{1}{2}\sin(2\omega\tau)$;(c)函数 spread 在围绕任意点的 M 个最靠近中心网格点处实施扩展插值,它的附加程序用非常有效的方法估算了拉格朗日插值多项式的系数

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::fasper(Vec_I_DP &x, Vec_I_DP &y, const DP ofac, const DP hifac,
    Vec_O_DP &wk1, Vec_O_DP &wk2, int &nout, int &jmax, DP &prob)
给定 n 个数据点的横坐标 x[0..n-1](并不需要是均匀间隔)和纵坐标 y[0..n-1],还给定期望的过量取样因子 ofac(典
型值为 4 或更大),此程序返回数组 wk1[0..mwk-1],它是 nout 个递增的频率(不是角频率)序列,最大值为 hifac 乘以"平
均"的 Nyquist 频率,还返回数组 wk2[0..mwk-1],其值是在这些频率处的 Lomb 归一化周期图的值。数组 x 和 y 保持不
变。数组 wk1 和 wk2 的维数 mwk 必须足够大,以便有中介计算空间,否则会出错。程序还返回 jmax,以使 wk2[jmax]成为
数组 wk2 中的最大元素,并且还返回 prob,它是在随机噪声假设下最大显著性的估算值。较小的 prob 值说明有一个明显
的周期信号出现。
{
    const int MACC=4;
    int j,k,ndim,nfreq,nfreqt;
    DP ave,ck,ckk,cterm,cwt,den,df,effm,expy,fac,fndim,hc2wt,hs2wt,
        hypo,pmax,sterm,swt,var,xdif,xmax,xmin;

    int n=x.size();
    int nwk=wk1.size();
    nout=0.5*ofac*hifac*n;
    nfreqt=ofac*hifac*n*MACC;                    将FFT的大小定为大于nfreqt的2的最小幂次。
    nfreq=64;
    while (nfreq < nfreqt) nfreq <<= 1;
    ndim=nfreq << 1;
    if (ndim > nwk) nrerror("workspaces too small in fasper");
    avevar(y,ave,var);                           计算数据的均值、方差和范围。
    if (var == 0.0) nrerror("zero variance in fasper");
```

```
    xmin=x[0];
    xmax=xmin;
    for (j=1;j<n;j++) {
        if (x[j] < xmin) xmin=x[j];
        if (x[j] > xmax) xmax=x[j];
    }
    xdif=xmax-xmin;
    Vec_DP wk1_t(0.0,ndim);                    置工作空间为0。
    Vec_DP wk2_t(0.0,ndim);
    fac=ndim/(xdif*ofac);
    fndim=ndim;
    for (j=0;j<n;j++) {                        外插数据于工作空间。
        ck=fmod((x[j]-xmin)*fac,fndim);
        ckk=2.0*(ck++);
        ckk=fmod(ckk,fndim);
        ++ckk;
        spread(y[j]-ave,wk1_t,ck,MACC);
        spread(1.0,wk2_t,ckk,MACC);
    }
    realft(wk1_t,1);                           进行快速傅里叶变换。
    realft(wk2_t,1);
    df=1.0/(xdif*ofac);
    pmax = -1.0;
    for (k=2,j=0;j<nout;j++,k+=2) {            对每个频率计算Lomb值。
        hypo=sqrt(wk2_t[k]*wk2_t[k]+wk2_t[k+1]*wk2_t[k+1]);
        hc2wt=0.5*wk2_t[k]/hypo;
        hs2wt=0.5*wk2_t[k+1]/hypo;
        cwt=sqrt(0.5+hc2wt);
        swt=SIGN(sqrt(0.5-hc2wt),hs2wt);
        den=0.5*n+hc2wt*wk2_t[k]+hs2wt*wk2_t[k+1];
        cterm=SQR(cwt*wk1_t[k]+swt*wk1_t[k+1])/den;
        sterm=SQR(cwt*wk1_t[k+1]-swt*wk1_t[k])/(n-den);
        wk1[j]=(j+1)*df;
        wk2[j]=(cterm+sterm)/(2.0*var);
        if (wk2[j] > pmax) pmax=wk2[jmax=j];
    }
    expy=exp(-pmax);                           估计最大峰值的显著性。
    effm=2.0*nout/ofac;
    prob=effm*expy;
    if (prob > 0.01) prob=1.0-pow(1.0-expy,effm);
}


#include "nr.h"

void NR::spread(const DP y, Vec_IO_DP &yy, const DP x, const int m)
给定数组 yy[0..n-1]扩展插值 y 到 m 个实际的数组元素中.这些元素非常近似"虚构"(也就是可能不是整数)的数组元
素 x. 所用的权重是拉格朗日插值多项式的系数.
{
    static int nfac[11]={0,1,1,2,6,24,120,720,5040,40320,362880};
    int ihi,ilo,ix,j,nden;
    DP fac;

    int n=yy.size();
    if (m > 10) nrerror("factorial table too small in spread");
    ix=int(x);
    if (x == DP(ix)) yy[ix-1] += y;
    else {
        ilo=MIN(MAX(int(x-0.5*m),0),int(n-m));
        ihi=ilo+m;
        nden=nfac[m];
        fac=x-ilo-1;
        for (j=ilo+1;j<ihi;j++) fac *= (x-j-1);
        yy[ihi-1] += y*fac/(nden*(x-ihi));
        for (j=ihi-1;j>ilo;j--) {
            nden=(nden/(j-ilo))*(j-ihi);
            yy[j-1] += y*fac/(nden*(x-j));
        }
    }
}
```

参考文献和深入阅读

Lomb, N.R. 1976, *Astrophysics and Space Science*, vol. 39, pp. 447–462. [1]

Barning, F.J.M. 1963, *Bulletin of the Astronomical Institutes of the Netherlands*, vol. 17, pp. 22–28. [2]

Vaníček, P. 1971, *Astrophysics and Space Science*, vol. 12, pp. 10–33. [3]

Scargle, J.D. 1982, *Astrophysical Journal*, vol. 263, pp. 835–853. [4]

Horne, J.H., and Baliunas, S.L. 1986, *Astrophysical Journal*, vol. 302, pp. 757–763. [5]

Press, W.H. and Rybicki, G.B. 1989, *Astrophysical Journal*, vol. 338, pp. 277–280. [6]

13.9 深入讨论:使用 FFT 计算傅里叶积分

在许多情况下,人们想计算如下积分式的精确数值:

$$I = \int_a^b e^{i\omega t} h(t)dt\,, \tag{13.9.1}$$

或等价的实部和虚部

$$I_c = \int_a^b \cos(\omega t)h(t)dt \qquad I_s = \int_a^b \sin(\omega t)h(t)dt\,, \tag{13.9.2}$$

的精确值。并且要对许多不同的 ω 值估算此积分式。在很多情况下,$h(t)$通常是一个光滑的函数,但并不需以$[a,b]$为周期,也并不要求它在 a 和 b 点处的值为 0。很明显,由于 FFT 的**巨大性能**,它应该能用于此问题。

让我们首先用最直观的方法来考虑这个问题,并且看看困难在哪里。将间隔$[a,b]$分成 M 个小的间隔,其中 M 是一个大的整数,并且定义

$$\Delta \equiv \frac{b-a}{M}, \quad t_j \equiv a + j\Delta, \quad h_j \equiv h(t_j), \quad j = 0,\ldots,M \tag{13.9.3}$$

注意 $h_0 \equiv h(a)$,$h_M = h(b)$,有 $M+1$ 个 h_j 值,可以用下面的和式近似积分 I,

$$I \approx \Delta \sum_{j=0}^{M-1} h_j \exp(i\omega t_j) \tag{13.9.4}$$

它具有一阶精度(如果将 h_j 和 t_j 置于小区间的中央,便能得到二阶精度)。现在对于 ω 和 M 的一些值,(13.9.4)式中的和式可以进行离散傅里叶变换,即 DFT,并且能通过 FFT(快速傅里叶)算法进行计算。特别是,可以选 M 为 2 的整数次幂,并且通过下式定义一组特殊的 ω 值:

$$\omega_m \Delta \equiv \frac{2\pi m}{M} \tag{13.9.5}$$

其中 m 的取值为 $m = 0,1,\ldots,M/2-1$,那么(13.9.4)式变为

$$I(\omega_m) \approx \Delta e^{i\omega_m a} \sum_{j=0}^{M-1} h_j e^{2\pi i m j/M} = \Delta e^{i\omega_m a}[\mathrm{DFT}(h_0 \ldots h_{M-1})]_m \tag{13.9.6}$$

虽然(13.9.6)式简单清楚,但不推荐使用此式,因为有可能导致错误的结果。

问题在于积分式(13.9.1)的振荡特性。如果 $h(t)$确实光滑,并且 ω 足够大,以致在间隔$[a,b]$中有几个周期——实际上(13.9.5)式中的 ω_m 恰好给出 m 个周期——那么积分值 I 通常很小,小到以致于很容易被一阶甚至二阶(利用中心值)的截断误差所淹没。还有,如果被积函数不是振荡的,出现在误差项中的特征"小参数"将不是$\Delta/(b-a) = 1/M$,而是 $\omega\Delta$。对于在 DFT 的 Nyquist 间隔中的 ω 值(参见(13.9.5)式),此值可以大到 π。其结果是当 ω 增加时,(13.9.6)式出现系统误差。

一个较好的练习就是用一个可以解析地求积函数对(13.9.6)式验证,就可以看出它的精确程度如何。建议读者试一试。

下面看一个更复杂的情况。给出采样点 h_j,可以利用 h_j 值附近的插值,在间隔$[a,b]$内各处逼近函数 $h(t)$。最简单的情况就是线性插值,有两个最接近的 h_j 值,一个在左边,一个在右边。高阶的插值,例如三次插值,左边有两点,右边有两点——除了数据中第一个间隔和最后间隔的情况,此时插值需在一边用三个 h_j 值,另一边则用一个。

这种插值方式是自变量 t 的(分段)多项式,但是系数却是 h_j 的线性函数。尽管通常不从这个角度考

虑，插值还是可以看成通过一系列核函数（只依赖于插值方式）和样本值（仅依赖于函数值）乘积之和来近似一个函数。可以写成

$$h(t) \approx \sum_{j=0}^{M} h_j\, \psi\left(\frac{t-t_j}{\Delta}\right) + \sum_{j=\text{endpoints}} h_j\, \varphi_j\left(\frac{t-t_j}{\Delta}\right) \tag{13.9.7}$$

其中$\psi(s)$是内点的核函数；如果 s 的绝对值足够大，那么它是零；仅当 s 处于和 h_j 的乘积确实是用来插值时，$\psi(s)$才不为零。因为正好在样本点上的插值应该给出采样函数值，所以永远有$\psi(0)=1$和$\psi(m)=0$，$m=\pm1,\pm2,\ldots$。对于线性插值，$\psi(s)$是分段线性的，当 s 在$(-1,0)$时，$\psi(s)$从 0 升到 1；当 s 在$(0,1)$时，$\psi(s)$又回到 0。更高阶的插值$\psi(s)$由拉格朗日插值多项式的几段构成。在整数 s 处，即各段连接处，由于插值点是离散地变化的，$\psi(s)$有间断的导数。

正如已经看到的那样，最靠近 a 和 b 的小区间要求不同的（非中心的）插值公式。这可以从(13.9.7)式中第二项得到反映，其中有一个特殊的端点核函数$\varphi_j(s)$。实际上，由于下面一些原因，我们将所有的点包括在第一个和式里（核函数为ψ），而φ_j实际上是真正的端点核函数和内点核函数ψ的差值。将任何阶插值的ψ_j写下来是一个枯燥但很直观的工作，每个φ_j由连接在一起的各段拉格朗日插值多项式的差值构成。

现在将积分运算$\int_a^b dt\,\exp(i\omega t)$运用到(13.9.7)式两边，交换积分和求和的顺序，并且对第一项求和进行变量替换 $s=(t-t_j)/\Delta$，对第二项求和进行变量替换 $s=(t-a)/\Delta$，结果是

$$I \approx \Delta e^{i\omega a}\left[W(\theta)\sum_{j=0}^{M} h_j e^{ij\theta} + \sum_{j=\text{endpoints}} h_j \alpha_j(\theta)\right] \tag{13.9.8}$$

其中 $\theta\equiv\omega\Delta$，函数 $W(\theta)$和$\alpha_j(\theta)$由下式定义：

$$W(\theta) \equiv \int_{-\infty}^{\infty} ds\, e^{i\theta s}\psi(s) \tag{13.9.9}$$

$$\alpha_j(\theta) \equiv \int_{-\infty}^{\infty} ds\, e^{i\theta s}\varphi_j(s-j) \tag{13.9.10}$$

非常重要的一点是，对任何插值方式，(13.9.9)式和(13.9.10)式可以解析地估算。(13.9.8)式表示了一个将"端点校正"运用到求和式的算法，此求和可以用 FFT 计算，由此给出高阶精度的结果。

将只考虑左右对称的插值，该对称意味着

$$\varphi_{M-j}(s) = \varphi_j(-s) \qquad \alpha_{M-j}(\theta) = e^{i\theta M}\alpha_j^*(\theta) = e^{i\omega(b-a)}\alpha_j^*(\theta) \tag{13.9.11}$$

其中的星号（*）代表了复共轭。还有$\psi(s)=\psi(-s)$意味着 $W(\theta)$是实数。

现在让我们看(13.9.8)式中的第一个和式，将用 FFT 方法计算。为了进行 FFT，首先选择 N，使得 $N\geq M+1$，并且 N 为 2 的整数次幂（注意 M 不必是 2 的幂次，因此 $M=N-1$ 是容许的）。如果 $N>M+1$，定义 $h_j=0$，$M+1<j\leq N-1$，即对数组 h_j 进行"零元填充"以使 j 的取值范围是 $0\leq j\leq N-1$。那么在下式确定的特殊值 $\omega=\omega_n$ 下，可对和式进行 DFT：

$$\omega_n\Delta \equiv \frac{2\pi n}{N} \equiv \theta \qquad n=0,1,\ldots,\frac{N}{2}-1 \tag{13.9.12}$$

对于固定的 M，N 选择得越大，频域的采样越细。另一方面，M 值还确定了最高的取样频率，因为当 M 增加时，Δ 减小((13.9.3)式)，而最大的 $\omega\Delta$ 值总是在 π 以下((13.9.12)式)。一般来说，过量取样因子至少取为 4 比较有利，即 $N>4M$（参看下面），最后可将(13.9.8)式写成如下的形式：

$$\begin{aligned} I(\omega_n) = \Delta e^{i\omega_n a}\Big\{ & W(\theta)[\mathrm{DFT}(h_0\ldots h_{N-1})]_n \\ & + \alpha_0(\theta)h_0 + \alpha_1(\theta)h_1 + \alpha_2(\theta)h_2 + \alpha_3(\theta)h_3 + \cdots \\ & + e^{i\omega(b-a)}\left[\alpha_0^*(\theta)h_M + \alpha_1^*(\theta)h_{M-1} + \alpha_2^*(\theta)h_{M-2} + \alpha_3^*(\theta)h_{M-3} + \ldots\right]\Big\} \end{aligned} \tag{13.9.13}$$

对于三次（或更低）的多项式插值，上式中明显地写出来的各项大多是非零值，而用省略号（...）略去的各项可以被忽略，因此只需要由(13.9.9)式和(13.9.10)式计算求得函数 $W,\alpha_0,\alpha_1,\alpha_2,\alpha_3$ 的明显表达式。对梯形（二阶）和立方（四阶）的情况已经计算出了表达式，下面是它们的结果，并且给出了对于 θ 较小时，它们的幂级数展开的前几项：

梯形情况:

$$W(\theta)=\frac{2(1-\cos\theta)}{\theta^2}\approx 1-\frac{1}{12}\theta^2+\frac{1}{360}\theta^4-\frac{1}{20160}\theta^6$$

$$\begin{aligned}\alpha_0(\theta)&=-\frac{(1-\cos\theta)}{\theta^2}+i\frac{(\theta-\sin\theta)}{\theta^2}\\&\approx-\frac{1}{2}+\frac{1}{24}\theta^2-\frac{1}{720}\theta^4+\frac{1}{40320}\theta^6+i\theta\left(\frac{1}{6}-\frac{1}{120}\theta^2+\frac{1}{5040}\theta^4-\frac{1}{362880}\theta^6\right)\end{aligned}$$

$$\alpha_1=\alpha_2=\alpha_3=0$$

立方情况:

$$W(\theta)=\left(\frac{6+\theta^2}{3\theta^4}\right)(3-4\cos\theta+\cos 2\theta)\approx 1-\frac{11}{720}\theta^4+\frac{23}{15120}\theta^6$$

$$\begin{aligned}\alpha_0(\theta)&=\frac{(-42+5\theta^2)+(6+\theta^2)(8\cos\theta-\cos 2\theta)}{6\theta^4}+i\frac{(-12\theta+6\theta^3)+(6+\theta^2)\sin 2\theta}{6\theta^4}\\&\approx-\frac{2}{3}+\frac{1}{45}\theta^2+\frac{103}{15120}\theta^4-\frac{169}{226800}\theta^6+i\theta\left(\frac{2}{45}+\frac{2}{105}\theta^2-\frac{8}{2835}\theta^4+\frac{86}{467775}\theta^6\right)\end{aligned}$$

$$\begin{aligned}\alpha_1(\theta)&=\frac{14(3-\theta^2)-7(6+\theta^2)\cos\theta}{6\theta^4}+i\frac{30\theta-5(6+\theta^2)\sin\theta}{6\theta^4}\\&\approx\frac{7}{24}-\frac{7}{180}\theta^2+\frac{5}{3456}\theta^4-\frac{7}{259200}\theta^6+i\theta\left(\frac{7}{72}-\frac{1}{168}\theta^2+\frac{11}{72576}\theta^4-\frac{13}{5987520}\theta^6\right)\end{aligned}$$

$$\begin{aligned}\alpha_2(\theta)&=\frac{-4(3-\theta^2)+2(6+\theta^2)\cos\theta}{3\theta^4}+i\frac{-12\theta+2(6+\theta^2)\sin\theta}{3\theta^4}\\&\approx-\frac{1}{6}+\frac{1}{45}\theta^2-\frac{5}{6048}\theta^4+\frac{1}{64800}\theta^6+i\theta\left(-\frac{7}{90}+\frac{1}{210}\theta^2-\frac{11}{90720}\theta^4+\frac{13}{7484400}\theta^6\right)\end{aligned}$$

$$\begin{aligned}\alpha_3(\theta)&=\frac{2(3-\theta^2)-(6+\theta^2)\cos\theta}{6\theta^4}+i\frac{6\theta-(6+\theta^2)\sin\theta}{6\theta^4}\\&\approx\frac{1}{24}-\frac{1}{180}\theta^2+\frac{5}{24192}\theta^4-\frac{1}{259200}\theta^6+i\theta\left(\frac{7}{360}-\frac{1}{840}\theta^2+\frac{11}{362880}\theta^4-\frac{13}{29937600}\theta^6\right)\end{aligned}$$

下面的程序 dftcor,对立方情况实施端点校正。给定输入值 ω、Δ、a、b 和具有 8 个值 $h_0,\ldots,h_3$,$h_{N-3},\ldots,h_N$ 的数组,它返回(13.9.13)式中的端点校正的实部和虚部以及因子 $W(\theta)$。程序的代码比较复杂,这完全是由于上面的表达式很复杂。上面的表达式中还放弃了 θ 的高次幂项,因此即使校正项只需单精度时,还是有必要用双精度计算等式的右边。对小值 θ 有必要采用级数展开。θ 的最优截取值是由机器的字长决定,但是也可以用实验确定它。在最大值处,这两种方法相对于机器精度给出同样的结果。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::dftcor(const DP w, const DP delta, const DP a, const DP b,
    Vec_I_DP &endpts, DP &corre, DP &corim, DP &corfac)
```

对于用离散傅里叶变换来近似求积分,此程序用来计算乘以 DFT 的校正因子和要加在端点上的校正项。输入是角频率 w,步长 delta,积分的上、下限 a 和 b,而数组 endpts 包含最初和最后 4 个函数值。校正因子 $W(\theta)$ 以 corfac 返回,端点校正项的实部和虚部分别以 corre 和 corim 返回。

```
{
    DP a0i,a0r,a1i,a1r,a2i,a2r,a3i,a3r,arg,c,cl,cr,s,sl,sr,t,t2,t4,t6,
        cth,ctth,spth2,sth,sth4i,atth,th,th2,th4,tmth2,tth4i;

    th=w*delta;
    if (a >= b || th < 0.0e0 || th > 3.1416e0)
        nrerror("bad arguments to dftcor");
    if (fabs(th) < 5.0e-2) {        使用级数。
        t=th;
        t2=t*t;
        t4=t2*t2;
        t6=t4*t2;
```

```
        corfac=1.0-(11.0/720.0)*t4+(23.0/15120.0)*t6;
        a0r=(-2.0/3.0)+t2/45.0+(103.0/15120.0)*t4-(169.0/226800.0)*t6;
        a1r=(7.0/24.0)-(7.0/180.0)*t2+(5.0/3456.0)*t4-(7.0/259200.0)*t6;
        a2r=(-1.0/6.0)+t2/45.0-(5.0/6048.0)*t4+t6/64800.0;
        a3r=(1.0/24.0)-t2/180.0+(5.0/24192.0)*t4-t6/259200.0;
        a0i=t*(2.0/45.0+(2.0/105.0)*t2-(8.0/2835.0)*t4+(86.0/467775.0)*t6);
        a1i=t*(7.0/72.0-t2/168.0+(11.0/72576.0)*t4-(13.0/6987520.0)*t6);
        a2i=t*(-7.0/90.0+t2/210.0-(11.0/90720.0)*t4+(13.0/7484400.0)*t6);
        a3i=t*(7.0/360.0-t2/840.0+(11.0/362880.0)*t4-(13.0/29937600.0)*t6);
    } else {                    Use trigonometric formulas.
        cth=cos(th);
        sth=sin(th);
        ctth=cth*cth-sth*sth;
        stth=2.0e0*sth*cth;
        th2=th*th;
        th4=th2*th2;
        tmth2=3.0e0-th2;
        spth2=6.0e0+th2;
        sth4i=1.0/(6.0e0*th4);
        tth4i=2.0e0*sth4i;
        corfac=tth4i*spth2*(3.0e0-4.0e0*cth+ctth);
        a0r=sth4i*(-42.0e0+5.0e0*th2+spth2*(8.0e0*cth-ctth));
        a0i=sth4i*(th*(-12.0e0+6.0e0*th2)+spth2*stth);
        a1r=sth4i*(14.0e0*tmth2-7.0e0*spth2*cth);
        a1i=sth4i*(30.0e0*th-5.0e0*spth2*sth);
        a2r=tth4i*(-4.0e0*tmth2+2.0e0*spth2*cth);
        a2i=tth4i*(-12.0e0*th+2.0e0*spth2*sth);
        a3r=sth4i*(2.0e0*tmth2-spth2*cth);
        a3i=sth4i*(6.0e0*th-spth2*sth);
    }
    cl=a0r*endpts[0]+a1r*endpts[1]+a2r*endpts[2]+a3r*endpts[3];
    sl=a0i*endpts[0]+a1i*endpts[1]+a2i*endpts[2]+a3i*endpts[3];
    cr=a0r*endpts[7]+a1r*endpts[6]+a2r*endpts[5]+a3r*endpts[4];
    sr= -a0i*endpts[7]-a1i*endpts[6]-a2i*endpts[5]-a3i*endpts[4];
    arg=w*(b-a);
    c=cos(arg);
    s=sin(arg);
    corre=cl+c*cr-s*sr;
    corim=sl+s*cr+c*sr;
}
```

因为 dftcor 的用法有时让人困惑，所以给出一个说明性的程序 dftint，它针对一般的 a、b、ω 和 $h(t)$。利用 dftcor 计算(13.9.1)式。此程序中有几点值得注意：参数 M 和 NDFT 对应于上面讨论的 M 和 N。仅当 a 或 b 或 $h(t)$ 发生变化时，通过连续调用，重新计算傅里叶变换。

因为 dftint 是用来计算满足 $\omega\Delta<\pi$ 的任何 ω 值的情况，而不仅仅是由 DFT((13.9.12)式)返回的特殊值，所以在 DFT 谱上进行 MPOL 阶多项式插值。还有一点要提醒的是为了使这种插值精确，必须选一个大的过量取样因子($N \gg M$)。在插值后，将 dftcor 中的端点校正加上，它可以在任何 ω 值处估算。

尽管 dftcor 非常好用，但 dftint 仅起说明的作用。它不是一个适于所有目标的程序，因为它不能将其参数 M，NDFT，MPOL 或它的插值方式用到任何特殊函数 $h(t)$ 中。需根据自己的实际情况进行实验。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::dftint(DP func(const DP), const DP a, const DP b, const DP w,
    DP &cosint, DP &sinint)
这是一个说明如何使用程序 dftcor 的范例程序。用户应提供一个外函数 func，它返回量 h(t)。程序以 cosint 返回积分
∫_a^b cos(ωt)h(t)dt 值，以 sinint 返回积分 ∫_a^b sin(ωt)h(t)dt 的值。
{
    static int init=0;
    static DP (*funcold)(const DP);
    static DP aold = -1.e30,bold = -1.e30,delta;
    const int M=64,NDFT=1024,MPOL=6;
```

```
const DP TWOPI=6.283185307179586476;
int j,nn;
DP c,cdft,cerr,corfac,corim,corre,en,s,sdft,serr;
static Vec_DP data(NDFT),endpts(8);
Vec_DP cpol(MPOL),spol(MPOL),xpol(MPOL);

if (init != 1 || a != aold || b != bold || func != funcold) {
    是否需要初始赋值或R 改变ω?
    init=1;
    aold=a;
    bold=b;
    funcold=func;
    delta=(b-a)/M;
    for (j=0;j<M+1;j++)                          将函数值赋于data数组。
        data[j]=func(a+j*delta);
    for (j=M+1;j<NDFT;j++)                       对data数组的其余部分进行零元填充。
        data[j]=0.0;
    for (j=0;j<4;j++) {                          对端点赋值。
        endpts[j]=data[j];
        endpts[j+4]=data[M-3+j];
    }
    realft(data,1);
    realft 在 data[1]中返回的是对应于 ω_{N/2} 无用的值。实际上,这个元素包含对应 ω_0 的虚部,所以其值为 0。
    data[1]=0.0;
}
对想要的频率在 DFT 结果上插值。如果频率是 ω_n,即量 en 是整数,则 cdft = data[2*en-2],sdft = data[2*en-1],并且可
以省略插值。
en=w*delta*NDFT/TWOPI+1.0;
nn=MIN(MAX(int(en-0.5*MPOL+1.0),1),NDFT/2-MPOL+1);    插值最左边的点。
for (j=0;j<MPOL;j++,nn++) {
    cpol[j]=data[2*nn-2];
    spol[j]=data[2*nn-1];
    xpol[j]=nn;
}
polint(xpol,cpol,en,cdft,cerr);
polint(xpol,spol,en,sdft,serr);
dftcor(w,delta,a,b,endpts,corre,corim,corfac);        求端点校正值和乘积因子W(θ)。
cdft *= corfac;
sdft *= corfac;
cdft += corre;
sdft += corim;                                        最后乘以Δ和 exp(iωa)。
c=delta*cos(w*a);
s=delta*sin(w*a);
cosint=c*cdft-s*sdft;
sinint=s*cdft+c*sdft;
}
```

有时候人们只对(13.9.5)式中的离散频率 ω_m 感兴趣,它是在区间$[a,b]$中有整数间隔处的点。对光滑的 $h(t)$,在这些 ω 处的积分 I 其幅度可比在这些 ω 之间频率处的积分幅度小很多,因为小间隔内两半的积分可能相互抵消(这也是为了插值的精确性必须过量取样的原因:$I(\omega)$在 ω_m 值附近处是小幅度的振荡值)。如果要求在这些 ω_m 处的积分值,而不采用大数目(可能不精确)的插值,那么必须使 N 为 M 的整数倍数(比较(13.9.5)式和(13.9.12)式)。在上面实行的方法中,N 必须至少是 $M+1$,因此最小的这样的乘数是 $2M$,导致因子 ~2 的不必要计算。或者可以导出类似(13.9.13)式的公式,但是从 DFT 中略去最后的样本函数 $h_M = h(b)$,而将它包括在 h_M 的端点校正项中。那么就可以设 $M = N$(2 的整数次幂),并且在没有附加开销下得到(13.9.5)式的特定频率。其修正公式如下:

$$
\begin{aligned}
I(\omega_m) = \Delta e^{i\omega_m a}\Big\{ & W(\theta)[\mathrm{DFT}(h_0 \ldots h_{M-1})]_m \\
& + \alpha_0(\theta)h_0 + \alpha_1(\theta)h_1 + \alpha_2(\theta)h_2 + \alpha_3(\theta)h_3 \\
& + e^{i\omega(b-a)}\left[A(\theta)h_M + \alpha_1^*(\theta)h_{M-1} + \alpha_2^*(\theta)h_{M-2} + \alpha_3^*(\theta)h_{M-3}\right]\Big\}
\end{aligned}
\tag{13.9.14}
$$

其中 $\theta \equiv \omega_m \Delta$,对于梯形情况,$A(\theta)$由下式给定:

$$A(\theta) = -\alpha_0(\theta) \tag{13.9.15}$$

对于立方情况：

$$\begin{aligned} A(\theta) &= \frac{(-6+11\theta^2)+(6+\theta^2)\cos 2\theta}{6\theta^4} - i\,\mathrm{Im}[\alpha_0(\theta)] \\ &\approx \frac{1}{3} + \frac{1}{45}\theta^2 - \frac{8}{945}\theta^4 + \frac{11}{14175}\theta^6 - i\,\mathrm{Im}[\alpha_0(\theta)] \end{aligned} \tag{13.9.16}$$

当计算光滑函数的傅里叶系数时，自然会得到像 $W(\theta)$ 这样的因子，有时称它们为稀释因子[1]。但是，为了得到精确的积分值，端点校正同样很重要。Narasimhan 和 Karthikeyan[2]给出了一个等式，从代数形式上看等价于我们的梯形公式。但是，他们的公式要求两个 FFT 估算，这是不必要的。这里采用的基本思想可以追溯到 Filon[3]在 1928 年发表的文章(在 FFT 之前)，他用 Simpson 法则(二次插值)。因为这种插值不是左右对称的，需要两个傅里叶变换。关于(13.9.4)式的另一个算法由 Lyness[4]给出，还可以参见本节参考文献[5]。据我们所知，导出的立方阶公式在以前的文献中未出现过

当积分范围是$(-\infty,\infty)$时，计算傅里叶变换将有一些问题。如果积分函数在无穷远处很快衰减，那么可以在足够大的 t 值处分段积分。例如对到 $+\infty$ 的积分可以写为

$$\begin{aligned} \int_a^\infty e^{i\omega t}h(t)\,dt &= \int_a^b e^{i\omega t}h(t)\,dt + \int_b^\infty e^{i\omega t}h(t)\,dt \\ &= \int_a^b e^{i\omega t}h(t)\,dt - \frac{h(b)e^{i\omega b}}{i\omega} + \frac{h'(b)e^{i\omega b}}{(i\omega)^2} - \cdots \end{aligned} \tag{13.9.17}$$

分段点 b 必须选择得足够大，以使剩下的(b,∞)积分很小。它的渐近展开形式中的系列项是通过分部积分得到的。在(a,b)内的积分可以用 dftint 计算。在计算容易的情况下保留尽可能多的渐近展开项。可以参见本节参考文献[6]中的示例。更强大的方法，特别适用于长尾函数的方法，却不用 FFT，在本节参考文献[7～9]中有叙述。

参考文献和深入阅读

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §2.3.4. [1]

Narasimhan, M.S. and Karthikeyan, M. 1984, *IEEE Transactions on Antennas & Propagation*, vol. 32, pp. 404–408. [2]

Filon, L.N.G. 1928, *Proceedings of the Royal Society of Edinburgh*, vol. 49, pp. 38–47. [3]

Giunta, G. and Murli, A. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 97–107. [4]

Lyness, J.N. 1987, in *Numerical Integration*, P. Keast and G. Fairweather, eds. (Dordrecht: Reidel). [5]

Pantis, G. 1975, *Journal of Computational Physics*, vol. 17, pp. 229–233. [6]

Blakemore, M., Evans, G.A., and Hyslop, J. 1976, *Journal of Computational Physics*, vol. 22, pp. 352–376. [7]

Lyness, J.N., and Kaper, T.J. 1987, *SIAM Journal on Scientific and Statistical Computing*, vol. 8, pp. 1005–1011. [8]

Thakkar, A.J., and Smith, V.H. 1975, *Computer Physics Communications*, vol. 10, pp. 73–79. [9]

13.10　小波变换

像快速傅里叶变换(FFT)一样，离散小波变换(DWT)是一个快速线性运算，它对长度是 2 整数次幂的数据向量进行操作，将它变换成同样长度但在数字上不同的向量。和 FFT 一样，小波变换是可逆的，事实上还是正交的——其逆变换，当看做是一个大矩阵时，就是变换的转置。因此 FFT 和 DWT 二者都可以看成是在函数空间中的旋转，从输入空间(或时间)域转换到一个不同的域，其中的基函数是单位向量 $\mathbf{e}_i$，或者从连续的极限角度看是 Dirac δ 函数。对于 FFT，这个新域的基函数是我们熟悉的正弦和余弦函数。在小波域中基函数要复杂一些，并且有一个特别的名字称“母函数”和“小波”。

当然，对于函数空间可能有无数多个基，它们中的大多数几乎没什么意思。关于小波基有意思的是，不

像正弦余弦函数,单个的小波函数在空间上是非常局部化的;同时,又像正弦和余弦函数一样,单个小波函数在频率或者(更精确地说)特征尺度上也是非常局部化的。如同在下面将会看到的,小波的这种双重局部性使得多种函数和算符变换到小波域时变得稀疏,或者在某一高精度下是稀疏的。和傅里叶域相比,一些运算,像卷积等,其计算变得很快;同样有一大类的运算——它可以利用稀疏性的运算——在小波域里计算变得也很快(参见本节参考文献[1])。

不像正弦和余弦定义了惟一的傅里叶变换,这里没有一个独特的小波集,实际上可能有无穷多个这样的函数集合。粗略地说,不同集合的小波在两个方面——空间上局部的疏密程度和光滑程度——进行权衡(还有一些更细微的区别)。

13.10.1 Daubechies 小波的滤波系数

一组特定的小波由一组称为**小波滤波系数**的特定的数集来标定。这里,我们将讨论限制在由 Daubechies 发现的一种滤波器[2]。这种滤波器包括的范围从高度局部到高度光滑。最简单的(也最局部的)一类,通常称为 DAUB4,仅有 4 个系数:$c_0,\ldots,c_3$。为了记号的简便,我们特别对这种滤波器进行讨论。

考虑下面的变换矩阵,将它作用于一数据列向量的左边:

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 & & & & & & \\ c_3 & -c_2 & c_1 & -c_0 & & & & & & \\ & & c_0 & c_1 & c_2 & c_3 & & & & \\ & & c_3 & -c_2 & c_1 & -c_0 & & & & \\ \vdots & \vdots & & & & & \ddots & & & \\ & & & & & & & c_0 & c_1 & c_2 & c_3 \\ & & & & & & & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & & & & & & & & c_0 & c_1 \\ c_1 & -c_0 & & & & & & & & c_3 & -c_2 \end{bmatrix} \tag{13.10.1}$$

其中的空白处表示零。注意此矩阵的结构:第一行产生一个数据与滤波系数 $c_0,\ldots,c_3$ 卷积的分量,与此类似,第三行、第五行和其余奇数行的结果一样。如果偶数行以这种形式出现,正负交替,那么矩阵将是循环的,也就是一般的卷积,可以用 FFT 方法计算(注意最后两行像具有周期边界条件下的卷积那样环绕起来)。但是偶数行并不是以系数 $c_0,\ldots,c_3$,而是以系数 $c_3,-c_2,c_1,-c_0$ 进行卷积。整个矩阵的作用就是进行两个相关的卷积,然后各去掉一半数值,将剩下的各一半交插在一起。

将滤波器 $c_0,\ldots,c_3$ 看成是一个光滑滤波器,称它为 H,H 有点像 4 个点的移动平均。那么因为负号,滤波器 $c_3,-c_2,c_1,-c_0$ 称为 G,不是一个光滑滤波器(在信号处理的文献中,H 和 G 被称为**求积镜像滤波器**[3])。实际上,c 值的选取是使得 G 对足够光滑的数据向量尽可能产生零响应。这可以通过要求序列 $c_3,-c_2,c_1,-c_0$ 有一定数目的没影矩来完成。当这是 p 矩的情形(从零开始)时,我们说这组小波满足"阶数为 p 的近似条件"。这导致 H 的输出在去掉一半以后,精确地代表了数据的"光滑"信息。G 的输出,同样去掉一半后,被称为数据的"细节"信息[4]。

要使这种特征有用,必须能从它的 $N/2$ 个光滑或(s 分量)和它的 $N/2$ 个细节(或 d 分量)重建长度为 N 的原始数据向量。这就要求矩阵(13.10.1)是正交的,以使它的逆矩阵就是转置矩阵:

$$\begin{bmatrix} c_0 & c_3 & & & \cdots & & & & & c_2 & c_1 \\ c_1 & -c_2 & & & \cdots & & & & & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 & & & & & & & \\ c_3 & -c_0 & c_1 & -c_2 & & & & & & & \\ & & & & \ddots & & & & & & \\ & & & & & c_2 & c_1 & c_0 & c_3 & & \\ & & & & & c_3 & -c_0 & c_1 & -c_2 & & \\ & & & & & & & c_2 & c_1 & c_0 & c_3 \\ & & & & & & & c_3 & -c_0 & c_1 & -c_2 \end{bmatrix} \tag{13.10.2}$$

可以马上看出,要使矩阵(13.10.2)是矩阵(13.10.1)的逆矩阵,当且仅当以下两个等式成立:

$$
\begin{aligned}
c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 1 \\
c_2 c_0 + c_3 c_1 &= 0
\end{aligned}
\tag{13.10.3}
$$

如果附加要求阶数为 $p=2$ 的近似条件，则还需要另外两个条件：

$$
\begin{aligned}
c_3 - c_2 + c_1 - c_0 &= 0 \\
0c_3 - 1c_2 + 2c_1 - 3c_0 &= 0
\end{aligned}
\tag{13.10.4}
$$

(13.10.3)式和(13.10.4)式总共有 4 个等式和 4 个未知量 $c_0, \ldots, c_3$，这 4 个等式第一次由 Daubechies 建立和求解。它的惟一解(能进行左右逆转)是

$$
\begin{aligned}
c_0 &= (1+\sqrt{3})/4\sqrt{2} \qquad & c_1 &= (3+\sqrt{3})/4\sqrt{2} \\
c_2 &= (3-\sqrt{3})/4\sqrt{2} \qquad & c_3 &= (1-\sqrt{3})/4\sqrt{2}
\end{aligned}
\tag{13.10.5}
$$

事实上，DAUB4 是小波组中惟一最紧密的序列：如果有 6 个系数而不是 4 个系数，那么在(13.10.3)式中需要 3 个正交条件(偏置值分别为 0,2 和 4)，在(13.10.4)式中则需要 $p=3$ 的没影矩。在这种情况下解得的系数，DAUB6 也可以用非常紧凑的形式表示：

$$
\begin{aligned}
c_0 &= (1+\sqrt{10}+\sqrt{5+2\sqrt{10}})/16\sqrt{2} \qquad & c_1 &= (5+\sqrt{10}+3\sqrt{5+2\sqrt{10}})/16\sqrt{2} \\
c_2 &= (10-2\sqrt{10}+2\sqrt{5+2\sqrt{10}})/16\sqrt{2} \qquad & c_3 &= (10-2\sqrt{10}-2\sqrt{5+2\sqrt{10}})/16\sqrt{2} \\
c_4 &= (5+\sqrt{10}-3\sqrt{5+2\sqrt{10}})/16\sqrt{2} \qquad & c_5 &= (1+\sqrt{10}-\sqrt{5+2\sqrt{10}})/16\sqrt{2}
\end{aligned}
\tag{13.10.6}
$$

对更高的 p，直至 10，Daubechies[2]已经列表在数值上给出这些系数。p 每增加 1，系数的数目将增加 2 倍。

13.10.2　离散小波变换

迄今为止，我们还没有定义离散小波变换(DWT)，但 DWT 可由逐级地应用像(13.10.1)式的小波系数矩阵而得到。首先是长度为 N 的全数据向量，接着是长度为 $N/2$ 的“光滑”向量，再接着是长度为 $N/4$ 的“光滑-光滑”向量，这样一直进行下去，直到一个数目非常小的(通常是 2)的“光滑-…-光滑”分量被留下来。这个过程有时被称为**角锥形算法**(参见本节参考文献[4])，这样命名的原因是很明显的。DWT 的输出是由余下的分量和所有的“细节”分量构成，“细节”分量是在这个过程中逐步积累起来的。一个方框图可用来清楚地表述这个过程：

$$
\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \end{bmatrix}
\xrightarrow{13.10.1}
\begin{bmatrix} s_0 \\ d_0 \\ s_1 \\ d_1 \\ s_2 \\ d_2 \\ s_3 \\ d_3 \\ s_4 \\ d_4 \\ s_5 \\ d_5 \\ s_6 \\ d_6 \\ s_7 \\ d_7 \end{bmatrix}
\xrightarrow{\text{permute}}
\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ \underline{s_7} \\ d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix}
\xrightarrow{13.10.1}
\begin{bmatrix} S_0 \\ D_0 \\ S_1 \\ D_1 \\ S_2 \\ D_2 \\ S_3 \\ \underline{D_3} \\ d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix}
\xrightarrow{\text{permute}}
\begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ \underline{S_3} \\ D_0 \\ D_1 \\ D_2 \\ \underline{D_3} \\ d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix}
\xrightarrow{\text{etc.}}
\begin{bmatrix} S_0 \\ \underline{S_1} \\ D_0 \\ \underline{D_1} \\ D_0 \\ D_1 \\ D_2 \\ \underline{D_3} \\ d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix}
\tag{13.10.7}
$$

如果数据向量长度是 2 的更高幂次，那么将有更多步运用(13.10.1)式(或其他的小波系数)及重新排列。终止点永远是由两个 $\mathcal{S}$，分级的 $\mathcal{D}$，D 和 d 构成。注意，d 一旦产生，将简单地沿以后各步传递。

任何一步的 d_i 被称为原始数据向量的“小波系数”，最后的 $\mathcal{S}_0$，$\mathcal{S}_1$ 应该严格地称为“母函数系数”，尽管最后的 d 和 $\mathcal{S}$ 通常不严格地都称为“小波系数”。因为整个过程由正交的线性运算构成，所以整个 DWT 本身是一个正交的线性算子。

为了对 DWT 求逆，只须简单地颠倒整个过程，从最低阶开始，按照(13.10.7)式从右向左进行。逆矩阵(13.10.2)当然被用来代替矩阵(13.10.1)。

正如已经注意到的，矩阵(13.10.1)和(13.10.2)体现了数据向量的周期性边界条件(“环绕”)。人们通常视这为一个小的不便：在每一步的最后几个小波系数受到数据向量两端的数据的影响。通过循环移动矩阵

(13.10.1)的 $N/2$ 列到左边,就可以将环绕对称化,但是这并不能消除它。实际上有可能通过改变(13.10.1)式中最前 N 行和最后 N 行的系数而完全去掉环绕,即给出一个纯带状对角的正交矩阵。这种变化在有些时候,例如数据向量一端数据的幅度是另一端的很多倍时特别有用,但这已超出讨论范围

这里的程序 wt1,对数据向量 a[0..n-1]实施角锥形算法(或它的逆算法,如果 isign 为负)。连续地应用小波滤波器和伴随的排列,假定由用户提供的程序 wtstep 完成(在下面给出了几个不同的 wtstep 程序示例)。

```
#include "nr.h"

void NR::wt1(Vec_IO_DP &a, const int isign,
    void wtstep(Vec_IO_DP &, const int, const int))
一维离散小波变换。此程序实现角锥算法,用它的小波变换代替 a[0..n-1](如果 isign = 1),或者执行逆运算(如果 isign
= -1),注意 n 必须是 2 的整数次幂。在调用本程序的时候提供程序 wtstep,它是基本的小波滤波器。wtstep 的例子是
daub4 和 pwt(在程序 pwtset 之后)
{
    int nn;

    int n=a.size();
    if (n < 4) return;
    if (isign >= 0) {                                小波变换。
        for (nn=n;nn>=4;nn>>=1) wtstep(a,nn,isign);
        从最大级开始,工作到最小级。
    } else {
        for (nn=4;nn<=n;nn<<=1) wtstep(a,nn,isign);
        从最小级开始,工作到最大级。
    }
}
```

作为 wtstep 的一个特殊示例,下面是 DAUB4 小波程序:

```
#include "nr.h"

void NR::daub4(Vec_IO_DP &a, const int n, const int isign)
将 Daubechies 4 系数的小波滤波器运用到数据向量 a[0..n-1](如果 isign = 1)或者运用它的转置(如果 isign = -1),分级
地被程序 wt1 和 wtn 使用。
{
    const DP C0=0.4829629131445341,C1=0.8365163037378079,
        C2=0.2241438680420134,C3=-0.1294095225512604;
    int nh,i,j;

    if (n < 4) return;
    Vec_DP wksp(n);
    nh=n >> 1;
    if (isign >= 0) {                                应用滤波器。
        for (i=0,j=0;j<n-3;j+=2,i++) {
            wksp[i]=C0*a[j]+C1*a[j+1]+C2*a[j+2]+C3*a[j+3];
            wksp[i+nh]=C3*a[j]-C2*a[j+1]+C1*a[j+2]-C0*a[j+3];
        }
        wksp[i]=C0*a[n-2]+C1*a[n-1]+C2*a[0]+C3*a[1];
        wksp[i+nh]=C3*a[n-2]-C2*a[n-1]+C1*a[0]-C0*a[1];
    } else {                                         应用转置滤波器。
        wksp[0]=C2*a[nh-1]+C1*a[n-1]+C0*a[0]+C3*a[nh];
        wksp[1]=C3*a[nh-1]-C0*a[n-1]+C1*a[0]-C2*a[nh];
        for (i=0,j=2;i<nh-1;i++) {
            wksp[j++]=C2*a[i]+C1*a[i+nh]+C0*a[i+1]+C3*a[i+nh+1];
            wksp[j++]=C3*a[i]-C0*a[i+nh]+C1*a[i+1]-C2*a[i+nh+1];
        }
    }
    for (i=0;i<n;i++) a[i]=wksp[i];
}
```

对于大的小波系数集,对最后几行或几列的环绕在编程上带来很大不便。一个有效的措施是,在主循环之外作为一个特殊情况处理环绕。这里,将采用一个更一般的方式,涉及到在运行时的一些额外算术运算,而能得到一个满意的结果。下面的程序设置了任意特定的小波系数,它们的值恰好是已知的。

```
#include "nr.h"

wavefilt *wfilt_p;                    wavefilt类定义在nrutil.h中。

void NR::pwtset(const int n)
pwt 的初始化程序，这里利用系数为 4,12 和 20 的 Daubechies 小波滤波器，系数的选择由 n 的输入值决定，进一步的小波滤
波器可以用明显的方式包括时去。此程序必须在 pwt 第一次使用之前调用(对 n = 4 的情况，程序 daub4 比 pwt 快很多)。注
意，在 wavefilt 类中的 ioff 和 joff 的默认值每级以小波的支柱为中心分布。小波的峰可以通过选择 ioff = -2 和 joff = -n + 2
来近似。同时注意到，daub4 和 pwtset 在 n = 4 时利用了不同的默认值。
{
    const static DP c4_d[4]=
        {0.4829629131445341,0.8365163037378079,
        0.2241438680420134,-0.1294095225512604};
    const static DP c12_d[12]=
        {0.111540743350, 0.494623890398, 0.751133908021,
        0.315250351709,-0.226264693965,-0.129766867567,
        0.097501605587, 0.027522865530,-0.031582039318,
        0.000553842201, 0.004777257511,-0.001077301085};
    const static DP c20_d[20]=
        {0.026670057901, 0.188176800078, 0.527201188932,
        0.688459039454, 0.281172343661,-0.249846424327,
        -0.195946274377, 0.127369340336, 0.093057364604,
        -0.071394147166,-0.029457536822, 0.033212674059,
        0.003606553567,-0.010733175483, 0.001395351747,
        0.001992405295,-0.000685856695,-0.000116466855,
        0.000093588670,-0.000013264203};

    if (n == 4)
        wfilt_p=new wavefilt(c4_d,n);
    else if (n == 12)
        wfilt_p=new wavefilt(c12_d,n);
    else if (n == 20)
        wfilt_p=new wavefilt(c20_d,n);
    else nrerror("unimplemented value n in pwtset");
}
```

一旦 pwtset 被调用，下面的程序可作为 wtstep 的一个特例：

```
#include "nr.h"

extern wavefilt *wfilt_p;                    在pwtset中定义。

void NR::pwt(Vec_IO_DP &a, const int n, const int isign)
部分小波变换：将一个任意的小波滤波器应用到数据向量 a[0..n-1]（对 isign = 1 情形）或使用它的转置（对 isign = -1
情形）。此程序被 wt1 和 wtn 分级调用。实际的滤波器，在进行滤波之前，(需要)调用 pwtset 来确定，它初始化结构 wfilt。
{
    DP ai,ai1;
    int i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod;

    if (n < 4) return;
    wavefilt &wfilt=*wfilt_p;                使用引用来简化编码。
    Vec_DP wksp(n);
    nmod=wfilt.ncof*n;                       能够整除n的正常数。
    n1=n-1;                                  因为n是2的幂次，所以屏蔽所有位。
    nh=n >> 1;
    for (j=0;j<n;j++) wksp[j]=0.0;
    if (isign >= 0) {                        应用滤波器。
        for (ii=0,i=0;i<n;i+=2,ii++) {
            ni=i+1+nmod+wfilt.ioff;          指针增加并环绕。
            nj=i+1+nmod+wfilt.joff;
            for (k=0;k<wfilt.ncof;k++) {
                jf=n1 & (ni+k+1);            使用位与并且环绕指针。
                jr=n1 & (nj+k+1);
                wksp[ii] += wfilt.cc[k]*a[jf];
                wksp[ii+nh] += wfilt.cr[k]*a[jr];
            }
        }
    } else {                                 应用转置滤波器。
        for (ii=0,i=0;i<n;i+=2,ii++) {
            ai=a[ii];
```

```
                ai1=a[ii+nh];
                ni=i+1+nmod+wfilt.ioff;          看上面的说明。
                nj=i+1+nmod+wfilt.joff;
                for (k=0;k<wfilt.ncof;k++) {
                    jf=n1 & (ni+k+1);
                    jr=n1 & (nj+k+1);
                    wksp[jf] += wfilt.cc[k]*ai;
                    wksp[jr] += wfilt.cr[k]*ai1;
                }
            }
        }
        for (j=0;j<n;j++) a[j]=wksp[j];          从工作空间复制回结果。
    }
```

13.10.3 小波的形状

现在来看一些实际小波的情况。为此,通过上面的离散小波变换简单地运行于单位向量,并且再使 isign 为负以进行逆变换。图 13.10.1 显示了 DAUB4 小波,它是长度为 1024 的向量的第 5 个分量构成的单位向量的逆 DWT,以及 DAUB20 小波,它是第 24 个分量的逆 DWT(必须选择 DAUB20 后面级别的小波,以避免带有环绕的尾巴)。其他的单位向量给出相同形状的小波,但具有不同的位置和标度。

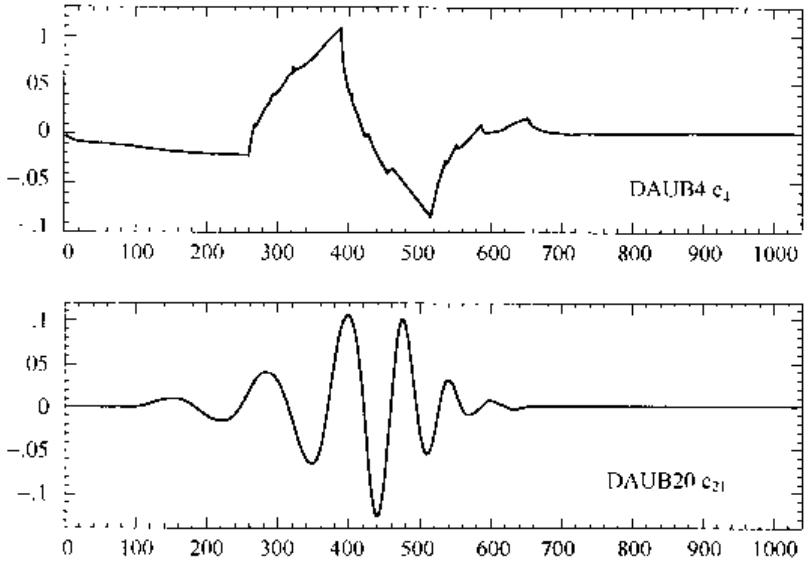


图 13.10.1 小波函数:小波族 DAUB4 和 DAUB20 中的单个基函数 一个完整的正交小波基包括这些函数的任一种平移和尺度变换。DAUB4 有无穷多个尖点;DAUB20 显示了有高阶导数相似的行为

可以看出 DAUB4 和 DAUB20 有连续的小波。DAUB20 还有更高阶的连续导数。DAUB4 有一个特别的性质,它的导数几乎存在于各处。不存在的地方则在 $p/2^n$ 点处,其中 p 和 n 是整数。在这些点处,DAUB4 左边可微,而右边不可微。这种间断性——至少在某些导数处——是带有紧支性质的小波的必然特征,如 Daubechies 数列。小波系数的数目每增加 2,Daubechies 小波获得一半的连续导数(但并不是精确的一半,精确的数值阶数是一个无理数)。

注意如下的事实,小波不光滑并不能阻止它们精确表示一些光滑函数,如同它们的近似阶数 p 要求的那样。小波的连续性和由一系列小波表示的函数的连续性并不一样。例如,DAUB4 可以表示(分段)任意斜率的线性函数;通过适当的线性组合,所有的尖点被消除。系数数目每增加 2,容许高 1 阶的多项式被精确表示。

图 13.10.2 显示了对输入向量 $\mathbf{e}_9+\mathbf{e}_{57}$(又是两个不同的特别小波)实施逆 DWT 所得的结果。因为 9 在等级范围 8~15 中处于较前的位置,所以对应的小波出现在图形的左边;因为 57 处在等级范围较后的位置(较小尺度),所以它是一个较窄的小波,而且处在范围 32~63 内,它接近末端,因此波形在图形的右边。注意较小尺寸的小波幅度较高,因此这两个小波有相同的平方积分。

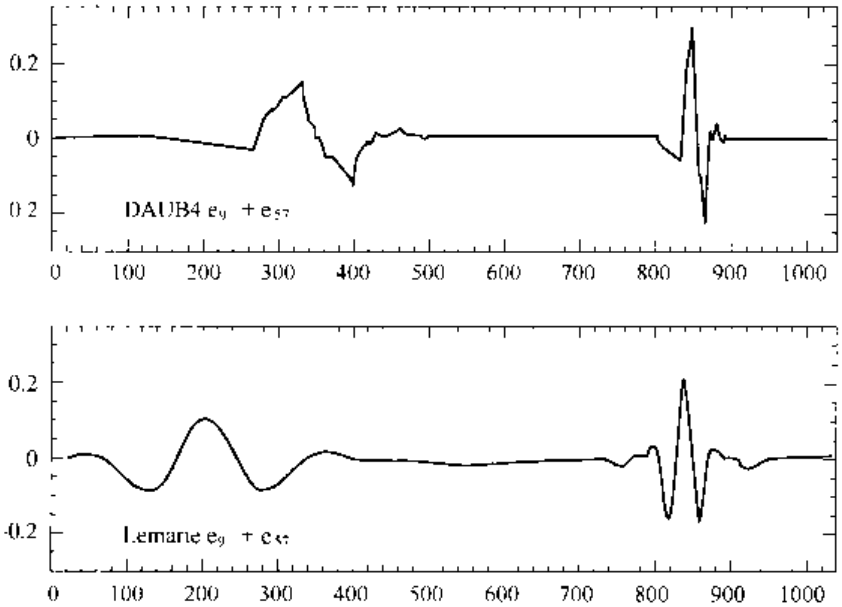


图 13.10.2　更多的小波示例。图中小波是由两个单位向量 $\mathbf{e}_9+\mathbf{e}_{57}$ 之和产生的，它们有不同的标度等级，也在不同的空间位置，DAUB4小波(a)是由在坐标空间的滤波器((13.10.5)式)定义的，而Lemarie小波(b)是由在傅里叶空间中很容易写出的滤波器((13.10.14)式)定义的

13.10.4　傅里叶域中的小波滤波器

傅里叶变换的一组滤波系数 c_j 由下式确定：

$$H(\omega)=\sum_j c_j e^{ij\omega} \tag{13.10.8}$$

其中 H 是周期为 2π 的周期函数，与前面叙述意义相同：它是小波滤波器，现在在傅里叶域中重新写出。一个非常有用的事实是，c 的正交条件(参见上面的(13.10.3)式)在傅里叶域中变成了两个简单的关系：

$$\frac{1}{2}|H(0)|^2=1 \tag{13.10.9}$$

和

$$\frac{1}{2}\left[|H(\omega)|^2+|H(\omega+\pi)|^2\right]=1 \tag{13.10.10}$$

类似地，p 阶的近似条件(参见前面的(13.10.3)式)有一个简单的公式，要求 $H(\omega)$ 在 $\omega=\pi$ 处有一个第 p 阶零点，或写为

$$H^{(m)}(\pi)=0 \qquad m=0,1,\ldots,p-1 \tag{13.10.11}$$

因此在傅里叶域中构造小波系列就比较直观。只要简单地构造一个满足(13.10.9)式～(13.10.11)式的函数 $H(\omega)$。为了得到实际的 c_j，它可应用于长度为 N 的数据向量(或 s 分量)，并且又具有如同矩阵(13.10.1)式和(13.10.2)式的周期环绕，因此就可以利用离散傅里叶变换得到(13.10.1)式和(13.10.2)式的周期环绕，利用离散傅里叶变换得到(13.10.8)式的逆：

$$c_j=\frac{1}{N}\sum_{k=0}^{N-1}H\left(\frac{2\pi k}{N}\right)e^{-2\pi ijk/N} \tag{13.10.12}$$

求积镜像滤波器 G(将 c_j 倒序并符号交替变换)恰好有傅里叶表达式

$$G(\omega)=e^{-i\omega}H^*(\omega+\pi) \tag{13.10.13}$$

其中的星号代表复共轭。

一般来说，上面的过程将不会产生具有紧支性质的小波滤波器。换言之，N 个 $c_j(j=0,\ldots,N-1)$ 一般是非零值(尽管它们在幅度上会很快减小)。Daubechies 小波，或其他具有紧支性质的小波都是特别选定的，

以使得$H(\omega)$是一个仅有少数几个傅里叶分量的三角多项式,以保证只有少数几个非零的 c_j。

另一方面,有时并没有特别的原因要求紧支性质。放弃这个要求将容许构造更光滑的小波(更高的 p 值)。即使没有紧支性,在矩阵(13.10.1)中隐含的卷积也可以用 FFT 方法有效地计算。

Lemarie 小波(参见本节参考文献[4])有 $p=4$,但没有紧支性质,并且通过选择 $H(\omega)$来定义,

$$H(\omega)=\left[2(1-u)^4\frac{315-420u+126u^2-4u^3}{315-420v+126v^2-4v^3}\right]^{1/2} \tag{13.10.14}$$

其中

$$u\equiv\sin^2\frac{\omega}{2}\qquad v\equiv\sin^2\omega \tag{13.10.15}$$

解释(13.10.14)式超出了讨论的范围。一个不正规的描述是:由(13.10.14)式导出的求积镜像滤波器 $G(\omega)$具有性质:当它运用到某种函数,这种函数的奇数样本值等于它的偶数样本值的三次样条插值时,它给出恒等于0的结果。因为这类函数包括很多光滑的函数,因此 $H(\omega)$选择一个函数的光滑信息内容时做得非常好。Lemarie 小波示例在图 13.10.2 中给出。

13.10.5 截断小波近似

小波有用性的一个重要方面在于这样一个事实:小波变换可以被严重截去而依然有用,也就是说对小波可以进行稀疏展开。傅里叶变换的情形就不同:FFT 通常是没有被截断地运用,例如,进行快速卷积就是如此。之所以能进行这种运算是因为卷积算法在傅里叶基中特别简单。但是在小波基中,没有任何标准的算术运算是特别简单的。

为了明白截断是怎样起作用的,考虑图 13.10.3 中的简单示例。上图中列出一个任意选取的检验函数,除了一个平方根的尖点外是光滑的,它被取样成长度为 2^{10}的向量。下图中(实线)用对数的刻度显示了向量经过 DAUB4 离散小波变换后的分量的绝对值。注意从右到左是不同的谱级别,512 ~ 1023,256 ~ 511,128 ~ 255等。在每一级,小波系数仅在紧靠近尖峰的地方,或者紧靠谱范围的左边界和右边界(边缘效应)才不可忽略。

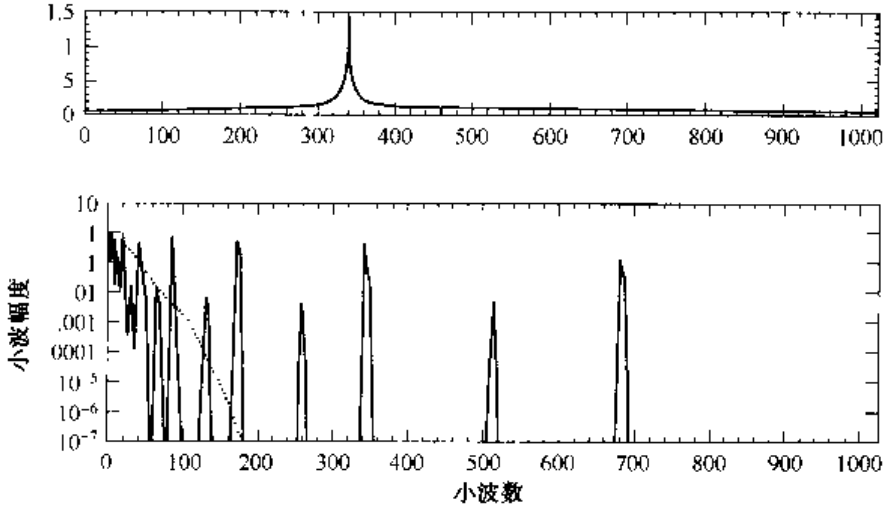


图 13.10.3 (a)带有尖顶的任意检验函数,取样成为长度为 1024 的向量;(b)对(a)进行离散小波变换后产生的 1024 个小波系数的绝对值。注意图中的对数坐标。当按降序排列时,图中点线具有同样的幅度,可以看出1024个系数中仅130个系数的值大于10^{-4}(或大于10^{-5}乘以最大系数值,这个系数值约为10)

在图 13.10.3 的下图中的点线和实线具有相同的幅度,但点线是按系数值的降序排列。例如,从图上马上可以看出第 130 个最大的小波系数的幅度值比小波的最大幅度的 10^{-5}倍还要小,小波系数最大幅度直接近 ~ 10(功率或平方积之比小于 10^{-10})。因此,作为示例的函数可以仅被 130 个系数精确表示,而不是 1024 个系数,余下系数可令其为 0。可以看出这种截断使向量变得稀疏,但效果不差于 1024 个系数的情形。还有

一点很重要，那就是在小波空间中向量的截断应根据分量的**幅度**，而不是各分量在向量的中的位置。但是保留向量的前 256 个分量(这向量占有除了最后两个级的所有谱系)将对函数给出一个非常差的、边缘参差不齐的近似。因此当用小波压缩一个函数时，必须同时记录非零系数的值和位置。

一般来说，紧支的(因此不光滑)小波更适合于较低精度的近似和带有不连续(像边缘)的函数，而光滑(因此不紧支)的小波则适合获得较高数值精确度。这种性质使得紧支小波成为图像压缩的较好选择，光滑小波成为积分方程快速计算的最佳选择。

13.10.6　多维小波变换

一个 d 维数组的小波变换很容易根据其下标顺序地变换而得到，首先是第一个下标(对其他下标的所有值)，接着是第二个下标，等等。每个变换对应于乘以一个正交矩阵。由于矩阵的结合性，所以其结果和下标被变换的顺序无关。此情形和多维 FFT 非常类似。因此有效进行多维 DWT 的程序可以用多维 FFT 程序如 fourn 为模型而建立起来。

```
#include "nr.h"

void NR::wtn(Vec_IO_DP &a, Vec_I_INT &nn, const int isign,
    void wtstep(Vec_IO_DP &, const int, const int))
如果 isign 输入值为 1，那么将用 a 的 ndim 维离散小波变换替换 a。nn[0..ndim-1]是一个整数数组，其值为每维的长度
(实的值数目)，并且还必须是 2 的整数次幂。a 是一个实数数组，其长度为上面各个长度的积。在 a 中数据以多维实型数
组的形式储存。如果 isign 输入值为 -1，那么 a 将被它的逆小波变换替换。程序 wtstep 是基本的小波滤波器，调用此程序
之前它必须提供。wtstep 的例子有 daub4 和 pwt(在程序 pwtset 之后)。
{
    int idim,i1,i2,i3,k,n,nnew,nprev=1,nt,ntot=1;

    int ndim=nn.size();
    for (idim=0;idim<ndim;idim++) ntot *= nn[idim];
    for (idim=0;idim<ndim;idim++) {            对维数的主循环。
        n=nn[idim];
        Vec_DP wksp(n);
        nnew=n*nprev;
        if (n > 4) {
            for (i2=0;i2<ntot;i2+=nnew) {
                for (i1=0;i1<nprev;i1++) {
                    for (i3=i1+i2,k=0;k<n;k++,i3+=nprev) wksp[k]=a[i3];
                    复制有关行或列到工作空间。
                    if (isign >= 0) {          进行一维小波变换。
                        for(nt=n;nt>=4;nt >>= 1)
                            wtstep(wksp,nt,isign);
                    } else {                   或逆变换。
                        for(nt=4;nt<=n;nt <<= 1)
                            wtstep(wksp,nt,isign);
                    }
                    for (i3=i1+i2,k=0;k<n;k++,i3+=nprev) a[i3]=wksp[k];
                    从工作空间复制回结果。
                }
            }
        }
        nprev=nnew;
    }
}
```

和以前一样，wtstep 是单独的小波滤波器，可以是 daub4 或 pwt。

13.10.7　图像压缩

多维变换 wtn 的一个直接应用是图像压缩。整个过程就是数字化的图像进行小波变换，接着用一种非均匀的、优化的方法在小波系数中"分配二进制位"。一般来说，大的小波系数得到精确的量化，而小的系数量化很粗糙，仅有一两位二进位，或者完全截断了。如果结果的量化等级从统计上看还不均匀，它们还可以用一些技巧例如 Huffman 编码(参见 20.4 节)进行压缩。

尽管这个过程"后部分"的更细微描述，也就是图像的量化和编码，超出了讨论的范围，但是对进行简单截断的"前部分"小波编码的说明还是很直观的：保留大于某个阈值的所有小波系数(保持充分的精确性)，删

除(置为0)所有较小的小波系数。可以通过调整阈值而改变被保留系数所占的百分比。

图 13.10.4 显示了保留不同数目的小波系数的一系列图像。原始图像(a),一幅 IEEE 官方的测试图像,有 256×256 个像素点以及 8 位灰度级。紧接着的两个重建图像是由 65 536 个小波系数的 23%(b)和 5.5%(c)构造的。后一幅图像显示了截去小波系数后的一种折衷。高对比度边缘(人物右脸颊和头发的亮光等)依然保持相对较高的分辨率,而低对比度区域(人物左眼和左颊等)则退化为相当于原始像素点的分辨率。图 13.10.4(d)是实施傅里叶变换而不是小波变换的等价结果:图像是由 65 536 个实傅里叶分量中具有最大幅度 5.5% 个分量后重新构造的。可以看出,由于正弦和余弦不是局部的,整个图像的分辨率普遍比较差。任何分量的删除处产生模糊的"边缘"(因此,实际的傅里叶图像压缩方式是将图像拆成小的像素块,例如 16×16,图像重建时,再对每块的边缘进行相当精细的光滑处理)。

图 13.10.4 (a)IEEE 的测试图像,256×256 个像素点,8 位灰度级;(b)将图像变换成小波基,其小波分量的77%被置为0(那些振幅最小的),接着从剩下的23%重建图像;(c)和(b)一样,但是94.5%的小波分量被删除;(d)和(c)一样,但是用傅里叶变换代替了小波变换。小波系数在保留相关细节方面优于傅里叶系数

13.10.8 线性系统的快速求解

小波的一个最有趣的潜在应用是线性代数的应用。其基本思想[1]是,将整数运算算子(也就是一个大矩阵)看成一幅数字图像。假设算子在二维小波变换下能很好地压缩,即它的小波系数大部分非常小以致可忽略,那么与算子有关的任何线性系数在小波基下成为一个稀疏系统,为了求解

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{13.10.16}$$

首先对算符 $\mathbf{A}$ 和右边的 $\mathbf{b}$ 进行小波变换:

$$\tilde{\mathbf{A}}\equiv\mathbf{W}\cdot\mathbf{A}\cdot\mathbf{W}^T,\qquad \tilde{\mathbf{b}}\equiv\mathbf{W}\cdot\mathbf{b}$$

其中 $\mathbf{W}$ 代表了一维小波变换,接着求解

$$\tilde{\mathbf{A}}\cdot\tilde{\mathbf{x}}=\tilde{\mathbf{b}} \tag{13.10.18}$$

最后通过逆小波变换得到结果

$$\mathbf{x} = \mathbf{W}^T \cdot \tilde{\mathbf{x}} \tag{13.10.19}$$

(注意,程序 wtn 完成将 $\mathbf{A}$ 到 $\tilde{\mathbf{A}}$ 的完全变换。)

一个典型的能很好进行小波压缩的整数算子是,在靠近它的主对角线有任意的(甚至是奇异的)元素,但在远离对角线处将变光滑,示例为

$$A_{ij} = \begin{cases} -1 & i = j \\ |i-j|^{-1/2} & \text{其他情况} \end{cases} \tag{13.10.20}$$

图 13.10.5 显示了这个矩阵小波变换的图形形式,其中 i 和 j 的变化范围是 0...255,采用 DAUB12 小波,将幅度大于最大元素幅度 10^{-1} 倍的元素用黑的像素点表示,在 10^{-1} 和 10^{-6} 之间的元素用灰的像素点表示,白的像素点表示元素幅度 $< 10^{-6}$。下标 i 和 j 都从最下端左边开始计算。

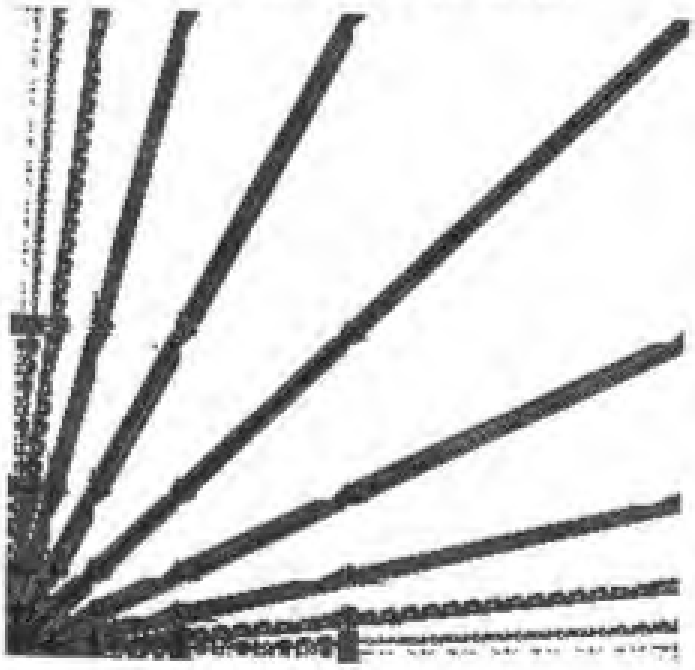


图 13.10.5　用图表示一个 256×256 矩阵的小波变换。原始矩阵在它的对角线上有不连续的尖顶,而到对角线两边逐渐光滑地衰减。在小波基下,矩阵变得稀疏:大于 10^{-1} 的分量以黑色表示,大于 10^{-6} 的分量以灰色表示,小于 10^{-6} 的分量以白色表示。矩阵下标 i 和 j 从最下端左边开始计算

从图中可以看出,谱系分解成大小为 2 的幂次的块。在每个块的边缘或角上,可以看到由于环绕的小波边界条件引起的边缘效应。除了边缘效应外,在每一块之内,不可忽略的元素是集中在块的对角线上。这说明,对于这种线性算符,一个小波主要将和它自己谱系的近邻(沿着主对角线的方块)和其他谱系的近邻(对角线之外的矩形块)联系在一起。

如图 13.10.5 所示,不可忽略的矩阵元素的个数,数量级仅为 N,即矩阵的线性尺寸;粗略地估计,其数目为 $10N \log_{10}(1/\epsilon)$,其中 ϵ 为截断水平,例如 10^{-6}。对于一个 2000×2000 的矩阵,矩阵的稀疏因子将接近 30。

有各种数值方法可以用来求解这种"谱系对角带状"形式的稀疏线性系统。Beylkin, Coifman 和 Rokhlin[1] 通过观察得到:(1)两个这种矩阵的乘积还是谱系对角带状矩阵(当然还是截断了新产生的元素中小于预先确定的阈值 ϵ);(2)乘积可用阶为 N 次的运算顺序完成。

由 Schultz 或 Hotelling 方法,快速的矩阵相乘法使得人们能对矩阵求逆,参见 2.5 节。

对于谱系对角带状矩阵的快速计算还有其他的方法。例如,可以采用 2.7 节中 linbcg 共轭梯度方法。

参考文献和深入阅读

Daubechies, I. 1992, *Wavelets* (Philadelphia: S.I.A.M.).

Strang, G. 1989, *SIAM Review*, vol. 31, pp. 614–627.

Beylkin, G., Coifman, R., and Rokhlin, V. 1991, *Communications on Pure and Applied Mathematics*, vol. 44, pp. 141–183. [1]

Daubechies, I. 1988, *Communications on Pure and Applied Mathematics*, vol. 41, pp. 909–996. [2]

Vaidyanathan, P.P. 1990, *Proceedings of the IEEE*, vol. 78, pp. 56–93. [3]

Mallat, S.G. 1989, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674–693. [4]

Freedman, M.H., and Press, W.H. 1992, preprint. [5]

13.11 深入讨论:取样定理的数值应用

在6.10节通过Rybicki的研究,我们对Dawson积分得到了一个近似公式。现在我们有了傅里叶变换的基础。可以看出公式来源于取样定理(参见12.1节)的数值应用,这时取样定理一般被看做是一个纯粹的分析工具。我们的讨论和Rybicki[1]一致。

对于目前的目的,取样定理表述为如下形式最为方便:考虑一个任意函数$g(t)$和取样点网格$t_n=\alpha+nh$,其中n按整数变化,α是一个常数,它允许取样网格的任意移动。那么有

$$g(t)=\sum_{n=-\infty}^{\infty} g(t_n)\operatorname{sinc}\frac{\pi}{h}(t-t_n)+e(t) \tag{13.11.1}$$

其中$\operatorname{sinc} x\equiv\sin x/x$。对取样点的求和被称为$g(t)$的**取样表示**,$e(t)$是误差项。取样定理表明,如果$g(t)$的傅里叶变换

$$G(\omega)=\int_{-\infty}^{\infty} g(t)e^{i\omega t}\,dt \tag{13.11.2}$$

在$|\omega|\geq\pi/h$时趋于0,则取样表示是精确的,也就是说$e(t)\equiv 0$。

取样表示在什么时候对函数的近似数值计算最为有利呢?为使得误差项尽可能小,傅里叶变换$G(\omega)$在$|\omega|\geq\pi/h$时必须足够小。另一方面,为了使(13.11.1)式中的求和能用合理的、数目较少的项近似,函数$g(t)$在t的某一范围之外必须非常小。因此为了使(13.11.1)式数值上有用,我们得到两个条件:函数$g(t)$和它的傅里叶变换$G(\omega)$在它们各自的自变量比较大时应很快趋于零。

遗憾的是,这两个条件是互相对立的,如同量子力学里的不确定性原理。两个自变量以多快的速度趋于零存在严格的限制。根据Hardy定理[2],如果当$|t|\to\infty$,$g(t)=O(e^{-t^2})$和$|\omega|\to\infty$,$G(\omega)=O(e^{-\omega^2/4})$,那么$g(t)\equiv Ce^{-t^2}$,其中$C$是一个常数。这可以解释为什么所有的函数中,高斯函数随t和ω的衰减都是最快的。从这个意义上讲,高斯函数在数值上是所有函数中能最好地进行取样表示的。

将高斯函数写为$g(t)=e^{-t^2}$,

$$e^{-t^2}=\sum_{n=-\infty}^{\infty} e^{-t_n^2}\operatorname{sinc}\frac{\pi}{h}(t-t_n)+e(t) \tag{13.11.3}$$

误差$e(t)$除了依赖于t外,还依赖于h和α。在目前的情况下,对$e(t)$给定如下的一个范围就足够了:

$$|e(t)|<e^{-(\pi/2h)^2} \tag{13.11.4}$$

它可以简单地理解为高斯函数的分界点$|\omega|>\pi/h$处的傅里叶变换的幅度的量级。

当和式(13.11.3)是以有限项近似,例如从N_0-N到N_0+N,其中N_0是最接近$-\alpha/h$的整数,将还有进一步的截断误差。但是,如果N被选择得使$N>\pi/(2h^2)$,那么截断误差将小于(13.11.4)式给定的范围,正因为这个范围是一个保守估计,我们将继续用它估计(13.11.3)式中的$e(t)$。即使在N取中等的数值时,被截断的求和式也会对高斯函数给出相当精确的表示,例如当$h=1/2$,$N=7$时,$|e(t)|<5\times10^{-5}$;当$h=1/3$,$N=15$时,$|e(t)|<2\times10^{-10}$;当$h=1/4$,$N=25$时,$|e(t)|<7\times10^{-18}$。

人们可能要问,高斯函数的这样一个数值表示,可以像指数函数一样容易和快速地计算,其用途是什么呢?答案在于很多超越函数可表示成与高斯函数有关的积分,通过(13.11.3)式的代换就可以用基本函数的和很好地近似求积分。

让我们首先考虑复变量为 $z = x + iy$ 的函数 $w(z)$ 的示例，它和复误差函数存在如下关系：

$$w(z) = e^{-z^2}\operatorname{erfc}(-iz) \tag{13.11.5}$$

它的积分表示是

$$w(z) = \frac{1}{\pi i}\int_C \frac{e^{-t^2}\,dt}{t-z} \tag{13.11.6}$$

其中的积分曲线 C 从 $-\infty$ 延伸到 ∞，从 z 轴通过（参见本节参考文献[3]），有许多方法可以用来估计这个函数（参见本节参考文献[4]）。将取样表示式(13.11.3)代入(13.11.6)式，并且进行基本的曲线积分，得到

$$w(z) \approx \frac{1}{\pi i}\sum_{n=-\infty}^{\infty} h e^{-t_n^2}\frac{1-(-1)^n e^{-\pi i(\alpha-z)/h}}{t_n - z} \tag{13.11.7}$$

其中忽略了误差项。注意对某些 $n = m$，在 $z \to t_m$ 时没有奇点，但是在这种情况下，对第 m 项必须进行特殊处理（例如，通过幂级数展开）。

(13.11.7)式的另一种表现形式，可以通过用三角函数表达式(13.11.7)中的复指数，并且利用 z 代替取样表达式(13.11.3)中的 t 而得到

$$w(z) \approx e^{-z^2} + \frac{1}{\pi i}\sum_{n=-\infty}^{\infty} h e^{-t_n^2}\frac{1-(-1)^n\cos\pi(\alpha-z)/h}{t_n - z} \tag{13.11.8}$$

此形式在 $|y| \ll 1$ 时，用来求 $\operatorname{Re} w(z)$ 特别有用。注意在估算(13.11.7)式时，求和号中的指数项是一个常数，只需估算一次，对(13.11.8)式中的余弦项也类似

通过选择特定的 α 值，可以从(13.11.7)式或(13.11.8)式导出一组不同的公式。有意义的 8 个可选值是：$\alpha = 0$、x、iy 或 z，再加上每个值增加 $h/2$ 后所得的值。因为(13.11.3)式中误差范围表达式已经假设 α 为实数，选择 α 为复数仅当 z 的虚部不是很大时才有用。这里不可能列出 16 个可能等式，仅给出能显示一些重要特征的两个特例。

首先在(13.11.8)式中设 $\alpha = 0$，则有

$$w(z) \approx e^{-z^2} + \frac{1}{\pi i}\sum_{n=-\infty}^{\infty} h e^{-(nh)^2}\frac{1-(-1)^n\cos(\pi z/h)}{nh - z} \tag{13.11.9}$$

在整个 z 平面此近似式应用得很好，如同前面所述，当某一些分母比较小时，必须采用幂级数展开。$\alpha = 0$ 的公式在本节参考文献[5]中有简单的讨论。它们和 Chiarella 和 Reichel[6]用 Goodwin[7]方法导出的公式相似，但并不等价。

下一步，在(13.11.7)式中，令 $\alpha = z$，有

$$w(z) \approx e^{-z^2} - \frac{2}{\pi i}\sum_{n\ \mathrm{odd}} \frac{e^{-(z-nh)^2}}{n} \tag{13.11.10}$$

其中的求和是对所有的奇整数（正的和负的）求和。注意在求和式中已经进行替换 $n \to -n$。此公式比(13.11.9)式简单，只包含一半的项数，但是当 y 较大时，误差将增大。(13.11.10)式就是在 6.10 节中用的 Dawson积分近似公式(6.10.3)的来源。

参考文献和深入阅读

Rybicki, G.B. 1989, *Computers in Physics*, vol. 3, no. 2, pp. 85–87. [1]

Hardy, G.H. 1933, *Journal of the London Mathematical Society*, vol. 8, pp. 227–231. [2]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [3]

Gautschi, W. 1970, *SIAM Journal on Numerical Analysis*, vol. 7, pp. 187–198. [4]

Armstrong, B.H., and Nicholls, R.W. 1972, *Emission, Absorption and Transfer of Radiation in Heated Atmospheres* (New York: Pergamon). [5]

Chiarella, C., and Reichel, A. 1968, *Mathematics of Computation*, vol. 22, pp. 137–143. [6]

Goodwin, E.T. 1949, *Proceedings of the Cambridge Philosophical Society*, vol. 45, pp. 241–245. [7]

第 14 章　数据的统计描述

14.0　引言

在本章和下一章，将主要讨论**数据**的概念。

数据自然是由数组成，但这些数是输入到计算机中去的，而不是由计算机自身产生的。这些数必须认真地处理，既不能篡改，也不应对它们进行一些其特征不完全为我们所了解的数值处理。对数据应有尊重的态度，而不能采取在其他数值中所允许的、甚至受到赞扬的"华而不实"的态度。

数据的分析不可避免地要涉及到某些统计学领域，统计学可以处理一些问题，这些问题既不能确切地说是数学分支的问题，或许也不能说是科学分支的问题。在以下各节，将经常遇到如下范例：

- 为了计算"统计量"，将对数据应用一些公式进行处理。
- 计算这一统计量落在概率分布的那一部分，这些概率分布是基于"零假说"来计算的。
- 如果它落在一个非常不可能的点上，即落在概率分布的末端以外，就可以说"零假说"对这个数据组来说是**错误的**。

如果一个统计量落在分布的**合理**部分，也不能说"零假说"被"验证"或"证明"。这是统计学的独特之处，它从来不能证明任何事物成立，而只能反驳事物。证明一个假说的最好方式是通过用统计方法排除一系列相互竞争的假说，而这些假说是已被人们一一提出的。一段时间后，对于和竞争者放弃寻求新的假说，那么从此以后，你所做的假说将被接受。

在本书中，我们对数据分析过程稍进行了主观的区分，一是**独立于模型的**，另一是**依赖于模型的**。前者包含了所谓的**描述性统计学**，即用一些常规的量来表征一组数据组，如均值、方差等。它也包括统计检验，即寻求建立两个或更多数据组之间"相同之处"或"不同之处"，或者寻求建立或测度两组数据之间的**相关程度**，这些内容将在本章中讨论。

在另一类依赖于模型的统计学中，将整个拟合数据的主题纳入这种理论，如参数估值、最小二乘方拟合等，这些内容将在第 15 章介绍。

在 14.1 节中将处理**集中趋势的测度**，如分布的矩、中值和众数。在 14.2 节将学习检验不同的数据组，是否是由这些集中趋势测度的不同值的分布而导出的。这又自然地引出 14.3 节中更一般的问题，即两个分布是否能被有效地表明是(显著)不同的分布。

从 14.4 节至 14.7 节将讨论关于两种分布之间的**关联的测度**。我们希望能决定两个变量是否是"相关"或互相"依赖"，如果是，希望能以某种简单的公式表征它们相关的程度。同时还强调了参数和非参数(秩)方法之间的差别。

在 14.8 节中引入数据平滑的概念，并且讨论了一个特例：Savitzky-Golay 平滑滤波器。

在数学方法上，本章使用了第 6 章的特殊函数，特别是 6.1 节至 6.4 节的一些内容。若想了解这些内容，复习这些章节。

参考文献和深入阅读

Bevington, P.R., and Robinson, D.K. 1992, *Data Reduction and Error Analysis for the Physical Sciences*, 2nd ed. (New York: McGraw-Hill).

Stuart, A., and Ord, J.K. 1994, *Kendall's Advanced Theory of Statistics*, 6th ed. (London: Edward Arnold) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*].

Norusis, M.J. 1999, *SPSS 9.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall).

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

14.1　分布的矩：均值、方差、偏斜度等

当一组数值有很强的集中趋势，即在某一特殊值周围聚集的趋势，则用与矩有关的一些量来表征这组数是非常有用的，**矩**为这些量的整数幂之和。

最常见的是数值 $x_0, \ldots, x_{N-1}$ 的**均值**：

$$\overline{x} = \frac{1}{N}\sum_{j=0}^{N-1} x_j \tag{14.1.1}$$

$\overline{x}$ 是对所有在某一中心聚集值的估算，横杠（¯）表示平均，有时也用尖括号表示，如 $\langle x \rangle$。应当注意的是，均值并不是这种中心聚集值的惟一可用的估算量，也不一定是最佳的估算量。对于从拖有很宽"尾部"的概率分布中抽取的数值来说，其平均值的收敛性也许较差，或者当样本点数目增加时完全没有收敛性。这时，作为替代的估算量是**中位数**和**众数**，这两个量将在本节后面介绍。

在表征了分布的中心值后，下面要讨论的常用量是分布围绕中心值的"宽度"和"变化量"。同样，对此的测度不止一种量，最常用的是**方差**，

$$\mathrm{Var}(x_0 \ldots x_{N-1}) = \frac{1}{N-1}\sum_{j=0}^{N-1}(x_j - \overline{x})^2 \tag{14.1.2}$$

或它的平方根，即**标准差**，

$$\sigma(x_0 \ldots x_{N-1}) = \sqrt{\mathrm{Var}(x_0 \ldots x_{N-1})} \tag{14.1.3}$$

(14.1.2)式估算了 x 对其均值的均方差。关于(14.1.2)式中的分母为什么取 $N-1$ 而不取 N，其间有一段较长的故事，若读者不知此故事，可以参阅任何一本较好的统计学教材。如果要测度一个分布的方差，其均值 $\overline{x}$ 是先验已知的，而不是从数据估算中得到的，则 $N-1$ 应该能换成 N。

由于均值依赖于数据的一阶矩，因此方差和标准差依赖于二阶矩。在实际生活中有时可能要处理二阶矩并不存在（即为无穷大）的分布。在这种情况下，方差和标准差作为对围绕中心值之数据宽度的度量是无用的：随着样本点数的增加，从(14.1.2)式或(14.1.3)式得到的值将不收敛，或从相同分布中抽取的数据组之间没有任何一致性。这种情况甚至出现在峰值宽度用肉眼看起来相当有限的场合。一个更稳健的宽度估算量是**平均偏差**或**平均绝对偏差**，其定义为

$$\mathrm{ADev}(x_0 \ldots x_{N-1}) = \frac{1}{N}\sum_{j=0}^{N-1}|x_j - \overline{x}| \tag{14.1.4}$$

经常用样本中值 x_{med} 代替(14.1.4)式中的 $\overline{x}$。对任何固定的样本，中值实际上是使平均绝对偏差最小。

历史上统计学家并不推荐使用(14.1.4)式代替(14.1.2)式，因为(14.1.4)式中的绝对值是"非解析的"，并且理论证明很困难。然而最近几年，传统已有改变，**稳健估算量**的问题已相当流行和重要（即估算较宽的分布，其具有相当显著的"分离"点数目）。较高的矩或涉及输入数据较高幂次的统计量，在稳健性上总是低于较低矩或仅涉及线性和（所有矩中最低矩）或计算的统计量。

事实正是如此，**偏斜度**即**三阶矩**，和**峭度**或**四阶矩**应当谨慎使用，甚至最好不用。

偏斜度是对一分布围绕均值不对称程度的表征，尽管均值、标准差和平均偏差都是有量纲的量（即其与所测量的 x_j 有着相同的单位），而依此方法对偏斜度的传统定义却使得其无量纲，其仅是一个纯粹的数来表征分布的形状，一般定义为

$$\mathrm{Skew}(x_0 \ldots x_{N-1}) = \frac{1}{N}\sum_{j=0}^{N-1}\left[\frac{x_j - \overline{x}}{\sigma}\right]^3 \tag{14.1.5}$$

其中 $\sigma = \sigma(x_0 \ldots x_{N-1})$ 是分布的标准差（参见(14.1.3)式），偏斜度为正值表示一个具有扩充到更多正 x 的非对称尾部的分布，而负偏斜度表示尾部扩充到更多负 x 的分布（参见图 14.1.1）。

当然，对(14.1.5)式，任何一个包含 N 个测量值的组合都可能给出非零的偏斜度值，即使其依从的分布是对称的（有零偏斜度）。为使(14.1.5)式有意义，必须对作为基本分布偏斜度的估算量之标准差有所了解。遗憾的是，它仅取决于基本分布的形状，并且相当关键地取决于其尾端。对于理想的正态（高斯）分布，

(14.1.5)式的标准差近似为$\sqrt{15/N}$,当$\overline{x}$是(14.1.1)给出的样本均值时是$\sqrt{6/N}$。在实际中,仅当偏斜度是这一个数的很多倍时,偏斜度才有很好的实用性。

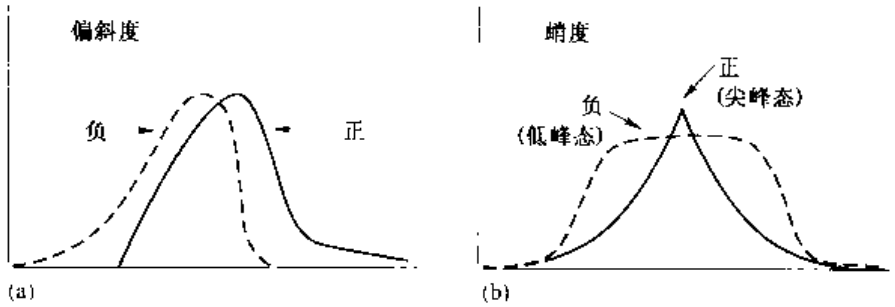


图 14.1.1 三阶和四阶矩显著不同于正态(高斯)分布的分布。(a)偏斜度或三阶矩;(b)峭度和四阶矩

峭度也是一个无量纲的量,用于测度分布的相对峰锐度或平坦度。这是相对于正态分布而言。一个带有正峭度的分布称为**尖峰态**,麦特喇叭(Matterhorn)的轮廓线就是一个例子。一个带有负峭度的分布称为**低峰态**,一个面包的轮廓即是它的一个例子(见图 14.1.1),介于两者之间的称为**常峰态分布**。

峭度的常见定义为

$$\mathrm{Kurt}(x_0 \ldots x_{N-1}) = \left\{ \frac{1}{N} \sum_{j=0}^{N-1} \left[\frac{x_j - \overline{x}}{\sigma} \right]^4 \right\} - 3 \tag{14.1.6}$$

其中 -3 是为了使正态分布时(14.1.6)式为零值。

当 σ 是真实标准差时,(14.1.6)式作为基本正态分布峭度的估算量,标准差为$\sqrt{96/N}$。当 σ 是如(14.1.3)式给出的样本估计时为$\sqrt{24/N}$。然而,若峭度取决于一个有许多实际分布的高阶矩,那对这些分布来说,作为估计量的(14.1.6)式的标准差几乎为无穷大。

对本章所定义的量的计算是相当直接的,许多教科书使用二项式定理,将定义式展开成数据各种幂次的和,例如我们所熟悉的

$$\mathrm{Var}(x_0 \ldots x_{N-1}) = \frac{1}{N-1} \left[\left(\sum_{j=0}^{N-1} x_j^2 \right) - N\overline{x}^2 \right] \approx \overline{x^2} - \overline{x}^2 \tag{14.1.7}$$

但这将增大较大因子造成的舍入误差,并且根据计算速度来看这也是不合理的。为了使舍入误差最小(特别是对大的样本值),较好的方式是使用**校正的二步算法**[1]:首先计算$\overline{x}$,然后由下式计算 $\mathrm{Var}(x_0 \ldots x_{N-1})$:

$$\mathrm{Var}(x_0 \ldots x_{N-1}) = \frac{1}{N-1} \left\{ \sum_{j=0}^{N-1} (x_j - \overline{x})^2 - \frac{1}{N} \left[\sum_{j=0}^{N-1} (x_j - \overline{x}) \right]^2 \right\} \tag{14.1.8}$$

如果$\overline{x}$是精确值,则第二项之和为零。除此之外,第一项将能很好地修正舍入误差。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::moment(Vec_I_DP &data, DP &ave, DP &adev, DP &sdev, DP &var, DP &skew,
    DP &curt)
给定数组 data[0...n-1],本程序返回均值 ave,平均偏差 adev,标准差 sdev,方差 var,偏斜度 skew 及峭度 curt。
{
    int j;
    DP ep=0.0,s,p;

    int n=data.size();
    if (n <= 1) nrerror("n must be at least 2 in moment");
    s=0.0;                                  第一步得到均值。
    for (j=0;j<n;j++) s += data[j];
    ave=s/n;
    adev=var=skew=curt=0.0;                 第二步得到偏离均值的一阶(绝对的)、
    for (j=0;j<n;j++) {                      二阶、三阶和四阶矩。
        adev += fabs(s=data[j]-ave);
```

```
            ep += s;
            var += (p=s*s);
            skew += (p *= s);
            curt += (p *= s);
        }
        adev /= n;
        var=(var-ep*ep/n)/(n-1);                校正的二步公式。
        sdev=sqrt(var);                         按照定义将这些部分结合在一起。
        if (var != 0.0) {
            skew /= (n*var*sdev);
            curt=curt/(n*var*var)-3.0;
        } else nrerror("No skew/kurtosis when variance = 0 (in moment)");
    }
```

14.1.1　深入讨论：半不变量

独立随机变量的均值和方差是可加的：如果 x 和 y 分别取样于两个不同的概率分布，则

$$\overline{(x+y)} = \overline{x} + \overline{y} \qquad \mathrm{Var}(x+y) = \mathrm{Var}(x) + \mathrm{Var}(x) \tag{14.1.9}$$

一般来说，高阶矩不可加，而它们一定的组合则是可加的，这种组合被称为**半不变量**。如果一种分布的中心矩用 M_k 表示：

$$M_k \equiv \left\langle (x_i - \overline{x})^k \right\rangle \tag{14.1.10}$$

因此，有 $M_2 = \mathrm{Var}(x)$，则前几个半不变量 I_k 由下式给出：

$$\begin{aligned} &I_2 = M_2 \qquad I_3 = M_3 \qquad I_4 = M_4 - 3M_2^2 \\ &I_5 = M_5 - 10M_2M_3 \qquad I_6 = M_6 - 15M_2M_4 - 10M_3^2 + 30M_2^3 \end{aligned} \tag{14.1.11}$$

注意，偏斜度和峭度((14.1.5)式和(14.1.6)式)仅是半不变量的简单幂，

$$\mathrm{Skew}(x) = I_3/I_2^{3/2} \qquad \mathrm{Kurt}(x) = I_4/I_2^2 \tag{14.1.12}$$

高斯分布所有高于 I_2 的半不变量都等于零，泊松分布所有半不变量等于其均值，更详尽的内容请参见本节参考文献[2]。

14.1.2　中位数和众数

概率分布函数 $p(x)$ 的中位数为 x_{med}，对此值而言，较大的 x 值和较小的 x 值是等概率的：

$$\int_{-\infty}^{x_{\mathrm{med}}} p(x)\,dx = \frac{1}{2} = \int_{x_{\mathrm{med}}}^{\infty} p(x)\,dx \tag{14.1.13}$$

由样本值 $x_0, \ldots, x_{N-1}$ 来估价一种分布的中位数的方式是，找出居中的那个数 x_i，应使其上下的样本数相等。当然对于 N 是偶数的情形是不可能的，对这种情况往往用两个中心值的平均来作为中位数。如果 x_j $(j = 0, \ldots, N-1)$ 以递增(或递减)的形式排序，则中位数的公式为

$$x_{\mathrm{med}} = \begin{cases} x_{(N-1)/2}, & N \text{ 为奇数} \\ \frac{1}{2}(x_{(N/2)-1} + x_{N/2}), & N \text{ 为偶数} \end{cases} \tag{14.1.14}$$

如果一种分布有较强的聚中趋势，使得它的大部分面积在某单一峰值下，则中位数是中心值的一种估算。它是比均值更强的估算量：中位数仅不适用于尾端的面积较大的情况，而均值则不适用于尾端一阶矩较大的情况。很容易构造这样的例子，即使尾端的面积可忽略，其一阶矩还是很大。

为了求得一组数据的中位数，首先要排序，然后利用(14.1.14)式，这是一个有 $N \log N$ 阶的过程。读者或许认为这是一种浪费，因为它产生了比中位数多得多的信息(即上下四分位点、十分位点等)。实际在 8.5 节中已看到由 N 阶数操作即可确定 $x_{(N-1)/2}$，可以参阅那一节的程序。

一个概率分布函数 $p(x)$ 的**众数**是 $p(x)$ 取极大值时的 x 值。众数主要在概率分布有一个单一尖锐的最大值时很有用，在该情况下，众数估算了中心值。有时一种分布将是**双峰的**，即有两个极大值，那么人们希望知道两个单独的众数。注意在这种情况下，均值和中位数并不是非常有用的，因为它们将仅能给出两峰值之间的“折衷”值。

参考文献和深入阅读

Bevington, P.R., and Robinson, D.K. 1992, *Data Reduction and Error Analysis for the Physical Sciences*, 2nd ed. (New York: McGraw-Hill), Chapter 1.

Stuart, A., and Ord, J.K. 1994, *Kendall's Advanced Theory of Statistics*, 6th ed. (London: Edward Arnold) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*], vol. 1, §10.15

Norusis, M.J. 1999, *SPSS 9.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall).

Chan, T.F., Golub, G.H., and LeVeque, R.J. 1983, *American Statistician*, vol. 37, pp. 242–247. [1]

Cramér, H. 1946, *Mathematical Methods of Statistics* (Princeton: Princeton University Press), §15.10. [2]

14.2 两种分布是否具有相同的均值和方差

很多时候我们很希望了解两种分布是否具有相同的均值。例如,第一组测量值是在某一事件前所收集的,而第二组测量值则是在该事件以后收集的,那么这一事件(一种"处理"或"在控制参数上的变化")是否会造成两组值之间的差异。

第一个问题是,一个样本的均值与另一个样本均值之间"有多大标准差"。实际上,这个数值是相当有用的。如果均值之差异是真实的,那么它与这种均值之差异的强度或"重要性"有关。然而,由其自身无法说明这种差别是真实的(即统计上是否显著)。如果数据点的数目较多,则均值之差要远小于标准差,但仍然可能是非常显著的;反之,如果数据是稀少的,则这一差可能中等大小,但统计上并不显著。在以下几章,将会多次遇到**强度**和**显著性**这些不同的概念。

测度均值之差显著性的量并不是区分它们的标准差之值,而是区分它们的**标准误**之值。一组数据的标准误差测度了用样本均值来估算总体(或"真正")均值的精度,一般来说,标准误等于差样本的标准差除以该样本点数的平方根。

14.2.1 针对显著不同均值的学生 t 检验

应用标准误的概念,测度均值之差显著性的常用统计量称之为**学生 t**。当两个分布被认为具有相同的方差、不同的均值时,学生 t 按以下方式计算,首先,由以下公式从"合并方差"估算均值之差的标准误 s_D:

$$s_D = \sqrt{\frac{\sum_{i\in A}(x_i - \overline{x_A})^2 + \sum_{i\in B}(x_i - \overline{x_B})^2}{N_A + N_B - 2}\left(\frac{1}{N_A} + \frac{1}{N_B}\right)} \tag{14.2.1}$$

其中每一次求和都是在第一组式第二组样本的所有点上进行的,每组样本都各有自己的均值,N_A 和 N_B 分别为第一组和第二组样本的点数。其次,计算 t:

$$t = \frac{\overline{x_A} - \overline{x_B}}{s_D} \tag{14.2.2}$$

第三步是利用(6.4.7)式和(6.4.9)式及6.4节中的程序 betai(不完全 B 函数)来计算 t 值的显著性值,它服从自由度为 $N_A + N_B - 2$ 的学生 t 分布。

显著性用0到1之间的数来表征,并且是对于具有相等均值的分布,其量 $|t|$ 恰好这样大或更大的概率。因此,小的显著性值(0.05或0.01)意味着所观察的差异是"非常显著的"。(6.4.7)式中的函数 $A(t/\nu)$ 等于1减去显著性。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::ttest(Vec_I_DP &data1, Vec_I_DP &data2, DP &t, DP &prob)
```

给定数组 data1[0..n1-1]和数组 data2[0..n2-1],此程序将学生 t 检验返回为 t,将它的显著性返回到 prob,小的 prob 值表明数组有显著不同的均值。数据数组假设取自于具有相同真实方差的总体。

```
{
    DP var1,var2,svar,df,ave1,ave2;

    int n1=data1.size();
    int n2=data2.size();
    avevar(data1,ave1,var1);
    avevar(data2,ave2,var2);
    df=n1+n2-2;                                自由度。
    svar=((n1-1)*var1+(n2-1)*var2)/df;         合并方差。
    t=(ave1-ave2)/sqrt(svar*(1.0/n1+1.0/n2));
    prob=betai(0.5*df,0.5,df/(df+t*t));        参见(6.4.9)式。
}
```

上述程序还使用了以下计算一组均值和方差的程序。

```
#include "nr.h"

void NR::avevar(Vec_I_DP &data, DP &ave, DP &var)
给定数组 data[0..n-1],返回它的均值为 ave,并且返回方差为 var。
{
    DP s,ep;
    int j;

    int n=data.size();
    ave=0.0;
    for (j=0;j<n;j++) ave += data[j];
    ave /= n;
    var=ep=0.0;
    for (j=0;j<n;j++) {
        s=data[j]-ave;
        ep += s;
        var += s*s;
    }
    var=(var-ep*ep/n)/(n-1);          校正的二步公式(14.1.8)。
}
```

以下考虑的情况是两种分布具有显著不同的方差,但是仍然想知道它们的均值是否相同或不同。(一种医治脱发的治疗已经使某些患者头发全部脱落,但使其它患者成为毛人,但我们想知道,平均来说,这种疗法是否对治疗脱发有效!)对不相等方差的 t 检验仍然有些疑问:如果两种分布有非常不同的方差,则它们在形状上可能极不相同,在这种情况下,均值的差异可能就不是特别有用了。

为了找出两组数据集中是否具有显著不同的方差,将使用本节后面叙述的 F 检验。

对于不相等方差 t 检验的相关统计量为

$$t=\frac{\overline{x_A}-\overline{x_B}}{[\mathrm{Var}(x_A)/N_A+\mathrm{Var}(x_B)/N_B]^{1/2}} \tag{14.2.3}$$

这个统计量近似为学生 t 分布,自由度为

$$\frac{\left[\dfrac{\mathrm{Var}(x_A)}{N_A}+\dfrac{\mathrm{Var}(x_B)}{N_B}\right]^2}{\dfrac{[\mathrm{Var}(x_A)/N_A]^2}{N_A-1}+\dfrac{[\mathrm{Var}(x_B)/N_B]^2}{N_B-1}} \tag{14.2.4}$$

表达式(14.2.4)一般不是整数,但对(6.4.7)式我们并不关心这点。程序如下:

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::tutest(Vec_I_DP &data1, Vec_I_DP &data2, DP &t, DP &prob)
给定数组 data1[0..n1-1]和 data2[0..n2-1],这个程序将学生 t 检验返回为 t,将它的显著性返回到 prob,小的 prob 值表
明这些数组有显著不同的均值。数据数组允许取自于不同方差的总体。
```

```
{
    DP var1,var2,df,ave1,ave2;

    int n1=data1.size();
    int n2=data2.size();
    avevar(data1,ave1,var1);
    avevar(data2,ave2,var2);
    t=(ave1-ave2)/sqrt(var1/n1+var2/n2);
    df=SQR(var1/n1+var2/n2)/(SQR(var1/n1)/(n1-1)+SQR(var2/n2)/(n2-1));
    prob=betai(0.5*df,0.5,df/(df+SQR(t)));
}
```

最后一个学生 t 检验的例子是**配对样本**。设想在两个样本中,方差的产生主要归因于两个样本逐点对应的完全相同的效应。例如,有两个工作候选者,每个人都要由招聘委员会的 10 个相同的委员做出评价。我们希望知道十个分数的均值是否显著不同。首先用上述程序 ttest 来试验,并得到一个并不特别显著的 prob 值(比如 >0.05)。但或许因为某些委员总是给高分、而其他委员总是给低分,这种倾向使显著性被消除,它增加了明显的方差,也因此减少了均值上任何差异的显著性。因此有配对样本公式:

$$\mathrm{Cov}(x_A,x_B)\equiv\frac{1}{N-1}\sum_{i=0}^{N-1}(x_{A_i}-\overline{x_A})(x_{B_i}-\overline{x_B}) \tag{14.2.5}$$

$$s_D=\left[\frac{\mathrm{Var}(x_A)+\mathrm{Var}(x_B)-2\mathrm{Cov}(x_A,x_B)}{N}\right]^{1/2} \tag{14.2.6}$$

$$t=\frac{\overline{x_A}-\overline{x_B}}{s_D} \tag{14.2.7}$$

其中 N 是每组样本的点数(配对数)。注意,重要的是每个特别的 i 值表示每个样本中的相应点,即配对的点。在(14.2.7)式中,t 统计量的显著性的计算仅有 $N-1$ 个自由度。程序如下:

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::tptest(Vec_I_DP &data1, Vec_I_DP &data2, DP &t, DP &prob)
给定配对数组 data1[0..n-1]和 data2[0..n-1],本程序将配对数据的学生 t 检验返回到 t,将它的显著性返回到 prob,小
的 prob 值表示均值有显著差别。
{
    int j;
    DP var1,var2,ave1,ave2,sd,df,cov=0.0;

    int n=data1.size();
    avevar(data1,ave1,var1);
    avevar(data2,ave2,var2);
    for (j=0;j<n;j++)
        cov += (data1[j]-ave1)*(data2[j]-ave2);
    cov /= df=n-1;
    sd=sqrt((var1+var2-2.0*cov)/n);
    t=(ave1-ave2)/sd;
    prob=betai(0.5*df,0.5,df/(df+t*t));
}
```

14.2.2 针对显著不同方差的 F 检验

F 检验是验证两个样本具有不同的方差的假设,它是通过否定它们方差实际上是一致的这一无效假设来进行的。统计量 F 是一个方差对另一个方差的比值,因此这个或 $\gg 1$ 或 $\ll 1$ 的值将意味着非常显著的差异。无效情形下的 F 的分布由(6.4.11)式给出,它是利用 betai 程序估值的。在一般情况下,可以通过非常大的 F 值或非常小的 F 值来证明(相同方差的)无效假设不成立,因此正确的显著性是**双尾的**,即两个不完整 Beta 函数的和。通过(6.4.3)式可以证明两个尾端总是相等的,因此仅需要计算一个,然后加倍它。偶而,当"无效假设"特别可行时,双尾的一致性会变得模糊,给出了一个大于 1 的指示概率。这时将概率变为 2 减去其自身,以使正确地交换尾端。基于这些考虑和(6.4.3)式,给出了如下程序:

```
#include "nr.h"

void NR::ftest(Vec_I_DP &data1, Vec_I_DP &data2, DP &f, DP &prob)
```

给定数组 data1[0..n1-1]和 data2[0..n2-1]，程序返回 f 值和它的显著性 prob。小的 prob 值表示这两个数组有显著不同的方差。

```
{
    DP var1,var2,ave1,ave2,df1,df2;

    int n1=data1.size();
    int n2=data2.size();
    avevar(data1,ave1,var1);
    avevar(data2,ave2,var2);
    if (var1 > var2) {                使 F 成为较大方差与较小方差之比。
        f=var1/var2;
        df1=n1-1;
        df2=n2-1;
    } else {
        f=var2/var1;
        df1=n2-1;
        df2=n1-1;
    }
    prob = 2.0*betai(0.5*df2,0.5*df1,df2/(df2+df1*f));
    if (prob > 1.0) prob=2.0-prob;
}
```

参考文献和深入阅读

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapter IX(B).

Norusis, M.J. 1999, *SPSS 9.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall).

14.3　两种分布是否不同

给出两组数据，我们能把上节提到的问题更一般化，并仅提出一个问题：两组数据是从同一分布函数中抽取的，还是从不同分布函数中抽取的？用适当的统计学语言来讲，即："在一定要求的显著性下是否能推翻'无效假设'，即两组数据集是从同一总体分布函数中抽取的假说。"否定了无效假设实际上就证明了数据集来自不同的分布，另外如果不能否定无效假设，也仅仅能证明数据集与一个单一的分布函数是一致的(相容的)。永远不能证明两组数据来自于单个的分布，因为，(例如)实际的数据量不能够区分两种仅相差 10^{10} 分之一量级的分布。

证明两种分布互不相同或者表明它们是相容的，这是一件一直出现在许多研究领域的任务：可见的星体是否均匀分布在天空中(即星体在天体中的位置作为赤纬的函数分布与天体区域作为赤纬函数的分布是否相同)；Brooklyn 的教育模式是否与 Bronx 的相同(即人的分布作为最后阶段受教育的函数是否相同)；两种牌号的荧光灯使用寿命的分布是否相同；对第一胎、第二胎、第三胎等等的孩子，他们出水痘的可能性是否相同。

这四个例子表明了以下两种不同的二分法的四种组合，即：(1)数据是连续的或离散的；(2)期望比较一组数据与一已知分布，或希望比较两个同样的未知分布的数据集。关于荧光灯和星体的数据集是连续的，因为能给出各个荧光灯的寿命和星球位置的表格，而水痘和教育水平的数据集则是离散的，因为我们得到的是以离散形式给出表：第一胎、第二胎等，或者 6 年级、7 年级等。另外，星球和水痘的"无效假设"是一个已知分布(天空中区域分布或在总人口中水痘的发生率)，而荧光灯和教育水准则涉及到两个同样都是未知数据组的比较(两种牌号，或 Brooklyn 和 Bronx)。

只要通过以下方式，人们总是能够将连续数据转化为量化的数据，即将事件分成不同的组，每一组的连续量有特定的范围：赤纬度是 0～10 度、10～20 度、20～30 度等。然而，离散量化过程总是损失一定的信息，量化级的选择也有相当的任意性。与其他研究者一样，我们也希望避免不必要的离散化。

对于离散分布之间的差异，可以接受的检验是 **χ^2 检验**，对于单变量函数的连续数据组，最常用的检验是 **Kolmogorov-Smirnov 检验**。将依次予以讨论。

14.3.1　χ^2 检验

假设观察到第 i 个量化区域内的事件数目为 N_i，并且根据某一已知分布所期望的数为 n_i，注意 N_i 是整

数，n_i 则未必是整数，则 χ^2 统计量为：

$$\chi^2=\sum_i\frac{(N_i-n_i)^2}{n_i} \tag{14.3.1}$$

其中求和是对所有量化级的，较大的 χ^2 值指出"无效假设"(即 N_i 是从以 n_i 表征的母体分布中抽取的)是相当不可能的。

(14.3.1)式中具有 $0=n_j=N_j$ 的任何第 j 项都应从和式中略去，如 $n_j=0, N_j\neq 0$，则 χ^2 趋于无穷大，这是正确的，因为这种情况下 N_i 所代表的离散值不可能从 n_i 所表征的总体中抽取。

χ^2 概率函数$Q(\chi^2|\nu)$是一个不完全的伽马函数，这已在 6.2 节中讨论过((6.2.18)式)。严格地说 $Q(\chi^2|\nu)$是ν 个具有单位方差的随机**正态**变量(均值为零)的平方和大于 χ^2 的概率。在求和式(14.3.1)中的项并不都是正态分布。然而，如果量化级的数较大($\gg 1$)或每一量化层中事件出现的次数较大时($\gg 1$)，则在无效假设情况下 χ^2 概率函数是对(14.3.1)式的一个很好的近似，用它来估计 χ^2 检验的显著性是标准的。

适当的 ν 值(自由度数)，引出了进一步的讨论。如果以固定的 n_i 表征的模型来收集数据，即不再为拟合总的事件观察数 $\sum N_i$ 而重新归一化，则 ν 等于离散量的总数 N_B(注意 N_B 并不是事件的总数)。更常见的是，n_i 事后被归一化，它的和等于 N_i 的和。在这种情况下，ν 的正确值等于 N_B-1，并且称模型有一约束(在下面程序中 knstrn = 1)。如果给出 n_i 的模型有一些附加的自由参数，这些参数为了与数据相符而根据实际情况来调整，则这些附加的"拟合"参数将减小 ν(且增加 knstrn)一个附加单位。

于是有如下程序。

```
#include "nr.h"

void NR::chsone(Vec_I_DP &bins, Vec_I_DP &ebins, const int knstrn, DP &df,
    DP &chsq, DP &prob)
给定数组 bins[0..nbins-1]包含事件的观测数，并给定数组 ebins[0..nbins-1]，包含所期望的事件数目，以及给定的约
束个数 knstrn(通常为 1)，本程序(平凡地)返回自由度的个数 df，并且(非平凡地)将 χ² 返回到 chsq，并将显著性返回到
prob。小的 prob 值表示分布 bins 和 ebins 之间有显著的差别，注意 bins 和 ebins 都是浮点型数组，虽然通常 bins 是整数值。
{
    int j;
    DP temp;

    int nbins=bins.size();
    df=nbins-knstrn;
    chsq=0.0;
    for (j=0;j<nbins;j++) {
        if (ebins[j] <= 0.0) nrerror("Bad expected number in chsone");
        temp=bins[j]-ebins[j];
        chsq += temp*temp/ebins[j];
    }
    prob=gammq(0.5*df,0.5*chsq);        χ² 概率函数，参见 6.2 节。
}
```

下面比较两个离散数据集，设 R_i 为第一组数据集第 i 个量化级中出现的事件数，S_i 为第二组数据集同样第 i 个量化级中出现的事件数。则 χ^2 统计量为

$$\chi^2=\sum_i\frac{(R_i-S_i)^2}{R_i+S_i} \tag{14.3.2}$$

比较(14.3.2)式和(14.3.1)式，应当注意(14.3.2)式中分母**并不是** R_i 和 S_i(它是(14.3.1)式中 n_i 的一个估计量)的平均。更确切地说，它是平均值的两倍，即为 R_i 与 S_i 之和。原因是假设在 χ^2 和式中的每一项都近似方差为 1 的正态分布量的平方。两正态量之差的方差等于它们各自方差的和，而不是平均。

如果数据是以这样的方式收集的，即 R_i 和式等于 S_i 和式，则自由度的个数等于量化级的数目减 1，N_B-1(即 knstrn = 1)。如果没有这一要求，自由度个数则为 N_B。例如，一位鸟类观察家想了解今年观察到的鸟种类的分布是否与去年相同。每一量化级对应于一个物种，如果观察者仅取他每年中首先看到的 1000 只鸟，则自由度的个数为 N_B-1。如果他选取随机样本日期中看到的所有鸟作为数据(每年选取同样的日期)，那么

自由度个数就是 N_B(knstrn = 0)。在后一种情况中,应注意观察者也还在检验这一年的鸟是否比其他年份的多,这是一额外的自由度。当然对数据组任何附加的约束都会降低自由度(即增加 knstrn 到一个更正的值)。

程序为:

```
#include "nr.h"

void NR::chstwo(Vec_I_DP &bins1, Vec_I_DP &bins2, const int knstrn, DP &df,
    DP &chsq, DP &prob)
给定数组 bins1[0..nbins-1]和数组 bins2[0..nbins-1]包含两组离散数据集,并且已知附加约束数 knstrn(正常情况为 0
或 1),本程序返回自由度的个数 df,χ² 检测的 chsq 和显著性 prob。小的 prob 值表示 bins1 和 bins2 两分布之间有显著差
别。注意 bins1 和 bins2 都是浮点型数组,尽管它们通常是整数值。
{
    int j;
    DP temp;

    int nbins=bins1.size();
    df=nbins-knstrn;
    chsq=0.0;
    for (j=0;j<nbins;j++)
        if (bins1[j] == 0.0 && bins2[j] == 0.0)
            --df;                          没有数据意味着自由度少 1。
        else {
            temp=bins1[j]-bins2[j];
            chsq += temp*temp/(bins1[j]+bins2[j]);
        }
    prob=gammq(0.5*df,0.5*chsq);           χ² 概率函数,参见 6.2 节。
}
```

(14.3.2)式和程序 chstwo 都用于如下情况,即两离散数据集有相同的总数据点数目。对于不相同数据点数目的情况,与(14.3.2)式类似的公式为

$$\chi^2 = \sum_i \frac{(\sqrt{S/R}R_i - \sqrt{R/S}S_i)^2}{R_i + S_i} \tag{14.3.3}$$

其中

$$R \equiv \sum_i R_i \qquad S \equiv \sum_i S_i \tag{14.3.4}$$

分别是数据点的对应数目。其程序可将 chstwo 相应改变而得到。

14.3.2　K－S 检验

K－S(Kolmogorov-Smirnov)检验主要用于非离散量化分布,其分布为某一自变量的函数,即对数据集来说,每一数据点都与一个单一的数相联系(如每一个灯泡的寿命或每一个星球的赤纬度)。这种情况下,数据集很容易被变换成从概率分布中抽取的累积分布函数的无偏估计量 $S_N(x)$:如果 N 个事件分别位于 x_i($i = 0,\ldots,N-1$)则 $S_N(x)$是一函数,其给出在每一给定 x 点的左边数据点的分数,这一函数在两连续(即排成递增顺序)点 x_i 之间是显然常数,并且在每一点 x_i 跳动相同的常数 $1/N$(参见图 14.3.1)。

由上面的过程,不同的分布或数据集给出不同的累积分布函数估计。然后所有累积分布函数在 x 的最小允许值(其为零)处和最大允许值(其为 1)处是一致的(当然最大值和最小值可能为 $\pm\infty$)。所以,正是最小值和最大值之间的状况区分了分布。

人们能够想到用任何统计量来测度两累积分布函数之间整体的差异:例如两者之间面积的绝对值,或者它们的积分均方差。K－S 检验中 D 是一种特别简单的测度:它被定义为两累积分布函数绝对差的最大值。因此,对于比较一组数据集的 $S_N(x)$与一已知分布累积函数 $P(x)$来说,K－S 统计量是

$$D = \max_{-\infty<x<\infty} |S_N(x) - P(x)| \tag{14.3.5}$$

而对于比较两个不同的累积分布函数 $S_{N_1}(x)$和 $S_{N_2}(x)$,则 K－S 统计量为

$$D = \max_{-\infty<x<\infty} |S_{N_1}(x) - S_{N_2}(x)| \tag{14.3.6}$$

使得 K－S 统计量有用的是在"无效假设"情况下(即数据组抽样于同一分布),至少对有用的近似来说,它的分布能够计算,因此给出了任何非零 D 的观察值的显著性。K－S 检验的主要特点是在 x 重新参数化过

程中的不变性,换言之,能够局部地滑动或伸缩图 14.3.1 中的 x 轴,而最大距离 D 保持不变。例如,使用 $\log x$ 与使用 x 可以得到相同的显著性。

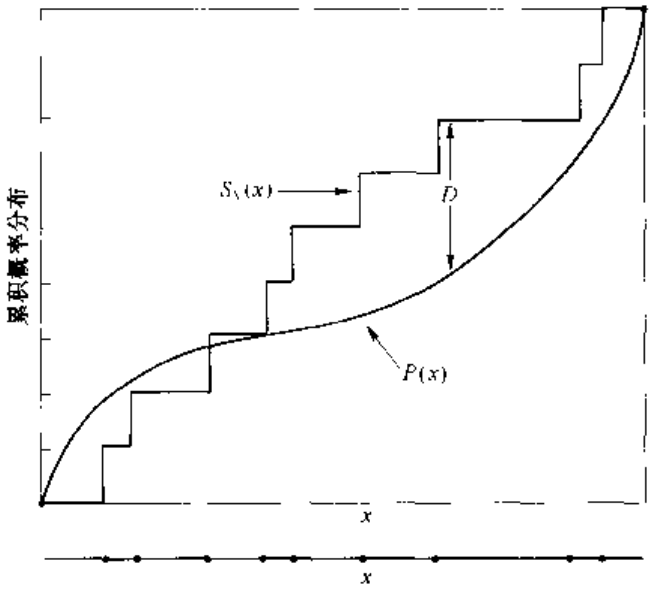


图 14.3.1 K-S统计量 D. 在 x 点值的已测分布(如图下面的横坐标上 N 个点)与一个理论分布进行比较,此理论分布的累积概率分布如 $P(x)$ 所示。构造了阶梯型累积分布 $S_N(x)$,这是一个在每个测量点等量上升的分布。D 是这两个累积分布之间的最大距离

参与显著性计算的函数可写成如下和式:

$$Q_{KS}(\lambda) = 2\sum_{j=1}^{\infty}(-1)^{j-1}\, e^{-2j^2\lambda^2} \tag{14.3.7}$$

这是一个单调函数,极限值为

$$Q_{KS}(0) = 1 \qquad Q_{KS}(\infty) = 0 \tag{14.3.8}$$

根据这个函数,正如两个分布是相同的这一无效假设不可证明一样,D 观察值的显著性水平近似由下式给出:

$$\text{Probability}\ (D > \text{observed}\ ) = Q_{KS}\left(\left[\sqrt{N_e} + 0.12 + 0.11/\sqrt{N_e}\right] D\right) \tag{14.3.9}$$

其中,N_e 是数据点的有效个数,当一个分布属于(14.3.5)式的情况时,$N_e = N$,而当两个分布属于(14.3.6)式的情况时:

$$N_e = \frac{N_1 N_2}{N_1 + N_2} \tag{14.3.10}$$

其中,N_1 是第一个分布中数据点的个数,N_2 是第二个分布中数据点的个数。

包含在(14.3.9)式中的逼近性质是当 N_e 变大时,它也渐近精确。但对 $N_e \geqslant 4$ 时,它已足够好,正如人们可能曾经实际使用过的同样小的数[1]。

因此,对一个式两个分布的情况我们有以下程序:

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::ksone(Vec_IO_DP &data, DP func(const DP), DP &d, DP &prob)
给定数组 data[0..n-1],并给定一个用户提供的单变量的函数 func,它是一个范围从 0(其变量的最小值)到 1(其变量的
最大值)的累积分布概率,本程序返回 K-S 统计量 d 以及显著性水平 prob。小的 prob 值表明 data 的累积分布函数明显
不同于 func。数组 data 修改为按递增次序排列。
{
    int j;
    DP dt,en,ff,fn,fo=0.0;

    int n=data.size();
    sort(data);                          如果数据已经按递增排序,则此调用可以省略。
    en=n;
    d=0.0;
```

```
    for (j=0;j<n;j++) {                          关于排序数据点的循环。
        fn=(j+1)/en;                             在这步以后，数据的累积分布函数。
        ff=func(data[j]);                        与用户提供的函数比较。
        dt=MAX(fabs(fo-ff),fabs(fn-ff));         最大距离。
        if (dt > d) d=dt;
        fo=fn;
    }
    en=sqrt(en);
    prob=probks((en+0.12+0.11/en)*d);            计算显著性。
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::kstwo(Vec_IO_DP &data1, Vec_IO_DP &data2, DP &d, DP &prob)
给定数组 data1[0..n1-1]和数组 data2[0..n2-1]，本程序返回 K-S 统计量 d，以及关于数据集取自于相同分布的无效
假设的显著性水平 prob。小的 prob 值表明 data1 的累积分布函数显著不同于 data2 的累积分布函数。数组 data1 和 data2
修改为按递增顺序排序
{
    int j1=0,j2=0;
    DP d1,d2,dt,en1,en2,en,fn1=0.0,fn2=0.0;

    int n1=data1.size();
    int n2=data2.size();
    sort(data1);
    sort(data2);
    en1=n1;
    en2=n2;
    d=0.0;
    while (j1 < n1 && j2 < n2) {                                  如果没做完……
        if ((d1=data1[j1]) <= (d2=data2[j2])) fn1=j1++/en1;       下一步在 data1 中。
        if (d2 <= d1) fn2=j2++/en2;                               下一步在 data2 中。
        if ((dt=fabs(fn2-fn1)) > d) d=dt;
    }
    en=sqrt(en1*en2/(en1+en2));
    prob=probks((en+0.12+0.11/en)*d);                             计算显著性。
}
```

上述两个程序都使用了如下程序进行函数Q_{KS}的计算：

```
#include <cmath>
#include "nr.h"
using namespace std;

DP NR::probks(const DP alam)
kolmogorov-Smirnov 概率函数。
{
    const DP EPS1=1.0e-6,EPS2=1.0e-16;
    int j;
    DP a2,fac=2.0,sum=0.0,term,termbf=0.0;

    a2 = -2.0*alam*alam;
    for (j=1;j<=100;j++) {
        term=fac*exp(a2*j*j);
        sum += term;
        if (fabs(term) <= EPS1*termbf || fabs(term) <= EPS2*sum) return sum;
        fac = -fac;              变换和式中的符号。
        termbf=fabs(term);
    }
    return 1.0;                  仅仅因为不收敛而到达此。
}
```

14.3.3　深入讨论：K－S 检验的变形

K－S 检验对一个积累分布函数 $P(x)$偏差的灵敏度是不独立于 x 的。实际上，K－S 检验在中位数附近最灵敏，此时 $P(x)=0.5$；在分布的最末端最不灵敏，$P(x)$接近 0 或 1。原因是在无效假设中，$|S_N(x)-P(x)|$并没有独立于变量 x 的概率分布。更正确地说，它的变化正比于 $P(x)[1-P(x)]$，其在 $P=0.5$ 处取最大

值。因为(14.3.5)式中的K-S统计量是两累积分布函数在整个 x 轴的最大差值,一个在其自身 x 值上统计显著的偏差,将变得可与 $P=0.5$ 处所希望的随机偏差相比拟,因此它是打折扣的。结果是K-S检验善于寻找一种概率分布的偏移,特别是中位数的改变,但它并不总是善于寻找发散,发散更能影响概率分布的末端,而使中位数保持不变。

增加K-S统计量末端作用的一种方式是,用一个称为**稳定的**或**加权的**统计量[2~4]代替(14.3.5)式中的 D,例如 **Anderson-Darling 统计量**,

$$D^* = \max_{-\infty<x<\infty} \frac{|S_N(x) - P(x)|}{\sqrt{P(x)[1-P(x)]}} \tag{14.3.11}$$

遗憾的是,虽然Noé[5]利用递推关系给出了一种计算方法,并提供了数值结果图,对此统计量,没有类似于(14.3.7)式和(14.3.9)式的简单公式。还有许多其他可能相似的统计量,例如

$$D^{**} = \int_{P=0}^{1} \frac{[S_N(x) - P(x)]^2}{P(x)[1-P(x)]} dP(x) \tag{14.3.12}$$

也是由Anderson和Darling讨论的[3]。

另一种更简单直接的方法应归于Kuiper[6,7],我们已经知道标准K-S检验在变量 x 的重新参数化下是不变的。一个更一般的能保证在所有 x 值上有相等灵敏度的对称性环绕 x 轴成一个圆($\pm\infty$点是等价的),并且寻找一个统计量,此量在该圆上所有偏移和参数化下都是不变的。例如,它允许在 x 的某一中心值"切断"概率分布并交换左右两边,而不改变统计量和其显著性。

Kuiper 的统计量定义为

$$V = D_+ + D_- = \max_{-\infty<x<\infty} [S_N(x) - P(x)] + \max_{-\infty<x<\infty} [P(x) - S_N(x)] \tag{14.3.13}$$

它是 $S_N(x)$ 在 $P(x)$ 上方和下方最大距离之和,应确信这一统计量在圆上具有所希望的不变性:因为角度可超过360度许多倍,因此可以作为围绕圆所张角度的函数,画出定义在圆上两概率分布的无穷限积分。如果改变积分的起点,D_+ 和 D_- 将各自发生变化,但它们的和则保持不变。

因此,对于统计量 V 的渐近分布有一简单的公式,可直接模拟(14.3.7)式~(14.3.10)式。设

$$Q_{KP}(\lambda) = 2\sum_{j=1}^{\infty}(4j^2\lambda^2 - 1)e^{-2j^2\lambda^2} \tag{14.3.14}$$

它是单调的,并满足

$$Q_{KP}(0) = 1 \qquad Q_{KP}(\infty) = 0 \tag{14.3.15}$$

根据这一函数,显著性水平为[1]

$$\text{概率}\ (V > \text{观察值}\ ) = Q_{KP}\left(\left[\sqrt{N_e} + 0.155 + 0.24/\sqrt{N_e}\right] V\right) \tag{14.3.16}$$

在一组样本情况下,N_e 等于 N;而在两组样本情况下,则为(14.3.10)式的值。

当然,Kuiper检验对原本定义在一个圆上的任何问题都是理想的,例如检验某物在经度上的分布是否与理论一致,或两物在经度上是否有不同的分布[8]。

将类似于ksone,kstwo和probks程序的编码留给读者(对 $\lambda<0.4$,不要试图对(14.3.14)式求和,它的值是1至7的数字,但该级数需要许多项才能收敛,并失去舍入精度)。

最后两个应当引起注意的问题是:第一,所有K-S检验的随机变量都缺少区分某种分布的能力。一个简单的例子是,一个带有窄"凹口"的分布,在"凹口"内概率为零。当然在这一"凹口"内即使只有一个数据点的存在都会将这一分布排除,但是因为它的积累性质,K-S检验在对偏差预告之前需要在"凹口"内有许多数据点。

其次应注意,如果从一数据中估算任何参数(如均值和方差),则对于使用了这种估计参数的累积分布函数 $P(x)$,K-S统计量 D 的分布再也不是由(14.3.9)式给出。一般来说,必须自己来决定新的分布,如利用蒙特卡罗(Monte Carlo)方法。

参考文献和深入阅读

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapters IX(C) and IX(E).

Stephens, M.A. 1970, *Journal of the Royal Statistical Society*, ser. B, vol. 32, pp. 115–122. [1]
Anderson, T.W., and Darling, D.A. 1952, *Annals of Mathematical Statistics*, vol. 23, pp. 193–212. [2]
Darling, D.A. 1957, *Annals of Mathematical Statistics*, vol. 28, pp. 823–838. [3]
Michael, J.R. 1983, *Biometrika*, vol. 70, no. 1, pp. 11–17. [4]
Noé, M. 1972, *Annals of Mathematical Statistics*, vol. 43, pp. 58–64. [5]
Kuiper, N.H. 1962, *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, ser. A., vol. 63, pp. 38–47. [6]
Stephens, M.A. 1965, *Biometrika*, vol. 52, pp. 309–321. [7]
Fisher, N.I., Lewis, T., and Embleton, B.J.J. 1987, *Statistical Analysis of Spherical Data* (New York: Cambridge University Press). [8]

14.4　两种分布的列联表分析

在本节和下两节，我们将处理两种分布之间**关联的测度**。情况是这样的：每个数据点都有两个或更多个与它相联系的不同的量，并且我们也希望知道是否对某一量的了解有利于预报另一量值。在许多情况下，一个变量是"独立"或"控制"变量，而另一变量则是"非独立"或"观测"变量。那么，可能希望知道是否后一变量实际上依赖于或关联于前一变量。如果确实如此，希望对这一关联的强度有定量的测度。人们通常不太严谨地将其表述为两变量是**相关**或是**不相关**，但我们将保留那些特殊关联（线性的，或者至少是单调的）的术语，这将在 14.5 节和 14.6 节讨论。

应当注意的是，就像前面几节一样，显著性和强度概念的差异表现在：即使两种分布之间的关联相当弱，如果数据量足够大，这种关联仍是非常显著的。

用不同类型形成松散的层次结构来区分某些不同种类的变量是很有用的。

- 一个变量，当它的值是某一无序集合的成员时，则称其为**公称的**。例如"居住的州"是一个公称变量，在美国，这是取 50 个值中的一个；在天文学上，"星系的类型"是一个公称变量，它取三个值："螺旋星系"、"椭圆星系"和"不规则星系"。
- 当一变量的值是某一离散而有序集合的元素时，则称该变量是**有序的**。例如：学校的年级、太阳系行星的次序（水星 = 1，金星 = 2，...）、子孙的数目等，在一有序变量的值之间不必有"相等度量距离"的概念，而仅需要它们是内在有序的。
- 我们将称一个变量为**连续的**，如果它的值是实数，如距离、时间、温度等（有的社会学家还区别区间和比率连续变量，但我们并不认为这种区分是必需的）。

一个连续变量总是可以通过放置入值域变为有序量，如果忽略放置顺序，则有序量又可变为公称变量。公称变量构成了最低层次的结构，因而也是最一般的。例如，一组几个连续或有序变量的集合可以粗糙地转变成单个的公称变量，即先量化分级，后将每个离散的量化级看成一个单一的公称变量，当多维数据较稀疏时，这是惟一明智的处理方式。

本节余下部分将讨论两个公称变量之间关联的测度。对任何一对公称变量，数据能用一**列联表**来显示，该表的行由一个公称变量的值来标号，列由另一个公称变量值来标号，表的项目是非负整数，它们给出了行和列的每一组合的观测事件的数目（参见图 14.4.1）。这种对公称变量之间关联的分析称为**列联表分析**或**交叉列表分析**。

我们将引入两种不同的方法，第一种方法是基于 χ^2 统计量，它能较好地表征这种关联的显著性，但它用作强度的测度却很一般（主要由于其数值没有任何非常直接的解释）。第二种方法是基于信息论中**熵**的概念，它根本没有引入关联显著性（要用到 χ^2），但却能非常明确地表征已知很显著的关联强度。

14.4.1　基于 χ^2 的关联测度

首先，设 N_{ij} 表示第一个变量 x 取第 i 个值、第二个变量 y 取第 j 个值时事件发生的数目。设 N 表示总事件数，即等于所有 N_{ij} 的和。设 $N_{i.}$ 表示第一变量 x 取第 i 值而无论第二变量 y 取何值时事件发生的数目，

$N_{\cdot j}$表示y取第j个值而无论x取何值时事件发生的数目,因此有:

$$N_{i\cdot} = \sum_j N_{ij} \qquad N_{\cdot j} = \sum_i N_{ij}$$
$$N = \sum_i N_{i\cdot} = \sum_j N_{\cdot j} \tag{14.4.1}$$

$N_{\cdot j}$和$N_{i\cdot}$有时称为**行和列总数**或**边缘值**,需谨慎使用这些术语。因为不能直接确定哪个是行,哪个是列。

	0 红	1 绿	…	
0 男	红男 N_{00}	绿男 N_{01}	…	男性 $N_{0\cdot}$
1 女	红女 N_{10}	绿女 N_{11}	…	女性 $N_{1\cdot}$
⋮	⋮	⋮	…	⋮
	红 $N_{\cdot 0}$	绿 $N_{\cdot 1}$	…	总数 N

图 14.4.1 两个公称变量(这里是性别和颜色)列联表的示例。行和列的旁注(总数)也被给出,变量是"公称"的,即它们的值排列的次序是任意的,并且不会影响列表结果。如果值次序有一定内在的意义,则变量是"有序的"或"连续的",并且可以使用14.5节~14.6节的相关分析

无效假设是两变量x和y没有联系。在这种情况下,给定某一特定y值时某一特定x值的概率,等于无论y取何值时x取该值的概率。因此,在无效假设的情况下,对任何N_{ij}所期望的数(记为n_{ij})可仅从行和列总数中计算出来:

$$\frac{n_{ij}}{N_{\cdot j}} = \frac{N_{i\cdot}}{N} \qquad \text{意味着} \qquad n_{ij} = \frac{N_{i\cdot}N_{\cdot j}}{N} \tag{14.4.2}$$

注意,如果某一列或行的总数为零,则在那一列或行中所有项的期望数也都为零。在这种情况下,x或y的从不发生的量化级非常简单,我们不予分析。

χ^2统计量现在由(14.3.1)式给出,在目前情况下,它是表中所有项的和:

$$\chi^2 = \sum_{i,j} \frac{(N_{ij} - n_{ij})^2}{n_{ij}} \tag{14.4.3}$$

自由度的个数等于表中项目的数目(即行和列大小的乘积)减去约束条件数,这些约束条件数是从数据本身的使用中得到,而这些数据决定了n_{ij}。每个行总数和列总数都是一个约束条件(去除已多计算的一个),因为行和列总数之和都等于N,即数据点的总数。因此,如果表的大小是$I \times J$,则自由度数等于$IJ - I - J + 1$。(14.4.3)式连同χ^2概率函数(参见6.2节)给出了变量x和y之间关联的显著性。

假如有显著的联系,那么如何才能定量它的强度,以使我们能比较一个关联与另一关联的强度?这里的想法是找到χ^2的某一重新参数化,其映射自身到更适当的区间,如从0到1,此区间中的结果并不取决于取样的数据量,而取决于数据取样的总体,有几种不同的方式可做到这一点,最常见的两种是**Cramer V**和**列联系数 C**。

Cramer V定义为

$$V = \sqrt{\frac{\chi^2}{N \min(I-1, J-1)}} \tag{14.4.4}$$

其中 I 和 J 是行和列的数目，N 是事件的总数。Cramer V 具有位于 0 和 1 之间(包括 0 和 1)的优良性质。当没有关联时等于 0，仅当关联完备时等于 1。任意行中所有事件都位于惟一的列中，反之亦然。(按(国际)象棋术语说，任何两个置于非零表项上的车都不能相互拼杀。)

在 $I=J=2$ 的情况下，Cramer V 也称为 **phi 统计量**。

列联系数 C 定义为：

$$C=\sqrt{\frac{\chi^2}{\chi^2+N}} \tag{14.4.5}$$

它也位于 0 和 1 之间。但它不能达到上限，当它被用于比较带有相同 I 和 J 的两个表的强度时，它的上限取决于 I 和 J。因此，它不能用于比较两个不同大小的表。

Cramer V 和列联系数 C 的问题是：当它们取两端部之间的值时，不能直接解释这些值意味着什么。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::cntabl(Mat_I_INT &nn, DP &chisq, DP &df, DP &prob, DP &cramrv, DP &ccc)
给定导整数型数组 nn[0..ni-1][0..nj-1]的二维列联表，程序返回 χ² 的 chisq，自由度个数 df，显著性水平 prob(小的值
表示显著关联)，以及两个关联量 Cramer V(cramrv)和关联系数 C(ccc)。
{
    const DP TINY=1.0e-30;                  一个小数。
    int i,j,nnj,nni,minij;
    DP sum=0.0,expctd,temp;

    int ni=nn.nrows();
    int nj=nn.ncols();
    Vec_DP sumi(ni),sumj(nj);
    nni=ni;                                 行数，
    nnj=nj;                                 列数，
    for (i=0;i<ni;i++) {                    得到行总数。
        sumi[i]=0.0;
        for (j=0;j<nj;j++) {
            sumi[i] += nn[i][j];
            sum += nn[i][j];
        }
        if (sumi[i] == 0.0) --nni;          通过减少个数消去任何零行。
    }
    for (j=0;j<nj;j++) {                    得到列的总数。
        sumj[j]=0.0;
        for (i=0;i<ni;i++) sumj[j] += nn[i][j];
        if (sumj[j] == 0.0) --nnj;          消去任何零列。
    }
    df=nni*nnj-nni-nnj+1;                   校正的自由度个数。
    chisq=0.0;
    for (i=0;i<ni;i++) {                    求 χ² 和。
        for (j=0;j<nj;j++) {
            expctd=sumj[j]*sumi[i]/sum;
            temp=nn[i][j]-expctd;
            chisq += temp*temp/(expctd+TINY);        这里 TINY 保证消去的行和列不
        }                                            再作用于和式。
    }
    prob=gammq(0.5*df,0.5*chisq);                    χ² 概率函数。
    minij = nni < nnj ? nni-1 : nnj-1;
    cramrv=sqrt(chisq/(sum*minij));
    ccc=sqrt(chisq/(chisq+sum));
}
```

14.4.2　基于熵的关联测度

考虑"二十个问题"游戏，即通过重复提问是/否，除了未知目标一种正确的可能性外，游戏者尽量消除所有的可能性。考虑一个更一般的游戏，即允许提问双项选择的问题(是/否)，也可以提问多项选择问题。但多项选择必须是相互排斥且详尽的(正像"是"和"否"一样)。

对你来说，一个答案的值将随所消除的可能性的数目的增加而增加。更特别的是，一个答案如仅提供以分数 p 表征的可能性，那将被赋予值 $-\ln p$(因为 $p<1$，它是一正数)。对数的作用是使这一值有可加性，例如

一个(仅能消除 1/6 可能性的问题与两个以因子 1/2 和 1/3 按顺序减少的问题是一样的。

因此这是一个答案的值,但什么是一个问题的值?如果一个问题有 I 种可能的答案($i=0,\ldots,I-1$),第 i 个答案的可能性的分数为 p_i(所有 p_i 的和等于 1),那么问题的值即为答案值的期望值,记为 H,

$$H=-\sum_{i=0}^{I-1}p_i\ln p_i \tag{14.4.6}$$

在计算(14.4.6)式时,应注意

$$\lim_{p\to 0}p\ln p=0 \tag{14.4.7}$$

值 H 在 0 和 $\ln I$ 之间,仅当一个 p_i 等于 1 其他为零时 $H=0$。在这种情况下,问题毫无意义,因为答案是预先规定好的。当所有 p_i 相等时,H 取最大值。在这种情况下,除去分数 $1/I$ 的可能性外问题定然会消去其他全部剩余可能性。

H 值通常被称为由 p_i 确定的概率分布的熵,这一术语出自于统计物理学。

迄今为止,还未讨论两变量的关联。假设我们正在决定在游戏中下一步提什么问题,并在两个候选者中选择,或者可能想按一种顺序或另一种顺序问两个问题。假设一个问题 x 有 I 种可能的答案,每个标记为 i。另一个问题 y 有 J 种可能的答案,每个标记为 j。那么,对两个问题提问的可能结果构成了列联表,其每一项为 N_{ij},如果总数 N 归一化,则能给出所有概率 p 的信息,特别是利用(14.4.1)式可以得到

$$\begin{aligned}
p_{ij}&=\frac{N_{ij}}{N}\\
p_{i\cdot}&=\frac{N_{i\cdot}}{N}\qquad(\text{只有问题 }x\text{ 时的结果})\\
p_{\cdot j}&=\frac{N_{\cdot j}}{N}\qquad(\text{只有问题 }y\text{ 时的结果})
\end{aligned} \tag{14.4.8}$$

问题 x 和 y 的熵分别是

$$H(x)=-\sum_i p_{i\cdot}\ln p_{i\cdot}\qquad H(y)=-\sum_j p_{\cdot j}\ln p_{\cdot j} \tag{14.4.9}$$

两个问题一起的熵为(共熵)

$$H(x,y)=-\sum_{i,j}p_{ij}\ln p_{ij} \tag{14.4.10}$$

什么是给定 x 情况下问题 y 的熵(即如果首先问 x)?它是位于列联表中(相当于对 x 的答案)某单一列 y 分布受限的熵在所有 x 上的期望值:

$$H(y|x)=-\sum_i p_{i\cdot}\sum_j\frac{p_{ij}}{p_{i\cdot}}\ln\frac{p_{ij}}{p_{i\cdot}}=-\sum_{i,j}p_{ij}\ln\frac{p_{ij}}{p_{i\cdot}} \tag{14.4.11}$$

反之,给定 y,x 的熵为

$$H(x|y)=-\sum_j p_{\cdot j}\sum_i\frac{p_{ij}}{p_{\cdot j}}\ln\frac{p_{ij}}{p_{\cdot j}}=-\sum_{i,j}p_{ij}\ln\frac{p_{ij}}{p_{\cdot j}} \tag{14.4.12}$$

很容易证明给定 x 时 y 的熵将永远不会超过单独 y 的熵,即先问 x 仅能减少提问 y 的有用性(在这种情况下,两变量是相关联的):

$$\begin{aligned}
H(y|x)-H(y)&=-\sum_{i,j}p_{ij}\ln\frac{p_{ij}/p_{i\cdot}}{p_{\cdot j}}\\
&=\sum_{i,j}p_{ij}\ln\frac{p_{\cdot j}p_{i\cdot}}{p_{ij}}\\
&\leqslant\sum_{i,j}p_{ij}\left(\frac{p_{\cdot j}p_{i\cdot}}{p_{ij}}-1\right)\\
&=\sum_{i,j}p_{i\cdot}p_{\cdot j}-\sum_{i,j}p_{ij}\\
&=1-1=0
\end{aligned} \tag{14.4.13}$$

其中使用了以下不等式：

$$\ln w \leqslant w - 1 \tag{14.4.14}$$

现在，已经具有定义 y 对 x"依赖性"(即关联的测度)的一切准备，这一测度有时称为 y 的**不确定系数**，记为 $U(y|x)$，

$$U(y|x) \equiv \frac{H(y) - H(y|x)}{H(y)} \tag{14.4.15}$$

这一测度在 0 和 1 之间，0 值意味着 x 和 y 没有联系，1 值意味着对 x 的了解完全能预测 y。至于中间值，$U(y|x)$给出了 y 的熵 $H(y)$的分数，如果 x 已知，则此熵丢失(即有 x 的信息，此熵就是冗余的)。在"二十个问题"游戏中，如果首先提问 x，则 $U(y|x)$是问题 y 有用性分数的损失。

如果希望以 y 作为自变量，x 作为依赖变量，可以通过交换 x 和 y 得到 x 对 y 的依赖性：

$$U(x|y) \equiv \frac{H(x) - H(x|y)}{H(x)} \tag{14.4.16}$$

如果希望对称地处理 x 和 y，则有用的组合是

$$U(x,y) \equiv 2\left[\frac{H(y) + H(x) - H(x,y)}{H(x) + H(y)}\right] \tag{14.4.17}$$

如果两个变量完全独立，则 $H(x,y) = H(x) + H(y)$，因此(14.4.17)式等于零。如果两个变量完全依赖，则 $H(x) = H(y) = H(x,y)$，因此(14.4.16)式等于 1。实际上，可使用等式(很容易从(14.4.9)式～(14.4.12)式得到)

$$H(x,y) = H(x) + H(y|x) = H(y) + H(x|y) \tag{14.4.18}$$

并得到

$$U(x,y) = \frac{H(x)U(x|y) + H(y)U(y|x)}{H(x) + H(y)} \tag{14.4.19}$$

即对称测度只是两非对称测度(14.4.15)式和(14.4.16)式的加权平均，权重分别为每一变量的熵。

以下是一程序，它可以计算所讨论的所有量 $H(x)$，$H(y)$，$H(x|y)$，$H(y|x)$，$H(x,y)$，$U(x|y)$，$U(y|x)$ 和 $U(x,y)$。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::cntab2(Mat_I_INT &nn, DP &h, DP &hx, DP &hy, DP &hygx, DP &hxgy,
    DP &uygx, DP &uxgy, DP &uxy)
给定一个整数型数组 nn[i][j]的二维列联表，其中 i 标记 x 变量，其范围从 0 至 ni-1；j 标记 y 变量，其范围从 0 至 nj-1。
本程序返回整个表的熵 h；x 分布的熵 hx，y 分布的熵 hy，已知 x 后 y 的熵 hygx，已知 y 后 x 的熵 hxgy，y 关于 x 的依赖性
uygx((14.4.15)式)，x 关于 y 的依赖性 uxgy((14.4.16)式)，对称依赖性 uxy((14.4.17)式)。
{
    const DP TINY=1.0e-30;                    一个小数。
    int i,j;
    DP sum=0.0,p;

    int ni=nn.nrows();
    int nj=nn.ncols();
    Vec_DP sumi(ni),sumj(nj);
    for (i=0;i<ni;i++) {                      得到行总数。
        sumi[i]=0.0;
        for (j=0;j<nj;j++) {
            sumi[i] += nn[i][j];
            sum += nn[i][j];
        }
    }
    for (j=0;j<nj;j++) {                      得到列总数。
        sumj[j]=0.0;
        for (i=0;i<ni;i++)
            sumj[j] += nn[i][j];
    }
    hx=0.0;                                   x 分布的熵，
```

```
    for (i=0;i<ni;i++)
        if (sumi[i] != 0.0) {
            p=sumi[i]/sum;
            hx -= p*log(p);
        }
    hy=0.0;                                  y 分布的熵。
    for (j=0;j<nj;j++)
        if (sumj[j] != 0.0) {
            p=sumj[j]/sum;
            hy -= p*log(p);
        }
    h=0.0;
    for (i=0;i<ni;i++)                       总熵：关于 x 和 y 的循环。
        for (j=0;j<nj;j++)
            if (nn[i][j] != 0) {
                p=nn[i][j]/sum;
                h -= p*log(p);
            }
    hygx=h-hx;                               使用(14.4.18)式。
    hxgy=h-hy;
    uygx=(hy-hygx)/(hy+TINY);                求(14.4.15)式。
    uxgy=(hx-hxgy)/(hx+TINY);                求(14.4.16)式。
    uxy=2.0*(hx+hy-h)/(hx+hy+TINY);          求(14.4.17)式。
}
```

参考文献和深入阅读

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

Norusis, M.J. 1999, *SPSS 9.0 Guide to Data Analysis* (Englewood Cliffs, NJ. Prentice-Hall).

Fano, R.M. 1961, *Transmission of Information* (New York: Wiley and MIT Press), Chapter 2.

14.5 线性相关

下面将继续讨论两变量之间关联的测度，这两个变量是顺序量或连续量，而不是公称量。最常用的是**线性相关系数**。对数据对(x_i, y_i)，$i=0,\ldots,N-1$，线性相关系数 r(也称为乘积矩相关系数，或 **Pearson *r***)定义为

$$r=\frac{\sum_i (x_i-\overline{x})(y_i-\overline{y})}{\sqrt{\sum_i (x_i-\overline{x})^2}\sqrt{\sum_i (y_i-\overline{y})^2}} \tag{14.5.1}$$

其中$\overline{x}$和$\overline{y}$分别是 x_i 和 y_i 的均值。

r 的值位于 -1 和 1 之间，包括 -1 和 1。当数据点完全位于一条正斜率的直线上(x 和 y 一起增加)时，r 取值为 1，称为**完全正相关**。值 1 表示斜率大小的独立性。当完全位于一条负斜率的直线上(x 增加而 y 减少)则 r 取 -1，它被称为**完全负相关**。r 接近 0 表明变量 x 和 y 是**不相关的**。

当已知一个相关是显著的，则 r 是测度强度的惯用方式。实际上，r 的值等价于以下表述，即当数据利用最小二乘方拟合成一条直线时(参见 15.2 节中(15.2.13)式和(15.2.14)式)，期望什么样的线差(偏差的均方根)呢？遗憾的是，若要确定一个观察到的相关性是否在统计上是显著的，或者一个观察到的相关性是否显著地比另一个强，r 是个相当差的统计量，其原因是 r 对 x 和 y 的单独分布是未知的，因此在无效假设情况下，没有任何一般的方式可计算其分布。

仅能做的工作是：如果无效假设为 x 和 y 是不相关的，而且如果关于 x 和 y 的分布都有足够的收敛矩(其末端下降很快)，又如果 N 很大(典型 >500)，则 r 近似为正态分布，均值为 0，标准差为 $1/\sqrt{N}$。在这种情况下，相关的(双边)显著性，即无效假设下，$|r|$ 大于观察值的概率为

$$\mathrm{erfc}\left(\frac{|r|\sqrt{N}}{\sqrt{2}}\right) \tag{14.5.2}$$

其中 erfc(x)是补余误差函数，参见(6.2.8)式，它由 6.2 节的程序 erffc 或 erfcc 计算。(14.5.2)式的较小值表明两分布是显著相关的(下面(14.5.9)式是更精确的检验)。

许多统计学教科书都试图讨论超出(14.5.2)式范围的内容，并给出可以利用 r 的另外的统计检验。然而，在几乎所有的情况下，这些检验仅对一些特殊的假设有效，即 x 和 y 的联合分布形成了一个围绕均值的**双正态**或**二维高斯**分布，联合概率密度为

$$p(x,y)\,dxdy = \text{const.} \times \exp\left[-\frac{1}{2}(a_{00}x^2 - 2a_{01}xy + a_{11}y^2)\right]\,dxdy \tag{14.5.3}$$

其中 a_{00}，a_{01} 和 a_{11} 是任意常数，对这一分布 r 取值为

$$r = -\frac{a_{01}}{\sqrt{a_{00}a_{11}}} \tag{14.5.4}$$

有时(14.5.3)式被看成一个好的数据模型，有时至少被看做粗糙而便利的假设，因为许多二维分布相似于双正态分布，至少在尾端相差不多。在两种情况下，能在以下几个方向超出(14.5.2)式的范围使用(14.5.3)式。

首先，考虑数据点数目 N 不很大时的可能情况，这里统计量

$$t = r\sqrt{\frac{N-2}{1-r^2}} \tag{14.5.5}$$

在无效假设情况(不相关)下的分布与 $\nu = N-2$ 自由度的学生 t 分布相似，学生 t 分布的双边显著性水平由 $1 - A(t|\nu)$((6.4.7)式)给出。当 N 增大时，这个显著性将逐次变得与(14.5.2)式相同。因此使用(14.5.5)式将不会很差，即使双正态假设还未被很好地证明。

其次，当 N 仅是中等大小(≥ 10)时，能够比较是否两个显著非零 r 值(例如取自不同的实验)之间的差异本身是显著的。换句话说，能定量地确定，在某一控制变量上的变化是否能显著地改变两个其他变量之间所存在的相关性，做到这点可利用 **Fisher 的 z 变换**将每个测量的 r 与相应的量 z 联系起来，

$$z = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right) \tag{14.5.6}$$

于是，每个 z 都近似为正态分布，其均值为

$$\bar{z} = \frac{1}{2}\left[\ln\left(\frac{1+r_{\text{true}}}{1-r_{\text{true}}}\right) + \frac{r_{\text{true}}}{N-1}\right] \tag{14.5.7}$$

其中 r_{true} 是相关系数的实际值或总体值，并且具有以下标准差：

$$\sigma(z) \approx \frac{1}{\sqrt{N-3}} \tag{14.5.8}$$

当(14.5.7)式和(14.5.8)式有效时，给出了几个有用的统计检验。例如 r 的测量值不同于某个假设值 r_{true} 的显著性水平为

$$\text{erfc}\left(\frac{|z-\bar{z}|\sqrt{N-3}}{\sqrt{2}}\right) \tag{15.5.9}$$

其中 z 和 $\bar{z}$ 由(14.5.6)式和(14.5.7)式给出，并且(14.5.9)式的值较小表示有显著差别(令 $\bar{z} = 0$，可以使(14.5.9)式能更精确地代替(14.5.2)式)。与此相似，两个被测的相关系数 r_1 和 r_2 之间差异的显著性为

$$\text{erfc}\left(\frac{|z_1 - z_2|}{\sqrt{2}\sqrt{\frac{1}{N_1-3} + \frac{1}{N_2-3}}}\right) \tag{14.5.10}$$

其中 z_1 和 z_2 可以利用(14.5.6)式由 r_1 和 r_2 得到，N_1 和 N_2 分别是 r_1 和 r_2 的度量中数据点的数目。

所有上面的显著性都是双边的。如果希望否定无效假设以有利于单边假设，例如 $r_1 > r_2$(其中不等式的估计方向被**先验地**决定了)，那么：(i)如果被测的 r_1 和 r_2 有**错误的**估计方向，则将不能证明所做的单边假设；(ii)但如果它们有正确的次序，则能够用 0.5 乘以上面给出的各种显著性，使它们更加显著。

但请记住：如果 x 和 y 变量的联合概率分布显著不同于双正态分布，则 r 统计量的解释可能完全无意义。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::pearsn(Vec_I_DP &x, Vec_I_DP &y, DP &r, DP &prob, DP &z)
给定两个数组 x[0..n-1]和 y[0..n-1],本程序计算它们的相关系数 r(返回为 r),即零相关的无效假设被否定处的显
著性水平(prob,它的小值表示显著相关),以及 Fisher z(返回 z),它的值可以用于如上所述的进一步的统计检验中。
{
    const DP TINY=1.0e-20;                          将正则化完全相关的异常情况。
    int j;
    DP yt,xt,t,df;
    DP syy=0.0,sxy=0.0,sxx=0.0,ay=0.0,ax=0.0;

    int n=x.size();
    for (j=0;j<n;j++) {                             寻找均值。
        ax += x[j];
        ay += y[j];
    }
    ax /= n;
    ay /= n;
    for (j=0;j<n;j++) {                             计算相关系数。
        xt=x[j]-ax;
        yt=y[j]-ay;
        sxx += xt*xt;
        syy += yt*yt;
        sxy += xt*yt;
    }
    r=sxy/(sqrt(sxx*syy)+TINY);
    z=0.5*log((1.0+r+TINY)/(1.0-r+TINY));           Fisher z 变换。
    df=n-2;
    t=r*sqrt(df/((1.0-r+TINY)*(1.0+r+TINY)));       (14.5.5)式。
    prob=betai(0.5*df,0.5,df/(df+t*t));             学生 t 概率。
    // prob=erfcc(fabs(z*sqrt(n-1.0))/1.4142136);
    对于大 n, 使用例程 erfcc, 它较易计算 prob, 并给出近似相同的值。
}
```

参考文献和深入阅读

Dunn, O.J., and Clark, V.A. 1974, *Applied Statistics: Analysis of Variance and Regression* (New York: Wiley).

Hoel, P.G. 1971, *Introduction to Mathematical Statistics*, 4th ed. (New York: Wiley), Chapter 7.

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapters IX(A) and IX(B).

Korn, G.A., and Korn, T.M. 1968, *Mathematical Handbook for Scientists and Engineers*, 2nd ed. (New York: McGraw-Hill), §19.7.

Norusis, M.J. 1999, *SPSS 9.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall).

14.6 非参数相关或秩相关

正是在线性相关系数 r 的显著性解释中的不确定性,导致了**非参数相关**或**秩相关**这些重要概念。像以前一样,给出 N 对测量值(x_i, y_i)。以前的困难在于我们未必知道从中抽取 x_i 和 y_i 的概率分布函数。

非参数相关的关键概念是:在样本中,如果将每一个 x_i 替换成它在所有其他 x_i 中的秩值,即 1,2,3,...,N,则作为结果的数表将取样于完全已知的分布函数,即均匀地来自从 1 到 N(包括 1 和 N)的整数。事实上,这比均匀更好,因为如果所有 x_i 的值都是不同的,则每个整数都将只出现一次。如果$\{x_i\}$中有相等的,通常赋予这些**结**以秩的均值(如果它们的值稍有不同,就会有此均值)。这个**中间秩**有时是整数,有时是半整数。在所有情况下,所有赋于秩的和都将等于从 1 到 N 所有整数的和,即$\frac{1}{2}N(N+1)$。

当然可以对$\{y_i\}$进行同样的处理,即用每个值在样本中其他 y_j 处的秩代替每个 y_i。

现在,可以自由地选择监测均匀整数组 1 和 N 之间相关性的统计量,应注意秩中结的可能性。当然在用秩代替原始值时会丢失一些信息。能构造一些相当人为的例子,其相关性能够参数化地(如以线性相关系数

r)检测，但并不能非参数化地检测。然而，这样的例子在实际生活中非常少见，并且在秩化过程中，微小的信息损失是以很小的代价换取很高的收益：当一相关性被证明是非参数的，那么它确实如此(即一定程度上取决于选择的显著性)。非参数相关比线性相关更稳健，也更宜于消除数据中未知的缺陷，就像中位数比均值更稳健一样。关于稳健性的更详细讨论见 15.7 节。

在统计学中总是有些已构造好的统计量为我们所用，下面将讨论两个统计量，**Spearman 秩阶相关系数**(r_s)和 **Kendall 的τ**(*tau*)。

14.6.1　Spearman 秩阶相关系数

设 R_i 为$\{x_i\}$中 x_i 的秩，S_i 为$\{y_i\}$中 y_i 的秩，如上所述，结被赋予适当的中间秩，则秩阶相关系数被定义为秩的线性相关系数，即

$$r_s = \frac{\sum_i (R_i - \overline{R})(S_i - \overline{S})}{\sqrt{\sum_i (R_i - \overline{R})^2}\sqrt{\sum_i (S_i - \overline{S})^2}} \tag{14.6.1}$$

检验 r_s 的非零值的显著性由下式计算：

$$t = r_s\sqrt{\frac{N-2}{1-r_s^2}} \tag{14.6.2}$$

其近似为 $N-2$ 自由度的学生 t 分布。关键是这一近似并不取决于 x 和 y 的原始分布，它始终是同样的近似，并且总是相当好。

可以证明，r_s 与另一种常用的非参数相关的测度密切相关，即**秩差的平方和**：

$$D = \sum_{i=0}^{N-1}(R_i - S_i)^2 \tag{14.6.3}$$

(D 有时用 D^* 表示，其中 * 号表明结是用中间秩处理的。)

当数据中没有结时，则 D 和 r_s 的精确关系为

$$r_s = 1 - \frac{6D}{N^3 - N} \tag{14.6.4}$$

当有结时，则精确关系稍微复杂一些：设 f_k 为$\{R_i\}$的第 k 个结组中结的数目，g_m 为$\{S_i\}$的第 m 个结组中结的数目，则得出下式精确地成立：

$$r_s = \frac{1 - \dfrac{6}{N^3 - N}\left[D + \frac{1}{12}\sum_k (f_k^3 - f_k) + \frac{1}{12}\sum_m (g_m^3 - g_m)\right]}{\left[1 - \dfrac{\sum_k (f_k^3 - f_k)}{N^3 - N}\right]^{1/2}\left[1 - \dfrac{\sum_m (g_m^3 - g_m)}{N^3 - N}\right]^{1/2}} \tag{14.6.5}$$

注意，如果所有 f_k 和g_m 都等于 1(即没有结)，则(14.6.5)式约化为(14.6.4)式。

在(14.6.2)式中给出了检验非零 r_s 显著性的 t 统计量。而用它直接检验 D 的显著性也是可行的，在无效假设(非相关数据组)中 D 的期望值为

$$\overline{D} = \frac{1}{6}(N^3 - N) - \frac{1}{12}\sum_k (f_k^3 - f_k) - \frac{1}{12}\sum_m (g_m^3 - g_m) \tag{14.6.6}$$

其方差为

$$\begin{aligned}\mathrm{Var}(D) = {} & \frac{(N-1)N^2(N+1)^2}{36} \\ & \times \left[1 - \frac{\sum_k (f_k^3 - f_k)}{N^3 - N}\right]\left[1 - \frac{\sum_m (g_m^3 - g_m)}{N^3 - N}\right]\end{aligned} \tag{14.6.7}$$

并且，它近似服从正态分布，因此显著性水平为补余误差函数(参见(14.5.2)式)。当然，(14.6.2)式和(14.6.7)式都不独立，而是同一检验的简单变形。在下面的程序中将同时计算由(14.6.2)式和(14.6.7)式给出的显著性水平，它们的差异将给出这样的概念：近似好到什么程度。还应注意到，将指派秩的工作(包括结的中间秩)分出来形成一个单独的函数由程序 crank 完成。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::spear(Vec_I_DP &data1, Vec_I_DP &data2, DP &d, DP &zd, DP &probd,
    DP &rs, DP &probrs)
```

给定两个数据数组 data1[0..n-1]和 data2[0..n-1],程序将它们的秩差之平方和返回到 D,D 偏差无效假设期望值的标准差数目为 zd,这种偏差的双边显著性水平为 probd.Spearman 秩相关 r, 返回为 rs,它偏离零双边显著性水平为 probrs。外部程序 crank(下面)和 sort2(参见8.2 节)被使用。无论小的 probd 值或小的 probrs 值都表示一个显著地相关(rs 为正)或反相关(rs 为负)。

```
{
    int j;
    DP vard,t,sg,sf,fac,en3n,en,df,aved;

    int n=data1.size();
    Vec_DP wksp1(n),wksp2(n);
    for (j=0;j<n;j++) {
        wksp1[j]=data1[j];
        wksp2[j]=data2[j];
    }
    sort2(wksp1,wksp2);          对每一数据数组排序，并将表项转换成秩，sf 和 sg 的值
    crank(wksp1,sf);             分别返回 ∑(f_k^3 - f_k) 和 ∑(g_m^3 - g_m) 的值。
    sort2(wksp2,wksp1);
    crank(wksp2,sg);
    d=0.0;
    for (j=0;j<n;j++)            对秩差的平方求和。
        d += SQR(wksp1[j]-wksp2[j]);
    en=n;
    en3n=en*en*en-en;
    aved=en3n/6.0-(sf+sg)/12.0;                          D 的期望值。
    fac=(1.0-sf/en3n)*(1.0-sg/en3n);
    vard=((en-1.0)*en*en*SQR(en+1.0)/36.0)*fac;          D 的方差给出标准差的个数
    zd=(d-aved)/sqrt(vard);                              和显著性。
    probd=erfcc(fabs(zd)/1.4142136);
    rs=(1.0-(6.0/en3n)*(d+(sf+sg)/12.0))/sqrt(fac);      秩相关系数。
    fac=(rs+1.0)*(1.0-rs);
    if (fac > 0.0) {
        t=rs*sqrt((en-2.0)/fac);                         和它的 t 值。
        df=en-2.0;
        probrs=betai(0.5*df,0.5,df/(df+t*t));            给出它的显著性。
    } else
        probrs=0.0;
}
```

```
#include "nr.h"

void NR::crank(Vec_IO_DP &w, DP &s)
```

给定一个已排序的数组 w[0..n-1],用它们的秩,包括结的中间秩来代替元素,并且将 f^3-f 的和式返回到 s,其中 f 是每个结中元素的个数。

```
{
    int j=1,ji,jt;
    DP t,rank;

    int n=w.size();
    s=0.0;
    while (j < n) {
        if (w[j] != w[j-1]) {            不是结。
            w[j-1]=j;
            ++j;
        } else {                         有结。
            for (jt=j+1;jt<=n && w[jt-1]==w[j-1];jt++);     它走了多远?
            rank=0.5*(j+jt-1);           这是结的平均秩。
            for (ji=j;ji<=(jt-1);ji++)   将此平均秩送入到所有的结表项中,
                w[ji-1]=rank;
            t=jt-j;
            s += (t*t*t-t);              并修正 s。
            j=jt;
        }
    }
    if (j == n) w[n-1]=n;                如果最后元素不是结，这就是它的秩。
}
```

14.6.2 Kendall 的 τ

Kendall 的 τ 甚至比 Spearman 的 r_s 或 D 更非参数化。它不用秩的数值差，而只用秩的相对次序关系：较高秩、较低秩或同秩。在这种情况下，甚至不必求秩，仅需通过值的大、小或相等就可分别确定秩的较高、较低或相等。总的来说，更喜欢用 r_s，因为它是直接的非参数检验，但是这两个统计量都很常用。事实上，τ 和 r_s 有很强的关联，并且在多数应用中是相同的检验。

为了定义 τ，以 N 个数据点 (x_i, y_i) 作为讨论的起点。现在考虑所有 $\frac{1}{2}N(N-1)$ 个数据点对，其中数据点自身不能成对，但按任一种顺序计数的点都可作为一对。如果两个 x 秩（或两个 x 自身）的相对次序关系与两个 y 秩或两个 y 自身的相对次序关系相同，则称该对是**和谐的**，如果次序关系相反，则称为是**不和谐的**。如果在两个 x 秩的任一个中或两个 y 秩的任一个中有一个结，则既不能把点对称为和谐的，也不能称为**不和谐的**，如果结在 x 中，则称该点对为"附加的 y 对"；如果结在 y 中，则称该点对为"附加的 x 对"。如果 x 和 y 同时有结，则根本不能对该点对下定义。

Kendall 的 τ 是以下这些量的简单组合：

$$\tau = \frac{\text{和谐量} - \text{不和谐量}}{\sqrt{\text{和谐量} + \text{不和谐量} + \text{附加}y\text{的量}}\ \sqrt{\text{和谐量} + \text{不和谐量} + \text{附加}x\text{的量}}} \tag{14.6.8}$$

显然，τ 在 -1 和 $+1$ 之间，并且仅在秩完全一致或完全相反时才能取得极值。

更重要的是，Kendall 已经由组合算出，在 x 和 y 之间没有联系的无效假设下 τ 的近似分布。在这种情况下，τ 近似为正态分布，期望值为 0，方差为

$$\mathrm{Var}(\tau) = \frac{4N+10}{9N(N-1)} \tag{14.6.9}$$

下面的程序实现了以上描述，并因此在所有数据点对上做循环。注意：这是一个 $O(N^2)$ 算法，它不像 r_s 算法，r_s 算法主要的排序运算是 $N\log N$ 阶。如果按常规对超过几千个点的数据组计算 Kendall 的 τ 值，则将非常麻烦。然而，如果愿意将数据合并成中等数目的离散值，则可用下述程序：

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::kendll(Vec_I_DP &data1, Vec_I_DP &data2, DP &tau, DP &z, DP &prob)
给定数组 data1[0..n-1]和 data2[0..n-1]，本程序将 Kendall 的 τ 返回为 tau，将它的偏离零点标准差数目返回到 z，并将
它的双边显著性水平返回到 prob。小的 prob 值表示显著地相关（tau 为正）或反相关（tau 为负）。
{
    int is=0,j,k,n2=0,n1=0;
    DP svar,aa,a2,a1;

    int n=data1.size();
    for (j=0;j<n-1;j++) {                          对点对的第一个元素循环，
        for (k=j+1;k<n;k++) {                      对第二个元素循环。
            a1=data1[j]-data1[k];
            a2=data2[j]-data2[k];
            aa=a1*a2;
            if (aa != 0.0) {                       两个数组都没有结。
                ++n1;
                ++n2;
                aa > 0.0 ? ++is : --is;
            } else {                               一个或两个数组有结。
                if (a1 != 0.0) ++n1;               一个"附加的x"事件。
                if (a2 != 0.0) ++n2;               一个"附加的y"事件
            }
        }
    }
    tau=is/(sqrt(DP(n1))*sqrt(DP(n2)));            (14.6.8)式。
    svar=(4.0*n+10.0)/(9.0*n*(n-1.0));             (14.6.9)式。
    z=tau/sqrt(svar);
    prob=erfcc(fabs(z)/1.4142136);                 显著性。
}
```

有时,恰巧对每个 x 和 y 只有几个可能的值。在这种情况下,数据能被记录成列联表(参见 14.4 节),该表对每一 x 和 y 的列联给出了数据点数目。

Spearman 的秩阶相关系数在这些情况下并不是非常自然的统计量,因为它赋予每个 x 和 y 量化级并不非常有意义的中间秩值,这导致了大量的等秩差。而另一方面,Kendall τ 由于计算简单,可以相当合乎实际。而且,它的 $O(N^2)$ 算法再也不是问题,因为可以安排它在列联表的项目对(每个包含许多数据点)上进行循环,而不是对数据点对进行循环。这点将在下面的程序中加以改进。

注意,Kendall τ 只能用于两变量都是有序的(即良序的)列联表,并且它只寻求单调相关,而不是任意的关联。这两点使它与 14.4 节中的方法相比减少了普遍性,14.4 节中的方法可应用于公称(即无序的)变量以及任意关联。

比较上面的 kendl1 和下面的 kendl2,将看到已将许多变量从整数型变成了浮点型。这是因为在列联表中事件的数目可能会相当大,以致在一些整型算术中会出现溢出,而在列表中单个数据点的数目都不可能那么大(对一个 $O(N^2)$ 程序而言)。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::kendl2(Mat_I_DP &tab, DP &tau, DP &z, DP &prob)
给定一个二维表 tab[0..i-1][0..j-1],使 tab[k][l]包含一个变量落入量化级 k,第二个变量落入量化级 l 的事件的个
数。本程序将 Kendall τ 返回到 tau,将它偏离零点的标准差数目返回到 z,并将它的双边显著水平返回到 prob。小的 prob
值表示两变量之间显著相关(tau 为正)或反相关(tau 为负)。虽然 tab 是浮点型数组,但通常它是整数值。
{
    int k,l,nn,mm,m2,m1,lj,li,kj,ki;
    DP svar,s=0.0,points,pairs,en2=0.0,en1=0.0;

    int i=tab.nrows();
    int j=tab.ncols();
    nn=i*j;                          列联表中表项的总数。
    points=tab[i-1][j-1];
    for (k=0;k<=nn-2;k++) {          对表中表项进行循环。
        ki=(k/j);                    对一行和一列进行译码。
        kj=k-j*ki;
        points += tab[ki][kj];       增加事件的总数。
        for (l=k+1;l<=nn-1;l++) {    对点对的其它成员进行循环。
            li=l/j;                  对它的行和列进行译码。
            lj=l-j*li;
            mm=(m1=li-ki)*(m2=lj-kj);
            pairs=tab[ki][kj]*tab[li][lj];
            if (mm != 0) {           不是相持。
                en1 += pairs;
                en2 += pairs;
                s += (mm > 0 ? pairs : -pairs);      和谐的或不合谐的。
            } else {
                if (m1 != 0) en1 += pairs;
                if (m2 != 0) en2 += pairs;
            }
        }
    }
    tau=s/sqrt(en1*en2);
    svar=(4.0*points+10.0)/(9.0*points*(points-1.0));
    z=tau/sqrt(svar);
    prob=erfcc(fabs(z)/1.4142136);
}
```

参考文献和深入阅读

Lehmann, E.L. 1975, *Nonparametrics: Statistical Methods Based on Ranks* (San Francisco: Holden-Day).

Downie, N.M., and Heath, R.W. 1965, *Basic Statistical Methods*, 2nd ed. (New York: Harper & Row), pp. 206–209.

Norusis, M.J. 1999, *SPSS 9.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall).

14.7　深入讨论:二维分布

下面讨论把 K－S 检验(参见 14.3 节)推广到二维分布。这一推广的过程应归功于 Fasano 和 Franceschini[1],实际上它是 Peacock[2]早期概念的变形。

在二维分布中,每个数据点都用一个值对(x,y)来表征。一个熟悉的例子是,所探测到的来自 Supernova 1987A 的 19 个中微子,它们每一个都由时间 t_i 和能量 E_i 来表征(见本节参考文献[3])。我们希望知道这些测量的数据对(t_i,E_i),$i=0,\ldots,18$ 是否与理论模型一致,该理论模型预期中微子流是时间和能量的函数,即(x,y)平面(这里是(t,E)平面)的二维概率分布。这是单样本检验。如果给出来自两个可比较的观测仪的两组中微子探测结果,我们希望知道这两者是否互相一致,这即是双样本检验。

希望将一维 K－S 检验引入(x,y)平面,以寻求两个二维分布之间的某种最大积累差。遗憾的是,在超过一维的情况下,不容易定义积累的概率分布。Peacock 的贡献是找到了一种好的替代物,即在给定点(x_i,y_i)周围**四个自然象限每一个象限中的综合概率**,亦即在$(x>x_i,y>y_i)$,$(x<x_i,y>y_i)$,$(x<x_i,y<y_i)$,$(x>x_i,y<y_i)$中的总概率(或数据的分数)。二维 K－S 统计量 D 现在可看成相应于综合概率的最大差异(它涉及数据点和象限两个范围)。当比较两个数据集时,D 值取决于数据集变化的范围。在这种情况下,定义一个有效的 D 作为所得到两个值的平均。如果对 D 的定义感到困惑,可参考相应的计算机程序提供的精确算法定义。

图 14.7.1 给出了一个例子,在平面中 65 个三角形和 35 个正方形似乎具有不同的分布。点线的中心位于使统计量 D 最大的三角形上,最大值出现在左上象限。在这个象限中只包含了所有三角形的 0.12,但包含了所有正方形的 0.56,所以 D 值为 0.44,这是否在统计上显著不同呢?

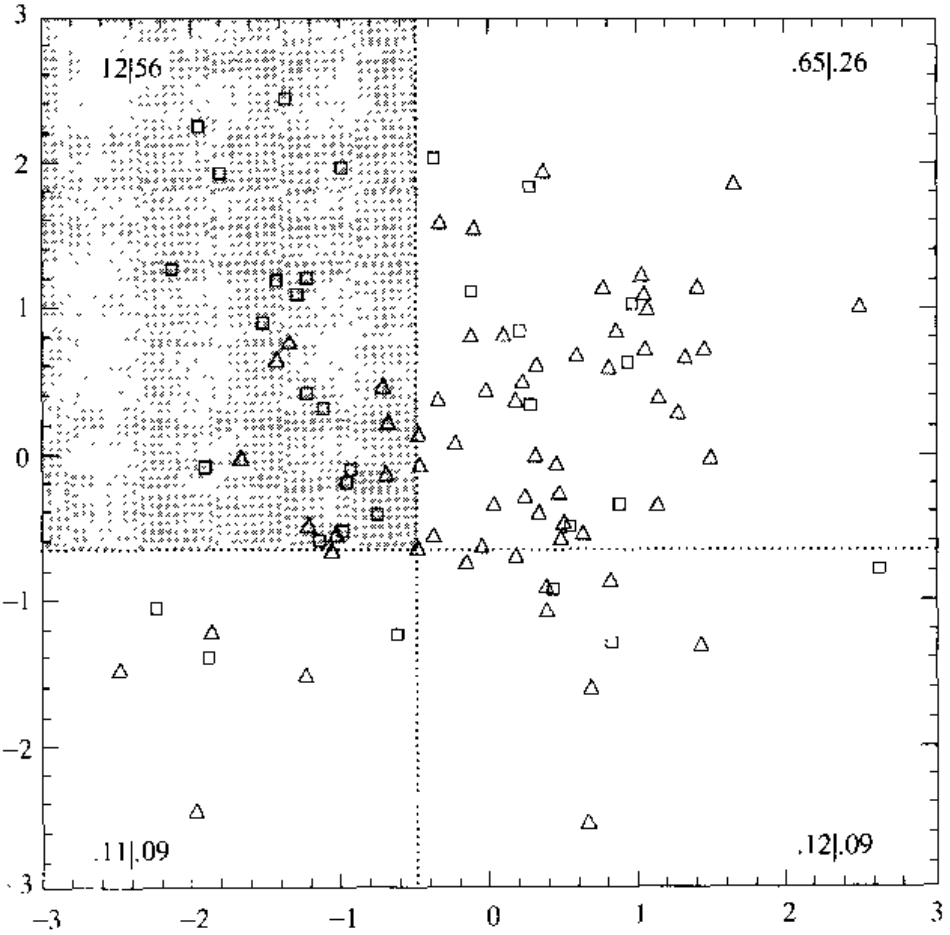


图 14.7.1　65 个三角形和 35 个正方形的二维分布　二维 K－S 检验找到了这样的点,其象限(点线所示)之一内三角形分数和正方形分数之间的差异最大。那么(14.7.1)式指出了这一差异是否在统计上是显著的,即三角形和正方形是否有不同的基本分布

遗憾的是,即使对固定的样本大小,下面陈述也不是严格正确的:在无效假设下,D 的分布独立于二维分布的形状。在这方面,二维 K-S 检验并不像一维那样实用。然而,广义的蒙特卡罗积分已表明,即使对非常不同的分布,二维 D 的分布也几乎完全相同,只要它们有相同的关系系数 r(通常是以(14.5.1)式定义的)。Fasano 和 Franceschini 在它们的文章中列出了 D 分布作为 D(这是理所当然的)、样本大小 N 和相关系数 r 的函数的蒙特卡罗结果。分析这些结果,可发现二维 K-S 检验的显著性水平可由下面一个简单但又近似的公式给出。

对单样本检验,

$$\text{概率}\,(D > \text{观察值}) = Q_{KS}\left(\frac{\sqrt{N}\,D}{1+\sqrt{1-r^2}(0.25-0.75/\sqrt{N})}\right) \tag{14.7.1}$$

对双样本检验,上面公式仍适用,只是有

$$N = \frac{N_1 N_2}{N_1 + N_2} \tag{14.7.2}$$

当 $N \gtrsim 20$ 且所指示的概率(显著性水平)小于(更加显著于)0.20 左右时,上面公式是足够精确的。如果所指示的概率大于 0.20,则其值不再精确,但也意味着数据和模型(或两数据集)并不显著地不同,这一点是正确的。注意,当 $r \to 1$(完全相关)时,(14.7.1)式和(14.7.2)式简化成了(14.3.9)式和(14.3.10)式:二维数据全位于直线上,并且二维 K-S 检验变成一维 K-S 检验。

顺便说一下,图 14.7.1 的显著性水平是 0.001 左右。这就近乎明确地说明三角形和正方形取自不同的分布(实际上也是如此)。

当然,如果不想依靠体现在(14.7.1)式中的蒙特卡罗实验,那么可以寻求自己的方式:从模型中造出许多假想数据集,每个集中的数据点都与真实数据集相同。对每一假想集,利用相应计算程序计算 D(但忽略它们计算出来的概率),数出这些假想 D 超过真实 D(来自实际数据组)的次数的百分比,这个分数就是所需的显著性。

与一维检验比较,二维检验的缺点是其需要 N^2 次运算,两个 N 阶嵌套循环占据了 $N\log N$ 次操作。对小型计算机来说,这使得检验只限于 N 小于几千的范围内才有用。

现在给出计算机执行程序。一个样本的情况包含在程序 ks2d1s(即二维,1 个样本)中。这个程序调用一个直接可用的程序 quadct 来计算四个象限中的点,并且调用了一个由用户提供的程序 quadvl,这个用户提供的程序必须能够返回任意一点(x,y)附近四个象限中的每一个象限的综合概率。下面也给出一个简单的示例 quadvl,而实际应用的 quadvl 是相当复杂的,并经常伴随解析二维分布的数值积分。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::ks2d1s(Vec_I_DP &x1, Vec_I_DP &y1, void quadvl(const DP, const DP,
    DP &, DP &, DP &, DP &), DP &d1, DP &prob)
一个样本对于一个模型的二维 K-S 检验。在数组 x1[0..n1-1]和 y1[0..n1-1]中给定 n1 个数据点的 x 和 y 的坐标,
以及给定一个用户提供的并作为模型例子的函数 quadvl,本程序返回二维 K-S 统计到 d1,它的显著性水平返回到 prob。
小的 prob 值表明样本显著地不同于模型。注意这种检验直接依赖于分布,所以 prob 只是一个估算值。
{
    int j;
    DP dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,rr,sqen;

    int n1=x1.size();
    d1=0.0;
    for (j=0;j<n1;j++) {                    对所有数据点循环。
        quadct(x1[j],y1[j],x1,y1,fa,fb,fc,fd);
        quadvl(x1[j],y1[j],ga,gb,gc,gd);
        d1=MAX(d1,fabs(fa-ga));
        d1=MAX(d1,fabs(fb-gb));
        d1=MAX(d1,fabs(fc-gc));
        d1=MAX(d1,fabs(fd-gd));
        对样本和模型,累积每一个象限中的分布,并且存储最大差。
    }
```

```
    pearsn(x1,y1,r1,dum,dumm);              求得线性相关系数 r1。
    sqen=sqrt(DP(n1));
    rr=sqrt(1.0-r1*r1);
    用 K-S 概率函数 probks 估算概率值。
    prob=probks(d1*sqen/(1.0+rr*(0.25-0.75/sqen)));
}
```

```
#include "nr.h"

void NR::quadct(const DP x, const DP y, Vec_I_DP &xx, Vec_I_DP &yy, DP &fa,
    DP &fb, DP &fc, DP &fd)
```

给定原点(x,y)，并给定 nn 个点的数组，其坐标为 xx[0..nn-1]和 yy[0..nn-1]，计算围绕原点的每个象限中有多少数据点，并将归一化的分数返回。每个象限用英文字母为下标标记，从右上角逆时针方向计算起。此程序用于 ks2d1s 和 ks2d2s 中。

```
{
    int k,na,nb,nc,nd;
    DP ff;

    int nn=xx.size();
    na=nb=nc=nd=0;
    for (k=0;k<nn;k++) {
        if (yy[k] > y)
            xx[k] > x ? ++na : ++nb;
        else
            xx[k] > x ? ++nd : ++nc;
    }
    ff=1.0/nn;
    fa=ff*na;
    fb=ff*nb;
    fc=ff*nc;
    fd=ff*nd;
}
```

```
#include "nr.h"

void NR::quadvl(const DP x, const DP y, DP &fa, DP &fb, DP &fc, DP &fd)
```

这是被用于 ks2d1s 中用户所提供的程序样本。在这种情况下，模型的分布在 $-1<x<1$，$-1<y<1$ 区间以内是均匀的。一般情况下，本程序应该对每点(x,y)返回围绕此点的四个象限之中每一个象限的总体分布的分数，总体分布的分数 fa，fb，fc 和 fd 相加必须为 1。每个象限用英文字母为下标标记，从右上角逆时针方向计算。

```
{
    DP qa,qb,qc,qd;

    qa=MIN(2.0,MAX(0.0,1.0-x));
    qb=MIN(2.0,MAX(0.0,1.0-y));
    qc=MIN(2.0,MAX(0.0,x+1.0));
    qd=MIN(2.0,MAX(0.0,y+1.0));
    fa=0.25*qa*qb;
    fb=0.25*qb*qc;
    fc=0.25*qc*qd;
    fd=0.25*qd*qa;
}
```

程序 ks2d2s 是两个样本的二维 K－S 检验。它也调用 quadct，pearsn 和 probks。进行两个样本检验时，不需要解析的模型。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::ks2d2s(Vec_I_DP &x1, Vec_I_DP &y1, Vec_I_DP &x2, Vec_I_DP &y2, DP &d,
    DP &prob)
```

两个样本的二维 K－S 检验。给定数组 x1[0..n1-1]和 y1[0..n1-1]为第一个样本 n1 个值的 x 和 y 坐标。同样，数组 x2 和 y2 为第二个样本 n2 个值的 x 和 y 坐标。本程序将二维，两个样本的 K－S 统计量返回到 d，它的显著性水平到 prob。小的 prob 值表明这两个样本显著不同。注意，这种检验直接依赖于分布，所以 prob 只是一种估算值。

```
{
    int j;
    DP d1,d2,dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,r2,rr,sqen;

    int n1=x1.size();
    int n2=x2.size();
    d1=0.0;
    for (j=0;j<n1;j++) {                    首先，用第一样本中的点作为原点。
        quadct(x1[j],y1[j],x1,y1,fa,fb,fc,fd);
        quadct(x1[j],y1[j],x2,y2,ga,gb,gc,gd);
        d1=MAX(d1,fabs(fa-ga));
        d1=MAX(d1,fabs(fb-gb));
        d1=MAX(d1,fabs(fc-gc));
        d1=MAX(d1,fabs(fd-gd));
    }
    d2=0.0;
    for (j=0;j<n2;j++) {                    其次，用第二个样本中的点作为原点。
        quadct(x2[j],y2[j],x1,y1,fa,fb,fc,fd);
        quadct(x2[j],y2[j],x2,y2,ga,gb,gc,gd);
        d2=MAX(d2,fabs(fa-ga));
        d2=MAX(d2,fabs(fb-gb));
        d2=MAX(d2,fabs(fc-gc));
        d2=MAX(d2,fabs(fd-gd));
    }
    d=0.5*(d1+d2);                          求 K-S 统计量的平均值。
    sqen=sqrt(n1*n2/DP(n1+n2));
    pearsn(x1,y1,r1,dum,dumm);              对每个样本求线性的关系数。
    pearsn(x2,y2,r2,dum,dumm);
    rr=sqrt(1.0-0.5*(r1*r1+r2*r2));
    用 K-S 概率函数 probks 估算概率。
    prob=probks(d*sqen/(1.0+rr*(0.25-0.75/sqen)));
}
```

参考文献和深入阅读

Fasano, G. and Franceschini, A. 1987, *Monthly Notices of the Royal Astronomical Society*, vol. 225, pp. 155–170. [1]

Peacock, J.A. 1983, *Monthly Notices of the Royal Astronomical Society*, vol. 202, pp. 615–627. [2]

Spergel, D.N., Piran, T., Loeb, A., Goodman, J., and Bahcall, J.N. 1987, *Science*, vol. 237, pp. 1471–1473. [3]

14.8 深入讨论:Savitzky-Golay 平滑滤波器

在 13.5 节已对数字滤波器的结构和应用有了初步的了解,但对一些**特殊用处**的滤波器知之甚少。当然这取决于要用滤波器完成什么任务。对**低通滤波器**而言,其明显的用途是平滑噪声数据。

数据平滑的前提是,正在测量的变量是慢变化的并受到随机噪声的干扰。有时用每一数据点周围数据点的某种局部平均代替该点的值是非常有用的。这是由于周围点值与该点的值差别不大,平均只会减少噪声而不会给该点值带来(太大的)偏差。

必须指出,数据的平滑是针对暗区的,因而已超出了某些更值得提出和推荐的技术的范围,这些技术在本书其他部分讨论。例如,如果正要拟合数据为一个参数模型(参见第 15 章),利用原始数据总是比利用已被预先平滑处理过的数据好。另一种很优秀的技术被称为"最优化"或维纳(Wiener)滤波(参见 13.3 节和 13.6 节)。数据平滑最适合简单应用于图形技术,能消除所有带有较大误差障碍的数据点;或是一种能从图形中做出初步而又**粗糙**的简单参数估算的工具。

在本节将讨论一种特殊的低通滤波器,它适合于数字平滑,被称为 **Savitzky-Golay**[1](后称为 S-G 滤波器)、**最小二乘方**[2]或 **DISPO**(数字平滑多项式)[3]滤波器。不像通常那样先在频域中定义特性,然后转换到时间域,S-G 滤波器是直接来自时间域内数据平滑问题的一种特殊公式。S-G 滤波器起初(现在仍然经常地)用于提取可见光在噪声谱数据中谱线的相对宽度和高度。

让我们回顾一下,数字滤波器是应用于一列等间隔的数据值 $f_i = f(t_i)$。这里 $t_i = t_0 + i\Delta$,其中 Δ 为常数

样本间隔，$i=\ldots,-2,-1,0,1,2,\ldots$。从 13.5 节我们看到最简单的数字滤波器(非递归或有限脉冲响应滤波器)是将每一个数据值 f_i 用其本身和某些邻近值的线性组合 g_i 来代替：

$$g_i=\sum_{n=-n_L}^{n_R}c_n f_{i+n} \tag{14.8.1}$$

其中 n_L 是数据点 i 的左边所用到的点数(在它之前)，n_R 则是右边(即在它之后)所用到的点数。称之为**因果滤波器**的 n_R 应等于零。

在讨论 S－G 滤波器时，首先考虑最简单的平均过程：对固定的 $n_L=n_R$，计算每一个 g_i(作为 f_{i-n_L} 到 f_{i+n_R} 数据点的平均)。有时称它为**活动窗口平均**，它相当于 $c_n=1/(n_L+n_R+1)$ 的(14.8.1)式。如果基础函数是常数或随时间线性变化(增加或减少)，那么结果中不会引入任何偏差。一般来说，在平均间隔一端的较高点被另一端的较低点所平衡。然而，如果基础函数有二阶非零导数，则偏差被引入。例如在局部极大值处，活动窗口平均总是减少了此处函数值。在光谱测定的应用中，就是使一束窄谱线的高度减少，宽度增加。因为这些参量本身有物理意义，所以不希望引入偏差。

应注意，活动窗口平均保留了谱线下的区域，即它的零阶矩，(如果窗口是对称的，有 $n_L=n_R$)那么还保留了它在时间上的平均位置，即它的一阶矩。受到扰动的是二阶矩，等价于线宽。

S－G 滤波的设计思想是能寻找到合适的滤波系数 c_n 以保护高阶矩。也就是说，在活动窗口内对基础函数进行近似时，不是用常数窗口(它的估算是均值)，而是用高阶多项式(典型的是二次、四次)：对每个点 f_i，利用最小二乘方拟合，将此活动窗口的所有 n_L+n_R+1 个点拟合成一个多项式，然后令 g_i 为多项式在 i 点的值(如果不熟悉最小二乘方拟合，可以先看一下第 15 章)，并不使用在其他点处多项式的值。当窗口移动到下一点 f_{i+1} 时，将对新窗口进行新的最小二乘方拟合。

如果照此做下去，显然所有最小二乘方拟合是相当复杂的。幸运的是，由于最小二乘方拟合过程仅涉及一个线性矩阵求逆，则拟合多项式系数本身在数据值上是线性的。这意味着可以事先对一组假设数据(除一个为 1 外其他都取零)做全部的拟合，然后仅通过线性组合来拟合真实数据。这里的关键是：对这组系数而言，存在一组特殊的滤波系数 c_n，(14.8.1)式能"自动"完成在活动窗口内的最小二乘方拟合过程。

为了导出这样的系数，让我们考虑 g_0 是怎样得到的。我们希望在 i 处对值 $f_{-n_L},\ldots,f_{n_R}$ 拟合一个 M 阶多项式，即 $a_0+a_1 i+\cdots+a_M i^M$，那么 g_0 即为 $i=0$ 时多项式的值，即 $g_0=a_0$，对这一问题(参见 15.4 节)的设计矩阵为

$$A_{ij}=i^j \qquad i=-n_L,\ldots,n_R,\quad j=0,\ldots,M \tag{14.8.2}$$

向量 $\{a_i\}$ 以向量 $\{f_i\}$ 表示的正规方程以矩阵形式给出：

$$(\mathbf{A}^T\cdot\mathbf{A})\cdot\mathbf{a}=\mathbf{A}^T\cdot\mathbf{f} \qquad 或 \qquad \mathbf{a}=(\mathbf{A}^T\cdot\mathbf{A})^{-1}\cdot(\mathbf{A}^T\cdot\mathbf{f}) \tag{14.8.3}$$

也可得到具体形式：

$$\left\{\mathbf{A}^T\cdot\mathbf{A}\right\}_{ij}=\sum_{k=-n_L}^{n_R}A_{ki}A_{kj}=\sum_{k=-n_L}^{n_R}k^{i+j} \tag{14.8.4}$$

和

$$\left\{\mathbf{A}^T\cdot\mathbf{f}\right\}_j=\sum_{k=-n_L}^{n_R}A_{kj}f_k=\sum_{k=-n_L}^{n_R}k^j f_k \tag{14.8.5}$$

因为当 $\mathbf{f}$ 用单位向量 $\mathbf{e}_n(-n_L\leqslant n<n_R)$ 代换时系数 c_n 就是相应的 a_0，则有

$$c_n=\left\{(\mathbf{A}^T\cdot\mathbf{A})^{-1}\cdot(\mathbf{A}^T\cdot\mathbf{e}_n)\right\}_0=\sum_{m=0}^{M}\left\{(\mathbf{A}^T\cdot\mathbf{A})^{-1}\right\}_{0m}n^m \tag{14.8.6}$$

注意，(14.8.6)式表明仅需知道逆矩阵的一行值(数值计算上讲，仅用单一的后代换的 LU 分解就可以求得这一值)。

下面的函数 savgol 实现了(14.8.6)式。作为输入，它取参数 nl = n_L，nr = n_R 和 m = M(需要的阶)。输出数组 c 的物理长度 np 也是输入量。为零的数据拟合参量 ld 同样为输入。实际上，ld 指定了 $\{a_i\}$ 中哪一个系数将返回，这里我们仅对 a_0 感兴趣。为了另外的目的，即计算数值微分(在 5.7 节中已提及)，则有用的选择是 ld ≥1

例如,ld = 1 时,滤出的一阶导数是除以步长 Δ 的卷积式(14.8.1)。对求导而言,总是希望 m = 4 或更大些。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::savgol(Vec_O_DP &c, const int np, const int nl, const int nr,
    const int ld, const int m)
在c[0..np-1]中返回一组S-G滤波系数,它是以环绕次序排列,与程序convlv中的变量respns是一致的。nl是所使用的左边
(过去)数据点的个数,而nr是右边(将来)数据点的个数.而所使用的全部数据点个数为nl+nr+1。ld是所求导数的阶(如,
对平滑函数ld=0、对k阶导数,必须将数组c乘以k)、m是平滑多项式的阶,即等于最高的保留矩,通常m=2或m=4。
{
    int j,k,imj,ipj,kk,mm;
    DP d,fac,sum;

    if (np < nl+nr+1 || nl < 0 || nr < 0 || ld > m || nl+nr < m)
        nrerror("bad args in savgol");
    Vec_INT indx(m+1);
    Mat_DP a(m+1,m+1);
    Vec_DP b(m+1);
    for (ipj=0;ipj<=(m << 1);ipj++) {        设置所求最小二乘方拟合的正规方程。
        sum=(ipj ? 0.0 : 1.0);
        for (k=1;k<=nr;k++) sum += pow(DP(k),DP(ipj));
        for (k=1;k<=nl;k++) sum += pow(DP(-k),DP(ipj));
        mm=MIN(ipj,2*m-ipj);
        for (imj = -mm;imj<=mm;imj+=2) a[(ipj+imj)/2][(ipj-imj)/2]=sum;
    }
    ludcmp(a,indx,d);                        求解: LU分解。
    for (j=0;j<m+1;j++) b[j]=0.0;
    b[ld]=1.0;
    右端向量是单位向量, 依赖于所求导数。
    lubksb(a,indx,b);                        求逆矩阵之行。
    for (kk=0;kk<np;kk++) c[kk]=0.0;         零化输出矩阵(可能大于系数的个数)。
    for (k = -nl;k<=nr;k++) {
        sum=b[0];                            每个S-G系数是整数幂与逆矩阵行的点乘。
        fac=1.0;
        for (mm=1;mm<=m;mm++) sum += b[mm]*(fac *= k);
        kk=(np-k) % np;                      存储环绕的顺序。
        c[kk]=sum;
    }
}
```

作为输出,savgol 返回了系数 $c_n(-n_L \leqslant n \leqslant n_R)$,这些都以"环绕顺序"的形式存储在 c 中,即 c_0 在 c[0] 中,c_{-1} 在 c[1] 中(对进一步的负标依此类推)等;c_1 存储在 c[np-1] 中,c_2 在 c[np-2] 中(其余正标依次类推)等。这一顺序看起来挺神秘,但对因果滤波器来说却很自然,它在低的 c 数组元素内有非零系数。这也是 13.1 节中函数 convlv 所要求的顺序,它是对一数据集进行数字滤波时所采用的次序。

下表给出了 savgol 的某些标准输出。对于 2 阶和 4 阶的情况,给出几种 n_L 和 n_R 选择的 S-G 滤波器系数;中间一列是为求得被平滑的 g_i 而应用于数据 f_i 的系数;左边的系数用于以前的数据;右边的系数用于以后的数据。系数之和总是 1(在舍入误差之内)。可以看到,作为平滑算子,系数总是在中心有一个的凸起,但常有小的、远离中心的、既有显示为正的又有显示为负的校正。事实上,S-G 滤波器最适合于非常大的 n_L 和 n_R,因为这些点很少的公式可用相对少量的平滑完成。

M	n_L	n_R	样本 Savitzky-Golay 系数										
2	2	2				-0.086	0.343	0.486	0.343	-0.086			
2	3	1			-0.143	0.171	0.343	0.371	0.257				
2	4	0		0.086	-0.143	-0.086	0.257	0.886					
2	5	5	-0.084	0.021	0.103	0.161	0.196	0.207	0.196	0.161	0.103	0.021	-0.084
4	4	4		0.035	-0.128	0.070	0.315	0.417	0.315	0.070	-0.128	0.035	
4	5	5	0.042	-0.105	-0.023	0.140	0.280	0.333	0.280	0.140	-0.023	-0.105	0.042

图 14.8.1 给出了使用 33 个点平滑滤波器的数值实验，即 $n_L = n_R = 16$。最上图为被测试的函数，有 6 个不同宽度隆起处都是 8 个单位高。对于这个函数，单位方差的高斯白噪声已经被叠加上了（没有被噪声干扰的测试函数由中图和最下图中的点线表示）。隆起部分的宽度（最大值一半处的全宽度即 FWHM）分别是 140，43，24，17，13 和 10。

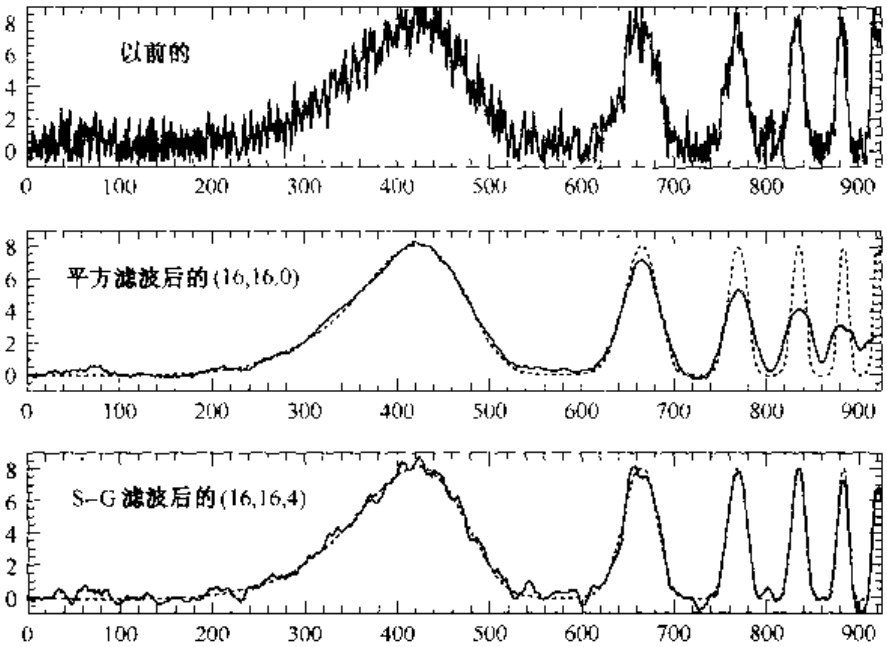


图 14.8.1　最上图：合成噪声数据，它由一系列不断变窄的隆起部分和附加的高斯白噪声组成。中图：用简单的活动窗平均进行数据平滑的结果。这个窗左边延伸16个点，右边延伸16个点，总共33个点。注意，其较窄隆起部分变宽而且相应幅度变低。点线曲线表示用来产生合成数据的基本函数。最下图：用S－G平滑滤波器进行数据平滑的结果（阶为4），采用了同样的33个点。其对较宽的隆起部分的平滑性稍差，但较窄的隆起部分的高度和宽度得到了保护

图 14.8.1 的中图给出了用活动窗平均平滑的结果。可以看出，宽度为 33 的窗口对平滑最宽的隆起部分非常有效，而对较窄的隆起则存在相当大的高度降低和宽度增加。用它描述原基本信号（点线表示）是非常差的。

最下图给出了同样宽度、阶 $M = 4$ 的 S－G 滤波器平滑的结果。可以看出隆起部分的高度和宽度都不寻常地保留住了，惟一的缺陷是最宽的隆起部分欠平滑。这是因为 S－G 滤波器系数主要的正凸处仅是整个 33 点宽度的分数。可以猜想最好的结果应在 4 阶 S－G 滤波器的全宽度是所期望的数据 FWGM 的 1 至 2 倍时得到（参阅本节参考文献[3,4]）。

图 14.8.2 给出用较宽的三个不同阶的 S－G 阶滤波器平滑相同噪声"数据"的结果。其中 $n_L = n_R = 32$（65 个点的滤波器），$M = 2,4,6$。可以看出，当隆起部分相对于滤波器的大小太窄时，则 S－G 滤波器也会在某些点发散。较高阶的滤波器对较窄的隆起有效，但以牺牲较宽隆起部分的平滑性为代价。

总而言之，在某些限制条件下，S－G 滤波器能提供没有分辨率损失的平滑。做到这点是基于假设相对稀疏的数据点具有能用来减少噪声水平的有效冗余度。所设冗余度的特点是基础函数应该被多项式局部地良好拟合。如果这点成立，当平滑线轮廓相对于滤波器宽度不是太窄时，S－G 滤波器的效果相当好。如果这点不成立，这一滤波器与其他平滑滤波系数相比就没有太大优势。

最后要讨论的是不规则取样数据，即 f_i 在时间轴上不是均匀分隔的。这种情况下，显然推广 S－G 滤波必须对每个数据点的活动窗口进行最小二乘方拟合，每个数据点的左右都包括固定的数据点数目（n_L 和 n_R）。然而因为是不规则分隔，没有任何办法得到用于一个点以上的一般性滤波系数。必须对每个点都进行最小二乘方拟合，故当 n_L、n_R 和 M 较大时，将会遇到特别繁重的计算。

作为一个简单的替代方法，可以仍把数据点视为等间隔的（实际上不是）。这相当于在每个活动窗口中，移动数据点到等间隔位置。这样的位移会给函数值引入等效附加噪声源。在这种情况下，平滑是有用的，因为这种噪声远小于已经存在的噪声。特别是，如果在窗内点的位置近似随机，则一个大致的判据为：如果在

$N = n_L + n_R + 1$点窗口的整个宽度内，f 的变化小于单一点的噪声测量值的$\sqrt{N/2}$倍，则这一简单方法可以使用

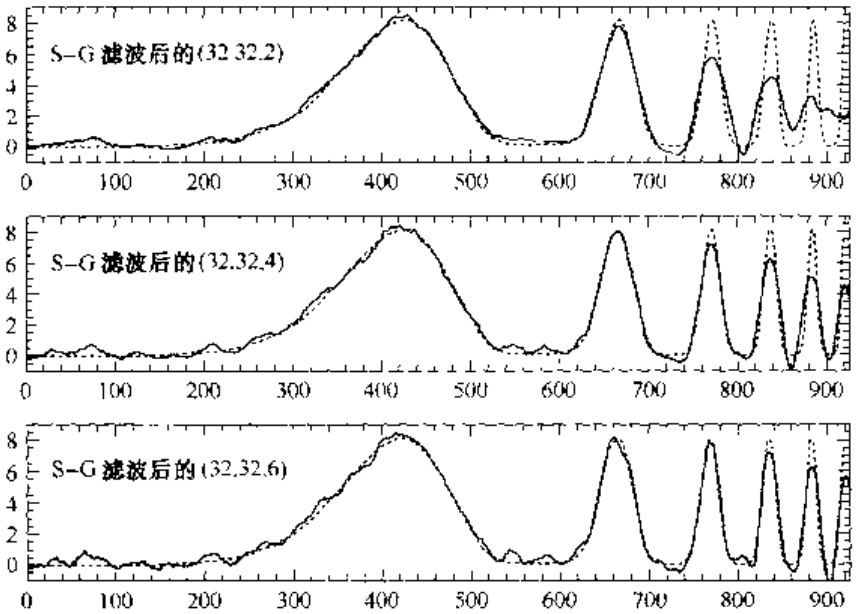


图 14.8.2 对与图 14.8.1 相同的数据组，使用较宽的 65 点 S-G 滤波器的结果。最上图：阶为2；中图：阶为4；最下图：阶为6。对于窄部分的分辨率来说，所有这些滤波器的宽度都不是最优的。高阶滤波器能较好地保留高度和宽度，但对较宽部分的平滑较差

参考文献和深入阅读

Savitzky A., and Golay, M.J.E. 1964, *Analytical Chemistry*, vol. 36, pp. 1627–1639. [1]

Hamming, R.W. 1983, *Digital Filters*, 2nd ed. (Englewood Cliffs, NJ: Prentice-Hall). [2]

Ziegler, H. 1981, *Applied Spectroscopy*, vol. 35, pp. 88–92. [3]

Bromba, M.U.A., and Ziegler, H. 1981, *Analytical Chemistry*, vol. 53, pp. 1583–1586. [4]

第15章 数据建模

15.0 引言

给定一组观察结果，我们经常要对这组数据进行浓缩和概括，使它适合依赖于可调整参数的**模型**。有时这种模型是一些简单方便的函数，例如多项式或高斯函数，从而让拟合的结果提供正确的系数。有时，模型的参数来自于数据假设所满足的基本理论，例如复杂的化学反应系统中的变化率方程的系数，或者一个双星的轨道参数。模型化还可以用做一种约束的插值方法，从中可以延伸出一些数据，形成一个连续函数。但是，对于这些函数的大致形状应有一个基本的概念。

对各种情况，基本方法通常是一样的：选择或设计一个**优值图形函数**（简称为**优值函数**），它必须能度量数据和具有特别参数选择的模型之间一致性的程度。优值函数通常这样设计，即小值代表接近的一致性。接着，对模型的参数进行调节，使优值函数取最小值，产生**最佳拟合参数**。这种调节过程也就是一个多维求极小化的问题。这种优化问题在第10章中讨论过，但是，还有一种特殊的、更有效的方法，具体地说就是建立模型的方法，我们将在本章中讨论。

除了寻找最佳拟合参数外，还有一些重要的问题。数据通常是不准确的，它们存在**测量误差**（在信号处理的文献中称为**噪声**）。如此说来，典型的数据永远不会精确地拟合于正在使用的模型，即便这种模型是正确的。因此，要有估价模型是否恰当的方法，也就是说要有检验**拟合优度**的某些统计标准。

通常还必须知道由数据集确定的参数的准确程度。换言之，要知道最佳拟合参数的似然误差。

最后，我们发现通常优值函数不是单峰的，不具单一的极小值。在有些情况下，可能感兴趣的是全局而不是局部的问题，不是"这种拟合优度如何"的问题，而是"在参数空间的某个区域，如何确信没有比这种拟合更好的了"的问题。正如第10章所述，特别是10.9节，这类问题通常很难解决。

要提醒注意的是，参数拟合不是一个参数估计的终结。一个真正有用的拟合过程必须提供：(i)参数；(ii)参数的误差估计；(iii)拟合优度的统计度量。如果(iii)中的结果表明这种模型不和数据一致，那么(i)和(ii)中的结果也就几乎没有价值。遗憾的是，许多参数估计的实践者永远只进行到第(i)步。如果数据的图表和模型"看起来很好"，他们就认为拟合是可接受的。这种方法被称为**肉眼的 χ 方法**，这种方法是不建议使用的。

参考文献和深入阅读

Bevington, P.R., and Robinson, D.K. 1992, *Data Reduction and Error Analysis for the Physical Sciences*, 2nd ed. (New York: McGraw-Hill).

Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley).

Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press).

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), Chapter X.

Korn, G.A., and Korn, T.M. 1968, *Mathematical Handbook for Scientists and Engineers*, 2nd ed. (New York: McGraw-Hill), Chapters 18–19.

15.1 最大似然估计的最小二乘方法

假定用具有 M 个可调参数 $a_j(j=0,\ldots,M-1)$ 的模型来拟合 N 个数据点 (x_i,y_i)，$i=0,\ldots,N-1$。这种模型给出了已测自变量和因变量之间的一个函数关系：

$$y(x)=y(x;a_0\ldots a_{M-1}) \tag{15.1.1}$$

上式右边明显地标出了对参数的依赖关系。

为了得到 a_j 的拟合值,我们要求对什么求极小化呢?首先想到的是所熟悉的最小二乘方拟合,通过 $a_0,\ldots,a_{M-1}$ 求

$$a_0\ldots a_{M-1}:\qquad \sum_{i=0}^{N-1}[y_i-y(x_i;a_0\ldots a_{M-1})]^2 \tag{15.1.2}$$

的极小值。但这个式子是从哪里来的呢?它以什么基本原理为基础呢?为了回答这些问题,必须进行**最大似然估计**主题的讨论。

给定 x_i 和 y_i 的一组特定的数据集,我们可能有一种直觉,某几组参数值 $a_0\ldots a_{M-1}$ 是非常不可能的——对这些参数,模型函数 $y(x)$ 和数据**相差甚远**,但是另一些参数却非常可能——对于这些参数,模型函数 $y(x)$ 和数据非常相似。可是我们怎样量化这种直觉呢?怎样选择"非常可能"是正确的拟合参数呢?若问"一组拟合的参数 $a_0\ldots a_{M-1}$ 正确的可能性有多大"这样的问题是毫无意义的,原因是不存在模型的统计全域,参数从该模型中抽取。只是存在一个模型、一个正确模型和一组数据集的统计全域,从中可抽取数据集。

实际情况尽管如上所述,但是可以将问题转换一下,问"**给定一组特定的参数**,这组数据集发生的概率多大"。在这样的问题中,如果 y_i 取连续的值,概率永远是零,除非加上这样的语句:"……对于每个数据点加或减某个固定的 Δy"。所以永远让我们将这句话视为不言自明的。如果得到的数据集的概率无限小,则可以认为被考虑的参数"不可能"是正确的。相反地,直觉告诉我们,对于正确选择的参数,数据集也不会不合适。

换言之,用给定参数下数据的概率(在数学上它是可计算的数)作为给定数据下的参数的**似然**。这种借鉴完全基于直觉,它没有任何正式的数学机理。正如我们已经指出的一样,统计**不是**数学的一个分支。

一旦有了这种直觉,只要再前进一小步,就能精确地拟合出参数 $a_0\ldots a_{M-1}$,也就是通过求上面定义的似然的**最大值**来确定这些参数。这种参数估计方法就是**最大似然估计**。

现在要和(15.1.2)式联系起来。假设每个数据点 y_i 的测量误差是随机独立的,并且围绕"真实"模型 $y(x)$ 呈正态(高斯)分布。还假设这些正态分布的标准差 σ 对所有的数据点都是一样的,那么,数据集的概率为各个数据点概率的乘积:

$$P\propto\prod_{i=0}^{N-1}\left\{\exp\left[-\frac{1}{2}\left(\frac{y_i-y(x_i)}{\sigma}\right)^2\right]\Delta y\right\} \tag{15.1.3}$$

注意,在乘积的每一项中有因子 Δy。求(15.1.3)式的极大值等价于极大化它的对数,或者极小化它的负对数,即

$$\left[\sum_{i=0}^{N-1}\frac{[y_i-y(x_i)]^2}{2\sigma^2}\right]-N\log\Delta y \tag{15.1.4}$$

因为 N、σ、Δy 都是常数,求上式的极小值等价于求(15.1.2)式的极小值。

正如上面讨论的,如果测量误差是独立的,并且服从具有常数标准差的正态分布,则最小二乘方拟合就是拟合参数的最大似然估计。注意,对模型 $y(x;a_0\ldots)$ 中的参数 $a_0\ldots a_{M-1}$ 是线性的还是非线性的未做任何假设。在下面,将不假设标准差为常数而得到非常相似的式子,它被称为"χ^2 拟合"或"加权最小二乘方拟合"。但是,首先还是进一步讨论严格假设下的正态分布。

在100年左右的时间里,数理统计学家面对这样一个事实——大量非常小的随机偏差之和的概率分布几乎总是收敛于一个正态分布(对于更准确的**中心极限定理**的叙述,参阅本节参考文献[1]或其他数理统计的著作)。这使人们的注意力远离了这样一个事实,即对实际的数据来说,如果能实现的话,正态分布常常是非常粗糙的实现。我们经常而不是偶然地被告知,平均地看,测量值在真值 $\pm\sigma$ 范围内的概率是68%,在 $\pm2\sigma$ 范围内的概率是95%,在 $\pm3\sigma$ 范围内的概率是99.7%。继续扩展下去,我们能预测测量的结果在 $\pm20\sigma$ 以外的概率为 $1/(2\times10^{88})$。我们都知道"假信号"比这种情况更有可能发生。

在有些情况中,正态分布的偏差很容易理解和数量化。例如在计数事件的测量中,测量的误差通常表现为泊松(Poisson)分布,它的积累概率函数已经在6.2节中讨论过。当涉及一个数据点的计数数目非常大时,泊松分布趋向于高斯分布。但是当按相对精度测量时,这种收敛不是一致的。分布尾部的标准差越多,则某个数值实现趋于高斯分布所需的计数数目就越大。这种效应的表现永远是一样的:高斯分布预计的"尾部"

事件远少于它们的实际情况(由泊松分布确定)。这就导致这样一些事件,如果发生的话,它比实际情况更容易偏离最小二乘方拟合。

在另外一些情况下,一个正态分布的偏差在一些细节上难于掌握。有些实验数据点有时**偏离正道**。也许在某一点的测量时,电源突然有一点起伏,或者某人踢了仪器,或者某人写下了一个错误的数字。这种点被称为**偏离点**。它们非常容易把原本对其他数据十分适合的最小二乘方拟合变得毫无意义。在假设的高斯模型下,它们发生的概率非常小,以致最大似然估计使整条曲线变形,把它们错误地变成直线。

稳健统计是解决当正态即高斯模型是一个比较差的逼近,或者"偏离点"非常重要的情况下的课题。将在 15.7 节中简要地讨论稳健统计方法。从现在起一直到 15.7 节我们都假设数据测量误差符合高斯模型。非常重要的一点是,必须记住这个模型的局限性,特别在应用由此假设导出的一些有用的方法时更应如此。

最后要注意的是,对测量误差的讨论局限于统计误差,这种误差在取大量的数据进行平均时是能去掉的。测量还易产生系统误差,它是不能通过大量数据平均消去的。例如,金属米棒的校准依赖于温度。如果我们的测量在同一错误的温度下进行,则任何数目的平均或数据处理都不能校正这个不可知的系统误差。

15.1.1 χ^2 拟合

在 14.3 节曾经讨论过 χ^2 统计,这里将在稍微不同的情况下进行讨论。

如果每个数据点(x_i, y_i)有它自己的已知标准差 σ_i,那么只要把(15.1.3)式略微变化一下,在符号 σ 下加一个下标 i。这个下标同样自然地加进(15.1.4)式,以致通过求下面这个量的极小值即可获得模型参数的最大似然估计:

$$\chi^2 \equiv \sum_{i=0}^{N-1} \left(\frac{y_i - y(x_i; a_0 \ldots a_{M-1})}{\sigma_i} \right)^2 \tag{15.1.5}$$

这个量被称为"chi 平方"。

不论在何种程度上,测量误差事实上确实是正态分布,量 χ^2 是相应的 N 个正态分布量的平方之和,其中每个量均归一化成单位方差。一旦调整参数 $a_0 \ldots a_{M-1}$ 使得 χ^2 取极小值,则在和式中的各项就不完全是统计独立的。但是对于在参数 a 中是线性的模型来说,已经说明在最小值处,不同取值的 χ^2 的概率分布还是能解析地导出的,它符合**自由度为 $N-M$ 的 χ^2 分布**。在 6.2 节已经知道怎样用不完全的伽马(gamma)函数 gammq 来计算这种概率函数。特别是(6.2.18)式给出了 χ^2 偶然超过某一特定的 χ^2 值的概率Q,其中 $\nu = N-M$ 是**自由度个数**。量Q,或者它的互补量 $P = 1-Q$经常在统计书中以列表形式给出,但是我们发现用 gammq 非常简单,例如计算我们要的值Q = gammq$(0.5\nu, 0.5\chi^2)$。当参数 a 不是严格的线性模型时,假设 χ^2 分布仍然是适用的,这种做法很普遍,不会有大的错误。

这个计算出来的概率值可以作为评估模型拟合好坏的标准。如果对于某一特定数据集,Q是一个非常小的概率数值,那么这种明显的不一致不可能是偶然的波动导致的。更可能是:(i)模型是错误的——从统计的观点可被抛弃,或者(ii)某人错误地告诉了测量误差 σ_i 的大小——它们实际上比所陈述的要大。

还有很重要的一点,那就是 χ^2 的概率 Q 并不能直接衡量测量误差呈正态分布的假设的可信度。我们只是假设误差确实如此。然而在多数但非全部情况下,非正态分布误差的效果是产生许多"偏离点"。这些点使概率值Q减小,所以除了上面列出的两点外,还可以增加一个可能的但不是确定的结论:(iii)测量误差可能不是呈正态分布。

(iii)的发生是很普遍的,并且也相当温和。正是由于这个原因使得明智的实验者常常宁可容忍低概率值Q。对于任何模型通常还可以确定一个可接受的Q值,比如$Q > 0.001$。这并不像它看起来那样粗糙:真正的错误模型会由于非常小的Q值(比如 10^{-18})而被舍弃。但是如果总是接受$Q \sim 10^{-3}$的模型,就应该寻找原因。

如果碰巧知道测量误差的实际分布规律,那么就可以用**蒙特卡罗(Monte Carlo)模拟**的方法从一个特定的模型中得出数据集,如 7.2 节~7.3 节所述。接着可以将模拟数据集应用于实际拟合过程,不但可以确定 χ^2 统计的概率分布,还可以确定由拟合所得出的模型参数精确性,将在 15.6 节对此做进一步的讨论。它的技巧是非常一般的,但它的代价非常昂贵。

在另外一个相反的极端,有些时候概率值Q太大,接近于 1,简直好得令人难以置信。非正态的测量误差一般不可能导致这种弊端,因为正态分布已达到任何一个可能分布所能达到的"紧凑"。一个非常好的 χ^2 拟

合的原因几乎永远是,实验者在一种保守的"拟合"中过高估计了测量误差。还有一种非常小的可能,一个过好的 χ^2 拟合实际上是为适合模型而捏造出来的。

首要的规则是,一个拟合"中等"好的典型 χ^2 值是 $\chi^2 \approx \nu$。更准确的表述是,χ^2 统计量具有均值 ν 和标准差 $\sqrt{2\nu}$,并且对于大的 ν,χ^2 统计量渐近地变成正态分布。

在某些情况下,和一组测量值相联系的不确定性不能事先知道,并且有关 χ^2 拟合的想法用来导出 σ 的值。如果假设所有的测量值有相等的标准差 $\sigma_i = \sigma$,且模型拟合得也很好,那么可以采取下面的步骤。首先对所有的点任意假设一个常数 σ,接着通过求 χ^2 的极小值拟合模型参数,最后重新计算

$$\sigma^2 = \sum_{i=0}^{N-1} [y_i - y(x_i)]^2/(N-M) \tag{15.1.6}$$

显然,这种方法使我们不能对拟合优度进行独立的估算,这也是被使用者偶然忽略的问题。但是,在测量误差未知的情况下,这种方法至少能为各个数据点提供某种误差上限。

如果在(15.1.5)式中分别对参数 a_k 求微分,我们就得到使 χ^2 取最小值的方程组:

$$0 = \sum_{i=0}^{N-1} \left(\frac{y_i - y(x_i)}{\sigma_i^2}\right)\left(\frac{\partial y(x_i; \ldots a_k \ldots)}{\partial a_k}\right) \qquad k = 0, \ldots, M-1 \tag{15.1.7}$$

一般情况下,(15.1.7)式是含有 M 个未知数 a_k 的 M 个非线性方程。在本章随后所叙述的许多过程都由(15.1.7)式及它的特例导出。

参考文献和深入阅读

Bevington, P.R., and Robinson, D.K. 1992, *Data Reduction and Error Analysis for the Physical Sciences*, 2nd ed. (New York: McGraw-Hill), Chapters 1–3.

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), §VI.C. [1]

15.2 拟合数据成直线

一个具体的实例将使上节的讨论更有意义。考虑一个用 N 个数据点(x_i, y_i)拟合成直线模型的问题,直线模型为

$$y(x) = y(x; a, b) = a + bx \tag{15.2.1}$$

这个问题被称为**线性回归**,这个术语很久以前源于社会科学。假设和每个测量值 y_i 相联系的不确定性值σ_i是已知的,还假设 x_i 的值(自变量值)是精确已知的。

为了衡量模型和数据的一致程度,采用 χ^2 优值函数(15.1.5)式,在直线拟合这种情况下,(15.1.5)式变为

$$\chi^2(a, b) = \sum_{i=0}^{N-1} \left(\frac{y_i - a - bx_i}{\sigma_i}\right)^2 \tag{15.2.2}$$

如果测量误差是正态分布,则上面的优值函数将给出参数 a 和 b 的最大似然估计;如果测量误差不是正态分布,则这种估计将不是最大似然估计,但在实际应用中仍然有用。在 15.7 节将处理偏离点太多以致优值函数 χ^2 毫无用处的情况。

求(15.2.2)式的最小值来确定 a 和 b。在(15.2.2)的最小值处,$\chi^2(a, b)$对 a 和 b 的导数为 0:

$$\begin{aligned} 0 &= \frac{\partial \chi^2}{\partial a} = -2 \sum_{i=0}^{N-1} \frac{y_i - a - bx_i}{\sigma_i^2} \\ 0 &= \frac{\partial \chi^2}{\partial b} = -2 \sum_{i=0}^{N-1} \frac{x_i(y_i - a - bx_i)}{\sigma_i^2} \end{aligned} \tag{15.2.3}$$

如果定义下面的和式，则上述条件可以再写成一个比较简便的形式：

$$S \equiv \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \quad S_x \equiv \sum_{i=0}^{N-1} \frac{x_i}{\sigma_i^2} \quad S_y \equiv \sum_{i=0}^{N-1} \frac{y_i}{\sigma_i^2}$$
$$S_{xx} \equiv \sum_{i=0}^{N-1} \frac{x_i^2}{\sigma_i^2} \quad S_{xy} \equiv \sum_{i=0}^{N-1} \frac{x_i y_i}{\sigma_i^2} \tag{15.2.4}$$

有了这些定义，(15.2.3)式变为

$$aS + bS_x = S_y$$
$$aS_x + bS_{xx} = S_{xy} \tag{15.2.5}$$

这两个含有两个未知数的方程的解可计算如下：

$$\Delta \equiv SS_{xx} - (S_x)^2$$
$$a = \frac{S_{xx}S_y - S_xS_{xy}}{\Delta} \tag{15.2.6}$$
$$b = \frac{SS_{xy} - S_xS_y}{\Delta}$$

(15.2.6)式给出了最佳拟合模型参数 a 和 b 的解。

但是不能到此结束。因为数据的测量误差必然会导致确定这些参数的不确定度，所以必须估计 a 和 b 估计的不确定度。如果各个数据点是独立的，则每个数据点的不确定度都会对参数造成影响。考虑到误差的传播，我们得到任何函数值的方差 σ_f^2 是

$$\sigma_f^2 = \sum_{i=0}^{N-1} \sigma_i^2 \left(\frac{\partial f}{\partial y_i}\right)^2 \tag{15.2.7}$$

对于上面的直线情形，a 和 b 对 y_i 的导致可以直接由解计算：

$$\frac{\partial a}{\partial y_i} = \frac{S_{xx} - S_x x_i}{\sigma_i^2 \Delta}$$
$$\frac{\partial b}{\partial y_i} = \frac{Sx_i - S_x}{\sigma_i^2 \Delta} \tag{15.2.8}$$

然后，按照(15.2.7)式那样对各个数据点求和，得到

$$\sigma_a^2 = S_{xx}/\Delta$$
$$\sigma_b^2 = S/\Delta \tag{15.2.9}$$

它们分别是 a 和 b 估计的方差。在 15.6 节我们将看到，为了正确地刻画参数估计的可能不确定度，还需要增加一个量。这个量就是 a 和 b 的**协方差**，并且由下式给出：

$$\mathrm{Cov}(a,b) = -S_x/\Delta \tag{15.2.10}$$

a 和 b 不确定度之间的相关系数是一个在 -1 和 1 之间的数值，根据(15.2.10)式(和(14.5.1)式比较)可得

$$r_{ab} = \frac{-S_x}{\sqrt{SS_{xx}}} \tag{15.2.11}$$

正的 r_{ab} 值说明 a 和 b 的误差可能同号，而负的 r_{ab} 值说明 a 和 b 的误差是反相关，可能有相反的符号。

我们的工作还不能到此结束。还必须估计数据和模型之间的拟合优度：缺少这一估计，将使模型中的参数 a 和 b 没有任何意义。一个和(15.2.2)式所确定的值一样粗劣的 χ^2 值之概率 Q，由下式计算：

$$Q = \mathbf{gammq}\left(\frac{N-2}{2}, \frac{\chi^2}{2}\right) \tag{15.2.12}$$

其中 gammq 是 6.2 节中计算不完全伽马函数$Q(a,x)$的程序。如果比 0.1 大,那么拟合优度是可信的,如果Q比 0.001 大,只要误差是非正态分布或者被适当低估,那么拟合或许是可以接受的。如果Q比 0.001 小,那么模型和/或估计过程有问题。对于后一种情况,在 15.7 节进行了深入的讨论。

如果不知道每个数据点的测量误差 σ_i,而用(15.1.6)式估计这些误差,则可以采取如下的步骤估计参数 a 和 b 的可能不确定度:在(15.2.6)式前后所有等式中设 $\sigma_i \equiv 1$,然后将(15.2.9)式中得到的 σ_a 和 σ_b 乘以附加的因子$\sqrt{\chi^2/(N-2)}$,其中 χ^2 由拟合出来的参数 a 和 b 根据(15.2.2)式计算。和上面讨论的一样,这个过程等价于假设拟合得很好,因此得不到独立的拟合优度的概率Q。

在 14.5 节中,我们保留了线性相关系数 r((14.5.1)式)和拟合优度测量值 χ^2 ((15.2.2)式)之间的关系,对于不加权的数据点(即所有的 $\sigma_i = 1$),这个关系是

$$\chi^2 = (1-r^2)\mathrm{NVar}(y_0 \ldots y_{N-1}) \tag{15.2.13}$$

其中

$$\mathrm{NVar}(y_0 \ldots y_{N-1}) \equiv \sum_{i=0}^{N-1}(y_i - \overline{y})^2 \tag{15.2.14}$$

对于有不同权重值 σ_i 的数据点,如果将(14.5.1)式中的求和乘以权重 $1/\sigma_i^2$,上式仍然有效。

下面的函数 fit 准确地实现了上面所讨论的运算。当权重 σ 事先已知时,函数的计算和上面的公式完全一致。但是,当权重 σ 未知时,程序过程假设每个点有相同的 σ,而且还假设拟合得很好,正如 15.1 节中讨论的那样。

(15.2.6)式对舍入误差很敏感,因此将它们改写如下 定义

$$t_i = \frac{1}{\sigma_i}\left(x_i - \frac{S_x}{S}\right), \qquad i = 0, 1, \ldots, N-1 \tag{15.2.15}$$

和

$$S_{tt} = \sum_{i=0}^{N-1} t_i^2 \tag{15.2.16}$$

那么,通过直接的代换可证明

$$b = \frac{1}{S_{tt}}\sum_{i=0}^{N-1}\frac{t_i y_i}{\sigma_i} \tag{15.2.17}$$

$$a = \frac{S_y - S_x b}{S} \tag{15.2.18}$$

$$\sigma_a^2 = \frac{1}{S}\left(1 + \frac{S_x^2}{S S_{tt}}\right) \tag{15.2.19}$$

$$\sigma_b^2 = \frac{1}{S_{tt}} \tag{15.2.20}$$

$$\mathrm{Cov}(a,b) = -\frac{S_x}{S S_{tt}} \tag{15.2.21}$$

$$r_{ab} = \frac{\mathrm{Cov}(a,b)}{\sigma_a \sigma_b} \tag{15.2.22}$$

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::fit(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sig, const bool mwt, DP &a,
    DP &b, DP &siga, DP &sigb, DP &chi2, DP &q)
```

给定一组数据点 x[0..ndata-1],y[0..ndata-1],它们具有标准差 sig[0..ndata-1],通过求 χ^2 的极小值将这些数据点拟合成直线 $y = a + bx$。返回值 a,b 和它们各自的不确定度 siga 和 sigb,χ^2 值 chi2,以及拟合优度概率值 q(拟合将使 χ^2

具有这么大的值或者更大)。如果在输入中 mwt = false,那么就认为不知道数据点标准差,q 将以 1.0 返回,chi2 归一化为所有各点的单位标准差。

```
{
    int i;
    DP wt,t,sxoss,sx=0.0,sy=0.0,st2=0.0,ss,sigdat;

    int ndata=x.size();
    b=0.0;
    if (mwt) {                                          累加求和……
        ss=0.0;
        for (i=0;i<ndata;i++) {                         ……考虑权重
            wt=1.0/SQR(sig[i]);
            ss += wt;
            sx += x[i]*wt;
            sy += y[i]*wt;
        }
    } else {
        for (i=0;i<ndata;i++) {                         ……或没有权重。
            sx += x[i];
            sy += y[i];
        }
        ss=ndata;
    }
    sxoss=sx/ss;
    if (mwt) {
        for (i=0;i<ndata;i++) {
            t=(x[i]-sxoss)/sig[i];
            st2 += t*t;
            b += t*y[i]/sig[i];
        }
    } else {
        for (i=0;i<ndata;i++) {
            t=x[i]-sxoss;
            st2 += t*t;
            b += t*y[i];
        }
    }
    b /= st2;                                           求 a, b, σa 和 σb。
    a=(sy-sx*b)/ss;
    siga=sqrt((1.0+sx*sx/(ss*st2))/ss);
    sigb=sqrt(1.0/st2);
    chi2=0.0;                                           计算 χ²。
    q=1.0;
    if (!mwt) {
        for (i=0;i<ndata;i++)
            chi2 += SQR(y[i]-a-b*x[i]);
        sigdat=sqrt(chi2/(ndata-2));                    对无权重数据，用chi2 求sig,
        siga *= sigdat;                                 并且调整标准差。
        sigb *= sigdat;
    } else {
        for (i=0;i<ndata;i++)
            chi2 += SQR((y[i]-a-b*x[i])/sig[i]);
        if (ndata>2) q=gammq(0.5*(ndata-2),0.5*chi2);       (15.2.12)式。
    }
}
```

参考文献和深入阅读

Bevington, P.R., and Robinson, D.K. 1992, *Data Reduction and Error Analysis for the Physical Sciences*, 2nd ed. (New York: McGraw-Hill), Chapter 6.

15.3　深入讨论:两个坐标数据都有误差的直线拟合

如果实验数据的测量误差不仅仅存在于 y_i 中,还在存在于 x_i 中,则拟合直线模型

$$y(x) = a + bx \tag{15.3.1}$$

的任务将困难得多。我们直接写出在这种情况下的 χ^2 优值函数,

$$\chi^2(a,b)=\sum_{i=0}^{N-1}\frac{(y_i-a-bx_i)^2}{\sigma_{y_i}^2+b^2\sigma_{x_i}^2} \tag{15.3.2}$$

其中 σ_{x_i} 和 σ_{y_i} 分别是第 i 个点 x 和 y 的标准误差。(15.3.2)式分母中的加权方差和可以从两个方面来理解，或者理解为在最小 χ^2 方向上每个数据点和斜率为 b 的直线的方差，或理解为随机变量 x_i 和 y_i 的线性组合 y_i-a-bx_i 的方差：

$$\mathrm{Var}(y_i-a-bx_i)=\mathrm{Var}(y_i)+b^2\mathrm{Var}(x_i)=\sigma_{y_i}^2+b^2\sigma_{x_i}^2\equiv 1/w_i \tag{15.3.3}$$

因此，N 个随机变量方差的平方和是一个 χ^2 分布，并且其中每个随机变量都已被它的方差归一化。

根据 a 和 b 的值最小化(15.3.2)式。遗憾的是，b 在(15.3.2)式分母中的出现，使得求斜率等式 $\partial\chi^2/\partial b=0$ 是非线性的，但是相应的截距条件 $\partial\chi^2/\partial a=0$ 仍然是线性的，并且有

$$a=\left[\sum_i w_i(y_i-bx_i)\right]\Bigg/\sum_i w_i \tag{15.3.4}$$

其中的 w_i 由(15.3.3)式定义。一个合理的方案是采用第10章中求一般的一维函数最小值的方法(参看程序brent)，求 b 的最值，同时在每一步中用式(15.3.4)保证对 b 求最小值时，也对 a 求最小值。

因为 x_i 的误差是有限的，当 b 为无穷大时(直线斜率为无穷大)，以 b 为变量的 χ^2 函数的最小值也是有限的，尽管这个值通常很大。因此角度 $\theta\equiv\arctan b$ 比 b 更适于作为斜率的参量。χ^2 的值从 θ 来看将是周期的，周期为 π(不是 2π)。如果任何数据点的 σ_y 比较小，σ_x 中等或很大，那么有可能在斜率 $\theta\approx 0$ 处 χ^2 取最大值。在这种情况下，有可能存在两个 χ^2 最小值，一个在正斜率处，另一个在负斜率处。其中只有一个是正确的整体极小。因此对 b(或 θ)有一个优良的开始猜测值是非常重要的。在下面采取的策略是对 y_i 的值进行量化，以使它的方差等于 x_i 的方差，然后根据从(量化的)和式 $\sigma_{x_i}^2+\sigma_{y_i}^2$ 导出的权重进行常见的线性拟合(如同15.2节)。如果数据点和线性模型确实合理地相关，这将对 b 产生一个非常好的猜测值。

要确定参数 a 和 b 的标准差 σ_a 和 σ_b 更加困难。我们在15.6节中将看到，在适当的情况下，a 和 b 的标准误分别是"置信区域边界"在 a 和 b 轴上的投影，在"置信区域边界"上 χ^2 取比它的最小值大1的值，$\Delta\chi^2=1$。在15.2节的线性情况下，这些投影遵从泰勒级数展开：

$$\Delta\chi^2\approx\frac{1}{2}\left[\frac{\partial^2\chi^2}{\partial a^2}(\Delta a)^2+\frac{\partial^2\chi^2}{\partial b^2}(\Delta b)^2\right]+\frac{\partial^2\chi^2}{\partial a\partial b}\Delta a\Delta b \tag{15.3.5}$$

因为在目前情况下，b 是非线性的，二阶导数的解析式非常复杂；更重要的是，低阶项对 $\Delta\chi^2$ 经常给出一个比较差的近似。因此我们的策略是，通过调整斜率 b 的值远离最小值来用数值方法求解 $\Delta\chi^2=1$ 的根。在下面的程序中采用了一般的寻根程序zbrent。有时可能一个根都没有。例如，如果所有的误差限非常大，会导致所有的数据点是彼此相容的。因此在精确确定一个根以前，划出一个求根区间就非常重要(参见9.1节)。

因为 a 是在变化 b 的每一步中得到最小值的，所以成功的数值求根会产生 Δa 值，对于能够使 $\Delta\chi^2=1$ 成立的 Δb 值，该 Δa 值最小化 χ^2。这(参见图15.3.1)直接给出了置信区域在 b 轴上的切线投影，亦即 σ_b。但是它没有给出置信区域在 a 轴上的切线投影。在图15.3.1中我们已经找到 B 点。为了得到 σ_a，必须寻找点 A。几何上能保证：在置信区域近似是椭圆时，可以证明(参见图15.3.1)$\sigma_a^2=r^2+s^2$。s 的值可以从图中的 B 点得到，而 r 的值可以将(15.3.2)式和(15.3.3)式应用到 χ^2 的最小值(即图中的 O 点)得到，给出

$$r^2=1\Bigg/\sum_i w_i \tag{15.3.6}$$

实际上，因为 b 能趋于无穷大，因此整个过程在(a,θ)空间比在(a,b)空间更有意义。这实际就是下面程序的工作过程。但是由于习惯上总是返回 a 和 b 的标准误，而不是 a 和 θ，最后采用关系

$$\sigma_b=\sigma_\theta/\cos^2\theta \tag{15.3.7}$$

必须注意，如果 b 和它的标准误都很大，以至于置信区域实际上将包括无穷大斜率，那么标准误 σ_b 没有很大

意义。函数 chixy 通常仅被程序 fitexy 调用。但是，如果用户想要的话，可以自己探寻置信区域，这只要在初始调用一次 fitexy 后，重复调用 chixy(它的自变量是一个角度 θ，而不是斜率 b)。

最后要注意，和 15.0 节相同，如果拟合优度不可接受(被返回的概率太小)，标准误 σ_a 和 σ_b 当然是不可信的。在不好的情况下，可能还要试着用一个常数因子标度 x 和 y 的误差限，直到被返回的概率是可接受的(例如 0.5)，以得到更合理的 σ_a 和 σ_b 值。

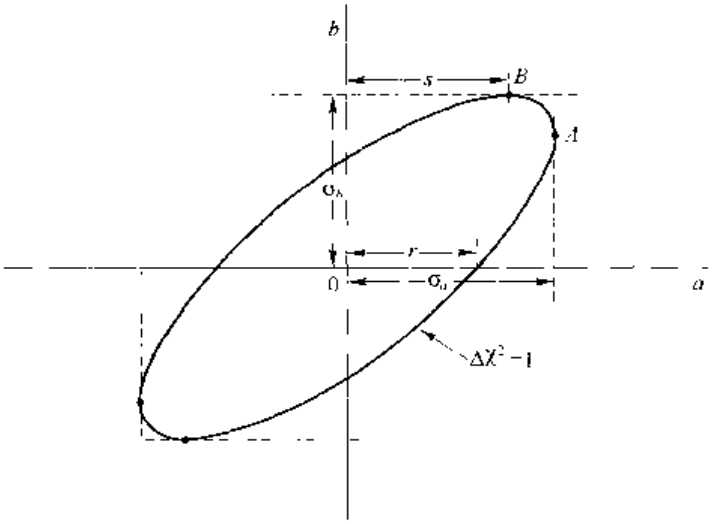


图 15.3.1　参数 a 和 b 的标准误。B 点可以由变化斜率 b 而同时最小化截距 a 的方法得到。这将给出标准误 σ_b 及 s 的值。标准误 σ_a 可以通过几何关系 $\sigma_a^2 = s^2 + r^2$ 得到

```
#include <cmath>
#include "nr.h"
using namespace std;

Vec_DP *xx_p,*yy_p,*sx_p,*sy_p,*ww_p;              与程序 chixy 通信的全局变量。
DP aa,offs;

void NR::fitexy(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sigx, Vec_I_DP &sigy,
    DP &a, DP &b, DP &siga, DP &sigb, DP &chi2, DP &q)
对 x 和 y 都具有误差的输入数据 x[0..ndat-1]和 y[0..ndat-1]进行直线拟合。它们各自的标准差在输入量 sigx[0..ndat
-1]和 sigy[0..ndat-1]中。输出量是使拟合直线 y = a + bx 的 χ² 值最小的 a 和 b。这个最小 χ² 以 chi2 返回。χ² 的概
率以 q 返回，较小的 q 值表明较差的拟合(有时表明低估了的误差)。a 和 b 标准误以 siga 和 sigb 返回。这些结果在如下
两种情况下将无意义：(i)拟合较差，(ii) b 太大以致数据和垂直线(无穷大 b)一致。如果 siga 和 sigb 的返回值为 BIG，则
数据将和 b 的所有值一致
{
    int j;
    const DP POTN=1.571000,BIG=1.0e30,ACC=1.0e-3;
    const DP PI=3.141592653589793238;
    DP amx,amn,varx,vary,ang[7],ch[7],scale,bmn,bmx,d1,d2,r2,
        dum1,dum2,dum3,dum4,dum5;

    int ndat=x.size();
    xx_p=new Vec_DP(ndat);
    yy_p=new Vec_DP(ndat);
    sx_p=new Vec_DP(ndat);
    sy_p=new Vec_DP(ndat);
    ww_p=new Vec_DP(ndat);
    Vec_DP &xx=*xx_p, &yy=*yy_p;                   使用引用以简化代码。
    Vec_DP &sx=*sx_p, &sy=*sy_p, &ww=*ww_p;
    avevar(x,dum1,varx);                           寻找 x 和 y 变量，量化数据成为全局
    avevar(y,dum1,vary);                           变量以便和函数 chixy 进行通信。
    scale=sqrt(varx/vary);
    for (j=0;j<ndat;j++) {
        xx[j]=x[j];
        yy[j]=y[j]*scale;
        sx[j]=sigx[j];
        sy[j]=sigy[j]*scale;
```

```
        ww[j]=sqrt(SQR(sx[j])+SQR(sy[j]));          在第一次试验拟合中考虑全部x和y的权重。
    }
    fit(xx,yy,ww,true,dum1,b,dum2,dum3,dum4,dum5);      对b试验拟合。
    offs=ang[0]=0.0;                                根据参考数据点构造几个角度,
    ang[1]=atan(b);                                 并使b为一角度。
    ang[3]=0.0;
    ang[4]=ang[1];
    ang[5]=POTN;
    for (j=3;j<6;j++) ch[j]=chixy(ang[j]);
    mnbrak(ang[0],ang[1],ang[2],ch[0],ch[1],ch[2],chixy);
    括出χ²的最小,并放置于brent。
    chi2=brent(ang[0],ang[1],ang[2],chixy,ACC,b);
    chi2=chixy(b);
    a=aa;
    q=gammq(0.5*(ndat-2),chi2*0.5);                 计算χ²概率。
    r2=0.0;
    for (j=0;j<ndat;j++) r2 += ww[j];               在最小值处保存权重之和的倒数。
    r2=1.0/r2;
    bmx=BIG;                                        现在在Δχ²=1处寻找b的标准误。
    bmn=BIG;
    offs=chi2+1.0;
    for (j=0;j<6;j++) {                             通过保存的值括出想要的根,
        if (ch[j] > offs) {                         注意倾斜角度的周期性。
            d1=fabs(ang[j]-b);
            while (d1 >= PI) d1 -= PI;
            d2=PI-d1;
            if (ang[j] < b)
                SWAP(d1,d2);
            if (d1 < bmx) bmx=d1;
            if (d2 < bmn) bmn=d2;
        }
    }
    if (bmx < BIG) {                                调用zbrent寻根。
        bmx=zbrent(chixy,b,b+bmx,ACC)-b;
        amx=aa-a;
        bmn=zbrent(chixy,b,b-bmn,ACC)-b;
        amn=aa-a;
        sigb=sqrt(0.5*(bmx*bmx+bmn*bmn))/(scale*SQR(cos(b)));
        siga=sqrt(0.5*(amx*amx+amn*amn)+r2)/scale;      a的误差多一项r2。
    } else sigb=siga=BIG;
    a /= scale;                                     将标度的结果复原。
    b=tan(b)/scale;
    delete ww_p; delete sy_p; delete sx_p; delete yy_p; delete xx_p;
}


#include <cmath>
#include "nr.h"
using namespace std;

extern Vec_DP *xx_p,*yy_p,*sx_p,*sy_p,*ww_p;
extern DP aa offs;

DP NR::chixy(const DP bang)
程序fitexy的辅助函数。对于斜率b=tan(bang),它返回(χ² - offs)值。被标度的数据和offs通过全局变量进行传递。
{
    const DP BIG=1.0e30;
    int j;
    DP ans,avex=0.0,avey=0.0,sumw=0.0,b;

    Vec_DP &xx=*xx_p, &yy=*yy_p;        使用引用以简化代码。
    Vec_DP &sx=*sx_p, &sy=*sy_p, &ww=*ww_p;
    int nn=xx.size();
    b=tan(bang);
    for (j=0;j<nn;j++) {
        ww[j] = SQR(b*sx[j])+SQR(sy[j]);
        sumw += (ww[j]=(ww[j] < 1.0/BIG ? BIG : 1.0/ww[j]));
        avex += ww[j]*xx[j];
        avey += ww[j]*yy[j];
    }
```

```
        avex /= sumw;
        avey /= sumw;
        aa=avey-b*avex;
        for (ans = -offs,j=0;j<nn;j++)
            ans += ww[j]*SQR(yy[j]-aa-b*xx[j]);
        return ans;
    }
```

注意,看起来比较直接的、有关本节内容的文献,一般令人疑惑,有时还完全是错的。Deming[1]早期的处理是合理的,但是这个处理依赖于泰勒展开,给出了不精确的误差估计。参考文献[2~4]是可靠的、更近代的,并以批判的态度对早期的工作进行了更一般的处理。York[5]和 Reed[6]对本节所处理的简单的直线情况进行了有用的讨论,但是后者的文献有些错误,这些错误在本节参考文献[7]中得到了纠正。所有这些讨论被 Bayesians[8~10]整理,但他还持有不同的观点。

参考文献和深入阅读

Deming, W.E. 1943, *Statistical Adjustment of Data* (New York: Wiley), reprinted 1964 (New York: Dover). [1]

Jefferys, W.H. 1980, *Astronomical Journal*, vol. 85, pp. 177–181; see also vol. 95, p. 1299 (1988). [2]

Jefferys, W.H. 1981, *Astronomical Journal*, vol. 86, pp. 149–155; see also vol. 95, p. 1300 (1988). [3]

Lybanon, M. 1984, *American Journal of Physics*, vol. 52, pp. 22–26. [4]

York, D. 1966, *Canadian Journal of Physics*, vol. 44, pp. 1079–1086. [5]

Reed, B.C. 1989, *American Journal of Physics*, vol. 57, pp. 642–646; see also vol. 58, p. 189, and vol. 58, p. 1209. [6]

Reed, B.C. 1992, *American Journal of Physics*, vol. 60, pp. 59–62. [7]

Zellner, A. 1971, *An Introduction to Bayesian Inference in Econometrics* (New York: Wiley); reprinted 1987 (Malabar, FL: R. E. Krieger Pub. Co.). [8]

Gull, S.F. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [9]

Jaynes, E.T. 1991, in *Maximum-Entropy and Bayesian Methods, Proc. 10th Int. Workshop*, W.T. Grandy, Jr., and L.H. Schick, eds. (Boston: Kluwer). [10]

Macdonald, J.R., and Thompson, W.J. 1992, *American Journal of Physics*, vol. 60, pp. 66–73.

15.4 一般的线性最小二乘方

在 15.2 节中讨论的直接推广是,拟合一组数据点(x_i, y_i)集的模型不只是 1 和 x 的线性组合(也就是 $a+bx$),而是 x 的任意 M 个特定函数的线性组合。例如,x 的函数可以是 $1, x, x^2, \ldots, x^{M-1}$,这种情况下,它们的一般线性组合

$$y(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{M-1} x^{M-1} \tag{15.4.1}$$

是一个 $M-1$ 阶的多项式。或者 x 的函数是正弦和余弦,这种情况下它们的线性组合就是一个调和级数。

这种模型的一般形式是

$$y(x) = \sum_{k=0}^{M-1} a_k X_k(x) \tag{15.4.2}$$

其中 $X_0(x), \ldots, X_{M-1}(x)$是 x 的任意固定函数,称为基函数。

注意,函数 $X_k(x)$可以是 x 的非线性函数。这里讨论的线性仅指模型和它的参数 a_k 之间的依赖关系。

对于这种线性模型我们推广前一节的讨论,定义优值函数

$$\chi^2 = \sum_{i=0}^{N-1} \left[\frac{y_i - \sum_{k=0}^{M-1} a_k X_k(x_i)}{\sigma_i} \right]^2 \tag{15.4.3}$$

和前面一样,σ_i 是第 i 个数据点的测量误差(标准差),并且假设是已知的。如果测量误差未知,那么和15.1节

中讨论的一样,可令所有测量误差为常数值 $\sigma = 1$。

现在将再次寻求使 χ^2 最小化的最佳参数。有几种不同的技巧可以求出这个最小值。其中两种特别有用,在本节我们逐一讨论。为了引入和阐明它们之间的关系,需要引入一些符号。

令 $\mathbf{A}$ 为 $N \times M$ 阶的矩阵,它的元素由 M 个基函数在 N 个横坐标 x_i 上的取值,以及 N 个测量误差 σ_i 计算求得,也就是如下的定义:

$$A_{ij} = \frac{X_j(x_i)}{\sigma_i} \tag{15.4.4}$$

矩阵 $\mathbf{A}$ 称为拟合问题的**设计矩阵**。注意,通常情况下 $\mathbf{A}$ 的行数大于列数,即 $N \geqslant M$,因为数据点比所求模型参数多(当然可以用两个数据点拟合一条直线,但不是非常有意义)。设计矩阵如图 15.4.1 所示。

$$\begin{array}{c} \longleftarrow \text{基函数} \longrightarrow \\ X_0(\ ) \quad X_1(\ ) \quad \cdots \quad X_{M-1}(\ ) \end{array}$$

$$\text{数据点}\left\{\begin{array}{c} x_0 \\ x_1 \\ \vdots \\ \\ \\ \\ x_{N-1} \end{array}\right. \begin{pmatrix} \frac{X_0(x_0)}{\sigma_0} & \frac{X_1(x_0)}{\sigma_0} & \cdots & \frac{X_{M-1}(x_0)}{\sigma_0} \\ \frac{X_0(x_1)}{\sigma_1} & \frac{X_1(x_1)}{\sigma_1} & \cdots & \frac{X_{M-1}(x_1)}{\sigma_1} \\ \vdots & & & \vdots \\ \frac{X_0(x_{N-1})}{\sigma_{N-1}} & \frac{X_1(x_{N-1})}{\sigma_{N-1}} & \cdots & \frac{X_{M-1}(x_{N-1})}{\sigma_{N-1}} \end{pmatrix}$$

图 15.4.1 由 M 个基函数线性组合,对 N 个数据点做最小二乘方拟合的设计矩阵。矩阵的元素包括由测量的自变量计算出来的基函数值,还考虑了已测量因变量的标准差。因变量的测量值不包括在此矩阵中

再通过下式定义长度为 N 的向量 $\mathbf{b}$:

$$b_i = \frac{y_i}{\sigma_i} \tag{15.4.5}$$

并且定义一个长度为 M 的向量 $\mathbf{a}$,它的分量是要拟合的参数 $a_0, \ldots, a_{M-1}$。

15.4.1 利用正规方程组求解

将(15.4.3)式中的 χ^2 分别对所有 M 个参数 a_k 求偏导,并令之等于0,将求得 χ^2 的最小值。(15.1.7)式在(15.4.2)式的模型情况下,将得到 M 个方程:

$$0 = \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=0}^{M-1} a_j X_j(x_i) \right] X_k(x_i) \qquad k = 0, \ldots, M-1 \tag{15.4.6}$$

交换求和的顺序,可以将(15.4.6)式写成矩阵方程:

$$\sum_{j=0}^{M-1} \alpha_{kj} a_j = \beta_k \tag{15.4.7}$$

其中,

$$\alpha_{kj} = \sum_{i=0}^{N-1} \frac{X_j(x_i) X_k(x_i)}{\sigma_i^2} \qquad \text{或等价地} \qquad [\alpha] = \mathbf{A}^T \cdot \mathbf{A} \tag{15.4.8}$$

$[\alpha]$是一个 $M\times M$ 矩阵，且

$$\beta_k=\sum_{i=0}^{N-1}\frac{y_iX_k(x_i)}{\sigma_i^2}\qquad \text{或等价地}\qquad [\beta]=\mathbf{A}^T\cdot\mathbf{b} \tag{15.4.9}$$

$[\beta]$是一个长为 M 的向量。

方程组(15.4.6)或(15.4.7)称为最小二乘方问题的**正规方程组**。它们可以利用第 2 章中的标准方法求解得到参数向量 $\mathbf{a}$，比较有名的是 LU 分解法和回代法，Cholesky 分解或 Gauss-Jordan 消元法。用矩阵的形式，正规方程可写为

$$[\alpha]\cdot\mathbf{a}=[\beta]\qquad \text{或}\qquad \left(\mathbf{A}^T\cdot\mathbf{A}\right)\cdot\mathbf{a}=\mathbf{A}^T\cdot\mathbf{b} \tag{15.4.10}$$

逆矩阵 $C_{jk}\equiv[\alpha]^{-1}_{jk}$ 和被估计参数 $\mathbf{a}$ 可能的(更精确地说是**标准的**)不确定度有紧密联系。为了估计这些不确定度，考虑

$$a_j=\sum_{k=0}^{M-1}[\alpha]^{-1}_{jk}\beta_k=\sum_{k=0}^{M-1}C_{jk}\left[\sum_{i=0}^{N-1}\frac{y_iX_k(x_i)}{\sigma_i^2}\right] \tag{15.4.11}$$

有关估计 a_j 的方差可类似(15.2.7)式的形式得出

$$\sigma^2(a_j)=\sum_{i=0}^{N-1}\sigma_i^2\left(\frac{\partial a_j}{\partial y_i}\right)^2 \tag{15.4.12}$$

注意α_{jk}独立于 y_i，因此

$$\frac{\partial a_j}{\partial y_i}=\sum_{k=0}^{M-1}C_{jk}X_k(x_i)/\sigma_i^2 \tag{15.4.13}$$

结果得到

$$\sigma^2(a_j)=\sum_{k=0}^{M-1}\sum_{l=0}^{M-1}C_{jk}C_{jl}\left[\sum_{i=0}^{N-1}\frac{X_k(x_i)X_l(x_i)}{\sigma_i^2}\right] \tag{15.4.14}$$

上式括号内的项恰好是矩阵$[\alpha]$。因为它是矩阵$[C]$的逆矩阵，(15.4.14)式可简化为

$$\sigma^2(a_j)=C_{jj} \tag{15.4.15}$$

换言之，矩阵$[C]$的对角元是拟合参数 $\mathbf{a}$ 的方差(不确定度的平方)。毫无疑问，$[C]$的非对角元 C_{jk} 是 a_j 和 a_k 的协方差(参见(15.2.10)式)。在 15.6 节再讨论这些问题。

我们将利用解正规方程的方法，给出一个实现上面所讨论的一般线性最小二乘方的程序。因为要计算的不仅是待求的向量 $\mathbf{a}$，还包括协方差矩阵$[C]$，所以更方便的是采用 Gauss-Jordan 消元法(2.1 节中的程序 gaussj)实现线性代数运算。在这种情况下的运算量不比 LU 分解法大。但是如果不需要计算协方差矩阵，就可以转而采用 LU 分解法，这将减少 1/3 的线性代数计算量，而无需计算逆矩阵。从理论上看，因为 $\mathbf{A}^T\cdot\mathbf{A}$ 是正定的，Cholesky 分解是最有效的解正规方程的方法。但是在实际中大部分计算时间被用来形成方程的数据循环上，所以 Gauss-Jordan 方法已足够了。

需要提醒注意的是，直接从正规方程得到的最小二乘方问题的解对舍入误差十分敏感。一个更常用的可替代它的方法是设计矩阵 $\mathbf{A}$ 的 QR 分解(参见 2.10 节、11.3 节和 11.6 节)。此方法本质上就是我们在 15.2 节最后所讨论的直线拟合，但为导出必需的公式时没有用到 QR 分解的全部。在本节的后面，将讨论最小二乘方问题的其他难题，解决这类难题的方法是**奇异值分解法**(SVD)，我们将给出它的运算。已经证明 SVD 还能解决舍入误差问题，因此除了"简单"的最小二乘方问题外，还建议采用这种方法。对于这些简单的问题，建议用下面的程序来解正规方程。

下面的程序引入了一个在实际工作中非常有用的技巧。通常我们要决定哪些模型参数 a_k 需要从数据集中拟合，哪些参数应设定为固定常数值，例如被其理论预测的值或在前一次实验测量的值，这通常是具有"艺术"技巧的。因此必须有"冻结"和"不冻结"参数 a_k 的方便手段。在下面的程序中，参数 a_k 的总数目以

ma 表示(前面是用 M 表示)。作为程序的输入量,需要提供数组 ia[0..ma-1]。分量是零则表示要求参数向量 a[0..ma-1]中相应元素保持它们的输入值不变。非零则表示要对参数进行拟合。输出时,任何冻结的参数都有它们的方差,而所有它们的协方差在协方差矩阵中都置为 0。

```
#include "nr.h"

void NR::lfit(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sig, Vec_IO_DP &a,
    Vec_I_BOOL &ia, Mat_O_DP &covar, DP &chisq,
    void funcs(const DP, Vec_O_DP &))
```

给定一组数据点 x[0..ndat-1],y[0..ndat-1],各点的标准差由 sig[0..ndat-1]给出,用求 χ^2 最小值的方法拟合一个函数的某些或全部系数 a[0..ma-1],这个函数和系数 a 之间是线性关系,即 $y=\sum_i a_i \times afunc_i(x)$。输入数组 ia[0..ma-1]中的非零项表示 a 中这些分量是需要拟合的,而零值表示 a 中这些分量应该保持它们的输入值不变。程序返回值 a[0..ma-1],χ^2 = chisq 及协方差矩阵 covar[0..ma-1][0..ma-1](参数保持固定时将返回零协方差)。用户必须提供一个程序 funcs(x,afunc,ma),它返回在 $x=x$ 处计算的 ma 个基函数值的数组 afunc[0..ma-1]。

```
{
    int i,j,k,l,m,mfit=0;
    DP ym,wt,sum,sig2i;

    int ndat=x.size();
    int ma=a.size();
    Vec_DP afunc(ma);
    Mat_DP beta(ma,1);
    for (j=0;j<ma;j++)
        if (ia[j]) mfit++;
    if (mfit == 0) nrerror("lfit: no parameters to be fitted");
    for (j=0;j<mfit;j++) {                  初始化(对称)矩阵。
        for (k=0;k<mfit;k++) covar[j][k]=0.0;
        beta[j][0]=0.0;
    }
    for (i=0;i<ndat;i++) {                  对数据循环以累积计算正规方程的系数。
        funcs(x[i],afunc);
        ym=y[i];
        if (mfit < ma) {                    排除对拟合函数已知部分的依赖关系。
            for (j=0;j<ma;j++)
                if (!ia[j]) ym -= a[j]*afunc[j];
        }
        sig2i=1.0/SQR(sig[i]);
        for (j=0,l=0;l<ma;l++) {
            if (ia[l]) {
                wt=afunc[l]*sig2i;
                for (k=0,m=0;m<=l;m++)
                    if (ia[m]) covar[j][k++] += wt*afunc[m];
                beta[j++][0] += ym*wt;
            }
        }
    }
    for (j=1;j<mfit;j++)                    根据对称性填充对角线以上部分。
        for (k=0;k<j;k++)
            covar[k][j]=covar[j][k];
    Mat_DP temp(mfit,mfit);
    for (j=0;j<mfit;j++)
        for (k=0;k<mfit;k++)
            temp[j][k]=covar[j][k];
    gaussj(temp,beta);                      矩阵求解。
    for (j=0;j<mfit;j++)
        for (k=0;k<mfit;k++)
            covar[j][k]=temp[j][k];
    for (j=0,l=0;l<ma;l++)
        if (ia[l]) a[l]=beta[j++][0];       适合系数 a 的部分解。
    chisq=0.0;
    for (i=0;i<ndat;i++) {                  求拟合的 χ² 值。
        funcs(x[i],afunc);
        sum=0.0;
        for (j=0;j<ma;j++) sum += a[j]*afunc[j];
        chisq += SQR((y[i]-sum)/sig[i]);
    }
    covsrt(covar,ia,mfit);                  将协方差矩阵排序成拟合系数的真实顺序。
}
```

上面程序中最后调用函数 covsrt 的目的，只是将协方差矩阵扩展为 ma × ma 的协方差矩阵，且有正确的行号和列号排列，并使"冻结"的变量的方差和协方差为 0。

函数 covsrt 如下。

```
#include "nr.h"

void NR::covsrt(Mat_IO_DP &covar, Vec_I_BOOL &ia, const int mfit)
在存储器中扩展协方差矩阵 covar，以便考虑那些被保持固定的参数(对后者，返回协方差为 0)。
{
    int i,j,k;

    int ma=ia.size();
    for (i=mfit;i<ma;i++)
        for (j=0;j<i+1;j++) covar[i][j]=covar[j][i]=0.0;
    k=mfit-1;
    for (j=ma-1;j>=0;j--) {
        if (ia[j]) {
            for (i=0;i<ma;i++) SWAP(covar[i][k],covar[i][j]);
            for (i=0;i<ma;i++) SWAP(covar[k][i],covar[j][i]);
            k--;
        }
    }
}
```

15.4.2　运用奇异值分解法求解

在某些情况中，正规方程对于解决最小二乘方问题是足够的。但是，在很多情况下，正规方程非常接近奇异性。在解线性方程时经常会碰到零主元素(例如在 gaussj 中)，在这种情况下，将得不到任何解。或者可能碰到一个非常小的主元素值，在这种情况下，拟合的参数 a_k 将非常大，即脆弱地(且不稳定地)平衡，以致在对拟合函数进行估计时就不可能得到精确解。

这种情况为什么会经常发生呢？原因在于作为一个实验者，你不得不承认，数据不能清楚地将所提供的两个或多个基函数区分开。如果两个这样的函数，或者函数的两个不同的组合，恰巧对数据能进行一样好的或一样坏的拟合，则矩阵[α]不能分辨它们，使两列或两行完全重叠形成奇异值。具有数学讽刺意味的是，在最小二乘方问题中，超定的(数据点大于参数个数)和欠定的(存在参数的二义性组合)同时存在，但这就是通常情况。在复杂问题中，很难预先注意到这种二义性。

现在考虑奇异值分解法(Singular value decomposition，SVD)，这将涉及 2.6 节中的内容，这里不再重述。在超定系统的情况下，SVD 将得出最小二乘方意义下的最佳近似解，参看(2.6.10)式，那正是我们所期望的。在欠定系统的情况下，SVD 得出一个解，它的值(对于我们来说就是 a_k 的值)在最小二乘方意义下是最小的，参看(2.6.8)式。这同样也是我们所要的：当某些基函数的组合和拟合不相干时，SVD 使这些组合得出一些小的无害的值，而不是上升为脆弱的无用的无穷大。

用设计矩阵 **A**((15.4.4)式)和向量 **b**((15.4.5)式)的形式，在(15.4.3)式中求 χ^2 的最小值可写为

$$\text{求}\quad \mathbf{a}\quad \text{使得}\quad \chi^2 = |\mathbf{A}\cdot\mathbf{a}-\mathbf{b}|^2 \ \text{取最小值} \tag{15.4.16}$$

和(2.6.9)式相比，可以看出这恰好是程序 svdcmp 和 svbksb 所要求解的问题。由(2.6.12)式所给出的解可重写成如下形式。如果根据(2.6.1)式，将 **U** 和 **V** 引入 **A** 的 SVD 分解中，如同 svdcmp 中所计算的那样，则设向量 $\mathbf{U}_{(i)}, i=0,\ldots,M-1$ 表示 **U** 的列(每一列都是长度为 N 的向量)；还设向量 $\mathbf{V}_{(i)}, i=0,\ldots,M-1$ 表示 **V** 的列(每一列是长度为 M 的向量)，则最小二乘方问题(15.4.16)式的解(2.6.12)式可写为

$$\mathbf{a}=\sum_{i=0}^{M-1}\left(\frac{\mathbf{U}_{(i)}\cdot\mathbf{b}}{w_i}\right)\mathbf{V}_{(i)} \tag{15.4.17}$$

其中 w_i 如 2.6 节中所示，是由 svdcmp 所计算的奇异值。

(15.4.17)式表示拟合参数 **a** 是 **V** 的列向量的线性组合，其系数为 **U** 的列向量和加权数据向量(15.4.5)式的点积。已经证明拟合参数的标准误(不严格地说，可能的误差)也是 **V** 列向量的线性组合，尽管这个证明超出了现在所讨论的范围。事实上，(15.4.17)式可写成呈现出这些误差的形式

$$\mathbf{a}=\left[\sum_{i=0}^{M-1}\left(\frac{\mathbf{U}_{(i)}\cdot\mathbf{b}}{w_i}\right)\mathbf{V}_{(i)}\right]\pm\frac{1}{w_0}\mathbf{V}_{(0)}\pm\cdots\pm\frac{1}{w_{M-1}}\mathbf{V}_{(M-1)} \tag{15.4.18}$$

其中,每个±号后面是一个标准差。分解成这种形式后,这些标准差是相互独立的(不相关的)。因此它们可以利用平方根形式加在一起。以后我们会看到向量 $\mathbf{V}_{(i)}$ 是拟合参数 a 的误差椭球的主轴(参见 15.6 节)。

拟合参数 a_j 的方差由下式给出:

$$\sigma^2(a_j)=\sum_{i=0}^{M-1}\frac{1}{w_i^2}[\mathbf{V}_{(i)}]_j^2=\sum_{i=0}^{M-1}\left(\frac{V_{ji}}{w_i}\right)^2 \tag{15.4.19}$$

这个结果应该和(15.4.14)式等价。和前面一样,对于协方差的公式,这里可以不加证明地给出:

$$\mathrm{Cov}(a_j,a_k)=\sum_{i=0}^{M-1}\left(\frac{V_{ji}V_{ki}}{w_i^2}\right) \tag{15.4.20}$$

在讨论这一小节时,我们曾提到由于遇到零主元,正规方程无效。但是到现在还没有说明 SVD 是如何解决这一问题的。答案在于:如果某一奇异值 w_i 是零,那么在(15.4.18)式中它的倒数将被置为 0,而不是无穷大(和前面的(2.6.7)式所讨论的比较)。这相当于给拟合参数 a 加一个零倍基函数的线性组合,而不是某个随机大倍数的基函数的线性组合,这些基函数在拟合中是退化的。

还有,如果一个奇异值 w_i 不是 0 但非常小,同样必须定义它的倒数为 0,因为它的表现值可能是人为的舍入误差,而不是一个有意义的数字。一个似乎合理的问题是"怎样小的程度才算是小值"。在本节中,我们用这种方法处理所有与最大奇异值之比值小于机器精度 ϵ 的 N 倍的奇异值(可能认为以 $\sqrt{N}$ 或者某个常数,而不是以 N 作为倍数更好,这就开始涉及与硬件有关的问题了)。

还有另外一个原因使我们处理甚至是附加的奇异值,这些奇异值太大,以至于舍入误差毫无影响。奇异值分解法能使用户鉴别出,碰巧对减小数据集的 χ^2 值作用不大的那些变量的线性组合。对这些奇异值的处理有时可以极大地减小系数的可能误差,而对最小值 χ^2 的增加却可以忽略。在 15.6 节将进一步讨论如何鉴别和处理这些情况。在下面的程序中,可能发生这种处理的奇异值点将被说明。

一般来说,建议读者尽量使用 SVD 方法,而不用正规方程。SVD 的惟一不足在于它要求大小为 $N\times M$ 的额外数组来存储整个设计矩阵。这些存储空间被矩阵 U 重新写过。SVD 还要求存储 $M\times M$ 的矩阵 V。但这用于代替在正规方程中同样大小的系数矩阵。SVD 比解正规方程明显要慢,但是它的最大优点,即它**永远不会无效**(理论上),这弥补了它在运行速度上的不足。

在下面的程序中,矩阵 u,v 和向量 w 作为工作空间的输入。问题的逻辑尺度由 ndata 个数据点和 ma 个基函数(及拟合参数)决定。如果只关心拟合参数 a 的值,则输出的 u,v,w 不包含任何有用信息。如果要得到拟合参数的可能误差,继续阅读下文。

```
#include "nr.h"

void NR::svdfit(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sig, Vec_O_DP &a,
    Mat_O_DP &u, Mat_O_DP &v, Vec_O_DP &w, DP &chisq,
    void funcs(const DP, Vec_O_DP &))
```

给定一组数据点 x[0..ndata-1]及 y[0..ndata-1],它们的标准差由 sig[0..ndata-1]给出,用求 χ^2 最小值的方法决定拟合函数 $y=\sum_i a_i\times \mathrm{afunc}_i(x)$ 的系数 a[0..ma-1]。在这里我们采用 ndatax ma 矩阵的奇异值分解求解拟合方程,正如 2.6 节所述。数组 u[0..ndata-1][0..ma-1],v[0..ma-1][0..ma-1]和 w[1..ma]为输入,提供工作空间;在输出时,它们定义了奇异值分解,并且可用来得到协方差矩阵。程序返回 ma 个拟合参数 a 以及 χ^2 值 chisq。程序使用者必须提供一个程序 funcs(x,afunc),它的返回值是在 $x=$ x 处 ma 个基函数的值,存放于数组 afunc[0..ma-1]中。

```
{
    int i,j;
    const DP TOL=1.0e-13;                    双精度的默认值以及变量换算为顺序单位。
    DP wmax,tmp,thresh,sum;

    int ndata=x.size();
    int ma=a.size();
```

```
    Vec_DP b(ndata),afunc(ma);
    for (i=0;i<ndata;i++) {                    累积计算拟合矩阵的系数。
        funcs(x[i],afunc);
        tmp=1.0/sig[i];
        for (j=0;j<ma;j++) u[i][j]=afunc[j]*tmp;
        b[i]=y[i]*tmp;
    }
    svdcmp(u,w,v);                             奇异值分解。
    wmax=0.0;                                  处理奇异值，在 #defined 语句中给定 TOL，
    for (j=0;j<ma;j++)                         由此处起……
        if (w[j] > wmax) wmax=w[j];
    thresh=TOL*wmax;
    for (j=0;j<ma;j++)
        if (w[j] < thresh) w[j]=0.0;           ……全此。
    svbksb(u,w,v,b,a);
    chisq=0.0;                                 估算 χ² 值。
    for (i=0;i<ndata;i++) {
        funcs(x[i],afunc);
        sum=0.0;
        for (j=0;j<ma;j++) sum += a[j]*afunc[j];
        chisq += (tmp=(y[i]-sum)/sig[i],tmp*tmp);
    }
}
```

将由上面程序所得的矩阵 v 和向量 w 输入下面的短程序，马上会得到拟合参数 a 的方差和协方差。方差的平方根即拟合参数的标准差。此程序是对(15.4.20)式的直接应用，同时遵从约定，即等于 0 的奇异值被视为已经从拟合中删除。

```
#include "nr.h"

void NR::svdvar(Mat_I_DP &v, Vec_I_DP &w, Mat_O_DP &cvm)
为了计算程序 svdfit 所得的 ma 个拟合参数的协方差矩阵 cvm[0..ma-1][0..ma-1]，用由程序 svdfit 所返回的矩阵
v[0..ma-1][0..ma-1]及 w[0..ma-1]调用本程序。
{
    int i,j,k;
    DP sum;

    int ma=w.size();
    Vec_DP wti(ma);
    for (i=0;i<ma;i++) {
        wti[i]=0.0;
        if (w[i] != 0.0) wti[i]=1.0/(w[i]*w[i]);
    }
    for (i=0;i<ma;i++) {          计算协方差矩阵(15.4.20)中和项。
        for (j=0;j<i+1;j++) {
            sum=0.0;
            for (k=0;k<ma;k++)
                sum += v[i][k]*v[j][k]*wti[k];
            cvm[j][i]=cvm[i][j]=sum;
        }
    }
}
```

15.4.3　示例

要提醒注意的是，某些明显的非线性问题可以表示成线性的。例如，一个带有两个参数 a 和 b 的指数函数模型

$$y(x) = a\exp(-bx) \tag{15.4.21}$$

可改写为

$$\log[y(x)] = c - bx \tag{15.4.22}$$

它关于参数 c 和 b 是线性的(当然必须提醒注意，这种变换不能完整地将高斯误差转换成高斯误差)。

再请注意那些“非参数”情况，如下式中

$$y(x) = a\exp(-bx + d) \tag{15.4.23}$$

参数 a 和 d 实际上是不可区分的。这是一个非常好的示例,显示了正规方程和 SVD 的区别,正规方程完全奇异,且 SVD 能找出零奇异值。SVD 将根据"最小二乘方"原理在 a 和 d 之间建立一种平衡(或者,更倾向于取对数后得到它们等价的线性模型)。但是,对于 SVD 无论在何时给出零奇异值,这都是正确的,建议最好还是通过分析鉴别出基函数组中的退化所在处,然后在基函数组中做出适当的删除。

下面是两个用户应用程序 funcs 的示例,第一个例子非常一般,用一组数据拟合普通的多项式。

```
#include "nr.h"

void NR::fpoly(const DP x, Vec_O_DP &p)
一个阶数为 np-1 次多项式的拟合程序,它们的系数放在数组 p[0..np-1]中
{
    int j;

    int np=p.size();
    p[0]=1.0;
    for (j=1;j<np;j++) p[j]=p[j-1]*x;
}
```

第二个例子稍微有点特别,它是用一组数据拟合 nl-1 阶的勒让德多项式。

```
#include "nr.h"

void NR::fleg(const DP x, Vec_O_DP &pl)
用 nl 次勒让德多项式 pl 扩展的拟合程序,多项式的值用如同 5.5 节中的递推关系求得。
{
    int j;
    DP twox,f2,f1,d;

    int nl=pl.size();
    pl[0]=1.0;
    pl[1]=x;
    if (nl > 2) {
        twox=2.0*x;
        f2=x;
        d=1.0;
        for (j=2;j<nl;j++) {
            f1=d++;
            f2+=twox;
            pl[j]=(f2*pl[j-1]-f1*pl[j-2])/d;
        }
    }
}
```

15.4.4 多维拟合

如果要测量的单一变量 y 是多个变量的函数,例如,变量为向量 $\mathbf{x}$,那么基函数将是一个向量 $X_0(\mathbf{x}),\ldots,X_{M-1}(\mathbf{x})$ 的函数。χ^2 优值函数现在是

$$\chi^2=\sum_{i=0}^{N-1}\left[\frac{y_i-\sum_{k=0}^{M-1}a_kX_k(\mathbf{x}_i)}{\sigma_i}\right]^2 \tag{15.4.24}$$

前面的所有讨论都不变,只是 x 被 $\mathbf{x}$ 代替。实际上,如果能容忍程序的切割,则可以直接用上面的程序而不加任何改动:在 lfit 和 svdfit 中数组元素 x[i]的惟一用处是,每个元素都返回到用户提供的程序 funcs,此程序必将给出在那一点的基函数的值。如果在调用 lfit 或 svdfit 前使 x[i]=i,并独立地将数据点的真实向量值(例如在全局变量中)提供给 funcs,则 funcs 就能在开始工作之前由虚构的 x[i]变为真实的数据点。

参考文献和深入阅读

Bevington, P.R., and Robinson, D.K. 1992, *Data Reduction and Error Analysis for the Physical Sciences*, 2nd ed. (New York: McGraw-Hill), Chapter 7.

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9.

15.5 非线性模型

我们现在考虑模型非线性地依赖于一组 M 个未知参数 $a_k(k=0,1,\ldots,M-1)$时的拟合。使用和前一节相同的方法，也就是定义 χ^2 优值函数，通过求它的最小值确定最佳拟合参数。在非线性依赖情况下，求最小值必须迭代进行。对参数给定一个实验值，设计一个改善这个初始实验解的过程。这个过程接着不断重复进行，直到 χ^2 值不再增长(或者不再明显增长)为止。

这个问题和第 10 章中已经解决的求一般非线性函数极小化的问题有何不同呢？表面上有不同，但不是完全不同：当非常接近极小值时，我们希望 χ^2 函数趋近于二次型，可以将它写为

$$\chi^2(\mathbf{a}) \approx \gamma - \mathbf{d}\cdot\mathbf{a} + \frac{1}{2}\mathbf{a}\cdot\mathbf{D}\cdot\mathbf{a} \tag{15.5.1}$$

其中，$\mathbf{d}$ 是长度为 M 的向量，而 $\mathbf{D}$ 是 $M\times M$ 矩阵(和(10.6.1)式比较)。如果这种近似是一个很好的近似，则可以知道怎样从当前的实验参数值 $\mathbf{a}_{\text{cur}}$ 只用一步就跳到极小值 $\mathbf{a}_{\min}$，也就是

$$\mathbf{a}_{\min} = \mathbf{a}_{\text{cur}} + \mathbf{D}^{-1}\cdot\left[-\nabla\chi^2(\mathbf{a}_{\text{cur}})\right] \tag{15.5.2}$$

(和(10.7.4)式相比。)

另一方面，(15.5.1)式对于在 $\mathbf{a}_{\text{cur}}$ 处进行极小化的函数形状是一个比较差的局域近似。在这种情况下，首先要做的是，沿负梯度的方向走一步，如同在最速下降法中那样(参见 10.6 节)。换言之，即

$$\mathbf{a}_{\text{next}} = \mathbf{a}_{\text{cur}} - \text{constant}\times\nabla\chi^2(\mathbf{a}_{\text{cur}}) \tag{15.5.3}$$

其中常数足够小，以保证不偏离下降方向。

为了利用(15.5.2)式或(15.5.3)式，必须能够计算在任意参数集 $\mathbf{a}$ 上 χ^2 函数的梯度。为了利用(15.5.2)式，也必须计算矩阵 $\mathbf{D}$，它是在 $\mathbf{a}$ 取任意值时，χ^2 优值函数的二阶导数矩阵(黑塞矩阵)。

这里所讨论的内容和第 10 章的重要区别在于：在这里没有直接计算黑塞矩阵(Hessian matrix)的方法。只能估算需要极小化的函数以及(在某些情况下)它的梯度。因此必须求助于迭代方法，不仅仅是因为函数是非线性，而且是为了要构造黑塞矩阵的内容。在 10.7 节和 10.6 节中已提到两种不同的构造这些内容的方法。

在这里，情况要简单得多。我们精确地知道 χ^2 的形式，因为它建立在我们自己已经规定的模型函数的基础上，因此黑塞矩阵对我们来说是已知的。只要愿意，便可无约束地采用(15.5.2)式。使用(15.5.3)式的惟一原因在于，(15.5.2)式不能成功地改善拟合，并标志着(15.5.1)式作为一个局域近似不可行。

15.5.1 梯度和黑塞矩阵的计算

需要拟合的模型是

$$y = y(x;\mathbf{a}) \tag{15.5.4}$$

χ^2 优值函数是

$$\chi^2(\mathbf{a}) = \sum_{i=0}^{N-1}\left[\frac{y_i - y(x_i;\mathbf{a})}{\sigma_i}\right]^2 \tag{15.5.5}$$

χ^2 对参数 $\mathbf{a}$ 的梯度，在 χ^2 取极小值时梯度为 0，它的各个分量是

$$\frac{\partial\chi^2}{\partial a_k} = -2\sum_{i=0}^{N-1}\frac{[y_i - y(x_i;\mathbf{a})]}{\sigma_i^2}\frac{\partial y(x_i;\mathbf{a})}{\partial a_k} \qquad k=0,1,\ldots,M-1 \tag{15.5.6}$$

再求一次偏导数得到

$$\frac{\partial^2\chi^2}{\partial a_k\partial a_l} = 2\sum_{i=0}^{N-1}\frac{1}{\sigma_i^2}\left[\frac{\partial y(x_i;\mathbf{a})}{\partial a_k}\frac{\partial y(x_i;\mathbf{a})}{\partial a_l} - [y_i - y(x_i;\mathbf{a})]\frac{\partial^2 y(x_i;\mathbf{a})}{\partial a_l\partial a_k}\right] \tag{15.5.7}$$

可以通过如下定义去掉因子2:

$$\beta_k \equiv -\frac{1}{2}\frac{\partial \chi^2}{\partial a_k} \qquad \alpha_{kl} \equiv \frac{1}{2}\frac{\partial^2 \chi^2}{\partial a_k \partial a_l} \tag{15.5.8}$$

令(15.5.2)式中的$[\alpha] = \frac{1}{2}\mathbf{D}$,利用(15.5.8)式可以把(15.5.2)式重写成一组线性方程组:

$$\sum_{l=0}^{M-1} \alpha_{kl}\,\delta a_l = \beta_k \tag{15.5.9}$$

这个方程组解出增量δa_l,将其与当前的近似相加给出下一步的近似。在最小二乘方问题的文献中,矩阵$[\alpha]$通常称为**曲率矩阵**,等于黑塞矩阵的二分之一。

(15.5.3)式,即最速下降等式,可写为

$$\delta a_l = \text{constant} \times \beta_l \tag{15.5.10}$$

注意,黑塞矩阵(15.5.7)式的分量α_{kl}依赖于基函数对它们参数的一阶及二阶导数。某些处理过程常常不加评注地忽略了二阶导数。我们也忽略二阶导数,但是稍加评注后才这样做。

当(15.5.6)式中的梯度依赖于导数$\partial y / \partial a_k$时,二阶导数便产生了,因此,接下来的求导必定包括有$\partial^2 y / \partial a_l \partial a_k$的项。当二阶微分项是0时自然可以去掉(例如在(15.4.8)式的线性情况下),或者和含有一阶导数的项相比足够小以至于可以忽略。在实际应用中,还有另外一个因为小而可忽略的可能情况:在(15.5.7)式中,乘以二阶导数的项是$[y_i - y(x_i;\mathbf{a})]$。对于一个成功的模型来说,这一项应当只是每一点的随机测量误差。这个误差可正可负,并且一般应该和模型无关。因此,当对i累加求和时,二阶导数项就相互抵消了。

如果模型拟合得不好,或者受到下述偏远点的影响,即这些偏远点不可能通过补上反号的点而被消去,则事实上把二阶导数项包括进来是不稳定的。从现在起,将一直用下式作为α_{kl}的定义:

$$\alpha_{kl} = \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2}\left[\frac{\partial y(x_i;\mathbf{a})}{\partial a_k}\frac{\partial y(x_i;\mathbf{a})}{\partial a_l}\right] \tag{15.5.11}$$

这个表达式和它在线性问题的同类项(15.4.8)式非常相似。必须明白$[\alpha]$是无足轻重的,它对于参数$\mathbf{a}$的最终结果毫无影响,只是影响获得此结果的迭代过程。χ^2取最小值的条件,即对所有的k存在$\beta_k = 0$,和$[\alpha]$如何定义毫无关系。

15.5.2 Levenberg-Marquardt 方法

在Levenberg的一个早期建议的启示下,Marquardt[1]提出了一个非常巧妙的方法,它是在逆黑塞矩阵方法(15.5.9)式的极限和最速下降法(15.5.10)式之间光滑地变动。后一种方法在远离最小值时运用,当接近最小值时它逐渐切换到前一种方法。这种**Levenberg-Marquardt 方法**(也称**Marquardt 方法**)在实际应用中非常有效,并且已经成为非线性最小二乘方问题的标准。

这种方法基于两个基本的但非常重要的思路。考虑(15.5.10)式中的"常数"应该是什么?数量级有多大?由什么来标度它?在梯度中没有关于这些答案的任何信息。梯度只说明了一个坡度,而没有说明这个坡度延伸到多远。Marquardt首先发现,黑塞矩阵的各个分量,尽管在任何精确的形式中毫无用的,也能对问题的数量级精度提供了一些信息。

量χ^2是无量纲的,也就是说它是一个纯粹的数,这可以从它的定义式(15.5.5)中明显看出。另一方面,β_k有和$1/a_k$一样的量纲,$1/a_k$很可能有量纲,例如为cm^{-1},或者千瓦小时,或者其他形式的单位(实际上,β_k的各个分量可能有不同的量纲)。因此在β_k和δa_k之间的比例常数必须具有a_k^2的量纲。检查一下$[\alpha]$的各个分量,会发现只有一个量明显地具有这种量纲,那就是$1/\alpha_{kk}$,即对角元的倒数。因此必须设置常数的标度。但是这种标度可能本身就很大,因此我们引入某个(无量纲的)人为的因子λ,用它除以常数,并尽可能置$\lambda \gg 1$以减少步长。换言之,用下式代替(15.5.10)式:

$$\delta a_l = \frac{1}{\lambda \alpha_{ll}}\beta_l \qquad \text{或} \qquad \lambda \alpha_{ll}\,\delta a_l = \beta_l \tag{15.5.12}$$

上式要求α_{jj}是正的，这可以由定义(15.5.11)式保证——这也是采用(15.5.11)式的另一个原因。

Marquardt 的第二个思路是，如果按照如下规定定义一个新矩阵α'(15.5.12)式和(15.5.9)式可以结合起来：

$$\begin{aligned} \alpha'_{jj} &\equiv \alpha_{jj}(1+\lambda) \\ \alpha'_{jk} &\equiv \alpha_{jk} \qquad (j \neq k) \end{aligned} \tag{15.5.13}$$

则(15.5.12)式和(15.5.9)式均可用下式代替

$$\sum_{l=0}^{M-1} \alpha'_{kl}\,\delta a_l = \beta_k \tag{15.5.14}$$

当λ非常大时，矩阵α'强制成**对角占优**，此时(15.5.14)式和(15.5.12)式等价。另一方面，当λ趋近于零时，(15.5.14)式便回到了(15.5.9)式。

对一组拟合参数 **a** 给定初始猜测值后，建议采用如下的 Marquardt 步骤：

① 计算 $\chi^2(\mathbf{a})$。

② 取一个适中的λ值，比如λ = 0.001。

③ 解线性方程(15.5.14)得到 $\delta\mathbf{a}$，并且计算 $\chi^2(\mathbf{a}+\delta\mathbf{a})$。

④ 如果 $\chi^2(\mathbf{a}+\delta\mathbf{a}) \geq \chi^2(\mathbf{a})$，则将λ**扩大** 10 倍(或者其他相当大的倍数)，再回到③。

⑤ 如果 $\chi^2(\mathbf{a}+\delta\mathbf{a}) < \chi^2(\mathbf{a})$，则将λ**减小** 10 倍，用 $\mathbf{a}+\delta\mathbf{a}$ 代替平凡解 **a**，再回到③。

还必须知道终止迭代的条件。一直迭代到收敛(对于机器精度或舍入极限来说)一般是浪费的，而且没有必要，因为最小值充其量是对参数 **a** 的统计估计。正如我们将在 15.6 节中看到的那样，当 χ^2 的变化量 $\ll 1$时，参数变化在统计上是毫无意义的。

此外，我们经常发现极小值位于一个复杂的拓扑形状的平谷处，参数围绕此极小值徘徊。其原因在于 Marquardt 方法是对正规方程(参见 15.4 节)的推广，因此它也遇到和正规方程一样的问题，即在极小值处的退化、零主元也有可能导致直接失败，但可能性不大。更经常遇到的情况是，一个小的主元将会产生一个大的修正值而被排斥，λ的值随之增加。对于足够大的λ，矩阵$[\alpha']$是正定的，没有很小的主元。因此这种方法的确能远离零主元，但付出的代价是，在一个下降坡度不大的谷地处做最速下降，从而使趋近于最小值比较缓慢。

根据这些考虑我们建议，实际中，在第一次或第二次 χ^2 减少很小量时停止迭代，例如，当 χ^2 绝对值的减小量小于 0.01，或者(以防舍入误差阻止获得这个精度)某些分数值如(10^{-3})时。不要在 χ^2 **增加**的那一步停止迭代；这只说明λ还没有调整得非常合适。

一旦可接受的极小值找到了，就可以令λ = 0，并计算矩阵：

$$[C] \equiv [\alpha]^{-1} \tag{15.5.15}$$

如同前面所讨论的一样，上式可用来计算拟合参数 **a** 的标准误的协方差矩阵(参看下节)。

下面的一对函数程序准确地体现了非线性参数估计的 Marquardt 方法。程序的大部分结构和在 15.4 节中所有的 fit 一致。特别要注意的是，在数组 ia[0..ma−1]中必须输入分量值 1 或 0，它们分别对应于数组 a[0..ma−1]中相应的参数值是需要被拟合，或是需要保持它们的输入值不变。

程序 mrqmin 执行 Marquardt 方法的一个迭代。在第一次调用(一次)它时令 alamda < 0，它标志着程序的开始。为了下一次的迭代，alamda 在第一次以及所有后续调用中设置成λ的标识值；a 和 chisq 作为迄今找到的最佳估计参数以及对应的 χ^2 返回。当对收敛程度满意时，在进行最后一次调用前令 alamda 为 0。矩阵 alpha 和 covar(在所有先前的调用中它们被用做工作空间)，于是被设置成收敛参数值的曲率矩阵和协方差。参量 alpha，a 和 chisq 在两次调用之间不能变化。alamda 除了在最后一次调用时设为零外，任何其他两次调用之间也保持不变。当遇到使 χ^2 增加的一个步骤时，chisq 和 a 返回它们的输入值(最佳值)，但是 alamda 将设置为一个增大的值。

程序 mrqmin 要调用程序 mrqcof 来计算矩阵$[\alpha]$(参见(15.5.11)式)和向量 β((15.5.6)式和(15.5.8)式)。

在 mrqcof 中又要调用用户提供的程序 funcs(x,a,y,dyda),它用来计算输入值 $x \equiv x_i$ 及 $a \equiv \mathbf{a}$ 时的模型函数 $y \equiv y(x_i;\mathbf{a})$ 以及微分向量 $dyda \equiv \partial y/\partial a_k$

```
#include "nr.h"

void NR::mrqmin(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sig, Vec_IO_DP &a,
    Vec_I_BOOL &ia, Mat_O_DP &covar, Mat_O_DP &alpha, DP &chisq,
    void funcs(const DP, Vec_I_DP &, DP &, Vec_O_DP &), DP &alamda)
```

采用 Levenberg-Marquardt 方法来拟合一组数据点 x[0..ndata-1],y[0..ndata-1] 和某一函数拟合的 χ^2 值,这些数据点具有标准差为 sig[0..ndata-1],而非线性函数依赖的 ma 个系数为 a[0..ma-1]。输入数组 ia[0..ma-1]的非零项表明数组 a 中的那些分量需要进行拟合,而零项表明 a 中的那些分量应该保持它们的输入量不变。程序返回当前参数 a[0..ma-1]的最佳拟合值和 χ^2 = chisq 值。在大多数迭代中,数组 covar[0..ma-1][0..ma-1],alpha[0..ma-1][0..ma-1]用做工作空间。用户提供一个程序 funcs(x,a,yfit,dyda),它能拟合函数 yfit,以及它关于拟合参数 a 在 x 处的导数 dyda[0..ma-1]。第一次调用时,对参数 a 提供一个初始猜测值,并初始赋值 alamda<0(然后置 alamda=.001)。如果一步成功,则 chisq 变得更小且 alamda 下降到 1/10。如果一步失败,则 alamda 增加到 10 倍。必须反复调用本程序,直到收敛。然后,用 alamda=0 进行最后一次调用,使得在 covar[0..ma-1][0..ma-1]中返回协方差矩阵,alpha 返回曲率矩阵(保持固定的参数返回零协方差)。

```
{
    static int mfit;
    static DP ochisq;
    int j,k,l;

    int ma=a.size();
    static Mat_DP oneda(ma,1);
    static Vec_DP atry(ma),beta(ma),da(ma);
    if (alamda < 0.0) {                    初始赋值。
        mfit=0;
        for (j=0;j<ma;j++)
            if (ia[j]) mfit++;
        alamda=0.001;
        mrqcof(x,y,sig,a,ia,alpha,beta,chisq,funcs);
        ochisq=chisq;
        for (j=0;j<ma;j++) atry[j]=a[j];
    }
    Mat_DP temp(mfit,mfit);
    for (j=0;j<mfit;j++) {                 通过增加对角元,改变线性化的拟合矩阵。
        for (k=0;k<mfit;k++) covar[j][k]=alpha[j][k];
        covar[j][j]=alpha[j][j]*(1.0+alamda);
        for (k=0;k<mfit;k++) temp[j][k]=covar[j][k];
        oneda[j][0]=beta[j];
    }
    gaussj(temp,oneda);                    矩阵解。
    for (j=0;j<mfit;j++) {
        for (k=0;k<mfit;k++) covar[j][k]=temp[j][k];
        da[j]=oneda[j][0];
    }
    if (alamda == 0.0) {                   一旦收敛,计算协方差矩阵。
        covsrt(covar,ia,mfit);
        covsrt(alpha,ia,mfit);             还扩展 α 到它的最大阶数。
        return;
    }
    for (j=0,l=0;l<ma;l++)                 试验成功了吗?
        if (ia[l]) atry[l]=a[l]+da[j++];
    mrqcof(x,y,sig,atry,ia,covar,da,chisq,funcs);
    if (chisq < ochisq) {                  成功则接收新的解。
        alamda *= 0.1;
        ochisq=chisq;
        for (j=0;j<mfit;j++) {
            for (k=0;k<mfit;k++) alpha[j][k]=covar[j][k];
                beta[j]=da[j];
        }
        for (l=0;l<ma;l++) a[l]=atry[l];
    } else {                               失败则增加 alamda 并返回。
        alamda *= 10.0;
        chisq=ochisq;
    }
}
```

注意其中使用了 15.4 节中的程序 covsrt。这仅仅是为了将协方差矩阵 covar 重排成所有 ma 个参数的顺序。上面的程序中也使用了以下函数。

```
#include "nr.h"

void NR::mrqcof(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sig, Vec_I_DP &a,
    Vec_I_BOOL &ia, Mat_O_DP &alpha, Vec_O_DP &beta, DP &chisq,
    void funcs(const DP, Vec_I_DP &,DP &, Vec_O_DP &))
本程序被程序 mrqmin 用来计算(15.5.8)式中线性化 f 的拟合矩阵 alpha 和向量 beta,并计算 χ^2 。
{
    int i,j,k,l,m,mfit=0;
    DP ymod,wt,sig2i,dy;

    int ndata=x.size();
    int ma=a.size();
    Vec_DP dyda(ma);
    for (j=0;j<ma;j++)
        if (ia[j]) mfit++;
    for (j=0;j<mfit;j++) {              初始化(对称的)alpha 和 beta。
        for (k=0;k<=j;k++) alpha[j][k]=0.0;
        beta[j]=0.0;
    }
    chisq=0.0;
    for (i=0;i<ndata;i++) {             对所有数据循环求和。
        funcs(x[i],a,ymod,dyda);
        sig2i=1.0/(sig[i]*sig[i]);
        dy=y[i]-ymod;
        for (j=0,l=0;l<ma;l++) {
            if (ia[l]) {
                wt=dyda[l]*sig2i;
                for (k=0,m=0;m<l+1;m++)
                    if (ia[m]) alpha[j][k++] += wt*dyda[m];
                beta[j++] += dy*wt;
            }
        }
        chisq += dy*dy*sig2i;           寻找 χ^2。
    }
    for (j=1;j<mfit;j++)                填补对称的一边。
        for (k=0;k<j;k++) alpha[k][j]=alpha[j][k];
}
```

15.5.3 示例

下面的函数 fgauss 是一个用户提供的函数 funcs 的示例。应用上面的程序 mrqmin(依次用到 mrqcof, covsrt 和 gaussj)可以拟合下面的模型:

$$y(x) = \sum_{k=0}^{K-1} B_k \exp\left[-\left(\frac{x-E_k}{G_k}\right)^2\right] \tag{15.5.16}$$

它是 K 个高斯函数的和,每个函数具有可变的位置、振幅和宽度。把各参数按 $B_0, E_0, G_0, B_1, E_1, G_1, \ldots, B_{k-1}, E_{K-1}, G_{k-1}$ 的顺序存储。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::fgauss(const DP x, Vec_I_DP &a, DP &y, Vec_O_DP &dyda)
y(x;a)是 na/3 个高斯函数式(15.5.16)的和。每个高斯函数的振幅、中心位置以及宽度连续存储在 a 中:a[3k] = B_k,
a[3k+1] = E_k, a[3k+2] = G_k, k = 0, ..., na/3 - 1。数组的维数为 a[0..na-1], dyda[0..na-1]。
{
    int i;
    DP fac,ex,arg;

    int na=a.size();
    y=0.0;
    for (i=0;i<na-1;i+=3) {
        arg=(x-a[i+1])/a[i+2];
        ex=exp(-arg*arg);
```

```
            fac=a[i]*ex*2.0*arg;
            y += a[i]*ex;
            dyda[i]=ex;
            dyda[i+1]=fac/a[i+2];
            dyda[i+2]=fac*arg/a[i+2];
        }
    }
```

15.5.4 非线性最小二乘方法的更先进方法

Levenberg-Marquardt 算法可以认为是一种模型-置信区域方法，应用于最小二乘方函数的特殊情况中求极小值(参见9.7节和本节参考文献[2])。由 Moré[3]编制的这类代码可以在 MINPACK[4]中找到。另外一种非线性最小二乘方算法，保留了在 Levenberg-Marquardt 方法中放弃的二阶导数项。任何时候保留这一项将更好。这些方法被称为"全牛顿型"方法，并且认为它比 Levenberg-Marquardt 方法更稳健，也更复杂。这些方法的一个实现是代码 NL2SOL[5]。

参考文献和深入阅读

Bevington, P.R., and Robinson, D.K. 1992, *Data Reduction and Error Analysis for the Physical Sciences*, 2nd ed. (New York: McGraw-Hill), Chapter 8.

Marquardt, D.W. 1963, *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, pp. 431–441. [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter III.2 (by J.E. Dennis).

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, reprinted 1996 (Philadelphia: S.I.A.M.). [2]

Moré, J.J. 1977, in *Numerical Analysis*, Lecture Notes in Mathematics, vol. 630, G.A. Watson, ed. (Berlin: Springer-Verlag), pp. 105–116. [3]

Moré, J.J., Garbow, B.S., and Hillstrom, K.E. 1980, *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL-80-74. [4]

Dennis, J.E., Gay, D.M, and Welsch, R.E. 1981, *ACM Transactions on Mathematical Software*, vol. 7, pp. 348–368, *op. cit.*, pp. 369–383. [5]

15.6 被估模型参数的置信界限

在本章中，已经好几次论述 M 个被估参数 $\mathbf{a}$ 的标准误或不确定度了。不仅给出了每对参数间的协方差公式((15.2.10)式，标出了下面的(15.4.15)式、(15.4.20)式及(15.5.15)式)，还给出了计算单个参数的标准差或方差的公式((15.2.9)式、(15.4.15)式及(15.4.19)式)。

在本节中，要更明确地表述这些量的不确定度的精确含义，还要对如何定量地估计拟合参数的置信界限进行深入的讨论。这个问题的讨论涉及到一些技巧，因此将尽可能精确地描述，即使未做任何证明。

图 15.6.1 呈现了"测量"一组参数的实验的概念性方法。某些基本的真实的参数 $\mathbf{a}_{true}$ 集合隐藏在实验背后，无法知道。这些真实的参数是通过统计方法实现的，如同测量数据那样，和被测数据集一样带有随机测量误差，我们将用 $\mathcal{D}_{(0)}$ 标记数据集。数据集 $\mathcal{D}_{(0)}$ 对实验者来说**确实**是可知的。通过求 χ^2 最小值或者其他的技巧拟合数据成为一个模型，获得参数测量值(即拟合值)，在这里用 $\mathbf{a}_{(0)}$ 标记。

因为测量误差具有随机分量，$\mathcal{D}_{(0)}$ 不是真实参数值 $\mathbf{a}_{true}$ 的惟一实现。相反，存在真实参数的其他无穷多个实现，可以把它们作为"假设的数据集"，其中每一组都可能已是一组测量结果，但恰巧又不是。将这些数据集用 $\mathcal{D}_{(1)}, \mathcal{D}_{(2)}, \ldots$ 标记。每一个数据集如果实现了，将分别给出稍微有些不同的拟合参数集 $\mathbf{a}_{(1)}, \mathbf{a}_{(2)}, \ldots$，因此这些参数集 $\mathbf{a}_{(i)}$ 将在所有可能的参数集 $\mathbf{a}$ 构成的 M 维空间中呈现出某种概率分布。实际测量到的一组参数 $\mathbf{a}_{(0)}$ 是从这个分布中导出的一个数。

比 $\mathbf{a}_{(i)}$ 的概率分布更有意思的是差 $\mathbf{a}_{(i)} - \mathbf{a}_{true}$ 的分布。这个分布和前面所讨论的分布的不同之处在于，一开始它就把无法得知的真实值考虑了进去。如果知道了这个分布，则将知道实验的测量值 $\mathbf{a}_{(0)}$ 的定量不确定度。

因此，工作目的是，在不知道 $\mathbf{a}_{true}$ 及不可能获得假设数据集的无穷多个全域时，寻找估计或近似 $\mathbf{a}_{(i)}-\mathbf{a}_{true}$ 的概率分布的方法。

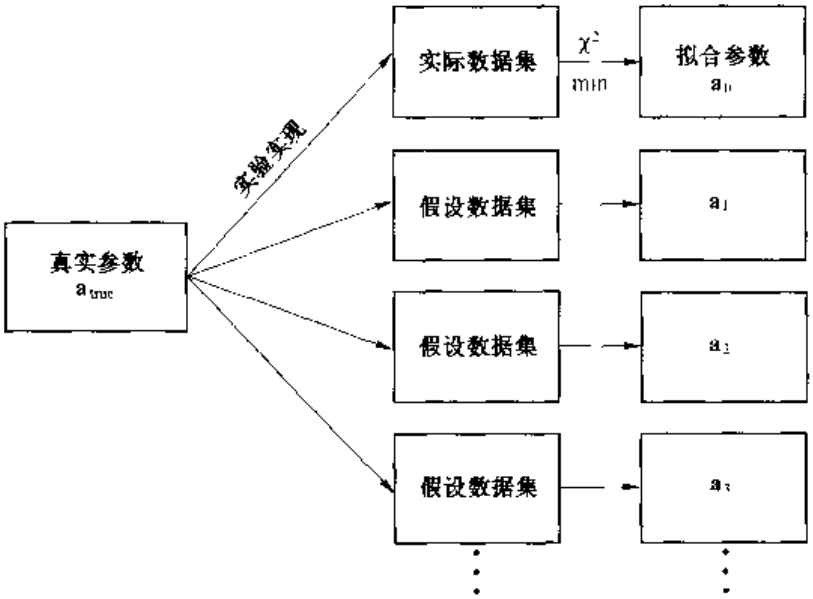


图 15.6.1　来自基本模型的数据集的统计全域。真实参数 $\mathbf{a}_{true}$ 通过一组数据得以实现，从这组数据拟合（观察）出参数 $\mathbf{a}_0$。如果实验重复多次，那么将得到新的数据集和新的拟合参数值

15.6.1　合成数据集的蒙特卡罗模拟

尽管测量的参数集 $\mathbf{a}_{(0)}$ 不是真实值，但可以设想一个虚构的空间，在这个空间里 $\mathbf{a}_{(0)}$ 视为真实值。因为总希望测量的参数值不要错误太大，所以希望虚构空间和 $\mathbf{a}_{true}$ 所在的实际空间相差不太远。特别是，假设在虚构空间中 $\mathbf{a}_{(i)}-\mathbf{a}_{(0)}$ 的概率分布的形状和真实空间中 $\mathbf{a}_{(i)}-\mathbf{a}_{true}$ 的概率分布的形状相同或者接近相同。注意没有假设 $\mathbf{a}_{(0)}$ 和 $\mathbf{a}_{true}$ 相等，它们肯定不相等。只假设引入实验和数据分析的、作为 $\mathbf{a}_{true}$ 函数的随机误差变化不快，因此 $\mathbf{a}_{(0)}$ 可作为一个合理的代替者。

现在在虚构空间里我们**完全**有能力计算 $\mathbf{a}_{(i)}-\mathbf{a}_{(0)}$ 的分布（参见图 15.6.2）。当给定一组假设的参数集 $\mathbf{a}_{(0)}$ 时，如果了解产生数据的过程，则通常可以将这些参数的"综合"实现的集合**模拟**成"合成数据集"。这个过程就是，从适合的分布中取随机数（参见 7.2 节～7.3 节），以便模拟对于基本过程和仪器中测量误差的正确理解，通过这样的随机抽取，构造了一些数据集，这些数据集和实测数据点的个数相等，并和所有控制（独立）变量的值精确地相同，如同实际的数据集的 $\mathcal{D}_{(0)}$ 一样，称这些模拟的数据集为 $\mathcal{D}^S_{(1)}, \mathcal{D}^S_{(2)}, \ldots$。根据构造，这些数据集和 $\mathbf{a}_{(0)}$ 的统计关系恰好与数据集 $\mathcal{D}_{(i)}$ 和 $\mathbf{a}_{true}$ 间具有的关系相同（若读者还不知道怎样对正在测量的数据做一个可信的模拟，请阅读下面的讨论）。

其次，对每个 $\mathcal{D}^S_{(j)}$ 精确地执行同样的参数估算过程，即 χ^2 最小值法，它与为得到参数 $\mathbf{a}_{(0)}$ 而对实际数据执行的过程完全相同，由此得到模拟的被测参数 $\mathbf{a}^S_{(1)}, \mathbf{a}^S_{(2)}, \ldots$。每个模拟的被测参数集产生一个点 $\mathbf{a}^S_{(i)}-\mathbf{a}_{(0)}$。模拟足够多的数据集和导出足够多的模拟被测参数，然后，就可以在 M 维空间中画出所期望的概率分布。

事实上，人们以这种方式进行**蒙特卡罗模拟**的能力已经使当代实验科学的许多领域发生了变革。人们不仅仅能够用一种非常精确的方法来表征参数估算的误差的特征，而且还能够在计算机上试验出不同的参数估算方法，或者试验出不同的数据约化技术，以及根据任何期望的准则进行探索使结果的不确定性达到极小。建议掌握实施统计的蒙特卡罗模拟的技能。

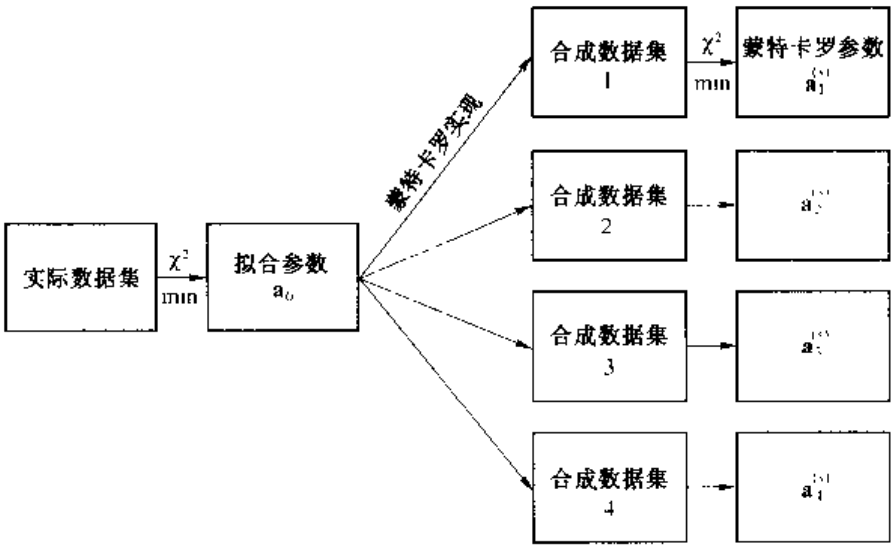


图 15.6.2 实验的蒙特卡罗模拟。从实际实验得到的拟合参数作为真实参数的替代物，计算机生成的随机数用于模拟许多合成数据集。对每个这种数据集进行分析获得它的拟合参数。于是，对这些拟合参数围绕(已知)替代的真实参数的分布进行研究

15.6.2 快速粗糙的蒙特卡罗方法:靴带法

当对基础的处理过程知之不多，或者对测量误差的性质不了解以致不能进行合理的蒙特卡罗模拟时，本节提供了一个强有力的技术。假设，数据组由 N 个**独立且分布完全相同的**(independent and identically distributed，简称 iid)"数据点"组成。每个数据点可能由几个数组成，例如一个或多个控制变量(例如在确定要测量的范围内均匀分布)以及相关的测量值。独立且完全相同的分布意味着数据点的先后顺序不会影响获得拟合参数 $\mathbf{a}$ 的处理。例如，(15.5.5)式即 χ^2 的和式与数据点相加顺序无关。最简单的例子是被测量的平均值或者被测量的某些函数的平均值。

靴带法(bootstrap method)[1]用 N 个数据点组成的真实数据集 $\mathcal{D}_{(0)}^{\delta}$ 产生任何数目的合成数据集 $\mathcal{D}_{(1)}^{\delta}$，$\mathcal{D}_{(2)}^{\delta}$，...，每个数据集也有 N 个数据点。这个过程就是简单地用**替换法**从 $\mathcal{D}_{(0)}^{\delta}$ 一次导出 N 个数据点。因为是替换，所以不必每次都回到原始数据集。在得到的数据集中，原始数据随机的一部分，典型的是 $\sim 1/e \approx 37\%$ 被**复制的**原始点所替代。现在，和前面讨论的完全一样，对这些数据集进行与实际数据处理完全相同的估计过程，得到一组模拟的被测参数 $\mathbf{a}_{(1)}^{\delta}$，$\mathbf{a}_{(2)}^{\delta}$，...，它们围绕 $\mathbf{a}_{(0)}$ 的分布和 $\mathbf{a}_{(0)}$ 围绕 $\mathbf{a}_{true}$ 的分布非常接近。

看起来似乎什么也没有得到，但实际上，为了使靴带法被统计学家接受，已花了十多年的时间。到现在，很多定理的证明都说明了靴带法的可靠性(参见本节参考文献[2])。在靴带法中隐含的基本思想是，实际数据组在被测量的数值处看做是由 δ 函数组成的概率分布。在大多数情况下，这是最好的甚至是惟一可能的对基本概率分布的估计。但是人们通常只简单地将**这种**分布作为蒙特卡罗模拟的基础。

注意，当靴带法的这种独立完全相同分布的假设不满足的情况。例如，对一些控制变量的空间等间隔地进行测量，通常可以排除这些点是独立完全相同分布的假设，它们是在测量的范围内均匀分布。但是，$\mathbf{a}$ 的某些估计(例如有关傅里叶方法)可能对出现在网格上的点特别敏感。在这种情况下，靴带法将给出错误的分布。还必须注意，当估计量看做是在 N 个数据点中的小尺度量的块，或者排序数据并且寻找相继的差异时，显然靴带法也不适用于这些情况(证明这些方法的公理还是正确的，但是其中的一些技巧性假设被这些实例破坏了)。

然而，在大多数情况下，靴带法能容易且非常迅速地产生被估参数集中的误差的蒙特卡罗估算。

15.6.3 置信界限

为了掌握分布的概貌，我们通常采用**置信界限**的形式，而不是提出在参数估算中的误差概率分布的所有细节。整个概率分布是定义在参数 $\mathbf{a}$ 的 M 维空间上的一个函数。一个**置信区域**(或者**置信区间**)恰好是 M

维空间的一个区域(希望是一个小的区域),这个区域包含全部概率分布的一定百分比(通常希望是大的百分比)。例如可以就一个置信区域说:"真实参数有 99%的机会落入围绕被测值的这个区域内。"

值得强调的是,作为一个实验者,必须同时获知**置信水平**(在上面的例子中为 99%)及置信区域的形状。惟一的要求是,置信区域必须包括所陈述的概率百分比。但是,一些百分比是科学中惯用的,它们是:68.3%(可引用的最低置信水平),90%,95.4%,99%,99.73%。更高的置信水平可方便地写为"99.9…9"。至于置信区域的形状,显然想要的是,一个以测量值 $\mathbf{a}_{(0)}$ 为中心的紧凑的合理区域,因为找一个置信界限的全部目的在于确定被测值的可信度。在一维问题中,置信区域是一个以被测值为中心的线段;在高维问题中,经常用到椭圆或椭球。

读者可能怀疑数字 68.3%,95.4%,99.73%,以及椭球的应用可能和正态分布有联系。从历史的角度看这是完全正确的,但是如今并不总是和正态分布有联系。一般来说,参数概率分布不是正态分布,前面用做置信水平的数,完全是出于惯例。

图 15.6.3 画出了在 $M=2$ 的情形下一个可能的概率分布图。该图显示了三个可能有用的不同的置信区间,它们都有相同的置信水平。两条竖直线围住的一条窄带(水平区间),它代表在不考虑 a_1 值时变量 a_0 的 68%的置信区间。同样,两条水平线为 a_1 围住了 68%置信区间。椭圆则代表了 a_0 和 a_1 联合的 68%的置信区间。注意,为了围住和两条带形区域相同的概率,椭圆必须扩展到两条带形之外(在后面还要返回来讨论这一点)。

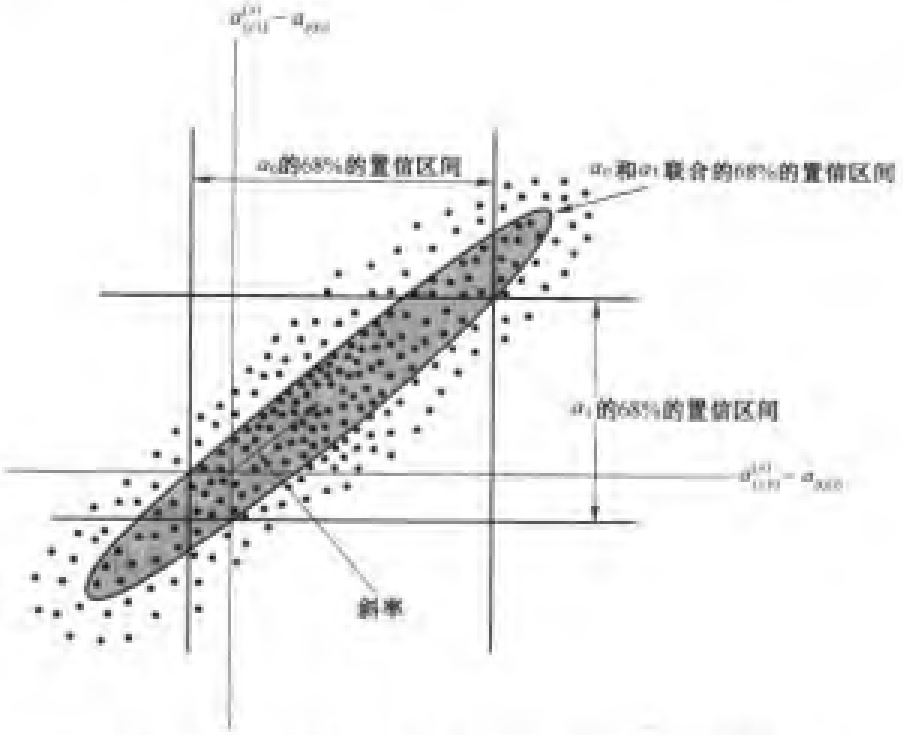


图 15.6.3　一维和二维的置信区间。测量数据点的相同部分(这里是 68%)位于:(i)两条垂直线间;(ii)两条水平线之间;(iii)椭圆内

15.6.4　作为置信界限常数 χ^2 边界

若用于估计参数 $\mathbf{a}_{(0)}$ 的方法是 χ^2 最小值方法,正如我们在本章前几节的讨论,则对于置信区间形状有一个自然的选择,它的应用很普遍。对于观测到的数据集 $\mathcal{D}_{(0)}$ 而言,χ^2 在 $\mathbf{a}_{(0)}$ 处取极小值,称这个极小值为 $\chi^2_{\min}$。如果参数向量 $\mathbf{a}$ 的值偏离 $\mathbf{a}_{(0)}$,则 χ^2 将增加。使 χ^2 增加值小于 $\Delta\chi^2$ 的区域定义了围绕 $\mathbf{a}_{(0)}$ 的 M 维空间的置信区域。如果 $\Delta\chi^2$ 是一个大的数值,则这个区域也较大;如果 $\Delta\chi^2$ 比较小,则这个区域也较小。如果想要这个区域包括上面所定义的 $\mathbf{a}$ 的概率分布的不同百分比,例如 68%,90%等,则必须选择不同的 $\Delta\chi^2$。这些区域可看做参数 $\mathbf{a}_{(0)}$ 的置信区域。

人们常常对全部 M 维置信区域不感兴趣，而对某些数目较少的 ν 个参数的单独置信区域感兴趣。例如，人们可能对每个参数的单独的置信区间感兴趣(如在图 15.6.3 中的带形)，在这种情况下 $\nu=1$。此时，M 维参数空间中的 ν 维子参数空间的自然置信区域，是固定 $\Delta\chi^2$ 定义的 M 维区域在我们感兴趣的 ν 维空间中的**投影**。在图 15.6.4 中 $M=2$ 的情况下，标出了不同 $\Delta\chi^2$ 值对应的区域。相应于由 $\Delta\chi^2=1$ 限定的区域，变量 a_i 的一维置信区间位于直线 A 和 A' 之间。

注意，采用的是高维区域在低维空间中的投影，而不是采用它们的交。在图 15.6.4 中，相交部分是 Z 和 Z' 之间的带形，它永远不会被使用，在图中加以显示的目的只是为了提醒读者不要和投影混淆。

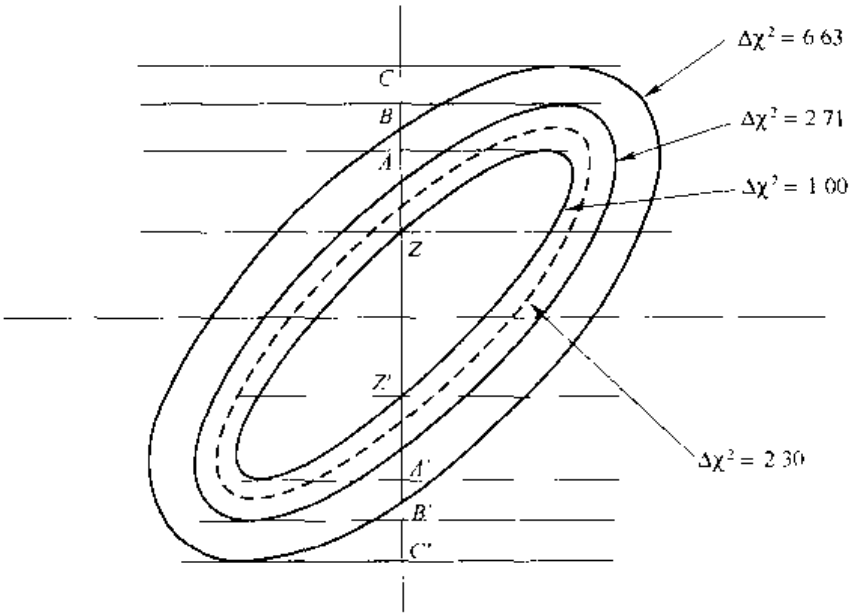


图 15.6.4 χ^2 值大于拟合极小值所对应的椭圆置信区域。将 $\Delta\chi^2=1.00, 2.71, 6.63$ 所对应的实线椭圆投影到一维轴上，得到区间 AA'、BB'、CC'。这些区间(不是椭圆本身)包含了 68.3%，90%，99% 的正态分布数据。包含 68.3% 的正态分布数据的椭圆用虚线表示，它对应的 $\Delta\chi^2$ 是 2.30。对其他的数据，请参看相关表格

15.6.5 正态情况下参数的概率分布

读者可能要问为什么迄今为止，我们的讨论还和 χ^2 拟合过程的误差估计没发生任何联系。最明显的是协方差矩阵 C_y。原因在于：χ^2 极小化方法是一个非常有用的参数估计方法，即使测量误差不是正态分布也适用。如果 χ^2 参数估计成为最大似然估计(参见 15.1 节)，则要求误差是正态分布，此时人们通常愿意放弃最大似然估计的这个特性，而运用相对方便的 χ^2 处理。只有在极端的情况下，测量误差分布带有很大的"尾巴"时，人们才放弃 χ^2 极小化方法，而采用更加稳健的技术，我们将在 15.7 节讨论这一方法。

不过，只有在测量误差实际上呈正态分布(或者达到这种程度)时，由 χ^2 极小化方法导出的形式协方差矩阵才有明确的量化解释。在误差分布为非正态时，可"允许"：

- 通过求 χ^2 极小值拟合参数。
- 用常数 $\Delta\chi^2$ 的一个轮廓线作为置信区域的边界线。
- 用蒙特卡罗模拟或者详细的解析计算确定哪一条 $\Delta\chi^2$ 轮廓线对于所要的置信水平是正确的。
- 把协方差矩阵 C_y 作为"拟合的形式协方差矩阵"给出。

但不允许：

- 使用在正态误差情况下给出的公式，这些公式建立了 $\Delta\chi^2$、C_y 以及置信水平之间的定量关系。

这里列举在以下条件不成立时的关键定理：(i)测量误差是正态分布，以及要么(ii)模型对于它的参数是线

性的；要么(iii)样本数目足够大，以致拟合参数 $\mathbf{a}$ 的不确定度没有扩展到区域以外。在此区域中模型可用一个合适的线性模型来代替(注意条件(iii)并没有排除使用一个非线性拟合程序(例如 mqrfit)来寻找拟合参数值)。

定理 A　$\chi^2_{\min}$ 的分布是自由度为 $N-M$ 的 χ^2 分布，这里 N 是数据点的个数，M 是拟合参数的个数。这是一个估算模型拟合优度的基本定理，如同在 15.1 节中所讨论的那样。首先将这条定理列出来是为了提醒读者，除非拟合优度是可信的，否则整个参数估计过程是不可靠的。

定理 B　如果 $\mathbf{a}^s_{(j)}$ 是从带有实际参数 $\mathbf{a}_{(0)}$ 的模拟数据集的全域中导出的，则 $\delta\mathbf{a}\equiv\mathbf{a}^s_{(j)}-\mathbf{a}_{(0)}$ 的概率分布是多元正态分布：

$$P(\delta\mathbf{a})\ da_0\ldots da_{M-1}=\text{常数}\times\exp\left(-\frac{1}{2}\delta\mathbf{a}\cdot[\alpha]\cdot\delta\mathbf{a}\right)da_0\ldots da_{M-1}$$

其中，$[\alpha]$ 是在(15.5.8)式中定义的曲率矩阵。

定理 C　如果 $\mathbf{a}^s_{(j)}$ 是从带有实际参数 $\mathbf{a}_{(0)}$ 的模拟数据集的全域中导出的，则量 $\Delta\chi^2\equiv\chi^2(\mathbf{a}_{(j)})-\chi^2(\mathbf{a}_{(0)})$ 的分布呈现为自由度为 M 的 χ^2 分布。在这里，χ^2 值都用固定的(实际的)数据集 $\mathcal{D}_{(0)}$ 估算。这个定理在特定的 $\Delta\chi^2$ 值和被这些值包围成 M 维区域的概率分布部分，也即 M 维置信区域的置信水平之间建立了联系。

定理 D　假设 $\mathbf{a}^s_{(j)}$ 是从模拟数据集的全域中导出的(如同上面一样)，还假设它的前 ν 个分量 $a_0,\ldots,a_{\nu-1}$ 保持固定值，剩下的 $M-\nu$ 个分量使 χ^2 取极小值。称这个极小值为 χ^2_ν。则 $\Delta\chi^2_\nu\equiv\chi^2_\nu-\chi^2_{\min}$ 是自由度 ν 的 χ^2 分布。如果查看图 15.6.4，就会明白这个定理将被投影的 $\Delta\chi^2$ 区域和置信水平联系了起来。在图中，一个在 a_1 保持固定且允许 a_0 变化、极小化 χ^2 的点能找出一个椭圆，椭圆的顶部和底部边界和常数 a_1 对应直线相切，这样的点也因此是将椭圆投影到较低维空间的线。

作为第一个例子，考虑 $\nu=1$ 的情况，其中要找某一个单参数(比如 a_0)的置信区间。注意自由度 $\nu=1$ 的 χ^2 分布和单个正态分布量的平方分布相等。$\Delta\chi^2_\nu<1$ 发生的概率为 68.3%(对于正态分布为 1-σ)，$\Delta\chi^2_\nu<4$ 发生的概率为 95.4%(对于正态分布为 2-σ)，$\Delta\chi^2_\nu<9$ 发生的概率为 99.73%(对于正态分布即3-σ)，等等。用这种方法就能找到和所想要的置信水平对应的 $\Delta\chi^2_\nu$(其他的数据在表 15.6.1 中给出)。

表 15.6.1　作为置信水平和自由度的函数 $\Delta\chi^2$

	ν					
p	1	2	3	4	5	6
68.3%	1.00	2.30	3.53	4.72	5.89	7.04
90%	2.71	4.61	6.25	7.78	9.24	10.6
95.4%	4.00	6.17	8.02	9.70	11.3	12.8
99%	6.63	9.21	11.3	13.3	15.1	16.8
99.73%	9.00	11.8	14.2	16.3	18.2	20.1
99.99%	15.1	18.4	21.1	23.5	25.7	27.8

设 $\delta\mathbf{a}$ 表示参数变化，它的第一个分量是任意的 δa_0，但是其余分量选择成使 $\Delta\chi^2$ 取极小值。于是可以应用定理 D。$\Delta\chi^2$ 值通常由下式给出：

$$\Delta\chi^2=\delta\mathbf{a}\cdot[\alpha]\cdot\delta\mathbf{a}\tag{15.6.1}$$

它由将(15.5.8)式用到 $\chi^2_{\min}$ 处而得出，其中 $\beta_k=0$。因为根据假设，$\delta\mathbf{a}$ 除了第 0 个分量外，其余分量使 χ^2 取最小值，所以正规方程(15.5.9)中第一个到第 $M-1$ 个方程依然有效。因此，(15.5.9)式的解是

$$\delta\mathbf{a}=[\alpha]^{-1}\cdot\begin{pmatrix}c\\0\\\vdots\\0\end{pmatrix}=[C]\cdot\begin{pmatrix}c\\0\\\vdots\\0\end{pmatrix}\tag{15.6.2}$$

其中 c 是一个任意的常数,可以调整 c,使(15.6.1)式给出期望的左端值。将(15.6.2)式代入(15.6.1)式,并且注意到$[C]$和$[\alpha]$互为逆矩阵,得到

$$c = \delta a_0 / C_{00} \qquad 和 \qquad \Delta\chi_\nu^2 = (\delta a_0)^2 / C_{00} \tag{15.6.3}$$

或者

$$\delta a_0 = \pm\sqrt{\Delta\chi_\nu^2}\sqrt{C_{00}} \tag{15.6.4}$$

注意最后一个等式,得到了置信区间 $\pm\delta a_0$ 和形式标准误 $\sigma_0 \equiv \sqrt{C_{00}}$ 之间的关系。毫无疑问,发现68%的置信区间是 $\pm\sigma_0$,95%的置信区间是 $\pm 2\sigma_0$,等等。

以上的讨论不只对单个参数 a_i 成立,而且对它们的任何线性组合也都成立。如果

$$b \equiv \sum_{k=0}^{M-1} c_k a_k = \mathbf{c}\cdot\mathbf{a} \tag{15.6.5}$$

则 b 的68%置信区间是

$$\delta b = \pm\sqrt{\mathbf{c}\cdot[C]\cdot\mathbf{c}} \tag{15.6.6}$$

但是,这种简单的、如同正态分布的数值关系在 $\nu > 1$ 的情况下不成立[3]。特别是在 $\nu > 1$ 时,$\Delta\chi^2 = 1$ 并不是68.3%的置信区域的边界,也不投影到68.3%置信区域的边界上。如果要计算不是单个参数的置信区间,而是两个参数联合的置信椭圆,或者三个参数的置信椭球,或者更高,那么必须按照下面的步骤实施上面所说的定理C和定理D。

- 令 ν 为所想显示的联合置信区域的拟合参数数目,$\nu \leqslant M$,称这些参数为"感兴趣参数"。
- 令 p 为所要的置信限,例如 $p = 0.68$ 或 $p = 0.95$。
- 寻找 Δ(也就是 $\Delta\chi^2$)使得一个自由度为 ν 的 χ^2 变量小于 Δ 的概率是 p。对某些 p 和 ν 的有用值,Δ 用表格给出。对于其他的值,可利用程序gammq 和一个简单的求根程序(例如二分法)寻找 Δ 使gammq $(\nu/2, \Delta/2) = 1 - p$。
- 令 χ^2 拟合的 $M \times M$ 协方差矩阵 $[C] = [\alpha]^{-1}$,将所感兴趣的参数对应的 ν 行和 ν 列相交处的元素复制到一个 $\nu \times \nu$ 的矩阵,标记为$[C_{\text{proj}}]$。
- 对矩阵$[C_{\text{proj}}]$求逆(在一维情况下,只需将元素 C_{00} 求倒数即可)。
- 在所感兴趣的 ν 维子空间中,所期望的置信区域的椭圆边界方程是

$$\Delta = \delta\mathbf{a}'\cdot[C_{\text{proj}}]^{-1}\cdot\delta\mathbf{a}' \tag{15.6.7}$$

其中 $\delta\mathbf{a}'$ 是感兴趣参数的 ν 维向量。

如果对这一点还有疑惑,那么将图15.6.4和表15.6.1进行比较会大有帮助。考虑 $\nu = 1$ 和 $\nu = 2$ 时 $M = 2$ 的情况,读者一定能证实下列论述:(i)在 C 和 C' 之间的水平带形包含了99%的概率分布,因此这个带形也就是此置信水平下的 a_1 单独的置信界限;(ii)和上面类似,在 B 和 B' 之间的水平带形是置信水平为90%的置信界限;(iii)以 $\Delta\chi^2 = 2.30$ 标出的虚线椭圆包含了68.3%的概率分布,因而它是在此置信水平下关于 a_0 和 a_1 的联合置信区域。

15.6.6 奇异值分解下的置信界限

如果是用奇异值分解法(参见15.4节)求 χ^2 拟合,那么拟合的形式误差的信息将以不同的形式出现,但通常是更加方便的形式。矩阵 $\mathbf{V}$ 的各列是 M 个向量的正交集,它们分别是 $\Delta\chi^2$ = 常数的椭球的主轴。将每一列分别表示为 $\mathbf{V}_{(0)}, \ldots, \mathbf{V}_{(M-1)}$。这些主轴的长度和对应的奇异值 $w_0, \ldots, w_{M-1}$ 成反比,参看图15.6.5,椭球的边界由下式给出:

$$\Delta\chi^2 = w_0^2(\mathbf{V}_{(0)}\cdot\delta\mathbf{a})^2 + \cdots + w_{M-1}^2(\mathbf{V}_{(M-1)}\cdot\delta\mathbf{a})^2 \tag{15.6.8}$$

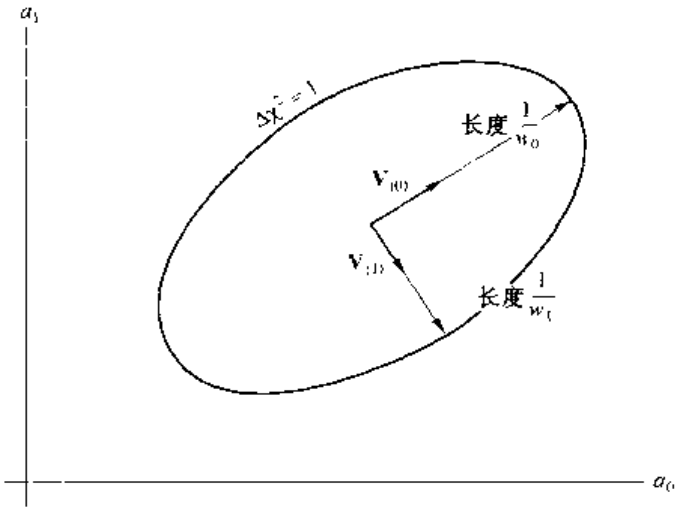


图 15.6.5 置信区域椭圆 $\Delta\chi^2=1$ 和由奇异值分解计算的量之间的关系，向量 $\mathbf{V}_{(i)}$ 是沿着置信区域主轴方向的单位向量。半轴的长度等于奇异值 w_i 的倒数。如果轴用某个常数因子α来标度，则 $\Delta\chi^2$ 将用因子α^2 来标度

上式是前面(15.4.18)式的改写形式。记住，在给出了一列椭球的向量主轴情况下，画椭球的图形，比给出它的二次矩阵形式后画椭球要**容易得多**。

协方差矩阵$[C]$的公式用列 $\mathbf{V}_{(i)}$表示成

$$[C]=\sum_{i=0}^{M-1}\frac{1}{w_i^2}\mathbf{V}_{(i)}\otimes\mathbf{V}_{(i)} \tag{15.6.9}$$

或者用分量，

$$C_{jk}=\sum_{i=0}^{M-1}\frac{1}{w_i^2}V_{ji}V_{ki} \tag{15.6.10}$$

参考文献和深入阅读

Efron, B. 1982, *The Jackknife, the Bootstrap, and Other Resampling Plans* (Philadelphia: S.I.A.M.). [1]
Efron, B., and Tibshirani, R. 1986, *Statistical Science* vol. 1, pp. 54–77. [2]
Avni, Y. 1976, *Astrophysical Journal*, vol. 210, pp. 642–646. [3]
Lampton, M., Margon, M., and Bowyer, S. 1976, *Astrophysical Journal*, vol. 208, pp. 177–190.
Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley).
Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press).

15.7 稳健估计

稳健的概念在前面已经提到过几次。在 14.1 节中注意到，中位数对中心值的估计比均值更稳健；在 14.6 节中曾提到秩相关比线性相关更加稳健。对于因实验误差而引起的不能用高斯模型的偏离点概念，我们也已在 15.1 节中进行了讨论。

"稳健"的概念是 1953 年由G.E.P.Box 在统计学中提出的。对此术语具有各种不同的严格或不太严格的数学定义。但是一般来说，稳健指的是一个统计估计量，它意味着"对最佳估计量的理想化假设，发生微小偏离的不敏感性"[1,2]。"微小"的意义有两种不同的解释，都是很重要的：或者是所有的数据点都有一个部分小的偏离，或者是数据点中很少的几个点存在部分大的偏离。对于后一种解释，引出了偏离点的概念，这在统计过程中通常是最强调的。

统计学家已经发展了各种不同的稳健统计估计的方法。它们中的很多(即使不是全部)可以分为三类。

M 估计由最大似然论述而得出，类似于由(15.1.3)式导出的(15.1.5)式和(15.1.7)式。M 估计通常是和

模型拟合,也就是参数的估计关系最密切的一类,因此在下面将详细地讨论这类估计。

L估计是"顺序统计量的线性组合"。这类估计方法最适用于中心值和集中趋势的估计,尽管它们偶尔也能用于某些参数估计问题。两种"典型"的L估计方法将给出一个一般的概念,它们是中位数和Tukey的三重均值,它定义为某个分布的第一、第二和第三个四分位点的加权平均,权重分别为1/4,1/2和1/4。

R估计是建立在秩检验上的估计,例如两个分布的相等或不等可以用**Wilcoxon检验**来估计,这种估计是在两个分布的组合样本中计算一个分布平均秩。Kolmogorov-Smirnov统计((14.3.6)式)和Spearman秩阶相关系数((14.6.1)式)本质上都是R估计,尽管它们定义的形式不是R估计。

其他的一些稳健估计方法,出自最优控制和滤波领域而不是数理统计领域,将在本节后面提到。一些非常需要采用稳健估计的例子示于图15.7.1中。

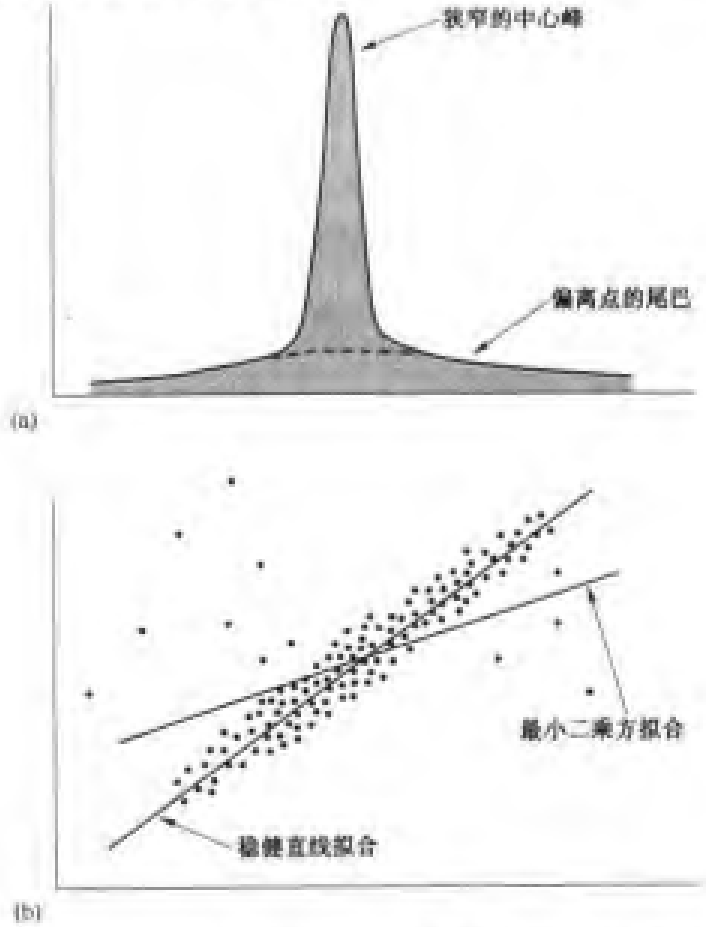


图15.7.1 采用稳健统计方法的两个例子。(a)一个带有偏离点尾巴的一维分布,在这些偏离点处的统计起伏会阻止中心峰位置的精确确定。(b)拟合成一条直线的二维分布,非稳健估计技术(例如最小二乘方拟合)对偏离点会产生不期望的敏感性

15.7.1 使用局部M估计法的参数估计

假设知道测量误差不服从正态分布,那么在导出模型 $y(x;\mathbf{a})$ 中被估参数 $\mathbf{a}$ 的最大似然公式时,将用下式代替(15.1.3)式:

$$P=\prod_{i=0}^{N-1}\{\exp[-\rho(y_i,y\{x_i;\mathbf{a}\})]\Delta y\} \tag{15.7.1}$$

其中函数 ρ 是概率密度的负对数。和(15.1.4)式类似,对(15.7.1)式取对数,发现要极小化下式:

$$\sum_{i=0}^{N-1}\rho(y_i,y\{x_i;\mathbf{a}\}) \tag{15.7.2}$$

常常会出现这种情况，即函数 ρ 并不独立地依赖于它的两个参数(即被测的 y_i 和预测的 $y(x_i)$)，而是只依赖于它们的差，至少在用我们能够分配到各点的权重因子 σ_i 度量时是如此。在这种情况下，M 估计称为**局部的**，可以用下面的表述代替(15.7.2)式：

$$\text{对 } \mathbf{a} \text{ 求} \quad \sum_{i=0}^{N-1} \rho\left(\frac{y_i - y(x_i;\mathbf{a})}{\sigma_i}\right) \quad \text{的极小化} \tag{15.7.3}$$

其中函数 $\rho(z)$ 的单变量 $z \equiv [y_i - y(x_i)]/\sigma_i$ 的函数。

如果定义 $\rho(z)$ 的微分为函数 $\psi(z)$，

$$\psi(z) \equiv \frac{d\rho(z)}{dz} \tag{15.7.4}$$

则推广(15.1.7)式到一般的 M 估计情形，为

$$0 = \sum_{i=0}^{N-1} \frac{1}{\sigma_i}\psi\left(\frac{y_i - y(x_i)}{\sigma_i}\right)\left(\frac{\partial y(x_i;\mathbf{a})}{\partial a_k}\right) \qquad k = 0, \ldots, M-1 \tag{15.7.5}$$

如果将(15.7.3)式和(15.1.3)式，(15.7.5)式和(15.1.7)式比较，就会发现在误差呈正态分布的特殊情形下有

$$\rho(z) = \frac{1}{2}z^2 \qquad \psi(z) = z \qquad (\text{正态}) \tag{15.7.6}$$

如果误差呈**二重或双边指数函数**分布，也就是

$$\text{Prob}\ \{y_i - y(x_i)\} \sim \exp\left(-\left|\frac{y_i - y(x_i)}{\sigma_i}\right|\right) \tag{15.7.7}$$

则对照

$$\rho(x) = |z| \qquad \psi(z) = \text{sgn}(z) \qquad (\text{二重指数人布}) \tag{15.7.8}$$

比较(15.7.3)式，发现在这种情形下，最大似然估计量是通过求**平均绝对偏差**的极小值来获得的，而不是对均方偏差取极小值来得到的。这里，尽管分布尾巴仍呈指数衰减，但是它的渐近结果比相应的高斯尾巴大很多。

带有更广泛的(因此有时也更加真实的)尾部的分布是**柯西(Cauchy)分布或者洛伦兹(Lorentz)分布**：

$$\text{Prob}\ \{y_i - y(x_i)\} \sim \frac{1}{1 + \frac{1}{2}\left(\frac{y_i - y(x_i)}{\sigma_i}\right)^2} \tag{15.7.9}$$

这隐含着

$$\rho(z) = \log\left(1 + \frac{1}{2}z^2\right) \qquad \psi(z) = \frac{z}{1 + \frac{1}{2}z^2} \qquad (\text{洛伦兹}) \tag{15.7.10}$$

注意，在推广的正规方程组(15.7.5)中，函数 ψ 作为一个权重函数出现，当误差呈正态分布时，(15.7.6)式说明数据点的偏离值越大，权重就越大。与此不同的是，当分布的尾部稍微突出些时，如(15.7.7)式，则(15.7.8)式说明所有的偏离点得到相同的相对权重，而只用到了它们的符号信息。最后，如果尾部更大，则(15.7.10)式说明随着偏离值的增大，ψ 增加，然后开始减小，以致非常偏远的点(真正的偏离点)在参数的估计中根本没有计算在内

由个别数据点给出的权重随着偏离值的增加开始递增，然后递减的一般概念促使人们对 ψ 进行一些附加的规定，它和标准的教科书上的概率分布并不对应。下面有两个例子：

Andrew 正弦函数

$$\psi(z) = \begin{cases} \sin(z/c) & |z| < c\pi \\ 0 & |z| > c\pi \end{cases} \tag{15.7.11}$$

如果测量误差恰好是正态分布，标准差为 σ_i，则可以证明常数 c 的最优值是 $c = 2.1$。

Tukey 双权

$$\psi(z)=\begin{cases} z(1-z^2/c^2)^2 & |z|<c \\ 0 & |z|>c \end{cases} \tag{15.7.12}$$

上式中 c 的最优值是 $c=6.0$。

15.7.2 M 估计的数值计算

用 M 估计的方法拟合一个模型,首先必须确定想要哪一种 M 估计方法,也就是想要用哪一对匹配的 ρ、ψ。倾向于使用(15.7.8)式或(15.7.10)式。

接着,必须在两个一样难度的问题中进行选择。要么解 M 方程构成的非线性方程组(15.7.5),要么求含 M 个变量的单一函数的极小值(15.7.3)。

注意,在(15.7.8)式的函数中,ψ 是不连续的,ρ 的导数也是不连续的。这些不连续性常常会在一般的非线性方程求解和一般的求函数极小化的程序中引起大的混乱。现在可能想舍弃(15.7.8)式,而采用(15.7.10)式,因为(15.7.10)式中的函数比较光滑。但是,将发现后一种选择对很多一般的方程求解或求极小化程序来说有很多不良之处:拟合参数的微小变化可能使 $\psi(z)$ 偏离它的峰值,而进入它的一个小的渐近区域。因此,方程中的不同项有时起作用,有时不起作用(和解析不连续性几乎一样差)。

但是如果读者的计算机能胜任,则这对于向下单纯极小化算法是一个非常好的应用。这一算法示于 10.4 节的程序 amoeba 和 10.9 节的程序 amebsa。这些算法对于连续性没有进行任何假设,只是体现向下滑的意图。对函数 ρ 的任何合理选择算法本质上都有效。

无论如何找一个好的初始值是非常有利的(更加经济),通常,首先用标准的 χ^2(非稳健)方法拟合模型,如在 15.4 节或 15.5 节中所叙述的那样。接着用所得的拟合参数作为 amoeba 的初始值,然后就可以使用 ρ 的稳健选择,并且极小化(15.7.3)式。

15.7.3 通过极小化绝对偏差拟合直线

有时偶然会出现比上面所讨论的一般技巧更容易的特殊情况。当模型是一条简单的直线

$$y(x;a,b)=a+bx \tag{15.7.13}$$

并且其中各点的权重 σ_i 都相等时,(15.7.7)式~(15.7.8)式就是这种情况。这个问题恰好是在(15.2.1)式中提到的要采用稳健估计方法的问题,也就是用一组数据点拟合一条直线。所求极小化的优值函数是

$$\sum_{i=0}^{N-1}|y_i-a-bx_i| \tag{15.7.14}$$

而不是由(15.2.2)式给出的 χ^2。

简化的关键基于以下事实:一组数 c_i 的中位数 c_M 也就是使偏差绝对值的和达到极小的值:

$$\sum_i |c_i-c_M|$$

(证明:在上式中对 c_M 求微分,并令它为 0。)

由此我们知道对固定的 b,极小化(15.7.14)式的 a 为

$$a=\text{median}\{y_i-bx_i\} \tag{15.7.15}$$

对于参数 b,(15.7.5)式变为

$$0=\sum_{i=0}^{N-1}x_i\,\text{sgn}(y_i-a-bx_i) \tag{15.7.16}$$

(其中 sgn(0)看做是零。)如果用(15.7.15)式中隐含的函数关系 $a(b)$ 代替上式中的 a,那么将得到一个含有单变量的方程,这个变量可以采用 9.1 节叙述的二分法或划界法求解(事实上,如果采用更高级的求根方法将非常危险,因为(15.7.16)式中的函数是不连续的)。

下面是一个完成以上所述处理的程序。它调用 8.5 节中的 select 求中位数。划界法和二分法都用于下面的程序中，用来计算 χ^2 的结果，此结果将产生 a 和 b 的初始猜测值。注意(15.7.16)式右端的计算采用程序 rofunc，它通过全局(最高级别)变量进行数据传输。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP aa,abdevt;
const Vec_DP *xt_p,*yt_p;

void NR::medfit(Vec_I_DP &x, Vec_I_DP &y, DP &a, DP &b, DP &abdev)
采用最小绝对偏差的准则拟合直线 y = a + bx。数组 x[0..ndata-1]和 y[0..ndata-1]是输入实验数据点。输出拟合参
数 a 和 b，同时输出的还有来自拟合直线的实验数据点(关于 y 的)的平均绝对偏差 abdev。此程序还要用到程序 rofunc，
它们之间通过全局变量进行数据交换。
{
    int j;
    DP bb,b1,b2,del,f,f1,f2,sigb,temp;
    DP sx=0.0,sy=0.0,sxy=0.0,sxx=0.0,chisq=0.0;

    int ndata=x.size();
    xt_p= &x;
    yt_p= &y;
    for (j=0;j<ndata;j++) {                作为 a 和 b 的初始猜测，我们将寻找最小二乘拟合直线。
        sx += x[j];
        sy += y[j];
        sxy += x[j]*y[j];
        sxx += x[j]*x[j];
    }
    del=ndata*sxx-sx*sx;
    aa=(sxx*sy-sx*sxy)/del;                最小二乘解。
    bb=(ndata*sxy-sx*sy)/del;
    for (j=0;j<ndata;j++)
        chisq += (temp=y[j]-(aa+bb*x[j]),temp*temp);
    sigb=sqrt(chisq/del);                  标准差给出迭代步长应该多大这一概念。
    b1=bb;
    f1=rofunc(b1);
    if (sigb > 0.0) {
        b2=bb+SIGN(3.0*sigb,f1);           沿着由 f1 已知的向下方向，猜测跨越 3-σ 的划界。
        f2=rofunc(b2);
        if (b2 == b1) {
            a=aa;
            b=bb;
            abdev=abdevt/ndata;
            return;
        }
        while (f1*f2 > 0.0) {              进行划界。
            bb=b2+1.6*(b2-b1);
            b1=b2;
            f1=f2;
            b2=bb;
            f2=rofunc(b2);
        }
        sigb=0.01*sigb;
        while (fabs(b2-b1) > sigb) {
            bb=b1+0.5*(b2-b1);             二分法。
            if (bb == b1 || bb == b2) break;
            f=rofunc(bb);
            if (f*f1 >= 0.0) {
                f1=f;
                b1=bb;
            } else {
                f2=f;
                b2=bb;
            }
        }
    }
    a=aa;
    b=bb;
    abdev=abdevt/ndata;
}
```

```
#include <cmath>
#include <limits>
#include "nr.h"
using namespace std;

extern DP aa,abdevt;                          在 medfit 中定义。
extern const Vec_DP *xt_p,*yt_p;

DP NR::rofunc(const DP b)
对于给定的 b 值计算(15.7.16)右端的值。通过全局变量和程序 medfit 进行数据交换
{
    const DP EPS=numeric_limits<DP>::epsilon();
    int j;
    DP d,sum=0.0;

    const Vec_DP &xt=*xt_p,&yt=*yt_p;         为简化代码使用引用。
    int ndatat=xt.size();
    Vec_DP arr(ndatat);
    for (j=0;j<ndatat;j++) arr[j]=yt[j]-b*xt[j];
    if (ndatat & 1 == 1) {
        aa=select((ndatat-1)>>1,arr);
    } else {
        j=ndatat >> 1;
        aa=0.5*(select(j-1,arr)+select(j,arr));
    }
    abdevt=0.0;
    for (j=0;j<ndatat;j++) {
        d=yt[j]-(b*xt[j]+aa);
        abdevt += fabs(d);
        if (yt[j] != 0.0) d /= fabs(yt[j]);
        if (fabs(d) > EPS) sum += (d >= 0.0 ? xt[j] : -xt[j]);
    }
    return sum;
}
```

15.7.4 其他的稳健估计方法

有时候,要从一组数据中估计出一些参数,而对这些参数的可能值和可能的不确定性有些**先验的**知识。在这种情况下,可以进行一个把这些预知的信息正确考虑进去的拟合,既不是用一个预先确定的值完全冻结一个参数(如 15.4 节中的 lfit),也不是使参数完全由数据组确定。这样做的形式体系称为"**先验的协方差的运用**"。

在信号处理和控制理论中有一个相关的问题,即要在噪声中找出随时间变化的信号,也就是维持这种信号的一个估计。如果已知信号由一些变化非常慢的参数表征,那么 **Kalman 滤波**形式体系指出了如何对接收到的、粗糙测量的、随时间而变的信号进行处理,产生作为时间的函数最佳的参数估计。例如,如果信号是一个频域调制的正弦波,则缓慢变化的参数可能是瞬态频率。这种情况下的 Kalman 滤波称为**锁相环**(phase-locked loop),它装置在高级的无线电接收器回路中(参见本节参考文献[3,4])。

参考文献和深入阅读

Huber, P.J. 1981, *Robust Statistics* (New York: Wiley). [1]

Launer, R.L., and Wilkinson, G.N. (eds.) 1979, *Robustness in Statistics* (New York: Academic Press). [2]

Bryson, A. E., and Ho, Y.C. 1969, *Applied Optimal Control* (Waltham, MA: Ginn). [3]

Jazwinski, A. H. 1970, *Stochastic Processes and Filtering Theory* (New York: Academic Press). [4]

第 16 章　常微分方程组的积分

16.0　引言

与常微分方程组(ODE)有关的问题通常都可以化成一系列一阶微分方程的问题来研究。例如,一个二阶常微分方程

$$\frac{d^2y}{dx^2}+q(x)\frac{dy}{dx}=r(x) \tag{16.0.1}$$

能够化成两个一阶方程

$$\begin{aligned}\frac{dy}{dx}&=z(x)\\ \frac{dz}{dx}&=r(x)-q(x)z(x)\end{aligned} \tag{16.0.2}$$

其中 z 是一个新变量。这是对任意高阶常微分方程的典型处理方法。新变量通常选择使它们互为(和方程原变量)导数。有时,在定义新变量时,也加上一些和方程有关的系数或某些自变量乘方,以减缓导致计算溢出或舍入误差的积累。可以遵循这样一条常识性的原则:如果发现在结果中,原来的变量变化平稳,而辅助变量变化不规则,则应该找出原因,并重新选择合适的变量。

这样,对于一般的常微分方程便可以通过一组 N 个函数 $y_i(i=1,2,\ldots,N)$ 的 N 个联立一阶微分方程组来处理。这些微分方程可以写成下面的一般形式:

$$\frac{dy_i(x)}{dx}=f_i(x,y_0,\ldots,y_{N-1}),\qquad i=0,\ldots,N-1 \tag{16.0.3}$$

其中等式右边的函数 f_i 是已知的。

常微分方程组的解不完全由其方程决定。要确定方程的解还需要边界值条件。边界值条件是指(16.0.3)式中的函数 y_i 的值的代数条件。一般来说,它们能在某些确定的离散点得到满足,但是在这些离散点之间就不成立了,即不能由微分方程自动得到满足。边界值条件有时很简单,例如,要求定义某些变量具有某些数值;有时候却较复杂,成为一组变量之间的非线性代数方程。

通常,正是边界值条件的属性决定了哪种数值解法是可行的。边界值条件可分为下面的两大类:

- 在**初值问题**中,所有的 y_i 都给出了起始值 x_s,希望找出在终点 x_f 的 y_i 值,或在某离散点列(如表列间隔上)的 y_i 值。
- 在**两点边值问题**中,在多个 x 处定义了边界条件。在典型情况下,有的条件在 x_s 处进行定义,并给出 x_f 处的隐含条件。

本章将讨论初值问题,有关两点边界值的问题将在第 17 章进行讨论,后者通常更困难一些。

解决初值问题的基本思路是:把(16.0.3)式中的 dy 和 dx 用有限步长 Δy 和 Δx 来代替。然后等式两边同乘以 Δx。这便给出了函数随自由变量 x 的"步长" Δx 变化而变化的代数公式。在把步长取得尽可能小的极限过程中,便可以得到微分方程的一个极好的近似。理论上,实现这个过程可以使用**欧拉(Euler)方法**(下面的(16.1.1)式),但实际上是不实用的。欧拉方法在理论上是很重要的,许多实际的方法都归结于欧拉方法同样的思想,将导数(等式右边)乘以步长,函数相应地增加一个小增量。

本章中,主要讨论对常微分方程组求解初值问题的三种主要的实用数值解法:

- Runge-Kutta 方法
- Richardson 外推法,其特例为 Bulirsch-Stoer 方法。
- 预测-校正法。

下面对这三种方法分别进行简要介绍。

1. **Runge-Kutta** 方法通过类似欧拉方法中的步长(每次都要计算出右边的 f 值),把各个步长所提供的信息结合起来,算出有一定间隔的解,然后用所得到的信息和泰勒展开式匹配,以得到高阶近似解。

2. **Richardson 外推**法利用了一种很有效的思想,它是在如果步长比实际值小得多时,对能得到的计算值进行外推。特殊情况下,趋近零的步长值进行外推是理想的目标。Bulirsch 和 Stoer 首次把这种方法应用到常微分方程的积分上,因此,外推法常被称为 Bulirsch-Stoer 方法。

3. **预测-校正**法把每步计算出的结果保留下来,并用那些结果外推出下一步的结果,然后,用新点上的导数信息进行校正。此方法对于平滑函数效果最好。

Runge-Kutta 方法用于:(i)不知道其他更好的方法;(ii)用 Bulirsch-Stoer 方法无法解决的问题;(iii)一个比较平常的、不需考虑计算效率的问题。Runge-Kutta 方法通常总是能成功的,但是它并不是最快的方法。在对 f 进行估值比较容易,而且只需达到一定精度($\lesssim 10^{-5}$)的时候,用这种方法通常是最快的。预测-校正法因为需要利用已计算的值,所以开始计算时比较困难,但对很多平滑函数来说,它比 Runge-Kutta 方法计算效率高一些。近几年来,Bulirsch-Stoer 方法在很多应用中取代了预测-校正法,但 Bulirsch-Stoer 方法一统天下的说法尚为时过早。然而实践表明,只有相当复杂的预测-校正法才能和这种方法的效果相当,因此在本书中没有给出预测-校正法的实施。将在 16.7 节中进一步讨论预测-校正法,以便当读者遇到适当的问题时,能使用一个表述比较粗略的程序。按我们的经验,本书给出的相对较简单的 Runge-Kutta 方法和 Bulirsch-Stoer 方法程序,对大多数问题来说已经足够了。

这三种方法都可以系统化编制来监视内部数据的协调性。这样便可通过改变基本步长,自动减缓一些可避免的引入解中的数字误差(**自适应地**)。一直建议在程序中实现自适应步长控制,下面也将这么做。

总而言之,所有三种方法都可以用于任何初值问题。每种方法各有优缺点,这是在使用之前必须了解的。

在本章中,把程序组织成相互调用的三个嵌套级别。最低一级称为**算法**程序。它实现了方法基本公式,从 x 处的因变量 y_i 开始,计算因变量在 $x+h$ 处的新值。算法程序中,也提供了有关该步计算结果的近似效果信息。但程序不能自适应判定这个解是否可以接受。

精度控制判定的代码编在**步进**程序中,步进程序调用算法程序,它可以拒绝接受计算的结果。设置一个较小的步长,重新调用算法程序,直至达到事先定义好的精确度为止。步进的基本任务是根据特定性能选用最大步长。只有这一点达到了,算法的威力才真正显示出来。

在步进程序之上是驱动程序,用以开始和结束积分,存储中间结果,以及用于和用户的接口。我们给出的驱动程序并不是标准的,读者应该把它们当做一些例子,并且能为特定的应用而修改这些程序。

在下面的程序中,rk4、rkck、mmid、stoerm 和 simpr 是算法程序;rkqs、bsstep、stiff 和 stifbs 是步进程序;rkdumb 和 odeint 是驱动程序。

本章的 16.6 节将讨论**刚度方程(stiff equation)**的问题,这和常微分方程以及偏微分方程有关(第 19 章)。

参考文献和深入阅读

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall).

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 5.

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), Chapter 7.

Lambert, J. 1973, *Computational Methods in Ordinary Differential Equations* (New York: Wiley).

Lapidus, L., and Seinfeld, J. 1971, *Numerical Solution of Ordinary Differential Equations* (New York: Academic Press).

16.1　Runge-Kutta 方法

欧拉方法的公式是

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{16.1.1}$$

这里，从 x_n 导出 x_{n+1} 为 $x_{n+1} \equiv x_n + h$。这个公式是不对称的：通过每次增加间隔 h 将解推进，但只是在间隔起点用到求导的信息(参见图 16.1.1)。这就意味着，引入(16.1.1)式的步长误差比修正值小 h 的一次幂，即 $O(h^2)$(可以通过幂级数展开式来证明)。

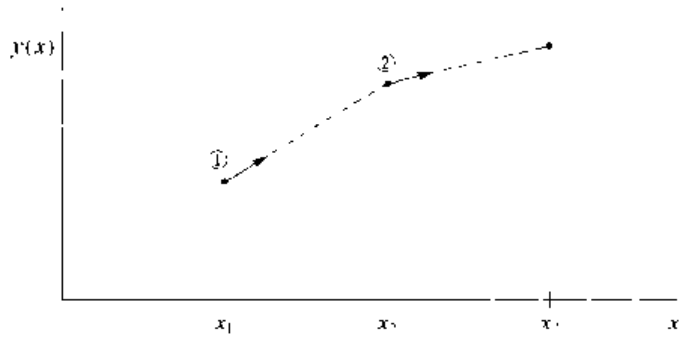


图 16.1.1　欧拉方法。在这种求解常微分方程最简单(也是最不精确)的方法中，通过对每段间隔起始点外推来求出下一个函数值，这种方法的精确度是一阶的

欧拉方法未得到实用有几个原因：(i)这种方法和其他更巧妙的方法相比，它不是非常精确的；(ii)这种方法的稳定性也不是很好(参见 16.6 节)。

但是，可以考虑用类似(16.1.1)式中的步骤，取间隔的中点为"试验"步。然后，用那中点的 x 和 y 的值计算整个间隔的"实际"步长。图 16.1.2 说明了这个思想。写成方程是

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{16.1.2}$$

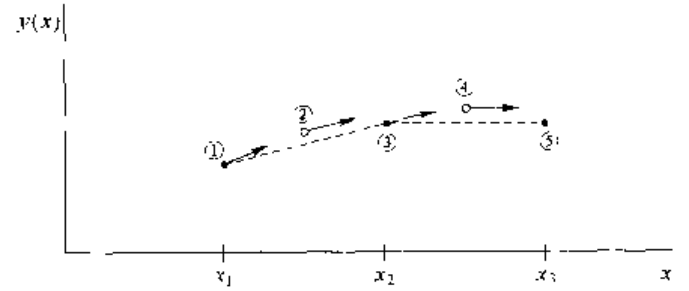


图 16.1.2　中点法。使用在每一步上的初始导数，求出间隔的中点，然后用这个中点导数求出整段间隔的值，这样可以得到二阶的精确度。在本图中，实心点代表最后的函数值；空心点代表的函数值，一旦它们的导数被计算并使用以后便被舍弃

在(16.1.2)式的误差项中已表明，这种对称性算法消除了一阶误差项，使其误差成为二阶的(如果某种算法的误差量 $O(h^{n+1})$，则习惯上称它为 n 阶的)。实际上，(16.1.2)式称为**二阶 Runge-Kutta 方法或中点法**。

下面继续讨论。有很多方法计算右边的 $f(x, y)$，它们都满足一阶误差的精度，但这些方法都各自有不同的高阶误差项。将这些方法适当结合，可以逐阶逐阶地消除误差项。这也正是 Runge-Kutta 方法的基本思想。Abramowitz 和 Stegun[1]，以及 Gear[2]都根据这种思想提出了各种不同的算式。但目前最常用、最有效的是**四阶 Runge-Kutta 公式**，这个公式有一定的光滑度：

$$
\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\
k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\
k_4 &= hf(x_n + h, y_n + k_3) \\
y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\end{aligned}
\tag{16.1.3}
$$

四阶 Runge-Kutta 方法在每个步长 h 中,对右端需要四次求值计算(参见图 16.1.3)。如果在(16.1.3)式中,用至少可能两倍大的步长就能达到中点法(16.1.2)式一样精度的话,则这种方法就优于中点法。通常情况是这样的,但不能说总是这样的。这让我们回到一个很重要的问题,即**高阶并不意味着高精度**。"四阶 Runge-Kutta 方法通常优于二阶"的说法是正确的,但应该把它当做科学实践中的暂时性法则,而非严格数学中的法则。也就是说,它反映了当前科学家解决问题的一种特性。

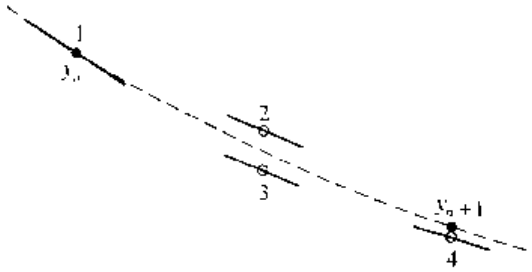


图 16.1.3 四阶 Runge-Kutta 方法。在每一步中求导通过四步来近似:一次在起始点,两次在试探中点,一次在最末试探点。由这些导数算出最后的函数值(用实心点表示)(详细说明参见正文)

对于很多搞科研的应用者来说,四阶 Runge-Kutta 方法并不仅是常微分方程最初使用的方法,而且是现在使用的一种选择方法。实际上,用这种旧算法,尤其像我们下一节那样把自适应步长算法结合进去,还是能收获很大的。但是注意,这种方法也有其局限性,对于需要高精度的情况,Bulirsch-Stoer 方法和预测-校正法会更加有效。在 16.2 节中,将给出一个 Runge-Kutta 方法的现代实用算法,在精度要求不高时,它是很有竞争力的。在本节参考文献[3]中有编制 Runge-Kutta 方法程序的精彩论述。

下面是对一组几个微分方程实施一步 Runge-Kutta 方法计算的程序。输入自变量的值,然后得到一个步长 h(可正可负)的新值。读者会注意到,程序不但需要用户提供计算方程右端项的函数 derivs,还需提供起始点的导数值。为什么不让程序调用 derivs 来计算初值呢?答案在下节阐明,简言之是这样的:这个调用不只是具有这些初始条件的惟一调用。也许先前步骤中取的步长值太大,这样便可更改。在这种情况下,就没有必要在起始点调用 derivs。注意下面这个程序,只调用了三次 derivs 函数。

```
#include "nr.h"

void NR::rk4(Vec_I_DP &y, Vec_I_DP &dydx, const DP x, const DP h,
    Vec_O_DP &yout, void derivs(const DP, Vec_I_DP &, Vec_O_DP &))
给定已知 x 的变量 y[0..n-1]及其导数 dydx[0..n-1]的值。利用四阶 Runge-Kutta 方法以间隔 h 递增求解,返回的增量
值存储到 yout[0..n-1]中,它并不需和 y 矩阵有明显不同。用户提供程序 derivs(x,y,dydx),计算在 x 处的 dydx 值。
{
    int i;
    DP xh,hh,h6;

    int n=y.size();
    Vec_DP dym(n),dyt(n),yt(n);
    hh=h*0.5;
    h6=h/6.0;
    xh=x+hh;
    for (i=0;i<n;i++) yt[i]=y[i]+hh*dydx[i];          第一步。
    derivs(xh,yt,dyt);                                第二步。
```

```
    for (i=0;i<n;i++) yt[i]=y[i]+hh*dyt[i];
    derivs(xh,yt,dym);                          第三步。
    for (i=0;i<n;i++) {
        yt[i]=y[i]+h*dym[i];
        dym[i] += dyt[i];
    }
    derivs(x+h,yt,dyt);                         第四步。
    for (i=0;i<n;i++)                           以适当权重积累增量。
        yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
}
```

Runge-Kutta 方法每步都采用相同的方法。以前的计算状况不用在以后的计算中，这在数学上是合适的，因为常微分方程轨迹线中的任一点都可作为初始点。这种每步处理方法都相同的事实，使得“驱动程序”很容易对 Runge-Kutta 方法进行调用。

对于精确计算中基本的自适应步长控制问题，将在下一节讨论。当然，偶尔有时只想在等距间隔上计算函数值列表，而且也不需要很高精确度。在通常情况下，想要画出函数图形，这时就需要一个简单的驱动程序，以一定的步数从初始的 x_i 计算到最终的 x_f。为了检查结果的精确性，可以加倍步数，重复计算并比较结果。这种逼近当然不会减少计算时间，而且对要求可变步长的情况也许失败。但是，这种方法减少了用户的额外考虑因素，对于小型的问题来说，这可能是考虑的最重要的一个方面。

下面是一个驱动程序，把积分的函数放在全局指针 xx_p 和 y_p 指向的数组中，注意要如下所述调用函数为它们分配内存。

```
    xx_p=new Vec_DP(nstep+1);
    y_p=new Mat_DP(nvar,nstep+1);

#include "nr.h"

extern Vec_DP *xx_p;                    与主程序通信。
extern Mat_DP *y_p;

void NR::rkdumb(Vec_I_DP &vstart, const DP x1, const DP x2,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &))
从已知 x1 处初始值 vstart[0..nvar-1]开始，利用四阶 Runge-Kutta 方法以相等增量递增 nstep 步到 x2，用户提供的程序
derivs(x,v,dvdx)估算导数值，每一步的结果存储在指针 xx_p 和 y_p 指向的数组 xx[0..nstep]，y[0..nvar-1][0..nstep]
中。nstep 和 nvar 由全局数组 *y_p 的维数决定。
{
    int i,k;
    DP x,h;

    Vec_DP &xx=*xx_p;                   另取名以简化代码。
    Mat_DP &y=*y_p;
    int nvar=y.nrows();
    int nstep=y.ncols()-1;
    Vec_DP v(nvar),vout(nvar),dv(nvar);
    for (i=0;i<nvar;i++) {              装载起始值。
        v[i]=vstart[i];
        y[i][0]=v[i];
    }
    xx[0]=x1;
    x=x1;
    h=(x2-x1)/nstep;
    for (k=0;k<nstep;k++) {             进行 nstep 步计算。
        derivs(x,v,dv);
        rk4(v,dv,x,h,vout,derivs);
        if (x+h == x)
            nrerror("Step size too small in routine rkdumb");
        x += h;
        xx[k+1]=x;                      存储中间步长。
        for (i=0;i<nvar;i++) {
            v[i]=vout[i];
            y[i][k+1]=v[i];
        }
    }
}
```

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.5. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 2. [2]

Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121. [3]

Rice, J.R. 1983, *Numerical Methods, Software, and Analysis* (New York: McGraw-Hill), §9.2.

16.2 Runge-Kutta 方法的自适应步长控制

一个好的常微分方程的积分解法,应在求解过程中加上一些自适应控制,不断地修改步长值。通常情况下,自适应步长的控制是为了用尽可能少的计算量达到解的预定精确度。在一些不可靠的地方,应该用一些较小的步长,而在一些平滑的无关紧要的地方,则用大一些的步长以加快计算过程。其结果的效率不只是百分之几或几倍,有时是十几倍、上百倍,甚至更多。所需的精确度不仅是指在解中,有时是指在某些可被监测的相关量中。

自适应步长控制的实现,需要关于每步执行算法过程中的有关信息,尤其是截断误差的有效估计。本节中,将要了解能得到的不少信息。显然,这种信息的计算会增加额外的计算开销,但是这种投资通常是能很快偿还的。

对于四阶 Runge-Kutta 方法,最直接的方法是**步长数加倍**(参见本节参考文献[1])。把每一步分成两次计算,第一次作为整个一步,然后再一次独立地分成两个半步来计算(参见图 16.2.1)。在这种情况下,右边式求值需要进行多少额外的计算呢?每三个 Runge-Kutta 步在求解计算中需要四次的估算,而每一步经过步长数加倍以后共用一个的起始点,所以总数是 11。因为用步长控制法可达到较少步长(原步长的一半)时的精度,所以应和 8 比而不是和 4 比,额外计算系数是 1.375。

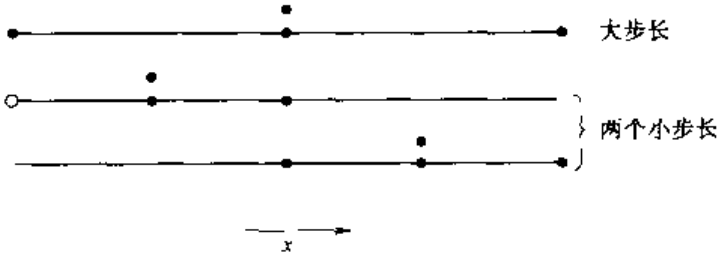


图 16.2.1 四阶 Runge-Kutta 方法通过步长数加倍进行自适应步长控制。实心点代表进行求导估算的点,空心点表示上一步进行过求估算(本来是实心)的点,所以每两步计算的总数是11次,通过比较大步长和两个小步长的精确度,来决定是在下一步调整步长值,还是因为精度不够重新计算这一点的结果

现在精确地论证一下从 x 到 $x+2h$ 计算 $y(x+2h)$,用两种近似解法:y_1(每一步长 $2h$)和 y_2(两个步长,每个步长 h),因为基本方法是四阶的,两种数值近似的结果是

$$\begin{aligned} y(x+2h) &= y_1 + (2h)^5\phi + O(h^6) + \dots \\ y(x+2h) &= y_2 + 2(h^5)\phi + O(h^6) + \dots \end{aligned} \tag{16.2.1}$$

其中,对于阶 h^5,ϕ的值相对于步长保持为常数(根据泰勒展开式,ϕ的值为 $y^{(5)}(x)/5!$量级)。在(16.2.1)式的第一个表达式中,因为步长是 $2h$,所以是$(2h)^5\phi$;而第二个表达式中,每一步步长的误差是 $h^5\phi$,所以结果是$2(h^5)\phi$,两种数值估算的差很方便地表明了截断误差

$$\Delta \equiv y_2 - y_1 \tag{16.2.2}$$

对这个差值应保持恰当的精确度，不要太大也不要太小。可以通过调整 h 来做到这一点。

读者也许会想到，忽略了 h^6 以及更高阶项，可以通过解(16.2.1)式中的两个方程来改善理想值 $y(x+2h)$ 的数值估计情况，如

$$y(x+2h)=y_2+\frac{\Delta}{15}+O(h^6) \tag{16.2.3}$$

这种近似精确到五阶，比最初的 Runge-Kutta 方法高一阶。但是不能想当然地这么做。(16.2.3)式也许是五阶精度，但是我们没有办法监测其截尾误差。更高阶并不代表永远是高的精确度。使用(16.2.3)式一般不会有多大坏处，但是无法直接了解到改善情况如何。因此应该用 Δ 作为误差估计，而使用(16.2.3)式计算以得到较高的精确度。使用类似(16.2.3)式的过程，技术上称为"局部外推"。

可选的步长调节算法是基于**嵌入的 Runge-Kutta 公式**，最初是 Fehlberg 发明的。一个很有意思的情况是，在 Runge-Kutta 公式中，对于高于 4 的阶次 M 来说，需要大于 M 个函数的估计值(尽管不会大于 $M+2$)。这说明了四阶算法的通用性：给不拘一格的想法反戈一击。但是，Fehlberg 发现了用 6 个函数估值的五阶算法，同时 6 个函数估值的另外一种组合将导出一个四阶算法，两个对 $y(x+h)$ 的估计值的差异能够用来估计截断误差并以调整步长。从 Fehlberg 的最初公式发现以来，另外也发现了几个其他的嵌入式 Runge-Kutta 公式。

很多实践者一度对 Runge-Kutta-Fehlberg 算法感到担心。这是因为用同样的估值点向前估测函数值和估计误差，比加倍步长的方法要危险得多。但是经验表明，这种考虑实际上是不必要的。一般来说，嵌入式 Runge-Kutta 公式，比基于加倍步长算法要提高效率一到两倍左右。

一般的五阶 Runge-Kutta 公式是

$$\begin{aligned}
k_1&=hf(x_n,y_n)\\
k_2&=hf(x_n+a_2h,y_n+b_{21}k_1)\\
&\cdots\\
k_6&=hf(x_n+a_6h,y_n+b_{61}k_1+\cdots+b_{65}k_5)\\
y_{n+1}&=y_n+c_1k_1+c_2k_2+c_3k_3+c_4k_4+c_5k_5+c_6k_6+O(h^6)
\end{aligned} \tag{16.2.4}$$

嵌入式四阶公式是

$$y^*_{n+1}=y_n+c^*_1k_1+c^*_2k_2+c^*_3k_3+c^*_4k_4+c^*_5k_5+c^*_6k_6+O(h^5) \tag{16.2.5}$$

误差估计值为

$$\Delta\equiv y_{n+1}-y^*_{n+1}=\sum_{i=1}^{6}(c_i-c^*_i)k_i \tag{16.2.6}$$

倾向于使用不同常量的特殊值，Cash 和 Karp[2]的发现列在表 16.2.1 中。这给出了比 Fehlberg 原来值更有效的算法，在误差特性方面更为好一些。

表 16.2.1　嵌入式 Runge-Kutta 方法的 Cash-Karp 参数

i	a_i	b_{ij}					c_i	c^*_i
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$
$j=$		1	2	3	4	5		

现在,至少已大约知道误差的所在,需要考虑如何将其控制在一定范围以内。Δ 和 h 之间的关系如何?根据(16.2.4)式~(16.2.5)式可知,Δ_0 和 h^5 成正比。如果取步长 h_1 并产生误差 Δ_1,则取步长 h_0 时,产生的误差可以估计为

$$h_0 = h_1 \left|\frac{\Delta_0}{\Delta_1}\right|^{0.2} \tag{16.2.7}$$

因此,用 Δ_0 来表示所需的精确度。这样(16.2.7)式可用于两个方面:如果 Δ_1 在数值上比 Δ_0 大,则从等式可以得出,再次试当前(失败)步时应该把步长减少多少;如果 Δ_1 比 Δ_0 小,则从等式可以得出,在下一步时提高步长值多大是安全的。局部外推接受五阶的值 y_{n+1},尽管误差估计实际上用于四阶值 y_{n+1}^*。

表达式中隐藏着这样一个事实:Δ_0 实际上是所需精确度的一个向量,每一组常微分方程的每个等式都有一个误差项。总的来说,我们的精度要求各等式的误差都在各自允许的误差范围以内。也就是说,可根据"最坏情况"等式的需求相应地缩放步长大小。

怎么把所需要的精度,例如叙述为"精确到 10^6 分之一",和 Δ_0 联系起来呢?这是一个比较难以确定的问题,这和实际应用问题有关。有时也许在处理一组因变量数值相差很大的方程,这种情况下,也许希望用相对误差,$\Delta_0 = \epsilon\ y$,其中ϵ是类似 10^{-6}或其他的值。有时也许在处理某些振荡函数,穿过零点但限制于某一最大值范围内,这时,也许把 Δ_0 设置成最大值的ϵ倍。

综合这些考虑设计出一个比较通用的程序。程序的一个参量,当然是在当前起始步中因变量的向量,把这称为 y[0..n-1]。再让用户为每一步定义另外一个相应的向量参量 yscal[0..n-1],以及总的容差限 eps。这样,对于第 i 个方程来说所需的精确度就为

$$\Delta_0 = \texttt{eps} \times \texttt{yscal[i]} \tag{16.2.8}$$

如果想用相对误差表示,则可以把指向 y 的指针放入 yscal 指针的调用位置(不需要把值复制到另外一个矩阵中)。如果读者想用与某一最大值有关的绝对误差形式,则可以把 yscal 的元素设置成那些最大值。对于那些不是"非常"靠近过 0 点的情况,有一种很有用的技巧,可以得到恒定的相对误差,那就是把 yscal[i]设置成和 |y[i]| + |h × dydx[i]| 相等即可(下面的程序 odeint 便是这样处理的)。

下面是更关键的技术要点。必须考虑 yscal 的另外一种可能性。目前所提到的误差准则都是"局部"的,是考虑每一步的误差。在某些实际情况下,通常想计算"全局"累积误差,从积分的起点到终点;以及在最坏情况下,所有的误差都取同一符号进行累加。这样,步长 h 越小,所要处理的值 Δ_0 也越小。这是因为从起点到终点,x 会有更多的步数。在这种情况下,就需要让 yscal 和 h 成比例,得到类似下面的式子:

$$\Delta_0 = \epsilon h \times \texttt{dydx[i]} \tag{16.2.9}$$

这便使相对精度ϵ不是作用于 y,而是(更加严格一些)作用于每一步值的增量。回看一下(16.2.7)式。如果 Δ_0 隐含和 h 成比例,则指数 0.20 就不再正确了:当步长值从一个很大的值减小时,当 yscal 也随新的预测值 h_1 改变时,这个 h_1 便达不到所需的精度。因此必须把原来的 0.20 = 1/5 放大到 0.25 = 1/4 才能正常使用。

指数 0.20 和 0.25 实际上并没有很大不同。这促使我们采纳了下面更合理的逼近方法,作为用户也就没有必要事先知道用步长来放缩 yscal 的值。无论何时减小步长,我们都用较大的指数值(不管需要不需要);而增加步长时,用较小一些的指数值。而且,因为对误差的估计不是精确的,只和 h 的主项阶的精度有关,所以建议加入一个安全系数 S。S 的值比单位 1 小百分之几。这样,(16.2.7)式便替换成

$$h_0 = \begin{cases} Sh_1 \left|\dfrac{\Delta_0}{\Delta_1}\right|^{0.20} & \Delta_0 \geqslant \Delta_1 \\ Sh_1 \left|\dfrac{\Delta_0}{\Delta_1}\right|^{0.25} & \Delta_0 < \Delta_1 \end{cases} \tag{16.2.10}$$

在实际应用中我们发现这种表示是有效的。

下面是采用了"控制计算质量"的 Runge-Kutta 方法的步进程序。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::rkqs(Vec_IO_DP &y, Vec_IO_DP &dydx, DP &x, const DP htry,
    const DP eps, Vec_I_DP &yscal, DP &hdid, DP &hnext,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &))
```

五阶 Runge-Kutta 方法具有截断误差的监控，以保证精度和调整步长。输入的是在自变量 x 处的初始值对应的因变向量 y[0..n-1]和其导数 dydx[0..n-1]，尝试的步长为 htry，所需精度为 eps，对误差进行标度的向量为 yscal[0..n-1]。输出中，y 和 x 由其新值替换，hdid 是实际采用的步长，hnext 是估算的下一步的步长值。derivs 是由用户提供的程序，用于计算右边的导数。

```
{
    const DP SAFETY=0.9, PGROW=-0.2, PSHRNK=-0.25, ERRCON=1.89e-4;
    ERRCON 的值等于 (5/SAFETY) 的 (1/PGROW) 次幂，参见下面的用法。
    int i;
    DP errmax,h,htemp,xnew;

    int n=y.size();
    h=htry;                                 设置步长为初始实验值。
    Vec_DP yerr(n),ytemp(n);
    for (;;) {
        rkck(y,dydx,x,h,ytemp,yerr,derivs);       进行一步。
        errmax=0.0;                         估计精度。
        for (i=0;i<n;i++) errmax=MAX(errmax,fabs(yerr[i]/yscal[i]));
        errmax /= eps;                      选择相应的容限。
        if (errmax <= 1.0) break;           成功，计算下一步大小。
        htemp=SAFETY*h*pow(errmax,PSHRNK);
        截断误差太大，减小步长。
        h=(h >= 0.0 ? MAX(htemp,0.1*h) : MIN(htemp,0.1*h));
        小于 10 的系数。
        xnew=x+h;
        if (xnew == x) nrerror("stepsize underflow in rkqs");
    }
    if (errmax > ERRCON) hnext=SAFETY*h*pow(errmax,PGROW);
    else hnext=5.0*h;                       不大于 5 增量。
    x += (hdid=h);
    for (i=0;i<n;i++) y[i]=ytemp[i];
}
```

程序 rkqs 要调用程序 rkck，它采用了 Cash-Karp Runge-Kutta 步骤：

```
#include "nr.h"

void NR::rkck(Vec_I_DP &y, Vec_I_DP &dydx, const DP x,
    const DP h, Vec_O_DP &yout, Vec_O_DP &yerr,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &))
```

给定在各 x 点处的 n 个变量 y[0..n-1]及其导数值 dydx[0..n-1]，用五阶 Cash-Karp Runge-Kutta 方法对间隔 h 推导解，并返回增加的变量 yout[0..n-1]。同时，用嵌入四阶算法将局部截断误差返回到 yout 中。用户提供程序 derivs(x,y,dydx)，它返回 x 点的导数值 dydx。

```
{
    static const DP a2=0.2, a3=0.3, a4=0.6, a5=1.0, a6=0.875,
        b21=0.2, b31=3.0/40.0, b32=9.0/40.0, b41=0.3, b42 = -0.9,
        b43=1.2, b51 = -11.0/54.0, b52=2.5, b53 = -70.0/27.0,
        b54=35.0/27.0, b61=1631.0/55296.0, b62=175.0/512.0,
        b63=575.0/13824.0, b64=44275.0/110592.0, b65=253.0/4096.0,
        c1=37.0/378.0, c3=250.0/621.0, c4=125.0/594.0, c6=512.0/1771.0,
        dc1=c1-2825.0/27648.0, dc3=c3-18575.0/48384.0,
        dc4=c4-13525.0/55296.0, dc5 = -277.00/14336.0, dc6=c6-0.25;
    int i;

    int n=y.size();
    Vec_DP ak2(n),ak3(n),ak4(n),ak5(n),ak6(n),ytemp(n);
    for (i=0;i<n;i++)                       第一步。
        ytemp[i]=y[i]+b21*h*dydx[i];
    derivs(x+a2*h,ytemp,ak2);               第二步。
    for (i=0;i<n;i++)
        ytemp[i]=y[i]+h*(b31*dydx[i]+b32*ak2[i]);
    derivs(x+a3*h,ytemp,ak3);               第三步。
    for (i=0;i<n;i++)
        ytemp[i]=y[i]+h*(b41*dydx[i]+b42*ak2[i]+b43*ak3[i]);
    derivs(x+a4*h,ytemp,ak4);               第四步。
```

```
    for (i=0;i<n;i++)
        ytemp[i]=y[i]+h*(b61*dydx[i]+b62*ak2[i]+b63*ak3[i]+b64*ak4[i]);
    derivs(x+a5*h,ytemp,ak5);          第五步。
    for (i=0;i<n;i++)
        ytemp[i]=y[i]+h*(b61*dydx[i]+b62*ak2[i]+b63*ak3[i]+b64*ak4[i]+b65*ak5[i])
    derivs(x+a6*h,ytemp,ak6);          第六步。
    for (i=0;i<n;i++)                  用适当的权重累积增量。
        yout[i]=y[i]+h*(c1*dydx[i]+c3*ak3[i]+c4*ak4[i]+c6*ak6[i]);
    for (i=0;i<n;i++)
        yerr[i]=h*(dc1*dydx[i]+dc3*ak3[i]+dc4*ak4[i]+dc5*ak5[i]+dc6*ak6[i]);
        估计误差，作为四阶方法和五阶方法的差别。
}
```

注意，上面的程序都是单精度的，在定义 eps 时要注意，应利用 Gilbert 和 Sullivan 法则：程序把步长取得很小，使各个 hy' 加上各个 y，就像加了零一样，程序始终能取得近似的误差。于是程序来回地运算，选用尤穷多个小步长，而丝毫不会改变一点因变量的值(为了避免这种严重错误，可以监视是否出现异常小的步长，或一步步计算时因变量值是不是没有变化。在个人工作站上，可以判定执行这个程序的时间不会太长)。

下面是一个成熟的自适应步长控制 Runge-Kutta 方法的"驱动程序"。推荐用这个程序或类似的程序来处理各种问题，包括普通常微分方程和常微分方程组，以及有限积分(增强了第 4 章中这种方法的功能)。对于中间结果的存储(如果想查看)，假定全程指针参数 *xp_p 和 *yp_p 已被有效初始化。因为每一步的间隔是不等的，结果只能在比 dxsav 大的间隔中存储起来。全局变量 kmax 指明了能够存储的更大的步数。如果 kmax = 0，则没有中间存储区，而且指针 xp_p 和 yp_p 不需要指向有效内存。如果超过了 kmax 值，便停止存储步长，除了终止值总要被存储的情况以外。而且这些控制只是为用户的需求而指定的。程序 odeint 应该常用于解决手头的问题。

```
#include <cmath>
#include "nr.h"
using namespace std;

extern DP dxsav;
extern int kmax,kount;
extern Vec_DP *xp_p;
extern Mat_DP *yp_p;
```

中间结果所需的存储空间。在程序调用中预置 kmax 和 dxsav。若 kmax≠0，结果将以 dxsav 的近似间隔存入数组 xp[0..kount-1]，yp[0..nvar-1][0..kmax-1]中，其中 kount 是程序 odeint 的输出。对这些变量的定义说明，对数组具有 xp[0..kmax-1]和 yp[0..nvar-1][0..kmax-1]的内存分配，都将在程序调用时给出。

```
void NR::odeint(Vec_IO_DP &ystart, const DP x1, const DP x2, const DP eps,
    const DP h1, const DP hmin, int &nok, int &nbad,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &),
    void rkqs(Vec_IO_DP &, Vec_IO_DP &, DP &, const DP, const DP,
    Vec_I_DP &, DP &, DP &, void (*)(const DP, Vec_I_DP &, Vec_O_DP &)))
```

具有自适应步长控制的 Runge-Kutta 方法驱动程序。积分的初值 ystart[0..nvar-1]从 x1 到 x2，精度为 eps，在全局变量中存储中间结果。h1 设置成第一步的预测步长，hmin 作为步长的最低限(可以是 0)，对于输出，nok 和 nbad 是进行的成功或失败(但重新计算和调整)的步数，ystart 由积分间隔末尾的值来取代。derivs 是用户提供用来计算右边求导结果的程序，rkqs 是所用的步进程序。

```
{
    const int MAXSTP=10000;
    const DP TINY=1.0e-30;
    int i,nstp;
    DP xsav,x,hnext,hdid,h;
    int nvar=ystart.size();
    Vec_DP yscal(nvar),y(nvar),dydx(nvar);
    Vec_DP &xp=*xp_p;                  另取名以简化代码。
    Mat_DP &yp=*yp_p;
    x=x1;
    h=SIGN(h1,x2-x1);
    nok = nbad = kount = 0;
    for (i=0;i<nvar;i++) y[i]=ystart[i];
    if (kmax > 0) xsav=x-dxsav*2.0;    保证第一步的存储空间。
    for (nstp=0;nstp<MAXSTP;nstp++) {
        derivs(x,y,dydx);
        for (i=0;i<nvar;i++)
```

```
            用于监测精确度的缩放，如果需要的话，这个选择可以进行修改。
            yscal[i]=fabs(y[i])+fabs(dydx[i]*h)+TINY;
        if (kmax > 0 && kount < kmax-1 && fabs(x-xsav) > fabs(dxsav)) {
            for (i=0;i<nvar;i++) yp[i][kount]=y[i];
            xp[kount++]=x;                          存入中间结果。
            xsav=x;
        }
        if ((x+h-x2)*(x+h-x1) > 0.0) h=x2-x;        如果步长超出尾端，则缩短步长。
        rkqs(y,dydx,x,h,eps,yscal,hdid,hnext,derivs);
        if (hdid == h) ++nok; else ++nbad;
        if ((x-x2)*(x2-x1) >= 0.0) {                完成了吗?
            for (i=0;i<nvar;i++) ystart[i]=y[i];
            if (kmax != 0) {
                for (i=0;i<nvar;i++) yp[i][kount]=y[i];
                xp[kount++]=x;                      存储最后一步。
            }
            return;                                 正常退出。
        }
        if (fabs(hnext) <= hmin) nrerror("Step size too small in odeint");
        h=hnext;
    }
    nrerror("Too many steps in routine odeint");
}
```

参考文献和深入阅读

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Cash, J.R., and Karp, A.H. 1990, *ACM Transactions on Mathematical Software*, vol. 16, pp. 201–222. [2]

Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall).

16.3　修正中点法

本节讨论**修正中点法**。从点 x 到点 $x+H$，通过每一步长为 h 的 n 子步来计算因变量 $y(x)$ 向量，其中

$$h = H/n \tag{16.3.1}$$

原则上，可用修正中点法作为一种常微分方程的积分器。实际上，这种方法最重要的应用是用在 16.4 节中介绍的 Bulirsch-Stoer 方法中。因此，也可以把本节当做 16.4 节的一个铺垫。

等式右边所需的估值计算，在修正中点法中需要 $n+1$ 次。这种方法的公式如下：

$$\begin{aligned} z_0 &\equiv y(x) \\ z_1 &= z_0 + hf(x, z_0) \\ z_{m+1} &= z_{m-1} + 2hf(x+mh, z_m) \qquad m = 1, 2, \ldots, n-1 \\ y(x+H) \approx y_n &\equiv \frac{1}{2}[z_n + z_{n-1} + hf(x+H, z_n)] \end{aligned} \tag{16.3.2}$$

其中所有 z 是沿每步 h 的中间近似值，而 y_n 是用于近似 $y(x+H)$ 的结果。除第一点和最后一点以外，这种方法根本来说是“中心差分”或“中点”法（和(16.1.2)式相比），因此给了个限定词“修正”。

修正中点法是一种二阶方法，像(16.1.2)式一样，但是有一个优点，每一步 h 只需要（对于比较大的 n）一个导数值，而不像二阶 Runge-Kutta 方法中需要两个导数值。也许因为(16.3.2)式的简单性，也很容易嵌入某些应用程序中。但是总的来说，修正中点法不只用在前一节中实施的自适应控制的嵌入式 Runge-Kutta 方法。

在 Bulirsch-Stoer 方法中,用修正中点法是为了从(16.3.2)式中求出一个"深刻"的解来。这已由 Gragg 证明,所得出的(16.3.2)式的误差可表达为 h 的一个级数,而且只含 h 的偶次项,

$$y_n - y(x+H) = \sum_{i=1}^{\infty} \alpha_i h^{2i} \tag{16.3.3}$$

其中 H 保持常量,但在(16.3.1)式中 h 随 n 而变。这种偶次方级数的重要性在于,如果仍运用通过组合步长的情况来达到消去高阶误差项的话,每一次可以提高两阶。

例如,假设 n 是偶的,并设 $y_{n/2}$ 表示应用(16.3.1)式和(16.3.2)式,且步数为一半 $n \to n/2$ 的结果,则我们估计式

$$y(x+H) \approx \frac{4y_n - y_{n/2}}{3} \tag{16.3.4}$$

是四阶的精确度,和四阶 Runge-Kutta 方法一样,但每步步长 h 只需要 1.5 次求导估值,而不是 Runge-Kutta 的四次。暂时不要着急实现(16.3.4)式,我们还能做得更好些。

现在最好回过头来看一看 4.2 节中的程序 qsimp,并把(4.2.4)式和上面的(16.3.4)式进行一下比较,就会发现在第 4 章中,根据 Richardson 外推思想过渡到 4.3 节龙贝格(Romberg)积分中的具体实现,恰好类似于从本节到下节的过渡。

下面是修正中点法的实现,在以后将要用到。

```
#include "nr.h"

void NR::mmid(Vec_I_DP &y, Vec_I_DP &dydx, const DP xs, const DP htot,
    const int nstep, Vec_O_DP &yout,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &))
修改中点法。在 xs,输入因变量向量 y[0..nvar-1]和它的求导向量 dydx[0..nvar-1]。输入还有总的步数 htot,以及每
一步中的子步的数目 nstep。输出以 yout[0..nvar-1]返回。
{
    int i,n;
    DP x,swap,h2,h;

    int nvar=y.size();
    Vec_DP ym(nvar),yn(nvar);
    h=htot/nstep;
    for (i=0;i<nvar;i++) {
        ym[i]=y[i];
        yn[i]=y[i]+h*dydx[i];          第一步。
    }
    x=xs+h;
    derivs(x,yn,yout);                 用 yout 暂存导数值。
    h2=2.0*h;
    for (n=1;n<nstep;n++) {            一般步。
        for (i=0;i<nvar;i++) {
            swap=ym[i]+h2*yout[i];
            ym[i]=yn[i];
            yn[i]=swap;
        }
        x += h;
        derivs(x,yn,yout);
    }
    for (i=0;i<nvar;i++)               最后一步。
        yout[i]=0.5*(ym[i]+yn[i]+h*yout[i]);
}
```

参考文献和深入阅读

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.4.

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §7.2.12.

16.4　Richardson 外推法和 Bulirsch-Stoer 方法

在本节介绍的方法不是针对含有非光滑函数的微分方程组的。例如，读者也许会碰到这样的差分方程。方程右边的函数值是通过查找表或插值来定义的。如果是这样的话，可以回到 Runge-Kutta 方法用自适应步长来计算。那种方法对于不固定的或不连续的定义域的情况处理得很好。同时也是对方程组的一种快速而较粗糙、低精度的解法。另外一点要说明的是，在本节介绍的方法对于在积分区间中有奇异点的微分方程处理得不是特别好。一个正规解需要很小心地逐渐逼近这些点，而带自适应步长的 Runge-Kutta 方法有时处理这个问题很有效。更普遍的是，采用其他特别的方法来解决这些问题，但这已超出了本书的讨论范围。

除了上边的两个缺点以外，我们认为本节中讨论的 Bulirsch-Stoer 方法，是用最小的计算达到常微分方程求解高精度的一个目前来说最好的方法(有时可能有例外，这种例外在实际中很少发生，将在 16.7 节讨论)。

这个方法有三个关键思想，第一个是 **Richardson 延迟趋向于极限**，我们已在 4.3 节中的龙贝格积分中遇到过了。其思路是把数值计算的结果，作为某一像步长 h 这样的可调整参数的解析函数(如果是一个比较复杂的)。这个解析函数可以通过进行不同 h 值的计算来估测，这里的 h 值没有必要为达到我们所需的精度而取得太小。当得到这个函数以后，把函数拟合成某个解析式，然后在很重要的点 $h=0$ 处进行估算(参见图 16.4.1)。Richardson 外推法确实是一种不错的方法。

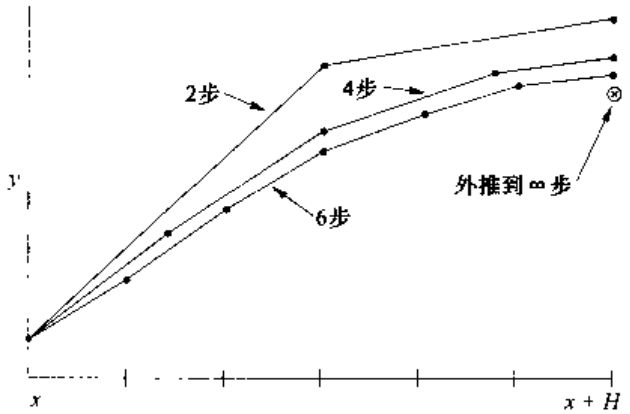


图 16.4.1　Bulirsch-Stoer 方法中所用的 Richardson 外推法。一个大的间隔 H 由逐渐细分的子步序列来组成。其结果是对设想的无穷细分的子步进行外推。在 Bulirsch-Stoer 方法中，积分法是采用修正中点法，外推法是采用有理函数或多项式的外推法

第二个关键思想是用什么样的匹配函数。Bulirsch 和 Stoer 首先认识到了 Richardson 方法中**有理函数外推**的作用。有理函数的外推突破了幂级数和其收敛半径的限制，只受复平面的第一次极点的影响，即使在 h 的各次项数量级相差不大的情况下，有理函数匹配仍能很好地近似解析函数，换言之，h 可以取得较大，让这种方法的"阶次"失去意义，而这种方法仍能取得很好的效果。此外，最近更多的经验表明：对光滑问题直接用多项式外推比用有理函数外推更有效。因此，我们将用多项式外推作为默认，但是下面的程序 bsstep 允许其他的外推形式来做简单的替换。关于这一点可以重新参阅 3.1 节～3.2 节，在那里已经讨论过有理函数外推的问题。

第三个关键思想在前一节已经讨论过，也就是让误差函数是严格偶函数，使有理函数或多项式逼近为变量 h^2 的形式而非 h 的形式。

把这三个主要思想综合在一起，便得出了 **Bulirsch-Stoer 方法**[1]，单步 Bulirsch-Stoer 方法是从 x 计算到 $x+H$，其中的 H 是一个比较大(而不是近似无穷小)的距离。每一单步是由很多(几百上千个)修正中点法的子步组成，它们都采用近拟于零步长的有理函数的外推算法。估算间隔 H 的序列是由 n(子步长数目)的递增序列组成的。通常 Bulirsch 和 Stoer 提出的 n 序列为

$$n = 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, \ldots, [n_j = 2n_{j-2}], \ldots \tag{16.4.1}$$

Deuflhard[2,3]最近发表的文章中建议使用序列

$$n = 2, 4, 6, 8, 10, 12, 14, \ldots, [n_j = 2(j+1)], \ldots \tag{16.4.2}$$

通常它更为有效。对于每一步来说,事先不知道沿这个序列要执行到多远。在顺序执行每一个 n 值以后,我们试图用有理外推函数进行试验。外推结果给出了外推值和误差估计。如果误差不满足要求,则采用更大的 n 值;如果误差满足要求,则继续下一步,从 $n=2$ 重新开始。

当然必须有一个上限,超过上限以后,可得出结论。在沿间隔 H 以内的路径中有障碍,因此必须缩小 H 值而不是对 H 进行更细的划分。在下面的程序实施中,所要试的 n 的最大数目称为 KMAXX。通常取这个值等于 8,因为(16.4.2)式的序列中第 8 个值是 16,所以这就是我们允许 H 的最大细划分数。

和 Runge-Kutta 方法一样,这是通过监视内部变元的一致性,以及根据局部截断误差的允许误差范围,来调整步长大小以控制误差的大小。修正中点法积分的每一个新的结果允许像 3.1 节中的图表那样被附加的一组对角元扩充。每一步增加的新校正量被采用为(保守的)误差估计。应该怎样用这个误差估值来调整步长呢?目前最好的方案是 Deuflhard[2,3]提出的。出于完整性,摘录如下。

假设从外推表第 k 列(也就是第 $k+1$ 行)返回的误差估计的绝对值为$\epsilon_{k+1,k}$。误差控制是通过在接受当前步时需要满足

$$\epsilon_{k+1,k} < \epsilon \tag{16.4.3}$$

来达到,其中ϵ是允许的误差范围。对于(16.4.2)式的偶序列,该方法的阶次是 $2k+3$:

$$\epsilon_{k+1,k} \sim H^{2k+3} \tag{16.4.4}$$

这样,为了在某一指定列 k 达到收敛,一个新步长 H_k 的简单估计值则为

$$H_k = H\left(\frac{\epsilon}{\epsilon_{k+1,k}}\right)^{1/(2k+3)} \tag{16.4.5}$$

我们的目的是,使哪一个 k 列达到收敛性呢?比较一下不同 k 列所需的工作量。假设 A_k 为达到外推图表的 k 行所需要的计算量,而 A_{k+1} 是达到 k 列的工作量。假定工作量是由微分方程右边的函数估值来决定的,对于 H 中 n_k 细分,函数估值的数目可由下面的递推式求出:

$$\begin{aligned} A_0 &= n_0 + 1 \\ A_{k+1} &= A_k + n_{k+1} \end{aligned} \tag{16.4.6}$$

达到 k 列的每一小步的工作量是A_{k+1}/H_k,用因子 H 进行无量纲化,并写做

$$W_k = \frac{A_{k+1}}{H_k} H \tag{16.4.7}$$

$$= A_{k+1}\left(\frac{\epsilon_{k+1,k}}{\epsilon}\right)^{1/(2k+3)} \tag{16.4.8}$$

在积分的过程中,W_k 的值可以进行计算。这样最优列下标 q 由下式定义:

$$W_q = \min_{k=0,\ldots,k_f} W_k \tag{16.4.9}$$

其中 k_f 是最后一列,在这一列中应满足(16.4.3)式的误差范围。由(16.4.9)式定义的 q 决定了用于下一个基本的步长值 H_q。这样可望在最优列 q 处达到收敛。

对于该算法必须有两个限制:

- 如果以前的 H"太小",则 k_f 也会"太小"。这样,q 也"太小"就可能需要增大 H 以便能在列 $q > k_f$ 上达到收敛。
- 如果当前的 H"太大",在当前步无法收敛,则必须减小 H 值。可以通过检测$\epsilon_{k+1,k}$的值来发现这一点,使得可以尽可能早地停止当前步。

Deuflhard 的文章中用通信原理中的思想,确定外推的"平均收敛期望"来处理这两个问题。在他的模型中,他提出了一个校正因子$\alpha(k,q)$,通过和 H_k 相乘以后试图在 q 列处得到收敛。因子$\alpha(k,q)$只依赖于ϵ及序列$\{n_i\}$,这样可以在初始化的时候一次算出:

$$\alpha(k,q)=\epsilon^{\frac{A_{k+1}-A_{q+1}}{(2k+3)(A_{q+1}-A_{0+1})}}\qquad \text{for}\quad k<q \tag{16.4.10}$$

而且$\alpha(q,q)=1$。

现在处理第一个问题，假设收敛发生在列 $q=k_f$。我们希望增加步长使在 $q+1$ 列达到收敛，而不是用 H_q 来进行下一步操作。因为不能计算出可用的 H_{q+1} 值，故而把它估计为

$$H_{q+1}=H_q\alpha(q,q+1) \tag{16.4.11}$$

根据(16.4.7)式，这种替代是有效的，即可以减小每一小步的计算量。如果

$$\frac{A_{q+1}}{H_q}>\frac{A_{q+2}}{H_{q+1}} \tag{16.4.12}$$

或

$$A_{q+1}\alpha(q,q+1)>A_{q+2} \tag{16.4.13}$$

在初始化过程中，这种不等关系对于 $q=0,1,\ldots$ 可以检查出来以决定 $k_{\max}$，即最大允许的列。这样，当(16.4.12)式满足时，用 H_{q+1} 总是有效的(实践中把 $k_{\max}$ 限制量为 8，即使当ϵ很小时，在这种情况下有效性不会有多大改进，而舍入误差却成了问题)。

步长减小的问题可以通过在当前步中估算步长值

$$\bar{H}_k\equiv H_k\alpha(k,q),\qquad k=0,\ldots,q-1 \tag{16.4.14}$$

来完成。$\bar{H}$ 在最优列 q 中，达到收敛的步长的估计值。如果 $\bar{H}_k$"太小"，则不再继续当前步，而是用 $\bar{H}_k$ 重新开始。"太小"由以下规则进行制定：

$$H_k\alpha(k,q+1)<H \tag{16.4.15}$$

α满足$\alpha(k,q+1)>\alpha(k,q)$。

在第一步中，我们并没有关于解的任何信息，步长减小的检查是针对所有 k 来完成的。以后，在一个"序号窗口"中检测收敛和可能的步长减小，该"序号窗口"如下：

$$\max(0,q-1)\leqslant k\leqslant\min(k_{\max},q+1) \tag{16.4.16}$$

之所以这样定义窗口是因为，一方面，如果对 $k<q-1$ 处似乎像收敛的样子，那往往是假的，这是在外推过程中小误差偶然作用的结果；另一方面，如果需要在 $k=q+1$ 以外的地方达到收敛，这个局部收敛模型显然不好，就需要减小步长重建此模型。

在程序 bsstep 中，这些各种各样的检测实际上是用

$$\epsilon(k)\equiv\frac{H}{H_k}=\left(\frac{\epsilon_{k+1,k}}{\epsilon}\right)^{1/(2k+3)} \tag{16.4.17}$$

在程序中命名为 err[k]的值来实现的。和以往一样，在步长选择中包括了一个"安全系数"，这是通过用 0.25ϵ 来代替ϵ达到的，其他的安全系数在程序注释中说明。

注意，尽管最优收敛列的限制为最多一次递增一步，但根据(16.4.9)式，顺序是允许打乱的。这就使得这种方法对不连续的问题有一定适应性。

再次提醒，变元的**尺度比例**通常对微分方程的成功求解是很关键的，在(16.2.8)式以后讨论的尺度比例技巧是一种很好的通用的选择，但并非是完全正确的。以变量最大值为尺度比例是较稳健的，但需要有一些先验的信息。

下面的程序实现了 Bulirsch-Stoer 方法的一步计算，它和控制计算质量的 Runge-Kutta 方法步进程序 rkqs 所调用的变量序列是完全一样的。这就是说，在 16.2 节中的驱动程序 odeint 能够像调用 Runge-Kutta 方法程序一样调用 Bulirsch-Stoer 方法程序，只需在 odeint 程序中将 rkqs 函数用 bsstep 替换，程序 bsstep 调用 mmid 采用修正中点序列，然后调用下面的 pzextr 进行多项式外推。

```
#include <cmath>
#include "nr.h"
using namespace std;

Vec_DP *x_p;
Mat_DP *d_p;
```

指针指向 pzextr 或 rzextr 调用的矩阵和向量。

```
void NR::bsstep(Vec_IO_DP &y, Vec_IO_DP &dydx, DP &xx, const DP htry,
    const DP eps, Vec_I_DP &yscal, DP &hdid, DP &hnext,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &))
```

Bulirsch-Stoer 步通过监视局部截尾误差来保证精度和调整步长。输入是自变量 x 的起始值处的因变向量 y[0..nv-1]及其导数 dydx[0..nv-1]。输入还有尝试的步长 htry,所需精度 eps,测度误差的向量 yscal[0..nv-1]。在输出中,y 和 x 由其新值替换,hdid 是实际采用的步长值,hnext 是步长的下一步估算值。derivs 是用户提供的程序,用以计算等式右端的导数。注意,在步骤成功时,把从前面步返回的 hnext 赋给下一连续步的 htry,以备此程序被 odeint 调用。

```
{
    const int KMAXX=8, IMAXX=(KMAXX+1);
    const DP SAFE1=0.25, SAFE2=0.7, REDMAX=1.0e-5, REDMIN=0.7;
    const DP TINY=1.0e-30, SCALMX=0.1;
```

其中 KMAXX 是外推中用到的行的最大数;SAFE1 和 SAFE2 是安全系数;
REDMAX 是步长变化的最大系数;REDMIN 是步长变化的最小系数;
TINY 避免被 0 除;并且 1/SCALMX 是步长增大的最大系数。

```
    static const int nseq_d[IMAXX]={2,4,6,8,10,12,14,16,18};
    static int first=1,kmax,kopt;
    static DP epsold = -1.0,xnew;
    static Vec_DP a(IMAXX);
    static Mat_DP alf(KMAXX,KMAXX);
    bool exitflag=false;
    int i,iq,k,kk,km,reduct;
    DP eps1,errmax,fact,h,red,scale,work,wrkmin,xest;
    Vec_INT nseq(nseq_d,IMAXX);
    Vec_DP err(KMAXX);

    int nv=y.size();
    Vec_DP yerr(nv),ysav(nv),yseq(nv);
    x_p=new Vec_DP(KMAXX);
    d_p=new Mat_DP(nv,KMAXX);
    if (eps != epsold) {                          新的容差,所以重新初始化。
        hnext = xnew = -1.0e29;                   "不可能"的值。
        eps1=SAFE1*eps;
        a[0]=nseq[0]+1;                           计算有效系数 A_k。
        for (k=0;k<KMAXX;k++) a[k+1]=a[k]+nseq[k+1];
    for (iq=1;iq<KMAXX;iq++) {                    计算 α(k,q)。
        for (k=0;k<iq;k++)
            alf[k][iq]=pow(eps1,(a[k+1]-a[iq+1])/
                ((a[iq+1]-a[0]+1.0)*(2*k+3)));
    }
    epsold=eps;
    for (kopt=1;kopt<KMAXX-1;kopt++)              确定收敛的最优行数。
        if (a[kopt+1] > a[kopt]*alf[kopt-1][kopt]) break;
    kmax=kopt;
}
h=htry;
for (i=0;i<nv;i++) ysav[i]=y[i];                  存储起始值。
if (xx != xnew || h != hnext) {                   新的步长或新的积分;
    first=1;                                      重新建立序号窗口。
    kopt=kmax;
}
reduct=0;
for (;;) {
    for (k=0;k<=kmax;k++) {                       估算修改中点积分序列的值。
        xnew=xx+h;
        if (xnew == xx) nrerror("step size underflow in bsstep");
        mmid(ysav,dydx,xx,h,nseq[k],yseq,derivs);
        xest=SQR(h/nseq[k]);                      开方,因为误差级数是偶数。
        pzextr(k,xest,yseq,y,yerr);               执行外推。
        if (k != 0) {                             计算归一化误差估计值 ε(k)。
            errmax=TINY;
```

```
                    for (i=0;i<nv;i++) errmax=MAX(errmax,fabs(yerr[i]/yscal[i]));
                    errmax /= eps;                          相对于容限缩放误差。
                    km=k-1;
                    err[km]=pow(errmax/SAFE1,1.0/(2*km+3));
                }
                if (k != 0 && (k >= kopt-1 || first)) {     在序号窗口中。
                    if (errmax < 1.0) {                     收敛。
                        exitflag=true;
                        break;
                    }
                    if (k == kmax || k == kopt+1) {         检查可能的步长缩减。
                        red=SAFE2/err[km];
                        break;
                    }
                    else if (k == kopt && alf[kopt-1][kopt] < err[km]) {
                        red=1.0/err[km];
                        break;
                    }
                    else if (kopt == kmax && alf[km][kmax-1] < err[km]) {
                        red=alf[km][kmax-1]*SAFE2/err[km];
                        break;
                    }
                    else if (alf[km][kopt] < err[km]) {
                        red=alf[km][kopt-1]/err[km];
                        break;
                    }
                }
            }
            if (exitflag) break;
            red=MIN(red,REDMIN);                            由最小值 REDMIN 和最大值 REDMAX
            red=MAX(red,REDMAX);                            减小步长。
            h *= red;
            reduct=1;
        }                                                   再试。
        xx=xnew;                                            成功完成一步。
        hdid=h;
        first=0;
        wrkmin=1.0e35;                                      为收敛计算最优行和相应步长。
        for (kk=0;kk<=km;kk++) {
            fact=MAX(err[kk],SCALMX);
            work=fact*a[kk+1];
            if (work < wrkmin) {
                scale=fact;
                wrkmin=work;
                kopt=kk+1;
            }
        }
        hnext=h/scale;
        if (kopt >= k && kopt != kmax && !reduct) {
        Check for possible order increase, but not if stepsize was just reduced.
            fact=MAX(scale/alf[kopt-1][kopt],SCALMX);
            if (a[kopt+1]*fact <= wrkmin) {
                hnext=h/fact;
                kopt++;
            }
        }
        delete d_p;
        delete x_p;
    }
```

多项式外推程序基于和 3.1 节 polint 相同的算法。它是比较简单的，因为总是外推到零，而非外推到任意值。但它也是较复杂的，因为必须单独地对某数值向量中的每一分量进行外推。

```
#include "nr.h"

extern Vec_DP *x_p;                         在 bsstep 中定义。
extern Mat_DP *d_p;

void NR::pzextr(const int iest, const DP xest, Vec_I_DP &yest, Vec_O_DP &yz,
    Vec_O_DP &dy)
```

```
使用多项式外推来计算 nv 个函数在 x = 0 处的值,它是通过将多项式拟合成一列具有逐次较小值 x = xest 和相应的函数
向量 yest[0..nv-1]的序列来计算的。这次是调用序列中的第 iest 次调用。所外推的函数值以 yz[0..nv-1]输出,它们
的估计值误差输出为 dy[0..nv-1]。
{
    int j,k1;
    DP q,f2,f1,delta;

    int nv=yz.size();
    Vec_DP c(nv);
    Vec_DP &x=*x_p;                          另取名以简化代码。
    Mat_DP &d=*d_p;
    x[iest]=xest;                            保存当前自变量。
    for (j=0;j<nv;j++) dy[j]=yz[j]=yest[j];
    if (iest == 0) {                         存储第一列估计值。
        for (j=0;j<nv;j++) d[j][0]=yest[j];
    } else {
        for (j=0;j<nv;j++) c[j]=yest[j];
        for (k1=0;k1<iest;k1++) {
            delta=1.0/(x[iest-k1-1]-xest);
            f1=xest*delta;
            f2=x[iest-k1-1]*delta;
            for (j=0;j<nv;j++) {             将表 1 的对角元延伸得更多。
                q=d[j][k1];
                d[j][k1]=dy[j];
                delta=c[j]-q;
                dy[j]=f1*delta;
                c[j]=f2*delta;
                yz[j] += dy[j];
            }
        }
        for (j=0;j<nv;j++) d[j][iest]=dy[j];
    }
}
```

目前人们在 Bulirsch-Stoer 方法中用多项式外推,而不用有理函数外推。但是我们认为,这种观点是在各种问题的测试中得出的,并非是某一种方法确实"更好"。相应地,我们也提供了有理函数外推的最优程序 rzextr,它完全可替代上面的 pzextr 程序。

```
#include "nr.h"

extern Vec_DP *x_p;                          在 bsstep 中定义。
extern Mat_DP *d_p;

void NR::rzextr(const int iest, const DP xest, Vec_I_DP &yest, Vec_O_DP &yz,
    Vec_O_DP &dy)
本程序可完全替代程序 pzextr,采用对角有理函数外推替换多项式外推。
{
    int j,k,nv;
    DP yy,v,ddy,c,b1,b;

    nv=yz.size();
    Vec_DP fx(iest+1);
    Vec_DP &x=*x_p;                          另取名以简化代码。
    Mat_DP &d=*d_p;
    x[iest]=xest;                            保存当前自变量。
    if (iest == 0)
        for (j=0;j<nv;j++) {
            yz[j]=yest[j];
            d[j][0]=yest[j];
            dy[j]=yest[j];
        }
    else {
        for (k=0;k<iest;k++)
            fx[k+1]=x[iest-(k+1)]/xest;
        for (j=0;j<nv;j++) {                 求表中下一对角元的值。
            v=d[j][0];
            d[j][0]=yy=c=yest[j];
            for (k=1;k<=iest;k++) {
                b1=fx[k]*v;
                b=b1-c;
```

```
                if (b != 0.0) {
                    b=(c-v)/b;
                    ddy=c*b;
                    c=b1*b;
                } else                    当心，必须避免零作除数。
                    ddy=v;
                if (k != iest) v=d[j][k];
                d[j][k]=ddy;
                yy += ddy;
            }
            dy[j]=ddy;
            yz[j]=yy;
        }
    }
}
```

参考文献和深入阅读

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §7.2.14. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.2.

Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 399–422. [2]

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535. [3]

16.5　深入讨论：二阶守恒方程组

通常情况下，当要解一组高阶微分方程组时，最好把它化成一个一阶微分方程组，如 16.0 节中讨论的那样。但实践中通常有一类特殊的方程组，通过直接差分，效率可提高两倍。这一类方程组是二阶系统，其在等式右端没有出现导数：

$$y'' = f(x, y), \qquad y(x_0) = y_0, \qquad y'(x_0) = z_0 \tag{16.5.1}$$

和以往一样，其中的 y 可以代表一个数值的向量。

离散化这样的问题，最通用的方法可追溯到 1907 年提出的 Stoermer 法则。设 $h = H/m$，我们有

$$\begin{aligned} y_1 &= y_0 + h[z_0 + \tfrac{1}{2}hf(x_0, y_0)] \\ y_{k+1} - 2y_k + y_{k-1} &= h^2 f(x_0 + kh, y_k), \qquad k = 1, \ldots, m-1 \\ z_m &= (y_m - y_{m-1})/h + \tfrac{1}{2}hf(x_0 + H, y_m) \end{aligned} \tag{16.5.2}$$

其中 z_m 是 $y'(x_0 + H)$。Henrici 说明了如何利用值 $\Delta_k \equiv y_{k+1} - y_k$ 重写(16.5.2)式来减小舍入误差。首先

$$\begin{aligned} \Delta_0 &= h[z_0 + \tfrac{1}{2}hf(x_0, y_0)] \\ y_1 &= y_0 + \Delta_0 \end{aligned} \tag{16.5.3}$$

然后对于 $k = 1, \ldots, m-1$，设

$$\begin{aligned} \Delta_k &= \Delta_{k-1} + h^2 f(x_0 + kh, y_k) \\ y_{k+1} &= y_k + \Delta_k \end{aligned} \tag{16.5.4}$$

最后从下式计算导数

$$z_m = \Delta_{m-1}/h + \tfrac{1}{2}hf(x_0 + H, y_m) \tag{16.5.5}$$

Gragg 重新说明了(16.5.3)式～(16.5.5)式包含的系列误差只含有 h 的偶数次方，这样，这种方法逻辑上便可适用于 Bulirsch-Stoer 方法，用下列程序 stoerm 代替 mmid：

```
#include "nr.h"

void NR::stoerm(Vec_I_DP &y, Vec_I_DP &d2y, const DP xs,
    const DP htot, const int nstep, Vec_O_DP &yout,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &))
```

对于 n = nv/2 个方程,用 Stoemer 法则求解 $y'' = f(x, y)$ 的程序。一个输入 y[0..nv-1]在前 n 个元素中包含 y,在后 n 个元素中包含在 xs 处的 y' 值,d2y[0..nv-1]前 n 个元素包含右边的函数值 f(也是在 xs 处),后 n 个元素没有用到。另一输入量 htot,总共要采取的步数及 nstep,要采用的子步数。输出返回在 yout[0..nv-1]中,和 y 的存储结构一样,derivs 是用户提供用于计算 f 的程序。

```
{
    int i,nn,n,neqns;
    DP h,h2,halfh,x;

    int nv=y.size();
    Vec_DP ytemp(nv);
    h=htot/nstep;                       这一步的步长。
    halfh=0.5*h;
    neqns=nv/2;                         方程个数。
    for (i=0;i<neqns;i++) {             第一步。
        n=neqns+i;
        ytemp[i]=y[i]+(ytemp[n]=h*(y[n]+halfh*d2y[i]));
    }
    x=xs+h;
    derivs(x,ytemp,yout);               利用 yout 暂时存储导数值。
    h2=h*h;
    for (nn=1;nn<nstep;nn++) {           一般情况。
        for (i=0;i<neqns;i++)
            ytemp[i] += (ytemp[neqns+i] += h2*yout[i]);
        x += h;
        derivs(x,ytemp,yout);
    }
    for (i=0;i<neqns;i++) {             最后一步。
        n=neqns+i;
        yout[n]=ytemp[n]/h+halfh*yout[i];
        yout[i]=ytemp[i];
    }
}
```

注意,为了和 bsstep 兼容,数组 y 和 d2y 的长度对 n 个二阶方程是 $2n$。y 值存储在 y 的前 n 个元素中,而后 n 个元素中存储第一阶求导值。等式右边的 f 存储在 d2y 矩阵的前 n 个元素中,后 n 个元素未使用。在这样的存储结构下,可以使用 bsstep,只需简单地用同样的参数值调用 stoerm 来代替对 mmid 的调用,只是要注意 bsstep 的参数 nv 设置成 $2n$。也应使用 Deuflhard 建议的更有效的步长序列:

$$n = 1, 2, 3, 4, 5, \ldots \tag{16.5.6}$$

并在 bsstep 中设置 KMAXX = 12。

参考文献和深入阅读

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535.

16.6 刚性方程组

当处理多于一个的一阶微分方程时,**刚性方程组**也许就出现了。刚性问题通常出现在自变量的取值有两个或多个不同尺度,而因变量正随此而变化的问题中。例如,考虑下面的方程组:

$$\begin{aligned} u' &= 998u + 1998v \\ v' &= -999u - 1999v \end{aligned} \tag{16.6.1}$$

其边界值条件为

$$u(0) = 1 \qquad v(0) = 0 \tag{16.6.2}$$

通过变形

$$u = 2y - z \qquad v = -y + z \tag{16.6.3}$$

得到解

$$u = 2e^{-x} - e^{-1000x}$$
$$v = -e^{-x} + e^{-1000x} \tag{16.6.4}$$

如果用本章内到目前为止介绍的任意一种方法，对(16.6.1)方程组进行积分求解，其中的 e^{-1000x} 项需要步长 $h \ll 1/1000$ 才能使方法稳定(其原因在下面介绍)。即使当 x 离开原点，在决定 u 和 v 值的过程中 e^{-1000x} 项完全可以忽略时也得这样(参见图 16.6.1)。

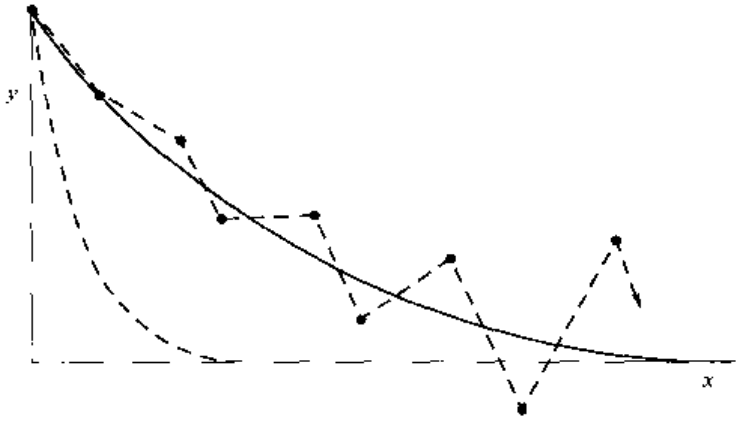


图 16.6.1　积分刚性方程中遇到不稳定性的例子，这里假定方程有两组解，如图中实线和短横线所示。尽管初始条件是提供满足实线解的，但积分的稳定性(如图中不稳定的带点的线段序列)则是由变化很快的短横线解所决定的，即使在那个解几乎趋近于零时也一样。要解决这个问题可以利用隐式积分

这是刚性方程的常见问题：需要在解的取值较小的度量范围内变化步长，以保持积分的稳定性，尽管积分精度可以允许一个比较大的步长值。

下面看一下怎么解决这个问题。考虑下面这个方程

$$y' = -cy \tag{16.6.5}$$

其中 $c > 0$ 是一个常数。用隐式(或向前)欧拉格式以步长 h 积分，这个方程是

$$y_{n+1} = y_n + hy'_n = (1 - ch)y_n \tag{16.6.6}$$

这种方法称为隐式解法，因为新的值 y_{n+1} 是以旧的值 y_n 的隐含方式给出的。显然，当 $h > 2/c$ 时，这种方法是不稳定的，因为当 $n \to \infty$ 时，$|y_n| \to \infty$。

最简便的解决方法是采取隐式差分，其中在新的 y 值处估算右端项。在这种情况下，得到了**向后欧拉**格式

$$y_{n+1} = y_n + hy'_{n+1} \tag{16.6.7}$$

或

$$y_{n+1} = \frac{y_n}{1 + ch} \tag{16.6.8}$$

这种方法是绝对稳定的：即使当 $h \to \infty$ 时，亦有 $y_{n+1} \to 0$，这实际上是微分方程的正确解。如果把 x 设想成代表时间，则隐式方法将以大的步长收敛到真实的平衡解(如时间稍晚一些的解)。这种隐式方法的良好特性只对线性系统成立，但即使在一般情况下，隐式方法也能给出比较好的稳定性。当然，如果用较大的步长逼近真实解时，我们得放弃**精确度**，但还是坚持**稳定性**优先。

这些思想能够很容易地推广到常系数线性方程的情况：

$$\mathbf{y}' = -\mathbf{C} \cdot \mathbf{y} \tag{16.6.9}$$

其中 $\mathbf{C}$ 是一个正定矩阵。隐含差分法为

$$\mathbf{y}_{n+1} = (\mathbf{1} - \mathbf{C}h) \cdot \mathbf{y}_n \tag{16.6.10}$$

一个矩阵 $\mathbf{A}^n$ 当 $n \to \infty$ 时趋近于零，仅当 $\mathbf{A}$ 的最大特征值小于单位 1。因此，当 $n \to \infty$ 时，$\mathbf{y}_n$ 有界的充分条件是，

仅当$\mathbf{1}-\mathbf{C}h$的最大特征值小于1,换言之

$$h<\frac{2}{\lambda_{\max}} \tag{16.6.11}$$

其中$\lambda_{\max}$是$\mathbf{C}$的最大特征值。

另一方面,隐含差分式给出了

$$\mathbf{y}_{n+1}=\mathbf{y}_n+h\mathbf{y}'_{n+1} \tag{16.6.12}$$

或

$$\mathbf{y}_{n+1}=(\mathbf{1}+\mathbf{C}h)^{-1}\cdot\mathbf{y}_n \tag{16.6.13}$$

如果$\mathbf{C}$的特征值是λ,则$(\mathbf{1}+\mathbf{C}h)^{-1}$的特征值应是$(1+\lambda h)^{-1}$,对所有$h$来说,它的量值都小于1(正定矩阵的所有特征值都是非负的)。因此这种方法对于所有的步长h都是稳定的。为达到这种稳定性,所付出的代价是每一步都对矩阵求逆。

遗憾的是,并非所有方程都是常系数的线性方程。对于诸如方程组

$$\mathbf{y}'=\mathbf{f}(\mathbf{y}) \tag{16.6.14}$$

隐式差分法导出

$$\mathbf{y}_{n+1}=\mathbf{y}_n+h\mathbf{f}(\mathbf{y}_{n+1}) \tag{16.6.15}$$

一般来说,这是一个比较复杂的非线性方程组,必须每步进行迭代处理。假设能化成线性方程组,像牛顿法中那样:

$$\mathbf{y}_{n+1}=\mathbf{y}_n+h\left[\mathbf{f}(\mathbf{y}_n)+\left.\frac{\partial\mathbf{f}}{\partial\mathbf{y}}\right|_{\mathbf{y}_n}\cdot(\mathbf{y}_{n+1}-\mathbf{y}_n)\right] \tag{16.6.16}$$

其中$\partial\mathbf{f}/\partial\mathbf{y}$是右边矩阵(雅可比矩阵)的偏导数。重新将(16.6.16)式安排一下,有

$$\mathbf{y}_{n+1}=\mathbf{y}_n+h\left[\mathbf{1}-h\frac{\partial\mathbf{f}}{\partial\mathbf{y}}\right]^{-1}\cdot\mathbf{f}(\mathbf{y}_n) \tag{16.6.17}$$

如果h不是很大,也许只用一次牛顿法迭代就足以精确地用(16.6.17)式求解方程(16.6.15)。也就是说,在每一步,必须把矩阵

$$\mathbf{1}-h\frac{\partial\mathbf{f}}{\partial\mathbf{y}} \tag{16.6.18}$$

求逆以求出$\mathbf{y}_{n+1}$。用线性化解决隐式解法的问题称为"半隐式解法"。这样,(16.6.17)式便是**半隐欧拉方法**。这种方法并不保证是稳定的,但通常是稳定的,因为其局部表现很类似于上面所述的常数矩阵$\mathbf{C}$的情况。

目前为止,讨论了有一阶精确度的隐式算法,这些算法可用性是很强的。大多数问题可从高阶算法中得到满意结果。下面是刚性方程组三种重要的高阶算法:

- 通用的Runge-Kutta方法,其中最有用的是Rosenbrock方法。首次用该思想进得实践的是Kaps和Rentrop,所以这些方法又被称为Kaps-Rentrop方法。
- 通用的Bulirsch-Stoer方法,尤其是Bader和Deuflhard提出的半隐式外推方法。
- 预测校正法,其主要部分是Gear后向差分法的继续。

将给出前两种方法的实现。注意,如果方程的右边显式地依赖于x,$\mathbf{f}(\mathbf{y},x)$,可以通过把x添加到因变量的列表中解决。这样,方程组变成

$$\begin{pmatrix}\mathbf{y}\\x\end{pmatrix}'=\begin{pmatrix}\mathbf{f}\\1\end{pmatrix} \tag{16.6.19}$$

在本节要给出的两个程序中,我们已把这种替换显式地完成了。这样,程序就可以解决$\mathbf{f}(\mathbf{y},x)$的右端。

现在,我们提醒很重要的一点:**在用自动调整步长来积分刚性问题时,适当地标度变元是很关键的**。和

非刚性方程求解程序一样，需提供一个向量 $\mathbf{y}_{scal}$，用其来测量误差。例如，为得到常数部分误差，只需简单地设置 $\mathbf{y}_{scal} = |\mathbf{y}|$。也可以通过把 $\mathbf{y}_{scal}$ 设置成其最大值，来得到常量相对于某一最大值的绝对误差。在刚性方程中，在其解中常常有递减极其迅速的部分，当其变得很小时，用户可能不十分感兴趣。这时，可以控制高于某一阈值 C 的相对误差，或是低于该阈值的绝对误差，通过设置

$$\mathbf{y}_{scal} = \max(\mathbf{C}, |\mathbf{y}|) \tag{16.6.20}$$

如果用某一适当的无量纲的单位，则 C 的每一分量是有序的单位值。如果不能确定应该给 C 采用什么样的中值，则只需试着把每个分量设成和单位值相同即可。**对于刚性问题，建议选择(16.6.20)式。**

最后警告一点：解决刚性问题有时可能导致精度的极大丢失。双精度常常是必需的，而不是可以选择的。

16.6.1　深入讨论：Rosenbrock 方法

这些方法在理解和实现起来相对比较简单。对于中等精度要求误差准则为($\epsilon \lesssim 10^{-4} \sim 10^{-5}$)以及中等复杂的问题($N \lesssim 10$)，它们和更加复杂的算法是可相比拟的。对于更加严格的参数，Rosenbrock 方法仍然有效，这些方法在有效性上只是比半隐含外推方法(参见后面的讨论)要差一些。

Rosenbrock 方法寻找了一个形式如下的解：

$$\mathbf{y}(x_0 + h) = \mathbf{y}_0 + \sum_{i=1}^{s} c_i \mathbf{k}_i \tag{16.6.21}$$

其中校正系数 $\mathbf{k}_i$ 通过综合(16.6.17)式中结构的 s 个线性方程进行求解而获得：

$$(\mathbf{1} - \gamma h \mathbf{f}') \cdot \mathbf{k}_i = h\mathbf{f}\left(\mathbf{y}_0 + \sum_{j=1}^{i-1} \alpha_{ij} \mathbf{k}_j\right) + h\mathbf{f}' \cdot \sum_{j=1}^{i-1} \gamma_{ij} \mathbf{k}_j, \qquad i = 1, \ldots, s \tag{16.6.22}$$

其中，用 $\mathbf{f}'$ 表示雅可比矩阵，系数 γ，c_i，α_{ij} 和 γ_{ij} 是和问题无关的固定参数。如果 $\gamma = \gamma_{ij} = 0$，这是简单的 Runge-Kutta方法，(16.6.22)式可被成功地解出 $\mathbf{k}_1, \mathbf{k}_2, \ldots$

成功求解刚性方程组的关键是步长的自动调整算法。Kaps 和 Rentrop[2] 发现了一种嵌入式或 Runge-Kutta-Fehlberg方法。如 16.2 节中所述：计算(16.6.21)式的两个估计值、"真实值"$\mathbf{y}$ 以及有不同系数 $\hat{c}_i$ ($i = 1, \ldots, \hat{s}$)的低阶估计值，这里 $\hat{s} < s$，但 $\mathbf{k}_i$ 是相同的。$\mathbf{y}$ 和 $\hat{\mathbf{y}}$ 之差可以导出局部截断误差的估计值，这可以用于步长控制。Kaps 和 Rentrop 指出，取嵌入的最小 s 可能是 $s = 4$，$\hat{s} = 3$，从而成为一个四阶方法。

为使(16.6.22)式右端的矩阵向量的乘积最小，用下列量重写方程：

$$\mathbf{g}_i = \sum_{j=1}^{i-1} \gamma_{ij} \mathbf{k}_j + \gamma \mathbf{k}_i \tag{16.6.23}$$

则方程变成下面形式：

$$\begin{aligned}
(1/\gamma h - \mathbf{f}') \cdot \mathbf{g}_1 &= \mathbf{f}(\mathbf{y}_0) \\
(1/\gamma h - \mathbf{f}') \cdot \mathbf{g}_2 &= \mathbf{f}(\mathbf{y}_0 + a_{21}\mathbf{g}_1) + c_{21}\mathbf{g}_1/h \\
(1/\gamma h - \mathbf{f}') \cdot \mathbf{g}_3 &= \mathbf{f}(\mathbf{y}_0 + a_{31}\mathbf{g}_1 + a_{32}\mathbf{g}_2) + (c_{31}\mathbf{g}_1 + c_{32}\mathbf{g}_2)/h \\
(1/\gamma h - \mathbf{f}') \cdot \mathbf{g}_4 &= \mathbf{f}(\mathbf{y}_0 + a_{41}\mathbf{g}_1 + a_{42}\mathbf{g}_2 + a_{43}\mathbf{g}_3) + (c_{41}\mathbf{g}_1 + c_{42}\mathbf{g}_2 + c_{43}\mathbf{g}_3)/h
\end{aligned} \tag{16.6.24}$$

在根据 Kaps-Rentrop 算法实施的程序 stiff 中，已经在(16.6.24)式中显式地实现了(16.6.19)式中的代换，所以用户不用为此担心了，只需简单地提供一个程序(在 stiff 中称为 derivs)，该程序返回 $\mathbf{f}$(称为 dydx)作为 x 和 $\mathbf{y}$ 的函数。同时也提供了一个程序 jacobn_s，该程序返回 $\mathbf{f}'$(dfdy)以及 $\partial \mathbf{f}/\partial x$(dfdx)作为 x 和 $\mathbf{y}$ 的函数。如果 x 在右端没有显式出现，则 dfdx 为零。通常雅可比矩阵可通过对右端 f 解析微分而得到。如果不能，用户的程序必须用适当的增量 $\Delta\mathbf{y}$ 通过数值差来计算。

Kaps 和 Rentrop 给出了两组不同的参数集，在稳定性方面有稍许的不同。另外还提出了其他几组参数。默认的选择是 Shampine[3]提供的参数，但也给出其中一组 Kaps-Rentrop 参数作为可选项。有些提出的参数集需要有在积分域外的函数估值，我们希望避开那些复杂度。

调用 stiff 的参数序列和早先给出的非刚性程序中的一样,因此它和一般常微分方程中所给程序 odeint 是兼容的。遗憾的是,这种兼容性需要有一些稍微不同:用户提供的程序 derivs 是一个哑元(无实际名字),而另一用户提供的程序不是一个参数,必须命名为 jacobn_s。

stiff 程序首先存储初始值,以备在误差超过时该步能重新计算。线性方程组(16.6.24)首先用程序ludcmp对矩阵 $1/\gamma h - \mathbf{f}'$ 进行 LU 分解。然后用 lubksb 对四个不同的右端项进行回代求出 $\mathbf{g}_i$。注意,积分的每一步需要调用 jacobn_s 一次,调用 derivs 三次(是在 stiff 之前调用一次获取 dydx,并在 stiff 内部调用两次)。为什么只需三次而非四次调用,原因是方程组(16.6.24)中最后的两个调用参数已经被选择,所以只需用相同的参数即可。雅可比矩阵的计算量大致等同于对右端 $\mathbf{f}$ 的 N 个估值的计算量,我们看出 Kaps-Rentrop 方法每步要进行 $N+3$ 次函数的估算。注意,如果 N 较大,而雅可比矩阵是稀疏矩阵,就应该用一合适的稀疏矩阵算法来代替 LU 分解。

步长控制依赖于

$$\begin{aligned}\mathbf{y}_{\text{exact}} &= \mathbf{y} + O(h^5)\\ \mathbf{y}_{\text{exact}} &= \hat{\mathbf{y}} + O(h^4)\end{aligned} \tag{16.6.25}$$

因此

$$|\mathbf{y} - \hat{\mathbf{y}}| = O(h^4) \tag{16.6.26}$$

参考(16.2.4)式~(16.2.10)式的起始步长,新的步长应该选为如(16.2.10)式,但是把指数 1/4 和 1/5 相应地用 1/3 和 1/4 代替。同样,经验表明在一步计算中,应避免步长发生大的变化,否则可能要在下一步中取消这种大的变化。采用 0.5 和 1.5 作为一步中允许 h 的最大减量和增量。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::stiff(Vec_IO_DP &y, Vec_IO_DP &dydx, DP &x, const DP htry,
    const DP eps, Vec_I_DP &yscal, DP &hdid, DP &hnext,
    void deriva(const DP, Vec_I_DP &, Vec_O_DP &))
四阶 Rosenbrock 求解刚度常微分方程组,用监视局部截断误差来调整步长。输入是自变量 x 的初始值处的因变量 y[0..
n-1]及其导数 dydx[0..n-1]。输入还有尝试的步长 htry,所需精度 eps,以及量测误差的向量 yscal[0..n-1]。在输出
中,y 和 x 由其新值替换,hdid 是实际采用的步长值,hnext 是步长的下一步估算值。derivs 是用户提供的程序,用以计算
等式右端相应于 x 的导数,而 jacobn_s(固定的名字)是用户提供的程序,用以计算相应 y 元素的右端项导数的 Jacobi
矩阵
{
    const DP SAFETY=0.9,GROW=1.5,PGROW=-0.25,SHRNK=0.5;
    const DP PSHRNK=(-1.0/3.0),ERRCON=0.1296;
    GROW 和 SHRNK 是在一步中步长能变化的最大和最小因子; ERRCON 等于(GROW/SAFETY)
    的(1/PGROW)次幂,并处理 errmax=0 的情况。
    const int MAXTRY=40;                步长变化判断的最大次数。
    const DP GAM=1.0/2.0,A21=2.0,A31=48.0/25.0,A32=6.0/25.0,C21=-8.0,
        C31=372.0/25.0,C32=12.0/5.0,C41=(-112.0/125.0),
        C42=(-54.0/125.0),C43=(-2.0/5.0),B1=19.0/9.0,B2=1.0/2.0,
        B3=25.0/108.0,B4=125.0/108.0,E1=17.0/54.0,E2=7.0/36.0,E3=0.0,
        E4=125.0/108.0,C1X=1.0/2.0,C2X=(-3.0/2.0),C3X=(121.0/50.0),
        C4X=(29.0/250.0),A2X=1.0,A3X=3.0/5.0;
    int i,j,jtry;
    DP d,errmax,h,xsav;

    int n=y.size();
    Mat_DP a(n,n),dfdy(n,n);
    Vec_INT indx(n);
    Vec_DP dfdx(n),dysav(n),err(n),ysav(n),g1(n),g2(n),g3(n),g4(n);
    xsav=x;                             保存初值。
    for (i=0;i<n;i++) {
        ysav[i]=y[i];
        dysav[i]=dydx[i];
    }
    jacobn_s(xsav,ysav,dfdx,dfdy);
    用户提供的程序返回阶 n×n 矩阵 dfdy 和向量 dfdx。
    h=htry;                             给初始判决值设置步长。
    for (jtry=0;jtry<MAXTRY;jtry++) {
```

```
        for (i=0;i<n;i++) {          建立矩阵 1 - γhf'。
            for (j=0;j<n;j++) a[i][j] = -dfdy[i][j];
            a[i][i] += 1.0/(GAM*h);
        }
        ludcmp(a,indx,d);            矩阵 LU 分解。
        for (i=0;i<n;i++)            为g1设置右边。
            g1[i]=dysav[i]+h*C1X*dfdx[i];
        lubksb(a,indx,g1);           解 g1。
        for (i=0;i<n;i++)            计算中间值 y 和 x。
             y[i]=ysav[i]+A21*g1[i];
         x=xsav+A2X*h;
         derivs(x,y,dydx);           计算中间值处的 dydx。
         for (i=0;i<n;i++)           为g2设置右边。
             g2[i]=dydx[i]+h*C2X*dfdx[i]+C21*g1[i]/h;
         lubksb(a,indx,g2);          解 g2。
         for (i=0;i<n;i++)           计算中间值 y 和 x。
             y[i]=ysav[i]+A31*g1[i]+A32*g2[i];
         x=xsav+A3X*h;
         derivs(x,y,dydx);           计算中间值处的 dydx。
         for (i=0;i<n;i++)           为g3设置右边。
             g3[i]=dydx[i]+h*C3X*dfdx[i]+(C31*g1[i]+C32*g2[i])/h;
         lubksb(a,indx,g3);          解 g3。
         for (i=0;i<n;i++)           为g4设置右边。
             g4[i]=dydx[i]+h*C4X*dfdx[i]+(C41*g1[i]+C42*g2[i]+C43*g3[i])/h;
         lubksb(a,indx,g4);          解 g4。
         for (i=0;i<n;i++) {         得到 y 的四阶估值及误差估值。
             y[i]=ysav[i]+B1*g1[i]+B2*g2[i]+B3*g3[i]+B4*g4[i];
             err[i]=E1*g1[i]+E2*g2[i]+E3*g3[i]+E4*g4[i];
         }
         x=xsav+h;
         if (x == xsav) nrerror("stepsize not significant in stiff");
         errmax=0.0;                 计算精度。
         for (i=0;i<n;i++) errmax=MAX(errmax,fabs(err[i]/yscal[i]));
         errmax /= eps;              相对于所需误差容限缩放。
         if (errmax <= 1.0) {        该步成功，计算下一步长并返回。
             hdid=h;
             hnext=(errmax > ERRCON ? SAFETY*h*pow(errmax,PGROW) : GROW*h);
             return;
         } else {                    截断误差太大，减小步长。
             hnext=SAFETY*h*pow(errmax,PSHRNK);
             h=(h >= 0.0 ? MAX(hnext,SHRNK*h) : MIN(hnext,SHRNK*h));
         }
     }                               返回重试。
     nrerror("exceeded MAXTRY in stiff");
 }
```

这里是 Kaps-Rentrop 参数，替换适当的 const DP 语句，就可简单地将其替换成 Shampine 参数：

```
const DP GAM=0.231,A21=2.0,A31=4.52470820736,A32=4.16352878860,
    C21=-5.07167533877,C31=6.02015272865,C32=0.159750684673,
    C41=-1.856343618677,C42=-8.50538085819,C43=-2.08407513602,
    B1=3.95750374663,B2=4.62489238836,B3=0.617477263873,
    B4=1.282612945268,E1=-2.30215540292,E2=-3.07363448539,
    E3=0.873280801802,E4=1.282612945268,C1X=GAM,
    C2X=-0.396296677520e-01,C3X=0.550778939579,
    C4X=-0.553509845700e-01,A2X=0.462,A3X=0.880208333333
```

作为一个例子，说明 stiff 是怎样使用的，可求解下面的问题：

$$
\begin{aligned}
y_0' &= -.013y_0 - 1000y_0y_2 \\
y_1' &= -2500y_1y_2 \\
y_2' &= -.013y_0 - 1000y_0y_2 - 2500y_1y_2
\end{aligned}
\tag{16.6.27}
$$

初始值为

$$
y_0(0) = 1, \qquad y_1(0) = 1, \qquad y_2(0) = 0 \tag{16.6.28}
$$

（这是本节参考文献[4]中的测试问题 D4。）用 odeint 程序，以初始步长 $h = 2.9 \times 10^{-4}$ 计算积分方程组到 $x = 50$。在(16.6.20)式中 **C** 的元素都设成单位值。该问题的程序 derivs 和 jacobn_s 如下所示。尽管最大的到

最小常数系数的比值差到10^6左右,但stiff还是成功地仅用29步就求解出了这个问题,而误差为$\epsilon = 10^{-4}$。相比较而言,Runge-Kutta程序rkqs需要51 012步。

```
#include "nr.h"

void NR::jacobn_s(const DP x, Vec_I_DP &y, Vec_O_DP &dfdx, Mat_O_DP &dfdy)
{
    int i;

    int n=y.size();
    for (i=0;i<n;i++) dfdx[i]=0.0;
    dfdy[0][0] = -0.013-1000.0*y[2];
    dfdy[0][1] = 0.0;
    dfdy[0][2] = -1000.0*y[0];
    dfdy[1][0] = 0.0;
    dfdy[1][1] = -2500.0*y[2];
    dfdy[1][2] = -2500.0*y[1];
    dfdy[2][0] = -0.013-1000.0*y[2];
    dfdy[2][1] = -2500.0*y[2];
    dfdy[2][2] = -1000.0*y[0]-2500.0*y[1];
}

void NR::derivs_s(const DP x, Vec_I_DP &y, Vec_O_DP &dydx)
{
    dydx[0] = -0.013*y[0]-1000.0*y[0]*y[2];
    dydx[1] = -2500.0*y[1]*y[2];
    dydx[2] = -0.013*y[0]-1000.0*y[0]*y[2]-2500.0*y[1]*y[2];
}
```

16.6.2 深入讨论:半隐式外推算法

Bulirsch-Stoer方法用修正中点的方法分离微分方程,但对刚性方程无效。Bader和Deuflhard[5]发现了半隐外推方法,效果很好,而且能和最初的Bulirsch-Stoer方法一样有效。

起始点是一个隐式中点规则形式:

$$\mathbf{y}_{n+1} - \mathbf{y}_{n-1} = 2h\mathbf{f}\left(\frac{\mathbf{y}_{n+1} + \mathbf{y}_{n-1}}{2}\right) \tag{16.6.29}$$

线性化该等式右端$\mathbf{f}(\mathbf{y}_n)$,把(16.6.29)式转换成半隐式,结果便是**半隐式中点规则**:

$$\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right] \cdot \mathbf{y}_{n+1} = \left[\mathbf{1} + h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right] \cdot \mathbf{y}_{n-1} + 2h\left[\mathbf{f}(\mathbf{y}_n) - \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \cdot \mathbf{y}_n\right] \tag{16.6.30}$$

使用了特殊的第一步,半隐式欧拉步(16.6.17),以及在最后一步特殊的"平滑",其中最后的$\mathbf{y}_n$用

$$\bar{\mathbf{y}}_n \equiv \tfrac{1}{2}(\mathbf{y}_{n+1} + \mathbf{y}_{n-1}) \tag{16.6.31}$$

代换。Bader和Deuflhard说明,这种方法的误差级数也仅与h的偶次方相关。

在实际应用中,最好用$\Delta_k \equiv \mathbf{y}_{k+1} - \mathbf{y}_k$改写一下方程。利用$h = H/m$,首先计算

$$\begin{aligned} \Delta_0 &= \left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot h\mathbf{f}(\mathbf{y}_0) \\ \mathbf{y}_1 &= \mathbf{y}_0 + \Delta_0 \end{aligned} \tag{16.6.32}$$

然后对于$k = 1, \ldots, m-1$,设置

$$\begin{aligned} \Delta_k &= \Delta_{k-1} + 2\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot [h\mathbf{f}(\mathbf{y}_k) - \Delta_{k-1}] \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + \Delta_k \end{aligned} \tag{16.6.33}$$

最后计算

$$\begin{aligned} \Delta_m &= \left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot [h\mathbf{f}(\mathbf{y}_m) - \Delta_{m-1}] \\ \bar{\mathbf{y}}_m &= \mathbf{y}_m + \Delta_m \end{aligned} \tag{16.6.34}$$

在上面的公式中,很容易和(16.6.19)式替换公式结合起来。在雅可比中,来自$\partial \mathbf{f}/\partial x$的额外项都抵消了半隐

性中点规则(16.6.30)　在特殊的第一步(16.6.17)及相应的(16.6.32)式中，项 $h\mathbf{f}$ 变成 $h\mathbf{f}+h^2\,\partial\mathbf{f}/\partial x$，其他方程都未变。

该算法在程序simpr中实现。

```
#include "nr.h"

void NR::simpr(Vec_I_DP &y, Vec_I_DP &dydx, Vec_I_DP &dfdx, Mat_I_DP &dfdy,
    const DP xs, const DP htot, const int nstep, Vec_O_DP &yout,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &))
```

执行一步半隐式中点规则算法。输入因变量y[0..n-1]，其导数dydx[0..n-1]，和对应于 x 的右端项导数dfdx[0..n-1]及在xs处的雅可比矩阵dfdy[0..n-1][0..n-1]。输入还有htot，总共要取的步数和nstep，被用的子步数。输出返回在yout[0..n-1]中，derivs是用户提供的程序，用来计算dydx。

```
{
    int i,j,nn;
    DP d,h,x;

    int n=y.size();
    Mat_DP a(n,n);
    Vec_INT indx(n);
    Vec_DP del(n),ytemp(n);
    h=htot/nstep;                          这一步的步长。
    for (i=0;i<n;i++) {                    设置矩阵 1 - hf'。
        for (j=0;j<n;j++) a[i][j] = -h*dfdy[i][j];
        ++a[i][i];
    }
    ludcmp(a,indx,d);                      矩阵 LU 分解。
    for (i=0;i<n;i++)                      为第一步建立右端项，用 yout 作暂时存储器。
        yout[i]=h*(dydx[i]+h*dfdx[i]);
    lubksb(a,indx,yout);
    for (i=0;i<n;i++)                      第一步。
        ytemp[i]=y[i]+(del[i]=yout[i]);
    x=xs+h;
    derivs(x,ytemp,yout);                  用 yout 暂存导数值。
    for (nn=2;nn<=nstep;nn++) {            一般步。
        for (i=0;i<n;i++)                  为一般步设右端项。
            yout[i]=h*yout[i]-del[i];
        lubksb(a,indx,yout);
        for (i=0;i<n;i++) ytemp[i] += (del[i] += 2.0*yout[i]);
        x += h;
        derivs(x,ytemp,yout);
    }
    for (i=0;i<n;i++)                      为最后一步设右端项。
        yout[i]=h*yout[i]-del[i];
    lubksb(a,indx,yout);
    for (i=0;i<n;i++)                      最后一步。
        yout[i] += ytemp[i];
}
```

程序simpr可以用于程序stifbs中，基本上和bsstep一样，惟一不同的是：

- 步长的序列是

$$n=2,6,10,14,22,34,50,\ldots, \tag{16.6.35}$$

其中每个值和前面一个值相差4的倍数，使得相邻项的比值 $\leqslant\frac{5}{7}$，参数KMAXX取为7。

- 每一步的工作量，不但包括函数求值的工作量，还包括雅可比计算的计算量。计算雅可比的工作量和 N 个函数的估计值相当，这里 N 是方程的数目。
- 用户提供的程序derivs是哑元，所以能取任何名字。但是，如果要和rkqs，bsstep及stiff保持最简单的兼容，jacobn_s就不是一个参数，而必须定为这个名字。每一步被调用后返回作为 x 和 $\mathbf{y}$ 的函数 $\mathbf{f}'$ (dfdy)以及 $\partial\mathbf{f}/\partial x$ (dfdx)

下面程序的注释中说明了和bsstep的不同之处：

```
#include <cmath>
#include "nr.h"
using namespace std;

Vec_DP *x_p;
```

```
Mat_DP *d_p;

void NR::stifbs(Vec_IO_DP &y, Vec_IO_DP &dydx, DP &xx, const DP htry,
    const DP eps, Vec_I_DP &yscal, DP &hdid, DP &hnext,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &))
```

半隐式外推法求解刚性微分方程，用监视局部截断误差以调整步长。输入是位于自变量 x 起始点处的因变量 y[0..nv-1]及其导数值 dydx[0..nv-1]。输入还有试用的步长 htry、所需精度 eps，以及向量 yscal[0..nv-1]通过该项来标度误差。在输出中，y 和 x 被其新值所替代，hdid 是实际采用的步长，hnext 是估算的下一个步长，derivs 是用户提供的程序，计算右端项相对于 x 的导数，而 jacobn_s(名字固定)是用户提供的程序，计算右端项相对于 y 元素的导数的雅可比矩阵。注意，在步骤成功时，把从前面步返回的 hnext 赋给 htry，以备此程序被 odeint 调用。

```
{
    const int KMAXX=7,IMAXX=KMAXX+1;
    const DP SAFE1=0.25,SAFE2=0.7,REDMAX=1.0e-5,REDMIN=0.7;
    const DP TINY=1.0e-30,SCALMX=0.1;
    bool exitflag=false;
    int i,iq,k,kk,km,reduct;
    static int first=1,kmax,kopt,nvold = -1;
    DP eps1,errmax,fact,h,red,scale,work,wrkmin,xest;
    static DP epsold = -1.0,xnew;
    static Vec_DP a(IMAXX);
    static Mat_DP alf(KMAXX,KMAXX);
    static int nseq_d[IMAXX]={2,6,10,14,22,34,50,70};        序列和 bsstep 不同。
    Vec_INT nseq(nseq_d,IMAXX);

    int nv=y.size();
    d_p=new Mat_DP(nv,KMAXX);
    x_p=new Vec_DP(KMAXX);
    Vec_DP dfdx(nv),err(KMAXX),yerr(nv),ysav(nv),yseq(nv);
    Mat_DP dfdy(nv,nv);
    if (eps != epsold || nv != nvold) {        如果 nv 改变则重新初始化。
        hnext = xnew = -1.0e29;
        eps1=SAFE1*eps;
        a[0]=nseq[0]+1;
        for (k=0;k<KMAXX;k++) a[k+1]=a[k]+nseq[k+1];
     for (iq=1;iq<KMAXX;iq++) {
         for (k=0;k<iq;k++)
             alf[k][iq]=pow(eps1,(a[k+1]-a[iq+1])/
                 ((a[iq+1]-a[0]+1.0)*(2*k+3)));
     }
     epsold=eps;
     nvold=nv;                                  保存 nv。
     a[0] += nv;                                把雅可比计算量添加到工作量系数中。
     for (k=0;k<KMAXX;k++) a[k+1]=a[k]+nseq[k+1];
     for (kopt=1;kopt<KMAXX-1;kopt++)
         if (a[kopt+1] > a[kopt]*alf[kopt-1][kopt]) break;
     kmax=kopt;
 }
 h=htry;
 for (i=0;i<nv;i++) ysav[i]=y[i];
 jacobn_s(xx,y,dfdx,dfdy);                      计算雅可比。
 if (xx != xnew || h != hnext) {
     first=1;
     kopt=kmax;
 }
 reduct=0;
 for (;;) {
     for (k=0;k<=kmax;k++) {
         xnew=xx+h;
         if (xnew == xx) nrerror("step size underflow in stifbs");
         simpr(ysav,dydx,dfdx,dfdy,xx,h,nseq[k],yseq,derivs);
         半隐式中点规则。
         pzextr(k,xest,yseq,y,yerr);            程序其余部分和 bsstep 相同。
         if (k != 0) {
             errmax=TINY;
             for (i=0;i<nv;i++) errmax=MAX(errmax,fabs(yerr[i]/yscal[i]));
             errmax /= eps;
             km=k-1;
```

```
                err[km]=pow(errmax/SAFE1,1.0/(2*km+3));
            }
            if (k != 0 && (k >= kopt-1 || first)) {
                if (errmax < 1.0) {
                    exitflag=true;
                    break;
                }
                if (k == kmax || k == kopt+1) {
                    red=SAFE2/err[km];
                    break;
                }
                else if (k == kopt && alf[kopt-1][kopt] < err[km]) {
                    red=1.0/err[km];
                    break;
                }
                else if (kopt == kmax && alf[km][kmax-1] < err[km]) {
                    red=alf[km][kmax-1]*SAFE2/err[km];
                    break;
                }
                else if (alf[km][kopt] < err[km]) {
                    red=alf[km][kopt-1]/err[km];
                    break;
                }
            }
        }
        if (exitflag) break;
        red=MIN(red,REDMIN);
        red=MAX(red,REDMAX);
        h *= red;
        reduct=1;
        }
        xx=xnew;
        hdid=h;
        first=0;
        wrkmin=1.0e35;
        for (kk=0;kk<=km;kk++) {
            fact=MAX(err[kk],SCALMX);
            work=fact*a[kk+1];
            if (work < wrkmin) {
                scale=fact;
                wrkmin=work;
                kopt=kk+1;
            }
        }
        hnext=h/scale;
        if (kopt >= k && kopt != kmax && !reduct) {
            fact=MAX(scale/alf[kopt-1][kopt],SCALMX);
            if (a[kopt+1]*fact <= wrkmin) {
                hnext=h/fact;
                kopt++;
            }
        }
        delete d_p;
        delete x_p;
    }
```

程序stifbs是一个解决刚性方程的极好程序，可以和最好的 Gear 类型的程序相比。stiff对于中等复杂度 N 以及$\epsilon \leq 10^{-4}$时还是很好的。当$\epsilon \sim 10^{-8}$时，stifbs大概要快一个数量级。另外还有一些措施可用来提高 stifbs 的性能。例如，很偶然地，simpr中的ludcmp会碰到奇异矩阵，这时可以让步长减小，可以减小当前nseq[k]的一个因子。另外在某些问题上，还有步长稳定性的限制，至于怎样自动解决这些问题参见本节参考文献[6]。

参考文献和深入阅读

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Kaps, P., and Rentrop, P. 1979, *Numerische Mathematik*, vol. 33, pp. 55–68. [2]

Shampine, L.F. 1982, *ACM Transactions on Mathematical Software*, vol. 8, pp. 93–113. [3]

Enright, W.H., and Pryce, J.D. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 1–27. [4]

Bader, G., and Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 373–398. [5]

Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 399–422.

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535.

Deuflhard, P. 1987, "Uniqueness Theorems for Stiff ODE Initial Value Problems," *Preprint SC-87-3* (Berlin: Konrad Zuse Zentrum für Informationstechnik). [6]

Enright, W.H., Hull, T.E., and Lindberg, B. 1975, *BIT*, vol. 15, pp. 10–48.

Wanner, G. 1988, in *Numerical Analysis 1987*, Pitman Research Notes in Mathematics, vol. 170, D.F. Griffiths and G.A. Watson, eds. (Harlow, Essex, U.K.: Longman Scientific and Technical).

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag)

16.7 多步法、多值法和预测-校正法

多步法和多值法描述了两类基于同样积分技巧的解常微分方程的方法。预测-校正法是这些方法的一种特殊情况——实际上也是应用最广泛的情况。因此,预测-校正算法有时泛指所有这些方法。

预测-校正算法可能已经过时,而且对大多数常微分方程来说已不是最佳解法了。对需要高精度的情况,或是对方程右边算式的估值计算量很大的情况,Bulirsch-Stoer 方法占据主要地位。对于比较简单或低精度的问题,自适应步长的 Runge-Kutta 方法占据主要地位。预测-校正法是介于其间的一种算法,也许只有一种情况最适合:需要比较高的精度,函数很平滑而且等式右边很复杂。这种情况在下面讨论。

不管怎么说,这些方法都经历了很长的历史过程,现在很多教科书上都是这些内容,而且有很多标准的常微分方程的程序都是基于预测-校正法解决的。很多研究者用预测-校正法很有经验,这些人没有必要马上改变原来的习惯。读者最好也能熟悉一下这些方法涉及的原理,以及这些方法局限性的各个细节。不然的话,当第一次突然碰到需用预测-校正法编程时,会感到束手无策。

首先考虑多步法逼近。考虑求解一个常微分方程和求函数积分有什么不同:对于一个函数,被积函数和自变量 x 的关系是已知的,而且是可估算的。对于一个常微分方程,"被积函数"是等式的右端部分,既和自变量 x,也和因变量 y 有关系,因此要计算 $y'=f(x,y)$ 的从 x_n 到 x 的解,有

$$y(x)=y_n+\int_{x_n}^{x} f(x',y)\,dx' \tag{16.7.1}$$

在像 Runge-Kutta 或 Bulirsch-Stoer 方法的单步计算中,在 x_{n+1} 处的 y_{n+1} 值只依赖于 y_n 的值。在多步法中,用通过前面几个点 $x_n, x_{n-1}, \ldots$ 或也可能通过 x_{n+1} 的一个多项式来近似 $f(x,y)$。计算 $x=x_{n+1}$ 处(16.7.1)式的积分,则成为下列形式:

$$y_{n+1}=y_n+h(\beta_0 y'_{n+1}+\beta_1 y'_n+\beta_2 y'_{n-1}+\beta_3 y'_{n-2}+\cdots) \tag{16.7.2}$$

其中 y'_n 代表 $f(x_n, y_n)$,依次类推。如果 $\beta_0=0$,该方法是显式的,否则是隐式的。该方法的阶次决定于用了多少已走的步数来得到 y 的每个新值。

考虑怎样才能解一个形为(16.7.2)式的隐式公式求出 y_{n+1}。有两种方法:**函数迭代法**和**牛顿法**。在函数迭代法中,取 y_{n+1} 的某些初始猜测值,将其插入到(16.7.2)式的右端,以得到 y_{n+1} 的变动值,再把该变动值代回右边,继续迭代。很容易得到 y_{n+1} 的初始猜测值,只需用类似(16.7.2)式的显式公式即可,这叫做**预测步**。在预测步中,主要是外推多项式,以拟合从以前点到新点 x_{n+1} 的导数值,并从 x_n 到 x_{n+1} 进行类似Simpson 方法求积分(16.7.1)。随后的类似 Simpson 方法的积分,用 y_{n+1} 的预测步值来内插导数,被称为**校正步**。预测步和校正步的函数值之差提供了局部截断误差的信息,可用于误差控制及调整步长。

如果一个校正步成功了,多几个校正步会不会更好些?为何不用校正步作为一个改进的预测步并在每

步迭代到收敛呢？答案是：即使有一个十全十美的预测，该步的精度仍只可能达到校正步的有限阶次的精度。其不能矫正的误差项和迭代能消除的误差是同阶的，所以最好只把误差项前的系数缩小为原来的几分之几。因此毫无疑问，进一步的提高是不值得的，最好还是把精力放在缩小步长上面。

迄今为止，读者也许想到值得而且必须在每一步之前预测几个间隔点，然后在类似 Simpson 方法的校正中，用不同的权重来使用所有这些间隔点。这并不是一个好主意。外推是这种方法中最不稳定的部分，应该尽量减少它的影响。因此，预测-校正法中的积分步骤是重叠进行的，每一步都涉及几个步长间隔 h，每一步只比前一步扩展了一个这样的步长间隔，而且只有那个被扩展的间隔是通过每次预测步进行外推的。

最常用的预测-校正法是 Adams-Bashforth-Moulton 格式。这种方法有很好的稳定性。Adams-Bashforth 部分是预测步。例如，三阶的情况是

$$\text{预测步：}\qquad y_{n+1} = y_n + \frac{h}{12}(23y'_n - 16y'_{n-1} + 5y'_{n-2}) + O(h^4) \tag{16.7.3}$$

其中在当前点 x_n 的信息和前两个点 x_{n-1} 和 x_{n-2}（假设设置等间距）的信息，一起用来预测在下一点 x_{n+1} 上的值 y_{n+1}。Adams-Moulton 部分是校正步。三阶的情况是

$$\text{校正步：}\qquad y_{n+1} = y_n + \frac{h}{12}(5y'_{n+1} + 8y'_n - y'_{n-1}) + O(h^4) \tag{16.7.4}$$

如果等式右端没有加入从预测步获得的 y_{n+1} 试用值的话，校正步将会变成一个蕴含 y_{n+1} 的很难处理的式子。

这种算法实际上要进行三个单独的步骤：预测步，称为 P；从最新的 y 值估算出导数值 y'_{n+1}，称为 E；校正步，称为 C。在这种表示中，用校正步进行 m 次迭代（这种做法我们曾批驳过）记做 $P(EC)^m$。也可以选择用一个 C 步或一个 E 步。最后一步 E 是最关键的，所以通常建议用 PECE。

注意，具有固定迭代数目的 PC 方法是一种显式方法。当事先固定了迭代的数目以后，则最后的值 y_{n+1} 可以写做一些已知变量的复杂函数式。因此固定迭代数目的 PC 方法失去了隐式方法的强稳定性，**只应用于非刚性问题**。

对于刚性问题，如果想要避免用细小步长的话，则**必须**用隐式方法（并非所有的隐式方法对于刚性问题都是很好，幸运的是有一些已知的方法如 Gear 公式是很有效的）。这样，解决隐式方程便似乎有两种选择：函数迭代到收敛，或是牛顿迭代法。但是，对刚性问题来说，无论预测是多么接近，函数迭代只有在我们用极小的步长时才可能收敛。因此，牛顿迭代法通常是多步法解刚性问题的基本方法。对于收敛，牛顿法对步长的大小不敏感，只要预测足够精确就行。

多步法，就我们所述而言，在实现时有两个严重问题：

- 因为公式需要等间距步长的结果，所以步长调整比较困难。
- 开始和停止当前问题时有些麻烦。在开始时，需要初始值加上前面几步来起动。停止时也有问题，因为等长的步距不可能刚好落在停止点上。

过去 PC 方法的实施中，用一些烦琐的方法来处理这些问题。例如，可以用 Runge-Kutta 方法来开始和停止。步长的变化需经相当复杂的记录信息进行某些内插过程。幸运的是这些缺陷可以由多值法来消除。

对于多值算法而言，用于积分器的最基本数据是，在当前点 x_n 解的泰勒级数展开式的头几项。其目标是求得解及其在下一个点 x_{n+1} 的展开式系数。这和多步法明显不同。在多步法中数据是在 $x_n, x_{n-1}, \ldots$ 上求解的值。通过考虑一个四值法来说明这种思路。这个四值算法中，基本的数据是

$$\mathbf{y}_n \equiv \begin{pmatrix} y_n \\ hy'_n \\ (h^2/2)y''_n \\ (h^3/6)y'''_n \end{pmatrix} \tag{16.7.5}$$

同样也按惯例用 $h = x_{n+1} - x_n$ 的幂次来测度导数项，如上式所示。注意，这里用向量记号 $\mathbf{y}$ 代表其解在某一点的头几项导数，而并不是求解一个有很多分量 y 的方程组。

在(16.7.5)式的数据形式中，可以近似某点 x 处的解 y：

$$y(x) = y_n + (x - x_n)y'_n + \frac{(x - x_n)^2}{2}y''_n + \frac{(x - x_n)^3}{6}y'''_n \tag{16.7.6}$$

在(16.7.6)式中,设 $x = x_{n+1}$ 即可得到 y_{n+1} 的估计值。微分(16.7.6)式,令 $x = x_{n+1}$ 即可得到 y'_{n+1} 的估计值,同样可以得到 y''_{n+1} 和 y'''_{n+1}。称其结果的近似值为 $\tilde{\mathbf{y}}_{n+1}$,这里波浪线提示我们所做的只是一个对解及其导数的多项式外推。目前还未用到差分方程,但很容易证明

$$\tilde{\mathbf{y}}_{n+1} = \mathbf{B} \cdot \mathbf{y}_n \tag{16.7.7}$$

其中矩阵 $\mathbf{B}$ 是

$$\mathbf{B} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{16.7.8}$$

现在,我们写出以后要用到的实际估计值 $\mathbf{y}_{n+1}$,它等于 $\tilde{\mathbf{y}}_{n+1}$ 增加一个校正量:

$$\mathbf{y}_{n+1} = \tilde{\mathbf{y}}_{n+1} + \alpha\mathbf{r} \tag{16.7.9}$$

其中 $\mathbf{r}$ 是一个固定的数字向量,同样 $\mathbf{B}$ 也是一固定矩阵。可通过满足下面微分方程来确定 α:

$$y'_{n+1} = f(x_{n+1}, y_{n+1}) \tag{16.7.10}$$

(16.7.9)式的第二个方程是

$$hy'_{n+1} = h\tilde{y}'_{n+1} + \alpha r_1 \tag{16.7.11}$$

这将和(16.7.10)式一致而且提供

$$r_1 = 1, \qquad \alpha = hf(x_{n+1}, y_{n+1}) - h\tilde{y}'_{n+1} \tag{16.7.12}$$

值 r_0, r_2 和 r_3 是任意的,可由给定的四值法选择。不同的选择给出不同的阶次的算法(例如,通过 h 的阶次,最后的表达式(16.7.9)实际上近似其解),以及不同的稳定性。

有一个很有意思的结果,解释中没有明显说明,就是多值法和多步法完全是等价的。换言之,由多值法给定 $\mathbf{B}$ 和 $\mathbf{r}$ 导出的 y_{n+1} 的值和多步法中用(16.7.2)式给出 β 而导出的值是一样的。例如,证明Adams-Bashforth公式(16.7.3)和一个 $r_0 = 0$, $r_2 = 3/4$, $r_3 = 1/6$ 的四值法是等价的。该算法是显式的,因为 $r_0 = 0$。Adams-Moulton 方法即(16.7.4)式相当于 $r_0 = 5/12$, $r_2 = 3/4$ 及 $r_3 = 1/6$ 的隐式四值算法。隐式多值算法和隐式多步算法的处理方法一样:或者通过预测校正逼近,用显示方法作预测步;或者是用牛顿迭代解决刚性问题。

为什么要不厌其烦地介绍一个新方法,而其结果和已知的方法等效。原因是多值法很容易解决上面提到的实现多步法时的两个困难。

首先考虑步长调整的问题。要在某一点 x_n 把步长从 h 变到 h',只需要简单地把(16.7.5)式中 $\mathbf{y}_n$ 的元素乘上 (h'/h) 的适当的幂,便可连续到 $x_n + h'$ 了

多值法也允许在算法阶次上进行相对简单的变化:只需简单地改变 $\mathbf{r}$。这样做通常的办法是首先从误差估计中用现有的阶次决定新的步长,然后检查应该用高一阶还是用低一阶的步长来预测。选择允许下一步取最大的那个阶次。能够选择阶次这就让起点问题也变得容易了,只需简单地从一阶算法开始,再让阶次自动增加到合适的阶次。

对于低精度的要求,类似 rkqs 的 Runge-Kutta 程序几乎总是最有效。对于高精度,bsstep 程序性能既好且有效。对于非常平滑的函数,一个变阶次的 PC 方法可达到高阶。如果等式右端相对较复杂,这样估值的计算量超过了记录信息的费用,则最好的 PC 程序在处理这类问题上比 Bulirsch-Stoer 方法要好。正如所想像的那样,这样变步长,高阶次的算法在程序中并不麻烦。如果用户觉得自己的问题适合于这种方法,建议采用 PC 软件包。详细情况参见 Gear[1]或 Shampine 和 Gordon[2]。

我们预计,像 Bulirsch-Stoer 这样的外推算法是越来越复杂,最终在所有应用中都将不使用 PC 算法。我们希望这点是对的。

参考文献和深入阅读

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Shampine, L.F., and Gordon, M.K. 1975, *Computer Solution of Ordinary Differential Equations. The Initial Value Problem.* (San Francisco: W.H Freeman). [2]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 5.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 8.

Hamming, R.W. 1962, *Numerical Methods for Engineers and Scientists*; reprinted 1986 (New York: Dover), Chapters 14–15.

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), Chapter 7.

第 17 章　两点边值问题

17.0　引言

当常微分方程要求在多于一个的自变量值上满足边界条件时，这种问题叫做**两点边值问题**。正如该术语所表明的，目前最普遍的情况是要求在两个点——通常是积分的起点和终点上满足边界条件。但是"两点边值问题"这种提法也广泛用于更加复杂的情况，如有的条件是在末端指定的，有的条件是在内部指定（通常是奇异点）。

初值问题（第 16 章）和两点边值问题（本章）的主要区别在于，前者可以从其起点处（初值）一个可接受的解开始，只需要通过数值积分到其终点（结果值）即可。而在现在的情况下，起始点的边界条件并不是惟一决定求解开始的条件——而且在满足（不完全）起始边界的解中"随便"选择一个解，它几乎并不能满足其他特定点的边界条件。

一般情况下，需要用迭代法把这些空间上分散的边界值条件融合成一个单一的微分方程的解。正因为这个原因，两点边值问题比初值问题要复杂。我们必须在有意义的区间上对微分方程积分，或是执行类似"松弛"过程（参见下文）几次，有时甚至是很多次，往往只有在特定的线性微分方程情况下，才能事先知道需要多少次迭代计算。

"标准"的两点边值问题有如下的形式：需要求一组 N 个联立的一阶常数微分方程的解，在起始点 x_1 满足 n_1 个边界条件，在最终点 x_2 满足 $n_2 = N - n_1$ 个边界条件（记住，所有高于一阶以上的微分方程都可以写成一组联立的一阶方程，参见 16.0 节）。

微分方程是

$$\frac{dy_i(x)}{dx} = g_i(x, y_0, y_1, \ldots, y_{N-1}) \qquad i = 0, 1, \ldots, N-1 \tag{17.0.1}$$

在点 x_1，解需满足

$$B_{1j}(x_1, y_0, y_1, \ldots, y_{N-1}) = 0 \qquad j = 0, \ldots, n_1 - 1 \tag{17.0.2}$$

而在点 x_2，应满足

$$B_{2k}(x_2, y_0, y_1, \ldots, y_{N-1}) = 0 \qquad k = 0, \ldots, n_2 - 1 \tag{17.0.3}$$

解两点边值问题有两种不同类型的数值解法，在**打靶法**（参见 17.1 节）中，在一个边界上选择了所有因变量的值，这些值必须和该边界的边界条件保持一致。但是在另外一个边界上将因变量设置成依赖于任意参数，其参数值是最初"随机"猜测的。然后，用初始值的方法积分常微分方程，一直到达另外一个边界（和/或是定义了边界条件的内点）。一般来说，我们发现和那里的边界值不吻合，但遇到过多维寻解问题，就像在 9.6 节和 9.7 节中那样：找到起始点处的调整量，从而把在其他边界上的误差量减小到零。如果把微分方程的积分过程比喻为弹道轨迹的话，选择初始条件就相当于瞄准的过程（参见图 17.0.1）。打靶法提供了有规律可循的逼近法，可以进行限定一定范围的"射击"，这可以有规律地增进"瞄准"作用。

作为打靶法的另一变异（参见 17.2 节），可以猜测在定义域两端都不知的自由参量的值。把方程积分到某一合适的中点，设法调整猜测的参数值让解在那一点上"平滑地"相连。在所有的打靶法算法中，试验解能够"精确地"（或考虑到积分算法是相对来说很精确地）满足微分方程，但是只有在迭代结束以后，试验解才能满足所需的边界值条件。

松弛法用了另外一种逼近方法。微分方程由覆盖积分限的一系列有限个差分方程来代替。试验解由各个网格点上因变量的值组成，并不满足所需的有限个差分方程，甚至也不需要满足所需的边界条件。迭代过程，这里称为**松弛**过程，是调整所有在网格上的值，使它们满足各个联立的差分方程，同时也满足边界条件（参见图 17.0.2）。例如，如果问题涉及 3 个联立的方程以及 100 个点的网格，则必须根据解先预测 300 个变量的值，然后进行改进来得到最后的结果。

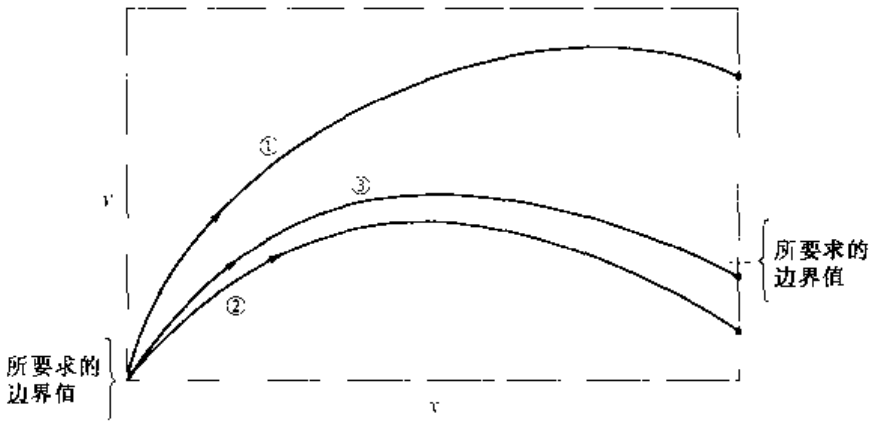


图 17.0.1　打靶法示意图：在满足边界条件的某一端点试验性地从那一点开始积分，在另外一端点的所需边界条件的偏差用以调节起始条件，直到在两个端点的边界条件都被满足

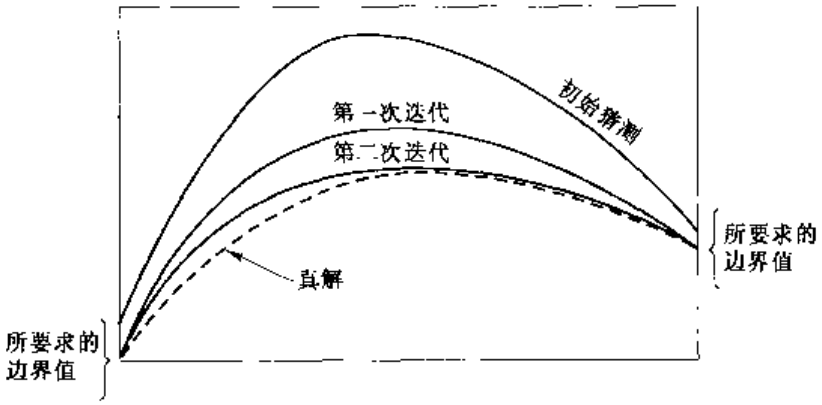


图 17.0.2　松弛法示意图：预测一初始值近似满足微分方程和边界条件，用迭代过程对其进行调整直到最后和真解完全吻合

需要这么多调整，可能会怀疑松弛法是否是有效的方法，但（对某些问题来说）它的确是一种很有效的方法。当边界值条件特别错综复杂的时候，或是涉及到不能用封闭形式进行求解的代数关系时，松弛法比打靶法的效果要好。如果解是平滑的而且不是高度振荡的情况，则松弛法效果最好。如果振荡得很厉害就需要很多的网格点来进行精确的表示。所需要的网格点的数目和位置也许都是事先未知的。这种情况下用打靶法通常比较好，因为打靶法中的变步长积分能够和解的特点相适应。

人们通常喜欢将松弛法用于常微分方程组的解有另外一部分，这部分不出现在满足所有边界条件的最后结果中，但在用打靶法进行初始值积分时它却会造成很多问题。典型的情况是，一个衰减的指数解中会有一个增长的指数部分。

为使松弛法有效地发挥作用，良好的初始预测值是一个关键。通常情况下，常常需要多次解决同一个问题，每次都是某几个参数略微变动一下。在这种情况下，当参数改变时，原先的结果也许就是一个好的初始预测值，并且松弛法能够有效地解决问题。

在挑选这两种方法时，如果还不太有经验，那么可以采用我们的建议：首先选用打靶法，然后再用松弛法。

17.0.1　能用标准边界值问题求解的问题

有两类重要的问题能够用(17.0.1)式～(17.0.3)式所描述的标准边界值问题来求解。第一类是**微分方程组的特征值问题**，这里的微分方程组的右端依赖于参数 λ，

$$\frac{dy_i(x)}{dx} = g_i(x, y_0, \ldots, y_{N-1}, \lambda) \tag{17.0.4}$$

而且必须满足 $N+1$ 个边界条件，而不只是 N 个。这种问题是超定性的。通常不是对任意 λ 值都有解，对某

些特殊的λ值,即特征值,(17.0.4)式确实有解。

引入一个新的因变量和另一个微分方程来推导这个问题:

$$y_N \equiv \lambda \tag{17.0.5}$$

$$\frac{dy_N}{dx} = 0 \tag{17.0.6}$$

在17.4节中给出了这种处理方法的一个例子。

另外一种能化成标准形式的是**自由边界值问题**。这里边界只有一个边界横坐标 x_1 是指定的,而另一个 x_2 是未确定的。这样使(17.0.1)式具有一个满足总共是 $N+1$ 个边界条件的解。这里再次添加一个额外的定值因变量:

$$y_N \equiv x_2 - x_1 \tag{17.0.7}$$

$$\frac{dy_N}{dx} = 0 \tag{17.0.8}$$

还定义了一个新自变量 t,并设置成

$$x - x_1 \equiv t\, y_N, \qquad 0 \leqslant t \leqslant 1 \tag{17.0.9}$$

原来总共 $N+1$ 个关于 dy_i/dt 的微分方程,现在即成标准形式,其中 t 在已知区间0到1之间变化。

参考文献和深入阅读

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems*; reprinted 1991 (New York: Dover).

Kippenhan, R., Weigert, A., and Hofmeister, E. 1968, in *Methods in Computational Physics*, vol. 7 (New York: Academic Press), pp. 129ff.

Eggleton, P.P. 1971, *Monthly Notices of the Royal Astronomical Society*, vol. 151, pp. 351–364.

London, R.A., and Flannery, B.P. 1982, *Astrophysical Journal*, vol. 258, pp. 260–269.

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §§7.3–7.4.

17.1 打靶法

在本节中讨论"纯正"的打靶法,其中积分过程是从 x_1 到 x_2,并且努力使积分结果在积分的终点和边界条件匹配。在下一节里,将介绍射向某一中间拟合点的情况,从区间的两端"射击",争取在某一中间点满足连续条件。然后求出满足方程和边界值条件的解。

实现的打靶法精确地运用了多维全局收敛Newton-Raphson(参见9.7节),它设法零化 n_2 个变元的 n_2 个函数,这些函数通过从 x_1 到 x_2 积分 N 个微分方程得到。下面讨论一下其原理。

在初始点 x_1 有 N 个起始值 y_i 要进行定义,但必须满足 n_1 个条件,因此就需要有 $n_2 = N - n_1$ 个自由定义的起始值。假设这些自由定义的值是 n_2 维向量空间的一个向量 $\mathbf{V}$,当知道了边界条件的函数形式(17.0.2),即可作出一函数,该函数生成 N 个起始值 $\mathbf{y}$ 的完备集,并在 x_1 点满足边界条件。该函数可以从任一向量 $\mathbf{V}$ 来生成。在该向量,对其中的 n_2 个元素没有任何限制。换言之,(17.0.2)式转换成一个表达式

$$y_i(x_1) = y_i(x_1; V_0, \ldots, V_{n_2-1}) \qquad i = 0, \ldots, N-1 \tag{17.1.1}$$

下面实现(17.1.1)式的函数称为load。

注意 $\mathbf{V}$ 的元素也许刚好是 $\mathbf{y}$ 的某些"自由"分量值,$\mathbf{y}$ 的其他分量则由边界条件来确定。另一方面,$\mathbf{V}$ 的元素也许可以以某种简便的方式,将构成满足起始边界条件的解参数化。边界值条件经常用一些代数关系来约束 y_i,而不是单个规定它们的特定值。对某一组 y_i,用一些附加的参数集往往比较容易"解决"边界值关系。无论采用哪种方法都可以,只要能通过向量空间 $\mathbf{V}$(用(17.1.1)式)生成所有允许的起始向量 $\mathbf{y}$。

给定一个特定的 $\mathbf{V}$,特定的 $\mathbf{y}(x_1)$ 即生成了。然后按初始值问题对常微分方程积分到 x_2(例如可用第16章的odeint),就可以将它转变成 $\mathbf{y}(x_2)$。在 x_2 点,定义了一个**偏差向量** $\mathbf{F}$,也是 n_2 维的,其分量用来测度离在 x_2 点处满足 n_2 个边界条件(17.0.3)的差距。最简单的方法是用(17.0.3)式的右端,

$$F_k = B_{2k}(x_2, \mathbf{y}) \qquad k = 0, \ldots, n_2 - 1 \tag{17.1.2}$$

对于向量空间 $\mathbf{V}$,也可以使用任何其他的参数化方法,只要 $\mathbf{F}$ 空间能够扩张成所要求的边界条件可能的偏差

空间，而且当且仅当在 x_2 点的边界条件得到满足时，**F** 的所有分量等于零。下面，将要求提供一个用户自编的函数 score，用(17.0.3)式把终止值 $\mathbf{y}(x_2)$ 的 N 维向量转换成一个 n_2 维偏差向量 **F**。

现在，就 Newton-Raphson 法而言，就得费一些周折了，需要找到一个 **V** 的向量值，让其使 **F** 向量值趋于零。通过调用在 9.7 节中，程序 newt 实现的全程收敛牛顿法来完成这一过程。牛顿法的核心涉及到求解 n_2 个线性方程组

$$\mathbf{J}\cdot\delta\mathbf{V}=-\mathbf{F} \tag{17.1.3}$$

的解，然后返回添加校正量

$$\mathbf{V}^{\text{new}}=\mathbf{V}^{\text{old}}+\delta\mathbf{V} \tag{17.1.4}$$

在(17.1.3)式中，雅可比矩阵 **J** 的元素由下式决定：

$$J_{ij}=\frac{\partial F_i}{\partial V_j} \tag{17.1.5}$$

用解析法计算这些偏导数是不现实的，而且每个值都需要对 N 个常微分方程单独积分，接着估值

$$\frac{\partial F_i}{\partial V_j}\approx\frac{F_i(V_0,\ldots,V_j+\Delta V_j,\ldots)-F_i(V_0,\ldots,V_j,\ldots)}{\Delta V_j} \tag{17.1.6}$$

这已在 newt 程序下的 fdjac 程序中自动完成。用户需要提供 newt 的输入只是程序 vecfunc，它是通过常微分方程组求积来计算 **F**。下面有一个合适的程序，称为 shoot，它作为一个实际参量传递到程序 newt 中。

```
#include "nr.h"

extern int nvar;                    用户需在主程序中定义和设置的变量。
extern DP x1,x2;

int kmax,kount;                     与程序 odeint 通信。
DP dxsav;
Vec_DP *xp_p;
Mat_DP *yp_p;

void derivs(const DP x, Vec_I_DP &y, Vec_O_DP &dydx);
void load(const DP x1, Vec_I_DP &v, Vec_O_DP &y);
void score(const DP xf, Vec_I_DP &y, Vec_O_DP &f);

void NR::shoot(Vec_I_DP &v, Vec_O_DP &f)
对 nvar 个联立常微分方程用 newt 程序通过从 x1 到 x2 打靶解决两点边值问题的程序。在 x1 处 nvar 个常微分方程的初始值是采用用户自编程序 load，利用 n2 个输入系数 v[0..n2-1]产生。程序积分常微分方程采用 Runge-Kutta 方法，容差限为 EPS，初始步长 h1，最小步长 hmin。在 x2 点调用用户自编程序 score 来计算 n2 个函数 f[0..n2-1]，f[0..n2-1]应该在满足 x2 处边界条件时变为零，函数值 f 在输出返回。newt 用个程收敛牛顿法来调整参数 v 直到 f 为零。用户提供的程序 derivs(x,y,dydx)提供常微分方程积分求导数信息(参见第 16 章)。上面第一组全程变量从主程序中接收值，以使 shoot 中能有 newt 的参数 vecfunc 所需的语法。
{
    const DP EPS=1.0e-14;
    int nbad,nok;
    DP h1,hmin=0.0;

    Vec_DP y(nvar);
    kmax=0;
    h1=(x2-x1)/100.0;
    load(x1,v,y);
    odeint(y,x1,x2,EPS,h1,hmin,nok,nbad,derivs,rkqs);
    score(x2,y,f);
}
```

对有些问题来说，初始化步长值 ΔV 也许与初始条件有很密切的关系。最直接的方法是修改 load 函数，使其包括一个作为另一个输出变量的参考步长 h1，并通过全程变量把它反馈给 fdjac。

这样一个完整的打靶法循环过程需要对 N 个联立常微分方程进行 n_2+1 次积分：一次积分用做估计当前误差程序，n_2 次对偏导数积分。每一次新的循环过程要求新的一轮 n_2+1 次积分。这也说明了两点边值问题和初值问题比较起来的确需要更多额外的工作量。

如果微分方程是线性的，则只需要一个完整的循环过程，因为(17.1.3)式～(17.1.4)式可以直接取得正确结果。但是再进行一次循环也是有用的，这可以减少部分(但不可能是全部)的舍入误差。

这里给出的程序 shoot 用 16.2 节的控制计算质量的 Runge-Kutta 方法来积分常微分方程组,不过第 16 章的任何方法都可以应用。

作为用户,必须为程序 shoot 提供:(i)函数 load(x1,v,y),用以计算 n 维向量 y[0..n-1](当然满足起始边界条件),在初始点 x1 给出 v[0..n2-1]的自定义变量;(ii)函数 score(x2,y,f),用以计算在终端边界条件偏差向量 f[0..n2-1],在终端点 x2 给出向量 y[0..n-1];(iii)起始向量 v[0..n2-1];(iv)函数 derivs 进行常微分方程的积分;以及其他在上面程序开头所说明的一些参数。

在 17.4 节中,将列举一个简单的程序,用以说明怎样使用程序 shoot。

参考文献和深入阅读

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America).

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems*; reprinted 1991 (New York: Dover).

17.2 对拟合点打靶

在 17.1 节中介绍的打靶法,约定了"射击"轨迹能够穿越整个的积分域,很快即可以收敛到正确的解上。有些问题发生的情况是,因为错误严重的起始条件,初始解从 x_1 到 x_2 要碰到某些不可计算的,或是灾难性的结果。例如,负数的平方根,导致了数值代码的混乱,使简单的打靶法复杂化。

一种不同的但是相关的情况是,两个端点都是常微分方程组的奇异点。通常需要用特别的方法在奇异点附近积分,例如用解析渐近法。在这样的情况下,沿着背离奇异点方向进行积分是可行的,用某些特殊的方法经过第一个点旁边一点点,然后求出"初始值"进行进一步的数值积分。但是,通常积分到某一奇异点是不行的,因为用户通常不用费很大的精力就得到奇异点附近的"错误"解的解析展开式(这些解并不满足所需的边界条件)。

解决上面所提困难的方法是用**拟合点打靶**的方法。我们并不从 x_1 积分到 x_2,而是首先从 x_1 积分到 x_1 和 x_2 之间的一点 x_f,然后再从 x_2(沿反方向)积分到 x_f。

如果和以前一样,在 x_1 点的边界条件数是 n_1,在 x_2 点的边界条件数是 n_2,则在起始 x_1 有 n_2 个可自由设定的初始值,而在 x_2 有 n_1 个可自由设定的起始值(如果这里感到有些疑惑,可以参看 17.1 节)。因此可以在 x_1 处定义一个 n_2 维起始参数向量 $\mathbf{V}^{(1)}$,以及一个把 $\mathbf{V}^{(1)}$ 映射到在 x_1 处满足边界条件的向量 $\mathbf{y}$ 的方法 load1(x1,v1,y),

$$y_i(x_1) = y_i(x_1; V_0^{(1)}, \ldots, V_{n_2-1}^{(1)}) \qquad i = 0, \ldots, N-1 \tag{17.2.1}$$

同样,在 x_2 处也能定义一个 n_1 维起始参数向量 $\mathbf{V}^{(2)}$,以及一个把 $\mathbf{V}^{(2)}$ 映射到在 x_2 处满足边界条件的向量 $\mathbf{y}$ 的方法 load2(x2,v2,y),

$$y_i(x_2) = y_i(x_2; V_0^{(2)}, \ldots, V_{n_1-1}^{(2)}) \qquad i = 0, \ldots, N-1 \tag{17.2.2}$$

这样,把 $\mathbf{V}^{(1)}$ 和 $\mathbf{V}^{(2)}$ 结合起来,就总共有 N 个可调整的参数。必须要满足的 N 个条件是,从两头积分得到的在 x_f 点处 $\mathbf{y}$ 向量的 N 个分量应该相同,即

$$y_i(x_f; \mathbf{V}^{(1)}) = y_i(x_f; \mathbf{V}^{(2)}) \qquad i = 0, \ldots, N-1 \tag{17.2.3}$$

在有些问题中,这 N 个匹配条件最好用 N 个不同函数 $F_i(i=0,\ldots,N-1)$(物理地、数学地或数值地)进行描述,每个函数也许都和 y_i 的 N 个元素有关。在这些情况下,(17.2.3)式由下式替代:

$$F_i[\mathbf{y}(x_f; \mathbf{V}^{(1)})] = F_i[\mathbf{y}(x_f; \mathbf{V}^{(2)})] \qquad i = 0, \ldots, N-1 \tag{17.2.4}$$

在下面的程序中,用户提供的函数 score(xf,y,f)是用来把输入的 N 维向量 $\mathbf{y}$ 映射成为输出的 N 维向量 $\mathbf{F}$。大多数情况下,可以把这个函数直接取为恒等映射 $\boldsymbol{F} = \mathbf{y}$。

射向某一拟合点的方法和 17.1 节中所使用的 Newton-Raphson 方法完全一样,参考前一节的程序 shoot 后,理解下面的程序 shootf 应该没有什么困难。在使用时的主要区别是,用户必须提供 load1 和 load2。而且,

在调用程序中必须给 v1[0..n2 − 1]和 v2[0..n1 − 1]提供初始猜测值。在 17.4 节中也给出了一个示例程序说明如何射向一拟合点。

```
#include "nr.h"

extern int n2;                    用户需要在主程序中定义和设置的变量。
extern DP x1,x2,xf;

int kmax,kount;                   与程序 odeint 传递。
DP dxsav;
Mat_DP *yp_p;
Vec_DP *xp_p;

void derivs(const DP x, Vec_I_DP &y, Vec_O_DP &dydx);
void load1(const DP x1, Vec_I_DP &v1, Vec_O_DP &y);
void load2(const DP x2, Vec_I_DP &v2, Vec_O_DP &y);
void score(const DP xf, Vec_I_DP &y, Vec_O_DP &f);

void NR::shootf(Vec_I_DP &v, Vec_O_DP &f)
用 newt 求解 nvar 个联立常微分方程组两点边值问题，从 x1 和 x2 射向某一拟合点 xf 的程序。nvar 个常微分方程组
x1(x2)的初始值由 n2(n1)个系数 v1(v2)产生(n1 = nvar − n2)，它使用用户提供的程序 load1(load2)获得。在主程序中，通
过语句 v1 属于 v[0,n2 − 1]和 v2 属于 v[n2,nvar − 1]把系数 v1 和 v2 存储在单个矩阵 v[0..n2 − 1]中。程序使用 Runge-
Kutta 方法积分常微方程到 xf，其容差限为 EPS，初始步长 h1 及最小步长 hmin。在 xf 处调用用户定义的程序 score，以计算
nvar 个函数 f1 和 f2，这些函数应在 xf 匹配。f 的差值应返回到输出。newt 采用全局收敛牛顿法，以调整 v 值，直到函数 f
等于零。用户提供的程序 derivs(x,y,dydx)给出常微分方程的导数信息(参见第 16 章)上面的第一个全局变量是从主程
序中接收它的值，以使 shoot 能有 newt 的参数 vectune 所要求的语法。
{
    const DP EPS=1.0e-14;
    int i,nbad,nok;
    DP h1,hmin=0.0;

    int nvar=v.size();
    Vec_DP f1(nvar),f2(nvar),y(nvar);
    Vec_DP v2(&v[n2],nvar-n2);
    kmax=0;
    h1=(x2-x1)/100.0;
    load1(x1,v,y);                具有最佳试验值 v1 的从 x1 到 xf 的路径。
    odeint(y,x1,xf,EPS,h1,hmin,nok,nbad,derivs,rkqs);
    score(xf,y,f1);
    load2(x2,v2,y);               具有最佳试验值 v2 的从 x2 到 xf 的路径。
    odeint(y,x2,xf,EPS,h1,hmin,nok,nbad,derivs,rkqs);
    score(xf,y,f2);
    for (i=0;i<nvar;i++) f[i]=f1[i]-f2[i];
}
```

有的边界值问题，即使采用射向某一拟合点也会出现问题——积分区间不得不用几个拟合点进行分段，而解则必须在每个拟合点上得到匹配，详见本节参考文献[1]。

参考文献和深入阅读：

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America).

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems*; reprinted 1991 (New York: Dover).

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §§7.3.5–7.3.6. [1]

17.3　深入讨论：松弛法

在**松弛法**中，把常微分方程用某些栅格或网格点上的近似**有限差分方程**(FDE)来代替，这些网格点张成了这个积分域。作为一个范例，可以把一个一般的一阶微分方程

$$\frac{dy}{dx} = g(x, y) \tag{17.3.1}$$

用一个与在两点 k，$k-1$ 函数值有关的代数方程

$$y_k - y_{k-1} - (x_k - x_{k-1})\, g\left[\tfrac{1}{2}(x_k + x_{k-1}), \tfrac{1}{2}(y_k + y_{k-1})\right] = 0 \tag{17.3.2}$$

代替,在(17.3.2)式中有限差分方程的形式只说明这种思想,但其形式并非惟一的:有很多方法可以把常微分方程转换成有限差分方程。当遇到用栅格上的 M 个点代替 N 个联立的一阶常微分方程的问题时,其解是 M 个栅格点上的 N 个因变函数的值,也就是总共 $N\times M$ 个变量。松弛法和前面介绍过的方法一样,首先预测一个解,然后对其进行迭代和修正以达到真实解。当用迭代法修正解时,结果便逐渐逼近到真实解。

可以用很多种迭代法,但对大多数问题来说,旧的、标准的多维牛顿法效果很好。这种方法需要求解一个矩阵方程,但该矩阵是一种特殊的"块对角化"形式。在求解的时候,可能比一般的$(MN)\times(MN)$矩阵节省很多时间和存储空间,因为 MN 很容易就达几千,这和方法是否可行关系密切。

在大多数成对的点上,实施过程都是匹配的,就像在(17.3.2)式中一样。匹配的点越多,则更增加了方法的复杂性。如果读者有耐心去编程,我们可以提供足够的知识背景。

求解一组由有限差分方程替代常微分方程的一般代数方程,常微分方程问题和(17.0.1)式~(17.0.3)式所表示的完全一样。这里,有 N 个联立的一阶方程,在区间一个端点满足 n_1 个边界条件,在另一个端点满足 $n_2=N-n_1$ 个边界条件。首先定义了一组 $k=0,1,\ldots,M-1$ 个点的栅格,我们提供了在这组栅格点上相应的变量 x_k 的值。特别是,x_0 是初始边界值,x_{M-1}是终点边界值。用 $\mathbf{y}_k$ 表示在点 x_k 处因变量 $y_0,y_1,\ldots,y_{N-1}$整组的值。在网格中间的任一点 k,可以用下列代数关系式来逼近 N 个一阶常微分方程的方程组:

$$0=\mathbf{E}_k\equiv\mathbf{y}_k-\mathbf{y}_{k-1}-(x_k-x_{k-1})\mathbf{g}_k(x_k,x_{k-1},\mathbf{y}_k,\mathbf{y}_{k-1}),\quad k=1,2,\ldots,M-1 \tag{17.3.3}$$

这个表达式表明 $\mathbf{g}_k$ 能用两点 $k,k-1$ 的信息进行计算,由 $\mathbf{E}_k$ 表示的有限差分方程提供了点 $k,k-1$ 的包含 $2N$ 个变量的 N 个方程。(17.3.3)形式的差分方程能解出 $M-1$ 个点,$k=1,2,\ldots,M-1$。这样,有限差分方程提供了$(M-1)N$ 个方程,求解 MN 个未知变量,其余的 N 个方程从边界条件导出。

在第一个边界值条件中,有

$$0=\mathbf{E}_0=\mathbf{B}(x_0,\mathbf{y}_0) \tag{17.3.4}$$

在第二个边界值条件中,有

$$0=\mathbf{E}_M\equiv\mathbf{C}(x_{M-1},\mathbf{y}_{M-1}) \tag{17.3.5}$$

向量 $\mathbf{E}_0$ 和 $\mathbf{B}$ 只有 n_1 个非零元,这对应于在 x_0 点的 n_1 个边界条件。把这些非零元放在最后的 n_1 个元素中,以后会有用的。也就是说,只对 $j=n_2,n_2+1,\ldots,N-1$ 来说 $E_{j,0}\neq0$。在另一个边界条件中,只有 $\mathbf{E}_M$和 $\mathbf{C}$ 的最初 n_2 个单元是非零的:$E_{j,M}$只对$j=0,1,\ldots,n_2-1$不为零。

在(17.3.3)式~(17.3.5)式中的有限差分方程问题的"解"由一组 $y_{j,k}$变量的值组成,也就是由在 M 个点 x_k 上 N 个变量的值组成。这样即可算出增量 $\Delta y_{j,k}$,而 $y_{j,k}+\Delta y_{j,k}$便是一个改进的近似解。

增量方程可以通过用相对较小的变化量 $\Delta\mathbf{y}_k$ 展开的一阶泰勒展开式来求解。在某一内点,$k=1,2,\ldots,M-1$,为

$$\begin{aligned}\mathbf{E}_k(\mathbf{y}_k+\Delta\mathbf{y}_k,\mathbf{y}_{k-1}+\Delta\mathbf{y}_{k-1})&\approx\mathbf{E}_k(\mathbf{y}_k,\mathbf{y}_{k-1})\\&+\sum_{n=0}^{N-1}\frac{\partial\mathbf{E}_k}{\partial y_{n,k-1}}\Delta y_{n,k-1}+\sum_{n=0}^{N-1}\frac{\partial\mathbf{E}_k}{\partial y_{n,k}}\Delta y_{n,k}\end{aligned} \tag{17.3.6}$$

对真实解来说,希望修正值 $\mathbf{E}(\mathbf{y}+\Delta\mathbf{y})$变成零。这样,一般的方程组在某一内点能写成如下矩阵形式:

$$\sum_{n=0}^{N-1}S_{j,n}\Delta y_{n,k-1}+\sum_{n=N}^{2N-1}S_{j,n}\Delta y_{n-N,k}=-E_{j,k},\quad j=0,1,\ldots,N-1 \tag{17.3.7}$$

其中

$$S_{j,n}=\frac{\partial E_{j,k}}{\partial y_{n,k-1}},\quad S_{j,n+N}=\frac{\partial E_{j,k}}{\partial y_{n,k}},\quad n=0,1,\ldots,N-1 \tag{17.3.8}$$

在每个点 k 的 $S_{j,n}$参量为一个 $N\times2N$ 矩阵。这样,每个内点提供一组 N 个方程,它把 $2N$ 个校正量匹配到点 $k,k-1$ 的解的变量上。

同样,边界值的代数关系式可以展开成一阶泰勒展开式来修正解。因为 $\mathbf{E}_0$ 只依赖于 $\mathbf{y}_0$,在第一个边界条件,发现有

$$\sum_{n=0}^{N-1}S_{j,n}\Delta y_{n,0}=-E_{j,0},\quad j=n_2,n_2+1,\ldots,N-1 \tag{17.3.9}$$

其中

$$S_{j,n}=\frac{\partial E_{j,0}}{\partial y_{n,0}},\quad n=0,1,\ldots,N-1 \tag{17.3.10}$$

在第二个边界条件，有

$$\sum_{n=0}^{N-1} S_{j,n}\Delta y_{n,M-1}=-E_{j,M},\quad j=0,1,\ldots,n_2-1 \tag{17.3.11}$$

其中

$$S_{j,n}=\frac{\partial E_{j,M}}{\partial y_{n,M-1}},\quad n=0,1,\ldots,N-1 \tag{17.3.12}$$

这样，通过(17.3.7)式～(17.3.12)式得到一组线性方程，可求解校正量 $\Delta\mathbf{y}$，迭代直到校正量足够小为止。这个方程组的结构比较特殊，因为每个 $S_{j,n}$ 只和点 k，$k-1$ 有关。图 17.3.1 说明了九个变量和四个网格点，且在第一边界处满足三个边界条件，在第二边界处满足两个边界条件的完全矩阵方程的典型情况。在矩阵左上角的那一块 3×5 的非零项来自点 $k=1$ 的边界条件 $S_{j,n}$；另外三个 5×10 的块是在内部点的 $S_{j,n}$，而且它在网格点(2,1)、(3,2)和(4,3)上把变量联立组合起来；最后一块是满足第二边界值条件的项。

利用高斯消元法求解(17.3.7)式～(17.3.12)式，求出增量 $\Delta\mathbf{y}$。这种方法利用了矩阵的特殊结构把总的计算量减至最少。而且这种方法通过把元素合并成一个特殊的块结构，而把矩阵系数的存储量减至最小(如果不太熟悉高斯消元法，可以参见第 2 章尤其是 2.2 节)。高斯消元法求解方程组时，用了初等运算的操作，如通过某行同除以某一公因子，使在对角线上产生单位元，并再加上其他行的适当倍数，把对角线以下的元素化为零。这里，利用了块结构的优点，比纯粹高斯消元法进一步简化，这样系数存储量也减少了。图 17.3.2表示通过消元想要得到的形式，这是在回代步骤以前的形式。在消元过程中，只有很小的一个经约简后的 $MN\times MN$ 矩阵元素很小的子集合需要进行存储。一旦矩阵变成了图 17.3.2 的形式，通过回代过程很快即可以得到解。

而且整个求解过程，除了最后的回代过程，每次都只对矩阵的一块进行操作。这个求解过程包括四种类型的操作：(1)利用前一步的结果经约简后把部分元素化为零；(2)通过消元把剩下的块结构进行消元，让矩阵对角线上的元素为单位 1，非对角线上元素为 0；(3)把余下非零系数存储起来以备后用；(4)回代。

```
X X X X X                                   V   B
X X X X X                                   V   B
X X X X X                                   V   B
X X X X X X X X X X                         V   B
X X X X X X X X X X                         V   B
X X X X X X X X X X                         V   B
X X X X X X X X X X                         V   B
X X X X X X X X X X                         V   B
          X X X X X X X X X X               V   B
          X X X X X X X X X X               V   B
          X X X X X X X X X X               V   B
          X X X X X X X X X X               V   B
          X X X X X X X X X X               V   B
                    X X X X X X X X X X     V   B
                    X X X X X X X X X X     V   B
                    X X X X X X X X X X     V   B
                    X X X X X X X X X X     V   B
                    X X X X X X X X X X     V   B
                              X X X X X     V   B
                              X X X X X     V   B
```

图 17.3.1　在两端点给定边界值条件的线性有限差分方程的矩阵结构。X 代表有限差分方程组的系数，V代表解向量中的未知元，B表示已知的方程右端的值，空白区表示为零。该矩阵方程将通过特殊形式的高斯消元法进行求解(详见正文)

```
1     X X                                   V   B
  1   X X                                   V   B
    1 X X                                   V   B
      1         X X                         V   B
        1       X X                         V   B
          1     X X                         V   B
            1   X X                         V   B
              1 X X                         V   B
                1         X X               V   B
                  1       X X               V   B
                    1     X X               V   B
                      1   X X               V   B
                        1 X X               V   B
                          1         X X     V   B
                            1       X X     V   B
                              1     X X     V   B
                                1   X X     V   B
                                  1 X X     V   B
                                    1       V   B
                                      1     V   B
```

图 17.3.2 高斯消元法的目标结构图。图 17.3.1 中的矩阵一旦化成这个结构以后，通过回代过程马上可以得出解

对于满足初始化边界条件，用以求解校正量的方程子块，有 n_1 个方程，N 个未知校正量。希望把第一子块进行变形，让子块的左边 $n_1 \times n_1$ 的方阵变成沿对角线为单位元，而非对角线都为零的形式。图 17.3.3 说明了矩阵第一块最初和最后的形式。在图中，把需要进行对角化的元素标明为"D"；要发生变化的元素标明为"A"；在最后的块结构中，要存储起来的元素标明为"S"，依次选择前 n_1 列的 n_1 个"主元"，把主元所在的行归一化，让主元的值为单位 1。然后，把该行适当的倍数加到其他行上去，使其他行中该主元所在列的元素都为零。在最后的形式中，约简的矩阵说明了，在网格点 0 的前 n_1 个变元的校正量的值，是如何根据网格点 0 处剩下的 n_2 个未知校正量的值进行表示的，即，我们现在知道了前 n_1 个分量是如何由其余 n_2 个分量决定的。因此，只需把初始块中最后 n_2 个非零的列，以及矩阵方程右端改变后的列存储起来。

```
(a) D D D A A     V   A
    D D D A A     V   A
    D D D A A     V   A

(b) 1 0 0 S S     V   S
    0 1 0 S S     V   S
    0 0 1 S S     V   S
```

图 17.3.3 对图 17.3.1 中矩阵的第一块(左上角)进行约简的过程。(a)块的最初形式，(b)最终形式(详见正文)

这里需要强调一下这个方法的一个很重要的问题。在操作块时要使用较小存储量的时候，有一点是必须考虑的，那就是导数矩阵 s 的列顺序应保证主元能在矩阵的前 n_1 行找到。就是说，在第一个点的 n_1 个边界条件必须和最初的 $j=0,1,\ldots,n_1-1$ 个因变量 y[j][0]相关。如果是的话，则第一块是最初的 $n_1 \times n_1$ 方阵就有可能是奇异的，这种方法便失败了。或者，必须在块中所有 N 列寻找主元，这就需要进行列的交换而需更多的记录工作。程序中提供了一个简便的方法重新对变量排序。例如对矩阵 S 的列重新排序，这项工作很容易完成。这便是重要的细节。

其次考虑表示有限差分方程组的 N 个方程。这 N 个方程描述了在点 1 和点 0 的 $2N$ 个校正量之间的关系。该块中的元素以及前一步的结果都在图 17.3.4 中说明了。注意，加上第一块中行的适当倍数，可以把该块中前 n_1 列(标为"Z")约简为零，而且这样做时只需要改变从第 n_1 到 $N-1$ 列以及方程右端的元素。对其他列可以把图中标有"D"的 $N \times N$ 方阵的元素对角化。最后，把图中标有"A"的 n_2 列的元素进行了改变。图 17.3.4 中的第二部分表示对矩阵进行处理后的结果，其中标有"S"的 $n_2 \times N$ 个元素被存储起来。

在对下一个方程组——它对应于在点 2 和点 1 耦合的校正量的有限差分方程组——进行运算时，会发

现，可用结果和新方程的状态准确地重现前一段中所介绍的情况。这样可以依次进行那些过程的计算，直到第 $M-1$ 子块。最后，在块 M 处，又遇到了另外一个边界条件。

图 17.3.5 表示定义在网格点 $M-1$ 处，N 个校正量的最后一块 n_2 个有限常微分方程，以及以前块的约简结果。首先，仍用前面的结果把该块的前 n_1 列化为零。在对剩下的方阵进行对角化的时候，即得到了有价值的结果：得出了在网格点 $M-1$ 处最终 n_2 个的校正量值。

```
(a)  1 0 0 S S                V  S
     0 1 0 S S                V  S
     0 0 1 S S                V  S
     Z Z Z D D D D D A A     V  A
     Z Z Z D D D D D A A     V  A
     Z Z Z D D D D D A A     V  A
     Z Z Z D D D D D A A     V  A
     Z Z Z D D D D D A A     V  A

(b)  1 0 0 S S                V  S
     0 1 0 S S                V  S
     0 0 1 S S                V  S
     0 0 0 1 0 0 0 0 S S      V  S
     0 0 0 0 1 0 0 0 S S      V  S
     0 0 0 0 0 1 0 0 S S      V  S
     0 0 0 0 0 0 1 0 S S      V  S
     0 0 0 0 0 0 0 1 S S      V  S
```

图 17.3.4　对图 17.3.1 中矩阵中间块的约简过程。(a)原始形式，(b)最终形式(详见正文)

```
(a)  0 0 0 1 0 0 0 0 S S      V  S
     0 0 0 0 1 0 0 0 S S      V  S
     0 0 0 0 0 1 0 0 S S      V  S
     0 0 0 0 0 0 1 0 S S      V  S
     0 0 0 0 0 0 0 1 S S      V  S
               Z Z Z D D     V  A
               Z Z Z D D     V  A

(b)  0 0 0 1 0 0 0 0 S S      V  S
     0 0 0 0 1 0 0 0 S S      V  S
     0 0 0 0 0 1 0 0 S S      V  S
     0 0 0 0 0 0 1 0 S S      V  S
     0 0 0 0 0 0 0 1 S S      V  S
               0 0 0 1 0     V  S
               0 0 0 0 1     V  S
```

图 17.3.5　对图 17.3.1 中最后一块(右下方)的约简过程。(a)初始形式，(b)最终形式(详见正文)

最后一子块约简以后，矩阵便为原先的图 17.3.2 表示的形式。矩阵即可以准备进行回代了。从最后一行开始，回推到最上面一行，在每一行，可以很简单地用已知量来决定未知校正量。

函数 solvde 把上面所述的各步组织起来。在算法中，所进行的基本过程是由函数 solvde 的内部调用函数来实现的。函数 red 把原来子块中 s 矩阵的头几列进行约简，函数 pinvs 把矩阵 s 中的各方阵对角化，并把未约简的系数存储起来。函数 bksub 实现往回替代的过程。作为函数 solvde 的用户必须了解所调用的参数，如下面所述，并提供一个被 solvde 调用的函数 difeq，用以计算矩阵 s 各块的值。

调用 solvde 的大多数变量都已经介绍过，但还需要一些讨论。数组 y[j][k]包含解的初始预测值，以 j 标号在网格点 k 处的因变量。问题涉及到 k = 0, . . . , m − 1 个扩展点上的 ne 个有限差分方程。在第一个点 k = 0 提供了 nb 个边界条件，数组 indexv[j]建立了 s 矩阵、(17.3.8)式、(17.3.10)式和(17.3.12)式各列之间与因变量的关系。如上所述，用在 k = 0 处的 nb 个边界条件来确定 s 矩阵前 nb 列所定义的因变量，这是很基本的一点。这样，用户能在程序 difeq 中排序 s 矩阵的第 j 列元素，以表示对应于因变量 indexv[j]的导数。

在返回以前，函数只准备进行 itmax 次校正循环，即使解尚未收敛也如此。参数 conv，slowc，scalv 和收敛有关。每次矩阵变换便产生在 m 个网格点处 ne 个变元的校正量。我们希望随着迭代过程的继续，这些校正量逐渐趋于零，但必须定义一个校正量大小的量度。误差的"限额"是和各问题有关的，所以用户也许想重写这部分程序代码。在下面的程序中，通过把所有校正量的绝对值加起来，并根据各种不同类型的变量加以不同的放大系数作为加权，来计算平均的校正误差 err：

$$\mathrm{err} = \frac{1}{\mathrm{m} \times \mathrm{ne}} \sum_{\mathrm{k}=0}^{\mathrm{m}-1} \sum_{\mathrm{j}=0}^{\mathrm{ne}-1} \frac{|\Delta Y[\mathrm{j}][\mathrm{k}]|}{\mathrm{scalv}[\mathrm{j}]} \tag{17.3.13}$$

当 err≤conv 时，这种方法收敛。注意，用户需提供一个数组 scalv 用以度量各个变量的标准大小。

显然，如果 err 很大，偏离解的范围就很大，而且也许从一阶泰勒系数得出的校正量是准确的这一想法本来就是错误的。slowc 数的大小调整了校正量的使用，在每次迭代以后，只应用了由矩阵求逆所求出校正量的一部分：

$$Y[\mathrm{j}][\mathrm{k}] \to Y[\mathrm{j}][\mathrm{k}] + \frac{\mathrm{slowc}}{\max(\mathrm{slowc}, \mathrm{err})} \Delta Y[\mathrm{j}][\mathrm{k}] \tag{17.3.14}$$

这样，当 err > slowc 时，只用了调整量的一部分，但当 err≤slowc 时，使用了整个调整量。

调用语句也提供给 solvde 函数包含初始试验解的数组 y[0..nyj-1][0..nyk-1],以及工作空间数组 c[0..ne-1][0..ne-nb][0..m],s[0..ne-1][0..2*ne]。数组 c 是一个较大的数据块,它存储了为回代步骤所准备的还未约简的矩阵元素。如果有 m 个网格点,就有 m+1 子块,每一子块又需要 ne 行和 ne-nb+1 列。尽管这个矩阵元素很多,但和未划分成块所需整个矩阵$(ne\times m)^2$ 个元素相比来说是很小的了。

现在介绍用户提供的函数 difeq 的工作过程,函数说明如下:

```
void difeq(const int k, const int k1, const int k2, const int jsf,
    const int is1, const int isf, Vec_I_INT &indexv,
    Mat_O_DP &s, Mat_I_DP &y);
```

从 difeq 传递给 solvde 的信息只是导数矩阵 s[0..ne-1][0..2*ne];所有其他参数都是输入给程序 difeq 的,而且不应被变动。k 表示当前网格点或是块序号,k1,k2 表示网格中第一个和最后一个点。如果 k=k1 或 k>k2 时,相应子块便要涉及在最初点和最终点的边界值条件,否则,相应子块只对与点 k-1,k 中变元有关的有限差分组起作用。

信息在数组 s[i][j]中存储的约定,遵循在(17.3.8)式、(17.3.10)式及(17.3.12)式中所用的约定:行 i 表示方程,列 j 表示解中的相应变元的导数值。记住,每个方程无论是在一个或两个点,都是依赖于 ne 个因变量元。这样,j 能从 1 变到 ne-1 或是 2*ne-1。每一点上因变量排列的列序必须和 indexv[j]中的列的顺序一致。这样,对于不在边界处的子块,第一列乘以 $\Delta Y(l=\text{indexv}[0],k-1)$,第 ne 列乘以 $\Delta Y(l=\text{indexv}[0],k)$。is1 和 isf 给出在 s 矩阵中,对于这一子块需要填充的开始和最后的行数。jsf 表示存储由(17.3.3)式~(17.3.5)式导出的差分方程 $E_{j,k}$ 的列数。这样,-s[i][jsf]即是矩阵右端的向量,之所以加上负号,是因为 difeq 提供的是实际的差分方程 $E_{j,k}$ 的值,而不是它的负值。注意,solvde 给 jsf 提供了一个值,这样恰好把该差分方程放在 s 矩阵中所有导数后边的列中。因此,difeq 希望能寻找放入 s[i][j]中的值,其中 is1≤i≤isf,而且 0≤j≤jsf。

最后,s[0..nsi-1][0..nsj-1]及 y[0..nyj-1][0..nyk-1]给函数 difeq 提供了 s 的存储空间和该次迭代的解变量 y。怎么使用这个程序将在下节中举例说明。

下面的程序代码中的很多思想出自 Eggleton[1]。

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include "nr.h"
using namespace std;

void NR::solvde(const int itmax, const DP conv, const DP slowc,
    Vec_I_DP &scalv, Vec_I_INT &indexv, const int nb, Mat_IO_DP &y)
用松弛法求解两点边值问题的驱动程序。itmax 是最大迭代次数,conv 是收敛准则(参见正文),slowc 控制每次迭代后实
际应用的校正量。scalv[0..ne-1]包含每个因变量的标准化大小,用以加权计算误差。indexv[0..ne-1]列出了用于构
造导数矩阵 s 中变量的列序(在第一个网格点处的 nb 个边界条件必须和在 indexv 中列出的前 nb 个变量相关)。该问题
在每点上涉及 ne 个方程,有 ne 个可调整的因变量。在第一个网格点上,有 nb 个边界值条件。总共有 m 个网格点。
y[0..ne-1][0..m-1]是一个二维数组,它包含在每一网点处所有因变量的初始预测值。在每次迭代中,它被所有计算
的校正量所更改。
{
    int ic1,ic2,ic3,ic4,it,j,j1,j2,j3,j4,j5,j6,j7,j8,j9;
    int jc1,jcf,jv,k,k1,k2,km,kp,nvars;
    DP err,errj,fac,vmax,vz;

    int ne=y.nrows();
    int m=y.ncols();
    Vec_INT kmax(ne);
    Vec_DP ermax(ne);
    Mat3D_DP c(ne,ne-nb+1,m+1);
    Mat_DP s(ne,2*ne+1);
    k1=0; k2=m;                          设置行、列标志。
    nvars=ne*m;
    j1=0,j2=nb,j3=nb,j4=ne,j5=j4+j1;
    j6=j4+j2,j7=j4+j3,j8=j4+j4,j9=j8+j1;
    ic1=0,ic2=ne-nb,ic3=ic2,ic4=ne;
    jc1=0,jcf=ic3;
    for (it=0;it<itmax;it++) {           初次迭代循环。
```

```
        k=k1;                                   在第一点的边界条件。
        difeq(k,k1,k2,j9,ic3,ic4,indexv,s,y);
        pinvs(ic3,ic4,j5,j9,jc1,k1,c,s);
        for (k=k1+1;k<k2;k++) {                 在所有成对点的有限差分方程。
            kp=k;
            difeq(k,k1,k2,j9,ic1,ic4,indexv,s,y);
            red(ic1,ic4,j1,j2,j3,j4,j9,ic3,jc1,jcf,kp,c,s);
            pinvs(ic1,ic4,j3,j9,jc1,k,c,s);
        }
        k=k2;                                   最后的边界条件。
        difeq(k,k1,k2,j9,ic1,ic2,indexv,s,y);
        red(ic1,ic2,j5,j6,j7,j8,j9,ic3,jc1,jcf,k2,c,s);
        pinvs(ic1,ic2,j7,j9,jcf,k2,c,s);
        bksub(ne,nb,jcf,k1,k2,c);               回代。
        err=0.0;
        for (j=0;j<ne;j++) {                    检查收敛性，累积平均误差。
            jv=indexv[j];
            errj=vmax=0.0;
            km=0;
            for (k=k1;k<k2;k++) {               对每因个变量，找出最大误差的点。
                vz=fabs(c[jv][0][k]);
                if (vz > vmax) {
                    vmax=vz;
                    km=k+1;
                }
                errj += vz;
            }
            err += errj/scalv[j];               给各因变量加权。
            ermax[j]=c[jv][0][km-1]/scalv[j];
            kmax[j]=km;
        }
        err /= nvars;
        fac=(err > slowc ? slowc/err : 1.0);
        误差较大时，减小所用校正量。
        for (j=0;j<ne;j++) {                    应用校正量。
            jv=indexv[j];
            for (k=k1;k<k2;k++)
            y[j][k] -= fac*c[jv][0][k];
        }
        cout << setw(8) << "Iter.";                     累积这一步的校正量。
        cout << setw(10) << "Error" << setw(10) << "FAC" << endl;
        cout << setw(6) << it;
        cout << fixed << setprecision(6) << setw(13) << err;
        cout << setw(12) << fac << endl;        可以通过写出 kmax 和 ermax 数组监视每个
        if (err < conv) return;                 变量的最大误差点。
    }
    nrerror("Too many iterations in solvde");           收敛失败。
}


#include "nr.h"

void NR::bksub(const int ne, const int nb, const int jf, const int k1,
    const int k2, Mat3D_IO_DP &c)
向后回代.在 solvde 中调用.
{
    int nbf,im,kp,k,j,i;
    DP xx;

    nbf=ne-nb;
    im=1;
    for (k=k2-1;k>=k1;k--) {            利用递归关系消去余下的依赖关系。
        if (k == k1) im=nbf+1;
```

```
            kp=k+1;
            for (j=0;j<nbf;j++) {
                xx=c[j][jf][kp];
                for (i=im-1;i<ne;i++)
                    c[i][jf][k] -= c[i][j][k]*xx;
            }
        }
        for (k=k1;k<k2;k++) {        重排第一列中的校正量。
            kp=k+1;
            for (i=0;i<nb;i++) c[i][0][k]=c[i+nbf][jf][k];
            for (i=0;i<nbf;i++) c[i+nb][0][k]=c[i][jf][kp];
        }
    }



#include <cmath>
#include "nr.h"
using namespace std;

void NR::pinvs(const int ie1, const int ie2, const int je1, const int jsf,
    const int jc1, const int k, Mat3D_O_DP &c, Mat_IO_DP &s)
把矩阵 s 中的子方阵对角化,并存储 c 中的递归系数;在 solvde 中调用。
{
    int jpiv,jp,je2,jcoff,j,irow,ipiv,id,icoff,i;
    DP pivinv,piv,dum,big;

    const int iesize=ie2-ie1;
    Vec_INT indxr(iesize);
    Vec_DP pscl(iesize);
    je2=je1+iesize;
    for (i=ie1;i<ie2;i++) {              隐式选主元,参见 2.1 节。
        big=0.0;
        for (j=je1;j<je2;j++)
            if (fabs(s[i][j]) > big) big=fabs(s[i][j]);
        if (big == 0.0)
            nrerror("Singular matrix - row all 0, in pinvs");
        pscl[i-ie1]=1.0/big;
        indxr[i-ie1]=0;
    }
    for (id=0;id<iesize;id++) {
        piv=0.0;
        for (i=ie1;i<ie2;i++) {          寻找主元素。
            if (indxr[i-ie1] == 0) {
                big=0.0;
                for (j=je1;j<je2;j++) {
                    if (fabs(s[i][j]) > big) {
                        jp=j;
                        big=fabs(s[i][j]);
                    }
                }
                if (big*pscl[i-ie1] > piv) {
                    ipiv=i;
                    jpiv=jp;
                    piv=big*pscl[i-ie1];
                }
            }
        }
        if (s[ipiv][jpiv] == 0.0)
            nrerror("Singular matrix in routine pinvs");
        indxr[ipiv-ie1]=jpiv+1;          同址约化,保存已排序后的列。
        pivinv=1.0/s[ipiv][jpiv];
        for (j=je1;j<=jsf;j++) s[ipiv][j] *= pivinv;     归一化主元行。
```

```
            s[ipiv][jpiv]=1.0;
            for (i=ie1;i<ie2;i++) {          约化列中的非主元素。
                if (indxr[i-ie1] != jpiv+1) {
                    if (s[i][jpiv] != 0.0) {
                        dum=s[i][jpiv];
                        for (j=je1;j<=jsf;j++)
                            s[i][j] -= dum*s[ipiv][j];
                        s[i][jpiv]=0.0;
                    }
                }
            }
        }
        jcoff=jc1-je2;                       排列并存储未约化的系数。
        icoff=ie1-je1;
        for (i=ie1;i<ie2;i++) {
            irow=indxr[i-ie1]+icoff;
            for (j=je2;j<=jsf;j++) c[irow-1][j+jcoff][k]=s[i][j];
        }
    }


    #include "nr.h"

    void NR::red(const int iz1, const int iz2, const int jz1, const int jz2,
        const int jm1, const int jm2, const int jmf, const int ic1,
        const int jc1, const int jcf, const int kc, Mat3D_I_DP &c,
        Mat_IO_DP &s)
    用存储在 c 矩阵的原先结果来简化 s 矩阵中第 jz1..jz2 - 1 列，只有 jm1..jm2 - 1 和 jmf 列会受以前结果的影响，red 被
    solvde 内部调用。
    {
        int loff,l,j,ic,i;
        DP vx;

        loff=jc1-jm1;
        ic=ic1;
        for (j=jz1;j<jz2;j++) {                  对要零化的列循环。
            for (l=jm1;l<jm2;l++) {              对要变动的列循环。
                vx=c[ic][l+loff][kc-1];
                for (i=iz1;i<iz2;i++) s[i][l] -= s[i][j]*vx;     对行循环。
            }
            vx=c[ic][jcf][kc-1];
            for (i=iz1;i<iz2;i++) s[i][jmf] -= s[i][j]*vx;       加上最终元素。
            ic += 1;
        }
    }
```

17.3.1　微分方程的“代数困难”集

松弛法允许利用的另一个便利之处虽然不很明显，但能极大地提高某些计算的速度。变量 $y_{j,k}$ 集没有必要直接和原始微分方程因变量相关，它们可以通过代数方程和那些变量联系。显然，解变量允许我们计算函数 $y,g,\mathbf{B},\mathbf{C}$，而它们是用于从常微分方程来构造有限差分方程的。这是惟一必要的。在某些问题中，g 依赖于隐式的 y 函数，需要用迭代解法来计算常微分方程中的函数。通常情况下，可以通过定义一个新的变量集来处理这种“内部”非线性问题，从这个新的变量集中，可以直接得出 y,g 和边界条件。典型的例子是在物理学中，其中要求一个复杂的状态方程的解。而在状态方程中，用其他变量而不是用常微分方程中的原始因变量的形式来表达会更加方便。这种近似过程，就类似于直接对原始的常微分方程进行了一次解析变换，这种解析也许是极为复杂的，但在松弛法中的变量替换很容易，而且不需要解析操作。

参考文献和深入阅读

Eggleton, P.P. 1971, *Monthly Notices of the Royal Astronomical Society*, vol. 151, pp. 351–364. [1]

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems*; reprinted 1991 (New York: Dover).

Kippenhan, R., Weigert, A., and Hofmeister, E. 1968, in *Methods in Computational Physics*, vol. 7 (New York: Academic Press), pp. 129ff.

17.4 实例:球体调和函数

理解前一节算法的最好方法是分析它们在实际问题中是如何使用的。作为范例,我们选择了球体调和函数的计算(更一般的名字是球体角函数,但我们更喜欢球体调和函数这种隐含说法),将说明怎样找球体调和函数,首先采用松弛法(参见17.3节),然后用打靶法(参见17.1节)和射向某一拟合点方法(参见17.2节)。

球体调和函数一般出现在球体坐标系中,用变量分离法求解某种偏微分方程组的情况。它们在区间$-1 \leqslant x \leqslant 1$上满足下面的微分方程:

$$\frac{d}{dx}\left[(1-x^2)\frac{dS}{dx}\right]+\left(\lambda-c^2x^2-\frac{m^2}{1-x^2}\right)S=0 \quad (17.4.1)$$

其中m是一个整数,c是"偏心率",λ是特征值。另外,c^2可正可负,对于$c^2>0$,函数叫做"长椭形",而$c^2<0$则称为"扁椭形"。方程在$x=\pm 1$处有奇异点,而且求解时,方程的边界值条件要在$x=\pm 1$处满足,只有对特征值λ的某些特定取值,这种情况才可能。

首先考虑第一种球面的情况,$c=0$,把微分方程当做勒让德(Legendre)函数$P_n^m(x)$。在这种情况下,特征值是$\lambda_{mn}=n(n+1)$,$n=m,m+1,\ldots$。整数n表示对固定m值的连续特征值:当$n=m$时,有最小的特征值,而且在区间$-1<x<1$中相应的特征函数没有结点;当$n=m+1$时,有下一个特征值,特征函数在$(-1,1)$中有一个结点,依次类推。

对于一般$c^2\neq 0$,有一种类似情况成立。把(17.4.1)式中特征值写做$\lambda_{mn}(c)$,特征函数记做$S_{mn}(x;c)$。对固定的m,$n=m,m+1,\ldots$标明了相继的特征值。

一般$\lambda_{mn}(c)$和$S_{mn}(x;c)$的计算就很困难。关于复杂的递归关系式,幂级数展开等都可以在本节参考文献[1~3]中找到。对微分方程直接求解的简便计算很容易实现。

第一步是考查奇异点$x=\pm 1$附近的解的性态。在(17.4.1)式中用幂级数展开式

$$S=(1\pm x)^{\alpha}\sum_{k=0}^{\infty}a_k(1\pm x)^k \quad (17.4.2)$$

来替换,我们发现正则解有$\alpha=m/2$(不失一般性,可以取$m\geqslant 0$,因为方程关于$m\to -m$的对称性,)。把这个解提出公因子,即得到一个更容易数值法跟踪求解的方程。相应地,得到

$$S=(1-x^2)^{m/2}y \quad (17.4.3)$$

这样,由(17.4.1)式发现y满足方程

$$(1-x^2)\frac{d^2y}{dx^2}-2(m+1)x\frac{dy}{dx}+(\mu-c^2x^2)y=0 \quad (17.4.4)$$

其中

$$\mu\equiv\lambda-m(m+1) \quad (17.4.5)$$

在$x\to -x$时,(17.4.1)式和(17.4.4)式都是不变的。因此函数S和y也必须是不变的,除非可能有一个放大因子(因为方程组是线性的,常数和解相乘以后仍是满足方程的解),因为解要被归一化,所以放大因子只可能是± 1。如果$n-m$是奇数,则在区间$(-1,1)$中能有奇数个零点。因此必须选择一个反对称的解$y(-x)=-y(x)$,它在$x=0$处有一个零点;反之,如果$n-m$是偶数,必须有对称解。这样,

$$y_{mn}(-x)=(-1)^{n-m}y_{mn}(x) \quad (17.4.6)$$

对S_{mn}是类似的。

对(17.4.4)式的边界条件要求y在$x=\pm 1$处应是正则的。换言之,在端点附近解的形式应是

$$y=a_0+a_1(1-x^2)+a_2(1-x^2)^2+\ldots \quad (17.4.7)$$

把这个表达式代入(17.4.4)式中,并让$x\to 1$,发现

$$a_1=-\frac{\mu-c^2}{4(m+1)}a_0 \quad (17.4.8)$$

等价地有,

$$y'(1) = \frac{\mu - c^2}{2(m+1)} y(1) \tag{17.4.9}$$

在 $x = -1$ 处，一个类似方程成立而等号右边有一负号。在端点处，非正则解在函数和导数之间有不同关系。

可以利用(17.4.6)式的对称性，从 0 到 1 积分来代替方程从 -1 到 1 的积分。在 $x = 0$ 处的边界条件为

$$\begin{aligned} y(0) &= 0, \quad n - m \text{ 为奇数} \\ y'(0) &= 0, \quad n - m \text{ 为偶数} \end{aligned} \tag{17.4.10}$$

第三个边界值条件来自这样一个事实：任意常数乘以某一解的结果仍是方程的解。这样可以对解进行**归一化**。采用这样的归一化，使在 $x = 1$ 时，函数 S_{mn} 和 P_n^m 有同样的极限特性：

$$\lim_{x \to 1} (1 - x^2)^{-m/2} S_{mn}(x; c) = \lim_{x \to 1} (1 - x^2)^{-m/2} P_n^m(x) \tag{17.4.11}$$

各种归一化方法的约定都由 Flammer[1]列表进行了说明。

对(17.4.4)式的二阶方程加上三个边界条件以后，问题即变成了求λ的特征值问题，或等价于求 μ 的特征值问题。把它写成标准形式，设置

$$y_0 = y \tag{17.4.12}$$

$$y_1 = y' \tag{17.4.13}$$

$$y_2 = \mu \tag{17.4.14}$$

则

$$y_0' = y_1 \tag{17.4.15}$$

$$y_1' = \frac{1}{1 - x^2} \left[2x(m+1) y_1 - (y_2 - c^2 x^2) y_0 \right] \tag{17.4.16}$$

$$y_2' = 0 \tag{17.4.17}$$

在这些符号表示中，$x = 0$ 处的边界条件是

$$\begin{aligned} y_0 &= 0, \quad n - m \text{ 为奇数} \\ y_1 &= 0, \quad n - m \text{ 为偶数} \end{aligned} \tag{17.4.18}$$

在 $x = 1$ 处，有两个条件：

$$y_1 = \frac{y_2 - c^2}{2(m+1)} y_0 \tag{17.4.19}$$

$$y_0 = \lim_{x \to 1} (1 - x^2)^{-m/2} P_n^m(x) = \frac{(-1)^m (n+m)!}{2^m m! (n-m)!} \equiv \gamma \tag{17.4.20}$$

现在，我们准备用前面章节的算法来处理这个问题。

17.4.1　松弛法

如果只需要几个λ或 S 的离散值，则打靶法也许是最快的方法。但是，如果需要一个较大的 c 的序列值，则用松弛法比较好。松弛法的好处是一个较好的初始预测解能较快地收敛，而且如果 c 变化较小时，先前求得的解应该是一个较好的初始预测解。

为简单起见，在区间 $0 \leqslant x \leqslant 1$ 中选择均匀的网格点。对总共 M 个网格点，有

$$h = \frac{1}{M - 1} \tag{17.4.21}$$

$$x_k = kh, \qquad k = 0, 1, \ldots, M - 1 \tag{17.4.22}$$

在内点 $k = 1, 2, \ldots, M - 1$ 处，(17.4.15)式给出

$$E_{0,k} = y_{0,k} - y_{0,k-1} - \frac{h}{2}(y_{1,k} + y_{1,k-1}) \tag{17.4.23}$$

(17.4.16)式给出

$$\begin{aligned} E_{1,k} = {} & y_{1,k} - y_{1,k-1} - \beta_k \\ & \times \left[\frac{(x_k + x_{k-1})(m+1)(y_{1,k} + y_{1,k-1})}{2} - \alpha_k \frac{(y_{0,k} + y_{0,k-1})}{2} \right] \end{aligned} \tag{17.4.24}$$

其中

$$\alpha_k = \frac{y_{2,k} + y_{2,k-1}}{2} - \frac{c^2(x_k + x_{k-1})^2}{4} \tag{17.4.25}$$

$$\beta_k = \frac{h}{1 - \frac{1}{4}(x_k + x_{k-1})^2} \tag{17.4.26}$$

最后,(17.4.17)式给出

$$E_{2,k} = y_{2,k} - y_{2,k-1} \tag{17.4.27}$$

在上节中说明过,(17.3.8)式中偏导数矩阵 $S_{i,j}$ 的定义,它用 i 标明方程,而 j 标明变量。在这种情况下,在 $k-1$ 处,y_j 中 j 从 0 到 2;而在 k 处,y_j 从 3 到 5。因此,(17.4.23)式给出

$$\begin{aligned} &S_{0,0} = -1, \quad S_{0,1} = -\frac{h}{2}, \quad S_{0,2} = 0 \\ &S_{0,3} = 1, \quad S_{0,4} = -\frac{h}{2}, \quad S_{0,5} = 0 \end{aligned} \tag{17.4.28}$$

同样,(17.4.24)式给出

$$\begin{aligned} &S_{1,0} = \alpha_k\beta_k/2, && S_{1,1} = -1 - \beta_k(x_k + x_{k-1})(m+1)/2, \\ &S_{1,2} = \beta_k(y_{0,k} + y_{0,k-1})/4, && S_{1,3} = S_{1,0}, \\ &S_{1,4} = 2 + S_{1,1}, && S_{1,5} = S_{1,2} \end{aligned} \tag{17.4.29}$$

而从(17.4.27)式发现

$$\begin{aligned} &S_{2,0} = 0, \quad S_{2,1} = 0, \quad S_{2,2} = -1 \\ &S_{2,3} = 0, \quad S_{2,4} = 0, \quad S_{2,5} = 1 \end{aligned} \tag{17.4.30}$$

在 $x=0$ 处,有边界值条件

$$E_{2,0} = \begin{cases} y_{0,0}, & n-m \text{ 为奇数} \\ y_{1,0}, & n-m \text{ 为偶数} \end{cases} \tag{17.4.31}$$

回忆在上节 solvde 程序中采用的约定,对 $k=0$ 处的一个边界条件,只有 $S_{2,j}$ 可以非零。同样,j 只能取 3 到 5 的值,这是因为边界条件只和 y_k 有关,而和 y_{k-1} 无关。相应地,在 $x=0$ 处 $S_{2,j}$ 的惟一非零值是

$$\begin{aligned} &S_{2,3} = 1, \quad n-m \text{ 为奇数} \\ &S_{2,4} = 1, \quad n-m \text{ 为偶数} \end{aligned} \tag{17.4.32}$$

在 $x=1$ 处,有

$$E_{0,M} = y_{1,M-1} - \frac{y_{2,M-1} - c^2}{2(m+1)} y_{0,M-1} \tag{17.4.33}$$

$$E_{1,M} = y_{0,M-1} - \gamma \tag{17.4.34}$$

这样

$$S_{0,3} = -\frac{y_{2,M-1} - c^2}{2(m+1)}, \quad S_{0,4} = 1, \quad S_{0,5} = -\frac{y_{0,M-1}}{2(m+1)} \tag{17.4.35}$$

$$S_{1,3} = 1, \quad S_{1,4} = 0, \quad S_{1,5} = 0 \tag{17.4.36}$$

下面给出实现上述算法的一个范例程序,需要一个主程序 sfroid,它调用 solvde 程序,而且还得提供一个由 solvde 调用的 difeq。为简单起见,选择用 m=41 个网格点的等距划分的网格,也就是说 $h=0.025$。正如我们将会看到的那样,这对特征值以及 $n-m$ 个调整值等都能保证很好的精确度。

因为如果 $n-m$ 为偶数的话,在 $x=0$ 处的边界条件和 y_0 无关,所以必须用 solvde 中的 indexv 特性。前面曾介绍过,indexv[j]描述了在 s[i][j]的哪一列中放入了变量 y[j]。如果 $n-m$ 是偶数,则必须把 y_0 和 y_1 的列相互对调,这样使 s[i][j]中没有零主元。

程序提示输入 m 和 n 的值,然后计算出满足勒让德函数 P_n^m 的 y 的初始预测值。其次提示输入 c^2,解出 y,再提示输入 c^2,用先前的值作为初始预测值解出 y,这样循环下去。

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include "nr.h"
using namespace std;

const int M=40;                                    与程序 difeq 通信的全局变量。
int mm,n,mpt=M+1;
DP h,c2=0.0,anorm;
Vec_DP *x_p;

int main(void)    // Program sfroid
使用 solvde 的范例程序。对 m≥0 和 n≥m 的情况计算球体调和函数 S_mn(x;c)的特征值。在程序中，m 是 mm，c^2 是 c2，
(17.4.20)式中的γ是 anorm。
{
    const int NE=3,NB=1,NYJ=NE,NYK=M+1;
    int i,itmax,k;
    DP conv,deriv,fac1,fac2,q1,slowc;
    Vec_INT indexv(NE);
    Vec_DP scalv(NE);
    Mat_DP y(NYJ,NYK);

    x_p=new Vec_DP(M+1);
    Vec_DP &x=*x_p;                                另取名以简化代码。
    itmax=100;
    conv=1.0e-14;
    slowc=1.0;
    h=1.0/M;
    cout << endl << "Enter m n" << endl;
    cin >> mm >> n;
    if ((n+mm & 1) != 0) {                         没必要调换。
        indexv[0]=0;
        indexv[1]=1;
        indexv[2]=2;
    } else {                                       调换 y0 和 y1。
        indexv[0]=1;
        indexv[1]=0;
        indexv[2]=2;
    }
    anorm=1.0;                                     计算γ。
    if (mm != 0) {
        q1=n;
        for (i=1;i<=mm;i++) anorm = -0.5*anorm*(n+i)*(q1--/i);
    }
    for (k=0;k<M;k++) {                            初始猜测值。
        x[k]=k*h;
        fac1=1.0-x[k]*x[k];
        fac2=exp((-mm/2.0)*log(fac1));
        y[0][k]=NR::plgndr(n,mm,x[k])*fac2;        P_n^m 在 6.6 节中介绍过。
        deriv = -((n-mm+1)*NR::plgndr(n+1,mm,x[k])-  从递归关系式求 P_n^m 导数。
            (n+1)*x[k]*NR::plgndr(n,mm,x[k]))/fac1;
        y[1][k]=mm*x[k]*y[0][k]/fac1+deriv*fac2;
        y[2][k]=n*(n+1)-mm*(mm+1);
    }
    x[M]=1.0;                                      在 x = 1 处的初始猜测值已单
    y[0][M]=anorm;                                 独给出。
    y[2][M]=n*(n+1)-mm*(mm+1);
    y[1][M]=(y[2][M]-c2)*y[0][M]/(2.0*(mm+1.0));
    scalv[0]=fabs(anorm);
    scalv[1]=(y[1][M] > scalv[0] ? y[1][M] : scalv[0]);
    scalv[2]=(y[2][M] > 1.0 ? y[2][M] : 1.0);
    for (;;) {
        cout << endl << "Enter c**2 or 999 to end" << endl;
        cin >> c2;
        if (c2 == 999) {
            delete x_p;
            return 0;
        }
        NR::solvde(itmax,conv,slowc,scalv,indexv,NB,y);
        cout << endl << " m = " << setw(3) << mm;
        cout << " n = " << setw(3) << n << " c**2 = ";
```

```
        cout << fixed << setprecision(3) << setw(7) << c2;
        cout << " lamda = " << setprecision(6) << (y[2][0]+mm*(mm+1));
        cout << endl;                                   返回c^2 的另一值。
    }
}


#include "nr.h"

extern int mm,n,mpt;                                    在 sfroid 中定义。
extern DP h,c2,anorm;
extern Vec_DP *x_p;

void NR::difeq(const int k, const int k1, const int k2, const int jsf,
    const int is1, const int isf, Vec_I_INT &indexv, Mat_O_DP &s,
    Mat_I_DP &y)
为 solvde 返回矩阵 s。
{
    DP temp,temp1,temp2;

    Vec_DP &x=*x_p;                                     第一点的边界条件。
    if (k == k1) {
        if ((n+mm & 1) != 0) {
            s[2][3+indexv[0]]=1.0;                      (17.4.32)式。
            s[2][3+indexv[1]]=0.0;
            s[2][3+indexv[2]]=0.0;
            s[2][jsf]=y[0][0];                          (17.4.31)式。
        } else {
            s[2][3+indexv[0]]=0.0;                      (17.4.32)式。
            s[2][3+indexv[1]]=1.0;
            s[2][3+indexv[2]]=0.0;
            s[2][jsf]=y[1][0];                          (17.4.31)式。
        }
    } else if (k > k2-1) {                              在末点的边界条件。
        s[0][3+indexv[0]] = -(y[2][mpt-1]-c2)/(2.0*(mm+1.0));        (17.4.35)式。
        s[0][3+indexv[1]]=1.0;
        s[0][3+indexv[2]] = -y[0][mpt-1]/(2.0*(mm+1.0));
        s[0][jsf]=y[1][mpt-1]-(y[2][mpt-1]-c2)*y[0][mpt-1]/          (17.4.33)式。
            (2.0*(mm+1.0));
        s[1][3+indexv[0]]=1.0;                          (17.4.36)式。
        s[1][3+indexv[1]]=0.0;
        s[1][3+indexv[2]]=0.0;
        s[1][jsf]=y[0][mpt-1]-anorm;                    (17.4.34)式。
    } else {                                            内点。
        s[0][indexv[0]] = -1.0;                         (17.4.28)式。
        s[0][indexv[1]] = -0.5*h;
        s[0][indexv[2]]=0.0;
        s[0][3+indexv[0]]=1.0;
        s[0][3+indexv[1]] = -0.5*h;
        s[0][3+indexv[2]]=0.0;
        temp1=x[k]+x[k-1];
        temp=h/(1.0-temp1*temp1*0.25);
        temp2=0.5*(y[2][k]+y[2][k-1])-c2*0.25*temp1*temp1;
        s[1][indexv[0]]=temp*temp2*0.5;                 (17.4.29)式。
        s[1][indexv[1]] = -1.0-0.5*temp*(mm+1.0)*temp1;
        s[1][indexv[2]]=0.25*temp*(y[0][k]+y[0][k-1]);
        s[1][3+indexv[0]]=s[1][indexv[0]];
        s[1][3+indexv[1]]=2.0+s[1][indexv[1]];
        s[1][3+indexv[2]]=s[1][indexv[2]];
        s[2][indexv[0]]=0.0;                            (17.4.30)式。
        s[2][indexv[1]]=0.0;
        s[2][indexv[2]] = -1.0;
        s[2][3+indexv[0]]=0.0;
        s[2][3+indexv[1]]=0.0;
        s[2][3+indexv[2]]=1.0;
        s[0][jsf]=y[0][k]-y[0][k-1]-0.5*h*(y[1][k]+y[1][k-1]);       (17.4.23)式。
        s[1][jsf]=y[1][k]-y[1][k-1]-temp*((x[k]+x[k-1])              (17.4.24)式。
            *0.5*(mm+1.0)*(y[1][k]+y[1][k-1])-temp2
            *0.5*(y[0][k]+y[0][k-1]));
        s[2][jsf]=y[2][k]-y[2][k-1];                    (17.4.27)式。
    }
}
```

读者可以运行这个程序，然后和 Flammer 书中[1]的表，或是在 Abramowitz 和 Stegun 书中[2]表 21.1 给出的$\lambda_{mn}(c)$的值进行对照。通常情况下，大约 3 次迭代后就收敛。表 17.4.1 给出了一些比较值。

表 17.4.1　从 sfroid 中选择的输出结果

m	n	c^2	λ_{exact}	λ_{sfroid}
2	2	0.1	6.014 27	6.014 27
		1.0	6.140 95	6.140 95
		4.0	6.542 50	6.542 53
2	5	1.0	30.4361	30.4372
		16.0	36.9963	37.0135
4	11	-1.0	131.560	131.554

17.4.2　打靶法

为用打靶法(参见 17.1 节)解决同样的问题，我们提供了一个函数 derivs 来实现(17.4.15)式～(17.4.17)式，将在区间 $-1\leqslant x\leqslant 0$ 上积分该方程组。提供了函数 load 把特征值 y_3 设置为当前最佳估计值 v[0]。而且用(17.4.20)式和(17.4.19)式设置 y_0 和 y_1 的边界值(有一个负号，相应于 $x=-1$)。注意，边界条件实际上提供了一个到边界的距离 dx，以避免计算边界右边的 y_1'。函数 score 来自于(17.4.18)式。

```
#include <iostream>
#include <iomanip>
#include "nr.h"
using namespace std;

int m,n;                                与 load, score 和 derivs 通信。
DP c2,dx,gmma;
int nvar;                               与 shoot 通信。
DP x1,x2;

int main(void)    // Program sphoot
使用打靶法的范例程序. 对 m≥0 和 n≥m 的情况计算球体调和函数 S_mn(x;c)的特征值。注意 shoot 如何为 newt 提供
程序 vecfunc(参见 17.1 节).
{
    const int N2=1;
    bool check;
    int i;
    DP q1;
    Vec_DP v(N2);

    dx=1.0e-8;                          避免在 x = -1 处求导数。
    nvar=3;                             方程个数。
    for (;;) {
        cout << endl << "input m,n,c-squared (999 to end)" << endl;
        cin >> m >> n >> c2;
        if (c2 == 999) break;
        if (n < m || m < 0) continue;
        gmma=1.0;                           计算 (17.4.20) 式的γ。
        q1=n;
        for (i=1;i<=m;i++) gmma *= -0.5*(n+i)*(q1--/i);
        v[0]=n*(n+1)-m*(m+1)+c2/2.0;        特征值的初始猜测值。
        x1= -1.0+dx;                        设积分区域。
        x2=0.0;
        NR::newt(v,check,NR::shoot);        找出使 score 中函数 f 为零的 v。
        if (check) {
            cout << "shoot failed; bad initial guess" << endl;
        } else {
            cout << " " << "mu(m,n)" << endl;
            cout << fixed << setprecision(6);
            cout << setw(12) << v[0] << endl;
        }
    }
    return 0;
}

void load(const DP x1, Vec_I_DP &v, Vec_O_DP &y)
```

```
提供在 x = -1+dx 处的积分初始值。
{
    DP y1 = ((n-m & 1) != 0 ? -gmma : gmma);
    y[2]=v[0];
    y[1] = -(y[2]-c2)*y1/(2*(m+1));
    y[0]=y1+y[1]*dx;
}

void score(const DP xf, Vec_I_DP &y, Vec_O_DP &f)
检测边界条件在 x=0 处是否满足。
{
    f[0]=((n-m & 1) != 0 ? y[0] : y[1]);
}

void derivs(const DP x, Vec_I_DP &y, Vec_O_DP &dydx)
计算 odeint 导数值。
{
    dydx[0]=y[1];
    dydx[1]=(2.0*x*(m+1.0)*y[1]-(y[2]-c2*x*x)*y[0])/(1.0-x*x);
    dydx[2]=0.0;
}
```

17.4.3 射向某一拟合点

为全面起见,我们说明一下 17.2 节中的程序 shootf,其积分域为 $-1+dx \leqslant x \leqslant 1-dx$,而拟合点选在 $x=0$ 处。程序 derivs 和 shoot 的调用一样。现在有两个 load 程序。对 $x=-1$,程序 load1 和上面的 load 大致相同,在 $x=1$ 处,load2 设置函数值 y_0 和特征值 y_2 为最佳当前值,分别为 v2[0]和 v2[1]。能够十分合理地使初始的特征猜测值在两个区间一样,则在迭代过程中 v1[0]和 v2[1]将保持相等。函数 score 用来简单地检查三个函数值在拟合点是否相等。

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include "nr.h"
using namespace std;

int m,n;                                    与 load1, load2, score, derivs 通信。
DP c2,dx,gmma;

int n2;                                     与 shootf 通信。
DP x1,x2,xf;

int main(void)    // Program sphfpt
使用 shootf 的范例程序。在 m≥0 和 n≥m 条件下计算球体调和函数 S_mn(x;c)的特征值。注意 shootf(参见 17.2 节)如何
为 newt 提供程序 vecfunc。程序 derivs 和上面的 sphoot 一样。
{
    const int N1=2,N2=1,NTOT=N1+N2;         方程数目。
    const DP DXX=1.0e-8;
    bool check;
    int i;
    DP q1;
    Vec_DP v(NTOT);

    n2=N2;
    dx=DXX;                                 避免在 x = ±1 处求导。
    for (;;) {
        cout << endl << "input m,n,c-squared (n >= m, c=999 to end)" << endl;
        cin >> m >> n >> c2;
        if (c2 == 999) break;
        if (n < m || m < 0) continue;
        gmma=1.0;                           计算 (17.4.20) 式的 γ。
        q1=n;
        for (i=1;i<=m;i++) gmma *= -0.5*(n+i)*(q1--/i);
        v[0]=n*(n+1)-m*(m+1)+c2/2.0;        初始化特征值和函数值的猜测值。
        v[2]=v[0];
        v[1]=gmma*(1.0-(v[2]-c2)*dx/(2*(m+1)));
        x1= -1.0+dx;                        设积分域。
        x2=1.0-dx;
        xf=0.0;                             拟合点。
```

```
        NR::newt(v,check,NR::shootf);      找出在 score 中使函数 f 为零的 v。
        if (check) {
            cout << "shootf failed; bad initial guess" << endl;
        } else {
            cout << " " << "mu(m,n)" << endl;
            cout << fixed << setprecision(6);
            cout << setw(12) << v[0] << endl;
        }
    }
    return 0;
}

void load1(const DP x1, Vec_I_DP &v1, Vec_O_DP &y)
提供在 x = 1 - dx 处的积分初始值。
{
    DP y1 = ((n-m & 1) != 0 ? -gmma : gmma);
    y[2]=v1[0];
    y[1] = -(y[2]-c2)*y1/(2*(m+1));
    y[0]=y1+y[1]*dx;
}

void load2(const DP x2, Vec_I_DP &v2, Vec_O_DP &y)
检测解在合适点 x = 1 - dx 处是否相等。
{
    y[2]=v2[1];
    y[0]=v2[0];
    y[1]=(y[2]-c2)*y[0]/(2*(m+1));
}

void score(const DP xf, Vec_I_DP &y, Vec_O_DP &f)
Computes the mismatch of the solutions at the fitting point x = 0
{
    int i;

    for (i=0;i<3;i++) f[i]=y[i];
}

void derivs(const DP x, Vec_I_DP &y, Vec_O_DP &dydx)
{
    dydx[0]=y[1];
    dydx[1]=(2.0*x*(m+1.0)*y[1]-(y[2]-c2*x*x)*y[0])/(1.0-x*x);
    dydx[2]=0.0;
}
```

参考文献和深入阅读

Flammer, C. 1957, *Spheroidal Wave Functions* (Stanford, CA: Stanford University Press). [1]

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §21. [2]

Morse, P.M., and Feshbach, H. 1953, *Methods of Theoretical Physics*, Part II (New York: McGraw-Hill), pp. 1502ff. [3]

17.5　深入讨论：网格点的自动分配

在松弛问题中，需要选择在网格点处自变量的值，这称为网格点的**分配**。通常是选用一些似乎合理的值作为其值。如果可行的话，则这组值满意；如果不行的话，增加网格点的数目，通常就可解决这个问题。

如果事先知道解在什么地方变化得较快，则可以在那里多设几个网格点，而在其他地方少设几个。或者可以用均匀的网格先求出解，然后查看应该在哪里多加一些点，接着用改进后的网格点重复求解过程。其目的都是为了设置网格点，以便尽可能精确地表示解。

同样，自动进行网格点的分配也是可能的，这样，即可以在松弛过程中“动态地”进行求解。这种高效的技巧不只改变了松弛法的求解精度，而且（正如下一节要介绍的）可以用很优良的方法解决内部奇异性的问题。下面学习自动分配网格点是如何实现的。

把注意力集中于自变量 x,以及考虑两种交替的再次参数化方法。首先,给出 q,这正是对应于网格点本身的坐标,这样在 $k=0$ 处,$q=0$;在 $k=1$ 处,$q=1$,依次类推。在任意两个网格点之间,有 $\Delta q=1$。在常微分方程中自变量 x 变为 q 后,

$$\frac{d\mathbf{y}}{dx}=\mathbf{g} \tag{17.5.1}$$

变成了

$$\frac{d\mathbf{y}}{dq}=\mathbf{g}\frac{dx}{dq} \tag{17.5.2}$$

按 q 的形式,(17.5.2)式作为有限差分方程可以写成

$$\mathbf{y}_k-\mathbf{y}_{k-1}-\frac{1}{2}\left[\left(\mathbf{g}\frac{dx}{dq}\right)_k+\left(\mathbf{g}\frac{dx}{dq}\right)_{k-1}\right]=0 \tag{17.5.3}$$

或其他相关的表达式。注意 dx/dq 应该是和 $\mathbf{g}$ 在一起,x 和 q 之间的变换只依赖于雅可比系数 dx/dq,其倒数 dq/dx 和网格点密度成正比。

现在,若给出函数 $\mathbf{y}(x)$,在当前松弛步骤的近似式,则能决定网格点的密度。例如,可能希望在 $\mathbf{y}$ 变化较快之处,或是在边界之处,或在上述两处,将 dq/dx 的值取得大一些。事实上,也许可以构造这样一个式子,让 dq/dx 和它们成正比,但问题是我们不知道这个比例常数。也就是说,构造的式子在 x 的取值范围内——根据其定义,使 q 从 0 变到 $M-1$——可能没有正确的积分。为了解决这个问题,我们引入了第二次再参数化$Q(q)$,其中Q是一个新的自变量。Q和 q 之间的关系取为线性的,这样网格划分间距公式对 dQ/dx 来说,不同的只是其未知的比例常数。由一个线性方程可知

$$\frac{d^2Q}{dq^2}=0 \tag{17.5.4}$$

或是用通常的联立一阶方程组

$$\frac{dQ(x)}{dq}=\psi \qquad \frac{d\psi}{dq}=0 \tag{17.5.5}$$

其中,ψ是一个新的中间变量。把这两个方程添加到求解的常微分方程组中。

完成变量描述以后,添加第三个常微分方程,也就是所需要的网格点密度函数,即

$$\phi(x)=\frac{dQ}{dx}=\frac{dQ}{dq}\frac{dq}{dx} \tag{17.5.6}$$

其中,$\phi(x)$由我们选定,若按网格变量 q 来写这个方程,即为

$$\frac{dx}{dq}=\frac{\psi}{\phi(x)} \tag{17.5.7}$$

注意,$\phi(x)$应该选为正定的,以使网点密度在各处都是正的,否则(17.5.7)式中分母可能为零。

用自动网格分划,把(17.5.5)式~(17.5.7)式这三个常微分方程加入方程组中,例如加到数组 y[j][k]中。现在 x 变成了一个因变量。Q和ψ也变成新的因变量。通常,计算ϕ不需要多少额外工作量,因为它可以从已有g的那些部分中构成。自动网格分划过程允许用户先看一下,数值结果会如何受各种网格点分划方法的影响(如果网格间距函数Q能通过解析得到,这是一种特例,即 dQ/dx 直接可积。这样,只需额外加上两个方程,也就是(17.5.5)式中的两个方程,以及两个新的变量 x,ψ)。

作为实现这种方法的一种典型范例,考虑一个因变量 $y(x)$的系统。可以设置

$$dQ=\frac{dx}{\Delta}+\frac{|d\ln y|}{\delta} \tag{17.5.8}$$

或是

$$\phi(x)=\frac{dQ}{dx}=\frac{1}{\Delta}+\left|\frac{dy/dx}{y\delta}\right| \tag{17.5.9}$$

其中 Δ 和 δ 是我们选择的常数。如果只有第一项,则它给出关于 x 的均匀分划。而第二项则是使在 y 迅速

变化的地方使用更多的网格点。常数的作用是使 y 的每个对数变化量 δ 对在 x 网格点变化量 Δ 有相当的影响。可以通过试验来调整这些常数。当然其他方法也是可以的，诸如可用 x 的对数分划，也就是在第一项中用 $d\ln x$ 来代替 dx。

参考文献和深入阅读

Eggleton, P. P. 1971, *Monthly Notices of the Royal Astronomical Society*, vol. 151, pp. 351–364.

Kippenhan, R., Weigert, A., and Hofmeister, E. 1968, in *Methods in Computational Physics*, vol. 7 (New York: Academic Press), pp. 129ff.

17.6　深入讨论：内部边界条件或奇异点的处理

奇异性可能会出现在两点边值问题的内部。典型地，假设有一个点 x_s，在该点的导数必须用下面的表达式计算：

$$S(x_s)=\frac{N(x_s,\mathbf{y})}{D(x_s,\mathbf{y})} \tag{17.6.1}$$

其中分母 $D(x_s,\mathbf{y})=0$。在有限解的物理问题中，奇异点的问题通常伴随着自身可以解决的对策：当 $D\to 0$ 时，物理解 $\mathbf{y}$ 必须同时让 $N\to 0$。这样，这个比例式便取有意义的值了。对解 $\mathbf{y}$ 的限制通常称为**正则性条件**。使 $D(x_s,\mathbf{y})$ 在 x_s 点满足某些特殊限制的条件，完全类似于一个额外的外界条件，即在某一点处因变量必须满足一个代数关系。

在 17.2 节中，介绍了用"拟合点方法"处理边界条件奇异的方程的积分问题，我们也曾讨论过这个相关的问题。在那些问题中，无法从积分域的一端积分到另一端，但是常微分方程在奇异点附近确实有很好特性的导数和解，所以可以从丢弃该点进行积分。松弛法和"射向某一拟合点"的方法解决这个问题都很容易。同样，在那些问题中，奇异特性的存在隔离了需要被满足进行求解的某些特殊的边界值。

这里所不同的是，我们考虑的奇异性出现在中间点上，奇异点的位置和解有关系，所有事先是未知的。因此，面对一个循环的问题：奇异性妨碍我们寻求数值解法，而我们又需要用数值法寻找奇异点的位置。这样的奇异性也和给某个变量选择一特殊值有关，这些变量允许解在奇异点处满足正则性条件。这样，内部奇异性的问题便采用了内部边界条件的问题。

处理内部奇异性的一种方法是，把这个问题当做一个自由边值问题，像我们在 17.0 节最后讨论的那样。作为一个简单的例子，考虑方程

$$\frac{dy}{dx}=\frac{N(x,y)}{D(x,y)} \tag{17.6.2}$$

其中 N 和 D 都需要在某一未知点 x_s 穿过零点。添加方程

$$z\equiv x_s-x_1 \qquad \frac{dz}{dx}=0 \tag{17.6.3}$$

其中 x_s 是未知的奇异点位置，并且通过设置

$$x-x_1=tz, \qquad 0\le t\le 1 \tag{17.6.4}$$

把自变量改成 t。在 $t=1$ 处的边界条件变成

$$N(x,y)=0, \qquad D(x,y)=0 \tag{17.6.5}$$

另一种方法是，用前一节讨论的自适应网格点控制来处理内部奇异性的问题。对于(17.6.2)式中的问题，添加网格点分划方程

$$\frac{dQ}{dq}=\psi \tag{17.6.6}$$

$$\frac{d\psi}{dq}=0 \tag{17.6.7}$$

增加一个简单的网格点分划函数,它把 x 直接映射到 q,其中 q 从 0 变到 $M-1$。网格点的数目为

$$Q(x)=x-x_1,\quad \frac{dQ}{dx}=1 \tag{17.6.8}$$

加入这三个一阶微分方程以后,还必须加入相应的边界条件。如果没有奇异性的话,这些将简便地是

$$\text{在 } q=0:\qquad x=x_1,\quad Q=0 \tag{17.6.9}$$

$$\text{在 } q=M-1:\qquad x=x_2 \tag{17.6.10}$$

以及在 $q=0$ 处指定的总共 N 个值 y_i。在这种情况下,问题变成一个在 x_1 处指定了所有边界值条件的初始值问题,并且网格分划函数已是多余的。

但实际上,眼下要处理的条件是

$$\text{在 } q=0:\qquad x=x_1,\qquad Q=0 \tag{17.6.11}$$

$$\text{在 } q=M-1:\quad N(x,y)=0,\quad D(x,y)=0 \tag{17.6.12}$$

以及在 $q=0$ 处的 $N-1$ 个 y_i 的值。"丢失"的那个 y_i 值是要进行调整的,换言之,应如此调整,使得解以正则形式(零除零型),而不是以非正则形式(有限值除零)经过奇异点。还应注意,这些边界条件并不直接给 x_2 强制定义一个解,而使 x_2 变成一个可调参数,不断变化以匹配正则性条件。

在这个例子中,奇异性出现在一个边界处,因为边界位置是未知的,从而增加了复杂度,在上述的例子中,可以简单地把常微分方程积分到奇异点,然后作为另一个单独的问题,重新从奇异点开始积分到需要求解的地方。但是,当其他的奇异点出现在内部的情况中,并不能完全确定问题的全部所在:沿网格点还有某些更多的边界条件需要进一步满足。这些情况在原理上并不困难,但对 17.3 节中所给出的松弛法程序代码确实需要一些改变。实际上所需要做的是,在内部边界条件出现的网格点处,加一块"特殊"的方程块,然后进行一些适当的记录工作。

图 17.6.1 说明了一个实例,其中整个问题包含 5 个方程,在第一点有两个边界条件,一个"内在"边界条件和两个最终边界条件。

图 17.6.1 具有内部边界条件的有限差分方程之矩阵结构。内部条件引入了一个特殊块。(a)初始形式,和图17.3.1比较;(b)最终的形式,和图17.3.2比较

此图说明了矩阵中，特殊子块附近沿对角线的整个矩阵的结构。在区域的中间，一般子块中有 5 个方程（行）和 10 个未知量（列）。对于特殊子块前面的每一块，初始条件提供了足够的信息，把块的头两列化成零。5 个有限常微分方程至少除去了多于 5 列，最后的 3 列存储起来进行回代（如 17.3 节所示）。为了要处理额外的条件，跳出正常循环，并增加一个只有一个方程的特殊子块：内部边界条件。这极有效地降低了一列未约简的其余网格点系数的存储量，并且允许把以后块中的头 3 列约简成零。函数 red, pinvs, bksub 只需进行少量的重新编码就能容易处理这些情况。但因为每个问题只针对一种特殊情况，所以读者必须根据需要自己修改。

参考文献和深入阅读

London, R.A., and Flannery, B.P. 1982, *Astrophysical Journal*, vol. 258, pp. 260–269.

第 18 章　积分方程和反演理论

18.0　引言

许多人(除具备数值知识的人外)都以为积分方程的数值求解是个极其复杂的课题,因为迄今为止,数值分析教科书中几乎都没有包含此内容。实际上,积分方程数值求解方面的文献很多,目前还在不断地增加,并且也已出版了几本专著[1~3]。之所以单独成册出版的一个原因是,存在许多不同类型的积分方程,而且每类都有许多不同的、可能隐含的陷阱,即便是一个简单的情况也需要多种不同的算法进行处理。

线性积分方程和简单而又古老的线性方程之间有着紧密的对应关系,前者是在一个无限维函数空间内,函数间确定了一种线性积分关系;而后者则是在一个有限维向量空间内,向量间确定了一种线性关系。因为这种对应关系是大部分计算算法的核心,所以在讨论积分方程的分类前,有必要明确此概念。

Fredholm 方程涉及具有固定上、下限的定积分。**第一类非齐次 Fredholm 方程**形式如下:

$$g(t)=\int_a^b K(t,s)f(s)\,ds \tag{18.0.1}$$

其中,$f(t)$是一个待求的未知函数,而$g(t)$是一个已知的"右端项"(在积分方程中,由于某些原因,熟知的"右端项"习惯上却写在左边),有两个变量的函数 $K(t,s)$称为**核**。(18.0.1)式对应的矩阵方程为

$$\mathbf{K}\cdot\mathbf{f}=\mathbf{g} \tag{18.0.2}$$

其解为 $\mathbf{f}=\mathbf{K}^{-1}\cdot\mathbf{g}$,其中 $\mathbf{K}^{-1}$是逆矩阵。与(18.0.2)式类似,当g非零($g=0$ 的齐次情形几乎毫无用处)且 K 可逆时,(18.0.1)式就有惟一解。但是将看到,后一条件如同规则一样常常出现例外。

类似于有限维特征值问题

$$(\mathbf{K}-\sigma\mathbf{1})\cdot\mathbf{f}=\mathbf{g} \tag{18.0.3}$$

称为**第二类 Fredholm 方程**,通常写为

$$f(t)=\lambda\int_a^b K(t,s)f(s)\,ds+g(t) \tag{18.0.4}$$

同样,记号约定并不严格对应:(18.0.4)式中的λ对应着(18.0.3)式中的 $1/\sigma$,而 $\mathbf{g}$ 对应着 $-g/\lambda$。如果g(或 $\mathbf{g}$)为零,则称方程是齐次的。如果核 $K(t,s)$有界,则同(18.0.3)式一样,(18.0.4)式的齐次形式在具有最多可数无限的特征值集$\lambda=\lambda_n$, $n=1,2,\ldots$上有解。相应的解 $f_n(t)$为**特征函数**。当核对称时特征值为实数。

在g(或 $\mathbf{g}$)非零的非齐次情形,(18.0.3)式和(18.0.4)式除了当λ(或 σ)为特征值外均可解——因为当λ(或 σ)为特征值时积分算子(或矩阵)是奇异的。积分方程中的这种二分法称为 **Fredholm 择一性**。

第一类 Fredholm 方程通常是极病态的。核作用到一个函数上通常起着光滑的作用,因而需要对该算子求逆的解,就会对输入的小变化或误差变得极其敏感。光滑的结果实际上常常丢失信息,而逆运算却无法找回它。这种方程已有了专门的处理方法,通常称为**反演问题**。一般来说,它必须利用解性质的一些先验知识来增加信息。这些先验知识以某种方式用来恢复丢失的信息。将在 18.4 节中介绍这些方法。

第二类齐次 Fredholm 方程的病态通常要低得多。(18.0.4)式可改写为

$$\int_a^b [K(t,s)-\sigma\delta(t-s)]f(s)\,ds=-\sigma g(t) \tag{18.0.5}$$

其中 $\delta(t-s)$是一个 Dirac(狄拉克)δ 函数(为明确起见,将λ变为相应的 σ)。如果 σ 的值足够大,则(18.0.5)式实际上是对角占优的,因而是良态的。只有当 σ 很小时,才又回到病态情形。

第二类齐次 Fredholm 方程同样没有特别的病态。如果 K 是一个光滑算子，它便会将许多 f 值映为零，或接近零。因而，在 $\sigma=0(\lambda\to\infty)$ 附近会出现大量退化或近似退化的特征值，但这对计算并未产生特别的困难。现在可以看出，实际上使非齐次(18.0.5)式不发生病态的 σ 值的大小，通常比对角占优情形所要求的 σ 值小得多。由于 σ 项会使所有特征值发生偏移，因此它要足够大，以使一个光滑算子的一系列零附近的特征值偏离零，那么合成的算子就变得可逆(当然，除离散特征值外)。

Volterra 方程是 Fredholm 方程的一种特殊情形，即当 $s>t$ 时，核 $K(t,s)=0$。去除不必要的积分部分，Volterra 方程可写为积分上限为自变量 t 的形式。**第一类 Volterra 方程**

$$g(t)=\int_a^t K(t,s)f(s)\,ds \tag{18.0.6}$$

和它的对应方程类似，也有矩阵方程(现在用分量写出)

$$\sum_{j=0}^{k} K_{kj}f_j=g_k \tag{18.0.7}$$

与(18.0.2)式相比，发现 Volterra 方程对应着一个下(即左)三角矩阵 $\mathbf{K}$，零项均在对角线上方。由第 2 章可知，这样的矩阵方程只需简单通过前向代换就可解出。因此，Volterra 方程的解法也同样直接。当实验测量的噪声不占绝对优势时，第一类 Volterra 方程基本上是非病态的。对积分取上限会产生一个极强的阶跃，它通常能破坏核所有的光滑性质。

第二类 Volterra 方程写为

$$f(t)=\int_a^t K(t,s)f(s)\,ds+g(t) \tag{18.0.8}$$

其对应的矩阵方程是

$$(\mathbf{K}-\mathbf{1})\cdot\mathbf{f}=\mathbf{g} \tag{18.0.9}$$

其中 $\mathbf{K}$ 为下三角型矩阵。这些方程中没有λ的原因是：(i)在非齐次情形(g非零)，λ可以被 K 吸收；(ii)而在齐次情形($g=0$)下，有一定理，即核有界的第二类 Volterra 方程没有具有平方可积的特征函数的特征根。

将定义都限定在线性积分方程的情形。(18.0.1)或(18.0.6)式在非线性情形下，被积函数将是 $K(t,s,f(s))$而非 $K(t,s)f(s)$；(18.0.4)式或(18.0.8)式在非线性情形下，被积函数将是 $K(t,s,f(t),f(s))$。非线性 Fredholm 方程比相应的线性方程要复杂得多。幸运的是，在实际应用中这类方程出现得不多，本章基本上不考虑它们。但是求解非线性 Volterra 方程，通常只需对线性方程的算法稍加修改即可，这一点后面将讨论。

几乎所有积分方程的数值解法都要利用**求积分法则**，通常用高斯积分。现在可以复习一下 4.5 节，特别是那节末尾深入讨论的一些内容。

在下面各节，先讨论具有光滑核的第二类 Fredholm 积分方程(参见 18.1 节)。该节也要讨论非平凡求积法则，但我们将只处理方程的良性系统。然后，回到 Volterra 方程(参见 18.2 节)，将会发现简单直接的方法通常很适合这些方程。

在 18.3 节将讨论如何处理奇异核情形，主要针对 Fredholm 方程(包括第一类和第二类)。奇异性要求有特殊的积分法则，因而它们会破坏核的光滑性并使问题良态。

在 18.4 节～18.7 节将讨论反演问题。在 18.4 节中将对这个大专题进行简介。

这里应该注意，在 13.10 节中讨论过的小波变换不仅适用于数据压缩和信号处理，而且还可以将一些积分方程变换成能快速求解的稀疏线性问题。作为阅读本章的一部分，建议读者复习一下 13.10 节。

必须指出，有些课题，比如**积分－微分方程**已超出本书的讨论范围。如果想了解积分－微分方程的一些方法，见 Brunner[4]。

毋庸置疑，仅用一章的篇幅只能论及积分方程这个复杂课题的一些最基本方法。

参考文献和深入阅读

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [1]

Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S.I.A.M.). [2]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.). [3]

Brunner, H. 1988, in *Numerical Analysis 1987*, Pitman Research Notes in Mathematics vol. 170, D.F. Griffiths and G.A. Watson, eds. (Harlow, Essex, U.K.: Longman Scientific and Technical), pp. 18–38. [4]

Smithies, F. 1958, *Integral Equations* (Cambridge, U.K.: Cambridge University Press).

Kanwal, R.P. 1971, *Linear Integral Equations* (New York: Academic Press).

Green, C.D. 1969, *Integral Equation Methods* (New York: Barnes & Noble).

18.1 第二类 Fredholm 方程

求解方程

$$f(t) = \lambda \int_a^b K(t,s) f(s)\, ds + g(t) \tag{18.1.1}$$

中 $f(t)$ 的数值解。使用的方法称为 **Nystrom 方法**，这是一个非常基本的方法。它需要选取一些近似的**积分法则**：

$$\int_a^b y(s)\, ds = \sum_{j=0}^{N-1} w_j y(s_j) \tag{18.1.2}$$

其中集 $\{w_j\}$ 是积分法则的权，而 N 点 $\{s_j\}$ 集是横坐标。

应该采用什么积分法则呢？用低阶积分法则，比如重复梯形或 Simpson 法则当然能积分方程。但我们将看到，解法涉及 $O(N^3)$ 阶运算，因而最有效的方法倾向于用高阶积分法以保持 N 尽可能小。对于光滑、非奇异问题，没有什么方法能比得上高斯积分(如 Gauss-Legendre 积分，参见 4.5 节)。对于非光滑或有奇异核，见 18.3 节。

Delves 和 Mohamed[1]提出了比 Nystrom 方法更复杂的方法。对于简单的第二类 Fredholm 积分方程，它们认为："这场比赛的胜利者当然是 Nystrom 程序……采用 N 点 Gauss-Legendre 法则。该程序极其简单……这样的结果足以使一个数值分析学者深感叹服。"

如果将(18.1.2)式的积分法则应用到(18.1.1)式上，得到

$$f(t) = \lambda \sum_{j=0}^{N-1} w_j K(t, s_j) f(s_j) + g(t) \tag{18.1.3}$$

在积分点估算(18.1.3)式：

$$f(t_i) = \lambda \sum_{j=0}^{N-1} w_j K(t_i, s_j) f(s_j) + g(t_i) \tag{18.1.4}$$

令 f_i 为向量 $f(t_i)$，g_i 为向量 $g(t_i)$，K_{ij} 为矩阵 $K(t_i, s_j)$，并定义

$$\widetilde{K}_{ij} = K_{ij} w_j \tag{18.1.5}$$

则(18.1.4)式用矩阵记号表示为

$$(\mathbf{1} - \lambda \widetilde{\mathbf{K}}) \cdot \mathbf{f} = \mathbf{g} \tag{18.1.6}$$

这是 N 个未知量的 N 个线性代数方程组，可以通过标准三角形分解法(参见 2.3 节)来求解——正是在此出现了 $O(N^3)$ 次运算。解通常是良态的，除非 λ 和一个特征值很接近。

求出积分点 $\{t_i\}$ 的解后，其他点 t 的解怎样呢？简单地使用多项式插值是不行的，这会破坏精度。Nystrom 方法的关键点在于，采用(18.1.3)式做插值公式以保持解的精度。

这里给出两个求解第二类线性 Fredholm 方程的程序。程序 fred2 先建立(18.1.6)式，然后调用 ludcmp 和 lubksb 程序，利用 LU 分解求解。求 Gauss-Legendre 积分是先通过调用 gauleg 得到权和横坐标。程序 fred2 要求读者提供一个返回$g(t)$的外部函数及另一个返回λK_{ij}的外部函数。然后它返回积分点的解 f。它也返回积分点和权重。第二个程序 fredin 用这些数据来实现(18.1.3)式的 Nystrom 插值，并返回在区间$[a,b]$内任意点的 f 值。

```
#include "nr.h"

void NR::fred2(const DP a, const DP b, Vec_O_DP &t, Vec_O_DP &f, Vec_O_DP &w,
    DP g(const DP), DP ak(const DP, const DP))
解第二类线性 Fredholm 方程。输入 a 和 b 是积分限，g 和 ak 是由用户提供的分别返回g(t)和λ K(t,s)
的外部函数。程序返回包含有高斯积分横坐标 t_i，以及这些横坐标上的解 f 的数组 t[0..n-1]及 f[0..n-1]。同时也返回高
斯权重数组 w[0..n-1]，与 Nystrom 插值程序 fredin 一起使用。n 是高斯积分用的点数，由程序中提供的数组的维数决定。
{
    int i,j;
    DP d;

    int n=t.size();
    Mat_DP omk(n,n);
    Vec_INT indx(n);                如果不用Gauss-Legendre积分
    gauleg(a,b,t,w);                则用另一个程序替换gauleg。
    for (i=0;i<n;i++) {
        for (j=0;j<n;j++)           形成 1 - λK̃。
            omk[i][j]=DP(i == j)-ak(t[i],t[j])*w[j];
        f[i]=g(t[i]);
    }
    ludcmp(omk,indx,d);             解线性方程。
    lubksb(omk,indx,f);
}
```

```
#include "nr.h"

DP NR::fredin(const DP x, const DP a, const DP b, Vec_I_DP &t, Vec_I_DP &f,
    Vec_I_DP &w, DP g(const DP), DP ak(const DP, const DP))
给定包含有高斯积分横坐标和权重的数组 t[0..n-1]和 w[0..n-1]，由 fred2 给出解数组 f[0..n-1]，该函数利用 Nystrom 插值
公式返回 x 点的 f 值。作为输入，a 和 b 是积分限，g 和 ak 是用户提供的分别返回g(t)和λ K(t,s)的外部函数。
{
    int i;
    DP sum=0.0;

    int n=t.size();
    for (i=0;i<n;i++) sum += ak(x,t[i])*w[i]*f[i];
    return g(x)+sum;
}
```

利用高斯积分法的一个缺点是，没有简单的办法来取得结果中误差的估计。最实用的方法是将 N 增加，例如 50%，将两次估计值之间的差别，作为用较大的 N 值取得的结果之误差的一个保守估计。

现在，再看齐次方程的解。如果令$\lambda = 1/\sigma$，$\mathbf{g} = 0$，则(18.1.6)式变为一个标准特征方程

$$\tilde{\mathbf{K}} \cdot \mathbf{f} = \sigma \mathbf{f} \tag{18.1.7}$$

于是可以利用任何合适的矩阵特征程序来求解(参见第 11 章)。注意，如果原始的问题有一个对称的核，那么矩阵 $\mathbf{K}$ 是对称的。但是，由于大部分积分法的权 w_j 不相同，矩阵 $\tilde{\mathbf{K}}$((18.1.5)式)并不对称。对称矩阵特征值问题要简单得多，因此应尽可能地保持对称性。假定权为正值(高斯积分就是如此)，可以定义对角矩阵 $\mathbf{D} = \mathrm{diag}(w_j)$ 及其平方根，$\mathbf{D}^{1/2} = \mathrm{diag}(\sqrt{w_j})$。(18.1.7)式则变为

$$\mathbf{K} \cdot \mathbf{D} \cdot \mathbf{f} = \sigma \mathbf{f}$$

乘以 $\mathbf{D}^{1/2}$，得到

$$\left(\mathbf{D}^{1/2} \cdot \mathbf{K} \cdot \mathbf{D}^{1/2}\right) \cdot \mathbf{h} = \sigma \mathbf{h} \tag{18.1.8}$$

其中，$\mathbf{h} = \mathbf{D}^{1/2} \cdot \mathbf{f}$。现在(18.1.8)式具有了对称特征值问题的形式。

(18.1.7)式或(18.1.8)式的解一般给出 N 个特征值，其中 N 是所用的积分点数。对于平方可积核，这些值将给

出积分方程最小值 N 个特征值的良好近似值。**有限秩**的核(也称为**退化**或**可分**核)仅有有限个非零特征值(也可能没有)。可以通过一组零特征值来判定这种情形以取得高精度。当增加 N 来提高它们的精度时,非零特征值的个数保持常值不变。这里需要注意:一个非退化的核可以有无穷多个以 $\sigma=0$ 为聚点的特征值。随着 N 的增大,可以根据解的性质来区分这两种情形。如果是退化核,通常可用解析法求解,所有教材中都有这部分内容。

参考文献和深入阅读

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [1]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.).

18.2 Volterra 方程

现在讨论 Volterra 方程,其原型是第二类 Volterra 方程,

$$f(t)=\int_a^t K(t,s)f(s)\,ds+g(t) \tag{18.2.1}$$

Volterra 方程的大部分算法从 $t=a$ 开始,随着不断地迭代逐渐求出解。在该意义上,它们不仅类似向前代换(参见 18.0 节),而且类似于常微分方程的初值问题。事实上,许多常微分方程的算法都有 Volterra 方程的对应算法。

最简单的方法是,在均匀步长的网格点上求解方程:

$$t_i=a+ih,\quad i=0,1,\ldots,N,\qquad h\equiv\frac{b-a}{N} \tag{18.2.2}$$

为此必须选取一个积分法则。对于均匀网格,最简单的方法是梯形法则,(4.1.11)式:

$$\int_a^{t_i} K(t_i,s)f(s)\,ds=h\left(\tfrac{1}{2}K_{i0}f_0+\sum_{j=1}^{i-1}K_{ij}f_j+\tfrac{1}{2}K_{ii}f_i\right) \tag{18.2.3}$$

因而(18.2.1)式的梯形法是

$$\begin{aligned} f_0&=g_0\\ (1-\tfrac{1}{2}hK_{ii})f_i&=h\left(\tfrac{1}{2}K_{i0}f_0+\sum_{j=1}^{i-1}K_{ij}f_j\right)+g_i,\qquad i=1,\ldots,N \end{aligned} \tag{18.2.4}$$

(对于第一类 Volterra 方程,左式前面的 1 将不出现,g 的符号相反,余下的讨论也有相应简单的变化。)

(18.2.4)式是一个通过 $O(N^2)$ 次运算给出解的一个显式方法。与 Fredholm 方程不同,它不需要解一个线性方程组。因此,Volterra 方程常常比相应的 Fredholm 方程简单,已知后者确实涉及到线性系统的反演,有时甚至是很大的线性系统的反演。

求解 Volterra 方程的高效率在一定程序上被下述事实所抵消,即在实际应用中,该方程以方程组形式出现更为常见,如果将(18.2.1)式改写成以 m 个函数 $f(t)$ 为向量的方程,则核 $K(t,s)$ 为一个 $m\times m$ 矩阵。(18.2.4)式也应看成为一个向量方程。对每个 i,必须利用高斯消元法求解 $m\times m$ 线性代数方程组。

下面的程序 voltra 实现了该算法。要求读者提供一个返回在点 t 向量 $g(t)$ 的第 k 个函数的外部函数,以及一个返回矩阵 $K(t,s)$ 在 (t,s) 处的 (k,l) 元素的外部函数。然后,程序 voltra 返回规则步长点 t_i 的向量 $f(t)$。

```
#include "nr.h"

void NR::voltra(const DP t0, const DP h, Vec_O_DP &t, Mat_O_DP &f,
    DP g(const int, const DP),
    DP ak(const int, const int, const DP, const DP))
```

用扩展梯形法求解 m 个第二类线性 Volterra 方程组。作为输入,t0 是积分的起始点,h 是步长,g(k,t)是用户提供的返回 $g_k(t)$ 的外部函数,而 ak(k,l,t,s)则是一个由用户提供的返回矩阵 $K(t,s)$ 的 (k,l) 元素的外部函数。该积分的解返回到 f[0..m-1][0..n-1],相应的横坐标为 t[0..n-1],n-1 是取定的参数。m 的值由解矩阵 f 的行维数决定。

```
{
    int i,j,k,l;
    DP d,sum;

    int m=f.nrows();
    int n=f.ncols();
    Vec_INT indx(m);

    Vec_DP b(m);
    Mat_DP a(m,m);
    t[0]=t0;
    for (k=0;k<m;k++) f[k][0]=g(k,t[0]);      初始赋值。
    for (i=1;i<n;i++) {                       取步长 h。
        t[i]=t[i-1]+h;
        for (k=0;k<m;k++) {
            sum=g(k,t[i]);                    将线性方程的右端项累加到sum中。
            for (l=0;l<m;l++) {
                sum += 0.5*h*ak(k,l,t[i],t[0])*f[l][0];
                for (j=1;j<i;j++)
                    sum += h*ak(k,l,t[i],t[j])*f[l][j];
                if (k == l)                   左端项放到矩阵a中。
                    a[k][l]=1.0-0.5*h*ak(k,l,t[i],t[i]);
                else
                    a[k][l] = -0.5*h*ak(k,l,t[i],t[i]);
            }
            b[k]=sum;
        }
        ludcmp(a,indx,d);                     解线性方程。
        lubksb(a,indx,b);
        for (k=0;k<m;k++) f[k][i]=b[k];
    }
}
```

对于非线性 Volterra 方程，(18.2.4)式仍然成立，但乘积 $K_{ij}f_i$ 要变为 $K_{ij}(f_i)$，另两个乘积 K 和 f 也做类似变化。因而对每个 i，都已知右端项来求非线性方程的解 f_i。如果步长不太大的话，则采用具有 f_{i-1} 的初始猜测值的牛顿法(参见 9.4 节或 9.6 节)，其效果通常很好。

我们认为，用高阶方法解 Volterra 方程，不如用高阶方法解 Fredholm 方程那样重要，因为 Volterra 方程相对易解。不过和这个课题有关的文献却很多，随之也产生了一些困难。首先，当初始几点的值未知时，任何同时在几个积分点运算以获取高阶的方法，都需要采用一个特别的方法来开始。

其次，稳定积分法则会使积分方程产生意外的不稳定性。例如，假定想用 Simpson 法来代替上述算法中的梯形法。Simpson 法自然地对长为 $2h$ 的间隔做积分，因此很容易得到偶网格点上的函数值。对于奇网格点，可以附加用一次梯形法。但应该加在积分的哪一端呢？可以用梯形法先做一步，再用 Simpson 法；也可以先用 Simpson 法，最后一步再用梯形法。遗憾的是，前者不稳定，而后者稳定。

和上文给出的梯形法可以一起使用的一个简单方法是 Richardson 外推法：用步长 h 和 $h/2$ 求解。然后，假定误差的量为 h^2，计算

$$f_{\mathrm{E}} = \frac{4f(h/2) - f(h)}{3} \tag{18.2.5}$$

这个方法也适用于龙贝格积分。

目前普遍认为，高阶方法中最好的是**逐块法**(见本节参考文献[1])。另一个重要的课题是，变步长法的应用，这在 K 或 f 有尖峰变化时更有效。变步长法比微分方程中相应的方法要复杂得多，建议读者参阅本节参考文献[1,2]进行深入研究。

还应该注意被积函数的奇异性。如果有奇异性，那么参考 18.3 节可以找到一些新方法。

参考文献和深入阅读

Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S.I.A.M.). [1]

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [2]

18.3 深入讨论:具有奇异核的积分方程

许多积分方程,或核或解或核与解都有奇异性。如果忽略这些奇异性,那么简单的积分法对 N 的收敛性很差。如何最好地处理奇异性,有时需要一定的技巧。

从一些简单的建议开始:

1. 可积奇异点常常可以通过变量替换消除掉,例如在 $s=0$ 附近 $K(t,s)$ 的 $s^{1/2}$ 或 $s^{-1/2}$ 奇异性通过变换 $z=s^{1/2}$ 而消除掉。注意,假定奇异性限制在 K 上,而积分实际上涉及的是乘积 $K(t,s)f(s)$,因此需要"固定"的正是这个乘积。理想情形下,必须证明乘积的奇异性后,才能进行数值求解和采用适当的方法。然而,奇异核常常并不产生奇异解 $f(t)$(例如,高奇异性核 $K(t,s)=\delta(t-s)$ 实际上是一个恒等算子)。

2. 如果 $K(t,s)$ 能分解为 $w(s)\bar{K}(t,s)$,其中 $w(s)$ 是奇异的,而 $\bar{K}(t,s)$ 是光滑的,那么以 $w(s)$ 为权函数的高斯积分法效果很好。甚至当分解近似成立时,收敛性常常也会改进很多。需要做的只是将 fred2 程序中的 gauleg 用另一个积分程序代替即可。在4.5节中介绍了如何构造这样的积分,或者也可以在本节参考文献[1,2]中查找列表的横坐标和权。当然,K 必须用 $\bar{K}$ 来替换。

该方法是**乘积 Nystrom 方法**(参见本节参考文献[3,4])的一种特殊情形,乘积 Nystrom 方法从 K 中分解出一个同时依赖于 t 和 s 的奇异项 $p(t,s)$,并为其高斯积分构造了合适的权。通常这种计算相当复杂,因为权不仅依赖于 $\{t_i\}$ 的选取,而且依赖于 $p(t,s)$ 的形式。

建议用均匀网格点来实现乘积 Nystrom 方法,采用把对任意权函数适用的扩展 Simpson 3/8 法((4.1.5)式)进行推广后得到的积分方案。将在下一节讨论该方法。

3. 当核不严格但却"几乎"奇异时,一些特殊的积分公式就很有用。例如,当核集中在 $t=s$ 附近,其变化范围远远小于解 $f(t)$ 的变化范围。在这种情形下,积分公式可以通过计算核 $K(t,s)$ 在表格点 t_i 的最初几个矩,并利用多项式或样条对 $f(s)$ 的局部近似来获取。在这种方案下,核的窄宽度变得有用了,而不成为负担;当核的宽度为零时,积分变为精确值。

4. 无限区域的积分也是奇异性的一种形式。在很大的有限值处将区域截断应该是不得已的做法。如果核很快地趋于零,那么 Gauss-Laguerre[$w\sim\exp(-\alpha s)$]或 Gauss-Hermite[$w\sim\exp(-s^2)$]积分的效果应该很好。长尾函数通常采用变换

$$s=\frac{2\alpha}{z+1}-\alpha \tag{18.3.1}$$

它将 $0<s<\infty$ 映射到 $1>z>-1$,这样就可以用 Gauss-Legendre 积分。其中 $\alpha>0$ 为一个可调整的常数,用来改进收敛性。

5. 实际应用中 $K(t,s)$ 常常是沿着对角线 $t=s$ 有奇异性。这时 Nystrom 方法完全不适用了,因为核在 (t_i,s_i) 处得到估计值。而**奇异性减法**可能是一个补救的方法:

$$\begin{aligned}\int_a^b K(t,s)f(s)\,ds &= \int_a^b K(t,s)[f(s)-f(t)]\,ds+\int_a^b K(t,s)f(t)\,ds\\ &= \int_a^b K(t,s)[f(s)-f(t)]\,ds+r(t)f(t)\end{aligned} \tag{18.3.2}$$

其中 $r(t)=\int_a^b K(t,s)\,ds$,可以解析或数值计算求得。现在如果右端第一项是奇异的,那么就可以用 Nystrom 方法。不同于(18.1.4)式,得到

$$f_i=\lambda\sum_{\substack{j=0\\ j\neq i}}^{N-1} w_j K_{ij}[f_j-f_i]+\lambda r_i f_i+g_i \tag{18.3.3}$$

有时,核完全正则化之前减法过程要重复多次,详见本节参考文献[3](建议阅读一下用不同的方法处理对角奇异性的文献)。

18.3.1 具有任意权的均匀网格上的积分

一般来说,可以找到 n 点线性积分法则,它能使函数 $f(x)$ 乘以任意权函数 $w(x)$,对任意积分区间 (a,b)

上的积分，近似为权乘以函数 $f(x)$ 的几个均匀分划值的和，比如取 $x=kh,(k+1)h,\ldots,(k+n-1)h$。推导这种积分法则的一般方法是，该积分法精确成立时，对函数 $f(x)=\text{const},x,x^2,\ldots,x^{n-1}$ 必须满足的 n 个线性方程，然后解这些方程求出系数。假定已知权函数在同一积分区域的矩，

$$W_n \equiv \frac{1}{h^n}\int_a^b x^n w(x)dx \tag{18.3.4}$$

那么，可以解析地使问题得到彻底的解决。若(通常也会这样)$b-a$ 和 h 成比例，则首因子 h^{-n} 用来使 W_n 和 h 的尺度相同。

在四点情况下，实现该方法可得出结果

$$\begin{aligned}\int_a^b w(x)f(x)dx =&\\ &\frac{1}{6}f(kh)\Big[(k+1)(k+2)(k+3)W_0-(3k^2+12k+11)W_1+3(k+2)W_2-W_3\Big]\\ &+\frac{1}{2}f([k+1]h)\Big[-k(k+2)(k+3)W_0+(3k^2+10k+6)W_1-(3k+5)W_2+W_3\Big]\\ &+\frac{1}{2}f([k+2]h)\Big[k(k+1)(k+3)W_0-(3k^2+8k+3)W_1+(3k+4)W_2-W_3\Big]\\ &+\frac{1}{6}f([k+3]h)\Big[-k(k+1)(k+2)W_0+(3k^2+6k+2)W_1-3(k+1)W_2+W_3\Big]\end{aligned} \tag{18.3.5}$$

虽然括号中的项表面上和 k^2 的尺度一致，但实际上 $O(k^2)$ 和 $O(k)$ 均被消去了。

(a,b) 的各种不同选择，使得(18.3.5)式有多种特殊化形式。较明显的选择是 $a=kh,b=(k+3)h$，在此情形下，得到一个由 Simpson 3/8 法则((4.1.5)式)推广产生的四点积分法。事实上，可以通过设置 $w(x)=1$ 来恢复这个特殊情形，此时(18.3.4)式变成

$$W_n=\frac{h}{n+1}[(k+3)^{n+1}-k^{n+1}] \tag{18.3.6}$$

(18.3.5)式中方括号里四项的每项都变得和 k 无关，(18.3.5)式实际上变为

$$\int_{kh}^{(k+3)h} f(x)dx=\frac{3h}{8}f(kh)+\frac{9h}{8}f([k+1]h)+\frac{9h}{8}f([k+2]h)+\frac{3h}{8}f([k+3]h) \tag{18.3.7}$$

再回到一般的 $w(x)$ 的情形，a 和 b 的另外一些选择也很有用。例如，可能想将 (a,b) 选取为 $([k+1]h,[k+3]h)$ 或 $([k+2]h,[k+3]h)$，只要构造一个其积分区间数不是 3 的倍数的扩展法。那么精度就不会有影响：积分将用四个值 $f(kh),\ldots,f([k+3]h)$ 来估计。更有用的是将 (a,b) 选取为 $([k+1]h,[k+2]h)$，这样就可用四点法来求集中在一个单一区间上的积分。当这些权被纳入一个扩展的公式时，它们给出了具有光滑系数的积分格式，即没有像 Simpson 法中那样进行 2,4,2,4,2 的交换(事实上，这是推导(4.1.14)时曾经用过的技巧，现在可以复习一下这些内容)

所有这些法则都和扩展 Simpson 法同阶，即 $f(x)$ 为精确的三次多项式。如果想用低阶法则，也可以类似地求得。三点公式是

$$\begin{aligned}\int_a^b w(x)f(x)dx=&\frac{1}{2}f(kh)\Big[(k+1)(k+2)W_0-(2k+3)W_1+W_2\Big]\\ &+f([k+1]h)\Big[-k(k+2)W_0+2(k+1)W_1-W_2\Big]\\ &+\frac{1}{2}f([k+2]h)\Big[k(k+1)W_0-(2k+1)W_1+W_2\Big]\end{aligned} \tag{18.3.8}$$

这里简单特殊的情形是取 $w(x)=1$，则

$$W_n=\frac{h}{n+1}[(k+2)^{n+1}-k^{n+1}] \tag{18.3.9}$$

(18.3.8)式变成了 Simpson 法，

$$\int_{kh}^{(k+2)h} f(x)dx=\frac{h}{3}f(kh)+\frac{4h}{3}f([k+1]h)+\frac{h}{3}f([k+2]h) \tag{18.3.10}$$

但是对于不是常数的权函数 $w(x)$，(18.3.8)式却给出了一个比 Simpson 低一阶的法则，因为常数值情形具有

特别的对称现象,会给它们带来很大的方便。

两点公式可简单地写为

$$\int_{kh}^{(k+1)h} w(x)f(x)dx = f(kh)[(k+1)W_0 - W_1] + f([k+1]h)[-kW_0 + W_1] \tag{18.3.11}$$

下面的程序 wwghts 是利用上述公式得到的,它返回在积分区间$(a,b)=(0,[N-1]h)$上的一个扩展的 N 点积分法。输入到 wwghts 中的是一个由用户提供的程序 kermom,调用它可求得 $w(x)$矩阵的最初四个不定积分矩,即

$$F_m(y) \equiv \int^y s^m w(s)ds \qquad m=0,1,2,3 \tag{18.3.12}$$

(下限任意,可以根据需要选取。)千万注意,当 $N<4$ 时,调用 wwghts 会返回一个比 Simpson 法低阶的法则,应该适当地组织问题以避免该情形的出现。

```
#include "nr.h"

void NR::wwghts(Vec_O_DP &wghts, const DP h,
    void kermom(Vec_O_DP &w, const DP y))
wghts[0..n-1]中构造了函数f(x)从0到(n-1)h的n点等区间积分的权重与任意一个(可能奇异)权函数w(x)的乘
积,而w(x)的不定积分矩F_n(y)从用户提供的kermom程序中获取。
{
    int j,k;
    Vec_DP wold(4);
    DP hh,hi,c,fac,a,b;

    int n=wghts.size();
    hh=h;
    hi=1.0/hh;
    for (j=0;j<n;j++) wghts[j]=0.0;
    零化权重以便累加它们。
    if (n >= 4) {                          在低端估计不定积分。
        Vec_DP wold(4),wnew(4),w(4);
        kermom(wold,0.0);                  用尽可能高的阶。
        b=0.0;                             对另一个问题,可能要改变该下限。
        for (j=0;j<n-3;j++) {              在(18.3.5)式中这称为k。
            c=j;                           设置该步的上限和下限。
            a=b;
            b=a+hh;
            if (j == n-4) b=(n-1)*hh;      最后一个区间:到结尾完全一致。
            kermom(wnew,b);
            for (fac=1.0,k=0;k<4;k++,fac*=hi)          式(18.3.4)。
                w[k]=(wnew[k]-wold[k])*fac;
            wghts[j] += (((c+1.0)*(c+2.0)*(c+3.0)*w[0]      式(18.3.5)。
                -(11.0+c*(12.0+c*3.0))*w[1]+3.0*(c+2.0)*w[2]-w[3])/6.0);
            wghts[j+1] += ((-c*(c+2.0)*(c+3.0)*w[0]
                +(6.0+c*(10.0+c*3.0))*w[1]-(3.0*c+5.0)*w[2]+w[3])*0.5);
            wghts[j+2] += ((c*(c+1.0)*(c+3.0)*w[0]
                -(3.0+c*(8.0+c*3.0))*w[1]+(3.0*c+4.0)*w[2]-w[3])*0.5);
            wghts[j+3] += ((-c*(c+1.0)*(c+2.0)*w[0]
                +(2.0+c*(6.0+c*3.0))*w[1]-3.0*(c+1.0)*w[2]+w[3])/6.0);
            for (k=0;k<4;k++) wold[k]=wnew[k]; 重置矩的下限。
        }
    } else if (n == 3) {                   低阶情形,不推荐。
        Vec_DP wold(3),wnew(3),w(3);
        kermom(wold,0.0);
        kermom(wnew,hh+hh);
        w[0]=wnew[0]-wold[0];
        w[1]=hi*(wnew[1]-wold[1]);
        w[2]=hi*hi*(wnew[2]-wold[2]);
        wghts[0]=w[0]-1.5*w[1]+0.5*w[2];
        wghts[1]=2.0*w[1]-w[2];

        wghts[2]=0.5*(w[2]-w[1]);
    } else if (n == 2) {
        Vec_DP wold(2),wnew(2),w(2);
        kermom(wold,0.0);
        kermom(wnew,hh);
        wghts[0]=wnew[0]-wold[0]-(wghts[1]=hi*(wnew[1]-wold[1]));
    }
}
```

下面给出一个如何应用 wwghts 来解奇异积分方程的例子。

18.3.2　实例:对角奇异核

作为一个特殊例子,考虑积分方程

$$f(x)+\int_0^{\pi}K(x,y)f(y)dy=\sin x \tag{18.3.13}$$

它具有一个(任意选取)很难处理的核

$$K(x,y)=\cos x\cos y\times\begin{cases}-\ln(x-y) & y<x\\ \sqrt{y-x} & y\geq x\end{cases} \tag{18.3.14}$$

对角线左侧出现对数奇异性,同时在右侧有平方根不连续。

第一步(该情形下用解析法)是对核的奇异部分求必需的积分矩,即(18.3.12)式。由于在一个固定 x 值上求这些积分,因此可选 x 作为下限。对任意确定的 y 值,则需要求的不定积分或是

$$F_m(y;x)=\int_x^y s^m(s-x)^{1/2}ds=\int_0^{y-x}(x+t)^m t^{1/2}dt \qquad \text{if } y>x \tag{18.3.15}$$

或是

$$F_m(y;x)=-\int_x^y s^m\ln(x-s)ds=\int_0^{x-y}(x-t)^m\ln t\,dt \qquad \text{if } y<x \tag{18.3.16}$$

(其中第二个等式中的变量在每种情形下都有变化。)为了用解析法求这些积分(实际上,我们用了一套符号积分包),我们将这些合成公式都组装在下述程序中。注意 w($j+1$)返回 $F_j(y;x)$。

```
#include <cmath>
#include "nr.h"
using namespace std;

extern DP x;                          在quadmx中定义。

void NR::kermom(Vec_O_DP &w, const DP y)
w[0..m-1]返回核的奇异部分的一行中前m个不定积分矩(本例中,m取为4)。输入变量y标记列,而全局变量x为
行。可将x取为积分的下限。因而积分矩完全返回到对角线左侧,或者完全返回到对角线右侧。
{
    DP d,df,clog,x2,x3,x4,y2;

    int m=w.size();
    if (y >= x) {
        d=y-x;
        df=2.0*sqrt(d)*d;
        w[0]=df/3.0;
        w[1]=df*(x/3.0+d/5.0);
        w[2]=df*((x/3.0 + 0.4*d)*x + d*d/7.0);
        w[3]=df*(((x/3.0 + 0.6*d)*x + 3.0*d*d/7.0)*x+d*d*d/9.0);
    } else {
        x3=(x2=x*x)*x;
        x4=x2*x2;
        y2=y*y;
        d=x-y;
        w[0]=d*((clog=log(d))-1.0);
        w[1] = -0.25*(3.0*x+y-2.0*clog*(x+y))*d;
        w[2]=(-11.0*x3+y*(6.0*x2+y*(3.0*x+2.0*y))
            +6.0*clog*(x3-y*y2))/18.0;
        w[3]=(-25.0*x4+y*(12.0*x3+y*(6.0*x2+y*
            (4.0*x+3.0*y)))+12.0*clog*(x4-(y2*y2)))/48.0;
    }
}
```

第二步,写一个构造积分矩阵的程序。

```
#include <cmath>
#include "nr.h"
using namespace std;

DP x;                                  与kermom通信。

void NR::quadmx(Mat_O_DP &a)
在a[0..n-1][0..n-1]中构造一个以第二类Fredholm积分方程为例的积分矩阵。该程序计算了核的非奇异部分,而积分
核的奇异部分的积分权通过调用wwghts得到。提供核奇异部分的不定积分矩阵的外部程序kermom被传输到wwghts中。
{
    const DP PI=3.14159263589793238;
    int j,k;
    DP h,xx,cx;

    int n=a.nrows();
    Vec_DP wt(n);
    h=PI/(n-1);
    for (j=0;j<n;j++) {
        x=xx=j*h;                      使x为全局变量供kermom使用。
        wwghts(wt,h,kermom);
        cx=cos(xx);                    非奇异核部分
        for (k=0;k<n;k++)              把核的所有部分加起来。
            a[j][k]=wt[k]*cx*cos(k*h);
        ++a[j][j];                     由于第二类方程,所以具有与h无关的对角部分。
    }
}
```

最后一步,对任意右端项,此处为 $\sin x$,求解线性方程组。

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include "nr.h"
using namespace std;

int main(void)    // Program fredex
该范程序显示了怎样用乘积Nystrom方法和怎样用专门针对一个特定的奇异核构造出的一个积分法则,求解第二类
Fredholm方程
{
    const int N=40;                    这里网格的大小是确定的。
    const DP PI=3.14159263589793238;
    int j;
    DP d,x;
    Vec_INT indx(N);
    Vec_DP g(N);
    Mat_DP a(N,N);

    NR::quadmx(a);                     进行积分矩阵,所有操作都在此。
    NR::ludcmp(a,indx,d);              分解矩阵。
    for (j=0;j<N;j++)                  构造右端项,这里为sin x。
        g[j]=sin(j*PI/(N-1));
    NR::lubksb(a,indx,g);              回代。
    for (j=0;j<N;j++) {                写出解。
        x=j*PI/(N-1);
        cout << fixed << setprecision(2) << setw(6) << (j+1);
        cout << setprecision(6) << setw(13) << x << setw(13) << g[j] << endl;
    }
    return 0;
}
```

当 $N=40$ 时,该程序给出的精度约在 10^{-5} 量级。尽管是高奇异性核,精度增加的速度为 N^4(Simpson 阶积分方案也应如此)。图 18.3.1 显示了求出的解,同时也画出了 N 值较小的解,可以看出,这些解相当可信。注意,尽管核在通常情形下是奇异的,但解却仍是光滑的

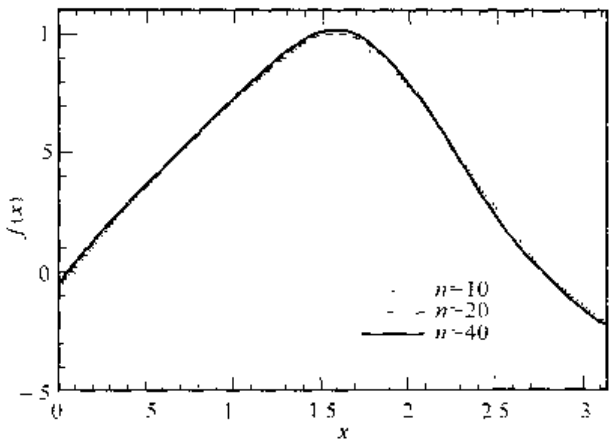


图 18.3.1　积分方程范例(18.3.14)的解。网格大小分别为 $N=10,20,40$。解的离散值通过直线连起来。实际应用中，可以将小 N 值的解作内插使得曲线更光滑

参考文献和深入阅读

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [1]

Stroud, A.H., and Secrest, D. 1966, *Gaussian Quadrature Formulas* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [3]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.). [4]

18.4　反演问题与先验信息的利用

列出一些数学上的基本性质会使得后面的讨论更方便。假定 $\mathbf{u}$ 是个"未知"向量，我们想通过一些最小化法则来确定它。令 $\mathcal{A}[\mathbf{u}]>0$ 及 $\mathcal{B}[\mathbf{u}]>0$ 为 $\mathbf{u}$ 的两个正值函数，则可以通过

$$\text{最小化：}\quad \mathcal{A}[\mathbf{u}] \qquad \text{或} \qquad \text{最小化：}\quad \mathcal{B}[\mathbf{u}] \tag{18.4.1}$$

来设法确定 $\mathbf{u}$(这些方法通常会确定出不同的 $\mathbf{u}$ 值)。另一种可能是，假定想先特别地**限制** $\mathcal{B}[\mathbf{u}]$ 的取值，比如取 b，然后使 $\mathcal{A}[\mathbf{u}]$ 最小。拉格朗日(Lagrange)乘子法给出变分

$$\frac{\delta}{\delta\mathbf{u}}\{\mathcal{A}[\mathbf{u}]+\lambda_1(\mathcal{B}[\mathbf{u}]-b)\}=\frac{\delta}{\delta\mathbf{u}}(\mathcal{A}[\mathbf{u}]+\lambda_1\mathcal{B}[\mathbf{u}])=0 \tag{18.4.2}$$

其中，λ_1 为拉格朗日乘子。注意第二个等式中没有 b，因为它和 $\mathbf{u}$ 无关。

接着，假定改变主意，限制 $\mathcal{A}[\mathbf{u}]$ 取特定的值 a，然后使 $\mathcal{B}[\mathbf{u}]$ 最小。与(18.4.2)式不同，得到

$$\frac{\delta}{\delta\mathbf{u}}\{\mathcal{B}[\mathbf{u}]+\lambda_2(\mathcal{A}[\mathbf{u}]-a)\}=\frac{\delta}{\delta\mathbf{u}}(\mathcal{B}[\mathbf{u}]+\lambda_2\mathcal{A}[\mathbf{u}])=0 \tag{18.4.3}$$

其中 λ_2 为拉格朗日乘子。(18.4.3)式乘以常数 $1/\lambda_2$，并令 $1/\lambda_2$ 等于 λ_1，我们发现这两种情形下的变化实际上是完全一样的。它们都满足同样的含一个参数的解系，如 $\mathbf{u}(\lambda_1)$。当 λ_1 从 0 变到 ∞ 时，$\mathbf{u}(\lambda_1)$ 沿着一条所谓的**权衡曲线**，在使 $\mathcal{A}$ 最小和使 $\mathcal{B}$ 最小之间变化。该曲线上的任何解都可以看做：或(i)限制 $\mathcal{B}$ 的取值，使 $\mathcal{A}$ 最小；或(ii)限制 $\mathcal{A}$ 的取值，使 $\mathcal{B}$ 最小；或(iii)使 $\mathcal{A}+\lambda_1\mathcal{B}$ 在权意义下最小。

这二个基本点与退化的最小化法则有关。在上述例子中，假定现在 $\mathcal{A}[\mathbf{u}]$ 具有特定的形式

$$\mathcal{A}[\mathbf{u}]=|\mathbf{A}\cdot\mathbf{u}-\mathbf{c}|^2 \tag{18.4.4}$$

其中 $\mathbf{A}$ 为矩阵，$\mathbf{c}$ 为向量。若 $\mathbf{A}$ 和行小于列，或 $\mathbf{A}$ 为方阵但却为退化矩阵(有一个非普通的零空间，见 2.6 节，特别是图 2.6.1)，则使 $\mathcal{A}[\mathbf{u}]$ 最小也不能得到 $\mathbf{u}$ 的惟一解(可参阅 15.4 节。注意，对于行小于列的一个"设计矩阵"$\mathbf{A}$，正规方程(15.4.10)中的矩阵 $\mathbf{A}^T\cdot\mathbf{A}$ 是退化的)。但是，如果给 $\mathcal{A}[\mathbf{u}]$ 加上 λ 与一个非退化二次型 $\mathcal{B}[\mathbf{u}]$

的乘积,比如 $\mathbf{u}\cdot\mathbf{H}\cdot\mathbf{u}$,其中 $\mathbf{H}$ 为一确定的正矩阵,则使 $\mathcal{A}[\mathbf{u}]+\lambda\mathcal{B}[\mathbf{u}]$ 最小,将得到 $\mathbf{u}$ 的惟一解(两个二次型的和仍是一个二次型,第二部分保证非退化性)。

综合以上两个点,可以得到这样的结论。当一个二次最小化法则加一个二次型限制,且两个都取正值时,只要其中之一非退化就能使整个问题适定。下面讨论反演问题。

18.4.1 零阶正则化反演问题

假定 $u(x)$ 是一些未知的或基本的物理过程(u 代表未知的和基本的),我们希望通过 N 个测量值 c_i, $i=0,1,\ldots,N-1$ 能确定 u。$u(x)$ 和 c_i 之间的关系是,每个 c_i 通过它自己的线性响应核 r_i 来测定 $u(x)$ 在某一方面的性质(希望这些性质截然不同),其测量误差为 n_i。换言之

$$c_i \equiv s_i + n_i = \int r_i(x)u(x)dx + n_i \tag{18.4.5}$$

(比较上式与(13.3.1)式和(13.3.2)式。)在线性假设的范围内,这是个相当一般的公式。c_i 可以近似地表示 $u(x)$ 在某些位置 x_i 上的值。此时 $r_i(x)$ 将具有集中在 $x=x_i$ 附近更窄或稍窄的响应的形式。或者,c_i 可以位于一个和 $u(x)$ 完全不同的函数空间里,例如测定 $u(x)$ 的不同傅里叶分量。

反演问题即是:给定 c_i 和 $r_i(x)$,也许还有误差 n_i 的一些信息,例如它们的协方矩阵

$$S_{ij} \equiv \mathbf{Covar}[n_i, n_j] \tag{18.4.6}$$

怎样才能找到一个 $u(x)$ 的最佳统计估算值,称为 $\widehat{u}(x)$?

非常明显,这是个病态问题。但无论如何,我们如何能够根据有限个离散值 c_i 来重建整个函数 $\widehat{u}(x)$ 呢?无论正式地还是非正式地,在科学研究中我们总在做这样的事。我们例行地测量"足够多的点",然后"过它们画条线"。这样做时总要进行一些假定,或是关于原函数 $u(x)$ 的性质,或是关于响应函数 $r_i(x)$ 的性质,或是二者皆有。现在的目的是,将这些假定公式化,并将研究工作最大限度地扩展到测量值和基本函数处于完全不同的函数空间的情况中去(如怎样通过一些傅里叶系数的散布来"画条曲线")。

并不真正想得到每个 x 点的函数 $\widehat{u}(x)$。只想得到 M 个离散点 x_μ, $\mu=0,1,\ldots,M-1$,其中 M 充分大,且 x_μ 充分均匀地分布,致使 $u(x)$ 或 $r_i(x)$ 在任意 x_μ 和 $x_{\mu+1}$ 之间的变化都不大(此处及下文中均用希腊字母,例如 μ,来表示基本处理空间中的值,而用罗马字,比如 i 来表示直接观察到的值)。对于 x_μ 这样稠密的一个集合,可用一个积分

$$c_i = \sum_\mu R_{i\mu}u(x_\mu) + n_i \tag{18.4.7}$$

来代替(18.4.5)式,其中 $N\times M$ 矩阵 $\mathbf{R}$ 分量为

$$R_{i\mu} \equiv r_i(x_\mu)(x_{\mu+1}-x_{\mu-1})/2 \tag{18.4.8}$$

(或任何其他简单的积分,用哪种积分并不重要。)实际应用中,认为(18.4.5)式和(18.4.7)式等价。

怎样从如(18.4.7)式这样的方程组中解出未知的 $u(x_\mu)$ 呢?下面的方法并不好,但却包含着一些正确的思路:构造 χ^2 来衡量模型 $\widehat{u}(x)$ 和测量数据的符合程度,

$$\begin{aligned}\chi^2 &= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left[c_i - \sum_{\mu=0}^{M-1}R_{i\mu}\widehat{u}(x_\mu)\right]S_{ij}^{-1}\left[c_j - \sum_{\mu=0}^{M-1}R_{j\mu}\widehat{u}(x_\mu)\right] \\ &\approx \sum_{i=0}^{N-1}\left[\frac{c_i - \sum_{\mu=0}^{M-1}R_{i\mu}\widehat{u}(x_\mu)}{\sigma_i}\right]^2\end{aligned} \tag{18.4.9}$$

(与(15.1.5)式比较。)其中 $\mathbf{S}^{-1}$ 是协方差矩阵的逆,如果不在对角线上的协方差可以忽略,则约等号成立,$\sigma_i \equiv (\mathrm{Covar}[i,i])^{1/2}$。

现在可以用15.4节中奇异分解法(SVD)来找使(18.4.9)式最小的向量 $\widehat{\mathbf{u}}$ 了。不要试图去用正规方程的方法,前面已经讨论过,由于 M 比 N 大,它们将是奇异的。因而SVD过程必定会找到很多零奇异值,表明解具有很高的不惟一性。在无穷多退化解中(大部分解对任意大的 $\widehat{u}(x_\mu)$ 具有不好的性质),SVD会选出一个在下述意义下的最小 $|\widehat{\mathbf{u}}|$ 值。

$$\sum_\mu [\widehat{u}(x_\mu)]^2 \qquad \text{最小化} \tag{18.4.10}$$

见图 2.6.1。这个解常称为**基本解**。它是所谓**零阶正则化**的一种极限情形，对应着λ较小时，对两个正值函数的和取最小值

$$\text{最小化}\ :\quad \chi^2[\hat{\mathbf{u}}] + \lambda(\hat{\mathbf{u}}\cdot\hat{\mathbf{u}}) \tag{18.4.11}$$

下面，将讨论如何求这样的最小值及一些更一般的情形，而不特别采用 SVD。

如果用一个非无穷小的λ值，通过(18.4.11)式来决定$\hat{\mathbf{u}}$，将会发生什么情况呢？首先注意，如果 $M \gg N$(未知量数远大于方程数)，则 $\mathbf{u}$ 常有足够的自由度使 χ^2 ((18.4.9)式)即使不为 0，也会相当不切实际的小。用 15.1 节中的话来说，自由度数 $\nu = N - M$(当 ν 很大时约等于 χ^2 的期望值)，可以减小到零(甚至无意义，超出范围)。但我们知道，对于真实的基础函数，由于没有可调整的参数，自由度数与 χ^2 的期望值大约应该满足 $\nu \approx N$。

增大λ会使解远离 χ^2 最小的情形，而趋向于使$\hat{\mathbf{u}}\cdot\hat{\mathbf{u}}$最小。根据上文对基本点的讨论，可以视这种情形为，限制 χ^2 取某个非零常值后使$\hat{\mathbf{u}}\cdot\hat{\mathbf{u}}$最小。事实上，一个常见的取法是找到满足 $\chi^2 = N$ 的λ值，即通过 χ^2 的合适取值使问题进一步正则化。求出的 $\hat{u}(x)$称为**零阶正则化反演问题的解**。

N 值实际上代表所有从均值为 N，标准差为$(2N)^{1/2}$(χ^2 的渐近分布)的高斯分布中得到的值。可分别试试λ的两个可能值，一个使 $\chi^2 = N + (2N)^{1/2}$，另一个使 $\chi^2 = N - (2N)^{1/2}$。

虽然还有更好的方法，但零阶正则化反映了反演问题理论中经常用到的大部分基本思想。一般有两个正函数，称为$\mathcal{A}$和$\mathcal{B}$。第一个正函数$\mathcal{A}$，是测量模型与数据(例如 χ^2)的某种一致性；或者是描述解和基本函数间映射的"鲜明程度"的相关量。当$\mathcal{A}$本身取最小值时，一致性或鲜明性变得很好(通常情形下不可能太好)，但解变得不稳定，剧烈地振荡；或者在另一方面极不现实地反映了由$\mathcal{A}$单独定义一个高退化法的最小值问题。

$\mathcal{B}$正是因此才引入的。它是一个对解的"光滑性"的测量标准，或者有时是一个对应于数据变动使解的稳定性参数化的相关量，或者有时是一个反映解的可能形式的先验判断的量，$\mathcal{B}$被称为**稳定化函数**或**正则化算子**。不论哪种情形，$\mathcal{B}$取最小值，均会给出一个"光滑的"、"稳定的"或"可能的"解——与测量数据毫不相关。

反演理论中惟一的中心思想就是规定

$$\text{最小化}\ :\quad \mathcal{A} + \lambda\mathcal{B} \tag{18.4.12}$$

其中λ沿所谓权衡曲线(见图 18.4.1)的变化范围是 $0 < \lambda < \infty$，然后，依据这样或那样的规则停在一个"最佳"λ值上，这可以从相当客观(比如使 $\chi^2 = N$)变到完全主观。不同的成功方法(将讨论其中的几种)的区别表现在：$\mathcal{A}$和$\mathcal{B}$选法的不同；规定式(18.4.12)是满足线性还是非线性方程；λ最终值的多种可能选择方法以及应用到计算机二维问题(如图像处理)的广泛程度的不同等。

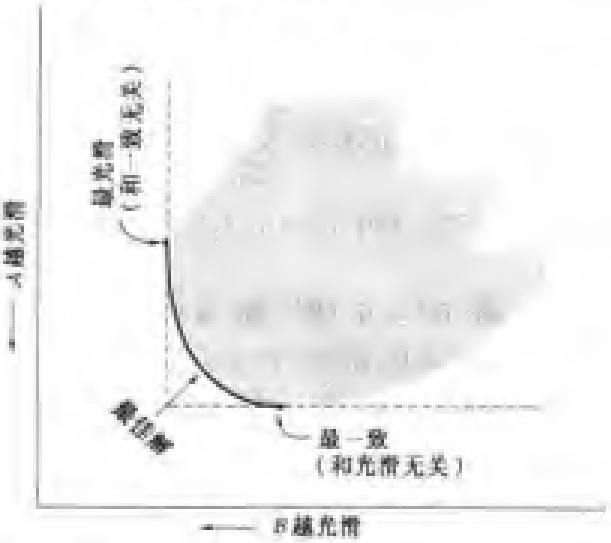


图 18.4.1　几乎所有的反演问题的方法都涉及两个最优化之间的一个交替换位：数据和解之间的一致性，实解和估算解之间映射的"鲜明程度"(此处用$\mathcal{A}$表示)，与解的光滑性和稳定性(此处用$\mathcal{B}$表示)。图中阴影区表示所有可能的解，那些位于边界上的曲线，是连接$\mathcal{A}$不受限制时的最小值和$\mathcal{B}$不受限制时的最小值的解，它们是"最佳"的解，从某种意义上说，任意其他解被控制在至少有一个解位于曲线上

区别还在于它们包含的(或它们的提倡者的)哲学思想的不同。迄今为止，一直没有提及 Bayes(贝叶斯)方法。要掩盖下述事实很难，而且我们也不想掩盖，即$\mathcal{B}$与某些先验期望，或是先验知识，或是先验解有关，$\mathcal{A}$则与某些后验知识有关。常数λ决定着二者之间一种微妙的协调关系。有些反演方法中 Bayes 的痕迹较重，但我们认为这完全是历史造成的。一个只注重方程的实际解法而不管它所包含的哲学思想的人，是很难区分 Bayes 方法与所谓的经验方法的。

下面三节讨论反演问题的三种不同方法，它们在不同的领域都有着相当成功的应用。三种方法都在我们概括出的一般框架内，但每种方法的细节和实施却有很大的区别。

参考文献和深入阅读

Craig, I.J.D., and Brown, J.C. 1986, *Inverse Problems in Astronomy* (Bristol, U.K.: Adam Hilger).

Twomey, S. 1977, *Introduction to the Mathematics of Inversion in Remote Sensing and Indirect Measurements* (Amsterdam: Elsevier).

Tikhonov, A.N., and Arsenin, V.Y. 1977, *Solutions of Ill-Posed Problems* (New York: Wiley).

Tikhonov, A.N., and Goncharsky, A.V. (eds.) 1987, *Ill-Posed Problems in the Natural Sciences* (Moscow: MIR).

Parker, R.L. 1977, *Annual Review of Earth and Planetary Science*, vol. 5, pp. 35–64.

Frieden, B.R. 1975, in *Picture Processing and Digital Filtering*, T.S. Huang, ed. (New York: Springer-Verlag).

Tarantola, A. 1987, *Inverse Problem Theory* (Amsterdam: Elsevier).

Baumeister, J. 1987, *Stable Solution of Inverse Problems* (Braunschweig, Germany: Friedr. Vieweg & Sohn) [mathematically oriented].

Titterington, D.M. 1985, *Astronomy and Astrophysics*, vol. 144, pp. 381–387.

Jeffrey, W., and Rosner, R. 1986, *Astrophysical Journal*, vol. 310, pp. 463–472.

18.5 线性正则化方法

线性正则化也称为 **Phillips-Twomey 方法**[1,2]、**约束线性反演方法**[3]、**正则化方法**[4]或 **Tikhonov-Miller 正则化方法**[5~7](也可能还有其他称法，因为它非常明显是个好方法)。该方法最简单的形式是零阶正则化(前面(18.4.11)式的一个直接推广)。和以前一样，函数$\mathcal{A}$取为χ^2偏差，见(18.4.9)式，但$\mathcal{B}$由更复杂的从一阶或高阶导数导出的度量光滑性的函数来代替。

例如，假定先验信息是：一个可能的$u(x)$和一个常数没有太多的不同，则一个求最小值的较合理函数应该是

$$\mathcal{B} \propto \int [\hat{u}'(x)]^2 dx \propto \sum_{\mu=0}^{M-2} [\hat{u}_\mu - \hat{u}_{\mu+1}]^2 \tag{18.5.1}$$

因为它非负且仅当$\hat{u}(x)$为常数时才等于零。其中$\hat{u}_\mu \equiv \hat{u}(x_\mu)$，并且第二个等式(比例式)中假定$x_\mu$是均匀分划的。可将$\mathcal{B}$的第二种形式写做

$$\mathcal{B} = |\mathbf{B}\cdot\hat{\mathbf{u}}|^2 = \hat{\mathbf{u}}\cdot(\mathbf{B}^T\cdot\mathbf{B})\cdot\hat{\mathbf{u}} \equiv \hat{\mathbf{u}}\cdot\mathbf{H}\cdot\hat{\mathbf{u}} \tag{18.5.2}$$

其中$\hat{\mathbf{u}}$是以$\hat{u}_\mu(\mu=0,\ldots,M-1)$为分量的向量，$\mathbf{B}$是一个$(M-1)\times M$的第一差分矩阵

$$\mathbf{B} = \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{18.5.3}$$

$\mathbf{H}$是$M\times M$矩阵

$$\mathbf{H}=\mathbf{B}^T\cdot\mathbf{B}=\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{18.5.4}$$

注意 $\mathbf{B}$ 的行数比列小 1。其次，对称的 $\mathbf{H}$ 是退化的，它确实有一个零特征值对应着一个常数函数的值，任意一个常数函数都会使 $\mathcal{B}$ 严格为零。

和 15.4 节一样，如果写

$$A_{i\mu}\equiv R_{i\mu}/\sigma_i \qquad b_i\equiv c_i/\sigma_i \tag{18.5.5}$$

那么应用(18.4.9)式，最小化原理(18.4.12)为

$$\text{最小化}\ :\quad \mathcal{A}+\lambda\mathcal{B}=|\mathbf{A}\cdot\hat{\mathbf{u}}-\mathbf{b}|^2+\lambda\hat{\mathbf{u}}\cdot\mathbf{H}\cdot\hat{\mathbf{u}} \tag{18.5.6}$$

这可以容易地化简成一个线性**正规方程组**，仍和 15.4 节中一样：解的分量 $\hat{u}_\mu$ 满足 M 个未知量的 M 个方程，

$$\sum_\rho\left[\left(\sum_i A_{i\mu}A_{i\rho}\right)+\lambda H_{\mu\rho}\right]\hat{u}_\rho=\sum_i A_{i\mu}b_i \qquad \mu=0,1,\ldots,M-1 \tag{18.5.7}$$

或用向量形式，

$$(\mathbf{A}^T\cdot\mathbf{A}+\lambda\mathbf{H})\cdot\hat{\mathbf{u}}=\mathbf{A}^T\cdot\mathbf{b} \tag{18.5.8}$$

(18.5.7)式或(18.5.8)式可用第 2 章中介绍的标准方法求解，例如 LU 分解法。这时有关正规方程组为病态的注意事项已不再适用，因为λ项的全部目的是为了消除相同的病态。但是注意，由于λ项没有选择一个合适的常数值，所以它本身是病态的。只能希望用户的数据能弥补这一点。

虽然矩阵($\mathbf{A}^T\cdot\mathbf{A}+\lambda\mathbf{H}$)求逆通常并不是求解$\hat{\mathbf{u}}$的最好方法，我们仍将(18.5.8)式的解先在形式上写成

$$\hat{\mathbf{u}}=\left(\frac{1}{\mathbf{A}^T\cdot\mathbf{A}+\lambda\mathbf{H}}\cdot\mathbf{A}^T\cdot\mathbf{A}\right)\mathbf{A}^{-1}\cdot\mathbf{b} \qquad \text{(只在形式上成立！)} \tag{18.5.9}$$

其中插进了形式上为 $\mathbf{A}\cdot\mathbf{A}^{-1}$ 的恒等矩阵。因为我们不但将矩阵的逆想像性地写为一个分母，而且逆矩阵 $\mathbf{A}^{-1}$ 通常并不存在，因此上式只是在形式上成立。但是将(18.5.9)式与(13.3.6)式进行最佳或维纳滤波，或者与(13.6.6)式进行一般的线性预测相比较，都会使我们受到很大的启发。可以看出，$\mathbf{A}^T\cdot\mathbf{A}$ 和 S^2 的作用相同，代表信号功率或自相关性，而λ $\mathbf{H}$ 和 N^2 所起作用相同，代表噪声功率或自相关性。(18.5.9)式括号中的项有些类似最佳滤波器，其作用是：当 $\mathbf{A}^T\cdot\mathbf{A}$ 足够大时，使病态的逆 $\mathbf{A}^{-1}\cdot\mathbf{b}$ 不修正地通过；而当 $\mathbf{A}^T\cdot\mathbf{A}$ 较小时，则将它抑制掉。

上述 $\mathbf{B}$ 和 $\mathbf{H}$ 的选法仅仅是一个明显的导数序列中最简单的情形。如果你的先验信息认为线性函数是对 $u(x)$ 的很好近似，则使下式最小：

$$\mathcal{B}\propto\int[\hat{u}''(x)]^2dx\propto\sum_{\mu=0}^{M-3}[-\hat{u}_\mu+2\hat{u}_{\mu+1}-\hat{u}_{\mu+2}]^2 \tag{18.5.10}$$

隐含着

$$\mathbf{B}=\begin{pmatrix} -1 & 2 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 2 & -1 \end{pmatrix} \tag{18.5.11}$$

以及

$$\mathbf{H}=\mathbf{B}^T\cdot\mathbf{B}=\begin{pmatrix} 1 & -2 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -2 & 5 & -4 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -4 & 6 & -4 & 1 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 1 & -4 & 6 & -4 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 & -4 & 6 & -4 & 1 \\ 0 & \cdots & 0 & 0 & 0 & 1 & -4 & 5 & -2 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 1 & -2 & 1 \end{pmatrix} \tag{18.5.12}$$

上式中的 $\mathbf{H}$ 有两个零特征值，对应于一个线性函数的两个未知确定参数。

如果先验信息是二次函数会更好，则将使下式最小：

$$\mathcal{B}\propto\int[\widehat{u}''(x)]^2dx\propto\sum_{\mu=0}^{M-4}[-\widehat{u}_\mu+3\widehat{u}_{\mu+1}-3\widehat{u}_{\mu+2}+\widehat{u}_{\mu+3}]^2 \tag{18.5.13}$$

且

$$\mathbf{B}=\begin{pmatrix} -1 & 3 & -3 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 3 & -3 & 1 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & -1 & 3 & -3 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 3 & -3 & 1 \end{pmatrix} \tag{18.5.14}$$

现在

$$\mathbf{H}=\begin{pmatrix} 1 & -3 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -3 & 10 & -12 & 6 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 3 & -12 & 19 & -15 & 6 & -1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 & \cdots & 0 \\ \vdots & & & & & \ddots & & & & & \vdots \\ 0 & \cdots & 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 \\ 0 & \cdots & 0 & 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 6 & -15 & 19 & -12 & 3 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 6 & -12 & 10 & -3 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 3 & -3 & 1 \end{pmatrix} \tag{18.5.15}$$

(将三次方及更高次方的计算留给有兴趣的读者。)

注意，如果愿意的话，也可以用"向微分方程近似"的方法来正则化，只需令 $\mathbf{B}$ 为几个适当的有限分算子(其系数可以依赖于 x)的和，并计算 $\mathbf{H}=\mathbf{B}^T\cdot\mathbf{B}$。可以不知道边界条件的值，因为前面已讲过，$\mathbf{B}$ 的行可以小于列，这样就极有希望有数据来确定它们。当然，如果知道某些边界条件，也可将其加入 $\mathbf{B}$ 中。

对于上述各式中所有的比例性符号，读者也许不知道应该取λ的什么实际值首先进行试验。可以先试一个简单的技巧，令

$$\lambda=\mathrm{Tr}(\mathbf{A}^T\cdot\mathbf{A})/\mathrm{Tr}(\mathbf{H}) \tag{18.5.16}$$

其中 Tr 是矩阵的迹(对角元素之和)。该选法使最小值的两部分有类似的权，因此就可以从这个λ开始对它们进行调整。

到于λ的"正确"值是什么，假设已知误差，而且有足够的精度，一个客观的标准便是使 χ^2 (即 $|\mathbf{A}\cdot\widehat{\mathbf{u}}-\mathbf{b}|^2$)等于测量次数 N。上文曾提过两个可以接受的孪生选法 $N\pm(2N)^{1/2}$。一个主观的准则是，根据对先验和后验信息的相对依赖程度，选取在 $0<\lambda<\infty$ 范围内的任意值(人们实际上常常是这样做的)。

18.5.1　二维问题和迭代方法

迄今为止，我们的符号一直是描述一维问题的，例如 $\hat{u}(x)$ 或 $\hat{u}_\mu = \hat{u}(x_\mu)$。不过，所有的讨论都很容易推广到估计一个二维未知量集合 $\hat{u}_{\mu k}(\mu = 0, \ldots, M-1; k = 0, \ldots, K-1)$ 的问题，例如，其中 $\hat{u}_{\mu k}$ 对应于一幅测量图像的像素强度。在此情形下，要解的仍然是(18.5.8)式。

在图像处理中，一幅"原始"或"受干扰"的测量图像的输入像素数，通常与经过处理后输出的"已知去噪声"的像素数相同，因而矩阵 $\mathbf{R}$ 和 $\mathbf{A}$((18.5.5)式)都是方阵，且均为 $MK \times MK$ 矩阵。$\mathbf{A}$ 一般很大，不能表示成满阵，但它通常是：(i)稀疏的，其系数将一个原像素(i,j)变模糊为$(i \pm$ 小量$, j \pm$ 小量$)$；或者是(ii)平移不变性 $A_{(i,j),(\mu,\nu)} = A(i-\mu, j-\nu)$。这两种情形都使问题极容易处理。

在平移不变情形，很明显应选用快速傅里叶变换(FFT)方法。原函数和测量值之间的一般线性关系，现在就成了类似(13.1.1)式的一个离散卷积。如果用 $\mathbf{k}$ 表示一个二维波向量，则二维 FFT 将下述变换对之间相互转换

$$A(i-\mu, j-\nu) \iff \widetilde{\mathbf{A}}(\mathbf{k}) \qquad b_{(i,j)} \iff \widetilde{b}(\mathbf{k}) \qquad \hat{u}_{(i,j)} \iff \widetilde{u}(\mathbf{k}) \tag{18.5.17}$$

还需要一个正则化或光滑算子 $\mathbf{B}$ 及由 $\mathbf{B}$ 导出的 $\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B}$。$\mathbf{B}$ 的一个流行选法是，拉普拉斯(Laplace)算子的五点有限差分近似，即是每点的值与它笛卡儿坐标四个邻域点的平均值之差。在傅里叶空间，该选法意味着

$$\begin{aligned} \widetilde{B}(\mathbf{k}) &\propto \sin^2(\pi k_1/M)\sin^2(\pi k_2/K) \\ \widetilde{H}(\mathbf{k}) &\propto \sin^4(\pi k_1/M)\sin^4(\pi k_2/K) \end{aligned} \tag{18.5.18}$$

在傅里叶空间，(18.5.7)式只是个代数方程，其解为

$$\widetilde{u}(\mathbf{k}) = \frac{\widetilde{A}^*(\mathbf{k})\widetilde{b}(\mathbf{k})}{|\widetilde{A}(\mathbf{k})|^2 + \lambda\widetilde{H}(\mathbf{k})} \tag{18.5.19}$$

其中星号表示复共轭。对于实数据，可利用 12.5 节中的 FFT 程序。

现在，再看一下 $\mathbf{A}$ 不是平移不变的情形。现在直接解(18.5.8)式是不可能的，因为矩阵 $\mathbf{A}$ 太大。需要某种迭代方案。

方法之一是，完全用 10.6 节中的共轭梯度法来找 $\mathcal{A} + \lambda\mathcal{B}$ 的最小值，见(18.5.6)式。在第 10 章的所有方法中，共轭梯度法是惟一最佳的方法，这是因为：(i)它不用存储一个黑塞矩阵，该矩阵在此处是求不出来的；(ii)它必须利用梯度信息，而梯度很容易求：(18.5.6)式的梯度是

$$\nabla(\mathcal{A} + \lambda\mathcal{B}) = 2[(\mathbf{A}^T \cdot \mathbf{A} + \lambda\mathbf{H}) \cdot \hat{\mathbf{u}} - \mathbf{A}^T \cdot \mathbf{b}] \tag{18.5.20}$$

(参看(18.5.8)式。)函数和梯度的估算当然都要利用 $\mathbf{A}$ 为稀疏矩阵的性质，例如通过 2.7 节中的sprsax和sprstx程序。在 18.7 节中，将要在(非线性)最大熵方法的基础上，进一步讨论共轭梯度法。其中一些讨论也适用于此。

除共轭梯度法外，可采用不太复杂的快速下降法(参见 10.6 节)，有时它也能得到很有效的结果，特别是当它与凸集投影联合起来使用时很有效(见下文)。如果 k 次迭代后的解用 $\hat{\mathbf{u}}^{(k)}$ 表示，则经过 $k+1$ 次迭代，得到

$$\hat{\mathbf{u}}^{(k+1)} = [\mathbf{1} - \epsilon(\mathbf{A}^T \cdot \mathbf{A} + \lambda\mathbf{H})] \cdot \hat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T \cdot \mathbf{b} \tag{18.5.21}$$

上式中的 ϵ 是一个反映在下降梯度方向移动大小的参数。当 ϵ 足够小时上述方法收敛，特别是当 ϵ 满足

$$0 < \epsilon < \frac{2}{(\mathbf{A}^T \cdot \mathbf{A} + \lambda\mathbf{H})\text{的最大特征值}} \tag{18.5.22}$$

寻找 ϵ 的最佳值或最佳序列有很多复杂的方法，见本节参考文献[7]，或者也可以采用实验方法，对(18.5.6)式进行估算以确保方法的实施是沿下降方向的。

在图像处理问题中，成功与否在一定程度是主观的，因此，迭代式(18.5.21)有时在远未达到收敛前，便会使图像效果有显著的改进。正因为如此，该方法虽然在数学意义上的收敛极其缓慢，但却被大量地应用。实际上，(18.5.21)式可以和 $\mathbf{H} = 0$ 一起使用，此时解完全不正则，而且充分收敛时还会产生严重的错误。这种方法

称为 **Van Cittert 方法**,可追溯到20世纪30年代。迭代至1000次的量级都是常见的(参考本节参考文献[7])。

18.5.2 确定性约束:凸集投影

一些可能的原函数(或图像)集合$\{\hat{\mathbf{u}}\}$,如果该集合中任意两个元素$\hat{\mathbf{u}}_a$和$\hat{\mathbf{u}}_b$的所有线性插值组合

$$(1-\eta)\hat{\mathbf{u}}_a+\eta\hat{\mathbf{u}}_b \qquad 0\leqslant\eta\leqslant 1 \tag{18.5.23}$$

仍在该集合中,则称该集合为凸的。反演问题中,人们想在解$\hat{\mathbf{u}}$上强制许多**确定性的约束**,实际上是定义了凸集,例如:

- 正值性
- 紧支集(即确定区域外取零值)
- 已知边界(即 $u_L(x)\leqslant\hat{u}(x)\leqslant u_U(x)$,对确定的函数 u_L 和 u_U)

(在最后一种情形,边界可能和初始估计值及误差规定有关,比如 $\hat{u}_0(x)\pm\gamma\,\sigma(x)$,其中$\gamma$是1或2。)注意,这些以及其他类似的限制可加在图像空间或傅里叶变换空间,也可(实际上)加在$\hat{\mathbf{u}}$的任意线性变换的空间。

假定 C_i 是一个凸集,那$\mathcal{P}_i$称为到该集的**非扩张投影算子**,如果:(i)$\mathcal{P}_i$将任意已在C_i中的$\hat{\mathbf{u}}$保留不变,(ii)$\mathcal{P}_i$将C_i外的任意点$\hat{\mathbf{u}}$在下述意义下映射到C_i中和$\hat{\mathbf{u}}$距离最近的一个元素,即

$$|\mathcal{P}_i\hat{\mathbf{u}}-\hat{\mathbf{u}}|\leqslant|\hat{\mathbf{u}}_a-\hat{\mathbf{u}}| \quad \text{for all } \hat{\mathbf{u}}_a \text{ in } C_i \tag{18.5.24}$$

该定义虽然看来复杂,实际例子却很简单。正$\hat{\mathbf{u}}$集的一个非扩张投影算子,"将$\hat{\mathbf{u}}$的所有负分量映为零"以$u_L(x)\leqslant\hat{u}(x)\leqslant u_U(x)$为边界的$\hat{u}(x)$集的一非扩张投影算子,"将所有小于下界的值映为下界,而将所有大于上界的值映为上界"。具有紧支集的函数集的一个非扩张投影算子,"将所有支集区域外的值映为零"。

这些定义对下述著名定理十分有用:令C为m个凸集$C_1,C_2,\ldots,C_m$的交集,则迭代

$$\hat{\mathbf{u}}^{(k+1)}=(\mathcal{P}_1\mathcal{P}_2\cdots\mathcal{P}_m)\hat{\mathbf{u}}^{(k)} \tag{18.5.25}$$

当$k\to\infty$时,将从所有起始点收敛到C中,而且,如果C为空集(无交集),那么这个迭代没有极限点。该定理的实际应用被称为**到凸集的投影方法**,或有时简写为POCS[7]。

POCS定理的一般形式是用一组$\mathcal{T}_i$来代替$\mathcal{P}_i$,

$$\mathcal{T}_i\equiv\mathbf{1}+\beta_i(\mathcal{P}_i-\mathbf{1}) \qquad 0<\beta_i<2 \tag{18.5.26}$$

适当选取β_i集可以加快到交集C的收敛速度。

有些反演问题,只通过迭代式(18.5.25)就可完全解决。例如,在天文成像及X射线衍射工作中,经常遇到的问题是要求恢复图像,但只给出图像傅里叶变换的**模**(相当于光谱强度或自相关值)而不给出**相**。这时,就可以利用下述三个凸集:其傅里叶变换在确定的误差范围内有确定模的所有图像的集合;所有正图像的集合;某特定区域外强度为零的所有图像的集合。在这种情形下,POCS迭代(18.5.25)在这三个集合之间循环,轮流施加每种约束。每当施加傅里叶约束时,就用FFT将其映入映出傅里叶空间。

POCS在空间域和傅里叶域交替施加约束的特定应用被称为 **Gerchberg-Saxton 算法**[8]。该算法虽然是非扩张的,而且实际上也常常是收敛的,但是还没有证明在所有情形下都收敛[9]。在上面提到的相恢复问题中,该算法在取得突然的、明显的改进之前,常常有许多次迭代都"粘滞"在一个平衡阶段。有时,需要进行多达10^4到10^5次迭代(对于"非沾滞"过程,见本节参考文献[10])。虽然我们相信有一定复杂度的二维图像的解是惟一的,但有关解的惟一性问题研究得还不十分清楚。

通过投影算子,确定性约束可以合并到线性正则化迭代方法中。特别地,改变各项的组合,可将迭代(18.5.21)写为

$$\hat{\mathbf{u}}^{(k+1)}=(\mathbf{1}-\epsilon\lambda\mathbf{H})\cdot\hat{\mathbf{u}}^{(k)}+\epsilon\mathbf{A}^T\cdot(\mathbf{b}-\mathbf{A}\cdot\hat{\mathbf{u}}^{(k)}) \tag{18.5.27}$$

如果迭代在每步都加上投影算子

$$\hat{\mathbf{u}}^{(k+1)}=(\mathcal{P}_1\mathcal{P}_2\cdots\mathcal{P}_m)[(\mathbf{1}-\epsilon\lambda\mathbf{H})\cdot\hat{\mathbf{u}}^{(k)}+\epsilon\mathbf{A}^T\cdot(\mathbf{b}-\mathbf{A}\cdot\hat{\mathbf{u}}^{(k)})] \tag{18.5.28}$$

(或用(18.5.26)式的算子$\mathcal{T}_i$代替$\mathcal{P}_i$。)则我们可以看到收敛条件(18.5.22)并未改变,而且在施加期望的非线性约束后,迭代将收敛到使二次函数(18.5.6)最小。对于更复杂的情形以及和迭代有关的更快的收敛,参见本节参考文献[7]。

参考文献和深入阅读

Phillips, D.L. 1962, *Journal of the Association for Computing Machinery*, vol. 9, pp. 84–97. [1]

Twomey, S. 1963, *Journal of the Association for Computing Machinery*, vol. 10, pp. 97–101. [2]

Twomey, S. 1977, *Introduction to the Mathematics of Inversion in Remote Sensing and Indirect Measurements* (Amsterdam: Elsevier). [3]

Craig, I.J.D., and Brown, J.C. 1986, *Inverse Problems in Astronomy* (Bristol, U.K.: Adam Hilger). [4]

Tikhonov, A.N., and Arsenin, V.Y. 1977, *Solutions of Ill-Posed Problems* (New York: Wiley). [5]

Tikhonov, A.N., and Goncharsky, A.V. (eds.) 1987, *Ill-Posed Problems in the Natural Sciences* (Moscow: MIR).

Miller, K. 1970, *SIAM Journal on Mathematical Analysis*, vol. 1, pp. 52–74. [6]

Schafer, R.W., Mersereau, R.M., and Richards, M.A. 1981, *Proceedings of the IEEE*, vol. 69, pp. 432–450.

Biemond, J., Lagendijk, R.L., and Mersereau, R.M. 1990, *Proceedings of the IEEE*, vol. 78, pp. 856–883. [7]

Gerchberg, R.W., and Saxton, W.O. 1972, *Optik*, vol. 35, pp. 237–246. [8]

Fienup, J.R. 1982, *Applied Optics*, vol. 15, pp. 2758–2769. [9]

Fienup, J.R., and Wackerman, C.C. 1986, *Journal of the Optical Society of America A*, vol. 3, pp. 1897–1907. [10]

18.6　Backus-Gilbert 方法

Backus-Gilbert 方法[1,2](摘要见[3~4])与其他正则化方法的区别在于，它的函数$\mathcal{A}$和$\mathcal{B}$具有不同的性质。对于$\mathcal{B}$，该方法力图使解 $\widehat{u}(x)$的稳定性最好，而不是像第一种情形那样注重光滑性。即

$$\mathcal{B} \equiv \mathrm{Var}[\widehat{u}(x)] \tag{18.6.1}$$

是一个用来度量数据在测量误差范围内变化时解的变化程度。注意，该方差不是 $\widehat{u}(x)$与真实 $u(x)$之间预期的偏差(这个偏差由$\mathcal{A}$来约束)，而是在进行多次重复实验时，测量估计值 $\widehat{u}(x)$的预期的实验和实验之间的分散程度。

对于$\mathcal{A}$，Backus-Gilbert 方法着眼于解 $\widehat{u}(x)$与真实函数 $u(x)$之间的关系，力图使二者之间的映射在无差错的数据范围内，尽可能地接近于恒等映射。该方法是线性的，因而 $\widehat{u}(x)$和 $u(x)$之间的关系可以写为

$$\widehat{u}(x) = \int \widehat{\delta}(x, x')u(x')dx' \tag{18.6.2}$$

其中$\widehat{\delta}(x, x')$是所谓的**分辨率函数**或**平均核**。Backus-Gilbert 方法力图使$\widehat{\delta}$的宽度或散布度最小(即使分辨率最大)，而$\mathcal{A}$被选做衡量散布度的某个正数。

虽然Backus-Gilbert的哲学思想与Phillips-Twomey的哲学思想，以及其相关联的方法有很大的差别，但在实际运用中这一差别比我们想像的要小得多。一个稳定解几乎毫无例外的是光滑的；一个非正则化解产生的剧烈的、不稳定的振动对数据的小变化常常十分敏感。同样，使 $\widehat{u}(x)$接近于 $u(x)$将肯定使无差错的数据与模型一致。因而$\mathcal{A}$和$\mathcal{B}$所起的作用与前两节相应的所起的作用极其相似。

Backus-Gilbert 公式的主要优点是，它能很好地控制它所注重测量的那些项的性质，例如稳定性和分辨率。而且在处理任何实际数据之前，λ(与它通常的作用一样，用来协调$\mathcal{A}$与$\mathcal{B}$)都有常规的取法，或至少很容易选取。因而避免了选取一个λ所产生的困难及其潜在的主观偏见。在需要进行数据转换的实验的设计与预测之类问题中，常可考虑选用 Backus-Gilbert 方法。

下面来讨论工作原理。从(18.4.5)式开始，

$$c_i \equiv s_i + n_i = \int r_i(x)u(x)dx + n_i \tag{18.6.3}$$

并且从一开始就建立线性关系，找一组**反演响应核** $q_i(x)$满足

$$\widehat{u}(x) = \sum_i q_i(x)c_i \tag{18.6.4}$$

并期望能用它来估计 $u(x)$。定义每个数据点的响应核的积分也很有用，

$$R_i \equiv \int r_i(x)dx \tag{18.6.5}$$

将(18.6.4)式代入(18.6.3)式，并与(18.6.2)式做比较，发现

$$\widehat{\delta}(x,x') = \sum_i q_i(x) r_i(x') \tag{18.6.6}$$

能够要求对每个 x 使平均核具有单位面积，从而得出

$$1 = \int \widehat{\delta}(x,x')dx' = \sum_i q_i(x) \int r_i(x')dx' = \sum_i q_i(x) R_i \equiv \mathbf{q}(x)\cdot\mathbf{R} \tag{18.6.7}$$

其中 $\mathbf{q}(x)$ 和 $\mathbf{R}$ 都是具有 N 个分量的向量，N 为测量数。

误差的标准传播及(18.6.1)式给出

$$\mathcal{B} = \mathrm{Var}[\widehat{u}(x)] = \sum_i \sum_j q_i(x) S_{ij} q_j(x) = \mathbf{q}(x)\cdot\mathbf{S}\cdot\mathbf{q}(x) \tag{18.6.8}$$

其中 S_{ij} 是协方差矩阵((18.4.6)式)。如果非对角线的协方差可以忽略(相当于 c_i 的误差是独立的)，则 $S_{ij} = \delta_{ij}\sigma_i^2$ 是对角矩阵。

现在，需要对每个 x 取值，定义一个 $\widehat{\delta}(x,x')$ 的宽度或散布度的测量标准。虽然有多种可能的选取，但 Backus 和 Gilbert 选择了平方的二阶矩。该测量标准就是函数 $\mathcal{A}$，

$$\begin{aligned}\mathcal{A} \equiv w(x) &= \int (x'-x)^2 [\widehat{\delta}(x,x')]^2 dx' \\ &= \sum_i \sum_j q_i(x) W_{ij}(x) q_j(x) \equiv \mathbf{q}(x)\cdot\mathbf{W}(x)\cdot\mathbf{q}(x)\end{aligned} \tag{18.6.9}$$

其中使用了(18.6.6)式并将**散布矩阵** $\mathbf{W}(x)$ 定义为

$$W_{ij}(x) \equiv \int (x'-x)^2 r_i(x') r_j(x') dx' \tag{18.6.10}$$

现在函数 $q_i(x)$ 通过最小化原则来确定

$$\text{最小化}\ :\ \mathcal{A} + \lambda\mathcal{B} = \mathbf{q}(x)\cdot\left[\mathbf{W}(x) + \lambda\mathbf{S}\right]\cdot\mathbf{q}(x) \tag{18.6.11}$$

并服从约束条件(18.6.7)式，即 $\mathbf{q}(x)\cdot\mathbf{R} = 1$。

(18.6.11)式的解是

$$\mathbf{q}(x) = \frac{[\mathbf{W}(x) + \lambda\mathbf{S}]^{-1}\cdot\mathbf{R}}{\mathbf{R}\cdot[\mathbf{W}(x) + \lambda\mathbf{S}]^{-1}\cdot\mathbf{R}} \tag{18.6.12}$$

(本节参考文献[4]给出了一个很容易接受的证明。)对于任意特定的数据集 $\mathbf{c}$(测量值 c_i 的集合)，解 $\widehat{u}(x)$ 为

$$\widehat{u}(x) = \frac{\mathbf{c}\cdot[\mathbf{W}(x) + \lambda\mathbf{S}]^{-1}\cdot\mathbf{R}}{\mathbf{R}\cdot[\mathbf{W}(x) + \lambda\mathbf{S}]^{-1}\cdot\mathbf{R}} \tag{18.6.13}$$

(不要被记号所迷惑，误以为是对整个矩阵 $\mathbf{W}(x) + \lambda\mathbf{S}$ 求逆。只要解线性系统 $(\mathbf{W}(x) + \lambda\mathbf{S})\cdot\mathbf{y} = \mathbf{R}$ 求出 $\mathbf{y}$，然后将 $\mathbf{y}$ 代入(18.6.12)式或(18.6.13)式的分子与分母即可。)

(18.6.12)式、(18.6.13)式与(18.5.7)式、(18.5.8)式的线性正则化解有着完全不同的特点。(18.6.12)式中的向量和矩阵的大小都为 N，N 为测量数；基础变量 x 没有离散化，因而 M 不会出现；对每预期的 x 值，都必须解不同的 $N\times N$ 线性方程组集。而在(18.5.8)式中，需解 $M\times M$ 线性方程组，但只需解一次。一般说来，反复解线性方程组这个计算上的负担使 Backus-Gilbert 方法对一维以外的其他问题均不适用。

Backus-Gilbert 方法的 λ 如何选取呢？前面曾提过，在掌握任何实际数据前就能(有时是应该)对 λ 作出选择。对给定的一个 λ 试验值及 x 的一个序列，先利用(18.6.12)式计算 $\mathbf{q}(x)$，然后再运用(18.6.6)式绘制出分辨率函数 $\widehat{\delta}(x,x')$ 为 x' 的函数。这个绘图中对应 x' 的幅度，将反映不同的 x' 值对那点 $\widehat{u}(x)$ 估计值所起的作用的大小。对同一 λ 值，利用(18.6.8)式再绘制出函数 $\sqrt{\mathrm{Var}[\widehat{u}(x)]}$(这需要对协方差矩阵的测量值先做出估计)。

当λ变动时，就会清楚地看到分辨率和稳定性之间的交替换位，这时挑选所需要的值。假如愿意的话，(18.6.12)式和(18.6.13)式中的λ甚至可以取选为 x 的函数，即$\lambda=\lambda(x)$(对每个 x 值都必须求解独立的一组方程这是一个优点)。通过对λ值的选取，现在读者已对反演求解过程有了定量的认识。　且进行真实数据的处理，需要判断某个特定的性质，例如一个尖峰或突变是否真实和(或)实际上是否可解，这时上述做法就极其有用。Backus-Gilbert 方法在地球物理学中已得到了极其成功的应用，用该方法可从地震数据中获取地球结构的信息(比如，密度随深度变化的情况)。

参考文献和深入阅读

Backus, G.E., and Gilbert, F. 1968, *Geophysical Journal of the Royal Astronomical Society*, vol. 16, pp. 169–205. [1]

Backus, G.E., and Gilbert, F. 1970, *Philosophical Transactions of the Royal Society of London A*, vol. 266, pp. 123–192. [2]

Parker, R.L. 1977, *Annual Review of Earth and Planetary Science*, vol. 5, pp. 35–64. [3]

Loredo, T.J., and Epstein, R.I. 1989, *Astrophysical Journal*, vol. 336, pp. 896–919. [4]

18.7　最大熵图像恢复

前面曾介绍过，某些反演方法与 Bayes 观点的结合在很大程度上是历史的偶然，而并非是理智的必然，所谓的**最大熵方法**(maximum entropy method)就是该结合的著名产物。对这些反演方法的概括总结如果不引用一些(至少是引论性的)Bayes 观点，将会变得黯然失色。同时应该指出，这里讲的最大熵反演方法与13.7 节中讨论过的最大熵谱估计之间的联系是非常没有实际意义的。虽然二者都称为**最大熵方法**或 **MEM**，但实际应用中，两种方法并无联系。

从概率论基本公理推出的 Bayes 定理描述了两个事件，比如 A 和 B 之间的条件概率：

$$\mathrm{Prob}(A|B)=\mathrm{Prob}(A)\frac{\mathrm{Prob}(B|A)}{\mathrm{Prob}(B)} \tag{18.7.1}$$

其中$\mathrm{Prob}(A|B)$是度量了事件 B 发生后 A 发生的概率，$\mathrm{Prob}(B|A)$类似，而$\mathrm{Prob}(A)$和$\mathrm{Prob}(B)$为绝对概率。

"Bayes论者"(不妨这样称呼)比所谓的"频率论者"对概率采纳了一种更广泛的解释。Bayes论者认为$\mathrm{Prob}(A|B)$是度量了事件 B 发生后，A 事件发生的可能性的大小，其值变化范围从 0 到 1。在这个较广泛的观点下，A 和 B 不一定为可重复事件，它们可以是命题或假设。于是概率论的公式变成了一组处理推理的相容规则[1,2]。由于可能性本身常常是以一些也许不能明确表达的假设为条件的，因而所有Bayes概率都看做是以一些整体的背景信息 I 为条件的。

假定 H 为某假设，即使在取得任何直接的数据之前，Bayes论者也能赋于 H 一定程度的可能性$\mathrm{Prob}(H|I)$，称为"Bayes先验"。当数据 D_1 出现后，Bayes定理告知如何重新估计 H 的可能性，

$$\mathrm{Prob}(H|D_1I)=\mathrm{Prob}(H|I)\frac{\mathrm{Prob}(D_1|HI)}{\mathrm{Prob}(D_1|I)} \tag{18.7.2}$$

(18.7.2)式右端分子中的因子，是给定假设后数据集的概率，它是可计算的(与 15.1 节中的"可能性"比较)。分母被称为数据的"先验预测概率"，在此情形下它不过是一个归一化常数，它可以根据所有假设的概率之和应该为 1 这个性质来计算出(在其他一些 Bayes 问题中，可用两个完全不同模型的先验预测概率来估计它们的相对可能性)。

如果又获得一些追加数据 D_2，还可将 H 的概率估计值进一步修正为

$$\mathrm{Prob}(H|D_2D_1I)=\mathrm{Prob}(H|D_1I)\frac{\mathrm{Prob}(D_2|HD_1I)}{\mathrm{Prob}(D_2|D_1I)} \tag{18.7.3}$$

利用概率的乘积法则，$\mathrm{Prob}(AB|C)=\mathrm{Prob}(A|C)\ \mathrm{Prob}(B|AC)$，发现(18.7.2)式和(18.7.3)式意味着

$$\mathrm{Prob}(H|D_2D_1I)=\mathrm{Prob}(H|I)\frac{\mathrm{Prob}(D_2D_1|HI)}{\mathrm{Prob}(D_2D_1|I)} \tag{18.7.4}$$

它表明如果所有的数据 $D_1 D_2$ 同时取得时,将得到相同的结果。

从 Bayes 论者的观点看,反演问题就是推理问题[3,4]。基本参数集 $\mathbf{u}$ 是一个假设,在给定测量数据值 $\mathbf{c}$ 后,$\mathbf{u}$ 的概率及 Bayes 先验 $\mathrm{Prob}(\mathbf{u}|I)$ 都可以计算出。也许想找一个单一的"最佳"反演 $\mathbf{u}$,使下式

$$\mathrm{Prob}(\mathbf{u}|\mathbf{c}I) = \mathrm{Prob}(\mathbf{c}|\mathbf{u}I)\frac{\mathrm{Prob}(\mathbf{u}|I)}{\mathrm{Prob}(\mathbf{c}|I)} \tag{18.7.5}$$

在 $\mathbf{u}$ 的所有可能选择下最大。Bayes 分析也允许记录额外信息可能性的存在,该额外信息刻划了 $\mathbf{u}$ 的具有较高的相对概率的可能区域,即所谓 $\mathbf{u}$ 的"后验泡"

给定假设 $\mathbf{u}$,数据 $\mathbf{c}$ 概率的计算与最大似然法中的步骤完全相同。例如,对于高斯误差,由下式给出:

$$\mathrm{Prob}(\mathbf{c}|\mathbf{u}I) = \exp(-\frac{1}{2}\chi^2)\Delta u_0 \Delta u_1 \cdots \Delta u_{M-1} \tag{18.7.6}$$

其中,χ^2 可以采用(18.4.9)式从 $\mathbf{u}$ 和 $\mathbf{c}$ 计算得到,并且所有 Δu_μ 均为 $\mathbf{u}$ 分量的小的常数区域,但这些分量的实际大小无关紧要,因为它们并不依赖于 $\mathbf{u}$(比较(15.1.3)式和(15.1.4)式)。

在最大似然估算中,实际上选取先验$\mathrm{Prob}(\mathbf{u}|I)$为常数。当从大批数据中估计少量的几个参数时,该方法对我们来说很不经济。其中,"参数"($\mathbf{u}$ 的分量)的个数与测量($\mathbf{c}$ 的分量)的个数可相比拟或较大些。并且还需要找一个非平凡的先验$\mathrm{Prob}(\mathbf{u}|I)$来解决解的退化性问题。

在最大熵图像恢复中,出现了**熵**的概念。一个物理系统在宏观状态下的熵,常用 S 表示,是在微观状态下不同结构数的对数,这些微观结构都具有同样的宏观可观测值(即与观察到的宏观状态一致)。实际上,会发现熵的负数也很有用,或称为**负熵**,表示为 $H \equiv -S$(该表示方法可以追溯到 Boltzmann)。在有充分理由确信微观结构的先验概率都相同的情况下(这种情形称为**各态历经的**),一个熵为 S 的宏观状态的 Bayes 先验 $\mathrm{Prob}(\mathbf{u}|I)$与 $\exp(S)$或 $\exp(-H)$成比例。

MEM 利用这个原理,对任意给定的基本函数 $\mathbf{u}$ 赋予一个先验概率。例如在本节参考文献[5~7]中,假定每个像素亮度的测量值可量化(以某种单位)成一个整数值。令

$$U = \sum_{\mu=0}^{M-1} u_\mu \tag{18.7.7}$$

为整个图像中亮度的量化值总数,那么,可根据下述思想来确定"先验",即每个亮度量级在任意像素上都有相等的先验可能性(对于该思想更抽象的证明,参见本节参考文献[8])。因此,获得一个特定结构 $\mathbf{u}$ 的方法的个数为

$$\frac{U!}{u_0! u_1! \cdots u_{M-1}!} \propto \exp\left[-\sum_\mu u_\mu \ln(u_\mu/U) + \frac{1}{2}\left(\ln U - \sum_\mu \ln u_\mu\right)\right] \tag{18.7.8}$$

上式左端可理解为,亮度量化值总数的不同顺序排列的数目,除以每个像素内等价的重新排序的数目,而上式右端是通过阶乘函数的Stirling近似得到的。取负对数,并用在阶为 U 项出现处,忽略阶为$\log U$ 的项,得到负熵为

$$H(\mathbf{u}) = \sum_{\mu=0}^{M-1} u_\mu \ln(u_\mu/U) \tag{18.7.9}$$

现在,根据(18.7.5)式、(18.7.6)式和(18.7.9)式使下式最大:

$$\mathrm{Prob}(\mathbf{u}|\mathbf{c}) \propto \exp\left[-\frac{1}{2}\chi^2\right]\exp[-H(\mathbf{u})] \tag{18.7.10}$$

或等价地使下式最小:

$$\text{最小化}: \quad -\ln[\mathrm{Prob}(\mathbf{u}|\mathbf{c})] = \frac{1}{2}\chi^2[\mathbf{u}] + H(\mathbf{u}) = \frac{1}{2}\chi^2[\mathbf{u}] + \sum_{\mu=0}^{M-1} u_\mu \ln(u_\mu/U) \tag{18.7.11}$$

这应当使读者想起(18.4.11)或(18.5.6)式,或实际上讨论过的沿曲线$\mathcal{A} + \lambda\mathcal{B}$的任意一个最小化原则,其中$\lambda\mathcal{B} = H(\mathbf{u})$是一个正则化算子。$\lambda$怎么取?根据在(18.4.11)式后讨论过的理由:退化反演可能会得到一个 χ^2 的不切实际的小值,需要一个可调整的参数,使 χ^2 落在预期的统计意义上的狭小区域 $N \pm (2N)^{1/2}$ 中。

在 18.4 节中的讨论表明：将λ归入哪一项并没有区别。为了保持记号的一致性，在λ中吸收一个因子 2，并将其归入熵项（一个不确定的λ因子的必要性还可从以下得知：如果在改变 $\mathbf{u}$ 的量化单位时，例如一个 8 位模数转换器用一个 12 位模数转换器来替换时，最小化原则应保持不变，则可看出λ因子是必需的）。现在，也可以用"前面的说法"来表明，这个过程正是获得所选择的统计估算器的过程，它使下式最小：

$$\text{最小化}\ :\quad A+\lambda B=\chi^2[\hat{\mathbf{u}}]+\lambda H(\hat{\mathbf{u}})=\chi^2[\hat{\mathbf{u}}]+\lambda\sum_{\mu=0}^{M-1}\hat{u}_\mu\ln(\hat{u}_\mu) \tag{18.7.12}$$

（也可以添加一项拉格朗日第二乘子$\lambda' U$，以限制总强度 U 为常数。）

不难看出，负熵 $H(\hat{\mathbf{u}})$实际上是一个正则化算子，与$\hat{\mathbf{u}}\cdot\hat{\mathbf{u}}$（(18.4.11)式）或$\hat{\mathbf{u}}\cdot\mathbf{H}\cdot\hat{\mathbf{u}}$（(18.5.6)式）类似。它的下述性质值得注意：

1. 当 U 保持为常数时，对于 $\hat{u}_\mu=U/M=$ 常数，使 $H(\hat{\mathbf{u}})$最小。因而在希望得到一个常数解的意义下，$H(\hat{\mathbf{u}})$是光滑的，这与(18.5.4)式类似。常数解是一个最小值，这个结论是根据 $u\ln u$ 的二阶导数为正值得到的。
2. 但是与(18.5.4)式不同，$H(\hat{\mathbf{u}})$是**局部的**，即对邻近的像素没有区别。它只是简单地将某函数 f 累加起来，此处在所有像素上，函数为
$$f(u)=u\ln u \tag{18.7.13}$$
事实上，当一幅图像中的像素被完全量化时，它是个不变量。这种形式表明，在低强度光滑背景上，嵌入少量极亮的像素（点源时），$H(\hat{\mathbf{u}})$并不显著增加。
3. 当任意一个像素为零时，$H(\hat{\mathbf{u}})$将无限变陡。这一点就强制地使图像保持正值性，这样就不再需要额外的确定性约束。
4. $H(\hat{\mathbf{u}})$与我们见过的其他正则化算子之间的最大区别是，$H(\hat{\mathbf{u}})$不是$\hat{\mathbf{u}}$的二次函数。因而在变化(18.7.12)式后，所得到的方程仍是**非线性的**。这个事实本身就值得进行进一步讨论。

非线性方程比线性方程难解。但是对于图像处理问题，通常有大量的方程需要用迭代过程来求解，即使这些方程是线性的。因而，非线性因素的实际效果得到了缓解。下面，将概括地总结一些 MEM 反演问题中成功应用的方法。

对某些问题，其中一个著名的例子是射电天文学中，根据一组完整的傅里叶系数进行图像恢复的问题，MEM 反演的高级实施甚至还可以部分地探测出 $H(\hat{\mathbf{u}})$的非线性特征。方法之一是（参见本节参考文献[5]），考虑正确测量值的极限 $\sigma_i\to 0$。这时，最小化原则(18.7.12)式中的 χ^2 项被一组约束所替代，其中每个约束都有它自己的拉格朗日乘子，这都为了使模型和数据之间保持一致性，即使式最小：

$$\text{最小化}\ :\quad \sum_j\lambda_j\left[c_j-\sum_\mu R_{j\mu}\hat{u}_\mu\right]+H(\hat{\mathbf{u}}) \tag{18.7.14}$$

（参看(18.4.7)式。）设对 $\hat{u}_\mu$ 的法向导数为零，可得

$$\frac{\partial H}{\partial\hat{u}_\mu}=f'(\hat{u}_\mu)=\sum_j\lambda_j R_{j\mu} \tag{18.7.15}$$

或定义一个函数 G 为f'的逆函数，

$$\hat{u}_\mu=G\left(\sum_j\lambda_j R_{j\mu}\right) \tag{18.7.16}$$

上式中的解只是形式上的，因为必须先求出λ_j，而它根据使(18.7.16)式满足所有添加到(18.7.14)式中的约束条件找出来的。但是，(18.7.16)式也的确表明了下述重要事实，即如果 G 是**线性**的，那么解$\hat{\mathbf{u}}$只包含基函数 $R_{j\mu}$的一个线性组合，其中 $R_{j\mu}$对应着实际测量值j。这相当于设置未测量的 c_j 为零。注意，从(18.4.11)式得到的基本解实际上就有一个线性的 G。

在不完整的傅里叶图像重建问题中，典型的 $R_{j\mu}$有 $\exp(-2\pi i\mathbf{k}_j\cdot\mathbf{x}_\mu)$形式，其中 $\mathbf{x}_\mu$ 是图像空间的一个二维向量，$\mathbf{k}_\mu$ 是一个二维波向量。如果图像包含强点源，那么设置未测量的 c_j 为零的效果，就等于在整个图像

平面产生一些单侧波纹。这些波纹会掩盖位于强点源之间原有的低强度图像的特征。但是,如果 G 的斜率在自变量数值小时会更小,在自变量数值大时更大,那么,图像低强度部分的波纹相对地受到抑制,而强点源相对地将变尖锐("超解"),G 的斜率上的这种性质等价于要求 $f'''(u)<0$。对于函数 $f(u)=u\ln u$,实际有 $f'''(u)=-1/u^2<0$。

用更形象化的语言来说,非线性用来"产生"未测量的 c_i 的非零值,从而使低强度的波纹受到抑制,并使点源变得尖锐。

18.7.1 MEM 特性

负熵函数(18.7.9)的惟一性是怎样的呢?注意,那个方程的前提是假设亮度元素先验地均匀分布到所有像素上。如果想得到另外一种先验图像,比如像素强度为 m_μ,则容易证明负熵函数变成了

$$H(\mathbf{u})=\sum_{\mu=0}^{M-1}u_\mu\ln(u_\mu/m_\mu)+\text{constant} \tag{18.7.17}$$

(常数可以忽略)然后就可以进行其余的讨论。

更重要的是,尽管有许多人极力提倡,但函数形式 $f(u)=u\ln u$ 实际上并不普遍成立。在其他一些物理现象中(例如,在射电天文中,每种模式有许多光量子的限制情况下,一个电磁场的熵),负熵函数实际上是 $f(u)=-\ln u$(其他例子见本节参考文献[5])。一般而言,"熵"的具体定义在任意特定的情形下并没有惟一的答案(对于在适当的条件下,寻找一种更普遍的规则来使熵函数具有某种形式,人们做了很多努力,请参阅本节参考文献[9])。

上面总结的四条性质,加上非线性的所需符号,$f'''(u)<0$,则对函数 $f(u)=-\ln u$ 及 $f(u)=u\ln u$ 都是适用的。事实上,一个很简单的非线性函数 $f(u)=-\sqrt{u}$ 也具有上述性质,但这个函数并不能在理论上确定出任何信息(没有算法)。利用这些熵函数形式,对试验图像进行 MEM 重建,实际上没有任何差别(参见本节参考文献[5])。

所有已知的证据表明,MEM 只不过是至今为止所讨论过的众多一般正则化方法 $\mathcal{A}+\lambda\mathcal{B}$ 中的一种。当将 MEM 应用到根据不完整傅里叶数据进行的图像重建,并且已估计到这些图像除有许多极亮点源外,还包括一些有趣的低强度分布源时,它的独特性使得它对该问题的处理变得强劲。但对于具其他性质的图像,没有理由说 MEM 方法通常比其他正则化方法更好,不管这些正则化方法是已知的还是未知的。

18.7.2 MEM 的算法

目的是寻找向量 $\hat{\mathbf{u}}$,使 $\mathcal{A}+\lambda\mathcal{B}$ 最小,其中各表达式见(18.5.5)式、(18.5.6)式及(18.7.13)式,

$$\mathcal{A}=|\mathbf{b}-\mathbf{A}\cdot\hat{\mathbf{u}}|^2 \qquad \mathcal{B}=\sum_\mu f(\hat{u}_\mu) \tag{18.7.18}$$

与"一般"的最小化问题相比,我们具有可以直接计算梯度及二阶偏导数矩阵(黑塞矩阵)这个有利条件,

$$\begin{aligned}\nabla\mathcal{A}&=2(\mathbf{A}^T\cdot\mathbf{A}\cdot\hat{\mathbf{u}}-\mathbf{A}^T\cdot\mathbf{b}) &\qquad \frac{\partial^2\mathcal{A}}{\partial\hat{u}_\mu\partial\hat{u}_\rho}&=[2\mathbf{A}^T\cdot\mathbf{A}]_{\mu\rho}\\ [\nabla\mathcal{B}]_\mu&=f'(\hat{u}_\mu) &\qquad \frac{\partial^2\mathcal{B}}{\partial\hat{u}_\mu\partial\hat{u}_\rho}&=\delta_{\mu\rho}f''(\hat{u}_\mu)\end{aligned} \tag{18.7.19}$$

重要的是要注意,虽然 $\mathcal{A}$ 的二阶偏导数矩阵不能存储(此矩阵的大小是像素数的平方),但它可以先通过 $\mathbf{A}$,再通过 $\mathbf{A}^T$ 作用到任意向量上。在根据不完整的傅里叶数据进行图像重建的情况中,或在与一个平移不变点分布函数做卷积的情况中,这些应用将包括几个 FFT。同样,在采用 $\mathbf{A}$ 和 $\mathbf{A}^T$ 的应用中,梯度 $\nabla\mathcal{A}$ 的计算也将包括几个 FFT。

虽然经典的共轭梯度法(见 10.6 节)得到了一些成功的应用,但是 $f(u)=u\ln u$ 中的非线性常常会使该方法受到很大的影响。如果试验步长使得 $\hat{\mathbf{u}}$ 取负值,即使只有一个分量为负值,那么该步长也必须被分成几个小步长,有时由于步长尺寸的剧烈减小,求解速度极其缓慢。根本问题在于,共轭梯度方法对黑塞矩阵的逆矩阵信息的收集是一点一点进行的,而且在搜索空间的位置也随之变动。当一个非线性函数与理论上的

二次函数有相当大的区别时，旧信息在得到充分利用之前，就已变得毫无用处。

Skilling 及其合作伙伴[6,7,10,11]提出了一种复杂但却相当成功的方法。该方法在不断搜寻最小值的过程中，不再沿某个单一的搜索方向，而是在一个由向量张成的低(通常是 3)维子空间中进行，这些向量在每点都要重新计算。子空间基向量的选取必须满足避开导致负值的方向。下面这个由向量张成的三维子空间就是最成功的选取方法之一，这些向量的分量由下式给出：

$$
\begin{aligned}
e_\mu^{(1)} &= \hat{u}_\mu[\nabla \mathcal{A}]_\mu \\
e_\mu^{(2)} &= \hat{u}_\mu[\nabla \mathcal{B}]_\mu \\
e_\mu^{(3)} &= \frac{\hat{u}_\mu \sum_\rho (\partial^2 \mathcal{A}/\partial \hat{u}_\mu \partial \hat{u}_\rho)\hat{u}_\rho[\nabla \mathcal{B}]_\rho}{\sqrt{\sum_\rho \hat{u}_\rho ([\nabla \mathcal{B}]_\rho)^2}} - \frac{\hat{u}_\mu \sum_\rho (\partial^2 \mathcal{A}/\partial \hat{u}_\mu \partial \hat{u}_\rho)\hat{u}_\rho[\nabla \mathcal{A}]_\rho}{\sqrt{\sum_\rho \hat{u}_\rho ([\nabla \mathcal{A}]_\rho)^2}}
\end{aligned}
\tag{18.7.20}
$$

(在这些方程中，不对 μ 求和。)如果认为点乘位于一个度量为 $g_{\mu\nu} = \delta_{\mu\nu}/u_\mu$ 的度量空间中，那么可以证明 $\mathbf{e}^{(3)}$ 的选取将“远离”零值，见本节参考文献[6]。

在三维子空间中，三个分量的梯度与九个分量的黑塞矩阵可以通过从大空间的投影来计算，子空间中的最小值可以根据(普通地)解三个联立线性方程组(如同 10.7 节中的(10.7.4)式一样)来估计。步长 $\Delta\hat{\mathbf{u}}$ 的大小必须满足下式：

$$
\sum_\mu (\Delta \hat{u}_\mu)^2/\hat{u}_\mu < (0.1 \text{ to } 0.5)U \tag{18.7.21}
$$

由于梯度方向 $\nabla\mathcal{A}$ 和 $\nabla\mathcal{B}$ 单独使用，因此可以同时进行最小值的搜寻及 λ 的调整，并使其最终满足预期的约束条件。这其中有多种技巧可以采用。

Cornwell 和 Evans[12]提出了一个虽不普遍，但在实际应用中却常常令人满意的方法。应该注意，$\mathcal{B}$ 的黑塞矩阵(二阶偏导数)是一个对角矩阵，那么 $\mathcal{A}$ 的黑塞矩阵是否也存在一个有用的对角矩阵与它近似相等，例如 $2\mathbf{A}^T\cdot\mathbf{A}$。如果用 Λ_μ 表示该近拟矩阵的对角元，那么 $\hat{\mathbf{u}}$ 的一个很有用的步长应该是

$$
\Delta \hat{u}_\mu = -\frac{1}{\Lambda_\mu + \lambda f''(\hat{u}_\mu)}(\nabla \mathcal{A} + \lambda \nabla \mathcal{B}) \tag{18.7.22}
$$

(与(10.7.4)式比较。)更极端的情形是，找一个对角元素为常数的近似对角矩阵，$\Lambda_\mu = \Lambda$，于是

$$
\Delta \hat{u}_\mu = -\frac{1}{\Lambda + \lambda f''(\hat{u}_\mu)}(\nabla \mathcal{A} + \lambda \nabla \mathcal{B}) \tag{18.7.23}
$$

由于 $\mathbf{A}^T\cdot\mathbf{A}$ 具有双卷积点分布函数的一些性质，以及由于实际情形中的点分布函数常常具有一个尖锐的中心峰值，因此这些极端情形也常常是富有成效的。先根据 A_{ij} 对 Λ 进行一个粗略的估计，例如

$$
\Lambda \sim \left\langle \sum_i [A_{i\mu}]^2 \right\rangle \tag{18.7.24}
$$

精确值并不重要，因为在实际情况下，Λ 要进行自适应调整：如果 Λ 太大，则(18.7.23)式的步长将会太小(即同一方向上较大的步长将使 $\mathcal{A} + \lambda\mathcal{B}$ 减小得更多)。如果 Λ 太小，则试验步长将位于一个不可能的区域($\hat{u}_\mu$ 的负值)，或者产生一个较大的 $\mathcal{A} + \lambda\mathcal{B}$。此处 Λ 的调整与 15.5 节中的 Levenberg-Marquardt 方法明显很相似。这是因为 MEM 与非线性最小二乘拟合问题极其相似，因此不应该觉得意外。本节参考文献[12]中还讨论了如何利用 $\Lambda + \lambda f''(\hat{u}_\mu)$ 值去调整拉格朗日乘子 λ，从而使问题收敛到 χ^2 的期望值。

所有实用的 MEM 算法都必须进行大约 30～50 次迭代后才能收敛。目前还没有什么方法可以使收敛问题得到根本解决。

18.7.3　“Bayes”与“历史性”的最大熵

最大熵图像恢复的一些最新进展都表明它们属于“Bayes”，而区别于先前的“历史性”方法。详见本节参考文献[13]。

- 更好的先验：我们已经注意到，最大熵函数((18.7.13)式)在量化所有像素时是不变的，而且也不涉及

光滑性。所谓的"本质关联函数"(ICF)模型(本节参考文献[13]中称之为"新最大熵")与熵函数十分类似,从而算法也很类似。但它要用到邻近关联像素的值,因此有光滑的作用。

- 更好的λ估计值:在上文中,我们适当地选取λ使 χ^2 落在预期的统计意义上的狭小区域 $N \pm (2N)^{1/2}$ 中。但这实际上是过大地估计了 χ^2,因为在重建过程中,参数的一些有效数γ是"固定"的。Bayes 方法则能得到该γ一个自相容的估计值,并使λ的选取更客观。

参考文献和深入阅读

Jaynes, E.T. 1976, in *Foundations of Probability Theory, Statistical Inference, and Statistical Theories of Science*, W.L. Harper and C.A. Hooker, eds. (Dordrecht: Reidel). [1]

Jaynes, E.T. 1985, in *Maximum-Entropy and Bayesian Methods in Inverse Problems*, C.R. Smith and W.T. Grandy, Jr., eds. (Dordrecht: Reidel). [2]

Jaynes, E.T. 1984, in *SIAM-AMS Proceedings*, vol. 14, D.W. McLaughlin, ed. (Providence, RI: American Mathematical Society). [3]

Titterington, D.M. 1985, *Astronomy and Astrophysics*, vol. 144, 381–387. [4]

Narayan, R., and Nityananda, R. 1986, *Annual Review of Astronomy and Astrophysics*, vol. 24, pp. 127–170. [5]

Skilling, J., and Bryan, R.K. 1984, *Monthly Notices of the Royal Astronomical Society*, vol. 211, pp. 111–124. [6]

Burch, S.F., Gull, S.F., and Skilling, J. 1983, *Computer Vision, Graphics and Image Processing*, vol. 23, pp. 113–128. [7]

Skilling, J. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [8]

Frieden, B.R. 1983, *Journal of the Optical Society of America*, vol. 73, pp. 927–938. [9]

Skilling, J., and Gull, S.F. 1985, in *Maximum-Entropy and Bayesian Methods in Inverse Problems*, C.R. Smith and W.T. Grandy, Jr., eds. (Dordrecht: Reidel). [10]

Skilling, J. 1986, in *Maximum Entropy and Bayesian Methods in Applied Statistics*, J.H. Justice, ed. (Cambridge: Cambridge University Press). [11]

Cornwell, T.J., and Evans, K.F. 1985, *Astronomy and Astrophysics*, vol. 143, pp. 77–83. [12]

Gull, S.F. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [13]

第 19 章　偏微分方程

19.0　引言

偏微分方程的数值解法就其自身来说，是一个广阔的研究课题。偏微分方程即使不是绝大多数，也是许多连续物理系统，如流体、电磁场及人体等的计算机分析或模拟的核心所在。本章旨在给出最简单明了的介绍，还可参阅本节参考文献[1～4]。

在大多数数学著作中，偏微分方程(PDE)根据其特征线或信息传播曲线分为三种类型，即**双曲型**、**抛物型**和**椭圆型**。双曲型方程的典型例子是一维**波动**方程

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \tag{19.0.1}$$

其中 v 是常数，为波的传播速度。抛物型方程的模型方程是**扩散**方程

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D\frac{\partial u}{\partial x}\right) \tag{19.0.2}$$

其中 D 是扩散系数。椭圆型方程的模型方程是**泊松**(Poisson)方程

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho(x,y) \tag{19.0.3}$$

其中源项 ρ 给定。如果该源项等于零，则方程称为**拉普拉斯(Laplace)方程**。

从计算的角度来看，这三种标准类型的分类并不是非常有意义的，至少不如一些其他的本质性区分更重要。(19.0.1)式和(19.0.2)式都定义了**初值问题**或称为**柯西(Cauchy)问题**：如果有关 u 的信息(可能包括 u 的时间导数信息)在某个初始时刻 t_0 对任意 x 是已知的，则方程描述了 $u(x,t)$ 如何随时间向前传播它自身。换言之，(19.0.1)式和(19.0.2)式描述了 u 随时间的演变过程。一个数值解的目的在于：按某预定的精度，追踪这个随时间的演变过程。

相反地，(19.0.3)式则要求找一个"静态"的函数 $u(x,y)$，使之在 (x,y) 的讨论区域里满足该方程，并且——特别要指明的是——还必须使之在讨论的区域边界上具有某些期望的特性。这类问题被称为**边值问题**。由于一个初值问题可"沿时间向前积分"，在相同意义下，一般边值问题不可能是稳定地正好"从边界向里积分"。因此，数值解的目的是：以某种方式尽快在任意位置收敛到正确解。

因此，从计算的角度来看，最重要的分类方法是：眼前的问题是**初值**问题(随时间的发展)还是**边值**问题(静态解)。图 19.0.1 形象表示了它们之间的区别。注意，虽然上面**加引号**的术语是标准用语，但从计算的角度来看，括号中的术语才是对这两类问题的更好描述。初值问题再分成双曲型和抛物型就不太重要了，因为(i)许多实际问题都是混和型的，(ii)正如我们将要看到的，在讨论实际的计算方案时，大多数双曲型问题都混杂有抛物型的成分。

19.0.1　初值问题

定义一个初值问题需要回答如下问题：

- 随时间向前传播的因变量是什么？
- 每个变量的发展方程是什么？通常这些发展方程都是耦合的，即在每个方程的右端，同时出现了不止一个因变量。
- 每个变量的发展方程中，出现的最高阶时间导数是什么？如果可能，关于时间的导数应该单独放在方程的左端。不仅是变量的值，而且包括它的所有时间导数的值(直至其最高阶导数)都必须确定，以定义这个发展方程。

- 约束着有关空间区域的边界点,随时间发展的特殊方程(即边界条件)是什么?例如,**Dirichlet 条件**,把边界点的值定义为时间的函数;**Neumann 条件**则赋予了边界上的法向梯度值;**行波边界条件**,正如所说的条件。

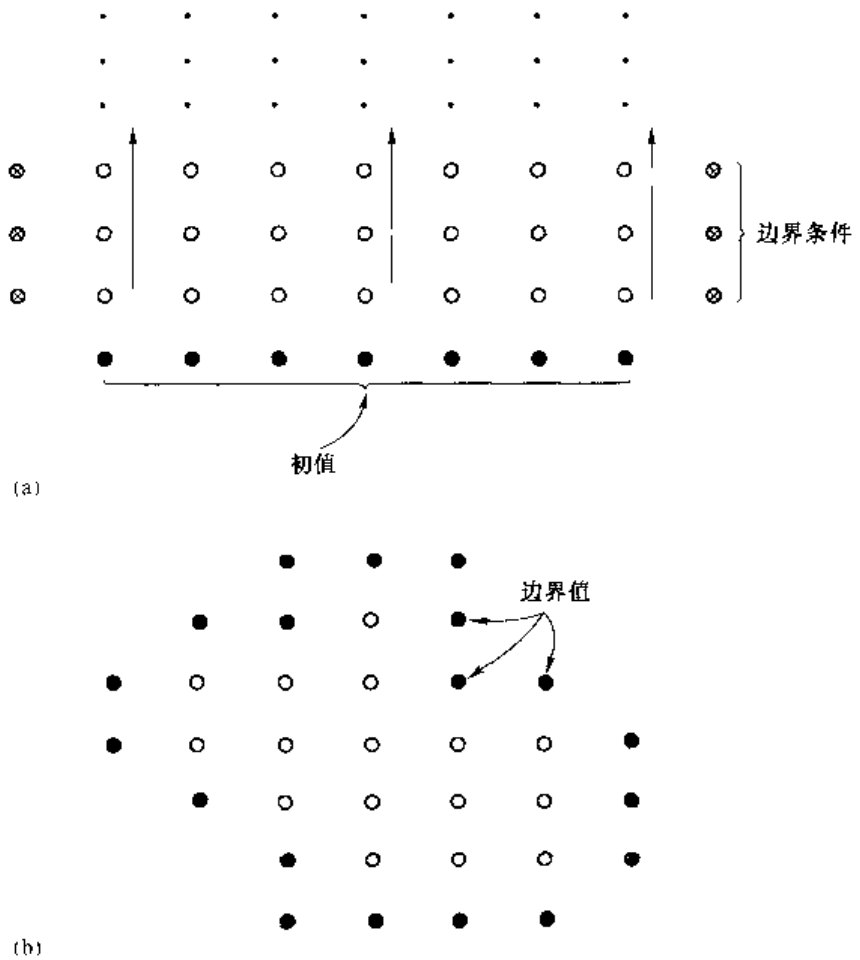


图 19.0.1 初值问题(a)和边值问题(b)的对比。在(a)中,初值被给定在一个"时间层"上,并且希望沿箭头所示方向逐行计算空心圆点的解,得到随时间向前发展的解。而在每一行左边和右边(⊗)的边界条件也必须满足,但是每一个时刻仅计算一行,在内存中,只需保留先前的一行或少数几行。在(b)中,边界值沿一个网格的边界给定,并使用一个迭代过程以确定所有内部点(空心点)的值。所有网格点都必须保留在内存中

本章 19.1 节 ~ 19.3 节将介绍几种不同形式的初值问题。本章的内容是一些经过仔细挑选的典型范例,提供了大量概括性的知识。这些例子说明一个重要的思想:在原则上,计算时需特别注意算法的稳定性,在初值问题中,许多貌似合理的算法,事实上并不能运行——数值上是不稳定的。

19.0.2 边值问题

定义一个边值问题需回答如下问题:

- 变量是什么?
- 哪些方程在讨论的区域内部成立?
- 哪些方程在讨论区域的边界上逐点成立?(对二阶椭圆方程,Dirichlet 条件和 Neumann 条件是可能的选择条件,但也可能遇到更加复杂的边界条件。)

与初值问题相反,对于边值问题,稳定性相对容易获得。这样一来,算法的效率,包括计算量和存储要求,就成了考虑的重点。

由于在边值问题中,所有的条件都必须"同时地"满足,因此这些问题至少在概念上通常可简化为大量联立代数方程的求解。当这些方程为非线性时,通常采用线性化或迭代的方法来求解。因此,不失一般性,可

以把边值问题看做求解特殊的大型线性方程组.

作为将在 19.4 节 ~ 19.6 节中作为"模型问题"引用的一个例子,让我们考虑一下(19.0.3)式的**有限差分法**。用函数 $u(x,y)$在离散点集上的取值来代表函数本身:

$$\begin{aligned} x_j &= x_0 + j\Delta, \qquad j = 0,1,\ldots,J \\ y_l &= y_0 + l\Delta, \qquad l = 0,1,\ldots,L \end{aligned} \tag{19.0.4}$$

其中 Δ 为**网格间距**。从现在起,用 $u_{j,l}$ 表示 $u(x_j, y_l)$,用 $\rho_{j,l}$ 表示 $\rho(x_j, y_l)$。用一个有限差分表达式(参见图 19.0.2)代替(19.0.3)式:

$$\frac{u_{j+1,l} - 2u_{j,l} + u_{j-1,l}}{\Delta^2} + \frac{u_{j,l+1} - 2u_{j,l} + u_{j,l-1}}{\Delta^2} = \rho_{j,l} \tag{19.0.5}$$

或者等价地为

$$u_{j+1,l} + u_{j-1,l} + u_{j,l+1} + u_{j,l-1} - 4u_{j,l} = \Delta^2 \rho_{j,l} \tag{19.0.6}$$

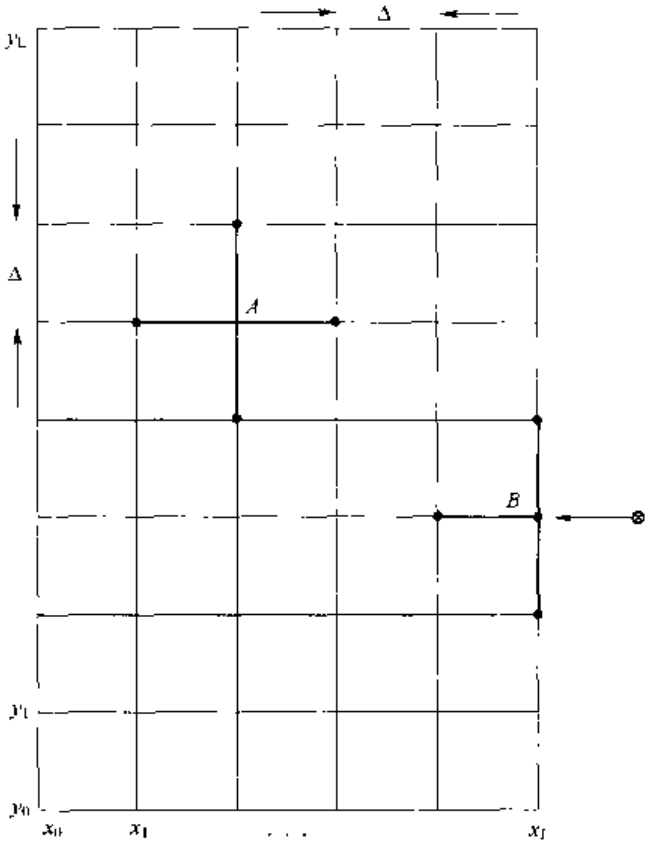


图 19.0.2　二阶椭圆方程在二维网格上的有限差分表示图。在 A 点的二阶导数用图中所示的与 A 相邻的点来计算;在 B 点的二阶导数也用相邻结点来计算,同时还用到"右端项"的边界信息,用符号⊗表示

为了把这个线性方程组系统写为矩阵形式,需要从 u 中构造一个向量。用如下定义的一个一维序列来标记二维网格点:

$$i \equiv j(L+1) + l \qquad \text{for} \qquad j = 0,1,\ldots,J, \qquad l = 0,1,\ldots,L \tag{19.0.7}$$

换言之,沿表示 y 值的列数,i 上升得最快。(19.0.6)式现在变为

$$u_{i+L+1} + u_{i-(L+1)} + u_{i+1} + u_{i-1} - 4u_i = \Delta^2 \rho_i \tag{19.0.8}$$

该方程仅在内点 $j = 1,2,\ldots,J-1$; $l = 1,2,\ldots,L-1$ 处成立。

位于

$$
\begin{aligned}
j &= 0 \qquad [\text{i.e.}, i = 0, \ldots, L] \\
j &= J \qquad [\text{i.e.}, i = J(L+1), \ldots, J(L+1)+L] \\
l &= 0 \qquad [\text{i.e.}, i = 0, L+1, \ldots, J(L+1)] \\
l &= L \qquad [\text{i.e.}, i = L, L+1+L, \ldots, J(L+1)+L]
\end{aligned}
\tag{19.0.9}
$$

处的点为边界点。在这些点上的 u,或者是 u 的导数已经给定,如果把所有的"已知"信息都放到(19.0.8)式的右端,则该方程具有形式为

$$\mathbf{A} \cdot \mathbf{u} = \mathbf{b} \tag{19.0.10}$$

其中矩阵 **A** 具有如图(19.0.3)所示的形式。

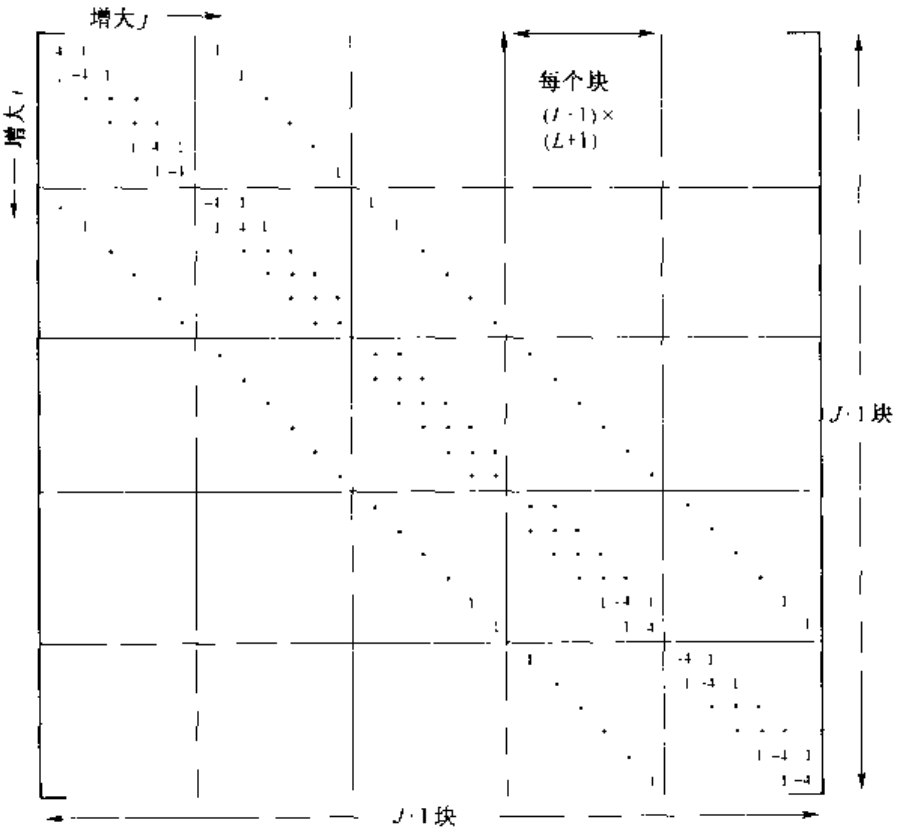


图 19.0.3 由二阶椭圆方程(在此为(19.0.6)式)导出的矩阵结构。所有未标明的元素为零,该矩阵有自身为三对角线型的对角块,这种形式的矩阵称为"带边的三对角型矩阵"。一个如此稀疏的矩阵不必按图示的完整形式存储

矩阵 **A** 称为"带边的三对角型"矩阵。一个常见的二阶线性椭圆方程:

$$
\begin{aligned}
a(x,y)\frac{\partial^2 u}{\partial x^2} + b(x,y)\frac{\partial u}{\partial x} + c(x,y)\frac{\partial^2 u}{\partial y^2} + d(x,y)\frac{\partial u}{\partial y} \\
+ e(x,y)\frac{\partial^2 u}{\partial x \partial y} + f(x,y)u = g(x,y)
\end{aligned}
\tag{19.0.11}
$$

其相应矩阵除了非零元素不是常数外,有着与 **A** 相似的结构。

粗略地分类,有三种不同的方法可求解(19.0.10)式,当然并非处处适用:松弛法、"快速"法(如傅里叶方法)和直接矩阵法。

松弛法直接利用稀疏矩阵 **A** 的结构,把矩阵 **A** 分成两部分:

$$\mathbf{A} = \mathbf{E} - \mathbf{F} \tag{19.0.12}$$

其中 **E** 是容易求逆的矩阵,**F** 是余量矩阵。因而(19.0.10)式变成

$$\mathbf{E} \cdot \mathbf{u} = \mathbf{F} \cdot \mathbf{u} + \mathbf{b} \tag{19.0.13}$$

松弛法包括选择 $\mathbf{u}$ 的初始值 $\mathbf{u}^{(0)}$，然后通过逐次求解，从

$$\mathbf{E}\cdot\mathbf{u}^{(r)}=\mathbf{F}\cdot\mathbf{u}^{(r-1)}+\mathbf{b} \tag{19.0.14}$$

迭代获得 $\mathbf{u}^{(r)}$。由于被选的 $\mathbf{E}$ 容易求逆，因而每步迭代都很迅速。将在 19.5 节和 19.6 节详细讨论松弛法。

所谓的快速法[5]仅适用于一类相当特殊的方程：这些方程具有常系数，或者更一般地，在已选择的坐标系中，它们是可分离的；另外，边界必须与坐标轴重合。这类特殊方程在实际应用中经常遇到，将在 19.4 节进行详细讨论。但请注意，将在 19.6 节讨论的多重网格松弛法可能比"快速法"更快。

矩阵法力图直接求解方程

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{19.0.15}$$

其实用程度极强地依赖于当前问题中矩阵 $\mathbf{A}$ 的具体结构，因此关于这一点的讨论，仅限于一些评述和提示。

矩阵的稀疏程度必须首先考虑，否则矩阵问题将非常巨大。例如，一个在 100×100 的空间网络上最简单的问题，将涉及到 10 000 个未知的 $u_{j,l}$，它蕴含一个 $10\,000\times10\,000$ 的矩阵 $\mathbf{A}$，包含 10^8 个元素。

正如在 2.7 节末尾讨论的，如果 $\mathbf{A}$ 是对称正定的（在椭圆问题中通常如此），那么可以使用共轭梯度法。在实际应用中，舍入误差经常破坏了共轭梯度法求解有限差分方程的效果。但是在这种方法中，如果首先改写方程，使矩阵 $\mathbf{A}$ 变换成与单位矩阵近似的矩阵 $\mathbf{A}'$，那么上述方法还是可用的。此时，这些方程定义的二次曲面都具有近似的球形轮廓，并且共轭梯度法运行效果很好。在 2.7 节的程序 linbcg 中，一个类似的**预条件器**被用来以更一般的双共轭梯度法求解非正定问题。至于在 PDE 中出现的正定情形，一个成功实现此方法的例子是**不完全 Cholesky 共轭梯度法**（**ICCG**）[6～8]。

另一种依赖于变换手段的方法是 Stone 提供的**强隐式过程**[9]，一个运用该过程的名为 SIPSOL 的程序已发表，参见本节参考文献[10]。

第三类矩阵法是如 2.7 节所描述的分析-因式分解-运算方法（Analyze-Factorize-Operate approach）。

总而言之，当所具有的内存可以使用上述矩阵法——通常是不超过 10^8，但却比松弛法所需容量大得多时——则可以考虑使用。此时只有多重网格松弛法（见 19.6 节）可与最佳的矩阵法相比。但当网格大于 300×300，一般会发现只有松弛法，或快速法（在其可用时），才是可能的方法。

19.0.3　有限差分以外的众多方法

除了有限差分法，还有一些方法可以求解偏微分方程，最重要的包括有限元法、蒙特卡罗法、谱方法和变分法。遗憾的是，在本章只能对有限差分法做一些仔细介绍，本书将不对其他方法进行讨论。有限元法[11～12]尤为固体机械学和结构工程学的实际工作者偏爱，这种方法在设置计算元素的位置时，提供了相当大的自由度，在处理高度不规则的几何区域问题时，也非常重要；谱方法[13～15]适于非常规则的几何区域问题和光滑函数，其收敛比有限差分法快得多（参见 19.4 节），但对不连续的问题，该方法效果并不好。

参考文献和深入阅读

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press). [1]

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience). [2]

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa). [3]

Mitchell, A.R., and Griffiths, D.F. 1980, *The Finite Difference Method in Partial Differential Equations* (New York: Wiley) [includes discussion of finite element methods]. [4]

Dorr, F.W. 1970, *SIAM Review*, vol. 12, pp. 248–263. [5]

Meijerink, J.A., and van der Vorst, H.A. 1977, *Mathematics of Computation*, vol. 31, pp. 148–162. [6]

van der Vorst, H.A. 1981, *Journal of Computational Physics*, vol. 44, pp. 1–19 [review of sparse iterative methods]. [7]

Kershaw, D.S. 1970, *Journal of Computational Physics*, vol. 26, pp. 43–65. [8]

Stone, H.J. 1968, *SIAM Journal on Numerical Analysis*, vol. 5, pp. 530–558. [9]

Jesshope, C.R. 1979, *Computer Physics Communications*, vol. 17, pp. 383–391. [10]

Strang, G., and Fix, G. 1973, *An Analysis of the Finite Element Method* (Englewood Cliffs, NJ: Prentice-Hall). [11]

Burnett, D.S. 1987, *Finite Element Analysis: From Concepts to Applications* (Reading, MA: Addison-Wesley). [12]

Gottlieb, D. and Orszag, S.A. 1977, *Numerical Analysis of Spectral Methods: Theory and Applications* (Philadelphia: S.I.A.M.). [13]

Canuto, C., Hussaini, M.Y., Quarteroni, A., and Zang, T.A. 1988, *Spectral Methods in Fluid Dynamics* (New York: Springer-Verlag). [14]

Boyd, J.P. 1989, *Chebyshev and Fourier Spectral Methods* (New York: Springer-Verlag). [15]

19.1 通量守恒的初值问题

一维空间的一大类偏微分方程初值(随时间发展)问题,可以归入**通量守恒方程**的形式:

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\partial \mathbf{F}(\mathbf{u})}{\partial x} \tag{19.1.1}$$

其中,$\mathbf{u}$ 和 $\mathbf{F}$ 为向量,且(在一些情况下)$\mathbf{F}$ 不但依赖于 $\mathbf{u}$,而且依赖于 $\mathbf{u}$ 的空间导数。向量 $\mathbf{F}$ 称为**守恒通量**。

例如,双曲型方程的模型,具有常数传播速度 v 的一维波动方程:

$$\frac{\partial^2 u}{\partial t^2} = v^2 \frac{\partial^2 u}{\partial x^2} \tag{19.1.2}$$

上式可以改写成两个一阶方程的方程组:

$$\begin{aligned} \frac{\partial r}{\partial t} &= v\frac{\partial s}{\partial x} \\ \frac{\partial s}{\partial t} &= v\frac{\partial r}{\partial x} \end{aligned} \tag{19.1.3}$$

其中

$$\begin{aligned} r &\equiv v\frac{\partial u}{\partial x} \\ s &\equiv \frac{\partial u}{\partial t} \end{aligned} \tag{19.1.4}$$

在这种情况下,r 和 s 成为 $\mathbf{u}$ 的两个分量,而通量 $\mathbf{F}$ 则由线性矩阵关系确定:

$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} 0 & -v \\ -v & 0 \end{pmatrix} \cdot \mathbf{u} \tag{19.1.5}$$

(学习过物理学的读者可能会发现,(19.1.3)式类似于 Maxwell 的一维电磁波传播方程。)

在本节,将考虑一般的通量守恒方程(19.1.1)的一个典型例子,即关于 u 为标量的方程:

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} \tag{19.1.6}$$

其中 v 是常数。对这种情形,通过分析知道,该方程的通解是一列沿 x 轴正方向传播的波:

$$u = f(x - vt) \tag{19.1.7}$$

其中 f 为一任意的函数。当然,探讨的数值解法将同样适用于(19.1.1)式所表示的更加一般化的方程。在一些文章中,(19.1.6)式被称为**对流方程**,因为量 u 是通过速度为 v 的"流体流动"来传送的。

如何对(19.1.6)式(若类似地,对(19.1.1)式)进行有限差分呢?一种简单明了的方法是,沿 t 轴和 x 轴做等距划分,等分点可表示为

$$\begin{aligned} x_j &= x_0 + j\Delta x, \qquad j = 0, 1, \ldots, J \\ t_n &= t_0 + n\Delta t, \qquad n = 0, 1, \ldots, N \end{aligned} \tag{19.1.8}$$

用 u_j^n 表示 $u(t_n, x_j)$。有几种方法表示 u 的时间导数项,一种明显的方法是取

$$\left.\frac{\partial u}{\partial t}\right|_{j,n} = \frac{u_j^{n+1} - u_j^n}{\Delta t} + O(\Delta t) \tag{19.1.9}$$

这称为**欧拉(Euler)向前差分法**(参见(16.1.1)式)。虽然欧拉向前差分法只有一阶精度,但它有一个优点,就是仅用第 n 时间层的已知值,即可计算 $n+1$ 时间层的值。对于空间导数,可以使用二阶精度的表示式,仍然

只用到 n 时间层的已知值：

$$\left.\frac{\partial u}{\partial x}\right|_{j,n}=\frac{u_{j+1}^n-u_{j-1}^n}{2\Delta x}+O(\Delta x^2) \tag{19.1.10}$$

由此而得到的(19.1.6)式的有限差分逼近称为 FTCS 表示(即时间向前差分空间中心差分)：

$$\frac{u_j^{n+1}-u_j^n}{\Delta t}=-v\left(\frac{u_{j+1}^n-u_{j-1}^n}{2\Delta x}\right) \tag{19.1.11}$$

很容易将上式变换成利用其他量计算 u_j^{n+1} 的公式。FTCS 格式的图解说明见图 19.1.1。它是一个很好的例子，该算法易于推导，占据极少内存且计算速度快。但实际上却差得无法运行(详看下文)。

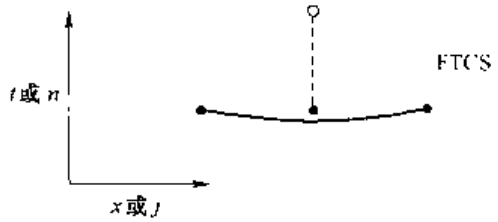


图 19.1.1　时间向前差分空间中心差分(FTCS)格式的图示。在该图和后面的图中，空心圆圈是希望求解所在的新点；实心圆圈是已知点，其函数值用于计算新点；实线连接用于计算空间导数的点；虚线则连接用于计算时间导数的点。对双曲型问题FTCS格式，通常不稳定，故常不使用

FTCS 表示是一种显格式，这意味着任意 j 值的 u_j^{n+1} 均可由已知的值显式地算出。后面将遇到隐格式，它要求我们对不同的 j 求解耦合 u_j^{n+1} 的隐式方程(对于常微分方程的显式和隐式方法在 16.6 节已讨论了)。FTCS 算法也是一个单层格式的例子，因为要计算第 $n+1$ 时间层的值，只需存储第 n 时间层的值。

19.1.1　von Neumann 稳定性分析

非常遗憾，(19.1.11)式的用处很有限，它是一种不稳定的方法，只能用来(如果确实要用)研究一小段振荡时间里的波。为了寻找具有更普遍适用性的替代方法，必须先介绍一下 **von Neumann 稳定性分析**。

von Neumann 分析是局部的：假设差分方程的系数变化非常缓慢，以至于可视为时间域和空间域中的常数。在此情况下，该差分方程的无关解或称为**特征模**，都具有形式

$$u_j^n=\xi^n e^{ikj\Delta x} \tag{19.1.12}$$

其中 k 是实的空间波数(可取任意值)，且 $\xi=\xi(k)$ 是依赖于 k 的复数。关键之处是，每个特征模的时间依赖性只不过是复数 ξ 的连续整数次幂，因此，如果对某些 k 值，$|\xi(k)|>1$，则差分方程是不稳定的(即有指数增长的趋势)。ξ 的值称为在已知波数 k 处的**放大因子**。

为了获得 $\xi(k)$，只需简单地将(19.1.12)式代入(19.1.11)式，除以 ξ^n，得

$$\xi(k)=1-i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.13}$$

对任何 k 值，它的模数均大于 1，因此，FTCS 格式是无条件不稳定的。

如果速度 v 是 t 和 x 的函数，则在(19.1.11)式中应写入 v_j^n。在 von Neumann 稳定性分析中，仍将 v 表示为常数，因为对 v 缓慢变化的情形，分析是局部进行的。事实上，即使在 v 为严格常数时，von Neumann 分析也不能严格地处理在 $j=0$ 和 $j=N$ 处的终端效应。

更一般地，如果方程右端对 u 是非线性的，那么通过改写 $u=u_0+\delta u$，将方程展开为 δu 的线性方程，则 von Neumann分析可以将其线性化。假设 u_0 的值已完全满足差分方程，则von Neumann分析旨在寻找 δu 的一个不稳定特征模。

尽管 von Neumann 方法不很严密，但它一般得出正确的答案，而且比其他更仔细的方法更容易使用，因此，只采用这种方法进行分析(例如，参见本节参考文献[1]对其他稳定性分析方法的讨论)。

19.1.2　Lax 方法

FTCS 方法的不稳定性可以通过 Lax 的一个简单变换得到修正：替换时间导数项中的 u_j^n 为其平均值(见图 19.1.2)：

$$u_j^n \to \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) \tag{19.1.14}$$

则(19.1.11)式变为

$$u_j^{n+1} = \frac{1}{2}\left(u_{j+1}^n + u_{j-1}^n\right) - \frac{v\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right) \tag{19.1.15}$$

将之代入(19.1.12)式中,我们发现放大因子

$$\xi = \cos k\Delta x - i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.16}$$

由稳定性条件$|\xi|^2 \leqslant 1$的要求有

$$\frac{|v|\Delta t}{\Delta x} \leqslant 1 \tag{19.1.17}$$

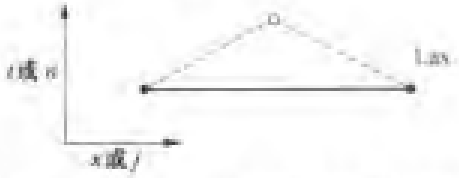


图 19.1.2 Lax 差分格式的图示。定义如前图,该格式的稳定性差别准则是 Courant 条件

这便是著名的 Courant-Friedrichs-Lewy 稳定性准则,通常被简称为 **Courant 条件**。直观地,这种稳定性条件可以理解如下(见图 19.1.3):在(19.1.15)式中,量 u_j^{n+1} 由 n 时刻在点 $j-1$ 和 $j+1$ 处的信息计算而得,换言之,x_{j-1} 和 x_{j+1} 是允许为 u_j^{n+1} 传递信息的空间区域的边界。现在想一想在连续的波动方程中,信息以最大速度 v 传播,如果点 u_j^{n+1} 在图 19.1.3 所示的阴影之外,则它需要从比该差分格式允许的位置更远处的点中获得信息,缺少的那些信息必导致不稳定。因此,Δt 不能选得太大。

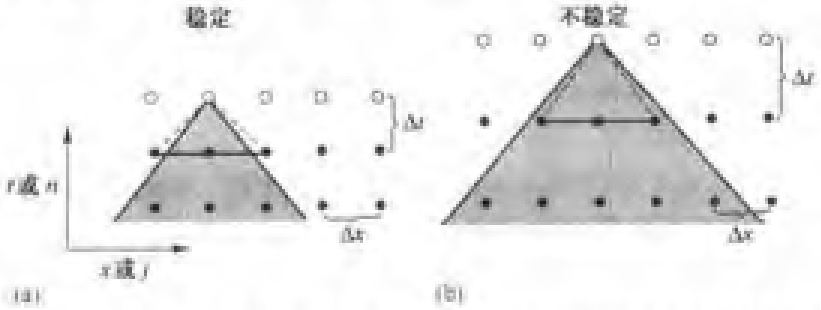


图 19.1.3 差分格式稳定性的Courant条件。双曲问题的解在某一点的值取决于过去的某依赖区域中的信息,在此示为阴影。差分格式(19.1.15)式也有其自身的依赖区域,该依赖区域由一个时间层上(以实心圆的连接表明)点的选取来确定,而时间层上的值用于确定一个新点(以虚线连接表明)。如果差分格式的依赖区域大于偏微分方程的依赖区域,则该差分方案是Courant稳定的,如图中(a)情形;而如果两者关系相反,则该差分格式不稳定,如图中(b)所示。对于较复杂的差分格式,依赖区域可能不单由最外层的点决定

(19.1.14)式的简单替换稳定了 FTCS 格式,这个结果使我们认识到,偏微方程的差分是一门科学,也是一种技巧。为了一定程度地减少这种技巧的神秘感,让我们改写(19.1.15)式,使之具有(19.1.11)式带有余项的形式,然后比较一下 FTCS 格式和 Lax 格式:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = -v\left(\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}\right) + \frac{1}{2}\left(\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta t}\right) \tag{19.1.18}$$

这恰恰就是方程

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} + \frac{(\Delta x)^2}{2\Delta t}\nabla^2 u \tag{19.1.19}$$

的 FTCS 表示。其中,一维情形下,$\nabla^2 = \partial^2/\partial x^2$。事实上,是在式中加入了一个扩散项,或者,回忆一下粘性流体运动的 Navier-Stokes 方程,就是加了一个耗散项。因此 Lax 格式被认为具有数值耗散或数值粘性。在放大因子中也会发现这一点:除非$|v|\Delta t$恰巧等于Δx,否则有$|\xi| < 1$,且波的振幅伪衰减。

伪衰减是不是比伪上升好些呢？回答是肯定的。我们希望精确研究的尺度是包含许多网格点，因此有 $k\Delta x \ll 1$（空间波数 k 在(19.1.12)式中已定义）。对于这些尺度，放大因子在稳定格式和不稳定格式中，都可视为非常接近于 1，因此稳定格式和不稳定格式基本上同等地精确。但是，对不感兴趣的小尺度，其中 $k\Delta x \sim 1$，使用不稳定格式将淹没解中有意义的部分。而稳定格式的情形就好得多，其中短波长部分毫无影响地消失。稳定格式和不稳定格式对这些短波长部分都是**不精确**的，但当格式稳定时，这种不精确是一种可以容忍的性质。

当自变量 $\mathbf{u}$ 是一个向量时，von Neumann 分析稍有些复杂。例如，可以考虑把(19.1.3)式改写为

$$\frac{\partial}{\partial t}\begin{bmatrix} r \\ s \end{bmatrix} = \frac{\partial}{\partial x}\begin{bmatrix} vs \\ vr \end{bmatrix} \tag{19.1.20}$$

对该方程使用 Lax 方法：

$$\begin{aligned} r_j^{n+1} &= \frac{1}{2}(r_{j+1}^n + r_{j-1}^n) + \frac{v\Delta t}{2\Delta x}(s_{j+1}^n - s_{j-1}^n) \\ s_j^{n+1} &= \frac{1}{2}(s_{j+1}^n + s_{j-1}^n) + \frac{v\Delta t}{2\Delta x}(r_{j+1}^n - r_{j-1}^n) \end{aligned} \tag{19.1.21}$$

通过假设特征模具有如下(向量)形式，继续进行 von Neumann 稳定性分析

$$\begin{bmatrix} r_j^n \\ s_j^n \end{bmatrix} = \xi^n e^{ikj\Delta x}\begin{bmatrix} r^0 \\ s^0 \end{bmatrix} \tag{19.1.22}$$

其中，右端向量(在空间域和时间域)是常特征向量，ξ 和以前一样是复数。把(19.1.22)式代入(19.1.21)式再除以幂 ξ^n，得到齐次向量方程：

$$\begin{bmatrix} (\cos k\Delta x) - \xi & i\dfrac{v\Delta t}{\Delta x}\sin k\Delta x \\ i\dfrac{v\Delta t}{\Delta x}\sin k\Delta x & (\cos k\Delta x) - \xi \end{bmatrix} \cdot \begin{bmatrix} r^0 \\ s^0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{19.1.23}$$

当且仅当左边矩阵的行列式等于 0 时，该方程有解；由此条件，很容易得到 ξ 的两个根：

$$\xi = \cos k\Delta x \pm i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.24}$$

稳定的条件是两个根都满足 $|\xi| \leqslant 1$，结果恰好又是(19.1.17)式的 Courant 条件。

19.1.3　其他种类的误差

迄今为止，已经详细探讨了**振幅误差**，它与差分格式的稳定性或不稳定性紧密相关。当转移注意力到精确度上而不是稳定性上时，还有其他种类的误差与之相应。

双曲型方程的有限差分格式可以呈现色散或者**相位误差**。例如(19.1.16)式可以重写成

$$\xi = e^{-ik\Delta x} + i\left(1 - \frac{v\Delta t}{\Delta x}\right)\sin k\Delta x \tag{19.1.25}$$

任意一个初始波群都是由不同 k 值的波形迭加而成。在每个时间层上，波形依据不同的 k 值乘上一个不同的相位因子(19.1.25)。如果 $\Delta t = \Delta x/v$，则因为每个波形乘以 $\exp(-ik\Delta x)$，所以能得到波群中每个波形的精确解 $f(x - vt)$。对于这个 Δt 值，(19.1.25)式表明有限差分方程解是精确的解析解。然而，若 $v\Delta t/\Delta x$ 不精确地等于 1，波形的相位关系可能变得混乱不堪，并且波群发散。由(19.1.25)式可知，只要波长变得与网格间隔 Δx 可比拟，这种色散将变大。

第三类误差与非线性双曲型方程相联系，因此有时被称为**非线性不稳定性**。例如，流体运动的欧拉方程或 Navier-Stokes 方程的一部分可表示为

$$\frac{\partial v}{\partial t} = -v\frac{\partial v}{\partial x} + \cdots \tag{19.1.26}$$

关于 v 的非线性项可能导致傅里叶空间中，从长波长向短波长的能量转换，这引起波形变陡，直至垂直形状或"激波"发生。von Neumann 分析中指出，稳定性取决于 $k\Delta x$，因此一种在平缓波形下稳定的格式可能在陡峭波形下变得不稳定。在此类差分格式中，傅里叶空间的级联终止在网格上可表示的最短波长处，即 $k \sim 1/\Delta x$ 处时，这类麻烦将会出现。如果能量简单地以这些方式累加，最终必然淹没长波长的能量。

非线性不稳定性和激波的形成,可一定程度地通过前面讨论的(19.1.18)式中的数值粘性来控制。但一些流体问题中,激波的形成是一种细节有待研究的、流体的实际物理行为,因此,仅仅有数值粘性尚不足够或尚不足以进行控制。将在后面从流体动力学角度对这个复杂的问题做进一步的讨论。

对于波动方程,传播误差(振幅或相位)经常是复杂的,而对于对流方程,**传导误差**则往往更令人关注。在(19.1.15)式的 Lax 格式中,对流量 u 在网格点 j 处的扰动,将传至下一个时间层的网格点 $j-1$ 和 $j+1$。不过在实际中,如果速度 v 是正的,则仅仅网格点 $j+1$ 会受到影响。

使传导性质得到"改善"的最简单方法是使用**迎风差分法**(见图 19.1.4):

$$\frac{u_j^{n+1}-u_j^n}{\Delta t}=-v_j^n\begin{cases}\dfrac{u_j^n-u_{j-1}^n}{\Delta x}, & v_j^n>0\\ \dfrac{u_{j+1}^n-u_j^n}{\Delta x}, & v_j^n<0\end{cases} \tag{19.1.27}$$

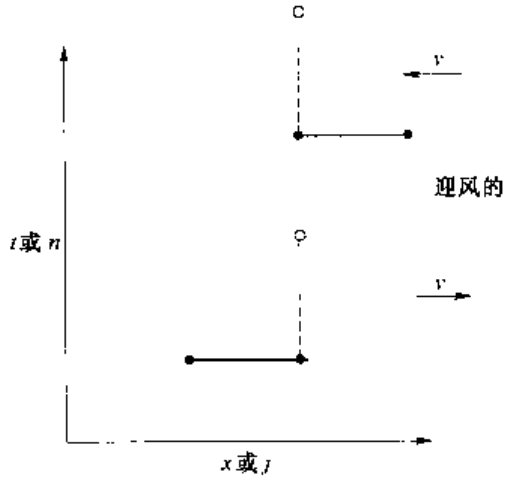


图 19.1.4 迎风差分格式的图示。当对流常数 v 是负的时,如图中所示,上面的格式是稳定的;而当对流常数 v 是正的时,也正如图中所示,下面的格式是稳定的。当然,Courant条件也是必须要满足的

注意,这种格式在空间导数的计算上仅是一阶精度,而不是二阶精度。怎样才能使它"更好"呢?答案令数学家们不太满意:按严格的数学意义,数值模拟的目标并不一定是"精确"的,但在较不严格、更着重实效的意义下,有些时候却能忠实于潜在的物理过程。由于这种原因,某些误差就比其他情形要容易接受得多。对于那些对流变量易于经历状态突变的问题,例如经历激波和其他间断情况那样,迎风差分在总体上更忠实于问题的实质。对这类问题,读者就得根据自己问题的特殊物理性质来具体分析解决。

对于差分格式(19.1.27),放大因子(对常值 v)是

$$\xi=1-\left|\frac{v\Delta t}{\Delta x}\right|(1-\cos k\Delta x)-i\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.28}$$

$$|\xi|^2=1-2\left|\frac{v\Delta t}{\Delta x}\right|\left(1-\left|\frac{v\Delta t}{\Delta x}\right|\right)(1-\cos k\Delta x) \tag{19.1.29}$$

由此可知,稳定性准则 $|\xi|^2\leqslant 1$ 恰好又是 Courant 条件(19.1.17)。

有多种方法可改进一阶迎风差分的精度。在连续性方程中,最初距离 $v\Delta t$ 的物质在 Δt 时间后到达预定的点。在一阶方法中,物质通常从相距 Δx 远的地方而来,如果 $v\Delta t\ll\Delta x$(以确保精度),可能会导致大的误差。一种减少这种误差的方法是,在传输之前,$j-1$ 和 j 之间对 u 进行插值,这就得到了一种有效的二阶方法。二阶迎风差分的各种格式在本节参考文献[2~3]中进行了讨论和比较。

19.1.4 时间域上的二阶精度

在使用一个时间域中一阶精度、空间域中二阶精度的方法时,人们往往不得不使 $v\Delta t$ 远远小于 Δx,例如至少有 $5v\Delta t\leqslant\Delta x$,以获得期望的精度。因而,在实际运用中,Courant 条件实质上并不是这些格式的限制因素。好在有在时间域中和空间域中均为二阶精度的格式,并且,这些格式通常可与它们的稳定性限制相吻

合，故而只用到相对较少的计算次数。

例如，对于守恒方程(19.1.1)的**交错跳点**方法可以定义如下(见图 19.1.5)：利用在时间 t^n 时的值 u^n，计算流量 F_j^n，然后利用流量的时间中心值，计算新值 u^{n+1}：

$$u_j^{n+1} - u_j^{n-1} = -\frac{\Delta t}{\Delta x}(F_{j+1}^n - F_{j-1}^n) \tag{19.1.30}$$

交错跳点的名称取自这样一个事实：时间导数项中的时间层按"跳步"跃过空间导数项中的时间层。这种方法需要存储 u^{n-1} 与 u^n 以计算 u^{n+1}。

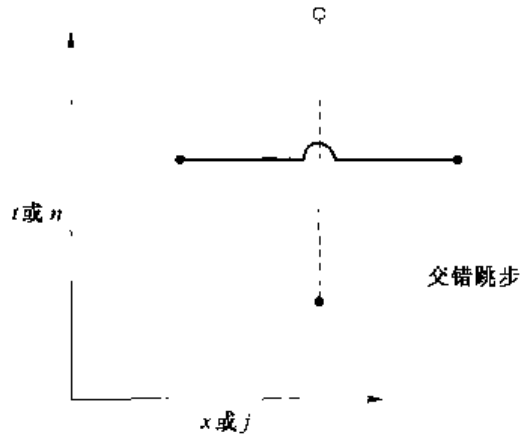


图 19.1.5　交错跳点差分格式的图示。注意，来自先前两个时间层的信息被用于获得期望点，这种格式在时间和空间中均是二阶精度的

对于简单的模型方程(19.1.6)，交错跳点法采用格式：

$$u_j^{n+1} - u_j^{n-1} = -\frac{v\Delta t}{\Delta x}(u_{j+1}^n - u_{j-1}^n) \tag{19.1.31}$$

由于把特征模表达式(19.1.12)代入(19.1.31)式，出现了 ξ 的三个相邻次幂，故此时 von Neumann 稳定性分析给出了 ξ 的一个二次方程，而不是线性一次方程。

$$\xi^2 - 1 = -2i\xi\frac{v\Delta t}{\Delta x}\sin k\Delta x \tag{19.1.32}$$

其解为

$$\xi = -i\frac{v\Delta t}{\Delta x}\sin k\Delta x \pm \sqrt{1 - \left(\frac{v\Delta t}{\Delta x}\sin k\Delta x\right)^2} \tag{19.1.33}$$

可见 Courant 条件再次成为稳定性条件。事实上，在(19.1.33)式中，对任意 $v\Delta t \leqslant \Delta x$，均有 $|\xi|^2 = 1$。这正是交错跳点法的巨大优势：没有振幅的发散。

如果变量处于适当的半网格点上，那么形如(19.1.20)式的交错跳点差分会更加清楚，记

$$\begin{aligned} r_{j+1/2}^n &\equiv v\left.\frac{\partial u}{\partial x}\right|_{j+1/2}^n = v\frac{u_{j+1}^n - u_j^n}{\Delta x} \\ s_j^{n+1/2} &\equiv \left.\frac{\partial u}{\partial t}\right|_j^{n+1/2} = \frac{u_j^{n+1} - u_j^n}{\Delta t} \end{aligned} \tag{19.1.34}$$

这只是一个记号约定，可认为这是定义 r 和 s 的网格，它比定义初始变量 u 的网格加细了一倍。(19.1.20)式的跳步差分为

$$\begin{aligned} \frac{r_{j+1/2}^{n+1} - r_{j+1/2}^n}{\Delta t} &= \frac{s_{j+1}^{n+1/2} - s_j^{n+1/2}}{\Delta x} \\ \frac{s_j^{n+1/2} - s_j^{n-1/2}}{\Delta t} &= v\frac{r_{j+1/2}^n - r_{j-1/2}^n}{\Delta x} \end{aligned} \tag{19.1.35}$$

若将(19.1.22)式代入(19.1.35)式中就会发现，又是 Courant 条件为稳定性所需，而且当该条件被满足后，将不

会出现振幅发散。

如果把(19.1.34)式代入(19.1.35)式中,便会发现(19.1.35)式等价于

$$\frac{u_j^{n+1}-2u_j^n+u_j^{n-1}}{(\Delta t)^2}=v^2\frac{u_{j+1}^n-2u_j^n+u_{j-1}^n}{(\Delta x)^2} \tag{19.1.36}$$

这正是波动方程(19.1.2)的"通常"的二阶差分形式。还发现这是一个双层格式,因为为获得 u^{n+1} 需已知 u^n 和 u^{n-1},这一点在(19.1.35)式中表现为:为求得下一时间层上的解,需知 $s^{n-1/2}$ 和 r^n。

对于比这种简单的模型方程更复杂的方程,特别对于非线性方程,此方法在梯度变大时,通常变得不稳定。这种不稳定性与奇偶网格点的完全解耦现象有关,如同图 19.1.6 所示的一个棋盘的黑白方块,通过一个数值粘性项耦合了奇偶网格点,例如在(19.1.31)式右端加入一个小系数($\ll 1$)乘以 $u_{j+1}^n-2u_j^n+u_{j-1}^n$ 就可以克服这种网格点漂移的不稳定性。至于通过增加数值耗散使差分格式稳定的方法,参见本节参考文献[4]。

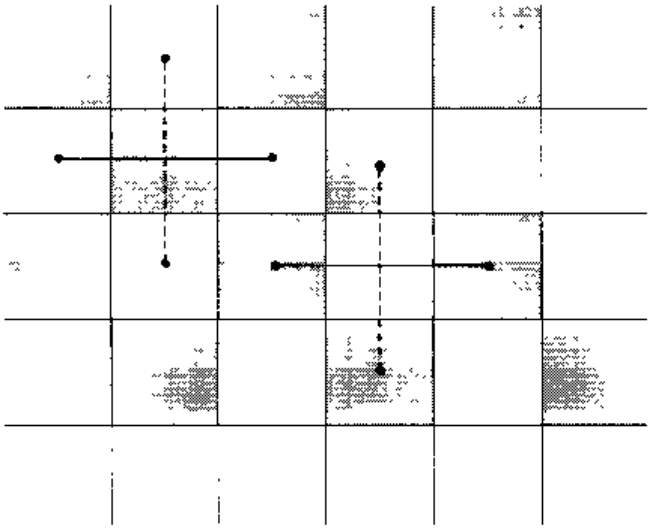


图 19.1.6 在交错跳点格式中网格点漂移不稳定性的起因。如果网格点被假想为放在一个棋盘的方块中,那么白方格与白方格耦合,黑方格与黑方格耦合,却没有黑白方格之间的耦合。修正的方法是引入一个小的扩散的网孔耦合项

两步 Lax-Wendroff 格式是一种在时间域上二阶精度的方法,它可以避免大的数值耗散和网格点飘移。人们在半个时间层 $t_{n+1/2}$ 和半个网格点 $x_{j+1/2}$ 处定义中间值 $x_{j+1/2}$,通过 Lax 格式计算为

$$u_{j+1/2}^{n+1/2}=\frac{1}{2}(u_{j+1}^n+u_j^n)-\frac{\Delta t}{2\Delta x}(F_{j+1}^n-F_j^n) \tag{19.1.37}$$

利用这些变量可以计算通量 $F_{j+1/2}^{n+1/2}$,然后新值 u_j^{n+1} 可通过中心差分格式来计算得到:

$$u_j^{n+1}=u_j^n-\frac{\Delta t}{\Delta x}\left(F_{j+1/2}^{n+1/2}-F_{j-1/2}^{n+1/2}\right) \tag{19.1.38}$$

临时值 $u_{j+1/2}^{n+1/2}$ 现在就被舍弃了(见图 19.1.7)。

让我们考虑一下对于模型对流方程这种方法的稳定性。其中 $F=vu$,把(19.1.37)式代入(19.1.38)式得

$$\begin{aligned}u_j^{n+1}=u_j^n-\alpha\Bigg[&\frac{1}{2}(u_{j+1}^n+u_j^n)-\frac{1}{2}\alpha(u_{j+1}^n-u_j^n)\\&-\frac{1}{2}(u_j^n+u_{j-1}^n)+\frac{1}{2}\alpha(u_j^n-u_{j-1}^n)\Bigg]\end{aligned} \tag{19.1.39}$$

其中

$$\alpha\equiv\frac{v\Delta t}{\Delta x} \tag{19.1.40}$$

由此可得

$$\xi=1-i\alpha\sin k\Delta x-\alpha^2(1-\cos k\Delta x) \tag{19.1.41}$$

故有

$$|\xi|^2=1-\alpha^2(1-\alpha^2)(1-\cos k\Delta x)^2 \tag{19.1.42}$$

因此稳定性准则 $|\xi|^2 \leqslant 1$，即是 $\alpha^2 \leqslant 1$ 或如通常的 $v\Delta t \leqslant \Delta x$。顺便说一下，不能认为 Courant 条件是出现在偏微分方程中的惟一的稳定性条件。之所以在模型例子中它一直是稳定性条件，仅仅因为例子在形式上都太简单了。不过，分析的方法是通用的。

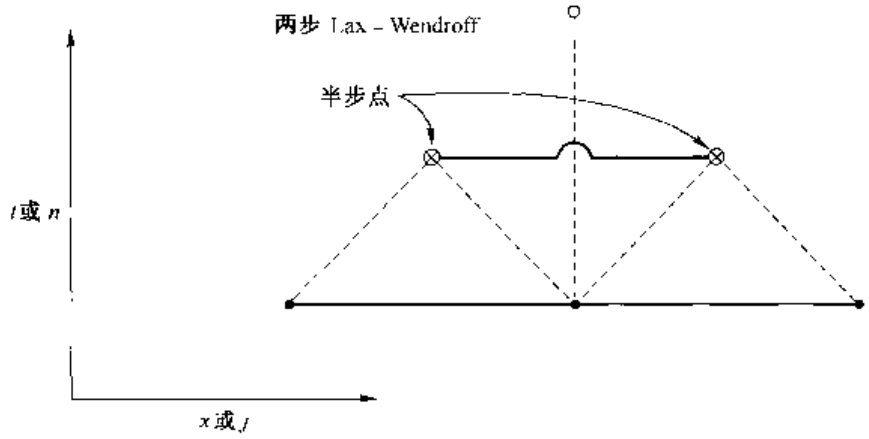


图 19.1.7　两步 Lax-Wendroff 差分格式的图示。两个半层上的点(⊗)用 Lax 方法计算，这两点加上起始点中的一个，运用交错跳点法，可计算新点。半步上的点仅暂时使用，不需要网格点上的存储分配。这种格式在时间和空间上均是二阶精度的

除了 $\alpha = 1$ 的时候，(19.1.42)式均有 $|\xi|^2 < 1$。因此会出现一些振幅衰减。不过由于波长是足以与网格大小 Δx 相比拟，因此这些影响相对较小。如果对小的 $k\Delta x$ 展开(19.1.42)式，可得

$$|\xi|^2 = 1 - \alpha^2(1-\alpha^2)\frac{(k\Delta x)^4}{4} + \ldots \tag{19.1.43}$$

对 1 的偏离仅出现在 k 的 4 次方处。该式应与使用 Lax 方法所得的(19.1.16)式相比，对小的 $k\Delta x$ 有

$$|\xi|^2 = 1 - (1-\alpha^2)(k\Delta x)^2 + \ldots \tag{19.1.44}$$

总的来说，建议对凡可归入流量守恒方程形式的初值问题，特别是与波动方程相关的问题，尽可能使用交错跳点方法，我们认为，使用前者比使用两步 Lax-Wendroff 方法要好。对那些对传导误差敏感的问题，可以考虑使用迎风差分或它的一种改进形式。

19.1.5　含有激波的流体动力学

前面曾提过，含有激波的流体动力学问题的处理是一个非常复杂的课题。在这里只想给读者介绍一些和该课题有关的基本内容。

激波的处理基本上有三个重要的一般性方法。最古老也是最简单的方法是由 von Neumann 和 Richtmyer 提出的，该方法是给方程加上**人工粘性**来模拟真实粘性对不连续体的光滑方式。实现该方法的一个较好的出发点是本节参考文献［1］12.11 节中的差分格式。该格式对几乎所有的一维空间问题都非常适合。

第二种方法是一个高阶差分格式和一个低阶差分格式的结合，前者适用于平稳流，而后者相当耗散并能滤除激波。特别是，各种不同的迎风差分格式的结合使用，除非较大的梯度出现，否则其权数的选取一般能将低阶格式调整为零，而且这些权数的选取也能施加各种"单调性"限制，从而防止数值解中出现非物理性振荡。通过本节参考文献[2,3,5]来研究这些方法，效果一定很好。

第三种，可能也是最有效的一种方法是 Godunov 方法。这里我们抛弃基于泰勒级数的有限差分法中所蕴含的简单线性化思想，而直接处理方程的非线性。对于由一个不连续体分隔的流体的两个均匀状态的演化问题，即 Riemann 激波问题有解析解。Godunov 的想法是，用大量均匀状态的单体来近似地表示流体，并利用 Riemann 解将其接合起来。Godunov 法现在已有很多的推广，其中最有效的可能是 PPM 方法[6]。

对所有这些方法的评述，以及对一维方法推广到多维方法的讨论，请参阅本节参考文献[7～9]。

参考文献和深入阅读

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press), Chapter 4.

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience). [1]

Centrella, J., and Wilson, J.R. 1984, *Astrophysical Journal Supplement*, vol. 54, pp. 229–249, Appendix B. [2]

Hawley, J.F., Smarr, L.L., and Wilson, J.R. 1984, *Astrophysical Journal Supplement*, vol. 55, pp. 211–246, §2c. [3]

Kreiss, H.-O. 1978, *Numerical Methods for Solving Time-Dependent Problems for Partial Differential Equations* (Montreal: University of Montreal Press), pp. 66ff. [4]

Harten, A., Lax, P.D., and Van Leer, B. 1983, *SIAM Review*, vol. 25, pp. 36–61. [5]

Woodward, P., and Colella, P. 1984, *Journal of Computational Physics*, vol. 54, pp. 174–201. [6]

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa). [7]

Woodward, P., and Colella, P. 1984, *Journal of Computational Physics*, vol. 54, pp. 115–173. [8]

Rizzi, A., and Engquist, B. 1987, *Journal of Computational Physics*, vol. 72, pp. 1–69. [9]

19.2 扩散初值问题

回顾抛物型方程模型,一维空间上的扩散方程为

$$\frac{\partial u}{\partial t}=\frac{\partial}{\partial x}\left(D\frac{\partial u}{\partial x}\right) \tag{19.2.1}$$

其中 D 为扩散系数,事实上,该方程是一个在上节讨论过的通量守恒方程,具有 x 方向的流量:

$$F=-D\frac{\partial u}{\partial x} \tag{19.2.2}$$

将假设 $D\geqslant 0$,否则(19.2.1)式具有物理上的不稳定解:一个小的扰动逐渐变得越来越集中,而不是发散开去(不要徒劳地去为一个偏微分方程自身就不稳定的问题去寻找稳定的差分格式)。

尽管(19.2.1)式是一种已经讨论过的形式,但它还是经常被当做一种以其自身方式出现的典型范例。流量(19.2.2)的特殊形式和它的直接推广形式,在实际运用中经常出现。更进一步地,在许多其他情形下我们已经看到,数值粘性和人工粘性也可能引入像(19.2.1)式右端所示的扩散项。

首先考虑 D 是常数时,方程

$$\frac{\partial u}{\partial t}=D\frac{\partial^2 u}{\partial x^2} \tag{19.2.3}$$

可用一种明显的方式进行差分得

$$\frac{u_j^{n+1}-u_j^n}{\Delta t}=D\left[\frac{u_{j+1}^n-2u_j^n+u_{j-1}^n}{(\Delta x)^2}\right] \tag{19.2.4}$$

除了方程右端进行差分的是一个二阶导数以外,上式即又是 FTCS 格式了。FTCS 格式对双曲型方程是不稳定的,而一个快速的计算将表明,(19.2.4)式中的放大因子为

$$\xi=1-\frac{4D\Delta t}{(\Delta x)^2}\sin^2\left(\frac{k\Delta x}{2}\right) \tag{19.2.5}$$

由 $|\xi|\leqslant 1$ 的要求可得稳定性准则:

$$\frac{2D\Delta t}{(\Delta x)^2}\leqslant 1 \tag{19.2.6}$$

限制条件(19.2.6)式的物理解释是:精确到一个数值因子的意义下,允许的最大时间步长是穿过宽度为 Δx 的网格的扩散时间。

更一般地,穿过一个大小为λ的空间区域的扩散时间τ的量级为

$$\tau\sim\frac{\lambda^2}{D} \tag{19.2.7}$$

通常，对精确地模拟$\lambda \gg \Delta x$的空间区域中的特征演变很感兴趣。如果限制时间步长满足(19.2.6)式，在获得感兴趣的区域中的有关信息之前，必须通过量级为$\lambda^2/(\Delta x)^2$的步数来发展。这个步数通常是多得不能运行的。因此，必须寻找一种稳定的方法，其中时间步长可与(19.2.7)式的时间量级相比拟；或者，为精度要求，一定程度地小于(19.2.7)式的时间量级。

这一目标提出了一个直接的"哲学"问题。显然，建议采用的大的时间步长，对于我们不感兴趣的小区域是极不精确的，而我们希望那些区域中的性能稳定"无害"，并且可能的话不应过分地在物理上不合理。我们想把这种无害性能加入差分格式中，它又将是怎样的呢？

有两种不同的答案，每个答案有其自身的优点和缺点。第一种方法是寻找一种差分格式，使小区域中的特征达到其平衡形式，例如，满足左端置为 0 的(19.2.3)式。这种方法一般具有最好的物理意义，但正如我们将看到的，它只能对应一种在我们感兴趣的区域中，关于时间域上是一阶精度的差分格式("全隐式")。第二种方法是，让小范围中的特征保持其初始振幅值，以使得我们感兴趣的较大区域中的特征变化，与小区域中的"冻结"(虽然有起伏)背景相迭加地进行。这种方法提供了一种在时间域上二阶精度的差分格式(Crank-Nicolson 格式)，不过在一个变化过程的计算快结束时，可能希望转换为另一种方法的一些时间层，以使小范围中的状态达到平衡。现在，让我们看一下这些不同的差分方法从何得来。

考虑(19.2.3)式的如下差分式：

$$\frac{u_j^{n+1}-u_j^n}{\Delta t}=D\left[\frac{u_{j+1}^{n+1}-2u_j^{n+1}+u_{j-1}^{n+1}}{(\Delta x)^2}\right] \tag{19.2.8}$$

除了右端的空间导数是计算的第 $n+1$ 时间层外，该式完全同 FTCS 差分格式。这种特征的格式与 FTCS(称为全显式)相对照，被称为**全隐式**或**时间向后**式。为了求解(19.2.8)式，人们不得不在每个时间层解一个联立的线性方程组以求得 u_j^{n+1}。幸运的是，该系统是三对角型的，所以这只是一个简单的问题：只需适当合并(19.2.8)式中的项：

$$-\alpha u_{j-1}^{n+1}+(1+2\alpha)u_j^{n+1}-\alpha u_{j+1}^{n+1}=u_j^n, \qquad j=1,2...J-1 \tag{19.2.9}$$

其中

$$\alpha \equiv \frac{D\Delta t}{(\Delta x)^2} \tag{19.2.10}$$

加上 $j=0$ 和 $j=J$ 处的 Dirichlet 边界条件或 Neumann 边界条件，(19.2.9)式是一个三对角系统，它在每个时间层都很容易用 2.4 节中的方法求解。

对于非常大的时间步长，(19.2.8)式的情形如何呢？当限极$\alpha\to\infty$ ($\Delta t\to\infty$)，在(19.2.9)式中可以看得很清楚：(19.2.9)式两边除以α，我们发现该差分方程就是平衡方程

$$\frac{\partial^2 u}{\partial x^2}=0 \tag{19.2.11}$$

的有限差分形式。

该差分方法稳定性如何？(19.2.8)式的放大因子是

$$\xi=\frac{1}{1+4\alpha\sin^2\left(\dfrac{k\Delta x}{2}\right)} \tag{19.2.12}$$

显然对任意步长 Δt 有$|\xi|<1$，这种格式是无条件稳定的。从初始条件发展变化的小区域的细节，对于大的 Δt 显然是不精确的。但正如前面已经证明的，正确的平衡解已经获得，这也正是隐式方法的特征所在。

另一方面，通过把隐式方法的稳定性和在时间、空间上均二阶精度的方法的精确性综合起来，让我们看看如何满足上述答案的第二个方面。直接把显式和隐式 FTCS 格式平均得

$$\frac{u_j^{n+1}-u_j^n}{\Delta t}=\frac{D}{2}\left[\frac{(u_{j+1}^{n+1}-2u_j^{n+1}+u_{j-1}^{n+1})+(u_{j+1}^n-2u_j^n+u_{j-1}^n)}{(\Delta x)^2}\right] \tag{19.2.13}$$

其中，左边和右边都以时间层 $n+\frac{1}{2}$ 为中心，故这种方法正如前面所称，是时间域上二阶精度的，其放大因子是：

$$\xi = \frac{1 - 2\alpha \sin^2\left(\frac{k\Delta x}{2}\right)}{1 + 2\alpha \sin^2\left(\frac{k\Delta x}{2}\right)} \tag{19.2.14}$$

故这种方法对任意大小的 Δt 均是稳定的。这种格式被称为 **Crank-Nicolson 格式**,也正是我们对任意的简单扩散问题要推荐的格式(可能在最后几层要补充一些全隐式步)(见图 19.2.1)。

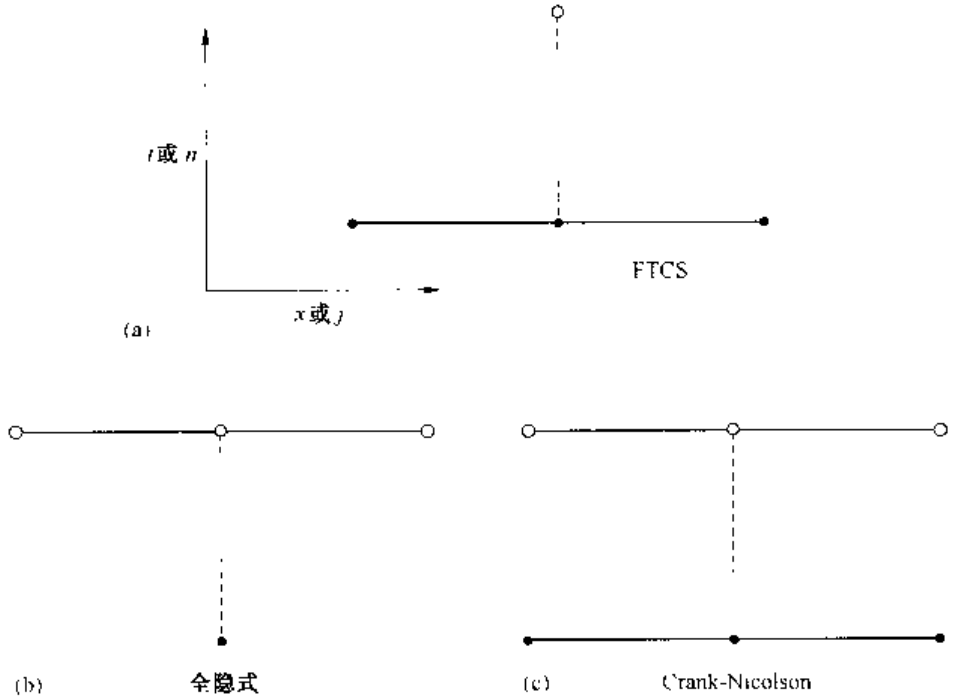


图 19.2.1 扩散问题的三种差分格式(同图 19.1.2 所示)。(a)是时间向前空间中心差分格式,具有一阶精度,但只对足够小的时间步长稳定;(b)是全隐式格式,它对任意大的时间步长均稳定,但仍然只是一阶精度的;(c)是Crank-Nicolson格式,该格式具有二阶精度,且对大的时间步长也通常是稳定的

现在转向简单扩散方程(19.2.3)的一些推广形式。首先,假设扩散系数 D 不是常数,设 $D = D(x)$,可以采纳上述两种方法中的任一种。首先,进行一个变量的解析变换:

$$y = \int \frac{dx}{D(x)} \tag{19.2.15}$$

那么

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x} D(x) \frac{\partial u}{\partial x} \tag{19.2.16}$$

变成了

$$\frac{\partial u}{\partial t} = \frac{1}{D(y)} \frac{\partial^2 u}{\partial y^2} \tag{19.2.17}$$

对适当的 y_j 可以估算 D。由此可得,显式格式中的稳定性准则(19.2.6)式变成了

$$\Delta t \leqslant \min_j \left[\frac{(\Delta y)^2}{2D_j^{-1}}\right] \tag{19.2.18}$$

注意,沿 y 是常数步长 Δy,并不意味着沿 x 也是常数步长。

一种并不要求 D 易解析处理形式的替代方法是,对扩散方程(19.2.16)很简单地差分,在该方法中,对每个变量适当地集中,从而 FTCS 格式变成了

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = \frac{D_{j+1/2}(u_{j+1}^n - u_j^n) - D_{j-1/2}(u_j^n - u_{j-1}^n)}{(\Delta x)^2} \tag{19.2.19}$$

其中

$$D_{j+1/2} \equiv D(x_{j+1/2}) \tag{19.2.20}$$

相应的稳定性准则是

$$\Delta t \leqslant \min_j \left[\frac{(\Delta x)^2}{2D_{j+1/2}}\right] \tag{19.2.21}$$

故 Crank-Nicolson 方法可以被很类似地推广。

第二种可以考虑的复杂情况是非线性扩散问题，例如方程 $D = D(u)$。显式格式可用简单的方法推广，如在(19.2.19)式中，记

$$D_{j+1/2} = \frac{1}{2}\left[D(u_{j+1}^n) + D(u_j^n)\right] \tag{19.2.22}$$

隐格式就不这么简单了。(19.2.22)式中用 $n \to n+1$ 的替换，在每个时间层都留给我们一组难以处理的耦合的非线性方程，通常有一种更简单的方法：设 $D(u)$ 的形式允许我们对 $z(u)$ 解析地积分：

$$dz = D(u)du \tag{19.2.23}$$

(19.2.1)式的右端将变为 $\partial^2 z/\partial x^2$，可对之隐式差分为

$$\frac{z_{j+1}^{n+1} - 2z_j^{n+1} + z_{j-1}^{n+1}}{(\Delta x)^2} \tag{19.2.24}$$

现在对(19.2.24)式右端的每项线性化，例如

$$\begin{aligned} z_j^{n+1} \equiv z(u_j^{n+1}) &= z(u_j^n) + (u_j^{n+1} - u_j^n)\left.\frac{\partial z}{\partial u}\right|_{j,n} \\ &= z(u_j^n) + (u_j^{n+1} - u_j^n)D(u_j^n) \end{aligned} \tag{19.2.25}$$

这使问题简化成三对角形式，而且在实际问题中，通常也保持了隐式差分的稳定性优势。

19.2.1　薛定谔方程

有时，尚待解决的物理问题给还未考虑的差分格式添加了限制。比如，在量子学中，依赖于时间的薛定谔(Schrödinger)方程，这是一个关于复数量 ψ 演变的典型双曲型方程。由于每个波群以一维位势 $V(x)$ 发散，方程具有形式

$$i\frac{\partial \psi}{\partial t} = -\frac{\partial^2 \psi}{\partial x^2} + V(x)\psi \tag{19.2.26}$$

(在此选择了单位使普朗克(Planck)常数 $\hbar = 1$，质点质量 $m = 1/2$。)已知初始波群 $\psi(x, t=0)$，并已知边界条件为 $x \to \pm\infty$ 时，$\psi \to 0$。假设满足于时间域上一阶精度，但希望使用隐格式以求稳定，则将(19.2.8)式稍做推广得

$$i\left[\frac{\psi_j^{n+1} - \psi_j^n}{\Delta t}\right] = -\left[\frac{\psi_{j+1}^{n+1} - 2\psi_j^{n+1} + \psi_{j-1}^{n+1}}{(\Delta x)^2}\right] + V_j\psi_j^{n+1} \tag{19.2.27}$$

其中

$$\xi = \frac{1}{1 + i\left[\dfrac{4\Delta t}{(\Delta x)^2}\sin^2\left(\dfrac{k\Delta x}{2}\right) + V_j\Delta t\right]} \tag{19.2.28}$$

这是无条件稳定的，但遗憾的却不是幺正的。物理问题内在要求，在某处发现微粒的总概率应保持为单位 1。这一点可以通过 ψ 的模平方范数保持为 1 来正式地表示：

$$\int_{-\infty}^{\infty} |\psi|^2 dx = 1 \tag{19.2.29}$$

将初始波函数 $\psi(x, 0)$ 归一化以满足(19.2.29)式。于是，薛定谔方程(19.2.26)便保证了在以后任意时刻这一条件均满足。

让我们把(19.2.26)式写成如下形式：

$$i\frac{\partial\psi}{\partial t}=H\psi \tag{19.2.30}$$

其中算子 H 为

$$H=-\frac{\partial^2}{\partial x^2}+V(x) \tag{19.2.31}$$

(19.2.30)式的形式解为

$$\psi(x,t)=e^{-iHt}\psi(x,0) \tag{19.2.32}$$

其中算子的指数形式由其幂级数展开来定义。

用不稳定的显式 FTCS 格式近似表示(19.2.32)式为

$$\psi_j^{n+1}=(1-iH\Delta t)\psi_j^n \tag{19.2.33}$$

其中 H 用沿 x 的中心有限差分的逼近来表示。相比之下,稳定的隐格式(19.2.27)式为

$$\psi_j^{n+1}=(1+iH\Delta t)^{-1}\psi_j^n \tag{19.2.34}$$

通过展开(19.2.32)式可以发现,这两种格式均在时间域上是一阶精度的,但(19.2.33)式中的算子和(19.2.34)式中的算子都不是幺正的。

对薛定谔方程差分的正确方法是,采用 Cayley 格式给出 e^{-iHt} 的有限差分表示,这种格式是二阶精度的,并且是幺正的:

$$e^{-iHt}\simeq\frac{1-\frac{1}{2}iH\Delta t}{1+\frac{1}{2}iH\Delta t} \tag{19.2.35}$$

换言之,有

$$\left(1+\tfrac{1}{2}iH\Delta t\right)\psi_j^{n+1}=\left(1-\tfrac{1}{2}iH\Delta t\right)\psi_j^n \tag{19.2.36}$$

此时一旦用沿 x 的有限差分的近似来代替 H,便得到一个待解的复三对角系统。这种求解薛定谔方程的方法是稳定的、幺正的,并且在时间和空间上均是二阶精度的。事实上,这种方法恰好就是 Crank-Nicolson 方法。

参考文献和深入阅读

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press), Chapter 2.

Goldberg, A., Schey, H.M., and Schwartz, J.L. 1967, *American Journal of Physics*, vol. 35, pp. 177–186. [1]

Galbraith, I., Ching, Y.S., and Abraham, E. 1984, *American Journal of Physics*, vol. 52, pp. 60–68. [2]

19.3 多维初值问题

在 19.1 节和 19.2 节中介绍的针对 1+1 维(一维空间和一维时间)问题的方法,可以很容易地推广到 $N+1$ 维空间,但是,求解相应方程所需的计算量却是巨大的。如果用 100 个空间网格点解决了一个一维问题,那么,解决具有 100×100 个网格点的二维问题至少需要多于 100 倍的计算量。因此,对多维问题,通常不得不满足于非常勉强的空间解。

让我们详细介绍一点关于求解和检验多维偏微分方程的解的经验:通常读者应在非常小的网格上,如 8×8,运行和检验自己的程序,即便相应结果的精度很差。只有程序被全面调试并证明了稳定性,才可以增大网格点的大小到一个适当的程度,并开始观察其结果。实际上有人说:程序对于粗糙的网格可能是不稳定的,但确信这种不稳定性在较大网格中将会消失。这种观点非常荒谬,它体现了对精度和稳定性概念的完全混淆。事实上,倒是新的不稳定性有时会在较大网格中出现,而旧的不稳定性却绝对不会(根据我们的经验)消失。

由于不得不采用中等大小的网格,有些人建议运用更高阶的方法,以力图改进精度,这是非常危险的。除非欲求的解已知是平滑的,并且所要用的更高阶方法已知是非常稳定的,否则绝不能建议采用任何时间域

中高于二阶的方法(对一系列的一阶方程组)。对于空间差分,建议采用原偏微分方程自身的阶数,可能也允许对空间上的一阶偏微分方程采用二阶空间差分。当提高一种差分方法的阶数,使之高于原偏微分方程的阶数时,将会对差分方程引入伪解。如果伪解恰巧都是指数衰减的,那不会有什么问题;反之,将面临一片混乱。

19.3.1　通量守恒方程的 Lax 方法

作为一个例子,让我们看看如何把 Lax 方法(19.1.15)式推广到二维守恒律方程:

$$\frac{\partial u}{\partial t} = -\nabla \cdot \mathbf{F} = -\left(\frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y}\right) \tag{19.3.1}$$

采用空间网格:

$$\begin{aligned} x_j &= x_0 + j\Delta \\ y_l &= y_0 + l\Delta \end{aligned} \tag{19.3.2}$$

为简单起见,选择了 $\Delta x = \Delta y \equiv \Delta$,故 Lax 格式为

$$\begin{aligned} u_{j,l}^{n+1} = &\frac{1}{4}(u_{j+1,l}^n + u_{j-1,l}^n + u_{j,l+1}^n + u_{j,l-1}^n) \\ &- \frac{\Delta t}{2\Delta}(F_{j+1,l}^n - F_{j-1,l}^n + F_{j,l+1}^n - F_{j,l-1}^n) \end{aligned} \tag{19.3.3}$$

注意,作为缩写记法,$F_{j+1,l}$ 和 F_{j-1} 对应 F_x,而 F_{l+1} 和 F_{l-1} 对应 F_y。

让我们对这个对流方程模型(类似(19.1.6)式)进行稳定性分析,设

$$F_x = v_x u, \qquad F_y = v_y u \tag{19.3.4}$$

这使每个特征模虽然在时间上仍简单地依赖于 ξ 的方幂,但在空间中却是两维数了:

$$u_{j,l}^n = \xi^n e^{ik_x j\Delta} e^{ik_y l\Delta} \tag{19.3.5}$$

代入(19.3.3)式,得到

$$\xi = \frac{1}{2}(\cos k_x\Delta + \cos k_y\Delta) - i\alpha_x \sin k_x\Delta - i\alpha_y \sin k_y\Delta \tag{19.3.6}$$

其中

$$\alpha_x = \frac{v_x\Delta t}{\Delta}, \qquad \alpha_y = \frac{v_y\Delta t}{\Delta} \tag{19.3.7}$$

$|\xi|^2$ 的表达式可以被整理成如下形式:

$$\begin{aligned} |\xi|^2 = &1 - (\sin^2 k_x\Delta + \sin^2 k_y\Delta)\left[\frac{1}{2} - (\alpha_x^2 + \alpha_y^2)\right] \\ &- \frac{1}{4}(\cos k_x\Delta - \cos k_y\Delta)^2 - (\alpha_y \sin k_x\Delta - \alpha_x \sin k_y\Delta)^2 \end{aligned} \tag{19.3.8}$$

最后两项是负的,所以稳定性要求 $|\xi|^2 \leqslant 1$ 成为

$$\frac{1}{2} - (\alpha_x^2 + \alpha_y^2) \geqslant 0 \tag{19.3.9}$$

或者

$$\Delta t \leqslant \frac{\Delta}{\sqrt{2}(v_x^2 + v_y^2)^{1/2}} \tag{19.3.10}$$

下面是 N 维 Courant 条件的一般结果:设 $|v|$ 是该问题中最大的传播速度,则称

$$\Delta t \leqslant \frac{\Delta}{\sqrt{N}|v|} \tag{19.3.11}$$

为 Courant 条件。

19.3.2　多维的扩散问题

让我们考虑一下二维扩散方程

$$\frac{\partial u}{\partial t} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{19.3.12}$$

的一种显式方法,如 FTCS,可以从一维情形很容易地进行推广。然而,已知扩散问题通常最好是隐式地处理。假设力图使用二维 Crank-Nicolson 格式,将有

$$u_{j,l}^{n+1} = u_{j,l}^{n} + \frac{1}{2}\alpha\left(\delta_x^2 u_{j,l}^{n+1} + \delta_x^2 u_{j,l}^{n} + \delta_y^2 u_{j,l}^{n+1} + \delta_y^2 u_{j,l}^{n}\right) \tag{19.3.13}$$

其中

$$\alpha \equiv \frac{D\Delta t}{\Delta^2} \qquad \Delta \equiv \Delta x = \Delta y \tag{19.3.14}$$

$$\delta_x^2 u_{j,l}^{n} \equiv u_{j+1,l}^{n} - 2u_{j,l}^{n} + u_{j-1,l}^{n} \tag{19.3.15}$$

对 $\delta_y^2 u_{j,l}^{n}$ 有类似的式子。这确实是一种可行的格式,而问题出现在求解耦合的线性方程组。尽管矩阵仍是非常稀疏的,但一维空间中的三对角型系统却不复存在了。一种可能性是使用适当的稀疏矩阵方法(见 2.7 节和 19.0 节)。

另一种可能的解法,也是我们一般更喜欢使用的,是对一般 Crank-Nicolson 算法的稍微变形。它仍是时间和空间上二阶精度的,并且是无条件稳定的,但其方程比起(19.3.13)式要容易解些。这种格式被称为**交替方向隐式法**(ADI),其中包含了算子分裂或时间分裂两这种非常有用的思想,将在后面讨论更多细节。这种思想是旨在把每个时间层以 $\Delta t/2$ 的大小分裂成两层,在每个子层,沿不同方向进行隐式处理。

$$\begin{aligned} u_{j,l}^{n+1/2} &= u_{j,l}^{n} + \frac{1}{2}\alpha\left(\delta_x^2 u_{j,l}^{n+1/2} + \delta_y^2 u_{j,l}^{n}\right) \\ u_{j,l}^{n+1} &= u_{j,l}^{n+1/2} + \frac{1}{2}\alpha\left(\delta_x^2 u_{j,l}^{n+1/2} + \delta_y^2 u_{j,l}^{n+1}\right) \end{aligned} \tag{19.3.16}$$

这种方法的优点是,在每个子层仅需要求解一个简单的三对角型系统。

19.3.3 一般算子分裂法

算子分裂,也称为**时间分裂**或**分数步法**,其中心思想是,假定有一个初值方程形为

$$\frac{\partial u}{\partial t} = \mathcal{L}u \tag{19.3.17}$$

其中 $\mathcal{L}$ 是某个算子,虽然 $\mathcal{L}$ 不一定必须是线性的,但假定它至少可以写成 m 个线性部分的和,叠加作用到 u 上,

$$\mathcal{L}u = \mathcal{L}_1 u + \mathcal{L}_2 u + \cdots + \mathcal{L}_m u \tag{19.3.18}$$

最后,假定对于每个线性部分,有一种差分格式从时间步 n 到时间步 $n+1$ 对变量 u 进行更新,而且只要该线性部分是右端惟一的一项,这种差分格式便可使用。将这些更新形式写为

$$\begin{aligned} u^{n+1} &= \mathcal{U}_1(u^n, \Delta t) \\ u^{n+1} &= \mathcal{U}_2(u^n, \Delta t) \\ &\cdots \\ u^{n+1} &= \mathcal{U}_m(u^n, \Delta t) \end{aligned} \tag{19.3.19}$$

现在,算子分裂的一种形式通过从 n 到 $n+1$ 的下述更新序列得到:

$$\begin{aligned} u^{n+(1/m)} &= \mathcal{U}_1(u^n, \Delta t) \\ u^{n+(2/m)} &= \mathcal{U}_2(u^{n+(1/m)}, \Delta t) \\ &\cdots \\ u^{n+1} &= \mathcal{U}_m(u^{n+(m-1)/m}, \Delta t) \end{aligned} \tag{19.3.20}$$

例如,一个复合对流-扩散过程方程

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} + D\frac{\partial^2 u}{\partial x^2} \tag{19.3.21}$$

对该方程中的对流项采用一种显式格式，而对扩散项采用 Crank-Nicolson 或其他隐格式可能会很有利。

交替方向隐式(ADI)方法，参见(19.3.16)式，是一个稍微不同的算子分裂的示例。让我们重新解释一下(19.3.19)式，使其具有一个不同的含义：令$\mathcal{U}_1$表示一种更新方法，这种方法包括整个算子$\mathcal{L}$的所有代数部分，而它只期望对$\mathcal{L}_1$部分稳定；其余$\mathcal{U}_2,\ldots,\mathcal{U}_m$与$\mathcal{U}_1$类似。那么，从$u^n$到$u^{n+1}$得出的一种新方法是

$$\begin{aligned} u^{n+1/m} &= \mathcal{U}_1(u^n, \Delta t/m) \\ u^{n+2/m} &= \mathcal{U}_2(u^{n+1/m}, \Delta t/m) \\ &\cdots \\ u^{n+1} &= \mathcal{U}_m(u^{n+(m-1)/m}, \Delta t/m) \end{aligned} \tag{19.3.22}$$

现在(19.3.22)式中，每个分数步的时间步长只是整个时间步长的$1/m$，因为每个偏微分运算作用在原始算子的所有项上。

对于算子$\mathcal{L}$，(19.3.22)式作为一个差分格式虽然不总是但也通常都是稳定的。事实上根据经验，通常只要使空间导数最高的算子项具有稳定的$\mathcal{U}_i$(其他$\mathcal{U}_i$可以不稳定)就能使整个格式稳定。

正因为这一点，讨论将从初值问题转向边值问题，下面将讨论这些内容。

参考文献和深入阅读

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press).

19.4　边值问题的傅里叶方法和循环约简法

正如在 19.0 节所讨论的，大多数边值问题(如椭圆型方程)可约简为求解如下形式的，大型稀疏线性系统：

$$\mathbf{A}\cdot\mathbf{u}=\mathbf{b} \tag{19.4.1}$$

或是对线性的边界值方程一次性求解，或是对非线性的边界值方程迭代求解。

当稀疏矩阵具有某种常见的确定形式时，两种重要的技巧可"快速"求得(19.4.1)式的解，**傅里叶变换法**直接适用于方程具有空间域上的常数系数时；而**循环约简法**则在一定程度更具普遍意义，它的适用性与方程是否可分(在"变量分裂"的意义上)这个问题有关。两种方法都要求边界与坐标轴重合。最后，对一些问题，有一种解法是对这两种方法的极有效的组合，称为 **FACR(傅里叶分析和循环约简)**方法。下面依次讨论每种方法，并使用带有限差分表达式(19.0.6)的(19.0.3)式作为典型范例。一般地说，本节的方法在使用时，比将在 19.5 节讨论的简单松弛法速度快，但却不一定比在 19.6 节介绍的较为复杂的多重网格松弛法速度快。

19.4.1　傅里叶变换方法

关于 x 和 y 的离散傅里叶逆变换为

$$u_{jl}=\frac{1}{JL}\sum_{m=0}^{J-1}\sum_{n=0}^{L-1}\widehat{u}_{mn}e^{-2\pi ijm/J}e^{-2\pi iln/L} \tag{19.4.2}$$

该式可用 FFT 在每维空间中进行独立地计算，或者运用 12.4 节的程序 fourn 或 12.5 节的程序 rlft3 直接计算。同样有

$$\rho_{jl}=\frac{1}{JL}\sum_{m=0}^{J-1}\sum_{n=0}^{L-1}\widehat{\rho}_{mn}e^{-2\pi ijm/J}e^{-2\pi iln/L} \tag{19.4.3}$$

如果把表达式(19.4.2)和(19.4.3)代入模型问题(19.0.6)式中，将有

$$\widehat{u}_{mn}\left(e^{2\pi im/J}+e^{-2\pi im/J}+e^{2\pi in/L}+e^{-2\pi in/L}-4\right)=\widehat{\rho}_{mn}\Delta^2 \tag{19.4.4}$$

或者

$$\widehat{u}_{mn}=\frac{\widehat{\rho}_{mn}\Delta^2}{2\left(\cos\frac{2\pi m}{J}+\cos\frac{2\pi n}{L}-2\right)} \tag{19.4.5}$$

故运用 FFT 方法求解(19.0.6)式的步骤为:

- 用傅里叶变换计算 $\widehat{\rho}_{mn}$,

$$\widehat{\rho}_{mn}=\sum_{j=0}^{J-1}\sum_{l=0}^{L-1}\rho_{jl}e^{2\pi imj/J}e^{2\pi inl/L} \tag{19.4.6}$$

- 从(19.4.5)式中计算 $\widehat{u}_{mn}$。
- 通过傅里叶逆变换(19.4.2)式计算 u_{jl}。

上述过程对周期性边界问题适用,换言之,解满足:

$$u_{jl}=u_{j+J,l}=u_{j,l+L} \tag{19.4.7}$$

下面考虑在矩形边界上,Dirichlet 边界条件 $u=0$ 的情形。此时,需要用一个正弦波的展开式代替展开式(19.4.2):

$$u_{jl}=\frac{2}{J}\frac{2}{L}\sum_{m=1}^{J-1}\sum_{n=1}^{L-1}\widehat{u}_{mn}\sin\frac{\pi jm}{J}\sin\frac{\pi ln}{L} \tag{19.4.8}$$

它满足在 $j=0,J$ 处和 $l=0,L$ 处,$u=0$ 的边界条件,如果把这个展开式和关于 ρ_{jl} 的类似的展开式代入(19.0.6)式中,将发现相应的求解过程与原周期性边界问题的求解过程是平行的:

- 通过正弦变换计算 $\widehat{\rho}_{mn}$,

$$\widehat{\rho}_{mn}=\sum_{j=1}^{J-1}\sum_{l=1}^{L-1}\rho_{jl}\sin\frac{\pi jm}{J}\sin\frac{\pi ln}{L} \tag{19.4.9}$$

(快速正弦变换算法在 12.3 节中给出。)

- 利用类似于(19.4.5)式的表达式计算 $\widehat{u}_{mn}$,

$$\widehat{u}_{mn}=\frac{\Delta^2\widehat{\rho}_{mn}}{2\left(\cos\frac{\pi m}{J}+\cos\frac{\pi n}{L}-2\right)} \tag{19.4.10}$$

- 运用逆正弦变换(19.4.8)计算 u_{jl}。

如果边界条件是非齐次的,例如除了边界 $x=J\Delta$ 处,$u=f(y)$,其他所有边界处满足 $u=0$,此时可在上述解中加入一个解 u^H,它满足所要求的边界条件的齐次方程

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0 \tag{19.4.11}$$

在连续的情形下,该解将具有如下形式的表达式:

$$u^H=\sum_n A_n\sinh\frac{n\pi x}{L\Delta}\sin\frac{n\pi y}{L\Delta} \tag{19.4.12}$$

其中 A_n 可以通过 $x=J\Delta$ 处 $u=f(y)$来确定,在离散情形下,有

$$u_{jl}^H=\frac{2}{L}\sum_{n=1}^{L-1}A_n\sinh\frac{\pi nj}{L}\sin\frac{\pi nl}{L} \tag{19.4.13}$$

如果 $f(y=l\Delta)\equiv f_l$,则从逆公式可得 A_n:

$$A_n=\frac{1}{\sinh(\pi nJ/L)}\sum_{l=1}^{L-1}f_l\sin\frac{\pi nl}{L} \tag{19.4.14}$$

问题的完全解为

$$u = u_{jl} + u_{jl}^{H} \tag{19.4.15}$$

通过加入形如(19.4.12)式的适当项,可以处理任何边界面上的非齐次项。

一种处理非齐次项的更简单方法是,注意凡边界项出现在(19.0.6)式的左端时,由于它们是已知的,故总可以被移到方程的右端,所以有效的源项是 ρ_{jl} 加上一个边界项。

为了形式上实现该想法,将解写为

$$u = u' + u^{B} \tag{19.4.16}$$

其中在边界上 $u' = 0$,而 u^{B} 是除边界外其余处均取零。在边界上,u^{B} 取给定的边界值。在上述示例中,u^{B} 的非零值将只有

$$u_{J,l}^{B} = f_l \tag{19.4.17}$$

模型方程(19.0.3)变为

$$\nabla^2 u' = -\nabla^2 u^{B} + \rho \tag{19.4.18}$$

或者,以有限差分的形式表示为

$$\begin{aligned} u'_{j+1,l} + u'_{j-1,l} + u'_{j,l+1} + u'_{j,l-1} - 4u'_{j,l} = \\ -(u^{B}_{j+1,l} + u^{B}_{j-1,l} + u^{B}_{j,l+1} + u^{B}_{j,l-1} - 4u^{B}_{j,l}) + \Delta^2 \rho_{j,l} \end{aligned} \tag{19.4.19}$$

(19.4.19)式中,由于所有 u^{B} 项除了 $j = J-1$ 处之外均为零,故当 $j = J-1$ 时,得

$$u'_{J,l} + u'_{J-2,l} + u'_{J-1,l+1} + u'_{J-1,l-1} - 4u'_{J-1,l} = -f_l + \Delta^2 \rho_{J-1,l} \tag{19.4.20}$$

因此该问题现在与零边界条件等价,但源项的一行需进行下面的替换:

$$\Delta^2 \rho_{J-1,l} \to \Delta^2 \rho_{J-1,l} - f_l \tag{19.4.21}$$

Neumann 边界条件 $\nabla u = 0$ 的情形,可通过余弦展开式(12.3.17)来处理:

$$u_{jl} = \frac{2}{J}\frac{2}{L}\sum_{m=0}^{J}{}'' \sum_{n=0}^{L}{}'' \hat{u}_{mn} \cos\frac{\pi j m}{J} \cos\frac{\pi l n}{L} \tag{19.4.22}$$

这里双撇号表示 $m = 0$ 及 $m = J$ 的项应该乘以 $\frac{1}{2}$,$n = 0$ 及 $n = L$ 也类似。非齐次项 $\nabla u = g$,通过加上齐次方程的一个合适的解被包含进去,或更简单地将边界项移到右端来实现。例如,条件

$$\frac{\partial u}{\partial x} = g(y) \qquad \text{at} \quad x = 0 \tag{19.4.23}$$

变为

$$\frac{u_{1,l} - u_{-1,l}}{2\Delta} = g_l \tag{19.4.24}$$

其中 $g_l \equiv g(y = l\Delta)$。再次将解写为(19.4.16)式的形式,其中在边界上 $\nabla u' = 0$。此时 ∇u^{B} 在边界上取所规定的值,而 u^{B} 除边界之外均取为零。

因此(19.4.24)式给出

$$u_{-1,l}^{B} = -2\Delta g_l \tag{19.4.25}$$

在(19.4.19)式中,所有 u^{B} 项除了当 $j = 0$ 以外均为零,得

$$u'_{1,l} + u'_{-1,l} + u'_{0,l+1} + u'_{0,l-1} - 4u'_{0,l} = 2\Delta g_l + \Delta^2 \rho_{0,l} \tag{19.4.26}$$

因此 u' 是零梯度问题的解,但源项需进行下面的替换:

$$\Delta^2 \rho_{0,l} \to \Delta^2 \rho_{0,l} + 2\Delta g_l \tag{19.4.27}$$

有时 Neumann 边界条件是通过利用一个交错网格来处理的,u 定义在区域边界的中间,它们的一阶导数集中在网格点上。如果采用余弦变换的其他形式,例如(12.3.23)式,则利用类似于上述的技巧,同样可以解决这类问题。

19.4.2 循环约简法

显然,FFT方法仅当初始偏微分方程具有常系数,并且边界与坐标轴相吻合时才适用。一种可用于稍微更为一般的方程的算法称为**循环约简法(CR)**。

以方程

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+b(y)\frac{\partial u}{\partial y}+c(y)u=g(x,y) \tag{19.4.28}$$

为例来阐述循环约简法。在实际应用中,这种方程形式经常由极坐标、柱坐标或球坐标体系下的Helmholtz或者泊松方程得到。更一般的可分离方程在本节参考文献[1]中介绍。

(19.4.28)式的有限差分格式可以写成一组向量方程的形式:

$$\mathbf{u}_{j-1}+\mathbf{T}\cdot\mathbf{u}_j+\mathbf{u}_{j+1}=\mathbf{g}_j\Delta^2 \tag{19.4.29}$$

其中下标 j 取自沿 x 方向的差分,而 y 方向的差分(由前面约定的下标 l 标记)则保留在向量中。矩阵 $\mathbf{T}$ 的形式为

$$\mathbf{T}=\mathbf{B}-2\mathbf{I} \tag{19.4.30}$$

其中 $2\mathbf{I}$ 取自 x 方向的差分,矩阵 $\mathbf{B}$ 取自 y 方向的差分。矩阵 $\mathbf{B}$ 以及由此而得的矩阵 $\mathbf{T}$,均是带有可变系数的三对角型矩阵。

CR方法通过写出像(19.4.29)式的三个相邻方程导出:

$$\begin{aligned}\mathbf{u}_{j-2}+\mathbf{T}\cdot\mathbf{u}_{j-1}+\mathbf{u}_j&=\mathbf{g}_{j-1}\Delta^2\\ \mathbf{u}_{j-1}+\mathbf{T}\cdot\mathbf{u}_j+\mathbf{u}_{j+1}&=\mathbf{g}_j\Delta^2\\ \mathbf{u}_j+\mathbf{T}\cdot\mathbf{u}_{j+1}+\mathbf{u}_{j+2}&=\mathbf{g}_{j+1}\Delta^2\end{aligned} \tag{19.4.31}$$

用矩阵 $\mathbf{T}$ 乘以中间的方程,并把这三个方程加起来,得

$$\mathbf{u}_{j-2}+\mathbf{T}^{(1)}\cdot\mathbf{u}_j+\mathbf{u}_{j+2}=\mathbf{g}_j^{(1)}\Delta^2 \tag{19.4.32}$$

这是一个与(19.4.29)式有相同形式的方程,其中

$$\begin{aligned}\mathbf{T}^{(1)}&=2\mathbf{I}-\mathbf{T}^2\\ \mathbf{g}_j^{(1)}&=\Delta^2(\mathbf{g}_{j-1}-\mathbf{T}\cdot\mathbf{g}_j+\mathbf{g}_{j+1})\end{aligned} \tag{19.4.33}$$

通过一层CR处理,将方程的个数减少一半;由于所得方程与原方程有相同的形式,所以可以重复上过程。为简便起见,取网格上数目为2的方幂,最后结束在一个关于变量中心线的单个方程上:

$$\mathbf{T}^{(f)}\cdot\mathbf{u}_{J/2}=\Delta^2\mathbf{g}_{J/2}^{(f)}-\mathbf{u}_0-\mathbf{u}_J \tag{19.4.34}$$

该方程中,由于 $\mathbf{u}_0$ 和 $\mathbf{u}_J$ 是已知的边界值,把它们写到右边。(19.4.34)式可用标准三对角算法求得 $\mathbf{u}_{J/2}$ 在 $f-1$ 层的两个方程含 $\mathbf{u}_{J/4}$ 和 $\mathbf{u}_{3J/4}$,含 $\mathbf{u}_{J/4}$ 的方程用到 $\mathbf{u}_0$ 和 $\mathbf{u}_{J/2}$,两个都是已知的,故可用通常的三对角算法求解。在每一层都有相似的情形,因此共需解决 $J-1$ 个三对角型系统。

在实际应用中,(19.4.33)式应该改写,以避免数值的不稳定性。对于这些问题和其他实际应用中的细节,请参阅本节参考文献[2]。

19.4.3 FACR方法

求解形如(19.4.28)式的方程,包括常系数问题(19.0.3),最佳方法是傅里叶分析和循环约简的联合使用,即FACR方法(参见本节参考文献[3~6])。如果在CR方法的第 r 步,对形如(19.4.32)式的方程沿 y 方向进行傅里叶分析,即对被压缩的向量用傅里叶变换,将得到一个 x 方向上关于每个 y 傅里叶模的三对角型系统:

$$\hat{u}_{j-2^r}^k+\lambda_k^{(r)}\hat{u}_j^k+\hat{u}_{j+2^r}^k=\Delta^2 g_j^{(r)k} \tag{19.4.35}$$

其中 $\lambda_k^{(r)}$ 是 $\mathbf{T}^{(r)}$ 中对应于第 k 个傅里叶模的特征值。对于(19.0.3)式,(19.4.5)式表明 $\lambda_k^{(r)}$ 将含有像 $\cos(2\pi k/L)-2$ 这样的自乘项。在 $j=2^r,2\times2^r,4\times2^r,\ldots,J-2^r$ 层,求解三对角型系统,得 $\hat{u}_j^k$。再用傅里叶

方法综合得到在这些 x 线上的 y 值，然后像在先前的 CR 算法一样，填充中间的 x 线

这种方法的技巧在于，适当选取 CR 的层数，使算法的操作步骤可达到最少。可以想像，对于一个典型的 128×128 网格情形，理想的层数 $r=2$；渐近地有 $r\to\log_2(\log_2 J)$。

对(19.0.3)式使用上述算法在运行时间上的粗略估计如下：FFT 方法(沿 x 方向和 y 方向)和 CR 方法基本上可以相比拟；FACR 方法在取 $r=0$(即在一维方向使用 FFT，在另一维方向用通常算法求解三对角线方程组)时，可在速度上约提高两倍。取 $r=2$ 的最优 FACR 方法则可以在速度上再提高两倍。

参考文献和深入阅读

Swartzrauber, P.N. 1977, *SIAM Review*, vol. 19, pp. 490–501. [1]

Buzbee, B.L., Golub, G.H., and Nielson, C.W. 1970, *SIAM Journal on Numerical Analysis*, vol. 7, pp. 627–656; see also *op. cit.* vol. 11, pp. 753–763. [2]

Hockney, R.W. 1965, *Journal of the Association for Computing Machinery*, vol. 12, pp. 95–113. [3]

Hockney, R.W. 1970, in *Methods of Computational Physics*, vol. 9 (New York: Academic Press), pp. 135–211. [4]

Hockney, R.W., and Eastwood, J.W. 1981, *Computer Simulation Using Particles* (New York: McGraw-Hill), Chapter 6. [5]

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314–329. [6]

19.5　边值问题的松弛法

正如在 19.0 节讨论的，松弛法涉及到分裂由有限差分产生的稀疏矩阵，然后迭代运算直到解被找到。

还有一种考虑松弛法的方式，它在一定程度上更具物理意义。假设要求解椭圆方程

$$\mathcal{L}u=\rho \tag{19.5.1}$$

其中$\mathcal{L}$表示某椭圆算子，ρ 是源项。改写方程为扩散方程：

$$\frac{\partial u}{\partial t}=\mathcal{L}u-\rho \tag{19.5.2}$$

当 $t\to\infty$ 时，初始分布 u 可以**松弛**到一个平衡解，这个平衡解在任意时刻时间导数为零，因此它就是原椭圆问题(19.5.1)式的解。这样一来，在 19.2 节中所有关于扩散的初值方程的技巧，均可通过松弛法用边值问题求解。

让我们把这个思想应用在模型问题(19.0.3)式上，扩散方程为

$$\frac{\partial u}{\partial t}=\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}-\rho \tag{19.5.3}$$

如果利用 FTCS 格式进行差分(参见(19.2.4)式)，得到

$$u_{j,l}^{n+1}=u_{j,l}^{n}+\frac{\Delta t}{\Delta^2}\left(u_{j+1,l}^{n}+u_{j-1,l}^{n}+u_{j,l+1}^{n}+u_{j,l-1}^{n}-4u_{j,l}^{n}\right)-\rho_{j,l}\Delta t \tag{19.5.4}$$

参见(19.2.6)式，我们知道，FTCS 差分在一维空间中仅在 $\Delta t/\Delta^2\leqslant\frac{1}{2}$ 时是稳定的；在两维空间中，条件变为 $\Delta t/\Delta^2\leqslant\frac{1}{4}$。假设试图选择可能的最大时间步长，即设 $\Delta t=\Delta^2/4$，那么(19.5.4)式变为

$$u_{j,l}^{n+1}=\frac{1}{4}\left(u_{j+1,l}^{n}+u_{j-1,l}^{n}+u_{j,l+1}^{n}+u_{j,l-1}^{n}\right)-\frac{\Delta^2}{4}\rho_{j,l} \tag{19.5.5}$$

因此算法中包括利用 u 在网格上的四个最近邻点求平均值(加上源项)。这个过程一直迭代到收敛为止。

这种方法实际上是一种经典方法，被称为**雅可比方法**(注意不要与求特征值的雅可比方法混淆)。由于这种方法收敛很慢，所以它并不实用。不过它是理解现代方法的基础，现代方法多与它进行比较。

另一种经典方法是 **Gauss-Seidel 方法**，这种方法在多重网格法(见 19.6 节)中很重要。该方法中，一旦(19.5.5)式右端 u 的新值可用，我们就利用它，换言之，求平均是“同址”进行的，而不是由较早时间层“复制”到较晚时间层。如果沿行进行该过程，即固定 l 值而逐步增大 j，有

$$u_{j,l}^{n+1}=\frac{1}{4}\left(u_{j+1,l}^{n}+u_{j-1,l}^{n+1}+u_{j,l+1}^{n}+u_{j,l-1}^{n+1}\right)-\frac{\Delta^2}{4}\rho_{j,l} \tag{19.5.6}$$

这种方法也收敛得很缓慢,仅有理论上的价值,不过它的一些分析还是很有启发性的。

从矩阵分裂的角度来看看雅可比方法和 Gauss-Seidel 方法。改变标记,把 $\mathbf{u}$ 记为 $\mathbf{x}$ 以与标准矩阵记法一致。为解

$$\mathbf{A}\cdot\mathbf{x}=\mathbf{b} \tag{19.5.7}$$

考虑把 $\mathbf{A}$ 分裂为

$$\mathbf{A}=\mathbf{L}+\mathbf{D}+\mathbf{U} \tag{19.5.8}$$

其中 $\mathbf{D}$ 是 $\mathbf{A}$ 的对角线部分;$\mathbf{L}$ 是 $\mathbf{A}$ 的下三角部分,对角线上元素取零;$\mathbf{U}$ 是 $\mathbf{A}$ 的上三角部分,对角线上元素取零。

在雅可比方法中,写出迭代的第 r 步为

$$\mathbf{D}\cdot\mathbf{x}^{(r)}=-(\mathbf{L}+\mathbf{U})\cdot\mathbf{x}^{(r-1)}+\mathbf{b} \tag{19.5.9}$$

对于模型问题(19.5.5)式,$\mathbf{D}$ 是一个简单的单位矩阵。雅可比方法对那些具有"对角占优"的矩阵 $\mathbf{A}$,在数学精确的意义上收敛,对于由有限差分产生的矩阵,这个条件通常是满足的。

雅可比方法的收敛速度是多少呢?详细的分析超出了讨论范围,不过,在此还是简单介绍一下:矩阵 $-\mathbf{D}^{-1}\cdot(\mathbf{L}+\mathbf{U})$ 是**迭代矩阵**,不管加项,它把一组 $\mathbf{x}$ 映射到下一组。这个迭代矩阵有特征值,每个特征值都反映了一个因子,在每次迭代中,正是通过该因子抑制了非期望残差的一个特殊特征模的振幅。显然,为了松弛法能够确实有效,这些因子最好都满足模小于 1。雅可比方法的收敛速度由特征模的最慢衰减速度决定,即具有最大模的因子的速率决定,因此,该最大因子的模应界于 0 和 1 之间,被称为松弛算子的**谱半径**,记为 ρ_s。

因此,使全局误差降低为原先的 10^{-p},所需迭代次数 r 可估计为

$$r\approx\frac{p\ln 10}{(-\ln\rho_s)} \tag{19.5.10}$$

一般来说,随着网格尺寸 J 的增大,谱半径 ρ_s 将逐渐逼近数值 1,这就要求有更多的迭代。对一任意给定的方程、网格形状和边界条件,谱半径在原则上可以解析地计算。例如,对于具有 $J\times J$ 网格和在四边均满足 Dirichlet 边界条件的(19.5.5)式,对大的 J 值,ρ_s 的渐近公式可证得为

$$\rho_s\simeq 1-\frac{\pi^2}{2J^2} \tag{19.5.11}$$

为使误差降低到原来的 10^{-p},所需迭代次数 r 为

$$r\simeq\frac{2pJ^2\ln 10}{\pi^2}\simeq\frac{1}{2}pJ^2 \tag{19.5.12}$$

换言之,迭代次数与网格点数 J^2 成比例。由于 100×100 的网格和更大的网格问题有相同的结论,所以很明显,雅可比方法只有理论上的意义。

对于(19.5.6)式,Gauss-Seidel 方法相应于矩阵分解为

$$(\mathbf{L}+\mathbf{D})\cdot\mathbf{x}^{(r)}=-\mathbf{U}\cdot\mathbf{x}^{(r-1)}+\mathbf{b} \tag{19.5.13}$$

$\mathbf{L}$ 在方程左端可根据适当的更新而得出,如按分量写出(19.5.13)式,很容易检验出这点。可以证明(参见本节参考文献[1~3]),该方法的谱半径正好是雅可比方法的谱半径的平方,故对模型问题有

$$\rho_s\simeq 1-\frac{\pi^2}{J^2} \tag{19.5.14}$$

$$r\simeq\frac{pJ^2\ln 10}{\pi^2}\simeq\frac{1}{4}pJ^2 \tag{19.5.15}$$

这个方法比雅可比法在迭代次数上减少了一半,但是,这仍然不能使该方法变得实用。

19.5.1 逐次超松弛法

如果在 Gauss-Seidel 迭代的第 r 步对 $\mathbf{x}^{(r)}$ 的值做一个过度修正,将得到一个更好的算法,它一直到1970年都是作为标准算法。解(19.5.13)式得 $\mathbf{x}^{(r)}$,并在(19.5.13)式的右端加减 $\mathbf{x}^{(r-1)}$,由此 Gauss-Seidel 方法可写为:

$$\mathbf{x}^{(r)} = \mathbf{x}^{(r-1)} - (\mathbf{L}+\mathbf{D})^{-1} \cdot [(\mathbf{L}+\mathbf{D}+\mathbf{U}) \cdot \mathbf{x}^{(r-1)} - \mathbf{b}] \tag{19.5.16}$$

在方括号中的项正是残余向量 $\xi^{(r-1)}$，于是有

$$\mathbf{x}^{(r)} = \mathbf{x}^{(r-1)} - (\mathbf{L}+\mathbf{D})^{-1} \cdot \xi^{(r-1)} \tag{19.5.17}$$

现在进行**过度修正**，定义

$$\mathbf{x}^{(r)} = \mathbf{x}^{(r-1)} - \omega(\mathbf{L}+\mathbf{D})^{-1} \cdot \xi^{(r-1)} \tag{19.5.18}$$

其中 ω 被称为**超松弛参数**，这种方法则叫做**逐次超松弛法(SOR)**。

以下原理可予以证明(参见本节参考文献[1~3])：

- 仅对 $0<\omega<2$ 该方法收敛；如果 $0<\omega<1$，称之为**低松弛**。
- 在有限差分产生的矩阵通常都满足的一些数学限制下，只有超松弛法($1<\omega<2$)可使收敛快于 Gauss-Seidel 方法。
- 设 ρ_{Jacobi} 是雅可比迭代的谱半径(其平方即是 Gauss-Seidel 迭代的谱半径)，则对 ω 的最佳选择为

$$\omega = \frac{2}{1+\sqrt{1-\rho_{\text{Jacobi}}^2}} \tag{19.5.19}$$

- 对 ω 的这个最佳选择，SOR 方法的谱半径为

$$\rho_{\text{SOR}} = \left(\frac{\rho_{\text{Jacobi}}}{1+\sqrt{1-\rho_{\text{Jacobi}}^2}}\right)^2 \tag{19.5.20}$$

作为以上结果的应用，考虑我们的谱半径 ρ_{Jacobi} 是由(19.5.11)式给出的模型问题。由(19.5.19)式和(19.5.20)式得

$$\omega \simeq \frac{2}{1+\pi/J} \tag{19.5.21}$$

$$\rho_{\text{SOR}} \simeq 1 - \frac{2\pi}{J} \qquad \text{for large} \quad J \tag{19.5.22}$$

(19.5.10)式给出使初始误差减少到原先的 10^{-p} 倍所需的迭代次数：

$$r \simeq \frac{pJ\ln 10}{2\pi} \simeq \frac{1}{3}pJ \tag{19.5.23}$$

与(19.5.12)式或(19.5.15)式对比，我们发现最优 SOR 方法不是 J^2 量级的迭代，只需要 J 量级迭代。由于 J 一般为 100 或者更大，这就产生了巨大的差别。(19.5.23)式说明，3 位有效数字精度($p=3$)需要的迭代次数等于沿网格一边的网点数；而对于 6 位数字精度，需要约两倍那么多次的迭代。

对于那些谱半径不是解析可解的情形，如何选择 ω 呢？这正是 SOR 方法的弱点。SOR 方法的优势，仅在围绕 ω 的正确值的一个相当狭小的范围内获得，并且宁可把 ω 选得稍偏大，也不要把它选择得偏小，当然最好还是获得 ω 的正确值。

一种选择 ω 的方法是，把问题近似地与一个已知问题对照起来，用平均值代换方程中的系数。不过要注意，已知的问题必须与实际问题有着相同的网格尺寸和边界条件。下面给出模型问题在矩形的 $J\times L$ 网格上的 ρ_{Jacobi} 值以供参考，允许存在 $\Delta x \neq \Delta y$ 的可能性：

$$\rho_{\text{Jacobi}} = \frac{\cos\dfrac{\pi}{J} + \left(\dfrac{\Delta x}{\Delta y}\right)^2 \cos\dfrac{\pi}{L}}{1+\left(\dfrac{\Delta x}{\Delta y}\right)^2} \tag{19.5.24}$$

该等式对 Dirichlet 边界条件或 Neumann 边界条件保持不变，对于周期性边界条件，进行代换 $\pi \to 2\pi$。

当想求解许多相似的椭圆方程，且每次只有系数有细微的变化时，下面的一种方法特别有用：考虑对第一个方程凭经验确定其最佳 ω 值，然后将此值用于其他方程。关于使用该方程法和"搜寻"最佳 ω 值的各种自动程序，在有关文献中有所介绍。

虽然前面介绍的矩阵标记对理论分析很有用，但对于 SOR 算法的实际运用，还是需要显式的格式。考虑一个一般的沿 x 和 y 方向的二阶椭圆方程，如同对模型方程一样，在一个正方形中进行有限差分，因此对应

于矩阵 A 的每一行,有一个方程形为

$$a_{j,l}u_{j+1,l}+b_{j,l}u_{j-1,l}+c_{j,l}u_{j,l+1}+d_{j,l}u_{j,l-1}+e_{j,l}u_{j,l}=f_{j,l} \tag{19.5.25}$$

对模型方程,有 $a=b=c=d=1$, $e=-4$,量 f 与源项成比例,迭代过程由对 $u_{j,l}$ 求解(19.5.25)式的过程来确定:

$$u^*{}_{j,l}=\frac{1}{e_{j,l}}\left(f_{j,l}-a_{j,l}u_{j+1,l}-b_{j,l}u_{j-1,l}-c_{j,l}u_{j,l+1}-d_{j,l}u_{j,l-1}\right) \tag{19.5.26}$$

$u_{j,l}^{\text{new}}$ 是一个加权平均值

$$u_{j,l}^{\text{new}}=\omega u^*{}_{j,l}+(1-\omega)u_{j,l}^{\text{old}} \tag{19.5.27}$$

计算 $u_{j,l}^{\text{new}}$ 如下:任一步的残差是

$$\xi_{j,l}=a_{j,l}u_{j+1,l}+b_{j,l}u_{j-1,l}+c_{j,l}u_{j,l+1}+d_{j,l}u_{j,l-1}+e_{j,l}u_{j,l}-f_{j,l} \tag{19.5.28}$$

由 SOR 算法(19.5.18)式或(19.5.27)式得

$$u_{j,l}^{\text{new}}=u_{j,l}^{\text{old}}-\omega\frac{\xi_{j,l}}{e_{j,l}} \tag{19.5.29}$$

该公式非常容易编程,且残差向量 $\xi_{j,l}$ 的范数可用做终止迭代的衡量标准。

另一个实际运用中的关键点是:注意网格点排列的顺序。一种直接的策略是简单地沿行(或沿列)排列;另一种方法,假设把网格分为"奇"网格和"偶"网格,如一个棋盘上的黑白方块,则(19.5.26)式即表示奇点仅依赖于偶网格点处的值,反之亦然,因此,可以完成一半扫描,更新奇点值,并根据这些新的奇点值完成另一半扫描,更新偶点值。从下面有关 SOR 方法的实施来看,将采纳奇-偶排序。

最后一个实际应用中的关键点是:SOR 方法的渐近收敛速度直到 J 阶迭代后才能达到,而在达到收敛之前,误差通常以 20 倍的速度增长。对 SOR 方法的一点小修改可以解决这个问题。基于观察可知,尽管 ω 为最优的渐近松弛因子,但它不必是一个好的初始选择值。在具有**切比雪夫加速**的 SOR 方法中,利用奇-偶排序,按照如下规则,可在每次扫描一半时改变 ω 值:

$$\begin{aligned}
\omega^{(0)}&=1\\
\omega^{(1/2)}&=1/(1-\rho_{\text{Jacobi}}^2/2)\\
\omega^{(n+1/2)}&=1/(1-\rho_{\text{Jacobi}}^2\omega^{(n)}/4),\qquad n=1/2,1,\ldots,\infty\\
\omega^{(\infty)}&\to\omega_{\text{optimal}}
\end{aligned} \tag{19.5.30}$$

切比雪夫加速的巧妙之处在于,在每次迭代中,误差的范数总在减少(这是指 $u_{j,l}$ 的实际误差范数,残差 $\xi_{j,l}$ 的范数不必单调下降)。尽管此方法中渐近收敛速度与通常 SOR 一样,但为了减少所需迭代的总数,没有任何理由不使用切比雪夫加速。

下面给出具有切比雪夫加速的 SOR 方法的程序。

```
#include <cmath>
#include "nr.h"
using namespace std;


void NR::sor(Mat_I_DP &a, Mat_I_DP &b, Mat_I_DP &c, Mat_I_DP &d, Mat_I_DP &e,
    Mat_I_DP &f, Mat_IO_DP &u, const DP rjac)
带有切比雪夫加速的逐次超松弛法求解方程(19.5.25)。a,b,c,d,e 和 f 是方程的输入系数,每个变量的尺寸为[0..jmax
-1][0..jmax-1];u 为解的初始猜测,一般设为 0,最后结果返回 u。rjac 作为雅可比迭代的谱半径或其估计值输入。
{
    const int MAXITS=1000;
    const DP EPS=1.0e-13;
    int j,l,n,ipass,jsw,lsw;
    DP anorm,anormf=0.0,omega=1.0,resid;

    int jmax=a.nrows();
    for (j=1;j<jmax-1;j++)
    计算残差的初始范数并当范数减少了EPS倍时终止迭代。
```

```
        for (l=1;l<jmax-1;l++)
            anormf += fabs(f[j][l]);          假设u的初始值为零。
    for (n=0;n<MAXITS;n++) {
        anorm=0.0;
        jsw=1;
        for (ipass=0;ipass<2;ipass++) {       奇-偶排序。
            lsw=jsw;
            for (j=1;j<jmax-1;j++) {
                for (l=lsw;l<jmax-1;l+=2) {
                    resid=a[j][l]*u[j+1][l]+b[j][l]*u[j-1][l]
                        +c[j][l]*u[j][l+1]+d[j][l]*u[j][l-1]
                        +e[j][l]*u[j][l]-f[j][l];
                    anorm += fabs(resid);
                    u[j][l] -= omega*resid/e[j][l];
                }
                lsw=3-lsw;
            }
            jsw=3-jsw;
            omega=(n == 0 && ipass == 0 ? 1.0/(1.0-0.5*rjac*rjac) :
                1.0/(1.0-0.25*rjac*rjac*omega));
        }
        if (anorm < EPS*anormf) return;
    }
    nrerror("MAXITS exceeded");
}
```

SOR 方法的主要优点是易于编程，其主要缺点是它对大问题仍不是非常有效。

19.5.2　交替方向隐式法

在 19.3 节中用于扩散方程的交替方向隐式(ADI)法可以转换成用于椭圆方程的松弛法(参见本节参考文献[1~4])。在 19.3 节中，曾讨论过用 ADI 方法来解时间依赖的热流方程

$$\frac{\partial u}{\partial t} = \nabla^2 u - \rho \tag{19.5.31}$$

通过设 $t \to \infty$，也可得到用来解椭圆方程

$$\nabla^2 u = \rho \tag{19.5.32}$$

的一种迭代方法。每种情形的算子分裂都具有形式

$$\mathcal{L} = \mathcal{L}_x + \mathcal{L}_y \tag{19.5.33}$$

其中$\mathcal{L}_x$ 表示对 x 进行差分，$\mathcal{L}_y$ 表示对 y 进行差分。

例如，在模型问题(19.0.6)中，取 $\Delta x = \Delta y = \Delta$，得到

$$\begin{aligned} \mathcal{L}_x u &= 2u_{j,l} - u_{j+1,l} - u_{j-1,l} \\ \mathcal{L}_y u &= 2u_{j,l} - u_{j,l+1} - u_{j,l-1} \end{aligned} \tag{19.5.34}$$

更复杂的算子也可进行类似地分裂，但要涉及一些技巧问题。选择不当的分裂可能会导致不收敛算法的产生。通常，我们是根据问题的物理性质来进行算子分裂的。我们知道对于模型问题，把一个初始瞬变值传播出去，于是建立起对 x 和 y 的算子分裂，来模拟每维的扩散。

选定算子分裂法后，按两个半步对时间依赖方程(19.5.31)进行隐式差分：

$$\begin{aligned} \frac{u^{n+1/2} - u^n}{\Delta t/2} &= -\frac{\mathcal{L}_x u^{n+1/2} + \mathcal{L}_y u^n}{\Delta^2} - \rho \\ \frac{u^{n+1} - u^{n+1/2}}{\Delta t/2} &= -\frac{\mathcal{L}_x u^{n+1/2} + \mathcal{L}_y u^{n+1}}{\Delta^2} - \rho \end{aligned} \tag{19.5.35}$$

(参见(19.3.16)式，)这里删掉了空间下标(j,l)。用矩阵记号，(19.5.35)式为

$$(\mathbf{L}_x + r\mathbf{1}) \cdot \mathbf{u}^{n+1/2} = (r\mathbf{1} - \mathbf{L}_y) \cdot \mathbf{u}^n - \Delta^2 \rho \tag{19.5.36}$$

$$(\mathbf{L}_y + r\mathbf{1}) \cdot \mathbf{u}^{n+1} = (r\mathbf{1} - \mathbf{L}_x) \cdot \mathbf{u}^{n+1/2} - \Delta^2 \rho \tag{19.5.37}$$

其中

$$r \equiv \frac{2\Delta^2}{\Delta t} \tag{19.5.38}$$

(19.5.36)式和(19.5.37)式左端的矩阵是三对角型矩阵(通常还是正定的),因而它们可以通过标准三对角算法来求解。给定 $\mathbf{u}^n$,解(19.5.36)式求解 $\mathbf{u}^{n+1/2}$,代入(19.5.37)式的右端则可解出 $\mathbf{u}^{n+1}$。关键的问题在于如何选择迭代参数 r,这与初值问题中时间步的选择类似。

和通常一样,目的是使迭代矩阵的谱半径最小。虽然详细的介绍已超出本节的讨论范围,但可以证明,r 的最佳选取会使 ADI 方法与 SOR 有相同的收敛速度。ADI 方法中,各个迭代步长比 SOR 中的要复杂得多,因此 ADI 方法显得要差一些。可以对每个迭代步都选同样的参数 r。若每步都选不同的 r 也是可能的。若能按最优方式实现,则 ADI 比 SOR 通常具有更高的效率。详细内容请参阅本节参考文献[1~4]。

这里没有完全采用 ADI 的一个原因是,在大部分应用中,它已经被下节介绍的多重网格法所替代。我们建议,对小规模问题(比如 20×20)或只解一次的较大问题可采用 SOR 方法。对于后一种情形,程序设计的容易程度比机时的耗费更重要。在 2.7 节中的稀疏矩阵方法,偶尔也用来直接求解一组差分方程。但是对于大椭圆问题的生成解,现在几乎都选用多重网格方法。

参考文献和深入阅读

Hockney, R.W., and Eastwood, J.W. 1981, *Computer Simulation Using Particles* (New York: McGraw-Hill), Chapter 6.

Young, D.M. 1971, *Iterative Solution of Large Linear Systems* (New York: Academic Press). [1]

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag), §§8.3–8.6. [2]

Varga, R.S. 1962, *Matrix Iterative Analysis* (Englewood Cliffs, NJ: Prentice-Hall). [3]

Spanier, J. 1967, in *Mathematical Methods for Digital Computers, Volume 2* (New York: Wiley), Chapter 11. [4]

19.6 边值问题的多重网格法

实用多重网格法是 Brandt 在 20 世纪 70 年代首次提出的。这些方法可以用 $O(N)$ 次运算,求解在 N 个网格点上的椭圆型偏微分方程。在 19.4 节中,讨论过的"快速"直接椭圆求解法,可用 $O(N\log N)$ 次运算,求解一些特殊类型的椭圆方程。这些估计值中的数值系数,使得多重网格方法在运行速度上可以和快速方法相比较。但是与快速方法不同的是,多重网格方法求解非常系数的一般椭圆型方程的效率几乎没有下降,甚至对非线性方程也有类似的求解速度。

遗憾的是,没有一种单独的多重网格算法,能解所有椭圆型问题。多重网格方法只是对这些问题的求解提供了一个框架,对于特定问题,必须对算法框架内的各种组成进行调整。在这里只能给出关于此专题的简单介绍,特别是将给出两个多网格程序范例,一个是线性,一个是非线性的。依照这些原型并参阅本节参考文献[1~4],读者能够学会编写程序,来求解自己的问题。

多重网格技术的应用有两种相互联系但又明显不同的方法。第一种称为"多重网格法",是用来提高传统松弛法的收敛速度的一种方法,由读者自己定义一个事先确定的精细网格。在这种情形,只需在该网格上定义自己的问题(例如,估计源项)。而其他一些较粗的,由方法定义的网格可以作为中间计算的附加值。

第二种方法称为(也许容易混淆)"完全多重网格(FMG)法",要求读者能够在任意尺寸的网格上定义该问题(即将同一个基本偏微分方程离散成不同尺度的有限差分方程组)。在该方法中,随着网格的不断加细得到一系列解。读者可以停在一个预先确定的精细的网格上,也可以监视由离散过程引起的截断误差,使其停在误差可以接受的网格上。

本节将先讨论"多重网格法",然后利用刚才建立的概念介绍 FMG 法。在附加程序中,将实施后一种算法。

19.6.1　从单网格，到两重网格，再到多重网格

多重网格法的关键思想，可以通过研究最简单的两重网格法来理解。假定要求解线性椭圆问题

$$\mathcal{L}u = f \tag{19.6.1}$$

其中$\mathcal{L}$是某个线性椭圆算子，f是源项。在一个网格大小为 h 的均匀网格上，离散化方程(19.6.1)，得到的线性代数方程组为

$$\mathcal{L}_h u_h = f_h \tag{19.6.2}$$

令 $\tilde{u}_h$ 表示(19.6.2)式的某近似解，用符号 u_h 表示差分方程(19.6.2)的精确解。则 $\tilde{u}_h$ 的**误差**或**修正值**为

$$v_h = u_h - \tilde{u}_h \tag{19.6.3}$$

残差或**亏损**为

$$d_h = \mathcal{L}_h \tilde{u}_h - f_h \tag{19.6.4}$$

(注意，有些作者将残差定义为负亏损，(19.6.4)式究竟定义这两个量中的哪一个还没有取得一致的看法。)由于$\mathcal{L}_h$是线性的，因此误差满足

$$\mathcal{L}_h v_h = -d_h \tag{19.6.5}$$

现在，需要用$\mathcal{L}_h$的一个近似值来找v_h。经典迭代方法，例如雅可比式 Gauss-Seidel 方法，是通过每步寻找下述方程的一个近似解来完成的：

$$\hat{\mathcal{L}}_h \hat{v}_h = -d_h \tag{19.6.6}$$

其中$\hat{\mathcal{L}}_h$是一个比$\mathcal{L}_h$"较简单"的算子。例如，在雅可比迭代中，$\hat{\mathcal{L}}_h$是$\mathcal{L}_h$的对角部分，或在 Gauss-Seidel 迭代中，$\hat{\mathcal{L}}_h$是$\mathcal{L}_h$的下三角部分。第二个近似是根据下式得到的：

$$\tilde{u}_h^{\text{new}} = \tilde{u}_h + \hat{v}_h \tag{19.6.7}$$

作为另一种选择，现在再考虑对$\mathcal{L}_h$的一种完全不同的近似。其中我们使它"粗糙化"，而不是"简单化"。即，在网格大小为 H(常取 $H=2h$，但其他的取法也可以)的一个更粗糙的网格上，构造$\mathcal{L}_h$的某个适当近似算子$\mathcal{L}_H$。则残差方程(19.6.5)近似为

$$\mathcal{L}_H v_H = -d_H \tag{19.6.8}$$

由于$\mathcal{L}_H$维数较小，该方程将比(19.6.5)式易解。为了定义粗网格上的亏损 d_H，需要一个**限制算子**$\mathcal{R}$，将 d_h 限制在粗网格上：

$$d_H = \mathcal{R} d_h \tag{19.6.9}$$

限制算子也称为**细-粗算子**或**嵌入算子**。一旦得到(19.6.8)式的一个解 $\tilde{v}_H$ 后，还需要一个**延拓算子**$\mathcal{P}$，它将修正值拓展或内插到细网格上：

$$\tilde{v}_h = \mathcal{P}\tilde{v}_H \tag{19.6.10}$$

延拓算子也称为**粗-细算子**或**内插算子**。$\mathcal{R}$和$\mathcal{P}$都取为线性算子。最后 $\tilde{u}_h$ 近似值可以校正为

$$\tilde{u}_h^{\text{new}} = \tilde{u}_h + \tilde{v}_h \tag{19.6.11}$$

因此，"粗网格修正格式"的步骤是：

- 根据(19.6.4)式计算细网格点上的亏损。
- 通过(19.6.9)式来限制亏损。
- 在粗网格上解(19.6.8)式，求修正值。
- 通过(19.6.10)式将修正值内插到细网格上。
- 通过(19.6.11)式计算第二个近似。

让我们来比较一下松弛法和粗网格修正格式的优缺点。考虑将误差 v_h 展开成一个离散傅里叶序列，称低半部分的频率谱分量为**光滑分量**，而高半部分为**非光滑分量**。我们已经看到，松弛法在极限 $h\to 0$ 时的情形，收敛速度变得很慢，这相当于网格点数目很多的情形。原因是，在每次迭代中光滑分量的幅度只稍微减小一点。然而，许多松弛法却使非光滑分量的幅度在每次迭代中都减少许多倍，所以它们是很好的光滑算子。

另一方面，对于两重网格迭代法来说。波长$\lesssim 2H$的误差分量，在粗网格点上甚至不能表示出来，因而在

该网格上不能降为零。但这些高频成分却可以通过松弛法在细网格上被减小。这就启发我们,将松弛法和粗网格修正的思想结合起来使用。

- 预先光滑:将松弛法中 $\nu_1 \geq 0$ 的步骤应用到 $\bar{u}_h$ 上来计算 $\bar{u}_h$。
- 粗网格修正:和前文一样,利用 u_h 给出 $\bar{u}_h^{\text{new}}$。
- 后期光滑:将松弛法中 $\nu_2 \geq 0$ 的步骤应用到 $\bar{u}_h^{\text{new}}$ 上,来计算出 $\bar{u}_h^{\text{new}}$。

从上述两重网格法到多重网格法就很容易。不用求解粗网格亏损方程(19.6.8),而我们可以通过引进一个更粗的网格,并应用两重网格迭代法,得到它的一个近似解。如果两重网格法的收敛因子足够小,则将只需几步迭代就能得到一个足够好的近似值。用 γ 来表示这样的迭代次数。显然,可能将该思想递归地用到一些很粗的网格上。在那网格上求解很容易,例如,通过直接矩阵求逆的方法,或通过迭代松弛格式到收敛值。

多重网格法中,从最细网格到较粗网格,再回到最细网格的一次迭代称为一个循环。循环的精确结构依赖于 γ 值,而 γ 值是在每个中间步骤两重网格法迭代的次数。$\gamma=1$ 的情形,称为 V 循环;而 $\gamma=2$ 的情形,称为 W 循环(参见图 19.6.1)。这些是实际应用中最重要的情形。

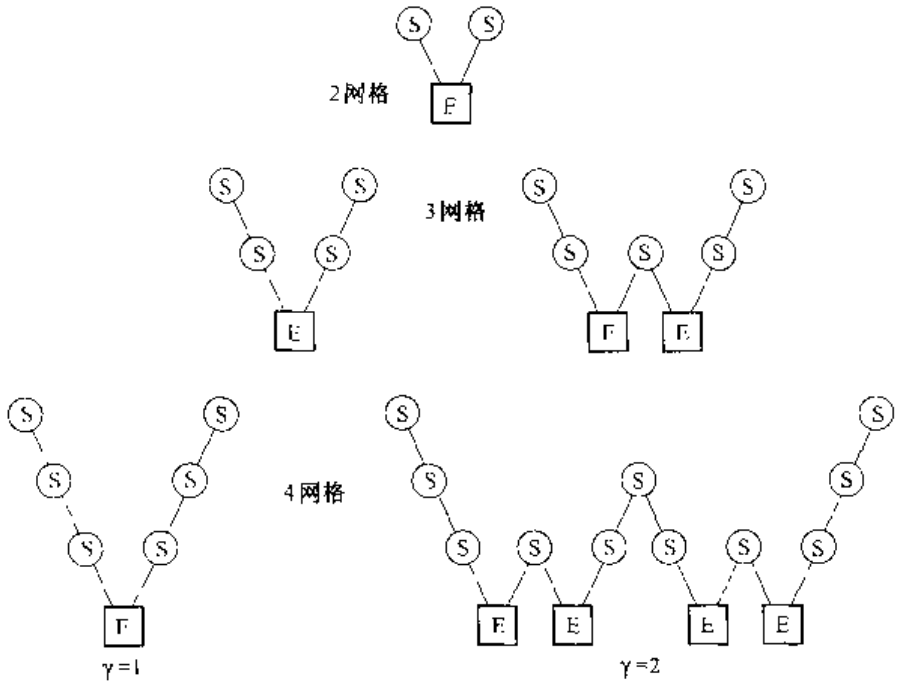


图 19.6.1 多重网格循环结构。S 表示光滑度,E 表示最粗网格上的精确解。每个下降线"\"表示限制($\mathcal{R}$),每个上升线"/"表示延拓($\mathcal{P}$)。最细的网格在每幅图的顶部。对于 V 循环($\gamma=1$),每当网格层数增加 1 时,E 步就被一个两重网格迭代所代替。对于 W 循环($\gamma=2$),E 步被两个两重网格迭代所代替

注意,一旦涉及两个以上的网格时,在最细网格上,第一步之后的预先光滑步需要误差 v 的初始近似值,该值应取为零。

19.6.2 光滑、限制及延拓算子

最流行,而且也是应该最先试验的光滑方法是 Gauss-Seidel 方法,因为它能产生很好的收敛速度。如果将网格点从 0 顺序编号到 $N-1$,则 Gauss-Seidel 格式为

$$u_i = -\left(\sum_{\substack{j=0\\ j\neq i}}^{N-1} L_{ij}u_j - f_i\right)\frac{1}{L_{ii}} \qquad i=0,\ldots,N-1 \tag{19.6.12}$$

其中 u 的新值放在右端项使用。Gauss-Seidel 方法的精确形式依赖于网格点所选取的排序。对于模型方程(19.0.3)的二阶椭圆方程,和(19.0.8)式的差分格式一样,最好用红-黑排序,使一个通过网格修正"偶"点(像

一个方格盘上的红格)，而使另一个通过网格修正“奇”点(黑格)。当沿某一维比沿另一维的耦合程度更强时，就需要沿该维同时将整条线松弛。最邻近耦合的线松弛涉及一个三对角系统的求解，因而仍然具有很高的效率。相继对奇偶线进行的松弛过程为斑状松弛，通常要优于简单的线松弛

注意，SOR 不能用做光滑算子，超松弛会破坏高频光滑成分，这对多重网格法是至关重要的。

延拓和限制算子的一个简洁的表示方法是，给出它们的记号。可以通过考虑 v_H 在某些网格点(x,y)上取 1，在其外点上取 0，再定义$\mathcal{P}v_H$，得到记号$\mathcal{P}$。最流行的延拓算子是简单的双线性插值，它给出 9 个点$(x,y),(x+h,y),\ldots,(x-h,y-h)$上的非零值，分别为$1,\frac{1}{2},\ldots,\frac{1}{4}$。因此符号为

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{19.6.13}$$

通过考虑在细网格的各点上定义 v_h 再将(x,y)处的$\mathcal{R}v_h$ 定义为这些值的一个线性组合，然后就得到记号$\mathcal{R}$的定义。$\mathcal{R}$最简单的一种可能选法是**直接插入**，即在每个粗网格点中简单地用对应的细网格点的值来填充，它的记号是“[1]”。但是，这种取法在实际应用中可能会产生困难。可以证明$\mathcal{R}$的安全选法是使它为$\mathcal{P}$的一个伴随算子。为了定义伴随算子，首先将网格尺度为 h 的两个网格函数 u_h 和 v_h 的标量积定义为

$$\langle u_h|v_h\rangle_h \equiv h^2\sum_{x,y} u_h(x,y)v_h(x,y) \tag{19.6.14}$$

然后，将$\mathcal{P}$的伴随算子用$\mathcal{P}^\dagger$表示，定义为

$$\langle u_H|\mathcal{P}^\dagger v_h\rangle_H = \langle \mathcal{P}u_H|v_h\rangle_h \tag{19.6.15}$$

现在，取$\mathcal{P}$为双线性插值，并选定(x,y)处的 $u_H=1$，其余各处为零。令(19.6.15)式中的$\mathcal{P}^\dagger=\mathcal{R}$，$H=2h$。可以看出

$$(\mathcal{R}v_h)_{(x,y)} = \tfrac{1}{4}v_h(x,y)+\tfrac{1}{8}v_h(x+h,y)+\tfrac{1}{16}v_h(x+h,y+h)+\cdots \tag{19.6.16}$$

因此$\mathcal{R}$的记号为

$$\begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix} \tag{19.6.17}$$

注意下述的简单法则：$\mathcal{R}$记号是$\mathcal{P}$记号定义的矩阵(19.6.13)乘以$\frac{1}{4}$后的转置矩阵。只要$\mathcal{R}=\mathcal{P}^\dagger$且 $H=2h$，该法则总成立。

(19.6.17)式中的$\mathcal{R}$的特定选取称为**完全加权**。$\mathcal{R}$的另一个常用选法是“半加权”，是完全加权和直接插入之间的“折衷”。它的记号是

$$\begin{bmatrix} 0 & \frac{1}{8} & 0 \\ \frac{1}{8} & \frac{1}{2} & \frac{1}{8} \\ 0 & \frac{1}{8} & 0 \end{bmatrix} \tag{19.6.18}$$

差分算子$\mathcal{L}_h$可用类似的符号来描述。例如，模型问题(19.0.6)式的标准差分由**五点差分星形**来表示：

$$\mathcal{L}_h = \frac{1}{h^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{19.6.19}$$

如果遇到一个新问题，而且不能确定选取怎样的$\mathcal{P}$和$\mathcal{R}$才能取得较好的效果时，下面是一个保险的法则：假定 m_p 是插值$\mathcal{P}$的阶(即，它对 m_p-1 次多项式精确插值)。假定 m_r 是$\mathcal{R}$的阶，$\mathcal{R}$是某个$\mathcal{P}$的伴随算子(不一定是想用的$\mathcal{P}$)。那么，如果 m 是差分算子$\mathcal{L}_h$的阶，则应当满足不等式 $m_p+m_r>m$。例如泊松方程的双线性插值及其伴随算子，完全加权，满足 $m_p+m_r=4>m=2$。

当然$\mathcal{P}$和$\mathcal{R}$算子应该将边界条件施加到要求的问题中。最简单的方法是，如果需要的话，可以通过改变源项将差分方程改写为具有齐次边界条件的形式(参见 19.4 节)。施加齐次边界条件，只要求$\mathcal{P}$算子在适当

的边界点简单地产生零值即可。相应的$\mathcal{R}$可以根据$\mathcal{R}=\mathcal{P}^T$得到。

19.6.3 完全多重网格算法

迄今为止,所描述的多重网格法都是一种迭代格式,从最细网格上的一些初始猜测值开始,经过足够的循环(V循环、W循环等)后达到收敛。这是多重网格最简单的一种用法:简单地应用足够多次循环,直到符合某个适当的收敛标准为止。但利用**完全多重网格算法(FMG)**可以提高效率,FMG也称为**嵌套迭代**。

不从最细网格上的任意近似值开始(例如,$u_h=0$),而第一个近似值是通过对一个粗网格解的内插得到:

$$u_h=\mathcal{P}u_H \tag{19.6.20}$$

粗网格解是利用类似FMG的过程从更粗网格上得到的,在最粗的网格上,用精确值开始。和图19.6.1中的过程不同,FMG是通过一系列越来越高的"N"到达解的,每个较高的位置都表示在较细的网格上得出的解。(见图19.6.2)。

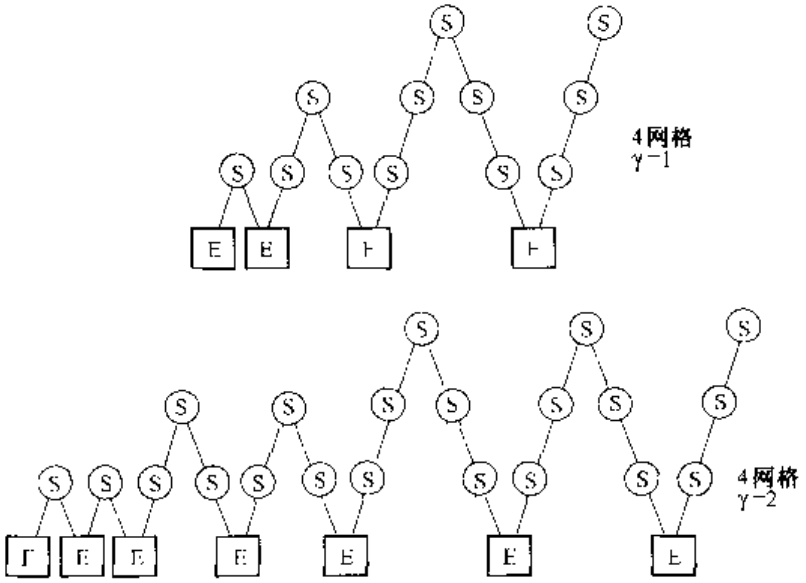


图19.6.2 完全多重网格算法(FMG)的循环结构。该方法从最粗的网格开始,先插值,然后加密(通过"V"),得到不断变细的网格上的解

注意,(19.6.20)式中的$\mathcal{P}$不一定和多重网格循环中的$\mathcal{P}$相同,它应该至少与离散化的$\mathcal{L}_h$同阶,但有时一个高阶算子可能会有更高的效率。

可以证明,在进入下个更细网格之前,在每一层次上通常需要一次或最多两次的多重网格循环。虽然所需循环次数有理论上的指导(例如本节参考文献[2]),但可以根据经验很容易地确定它。先固定最细的网格,然后增加每一层网格上的循环次数,并研究解值的变化。这些解的渐近值就是差分方程的精确值。该精确解与小数目循环次数得到的解之差为迭代误差。现在,再固定循环次数为某个较大的值,改变网格层数,即所使用的最小h值。用这种方法可以估算一个给定h的截断误差。最终得到的结论是,用多次循环毫无意义,并不能使迭代误差降低到截断误差的大小水平。

简单多重网格迭代(循环),只在最细网格上需要用右端项f,而FMG则在所有网格上都需要f。如果边界条件是齐次的,则可以使用$f_H=\mathcal{R}f_h$。对于非齐次边界条件,该等式不总是成立的。此时,最好在每种粗网格上都将f离散。

注意,FMG算法能求得所有网格上的解。因而它可以和Richardson外推法之类的技巧一起使用。

现在,给出一个程序mglin,它利用完全多重网格算法来解一个线性方程,如模型问题(19.0.6)。它用红-黑、Gauss-Seidel作为光滑算子,取$\mathcal{P}$为双线性插值,$\mathcal{R}$为半加权。为使程序能处理其他线性问题,只需适当修改函数relax,resid及slvsml即可。该程序的一个特点是,定义在不同网格上的变量采用动态存储。

```
#include "nr.h"

namespace {
    void mg(int j, Mat_IO_DP &u, Mat_I_DP &rhs)
```

递归调用多重网格法，输入变量 j 为当前层号，u 存储当前解的值，rhs 为右端向量。输出 u 存储下一层的更新的解的值。

```
{
    using namespace NR;
    const int NPRE=1,NPOST=1;              在粗网格修正计算前
    int jpost,jpre,nc,nf;                  后松驰扫描的次数。

    nf=u.nrows();
    nc=(nf+1)/2;
    if (j == 0)                            V的底端：在最粗网格上的求解。
        slvsml(u,rhs);
    else {                                 V的向下行程。
        Mat_DP res(nc,nc),v(0.0,nc,nc),temp(nf,nf);
        在每一步松驰中的初始猜测值置为0。
        for (jpre=0;jpre<NPRE;jpre++)
                relax(u,rhs);              预光滑处理。
            resid(temp,u,rhs);
            rstrct(res,temp);              残差的限制是下一步rhs
            mg(j-1,v,res);                 递归调用粗网格修正。
            addint(u,v,temp);              V的向上行程。
            for (jpost=0;jpost<NPOST;jpost++)
                relax(u,rhs);              后期光滑处理。
        }
    }
}


void NR::mglin(Mat_IO_DP &u, const int ncycle)
完全多重网格算法求解线性椭圆型方程，这里为模型问题(19.0.6)。输入变量 u[0..n-1][0..n-1]包含右端ρ，而输
出则返回解。维数 n 必须是 2^j + 1，j 为某个整数(j 实际上是求解中用到的网格层数，程序中称为 ng)，ncycle 是每层用
到的 V 循环数。
{
    int j,jcycle,ng=0,ngrid,nn;
    Mat_DP *uj,*uj1;

    int n=u.nrows();

    nn=n;
    while (nn >>= 1) ng++;
    if ((n-1) != (1 << ng))
        nrerror("n-1 must be a power of 2 in mglin.");
    Vec_Mat_DP_p rho(ng);                  在每一层上，rho是指向ρ的指针向量。
    nn=n;
    ngrid=ng-1;
    rho[ngrid] = new Mat_DP(nn,nn);        对网格ng-1上的r.h.s分配存储空间，并赋值。
    copy(*rho[ngrid],u);
    while (nn > 3) {                       类似地对所有粗网格上的r.h.s分配空间，
        nn=nn/2+1;                         并通过细网格上的限制进行赋值。
        rho[--ngrid]=new Mat_DP(nn,nn);
        rstrct(*rho[ngrid],*rho[ngrid+1]);
    }
    nn=3;
    uj=new Mat_DP(nn,nn);
    slvsml(*uj,*rho[0]);                   最粗网格上的初始解。
    for (j=1;j<ng;j++) {                   嵌套迭代循环。
        nn=2*nn-1;
        uj1=uj;
        uj=new Mat_DP(nn,nn);
        interp(*uj,*uj1);                  从网格j-1到更细的网格j的插值。
        delete uj1;
        for (jcycle=0;jcycle<ncycle;jcycle++)          V-循环。
            mg(j,*uj,*rho[j]);
    }
    copy(u,*uj);                                       返回u中的解。
    delete uj;
    for (j=0;j<ng;j++)
        delete rho[j];
}


#include "nr.h"
```

```
void NR::rstrct(Mat_O_DP &uc, Mat_I_DP &uf)
```

半加权限制。nc是粗网格的维数,在uf[0..2*nc-2][0..2*nc-2]中输入是细网格解,粗网格返回到uc[0..nc-1][0..nc-1]中。

```
{
    int ic,iif,jc,jf,ncc;
    int nc=uc.nrows();
    ncc=2*nc-2;
    for (jf=2,jc=1;jc<nc-1;jc++,jf+=2) {          内部点。
        for (iif=2,ic=1;ic<nc-1;ic++,iif+=2) {
            uc[ic][jc]=0.5*uf[iif][jf]+0.125*(uf[iif+1][jf]+uf[iif-1][jf]
                +uf[iif][jf+1]+uf[iif][jf-1]);
        }
    }
    for (jc=0,ic=0;ic<nc;ic++,jc+=2) {            边界点。
        uc[ic][0]=uf[jc][0];
        uc[ic][nc-1]=uf[jc][ncc];
    }
    for (jc=0,ic=0;ic<nc;ic++,jc+=2) {
        uc[0][ic]=uf[0][jc];
        uc[nc-1][ic]=uf[ncc][jc];
    }
}
```

```
#include "nr.h"

void NR::interp(Mat_O_DP &uf, Mat_I_DP &uc)
```

通过双线性插值得到粗-细网格延拓。nf是细网格的维数。uc[0..nc-1][0..nc-1]作为粗网格解的输入,其中nc=nf/2+1。细网格解返回到uf[0..nf-1][0..nf-1]。

```
{
    int ic,iif,jc,jf,nc;

    int nf=uf.nrows();
    nc=nf/2+1;
    for (jc=0;jc<nc;jc++)                         复制循环。
        for (ic=0;ic<nc;ic++) uf[2*ic][2*jc]=uc[ic][jc];
    for (jf=0;jf<nf;jf+=2)                        偶数列循环, 竖直插值。
        for (iif=1;iif<nf-1;iif+=2)
            uf[iif][jf]=0.5*(uf[iif+1][jf]+uf[iif-1][jf]);
    for (jf=1;jf<nf-1;jf+=2)                      奇数列循环, 水平插值。
        for (iif=0;iif<nf;iif++)
            uf[iif][jf]=0.5*(uf[iif][jf+1]+uf[iif][jf-1]);
}
```

```
#include "nr.h"

void NR::addint(Mat_O_DP &uf, Mat_I_DP &uc, Mat_O_DP &res)
```

作粗-细网格插值并将结果放在uf中,nf是细网格的维数。uc[0..nc-1][0..nc-1]中输入是粗网格解,其中nc=nf/2+1。细网格解返回到uf[0..nf-1][0..nf-1]。res[0..nf-1][0..nf-1]作为暂时存储空间。

```
{
    int i,j;

    int nf=uf.nrows();
    interp(res,uc);
    for (j=0;j<nf;j++)
        for (i=0;i<nf;i++)
            uf[i][j] += res[i][j];
}
```

```
#include "nr.h"

void NR::slvsml(Mat_O_DP &u, Mat_I_DP &rhs)
```

在最粗网格上求解模型问题,其中h=1/2。在rhs[0..2][0..2]中是右端项的输入,解返回到u[0..2][0..2]中。

```
{
    int i,j;
    DP h=0.5;

    for (i=0;i<3;i++)
        for (j=0;j<3;j++)
            u[i][j]=0.0;
```

```
    u[1][1] = -h*h*rhs[1][1]/4.0;
}

#include "nr.h"

void NR::relax(Mat_IO_DP &u, Mat_I_DP &rhs)
模型问题的红－黑 Gauss－Seidel 松弛。利用右端项函数 rhs[0..n-1][0..n-1],更新解 u[0..n-1][0..n-1]的当前值。
{
    int i,ipass,isw,j,jsw=1;
    DP h,h2;

    int n=u.nrows();
    h=1.0/(n-1);
    h2=h*h;
    for (ipass=0;ipass<2;ipass++,jsw=3-jsw) {        红和黑扫描。
        isw=jsw;
        for (j=1;j<n-1;j++,isw=3-isw)
            for (i=isw;i<n-1;i+=2)                    Gauss-Seidel公式。
                u[i][j]=0.25*(u[i+1][j]+u[i-1][j]+u[i][j+1]
                    +u[i][j-1]-h2*rhs[i][j]);
    }
}

#include "nr.h"

void NR::resid(Mat_O_DP &res, Mat_I_DP &u, Mat_I_DP &rhs)
返回模型问题的负残差。输入量是 u[0..n-1][0..n-1]和 rhs[0..n-1][0..n-1]而返回到 res[0..n-1][0..n-1]。
{
    int i,j;
    DP h,h2i;

    int n=u.nrows();
    h=1.0/(n-1);
    h2i=1.0/(h*h);
    for (j=1;j<n-1;j++)          内部点。
        for (i=1;i<n-1;i++)
            res[i][j] = -h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]
                +u[i][j-1]-4.0*u[i][j])+rhs[i][j];
    for (i=0;i<n;i++)            边界点。
        res[i][0]=res[i][n-1]=res[0][i]=res[n-1][i]=0.0;
}

#include "nr.h"

void NR::copy(Mat_O_DP &aout, Mat_I_DP &ain)
复制 ain[0..n-1][0..n-1]到 aout[0..n-1][0..n-1]中。
{
    int i,j;

    int n=ain.nrows();
    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
            aout[j][i]=ain[j][i];

}
```

程序 mglin 的编写是为了使算法更加明确，而不是为了追求最高的效率，因而很容易对它进行改进。下面几个简单的修改将加快运行速度。

- 细－黑 Gauss-Seidel 步后，所有黑网格点上的亏损 d_h 均为零。因此，半加权 $d_H = \mathcal{R} d_h$ 变为只需简单地将细网格上亏损的一半复制到相应的粗网格上。在 V 循环的第一部分中 rstrct 前对 resid 的调用可以通过一个只在粗网格上循环的程序来代替，它在粗网格上填进一半亏损。
- 类似地，红网格点上的量 $\bar{u}_h^{new} = \bar{u}_h + \mathcal{P}\hat{v}_H$ 不用计算，因为它们在接着进行的 Gauss-Seidel 扫描中将立即被重新定义。这意味着 addint 只需在黑网点上进行循环。
- 可以用多种方法使 relax 提高运行速度。首先，当初始猜测值取零时，可以得到一个特殊的形式。其次，可以将各种网格上的 $h^2 f_h$ 存起来，节省了一次乘法运算。最后，可以通过中间变量重写 Gauss-Seidel

公式,将公式中的一个加法存起来。

- 对于模型问题,ncycle = 1 时,对于给定的 h,由 mglin 返回的解其迭代误差将大于截断误差。为了将误差降低到截断误差的水平,必须置 ncycle = 2,或更经济地置 NPRE = 2。一个更有效的方法是,在(19.6.20)式中采用了一个更高阶的$\mathcal{P}$,它比 V 循环中使用的线性插值阶次更高。

实现所有上述特性通常可以使运行时间缩短到原来的 1/2,并且最终结果很有用。

19.6.4 深入讨论:非线性多重网格法:FAS 算法

现在,再来求解非线性椭圆方程,将其形式写为

$$\mathcal{L}(u) = 0 \tag{19.6.21}$$

任意显式的源项都移到了左端。假定(19.6.21)式适合离散为

$$\mathcal{L}_h(u_h) = 0 \tag{19.6.22}$$

下面,将会看到在多重网格算法中,必须考虑求解过程中产生非零右端项的方程:

$$\mathcal{L}_h(u_h) = f_h \tag{19.6.23}$$

多重网格解非线性问题的一个途径是,利用牛顿法:即在每步迭代时,对修正项产生线性方程,于是,可以利用线性多重网格法来解这些方程。然而,多重网格法思想的巨大优势表现在它可以直接应用到非线性问题。要做的只是,选一个适当的非线性松弛法来光滑误差,再加上一个估算粗网格上近似修正值的过程。这个直接方法就是,Brandt 的完全近似存储算法(FAS)。也许除最粗网格外,一般没有非线性方程需要求解。

这为实现非线性算法,假定像线性情形一样,已经有一个能光滑残差向量的松弛法过程。于是可以找一个光滑的修正值 v_h 来解(19.6.23)式:

$$\mathcal{L}_h(\tilde{u}_h + v_h) = f_h \tag{19.6.24}$$

为了找 v_h,注意

$$\begin{aligned}\mathcal{L}_h(\tilde{u}_h + v_h) - \mathcal{L}_h(\tilde{u}_h) &= f_h - \mathcal{L}_h(\tilde{u}_h)\\ &= -d_h\end{aligned} \tag{19.6.25}$$

经过几次非线性松弛扫描后,右端是光滑的。因而,可以将左端转移到一个粗网格上:

$$\mathcal{L}_H(u_H) - \mathcal{L}_H(\mathcal{R}\tilde{u}_h) = -\mathcal{R}d_h \tag{19.6.26}$$

即,在粗网格上解

$$\mathcal{L}_H(u_H) = \mathcal{L}_H(\mathcal{R}\tilde{u}_h) - \mathcal{R}d_h \tag{19.6.27}$$

(非零右端项就是这样出现的。)假定近似解为 $\tilde{u}_H$,则粗网格上的修正值为

$$\tilde{v}_H = \tilde{u}_H - \mathcal{R}\tilde{u}_h \tag{19.6.28}$$

以及

$$\tilde{u}_h^{\text{new}} = \tilde{u}_h + \mathcal{P}(\tilde{u}_H - \mathcal{R}\tilde{u}_h) \tag{19.6.29}$$

注意,一般情况下$\mathcal{P}\mathcal{R} \neq 1$,因此 $\tilde{u}_h^{\text{new}} \neq \mathcal{P}\tilde{u}_H$。这是关键点:(19.6.29)式中的插值误差只由修正值产生,而不是由整个解 $\tilde{u}_H$ 产生。

(19.6.27)式表明求解的是完全近似 u_H,而不是线性算法中只求误差。这就是 FAS 名称的由来。

因此,FAS 多重网格算法与线性多重网格算法看来极其相似。仅有的差别是,亏损 d_h 和松弛近似值 u_h 都必须被限制在粗网格上,通过对算法的递归调用,求解(19.6.27)式。但是,不通过这种途径也可实现算法,将先介绍所谓的对偶观点,它能使我们看到多重网格思想的另一个方面优势。

对偶观点考虑局部截断误差,定义为

$$\tau \equiv \mathcal{L}_h(u) - f_h \tag{19.6.30}$$

其中 u 是原始连续方程的精确解。若将上式改写为

$$\mathcal{L}_h(u) = f_h + \tau \tag{19.6.31}$$

看到 τ 可以作为对 f_h 的修正值,因而细网格上方程的解将成为精确解 u。

现在,考虑**相对截断误差** τ_h,它定义在与 h 网格相关的 H 网格上:

$$\tau_h \equiv \mathcal{L}_H(\mathcal{R}u_h) - \mathcal{R}\mathcal{L}_h(u_h) \tag{19.6.32}$$

由于$\mathcal{L}_h(u_h)=f_h$，上式可改写为

$$\mathcal{L}_H(u_H)=f_H+\tau_h \tag{19.6.33}$$

换言之，可以认为 f_H 经过 τ_h 修正后，使得粗网格方程解与细网格方程解相等。当然，不能计算出 τ_h，但确实可以利用(19.6.32)式中的 $\tilde{u}_h$ 给出它的一个近似值：

$$\tau_h\simeq\tilde{\tau}_h\equiv\mathcal{L}_H(\mathcal{R}\tilde{u}_h)-\mathcal{R}\mathcal{L}_h(\tilde{u}_h) \tag{19.6.34}$$

将(19.6.33)式中的 τ_h 用 $\tilde{\tau}_h$ 代替，得出

$$\mathcal{L}_H(u_H)=\mathcal{L}_H(\mathcal{R}\tilde{u}_h)-\mathcal{R}d_h \tag{19.6.35}$$

正是粗网格方程(19.6.27)。

因此看到，粗细网格之间的关系有两种互补的观点：

- 粗网格用来提高细网格残差的光滑成分的收敛速度
- 细网格用来计算对粗网格方程的修正值，使粗网格上得到细网格上的精度。

这种新观念的一个优点是，对于多重网格迭代，利用它可以得到一个自然的停止标准。这个标准通常为

$$\|d_h\|\leqslant\epsilon \tag{19.6.36}$$

问题在于如何选取ϵ。显然，当剩余误差基本上为局部截断误差τ的量级时，再进行迭代就毫无益处了。而可计算量是$\tilde{\tau}_h$，那么τ和$\tilde{\tau}_h$之间的关系是什么呢？对于二阶精确差分格式，典型情形

$$\tau=\mathcal{L}_h(u)-\mathcal{L}_h(u_h)=h^2\tau_2(x,y)+\cdots \tag{19.6.37}$$

假定解满足 $u_h=u+h^2u_2(x,y)+\cdots$，然后假定R的阶足够高，以至于可以忽略它的作用，则(19.6.32)式得出

$$\begin{aligned}\tau_h&\simeq\mathcal{L}_H(u+h^2u_2)-\mathcal{L}_h(u+h^2u_2)\\&=\mathcal{L}_H(u)-\mathcal{L}_h(u)+h^2[\mathcal{L}'_H(u_2)-\mathcal{L}'_h(u_2)]+\cdots\\&=(H^2-h^2)\tau_2+O(h^4)\end{aligned} \tag{19.6.38}$$

对于通常情形 $H=2h$，可以得到

$$\tau\simeq\tfrac{1}{3}\tau_h\simeq\tfrac{1}{3}\tilde{\tau}_h \tag{19.6.39}$$

因此，停止的标准是(19.6.36)式中

$$\epsilon=\alpha\|\tilde{\tau}_h\|,\qquad \alpha\sim\tfrac{1}{3} \tag{19.6.40}$$

在实现非线性多重网格算法前，还有一项工作：选取一个非线性松弛格式。首先可能的选择仍是非线性 Gauss-Seidel 格式。如果离散化方程(19.6.23)在选定排序后，写为

$$L_i(u_0,\ldots,u_{N-1})=f_i,\qquad i=0,\ldots,N-1 \tag{19.6.41}$$

则非线性 Gauss-Seidel 格式为求 u_i^{new} 而解下述方程：

$$L_i(u_0,\ldots,u_{i-1},u_i^{\text{new}},u_{i+1},\ldots,u_{N-1})=f_i \tag{19.6.42}$$

和以前一样，一旦新 u 计算出来就要代替旧 u。对于 u_i^{new}，(19.6.42)式通常是线性的，这是因为非线性项通过邻近项被离散化了。如果不是这样，用一步牛顿迭代来代替方程(19.6.42)

$$u_i^{\text{new}}=u_i^{\text{old}}-\frac{L_i(u_i^{\text{old}})-f_i}{\partial L_i(u_i^{\text{old}})/\partial u_i} \tag{19.6.43}$$

例如，考虑简单的非线性方程

$$\nabla^2u+u^2=\rho \tag{19.6.44}$$

用二维记号，得到

$$\mathcal{L}(u_{i,j})=(u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}-4u_{i,j})/h^2+u_{i,j}^2-\rho_{i,j}=0 \tag{19.6.45}$$

由于

$$\frac{\partial\mathcal{L}}{\partial u_{i,j}}=-4/h^2+2u_{i,j} \tag{19.6.46}$$

因此，牛顿 Gauss-Seidel 迭代为

$$u_{i,j}^{\text{new}}=u_{i,j}-\frac{\mathcal{L}(u_{i,j})}{-4/h^2+2u_{i,j}} \tag{19.6.47}$$

下面的程序 mgfas 利用完全多重网格算法及 FAS 格式求解(19.6.44)式。限制和延拓与 mglin 中类似。

收敛试验根据(19.6.40)式进行。某个问题的一个成功的多网格解应该以满足该条件为目标，并使V循环数的最大值maxcyc等于1或2。mgfas程序中，用到的copy，interp及rstrct函数与mglin中的函数一样。

```
#include "nr.h"

namespace {
    void mg(const int j, Mat_IO_DP &u, Mat_I_DP &rhs, Vec_Mat_DP_p &rho,
        DP &trerr)
    递归多重网格法。输入j为当前层，u为当前解的值。对于给定层的第一次调用，右端项为0，而且自变量rhs是响
    的，这一步通过输入正的trerr来表示。随后的递归调用提供非零的rhs，如同在方程(19.6.33)。这一步通过输入
    正的trerr来表示。在每一层，rho是指向ρ的指针向量。在输出时，u包含当前层上的被改善的解。当第一次在给
    定层上调用时，相对截断误差τ返回于trerr中。
    {
        using namespace NR;
        const int NPRE=1,NPOST=1;
        粗网格修正前后松驰扫描的步数。
        const DP ALPHA=0.33;                估计截断误差与残差的范数的相关数。
        int jpost,jpre,nc,nf;
        DP dum=-1.0;

        nf=u.nrows();
        nc=(nf+1)/2;
        Mat_DP temp(nf,nf);
        if (j == 0) {                       V的底部：在最粗网格上求解。
            matadd(rhs,*rho[j],temp);
            slvsm2(u,temp);
        } else {                            V的向下行程。
            Mat_DP v(nc,nc),ut(nc,nc),tau(nc,nc),tempc(nc,nc);
            for (jpre=0;jpre<NPRE;jpre++) {          预光滑处理。
                if (trerr < 0.0) {
                    matadd(rhs,*rho[j],temp);
                    relax2(u,temp);
                }
                else
                    relax2(u,*rho[j]);
            }
            rstrct(ut,u);                   $\mathcal{R}\tilde{u}_h$.
            copy(v,ut);                     复制到v中。
            lop(tau,ut);                    $\mathcal{L}_H(\mathcal{R}\tilde{u}_h)$暂存于$\tilde{\tau}_h$中。
            lop(temp,u);                    $\mathcal{L}_h(\tilde{u}_h)$.
            if (trerr < 0.0)                $\mathcal{L}_h(\tilde{u}_h) - f_h$.
                matsub(temp,rhs,temp);
            rstrct(tempc,temp);             $\mathcal{R}\mathcal{L}_h(\tilde{u}_h) - f_H$.
            matsub(tau,tempc,tau);          $\tilde{\tau}_h + f_H = \mathcal{L}_H(\mathcal{R}\tilde{u}_h) - \mathcal{R}\mathcal{L}_h(\tilde{u}_h) + f_H$.
            if (trerr > 0.0)
                trerr=ALPHA*anorm2(tau);    估计截断误差。
            mg(j-1,v,tau,rho,dum);          递归调用粗网格修正。
            matsub(v,ut,tempc);             V的向上行程：令$\tilde{u}_h^{new} = \tilde{u}_h + \mathcal{P}(\tilde{u}_H - \mathcal{R}\tilde{u}_h)$。
            interp(temp,tempc);
            matadd(u,temp,u);
            for (jpost=0;jpost<NPOST;jpost++) {      后光滑处理。
                if (trerr < 0.0) {
                    matadd(rhs,*rho[j],temp);
                    relax2(u,temp);
                }
                else
                    relax2(u,*rho[j]);
            }
        }
    }
}

void NR::mgfas(Mat_IO_DP &u, const int maxcyc)
```

用完全多重网格算法求解非线性椭圆方程的FAS解，方程如(19.6.44)。输入u[0..n-1][0..n-1]中包含右端项，输出则返回解。维数n必须是2^j+1的形式，其中j为某个整数(实际上，j是解中用到的网格层数，程序中称为ng)。maxcyc是每层网格中，使用V循环的最大次数。

```
{
    int j,jcycle,ng=0,ngrid,nn;
    DP res,trerr;
    Mat_DP *uj,*uj1;

    int n=u.nrows();
    nn=n;
    while (nn >>= 1) ng++;
    if ((n-1) != (1 << ng))
        nrerror("n-1 must be a power of 2 in mgfas.");
    Vec_Mat_DP_p rho(ng);                    在每一层上，rho是指向ρ的指针向量。
    nn=n;
    ngrid=ng-1;
    rho[ngrid]=new Mat_DP(nn,nn);            对网格ng-1上的rhs分配存储空间，并赋值。
    copy(*rho[ngrid],u);
    while (nn > 3) {                         类似地对所有粗网格上的rhs分配空间，并通
        nn=nn/2+1;                           过细网格上的限制进行赋值。
        rho[--ngrid]=new Mat_DP(nn,nn);
        rstrct(*rho[ngrid],*rho[ngrid+1]);
    }
    nn=3;
    uj=new Mat_DP(nn,nn);
    slvsm2(*uj,*rho[0]);                     最粗网格上的初始解。
    for (j=1;j<ng;j++) {                     嵌套迭代循环。
        nn=2*nn-1;
        uj1=uj;
        uj=new Mat_DP(nn,nn);
        Mat_DP temp(nn,nn);
        interp(*uj,*uj1);                    从网格j-1到更细网格j的插值。
        delete uj1;
        for (jcycle=0;jcycle<maxcyc;jcycle++) {          V循环。
            trerr=1.0;
            mg(j,*uj,temp,rho,trerr);
            lop(temp,*uj);                   构造残参‖$d_h$‖。
            matsub(temp,*rho[j],temp);
            res=anorm2(temp);
            if (res < trerr) break;          如果残差足够小，就不再进行V循环。
        }
    }
    copy(u,*uj);                             返回u中的解。
    delete uj;
    for (j=0;j<ng;j++)
        delete rho[j];
}


#include "nr.h"

void NR::relax2(Mat_IO_DP &u, Mat_I_DP &rhs)
红－黑 Gauss－Seidel 松弛方程(19.6.44)．利用右端项函数 rhs[0..n－1][0..n－1]更新解 u[0..n－1][0..n－1 的当前值．
{
    int i,ipass,isw,j,jsw=1;
    DP foh2,h,h2i,res;

    int n=u.nrows();
    h=1.0/(n-1);
    h2i=1.0/(h*h);
    foh2 = -4.0*h2i;
    for (ipass=0;ipass<2;ipass++,jsw=3-jsw) {                红和黑扫描。
        isw=jsw;
        for (j=1;j<n-1;j++,isw=3-isw) {
            for (i=isw;i<n-1;i+=2) {
                res=h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                    4.0*u[i][j])+u[i][j]*u[i][j]-rhs[i][j];
                u[i][j] -= res/(foh2+2.0*u[i][j]);          Newton Gauss-Seidel公式。
            }
        }
    }
}
```

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::slvsm2(Mat_O_DP &u, Mat_I_DP &rhs)
最粗网格上求解方程(19.6.44),其中 h = 1/2.右端项输入到 rhs[0..2][0..2],并且解返回到 u[0..2][0..2]中。
{
    int i,j;
    DP disc,fact,h=0.5;

    for (i=0;i<3;i++)
        for (j=0;j<3;j++)
            u[i][j]=0.0;
    fact=2.0/(h*h);
    disc=sqrt(fact*fact+rhs[1][1]);
    u[1][1]= -rhs[1][1]/(fact+disc);
}


#include "nr.h"

void NR::lop(Mat_O_DP &out, Mat_I_DP &u)
给定 u[0..n-1][0..n-1],将方程(19.6.44)的$\mathcal{L}_h(\tilde{u}_h)$返回到 out[0..n-1][0..n-1]。
{
    int i,j;
    DP h,h2i;

    int n=u.nrows();
    h=1.0/(n-1);
    h2i=1.0/(h*h);
    for (j=1;j<n-1;j++)            内部点。
        for (i=1;i<n-1;i++)
            out[i][j]=h2i*(u[i+1][j]+u[i-1][j]+u[i][j+1]+u[i][j-1]-
                4.0*u[i][j])+u[i][j]*u[i][j];
    for (i=0;i<n;i++)              边界点。
        out[i][0]=out[i][n-1]=out[0][i]=out[n-1][i]=0.0;
}

#include "nr.h"

void NR::matadd(Mat_I_DP &a, Mat_I_DP &b, Mat_O_DP &c)
将 a[0..n-1][0..n-1]和 b[0..n-1][0..n-1]相加,并将结果返回到 c[0..n-1][0..n-1]中。
{
    int i,j;

    int n=a.nrows();
    for (j=0;j<n;j++)
        for (i=0;i<n;i++)
            c[i][j]=a[i][j]+b[i][j];
}

#include "nr.h"

void NR::matsub(Mat_I_DP &a, Mat_I_DP &b, Mat_O_DP &c)
从 a[0..n-1][0..n-1]中减去 b[0..n-1][0..n-1],并将结果返回到 c[0..n-1][0..n-1]。
{
    int i,j;

    int n=a.nrows();
    for (j=0;j<n;j++)
        for (i=0;i<n;i++)
            c[i][j]=a[i][j]-b[i][j];
}


#include <cmath>
#include "nr.h"
using namespace std;

DP NR::anorm2(Mat_I_DP &a)
```

```
返回矩阵 a[1..n][1..n]的欧几里得范数。
{
    int i,j;
    DP sum=0.0;

    int n=a.nrows();
    for (j=0;j<n;j++)
        for (i=0;i<n;i++)
            sum += a[i][j]*a[i][j];
    return sqrt(sum)/n;
}
```

参考文献和深入阅读

Brandt, A. 1977, *Mathematics of Computation*, vol. 31, pp. 333–390. [1]

Hackbusch, W. 1985, *Multi-Grid Methods and Applications* (New York: Springer-Verlag). [2]

Stuben, K., and Trottenberg, U. 1982, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds. (Springer Lecture Notes in Mathematics No. 960) (New York: Springer-Verlag), pp. 1–176. [3]

Brandt, A. 1982, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds. (Springer Lecture Notes in Mathematics No. 960) (New York: Springer-Verlag). [4]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill).

Briggs, W.L. 1987, *A Multigrid Tutorial* (Philadelphia: S.I.A.M.).

Jespersen, D. 1984, *Multigrid Methods for Partial Differential Equations* (Washington: Mathematical Association of America).

McCormick, S.F. (ed.) 1988, *Multigrid Methods: Theory, Applications, and Supercomputing* (New York: Marcel Dekker).

Hackbusch, W., and Trottenberg, U. (eds.) 1991, *Multigrid Methods III* (Boston: Birkhauser).

Wesseling, P. 1992, *An Introduction to Multigrid Methods* (New York: Wiley).

第 20 章　非典型的数值算法

20.0　引言

迄今为止，已完成了**数值算法**的论述，本章是一类“**非典型的**数值算法”的特集。出于种种原因，我们还是决定把它包括在这本以**典型**的数值算法为主的书中。我们注意到，计算机科学教材的作者喜欢在他们的书中增加一些数值计算的内容(通常是一类呆板的内容——例如求积分)，本书也不例外。

本书对素材的选择并不是完全随意的。其中一个主题即格雷码，已经用于构造伪随机序列(见 7.7 节)，在此只需做一些补充说明。另外两个主题，讨论了计算机浮点参数的诊断和任意精度的运算，它能增进对计算机的认识，即计算机更善于处理数(相对于字符和二进位)。后者这个主题，还显示了第 12 章快速傅里叶变换的一个非常不同的用法。

另外三个主题(校验和式、霍夫曼编码和算术码)涉及了数据编码、压缩和认证的不同方面。如果需处理大量数据——甚至是数值数据——那么顺便熟悉一下这些内容可能会有帮助。例如，在 13.6 节中，已遇到霍夫曼编码的一个很好的运用。

但要再次说明，可不必阅读本章内容。

20.1　诊断机器的参数

一个简单的假设是，计算机的浮点运算是“足够精确”的。如果相信这条假设，则数值分析就变成非常清楚的课题。不考虑舍入误差，使许多有限算法变得“精确”，所算结果和相当好的计算结果之间只差了一个截断误差(见 1.4 节)。这样说是不太实际的。

尽管如此，它仍贯穿于本书大部分内容中，是我们所必须接受的假设。对于前面所讨论过的每一种算法，它们的舍入误差是如何传递或限定的？要回答好这个问题是不现实的，事实上也是不可能的，对许多实际算法都没有做过严格的分析。

当听到用户说，“我遇到了单精度的舍入误差，所以我改成了双精度”，这使真正的数值分析员畏缩了。这句话的实际意思是：“对这个特定的算法以及某特定数据来说，双精度似乎能修补我在上述简单假设中的错误信念”。我们承认在本书中，所提到的**数值算法**中的精度和舍入误差大部分在特性上稍微更定量些，这是由于我们要求“实用”所引起的。

了解自己机器的浮点运算的实际限度是很重要的，当用户直觉地、实验性地或随意地处理浮点舍入误差时尤其重要。实验性地确定有用的浮点参数的方法，已由 Cody[1]、Malcolm[2]和其他人得出。Cody 所提供的实施办法由后面的程序 machar 来具体实现。许多浮点参数可以通过在标准程序库限度内调用类似 numeric_limits < double > ::epsilon( )的函数来获得，所以像 machar 这样的程序在实际中就不再那么重要了。

程序 machar 的所有变量都是返回值，它们的含义是：

- ibeta(在 1.3 节中称为 B)是数值计数法的基数，通常为 2，偶尔为 16 或甚至是 10。
- it 是浮点尾数 M 中基 ibeta 的位数(见图 1.4.1)。
- machep 是基 ibeta 的最小(通常是负的)幂次的指数值，它加 1.0 后，得到与 1.0 不同的数值。
- eps 是浮点数 $\text{ibeta}^{\text{machep}}$，它可粗略看做为“浮点精度”。
- negep 是 ibeta 的最小幂次的指数值，减 1.0 得到与 1.0 不同的数值。
- epsneg 就是 $\text{ibeta}^{\text{negep}}$，另一种决定浮点精度的方法。通常，epsneg 是 eps 的 0.5 倍，偶尔 eps 和 epsneg 相等。
- iexp 是指数值的位数(包括它的符号和偏置)。
- minexp 是尾数不以零开头的 ibeta 的最小(通常是负的)幂。

- xmin 是浮点数 $ibeta^{maxexp}$，通常是最小可用的浮点数值。
- maxexp 是导致溢出的 ibeta 的最小(正的)幂。
- xmax 是 $(1-epsneg)\times ibeta^{maxexp}$，通常是最大可用浮点数值。
- irnd 返回一个范围从 0 至 5 的代码，它给出使用何种舍入方法及如何处理下溢的信息，参见下文。
- ngrd 是当截断两尾数的乘积以适应计数法时所用的"保护位"位数。

在像 machar 这样的程序中，有许多细微之处，对用户来说其目的是找出对用户透明的机器特性。另外之所以必须这样做，是为了避免像上溢和下溢这种可能中断程序执行的错误情况发生。在有些情况下，程序只有靠辨认"标准"表示法的某些特征来实现这一点。例如，它根据舍入方式来辨认 IEEE 标准表示法[3]，并且假设该指数表示法的某些特征作为结果。详细内容请参阅本节参考文献[1]及有关的参考文献。注意，在非标准机器上程序 machar 可能给出不正确的结果。

关于参数 irnd 还需一些附加说明。在 IEEE 标准中，二进制模式对应于精确的、可表示的数值。加法舍入的具体方法是，把两个可表示的值"精确"地加起来，然后把和数舍入成最邻近的可表示的数值。如果这个和数精确地处在两个可表示的数中间，则应该将它舍入为那个偶数(即低位二进制数为零)。对于所有其他的运算操作应保持同样的方法，也就是说，应把它们等同于无限精确来处理，然后舍入到最邻近的可表示的数值。

如果 irnd 返回 2 或 5，则用户的计算机就是服从这一标准的。如果它返回 1 或 4，则就是采用 IEEE 标准以外的舍入方法。如果它返回 0 或 3，则它截断了结果而不是舍入结果，一般不希望出现这种情况。

由 irnd 引发的另一个问题涉及到下溢。如果一个浮点值小于 xmin，许多计算机将其值下溢至零。若 irnd 的值等于 0，1 或 2，这就说明计算机采用的是这种方式。IEEE 标准规定了一种更适度的下溢：当一个数值变得小于 xmin 时，它的指数值就固定在最小允许值上，而它的尾数下降，使尾数开头处为零并且"适度地"丢失精度。若 irnd = 3，4 或 5 时，表明采用的是这种方式。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::machar(int &ibeta, int &it, int &irnd, int &ngrd, int &machep,
    int &negep, int &iexp, int &minexp, int &maxexp, DP &eps, DP &epsneg,
    DP &xmin, DP &xmax)
确定和返回影响浮点运算的机器的特性参数。返回值包括：ibeta，浮点数的基数；it，浮点型尾数基数 ibeta 的位数；eps，加
到 1.0 上后不等于 1.0 的最小正数；epsneg，减去 1.0 后不等于 1.0 的最小正数；xmin，最小可表示的正数；xmax，最大可表
示的正数；其他返回参数的说明参见正文。将 DP 变成浮点型以找到单精度参数。
{
    int i,itemp,iz,j,k,mx,nxres;
    DP a,b,beta,betah,betain,one,t,temp,temp1,tempa,two,y,z,zero;

    one=DP(1);
    two=one+one;
    zero=one-one;
    a=one;                          采用 M Malcolm 的方法确定 ibeta 和 beta。
    do {
        a += a;
        temp=a+one;
        temp1=temp-a;
    } while (temp1-one == zero);
    b=one;
    do {
        b += b;
        temp=a+b;
        itemp=int(temp-a);
    } while (itemp == 0);
    ibeta=itemp;
    beta=DP(ibeta);
    it=0;                           确定 it 和 irnd。
    b=one;
    do {
        ++it;
        b *= beta;
        temp=b+one;
        temp1=temp-b;
    } while (temp1-one == zero);
```

```
  irnd=0;
  betah=beta/two;
  temp=a+betah;
  if (temp-a != zero) irnd=1;
  tempa=a+beta;
  temp=tempa+betah;
  if (irnd == 0 && temp-tempa != zero) irnd=2;
  negep=it+3;                          确定 negep 和 epsneg。
  betain=one/beta;
  a=one;
  for (i=1;i<=negep;i++) a *= betain;
  b=a;
  for (;;) {
      temp=one-a;
      if (temp-one != zero) break;
      a *= beta;
      --negep;
  }
  negep = -negep;
  epsneg=a;
  machep = -it-3;                      确定 machep 和 eps。
a=b;
for (;;) {
    temp=one+a;
    if (temp-one != zero) break;
    a *= beta;
    ++machep;
}
eps=a;
ngrd=0;                                确定 ngrd。
temp=one+eps;
if (irnd == 0 && temp*one-one != zero) ngrd=1;
i=0;                                   确定 iexp。
k=1;
z=betain;
t=one+eps;
nxres=0;
for (;;) {                             循环直至下溢发生，然后退出。
    y=z;
    z=y*y;
    a=z*one;                           此处检查是否发生下溢。
    temp=z*t;
    if (a+a == zero || fabs(z) >= y) break;
    temp1=temp*betain;
    if (temp1*beta == z) break;
    ++i;
    k += k;
}
if (ibeta != 10) {
    iexp=i+1;
    mx=k+k;
} else {                               只用于十进制机器。
    iexp=2;
    iz=ibeta;
    while (k >= iz) {
        iz *= ibeta;
        ++iexp;
    }
    mx=iz+iz-1;
}
for (;;) {                             为确定 minexp 和 xmin，循环直至下溢发生然后退出。
    xmin=y;
    y *= betain;
    a=y*one;                           此处检查是否发生下溢。
    temp=y*t;
    if (a+a != zero && fabs(y) < xmin) {
        ++k;
        temp1=temp*betain;
        if (temp1*beta == y && temp != y) {
```

```
                nxres=3;
                xmin=y;
                break;
            }
        }
        else break;
    }
    minexp += -k;                          确定 maxexp, xmax。
    if (mx <= k+k-3 && ibeta != 10) {
        mx += mx;
        ++iexp;
    }
    maxexp=mx+minexp;
    irnd += nxres;                         调整 irnd 来反映部分下溢。
    if (irnd >= 2) maxexp -= 2;            为 IEEE 型机调整。
    i=maxexp+minexp;
    对在二进制尾数中具有隐含起首位的机器,以及基点在尾数最右端的机器进行调整.
    if (ibeta == 2 && !i) --maxexp;
    if (i > 20) --maxexp;
    if (a != y) maxexp -= 2;
    xmax=one-epsneg;
    if (xmax*one != xmax) xmax=one-beta*epsneg;
    xmax /= (xmin*beta*beta*beta);
    i=maxexp+minexp+3;
    for (j=1;j<=i;j++) {
        if (ibeta == 2) xmax += xmax;
        else xmax *= beta;
    }
}
```

表 20.1.1 给出了由 machar 返回的一些典型值。表中 IEEE 依从机是指大多数 UNIX 工作站(SUN、DEC、MIPS)及 Apple Macintosh II。带有嵌入芯片内部的或分离的浮点协处理器的奔腾(及类似)芯片,一般是 IEEE 依从机,但是有些编辑器会不适当地下溢出中间结果,导致 irnd = 2 而不是 5。注意,对于 VAX(第四列)的情形来说,它是采用在尾数中带有一位"幻象"的起首位的记数法,因此对于同样字长虽然产生一个较小的 eps,但不能适当地处理下溢

表 20.1.1　程序 machar 返回的典型结果

精　度	典型的 IEEE 依从机		DEC VAX
	单	双	单
ibeta	2	2	2
it	24	53	24
machep	-23	-52	-24
eps	1.19×10^{-7}	2.22×10^{-16}	5.96×10^{-8}
negep	-24	-53	-24
epsneg	5.96×10^{-8}	1.11×10^{-16}	5.96×10^{-8}
iexp	8	11	8
minexp	-126	-1022	-128
xmin	1.18×10^{-38}	2.23×10^{-308}	2.94×10^{-39}
maxexp	128	1024	127
xmax	3.40×10^{38}	1.79×10^{308}	1.70×10^{38}
irnd	5	5	1
ngrd	0	0	0

参考文献和深入阅读

Goldberg, D. 1991, *ACM Computing Surveys*, vol. 23, pp. 5–48.

Cody, W.J. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 303–311. [1]

Malcolm, M.A. 1972, *Communications of the ACM*, vol. 15, pp. 949–951. [2]

IEEE Standard for Binary Floating-Point Numbers, ANSI/IEEE Std 754–1985 (New York: IEEE, 1985). [3]

20.2 格雷码

格雷码是整数 i 的一个函数 $G(i)$，对于每一个整数 $N \geq 0$，有与 $0 < i \leq 2^N - 1$ 一一对应的 $G(i)$。该函数有以下显著特性：表达 $G(i)$ 和 $G(i+1)$ 的二进制数中只有一位二进数不同。格雷码的一个(实际上是最常用的)例子是序列 0000，0001，0011，0010，0110，0111，0101，0100，1100，1101，1111，1110，1010，1011 和 1001，分别对应于 $i=0,\ldots,15$。产生这个码的算法是，简单地求 i 与 $i/2$(整数部分)的逐位异或(XOR)(参见图 20.2.1(a))。请想一想，当把 1 加到一个二进制数上时，加法是如何进行的，就会明白异或怎么进行。读者将看到：$G(i)$ 和 $G(i+1)$ 不同之处在于对应 i 的最右端零位的位置(如有必要，可前置一个起首的零)。

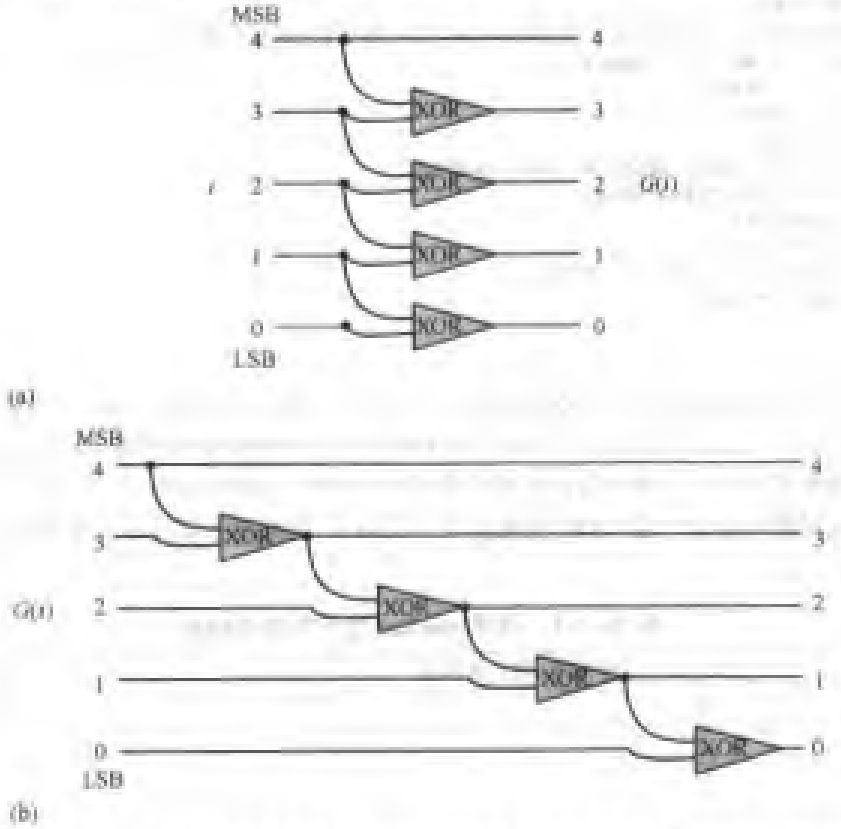


图 20.2.1 (a)从 i 计算格雷码 $G(i)$ 的逐位运算；(b)格雷码的译码运算((a)的逆运算)。其中，LSB和MSB分别表示最低和最高有效位。XOR表示异或

格雷的英文拼写是"Gray"，而不是"gray"，这种码是因 Frank Gray 而得名的，是他第一个申请关于使用轴轮或编码器想法的专利。这种编码器是一个带有许多同心编码条纹的轮子，每一个条纹的码位经固定导电刷而"读出"。这种想法是，产生一个描述轮子角度的二进制码。制造轴轮式编码器的一个显然但却错误的方法是，使轮子中一个(例如最内部的)条纹的一半与电刷导通，而另一半与电刷绝缘；下一个条纹在一、三象限导通；再下一个条纹在一、三、五、七的八等分区间内导通，依此类推。所有导电刷加在一起，就直接读出了轮子位置的二进制码。

这种方法之所以差，是因为不能保证轮子转动时所有电刷能精确同步地接通或断开。当从位置 7(0111)到位置 8(1000)时，由于不同导电刷不能同时导通或断开，就造成了编码虚假，会短暂地出现 6(0110)，14(1110)和10(1010)。而用格雷码编制条纹，可以确保在 7(上述序列中的 0100)与 8(1100)之间没有临时状态出现。

当然，需要用电路或算法把 $G(i)$ 译回成 i。图 20.2.1(b)显示了怎样通过异或门的级联来完成此过程。想法是，每一个输出位都应是所有有效输入位的异或。对 N 位格雷码进行译码，在电路中需要经过 $N-1$ 步(或者门延迟)(尽管如此，这在电路中通常是非常快的)。在一个具有一字节宽度的二进制运算的寄

存器中，不必做 N 次连续运算，而只需做 $\ln_2 N$ 次。其中的技巧是，使用异或联合，并有层次地把运算分组。这涉及到按 1,2,4,8,...位顺序右移直到字长结束，下面是一段求格雷码及其译码的程序。

```
#include "nr.h"

unsigned long NR::igray(const unsigned long n, const int is)
当 is 值为正或零，返回 n 的格雷码；若 is 为负，则返回格雷码的译码 n
{
    int ish,
    unsigned long ans,idiv;

    if (is >= 0)                    这是简单方向。
        return n ^ (n >> 1);
    ish=1,                          这是较复杂的方向：在层次阶段中，从一个一位右移开
    ans=n;                          始，进行每一位与所有较高有效位相异或
    for (;;) {
        ans ^= (idiv=ans >> ish);
        if (idiv <= 1 || ish == 16) return ans;
        ish <<= 1;                  在下一轮循环中加倍移位的量。
    }
}
```

在数值计算中，当执行的任务密切地依赖于 i 的二进制数，并且需对许多 i 值循环执行时，格雷码是非常有用的。如果对只相差一位二进位的值重复执行某任务能提高效率的话，那么使用格雷码顺序而不是使用一般二进数的连续顺序将会很有意义。在 7.7 节中，已看到了用格雷码顺序产生伪随机序列的例子。

参考文献和深入阅读

Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (New York: Cambridge University Press), §8.02.

Knuth, D.E. *Combinatorial Algorithms*, vol. 4 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §7.2.1. [Unpublished. Will it be always so?]

20.3　循环冗余度校验和其他的校验和式

当从 A 传送一串二元序列到 B 时，发送者希望知道它是否能准确无误地传送到达 B。一个通常的可靠方式是，在 7 位的 ASCII 字符上附加上位"奇偶校验码"，使它们变成 8 位格式。这位奇偶校验码的选择是，使序列中含"1"位（相对于"0"位）的总数或总是为偶（"偶校验"）或总是为奇（"奇校验"）。因此，就能检测出字符中的任何单一位的错误。当错误足够少，并且在时间上不是成群出现时，使用奇偶校验码就可提供足够的错误检测。

遗憾的是，在实际情况中，单独噪声"事件"可能会破坏不止一位。由于奇偶校验码有两个可能值（0 和 1），因此平均地说，检测出多于一个错误二进位的错字符只有 50% 的机会。对大多数应用来说，50% 的检错概率都是不够好的。大多数通信协议（参见本节参考文献[1]）中，使用对奇偶位的一种多位推广，这种推广称为"循环冗余度校验"（cyclic redundancy check）或 CRC。在典型的应用中，CRC 有 16 位长（两个字节或两个字符），所以发生未被检测到随机错误的机会是 $2^{16} = 65\,536$ 分之一。而且对任意长度的消息来说，M 位 CRC 具有能检测出所有小于等于 M 个连续位中错误的数学特性（这一点将在下面证明）。由于通信信道中噪声倾向于"突发"，伴随相连位的短序列被破坏，因此这种连续位纠错特性是非常需要的。

通常，CRC 出现在通信软件专家和芯片设计者——他们对二进数十分了解——的工作领域中。但是，至少在两种情况下，了解关于 CRC 会对我们有所帮助。首先，有时需要能与低层硬件片或软件段通信，这些硬件片或软件段需要有效的 CRC 作为其输入的一部分。例如，用一个程序把 XMODEM 或 Kermit[2]软件包直接连接到通信线中，而不必将数据存于局部文件中，这将是非常方便的。

第二，在大量（例如实验的）数据的处理中，能用数据集合（数字、记录、行或完整文件）的统计上独有的"关键码"——它的 CRC 来对其进行标记，这也是有用的。这样一来，任意大小的集合都可通过只比较其短的 CRC 关键码来比较其一致性。不同的关键码意味着不同的记录。一致的关键码，在很高的统计确定性上，意味着一致的记录。若无法接受小的错误概率，那么在关键码一致时还可做记录的完整比较。当文件或数据

记录有可能被偶然地或不负责任地修改(例如计算机病毒)时,把它们先验的 CRC 存储于外部的、物理上安全的介质(如软盘上)是很有用的。

有时,CRC 可用于记录数据时对其进行压缩。如果一致的数据记录经常出现,就可将原先遇到的记录的 CRC 存于内存。如果一个新记录的 CRC 是不同的,则就将新记录完整地归档。否则,就只有一个指针指向原先需归档的记录。在这项应用中,需要一个 4 或 8 字节的 CRC,以使错误地放弃一个不同的数据记录的可能性小到可接受的程度。或者,如果能够随机地存取原先的记录,则可以做一个完整的比较,以决定具有一致 CRC 的记录是否真的一致。

现在,简要地讨论 CRC 理论,然后将提供各种(有关的)CRC 的应用,这些 CRC 用于正式的或事实上的标准协议中[1~3],参见表 20.3.1。

表 20.3.1 各种 CRC 协议的约定和测试值

	icrc 参数		测试值(C_2C_1 为十六进制数)		包	
协议	jinit	jrev	T	CatMouse987654321	格式	CRC
XMODEM	0	1	1A71	E556	$S_1S_2\ldots S_NC_2C_1$	0
X.25	255	-1	1B26	F56E	$S_1S_2\ldots S_N\overline{C_1C_2}$	F0B8
(无名)	255	-1	1B26	F56E	$S_1S_2\ldots S_NC_1C_2$	0
SDLC(IBM)	与 X.25 同					
HDLC(ISO)	与 X.25 同					
CRC-CCITT	0	-1	14A1	C28D	$S_1S_2\ldots S_NC_1C_2$	0
(无名)	0	-1	14A1	C28D	$S_1S_2\ldots S_N\overline{C_1C_2}$	F0B8
Kermit	与 CRC-CCITT 同				见注释	

注释:上划线表示补码。$S_1\ldots S_N$ 是字符数据。C_1 是 CRC 的 8 位最低有效位,C_2 是 8 位最高有效位,因此 $CRC=256C_2+C_1$(用十六进制表示)。Kermit(块校校验水平 3)把 CRC 作为 3 个可打印的 ASCII 字符传送(传送值+32)。这些分别包含 4 位最高有效位、6 位中间位、6 位最低有效位。

CRC 的数学基础是"模 2 整数的多项式"。任何二进制消息可被认为是系数为 0 和 1 的多项式。例如,消息"1100001101"是多项式 $x^9+x^8+x^3+x^2+1$。由于 0 和 1 是模 2 仅有的整数,所以多项式中 x 的幂次项或者存在(1),或者不存在(0)。模 2 整数的多项式可能是不可约的,也就是说它不能被因式分解。不可约多项式的一个子集是"本原"多项式。这种多项式当被用于移位寄存器(如 7.4 节所述)中时能产生最大长度的序列。多项式 x^2+1 不是不可约:$x^2+1=(x+1)(x+1)$,所以它也不是本原的。多项式 $x^4+x^3+x^2+x+1$ 不可约,但它被证明不是本原的。多项式 x^4+x+1 既不可约又是本原的。

一个 M 位长的 CRC 是基于一个特殊的 M 次本原多项式,称它为生成多项式。生成多项式是可以选择的,我们可以选定它为一个本原多项式乘以$(1+x)$(这可查出全部奇偶错误)。对于 16 位 CRC,CCITT(Comité Consultatif International Télégraphique et Téléphonique)已规定"CCITT 多项式"为 $x^{16}+x^{12}+x^5+1$。列在表 20.3.1 中的所有协议都使用这个多项式。另一个通常选择是"CRC-16"多项式 $x^{16}+x^{15}+x^2+1$。它被用于 IBM 的 BISYNCH(参见本节参考文献[1])的 EBCDIC 消息中。常见的 12 位选择"CRC-12"是 $x^{12}+x^{11}+x^3+x+1$。常见的 32 位选择"AUTODIN-Ⅱ"是 $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$。关于一些其他本原多项式的表格,参见 7.4 节。

已知 M 次生成多项式 G(可被写成多项式形式或二进位序列,例如 CCITT 多项式可写成 10001000000100001),计算一个二进制序列 S 的 CRC 的方法如下:首先,用 x^M 乘 S,也就是把 M 个零位添加到 S 序列后面。第二,用 G 长除 Sx^M。记住长除中的减法是模 2 运算,因此永远不会有"借位":模 2 减法与逻辑上的异或(XOR)相同。第三,忽略所得的商数。第四,最终得到一个余数,这就是循环冗余校验式,称为 C。C 将是一个小于等于 $M-1$ 次的多项式,否则就还没有完成长除。因此,在二进位序列形式中,循环冗余校验式 C 的二进序列有 M 位,可能包括起首的一些零(C 甚至可能全为零,见下文),实例见本节参考文献[3]。

如果在一个例子中按上述步骤进行,读者会看到,在长除表中写下的东西大部分是多余的,实际上,只需把 S 的连续位从右向左移至一个 M 位的寄存器中。每一次有一位移出寄存器的左端,通过与 G 的 M 个低位

(也就是说,除起首 1 以外的所有 G 的位)进行异或,改变这个寄存器。当一个零位移出寄存器左端时,寄存器并不改变。当 S 的原序列部分的最后一位移出寄存器左端后,寄存器中所剩下的就是 S 的 CRC。

由此看到,这些过程用硬件实现是很有效的。它只要求一个连接一些异或门的移位寄存器。这就是在通信设施中,由单片芯片(或一片芯片的一小部分)来计算 CRC 的过程。在软件中,实现起来就没有这么容易了,因为二进位的移动一般不是很有效。因此,人们发明(正如下面的实施)了表格驱动程序,它可预先计算一系列移位和异或的结果。例如,8 位输入中所有 256 种可能的序列的每一种(参见本节参考文献[4])。

现在,讨论 CRC 如何获得对 M 个连续位中所有错误的检测能力。假设有两则消息 S 和 T,它们的区别只存在于 M 位框架之内。于是,它们的 CRC 只相差一个数,即 G 除 $(S-T)x^M \equiv D$ 的余数。现在,D 的形式是起首位置是一些零(可忽略),紧跟着的是 M 位框架中的一些 1,后跟尾随的零(正好是 x 的乘法因子):$D = x^n F$,其中 F 是至多 $M-1$ 次的多项式,而且 $n>0$。由于 G 总是本原的,或者是本原的 $(1+x)$ 倍,所以 G 就不能被 x 除尽。故 G 不能被 D 除尽。因此,S 和 T 一定具有不同的 CRC。

在许多协议中,一个传递的数据块由 N 个数据位和后面紧跟着它们的 M 位循环冗余校验码(或与一个常数异或的 CRC,见下文)组成。在接收端,恢复数据块有两种等价方法。最明显的方法是,在接收端计算数据位的 CRC,并将其与传递的 CRC 相比较。另一种不太明显但更精巧的方法是,接收端简单地计算整个 $N+M$ 位数据块的 CRC,并确证得到了零结果。证明:整个数据块是多项式 $Sx^M + C$(数据左移,以留出 CRC 位的位置),而 C 的定义是 $Sx^M = QG + C$,其中 Q 是被放弃的商数。但此时,$Sx^M + C = QG + C + C = QG$(请记住,模 2),它是 G 的完整倍数。因此,当在接收端,将整个数据块乘以附加的 x^M 时,它仍旧是 G 的乘积,因此它的 CRC 为零,证毕。

这个基本程序中有一两处小变化需要提一下(参见本节参考文献[1,3])。首先,在计算 CRC 时,M 位寄存器不需初始化为零。将它初始化为其他 M 位值(例如全为 1),其在效果上等于所有块的前端加了一个虚假消息,此信息本应将初始化值作为余值给出。这样做是有优点的,因为不如此描述 CRC 就不能检测出任何起始零位个数的增减(丢失起始位,或插入零位是普通的"时钟错误")。第二,可以在 CRC 被传递之前加上(异或)任何 M 位常数 K。这个常数或在接收端被异或掉,或只是对全部块的所期望的 CRC 改变一个已知量,即 G 除 Kx^M 的余数。常数 K 经常是"全部位",它把 CRC 变换成其补码。这样做有一个优点就是,能检测出 CRC 不能发现的另一种错误:在块结尾处突然插入一位"1",而使消息的一个起始位为"1"的位缺失。

下面函数 icrc 实现了上述 CRC 计算,并且包括了所提及的变化的可能性。该函数的输入是指向一个字符数组的指针以及该数组的长度。icrc 有两个"开关"参数,详细规定了 CRC 计算中的变化。jinit 取零值或正值,会引起 16 位寄存器用 jinit 值,对每一个字节进行初始化。jrev 取负值,会引起每一个输入字符被译成它的位倒置像(bit-reverse image),并且在输出的 CRC 中也完成相似的位的倒置。读者不必非理解这些内容,仅仅使用表 20.3.1 规定的 jinit 和 jrev 值即可(其简单解释是,串行数据端口是**先传送字符的最低有效位**,并且许多协议准确地按接收顺序将二进数移进 CRC 寄存器)。表 20.3.1 显示了如何由 icrc 的输入数组和输出 CRC 来构造一个字符块。无需在 icrc 之外做任何附加的位的倒置。

开关 jinit 有一个附加用途:当它为负,用数组 crc 的输入值初始化寄存器。如果设置 crc 为最后一次调用 icrc 的结果,这实际上是把当前输入数组连接到前一次(或前几次)调用中的数组上。举个例子,使用这个特性,可一次一行地为整个文件建起 CRC,而不必把整个文件都保存在内存中。

程序 icrc 基本是根据本节参考文献[4]中函数编写的,其操作可理解如下:首先看看函数 icrc1。它把一个输入字符输入一个 16 位 CRC 寄存器。这里用到的惟一技巧是,在寄存器每次移位时,把字符位异或成最高有效位,一次八位,而不是一次一位地送入最低有效位,并一次异或一位。这是因为异或是关联和可交换的——可以在决定是否用本原生成多项式改变寄存器之前的任何时候输进字符位(十进制常数 4129 就含有生成多项式的位)。

```
#include "nr.h"

unsigned short NR::icrc1(const unsigned short crc, const unsigned char onech)
给定当前的余数,返回一个字符加入后的新 CRC。此程序的功能等价于 icrc(..,-1,1),但速度较慢。一般它被 icrc 用来
初始化表格。
{
    int i;
    unsigned short ans=(crc ^ onech << 8);

    for (i=0;i<8;i++) {              在此完成 8 个一位位移,并与生成多项式异或。
```

```
            if (ans & 0x8000)
                ans = (ans <<= 1) ^ 4129;
            else
                ans <<= 1;
        }
        return ans;
    }
```

现在看一看 icrc。有两部分内容要理解,当初始化时如何建立表,此后如何使用该表。回想一下,把字符的位从最低有效位末端移入 CRC 寄存器。关键的监测是当 8 位二进数移进寄存器低端时,所有生成的要改变的位都已经由已存于高端的位所决定。由于异或是可交换和关联的,所以对每一个 256 种可能的高位序列,所需的是所有这种改变的结果表。然后,可以进行捕捉和把输入字符异或成这个数中查找的结果。icrc 的另一内容是,在初始化时从存于 it 中的 4 位表来构造一个 8 位位倒置的表,以及与执行位倒置有关的逻辑。本节参考文献[4~6]给出了表格驱动的 CRC 计算的细节。

```
#include "nr.h"

namespace {
    inline unsigned char lobyte(const unsigned short x)
    {
        return (unsigned char)((x) & 0xff);
    }

    inline unsigned char hibyte(const unsigned short x)
    {
        return (unsigned char)((x >> 8) & 0xff);
    }
}

unsigned short NR::icrc(const unsigned short crc, const string &bufptr,
    const short jinit, const int jrev)
对串 bufptr 计算其 16 位循环冗余校验,正如被 jinit 和 jrev(见表 20.3.1)的设置决定一样,它采用几种约定中的任一种进行计算。若 jinit 取负值,则 crc 作为输入以便初始化余数寄存器,结果是(对于被设置为最后一个返回值的 crc)将 bufptr 连接到上一次调用上。
{
    static unsigned short icrctb[256],init=0;
    static unsigned char rchr[256];
    unsigned short j,cword=crc;
    static unsigned char it[16]={0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15};
    4 位倒置后的数值表。

    unsigned long len=bufptr.length();
    if (init == 0) {                    需要初始化表格。
        init=1;
        for (j=0;j<256;j++) {
        这两个表是:所有字符的 CRC,和所有字符的位倒置。
            icrctb[j]=icrc1(j << 8,0);
            rchr[j]=(unsigned char)((it[j & 0xf] << 4) | (it[j >> 4]));
        }
    }
    if (jinit >= 0)                     初始化余数寄存器。
        cword=(jinit | (jinit << 8));
    else if (jrev < 0)                  如果不初始化,需倒置寄存器吗?
        cword=(rchr[hibyte(cword)] | (rchr[lobyte(cword)] << 8));
    for (j=0;j<len;j++) {               对数组中字符的主循环。
        cword=icrctb[(jrev < 0 ? rchr[(unsigned char) bufptr[j]] :
            (unsigned char) bufptr[j]) ^ hibyte(cword)] ^ (lobyte(cword) << 8);
    }
    return (jrev >= 0 ? cword :         需对输出倒置吗?
        rchr[hibyte(cword)] | (rchr[lobyte(cword)] << 8));
}
```

若需要 32 位校验和式怎么做呢?对一个真正的 32 位 CRC,需要用一个更长的生成多项式来重写这些程序。例如,$x^{32}+x^7+x^5+x^3+x^2+x+1$ 是模 2 本原多项式,并且(为了简单)仅在它的最低有效字节中具有非零的非起首位。因此只对 CRC 寄存器的最高有效位的字节进行查找的思想没有改变。

如果不关心校验和式的 M 个连续位的特性,而只是需要校检一个统计随机的 32 位,那么可使用此处给出的 icrc:以 jrev = 1 调用它一次以得到 16 位,再以 jrev = −1 调用一次得到另一个 16 位。内部的位倒置使得这两个 16 位 CRC 彼此完全独立。

20.3.1　其他种类的校验和式

与 CRC 非常不同的技术是，常常在人们处理的(例如键入计算机的)数字后附加一个十进制的"校验码"(check digit)，校验码需要用来防阻犯具有高度结构性的错误，例如传送连续数字。Wagner 和 Putter[7]就这个问题给出了有意义的介绍，包括详细的算法。

目前，所有广泛传播的校验和式，从适度到低劣程度不等。在大多数书上的 10 位 ISBN(国际标准书号)都使用校验方程

$$10d_1 + 9d_2 + 8d_3 + \cdots + 2d_9 + d_{10} = 0 \quad (\text{模 } 11) \tag{20.3.1}$$

其中 d_{10} 是右端检验位，字符"X"用于代表 10 的校验位值。另一种流行的格式是所谓的"IBM 校验"，常用于记乘账号(例如万事达卡)。这里，校验方程是

$$2\#d_1 + d_2 + 2\#d_3 + d_4 + \cdots = 0 \quad (\text{模 } 10) \tag{20.3.2}$$

其中 $2\#d$ 的含义是"用 2 乘 d 并加上所得的十进制数字"。美国银行码用一个 9 位数进行校验，它的校验方程是

$$3a_1 + 7a_2 + a_3 + 3a_4 + 7a_5 + a_6 + 3a_7 + 7a_8 + a_9 = 0 \quad (\text{模 } 10) \tag{20.3.3}$$

印在许多美国邮政服务信封上的条形码是通过去掉两端等高的标记条，然后 5 条一组，把剩余的条形编成 6 组或 10 组进行解码。在每一组中，5 个条形(从左到右)分别表示数值 7,4,2,1,0。它们当中准确地有两个是高的。它们之和就是所代表的数字，除了零是由 7+4 来代表的。5 位或 9 位的邮政编码后跟一个校验码，校验方程是

$$\sum d_i = 0 \quad (\text{模 } 10) \tag{20.3.4}$$

这些格式中，没有一个是接近最优的。Verhoeff 提出一个精致的格式，详细叙述参见本节参考文献[7]。其基本思想是使用十个元素的**二面体群** D_5——它对应于五边形的对称——而不是模 10 整数的循环群。这个校验的方程是

$$a_1 * f(a_2) * f^2(a_3) * \cdots * f^{n-1}(a_n) = 0 \tag{20.3.5}$$

其中 $*$ 是 D_5 中的(不可交换的)乘法，且 f^i 表示对某固定排列的第 i 次迭代。Verhoeff 的方法可发现一串字符中的**所有**单个错误和**所有**相邻换位错误。它还可发现大约 95% 的成对错误($aa \to bb$)、跳跃换位错误($acb \to bca$)和成对跳换错误($aca \to bcb$)。下面是一种实施方案。

```
#include "nr.h"

bool NR::decchk(string str, char &ch)
计算和验证十进制校验码。返回一个校验码 ch，将它附加到串 string[0..n-1]上，即存入 string[n]中。在本模式中，忽略
所返回的布尔(整数)值。若 string[0..n-1]已经用一校验码 string[n-1]做结尾，且校验码是有效的则返回函数值为真，
否则为假。在本模式中忽略 ch 的返回值。注意，string 和 ch 对应于数字 0~9 包含 ASCII 字符，不是 0~9 范围中的字节
值。string 中也允许有其他 ASCII 字符，并且在计算校验码时忽略这些字符。
{
    char c;
    int j,k=0,m=0;
    static int ip[10][8]={{0,1,5,8,9,4,2,7},{1,5,8,9,4,2,7,0},
        {2,7,0,1,5,8,9,4},{3,6,3,6,3,6,3,6},{4,2,7,0,1,5,8,9},
        {5,8,9,4,2,7,0,1},{6,3,6,3,6,3,6,3},{7,0,1,5,8,9,4,2},
        {8,9,4,2,7,0,1,5},{9,4,2,7,0,1,5,8}};
    static int ij[10][10]={{0,1,2,3,4,5,6,7,8,9},{1,2,3,4,0,6,7,8,9,5},
        {2,3,4,0,1,7,8,9,5,6},{3,4,0,1,2,8,9,5,6,7},{4,0,1,2,3,9,5,6,7,8},
        {5,9,8,7,6,0,4,3,2,1},{6,5,9,8,7,1,0,4,3,2},{7,6,5,9,8,2,1,0,4,3},
        {8,7,6,5,9,3,2,1,0,4},{9,8,7,6,5,4,3,2,1,0}};
        群乘和排列表。

    int n=str.length();
    for (j=0;j<n;j++) {                    查看连续字符。
        c=str[j];
        if (c >= 48 && c <= 57)            除数字外忽略其他。
```

```
            k=ij[k][ip[(c+2) % 10][7 & m++]];
        }
        for (j=0;j<10;j++)                    找出附加的一致校验数。
            if (ij[k][ip[j][m & 7]] == 0) break;
        ch=char(j+48);                        转换成 ASCII 码。
        return k==0;
    }
```

参考文献和深入阅读

McNamara, J.E. 1982, *Technical Aspects of Data Communication*, 2nd ed. (Bedford, MA: Digital Press). [1]

da Cruz, F. 1987, *Kermit, A File Transfer Protocol* (Bedford, MA: Digital Press). [2]

Morse, G. 1986, *Byte*, vol. 11, pp. 115–124 (September). [3]

LeVan, J. 1987, *Byte*, vol. 12, pp. 339–341 (November). [4]

Sarwate, D.V. 1988, *Communications of the ACM*, vol. 31, pp. 1008–1013. [5]

Griffiths, G., and Stones, G.C. 1987, *Communications of the ACM*, vol. 30, pp. 617–620. [6]

Wagner, N.R., and Putter, P.S. 1989, *Communications of the ACM*, vol. 32, pp. 106–110. [7]

20.4 霍夫曼编码与数据压缩

一个不丢失数据的压缩算法是将一串符号(一般为 ASCII 字符或字节)**可逆地**转换成另一串符号,转换后的符号在长度上**平均地**要短一些。"平均"的含意是很重要的,很显然没有可逆的算法能把所有的字符串都变短——因为没有足够多的短串与较长的串一一对应。只有当在输入端有一些串或输入符号比起其他的符号要更常用时,压缩算法才是可能的。于是这些常见的串用较少的位,而比较少见的串或符号用较多的位进行编码,这样在平均意义上长度为最短。

对应于不同的检测方法和在输入串中使用不同的偏离等概率方法,有许多差异非常大的压缩编码的技术。在本节和下一节中,我们只考虑以**定义字**输入的**变长码**。在定义字的输入中,输入被分成固定长度单位,例如 ASCII 字符,而相应的输出为可变长度块。这种最简单的方法就是本节中讨论的霍夫曼(Huffman)编码[1]。另一个例子,**算术压缩编码**将在 20.5 节中讨论。

与定义字相反,变长码把输入分成变长的单位(例如,英语课文中的单词或词组),然后经常以固定长度的输出码将它们传输。这种类型中,最常用的编码是 Ziv-Lempel 编码[2]。本节参考文献[3~6]给出了某些其他压缩技术,并列出了有关的大量参考文献。

霍夫曼编码的思想很简单,它是对较常用的字符使用较短的位样式(bit pattern)。根据**熵**的概念,可使这种思想定量化。假设输入字母表中有 N_{ch} 个字符,分别以概率 p_i($i=1,\ldots,N_{ch}$)出现在输入字符串中,因此 $\sum p_i = 1$。根据信息论的基本定理,由这些字符组成的相互独立的随机序列(一种保守但并不总是现实的假设)平均地要求每个字符至少是 H 位,

$$H = -\sum p_i \log_2 p_i \tag{20.4.1}$$

其中 H 是概率分布的熵。另外,一定存在可任意接近这个边界值的编码方式。对于所有字符等概率 $p_i = 1/N_{ch}$ 的情况,很容易得出 $H = \log_2 N_{ch}$,它是无压缩的情况。p_i 的任意其他集合给出较小的熵,它允许进行有效的压缩。

注意,如果用长度为 $L_i = -\log_2 p_i$ 位的码对字符 i 进行编码,这样(20.4.1)式的边界就可以达到。这样(20.4.1)式将为平均的 $\sum p_i L_i$。这种编码方式的问题是,一般 $-\log_2 p_i$ 不是一个整数。怎样能把字母"Q"用 5.32 位编码呢?霍夫曼编码进行了一种尝试,即实际上用 1/2 的整数幂来近似所有的概率 p_i,因此所有 L_i 都为整数。如果所有 p_i 实际上都是 1/2 的整数次幂,则霍夫曼码的确达到熵边界 H。

霍夫曼码的构造最好用示例来说明。想像一种语言,Vowellish,它有 $N_{ch} = 5$ 个字符的字母 A,E,I,O 和 U,分别以 0.12,0.42,0.09,0.30 和 0.07 的概率出现。于是为 Vowellish 构造的霍夫曼码按表 20.4.1 中的步骤完成。

表 20.4.1　霍夫曼码的构造步骤表

节点	阶段：	1	2	3	4	5
1	A：	0.12	0.12 ▮			
2	E：	0.42	0.42	0.42	0.42 ▮	
3	I：	0.09 ▮				
4	O：	0.30	0.30	0.30 ▮		
5	U：	0.07 ▮				
6		UI：	0.16 ▮			
7			AUI：	0.28 ▮		
8				AUIO：	0.58 ▮	
9					EAUIO：	1.00

它是通过 N_{ch} 个阶段顺序进行的，由表中的列表示。第一阶段从 N_{ch} 个节点开始，每一节点表示字母表中的一个字母，分别包含它们的相应概率。在每一阶段，将两个最小的概率加合起来构成一个新的节点。然后将这新节点从实际节点列中除去(表中小黑方块表示该节点被弃的阶段)。所有活动节点(包括新合成的)都列入下一阶段(列)。在表 20.4.1 中，赋给新节点的名称(比如 AUI)是不连贯的，在所举例子中(阶段 1 之后)，碰巧每阶段的两个最小节点总有一个是原始节点，另一个是合成节点。一般情况是：两个最小概率可能都是原始节点，或都是合成的，或各占一个。在最后阶段，所有节点都要被集中到一个整体概率为 1 的总合成中。

现在，为了得到这些码字，把表 20.4.1 中的数据重新以树形表示(见图 20.4.1)。如图所示，树中的每一节点对应于表中的一个节点(行)，其左边标着节点的整数序号，右边标着它的概率值。所谓终端节点由圆圈表示，它们是单个字母表的字符。在树枝上标记 0 和 1。字符的码字就是从顶端往下指向它的 0 和 1 的序列。例如 E 的码字简单地为 0，而 U 的码字是 1010。

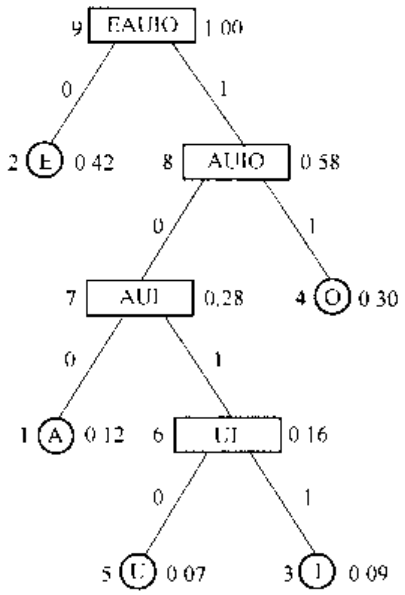


图 20.4.1　虚构语言 Vowellish 的霍夫曼码，以树形表示。沿着树从顶向下移动，对字母(A，E，I，O和U)进行译码的编码。码字是所经过树枝上所对应的0或1的序列，每个端点右边的值是它的概率，左边的值是表20.4.1中的端点编号

现在，任意一个 0，1 串都能译成对应的字母序列。例如考虑串 1011111010。从树的顶端开始下降经过 1011 到达 I——第一个字符。由于已到达了一个终端节点，所以重置于树的顶端，下一次经过 11 下降到 O。最后，1010 给出 U。因此串译成为 IOU。

这些思想体现在下面的程序中，第一个程序 hufmak 的输入是，nchin $\equiv N_{ch}$ 个字母表的字符发生频率的整

数向量,即与 p_i 成正比的整数集。hufmak 和它所调用的 hufapp,实行了表 20.4.1 的构造和图 20.4.1 中树的构造。为了提高效率,这个程序使用堆结构(见 8.3 节)。关于详细描述,参见 Sedgewick[7]。

```
#include "nr.h"

void NR::hufmak(Vec_I_ULNG &nfreq, const unsigned long nchin,
    unsigned long &ilong, unsigned long &nlong, huffcode &hcode)
给定 nchin 个字符的发生频率表 nfreq[0...nchin-1],在类对象 hcode 中构造霍夫曼码。返回值 ilong 和 nlong,它们是所
产生最长码符号的字符编号及符号的长度。应该检查 nlong 不大于所用机器的字长。
{
    int ibit,j,node;
    unsigned long k,n,nused;
    static unsigned long setbit[32]={0x1L,0x2L,0x4L,0x8L,0x10L,0x20L,
        0x40L,0x80L,0x100L,0x200L,0x400L,0x800L,0x1000L,0x2000L,
        0x4000L,0x8000L,0x10000L,0x20000L,0x40000L,0x80000L,0x100000L,
        0x200000L,0x400000L,0x800000L,0x1000000L,0x2000000L,0x4000000L,
        0x8000000L,0x10000000L,0x20000000L,0x40000000L,0x80000000L};

    hcode.nch=nchin;                          初始化。
    Vec_ULNG index(2*hcode.nch-1);
    Vec_ULNG nprob(2*hcode.nch-1);
    Vec_INT up(2*hcode.nch-1);                将跟踪堆的矢量。
    for (nused=0,j=0;j<hcode.nch;j++) {
        nprob[j]=nfreq[j];
        hcode.icod[j]=hcode.ncod[j]=0;
        if (nfreq[j] != 0) index[nused++]=j;
    }
    for (j=nused-1;j>=0;j--)                  在 index 中将 nprob 排序成堆结构。
        hufapp(index,nprob,nused,j);
    k=hcode.nch;
    while (nused > 1) {                       连接堆的节点,在每一步中重新修改堆。
        node=index[0];
        index[0]=index[(nused--)-1];
        hufapp(index,nprob,nused,0);
         nprob[k]=nprob[index[0]]+nprob[node],
         hcode.left[k]=node;                  存储一个节点的左和右子码。
         hcode.right[k++]=index[0];
         up[index[0]] = -int(k);              指示一个节点是否是左和右子码的父码。
         index[0]=k-1,
         up[node]=k;
         hufapp(index,nprob,nused,0);
     }
     up[(hcode.nodemax=k)-1]=0;
     for (j=0;j<hcode.nch;j++) {              由树构造霍夫曼码。
         if (nprob[j] != 0) {
             for (n=0,ibit=0,node=up[j];node;node=up[node-1],ibit++) {
                 if (node < 0) {
                     n |= setbit[ibit];
                     node = -node;
                 }
             }
             hcode.icod[j]=n;
             hcode.ncod[j]=ibit;
         }
     }
     nlong=0;
     for (j=0;j<hcode.nch;j++) {
         if (hcode.ncod[j] > nlong) {
             nlong=hcode.ncod[j];
             ilong=j;
         }
     }
}


#include "nr.h"

void NR::hufapp(Vec_IO_ULNG &index, Vec_I_ULNG &nprob, const unsigned long n,
    const unsigned long m)
```

用于 hufmak，以在数组 index[0...m-1]中维持堆的结构。

```
{
    unsigned long i=m,j,k;

    k=index[i];
    while (i < (n >> 1)) {
        if ((j = 2*i+1) < n-1
            && nprob[index[j]] > nprob[index[j+1]]) j++;
        if (nprob[k] <= nprob[index[j]]) break;
        index[i]=index[j];
        i=j;
    }
    index[i]=k;
}
```

注意，类 huffcode 定义于 nrutil.h 中。Huffcode 型的变量 hcode 必须在用户的主程序中用以下方式建立：

```
#include "nr.h"
    ...
    const unsigned int MC=512;          hufmak 中 nchun 的最大预期值。
    const unsigned int MQ=2*MC-1;
    huffcode hcode(MQ,MQ,MQ,MQ);
    for (i=0;i<MQ;i++)
        hcode.icod[i]=hcode.ncod[i]=0;
```

一旦码构造完毕，可通过反复调用 hufenc 对字符串进行编码，程序 hufenc 简单地进行码的表查找，并且将其加入到输出消息中。

```
#include <string>
#include "nr.h"
using namespace std;

void NR::hufenc(const unsigned long ich, string &code, unsigned long &nb,
    huffcode &hcode)
利用类对象 hcode 中的码，对单个字符 ich(在 0..nch-1 范围内)进行霍夫曼编码，将结果写入串codep中，并以 nb 位起始
(它的最小有效数值为零)，并且适当地增加 nb。对消息中连续字符进行编码需反复调用本程序，但必须提前调用构造
hcode 的程序 hufmak 进行初始化。
{
    int m,n;
    unsigned long k,nc;
    static unsigned long setbit[32]={0x1L,0x2L,0x4L,0x8L,0x10L,0x20L,
        0x40L,0x80L,0x100L,0x200L,0x400L,0x800L,0x1000L,0x2000L,
        0x4000L,0x8000L,0x10000L,0x20000L,0x40000L,0x80000L,0x100000L,
        0x200000L,0x400000L,0x800000L,0x1000000L,0x2000000L,0x4000000L,
        0x8000000L,0x10000000L,0x20000000L,0x40000000L,0x80000000L};

    k=ich;
    if (k >= hcode.nch)
        nrerror("ich out of range in hufenc.");
    for (n=hcode.ncod[k]-1;n >= 0;n--,++nb) {        在已存储的霍夫曼码中对 ich 进行循环。
        nc=nb >> 3;
        if (code.length() < nc+1)                    增加串的长度。
            code.resize(2*(nc+1));
        m=nb & 7;
        if (m == 0) code[nc]=0;                      设置码中适当的二进位。
        if ((hcode.icod[k] & setbit[n]) != 0) code[nc] |= setbit[m];
    }
}
```

对霍夫曼码的消息进行译码较复杂些，必须使用变化的二进位数从码树的顶部沿树向下来回移动。

```
#include <string>
#include "nr.h"
using namespace std;

void NR::hufdec(unsigned long &ich, string &code, const unsigned long lcode,
    unsigned long &nb, huffcode &hcode)
在串 code 中，以二进位数 nb 起始，使用存储在类对象 hcode 中的霍夫曼码，将码字译成单个字符(返回到处于范围 0..
nch-1 中的 ich)，并且适当地增加 nb。在 code[0..lcode-1]中 code 有有效数据。以 nb=0 起始，反复调用本程序，将返
回压缩消息中的连续的字符。返回值 ich=nch 表示消息结束。在用户的主程序中，类对象 hcode 必须已经定义和分配
存储空间，并且通过调用 hufmak 已被填充。
```

```
{
    unsigned long nc;
    static unsigned char setbit[8]={0x1,0x2,0x4,0x8,0x10,0x20,0x40,0x80};

    int node=hcode.nodemax-1;
    for (;;) {                          设置 node 为译码树的顶端，并循环直至得到有效字符。
        nc=nb >> 3;
        if (nc >= lcode) {              用完输入，并用 ich=nch 指出消息的结尾。
            ich=hcode.nch;
            return;
        }
        node=((code[nc] & setbit[7 & nb++]) != 0 ?
            hcode.right[node] : hcode.left[node]);
            从树左还是树右分叉取决于它的值。
        if (node < hcode.nch) {         若到达端点则得一个完整字符并返回。
            ich=node;
            return;
        }
    }
}
```

为了简便，当全部用完码字节时 hufdec 就中止退出。若已编码的消息不是整数个字节，且如果 N_{ch} 小于 256，则 hufdec 就能从最后码字节的假造结尾中返回译码得来的最后一个或二个假造的字符。若用户掌握了有关发送字符数的其他知识，就可轻松地放弃这些假字符。否则，可以通过提供一个二进位数，而不是一个字节数来固定这些操作，并且相应地修改程序(当 N_{ch} 大于等于 256 时，常规地，hufdec 将在一个假造字符的中间中止而退出，并且将其放弃掉)。

20.4.1 游程编码

对于高度相关的二进制数据流的压缩(例如沿着一条传真扫描线的黑白值)，霍夫曼压缩常常与**游程编码**(run-length encoding)相结合：不再分别传递每一个二进位，而是将输入流转换成一系列整数，表明有多少个连续位具有相同值，然后这些整数再进行霍夫曼压缩。CCITT 第三组根据这种方式，对一套八种标准文件规定了固定的、不可改变的、最佳霍夫曼码的传真标准(参见本节参考文献[8,9])。

参考文献和深入阅读

Gallager, R.G. 1968, *Information Theory and Reliable Communication* (New York: Wiley).

Hamming, R.W. 1980, *Coding and Information Theory* (Englewood Cliffs, NJ: Prentice-Hall).

Storer, J.A. 1988, *Data Compression: Methods and Theory* (Rockville, MD: Computer Science Press).

Nelson, M. 1991, *The Data Compression Book* (Redwood City, CA: M&T Books)

Huffman, D.A. 1952, *Proceedings of the Institute of Radio Engineers*, vol. 40, pp. 1098–1101. [1]

Ziv, J., and Lempel, A. 1978, *IEEE Transactions on Information Theory*, vol. IT-24, pp. 530–536. [2]

Cleary, J.G., and Witten, I.H. 1984, *IEEE Transactions on Communications*, vol. COM-32, pp. 396–402. [3]

Welch, T.A. 1984, *Computer*, vol. 17, no. 6, pp. 8–19. [4]

Bentley, J.L., Sleator, D.D., Tarjan, R.E., and Wei, V.K. 1986, *Communications of the ACM*, vol. 29, pp. 320–330. [5]

Jones, D.W. 1988, *Communications of the ACM*, vol. 31, pp. 996–1007. [6]

Sedgewick, R. 1998, *Algorithms in C*, 3rd ed. (Reading, MA: Addison-Wesley), Chapter 22. [7]

Hunter, R., and Robinson, A.H. 1980, *Proceedings of the IEEE*, vol. 68, pp. 854–867. [8]

Marking, M.P. 1990, *The C Users' Journal*, vol. 8, no. 6, pp. 45–54. [9]

20.5 算术编码

在前一节中，我们看到一种良好的(以熵为界的)编码方案，若 p_i 为字符 i(在范围 $1 \leq i \leq N_{ch}$ 中)的出现概率，则用 $L_i = -\log_2 p_i$ 位来对其编码。霍夫曼编码给出了一种把 L_i 舍入成接近的整数值，并用这一长度

来构造码的方法。现在，讨论**算术编码**(参见本节参考文献[1])，实际上它确实实现了用非整数个二进位来对字符编码。它还能提供一种方便的方法，即不是把结果输出为二进位流，而是输出以任何所期望的基数的形式输出符号流。这种特性在需要时特别有用，例如，把数据从字节(基数为 256)转换成可打印的 ASCII 字符(基数为 94)，或者转换成只包括 A～Z 和 0～9(基数为 36)的字母数字序列。

在算术编码中，一个任意长度的输入消息被表示为在 $0 \leqslant R < 1$ 范围中的实数 R。消息越长，对 R 的精度要求越高。这一点通过例子能得到最好的说明，所以回到上一节虚构的语言 Vowellish 上。回想一下，Vowellish 有一个 5 字符的字母表(A,E,I,O,U)，出现概率分别为 0.12,0.42,0.09,0.30 和 0.07。图 20.5.1 显示了从"IOU"开始的消息是如何编码的：区间[0,1)被分成对应于 5 个字母表字符的段，每一段的长度正比于 p_i。看到第一个消息字符"I"，其 R 的范围缩小为 $0.37 \leqslant R < 0.46$。再把这个区间分成五个子区间，长度也是正比于 p_i。对于第二个消息字符"O"，R 的范围缩小到 $0.3763 \leqslant R < 0.4033$。"U"字符再进一步把 R 的范围缩小到 $0.37630 \leqslant R < 0.37819$。在这个范围内，$R$ 的任何值都可作为"IOU"的码字传送。特别地，二进制小数 .011000001 就在这个范围中，所以"IOU"可用 9 位传送(在这个例子中，霍夫曼码需用 10 位，参见 20.4 节)。

当然，问题是要知道何时停止译码。小数 .011000001 不仅代表"IOU"，也代表"IOU...",其中省略号代表一个后继无限的字符串。为解决这个模棱两可的问题，算术编码通常假定存在特殊的第 $N_{ch}+1$ 个字符——EOM(消息的结尾)，它只在输入的末端出现一次。由于 EOM 出现的概率很小，它在数据线上只分配了非常小的一段。

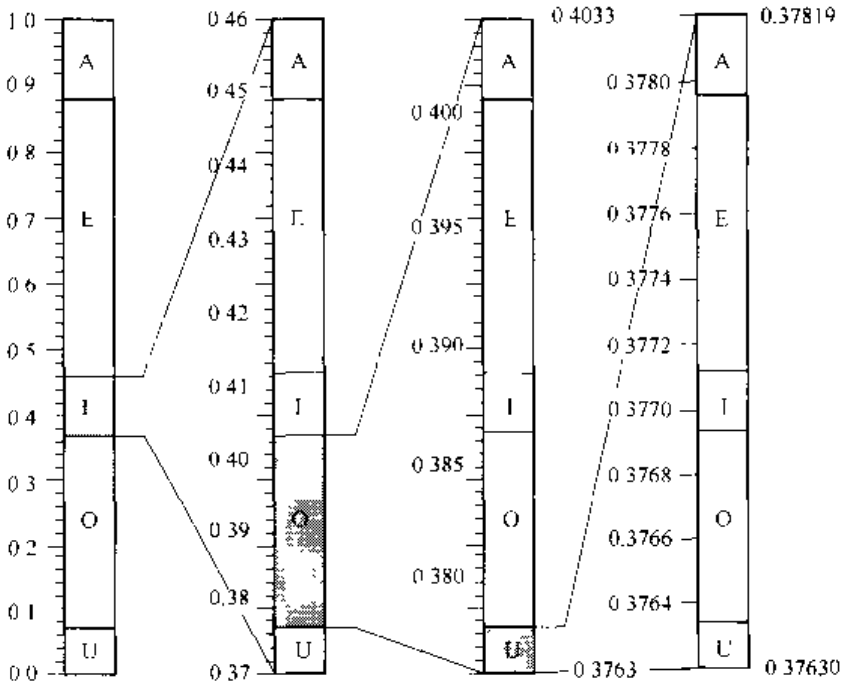


图 20.5.1　虚构的语言 Vowellish 中消息"IOU..."的算术编码　从最初的 0 至 1 区间对逐个字符不断地细分，最后的数值就可用任何期望的基数的形式用小数数字来输出，注意，每个字符分配的子区间正比于它出现的概率

在以上例子中，把 R 作为二进制小数给出。也可以把它以任何其他的基数输出，例如基数 94 或基数 36，只要方便于预期的存储或通信信道即可。

可能会疑惑，对于一个长消息来说，如何处理这种似乎难以置信的对 R 的精度要求。答案是，实际上 R 不会一次完全表示出来。在任何给定阶段中，都有 R 的上边界和下边界，它在输出基数中表示成一个有限的数。当上界数和下界数同一时，可以把它们左移掉，并在最低有效端带进新的位。下面的程序有一个参数 NWK，它是保留工作位数目的参数。它必须足够大，以使退化偶然出现的机会接近于零(一旦出现退化，本程序会给出信号)。由于放弃旧位和带进新位是一致地在编码和解码中进行的，所以整个程序都要保持同步。

程序 arcmak 构造累积频率分布表，用以在每一阶段中分割区间。在主要程序 arcode 中，当一个大小为

jdif的区间以某个n为比例剖分成某个ntot时,则必须计算(n*jdif)/ntot。用整数运算,这个分数中的分子可能会溢出。遗憾的是,像jdif/(ntot/n)这样的表达式与上式并不等价。在以下实现中,采用双精度的浮点运算来进行此项计算。这不仅使效率很差,而且因为在编码和解码过程中会出现同一舍入误差,所以尽管任何机器都是严格按编码方式来解码,但是不同的舍入误差(虽很少见)还是会使不同机器的编码有所不同。为了严格地使用,就需在一个双寄存器中用整数计算(该计算在C++程序中做不到)来替换这种浮点运算。

内设变量minint作为上、下界之间允许的最小高散步长数目,决定着何时加入新的低有效位。minint必须足够大,以能够分辨所有输入的字符。也就是说,对所有 i,必须有 $p_i \times \text{minint} > 1$。无论怎么大,$100N_{ch}$ 或 $1.1/\min p_i$ 一般已足够了。但是,为了安全起见,以下程序让minint尽可能地大,使minint * nradd的积刚好小于溢出。有时,这个结果不是有效的,导致在消息的结尾会输出一些不必要的字符。如果想靠近边缘,可以减小minint。

arcmak中最后一个安全特性是,它对表nfreq中的零值像1一样进行处理。如果不这样,nfreq入口为零的单个字符消息会搅乱剩下的整个消息。如果想要稍微更有效地编码而愿冒险的话,则可以删除MAX(,(unsigned long)1)操作。

```
#include <limits>
#include "nr.h"
using namespace std;

void NR::arcmak(Vec_I_ULNG &nfreq, unsigned long nchh, unsigned long nradd,
    arithcode &acode)
给定nchh个符号的发生频率表nfreq[0..nchh-1],并给定一个期望的输出基数nradd,为在类对象acode中做算术压缩初
始化累积频率表和其他变量。
{
    const unsigned long MAXULNG=numeric_limits<unsigned long>::max();
    unsigned long j;

    unsigned long MC=acode.ncumfq.size()-2;
    if (nchh > MC) nrerror("input radix may not exceed MC in arcmak.");
    if (nradd > 256) nrerror("output radix may not exceed 256 in arcmak.");

    acode.minint=MAXULNG/nradd;
    acode.nch=nchh;
    acode.nrad=nradd;
    acode.ncumfq[0]=0;
    for (j=1;j<=acode.nch;j++)
        acode.ncumfq[j]=acode.ncumfq[j-1]+MAX(nfreq[j-1],(unsigned long) 1);
    acode.ncum=acode.ncumfq[acode.nch+1]=acode.ncumfq[acode.nch]+1;
}
```

类arithcode定义于nrutil.h中。arithcode型变量acode必须在用户的主程序中用以下方式建立:

```
#include "nr.h"
    ..
    const unsigned long MC=512;          在arcmak中nchh的最大预期值。
    const unsigned long NWK=20;          如果它太小,arcode返回错误。
    arithcode acode(NWK,NWK,MC+2);
```

消息中单个字符是用程序arcode进行编码和译码的,程序arcode反过来用到了arcsum的功能。

```
#include "nr.h"

namespace {
    inline unsigned long JTRY(const unsigned long j, const unsigned long k,
        const unsigned long m)
        该函数用于计算无溢出时的(k*j)/m。若用一个对双精度寄存器执行整数相乘的
        汇编语言程序来替换它,本程序的有效性能得到改进。
    {
        return (unsigned long) (DP(j)*DP(k)/DP (m));
    }
}

void NR::arcode(unsigned long &ich, string &code, unsigned long &lcd,
    const int isign, arithcode &acode)
```

将单个字符 ich 压缩(isign = 1)入串 code,或从串 code 中解压缩(isign = −1)出来,以字节 code[lcd] 起始,并且(若需要)增加 lcd,以使指针 lcd 指向即返回到 code 中第一个未用的字节。注意,类对象 acode 既包含码字信息,也包含写入串 code 中的特殊输出的状态信息。无论对于编码或译码,在任意串 code 中开始前,需要用 isign = 0 进行初始化调用。除初始化调用 arcmak 外,要求 arcmak 初始化码本身。若用 ich = nch(在 arcmak 中设置)调用,表示"消息结束"。

```
{
    int j,k;
    unsigned long ihi,ja,jh,jl,m;

    int NWK=acode.ilob.size();
    if (isign == 0) {            初始化上、下限的足够数字。
        acode.jdif=acode.nrad-1;
        for (j=NWK-1;j>=0;j--) {
            acode.iupb[j]=acode.nrad-1;
            acode.ilob[j]=0;
            acode.nc=j;
            if (acode.jdif > acode.minint) return;        初始化结束。
            acode.jdif=(acode.jdif+1)*acode.nrad-1;
        }
        nrerror("NWK too small in arcode.");
    } else {
        if (isign > 0) {         若编码，检验有效的输入字符。
            if (ich > acode.nch) nrerror("bad ich in arcode.");
        } else {                 若译码，用二分法查找字符 ich。
            ja=(unsigned char) code[lcd]-acode.ilob[acode.nc];
            for (j=acode.nc+1;j<NWK;j++) {
                ja *= acode.nrad;
                ja += ((unsigned char) code[lcd+j-acode.nc]-acode.ilob[j]);
            }
            ihi=acode.nch+1;
            ich=0;
            while (ihi-ich > 1) {
                m=(ich+ihi)>>1;
                if (ja >= JTRY(acode.jdif,acode.ncumfq[m],acode.ncum))
                    ich=m;
                else ihi=m;
            }
            if (ich == acode.nch) return;         检查出消息结束。
        }
        以下对编码和译码是公用的。把字符ich转换成新的子区域 [ilob,iupb)。
        jh=JTRY(acode.jdif,acode.ncumfq[ich+1],acode.ncum);
        jl=JTRY(acode.jdif,acode.ncumfq[ich],acode.ncum);
        acode.jdif=jh-jl;
        arcsum(acode.ilob,acode.iupb,jh,NWK,acode.nrad,acode.nc);
        arcsum(acode.ilob,acode.ilob,jl,NWK,acode.nrad,acode.nc);
        有多少起首数字要输出（如果编码）或忽略?
        for (j=acode.nc;j<NWK;j++) {
            if (ich != acode.nch && acode.iupb[j] != acode.ilob[j]) break;
            if (isign > 0) code += (unsigned char)acode.ilob[j];
            lcd++;
        }
        if (j+1 > NWK) return;  超出消息，是否忘记对端点 nch 编码?
        acode.nc=j;
        for(j=0;acode.jdif<acode.minint;j++)      多少数字要移位?
            acode.jdif *= acode.nrad;
        if (j > acode.nc) nrerror("NWK too small in arcode.");
        if (j != 0) {            对它们移位。
            for (k=acode.nc;k<NWK;k++) {
                acode.iupb[k-j]=acode.iupb[k];
                acode.ilob[k-j]=acode.ilob[k];
            }
        }
        acode.nc -= j;
        for (k=NWK-j;k<NWK;k++) acode.iupb[k]=acode.ilob[k]=0;
    }
    return;                      正常返回。
}
```

```
#include "nr.h"

void NR::arcsum(Vec_I_ULNG &iin, Vec_O_ULNG &iout, unsigned long ja,
    const int nwk, const unsigned long nrad, const unsigned long nc)
用于 arcode 把整数 ja 添加到基数 nrad 成倍精度的整数 iin[nc..nwk-1]中。返回结果 iout[nc..nwk-1]。
{
    int karry=0;
    unsigned long j,jtmp;

    for (j=nwk-1;j>nc;j--) {
        jtmp=ja;
        ja /= nrad;
        iout[j]=iin[j]+(jtmp-ja*nrad)+karry;
        if (iout[j] >= nrad) {
            iout[j] -= nrad;
            karry=1;
        } else karry=0;
    }
    iout[nc]=iin[nc]+ja+karry;
}
```

若用户的主要目的是为了改变基数,而不是压缩(例如将任意文件转换成可打印的字符),则当然可以自由设置 nfreq 中的所有分量相等,如设为 1。

参考文献和深入阅读

Bell, T.C., Cleary, J.G., and Witten, I.H. 1990, *Text Compression* (Englewood Cliffs, NJ: Prentice-Hall).

Nelson, M. 1991, *The Data Compression Book* (Redwood City, CA: M&T Books).

Witten, I.H., Neal, R.M., and Cleary, J.G. 1987, *Communications of the ACM*, vol. 30, pp. 520–540. [1]

20.6 任意精度的运算

让我们计算 π 到十进制的数千位。为此,将学习计算机的一些关于多精度运算的问题,并将遇到非常不寻常的关于快速傅里叶变换(FFT)的应用。还将演变出一些可以用来在任何所要求的运算精度的水平上进行其他计算的程序。

首先,需要一个 π 的分析算法。有用的算法是二次收敛的,即每一次迭代使有效数字增加一倍。π 二次收敛算法是基于 AGM 法(**算术几何平均法**),该方法也可应用于椭圆积分(见 6.11 节)的计算,以及椭圆偏微分方程(见 19.5 节)所采用的 ADI 法的高级应用。Borwein 和 Borwein(参见本节参考文献[1])处理了这个课题,但它已超出了本书的讨论范围。它们对于 π 的算法之一是首先设置初始值为

$$
\begin{aligned}
X_0 &= \sqrt{2} \\
\pi_0 &= 2+\sqrt{2} \\
Y_0 &= \sqrt[4]{2}
\end{aligned}
\tag{20.6.1}
$$

然后,当 $i=0,1,\ldots$,重复迭代

$$
\begin{aligned}
X_{i+1} &= \frac{1}{2}\left(\sqrt{X_i}+\frac{1}{\sqrt{X_i}}\right) \\
\pi_{i+1} &= \pi_i\left(\frac{X_{i+1}+1}{Y_i+1}\right) \\
Y_{i+1} &= \frac{Y_i\sqrt{X_{i+1}}+\dfrac{1}{\sqrt{X_{i+1}}}}{Y_i+1}
\end{aligned}
\tag{20.6.2}
$$

π 的值就是极限 π_∞。

下面,讨论怎样解决任意精度算法的问题:在高级语言中,如 C++ 语言,一个自然的选择就是以 256 为

基数进行运算，因此，字符数组可直接表示成数字串，一般在计算结束总希望将结果转换成以 10 为基的数，因为人的思维习惯于这种熟悉的读法“叁点壹肆壹伍玖…”。然而对于计算机计算，可能永远离不开基 256（或者通常可达到的十六进制、八进制或二进制基）。

将采取以“人为”顺序存储数字串的习惯，也就是说，数组中第一位存储的数字是最高有效位，最后一位存储的数字是最低有效位。当然，与此相反的习惯也可以适用。“进位”，即需要将比 255 大的数划分成低位字节和高位进位，这就出现了一个编程的麻烦，在下面程序中采用函数 lobyte 和 hibyte 来解决这个问题。我们通常的习惯是，假设用处于一个数字之后的基数小数点来表示浮点数。当一项操作产生了一个数，此数要求在十进制小数点之前有更多数字，用户就必须将这些数字移位至右端，并且注意这种做法所隐含的 256 的任何附加因子。

根据 Knuth[2]编写一个“最快”算术运算的程序：捷加法（加一个单个字节加到一个串中）、加法、减法、捷乘法（用一单个字节乘一个串）、捷除法、一次补码非操作，以及一对实用的操作，即复制和左移串（在本书程序机器可读的版本中，这些函数都在单一文件 mpops.cpp 内）。

```
#include "nr.h"

在第一个数字后用基点对译成以 256 为基数的字符串进行多精度算术运算。这组程序搜集了简单的运算。

namespace {
    unsigned char lobyte(unsigned short x)
    {return (x & 0xff);}

    unsigned char hibyte(unsigned short x)
    {return ((x >> 8) & 0xff);}
}

void NR::mpadd(Vec_O_UCHR &w, Vec_I_UCHR &u, Vec_I_UCHR &v)
以 256 为基的无符号数 u 和 v 相加，产生无符号结果 w。为实现全部可获得的精度，数组 w 每一个元素都必须比两个较
短的数组 u 和 v 长一些。
{
    int j,n_min,p_min;
    unsigned short ireg=0;

    int n=u.size();
    int m=v.size();
    int p=w.size();
    n_min=MIN(n,m);
    p_min=MIN(n_min,p-1);
    for (j=p_min-1;j>=0;j--) {
        ireg=u[j]+v[j]+hibyte(ireg);
        w[j+1]=lobyte(ireg);
    }
    w[0]=hibyte(ireg);
    if (p > p_min+1)
        for (j=p_min+1;j<p;j++) w[j]=0;
}

void NR::mpsub(int &is, Vec_O_UCHR &w, Vec_I_UCHR &u, Vec_I_UCHR &v)
从以 256 为基数的无符号数 u 减去 v，产生无符号结果 w。若相减结果为负，则 is 返回 -1，反之，则返回 0。为实现全部
可获得的精度，数组 w 每一个元素都必须比两个较短的数组 u 和 v 长一些。
{
    int j,n_min,p_min;
    unsigned short ireg=256;

    int n=u.size();
    int m=v.size();
    int p=w.size();
    n_min=MIN(n,m);
    p_min=MIN(n_min,p);
    for (j=p_min-1;j>=0;j--) {
        ireg=255+u[j]-v[j]+hibyte(ireg);
```

```
        w[j]=lobyte(ireg);
    }
    is=hibyte(ireg)-1;
    if (p > p_min)
        for (j=p_min;j<p;j++) w[j]=0;
}

void NR::mpsad(Vec_O_UCHR &w, Vec_I_UCHR &u, const int iv)
```

捷加法:将整数 iv($0\leq iv\leq 255$)加以 256 为基数的无符号数 u 的最低有效基数位置相加,产生结果 w。为了保证结果在基数的小数点之前不要求两个数字,我们可以首先将操作数 u 右移,以使第一个数字变成 0,并且要单独地注意 256 的倍数。

```
{
    int j;
    unsigned short ireg;

    int n=u.size();
    int p=w.size();
    ireg=256*iv;
    for (j=n-1;j>=0;j--) {
        ireg=u[j]+hibyte(ireg);
        if (j+1 < p) w[j+1]=lobyte(ireg);
    }
    w[0]=hibyte(ireg);
    for (j=n+1;j<p;j++) w[j]=0;
}

void NR::mpsmu(Vec_O_UCHR &w, Vec_I_UCHR &u, const int iv)
```

捷乘法:以 256 为基数的无符号数 u 乘以整数 iv($0\leq iv\leq 255$),产生结果 w。为了保证结果在基数的小数点之前不要求两个数字,我们可以首先将操作数 u 右移,以使第一个数字变成 0,并且要单独地注意 256 的倍数。

```
{
    int j;
    unsigned short ireg=0;

    int n=u.size();
    int p=w.size();
    for (j=n-1;j>=0;j--) {
        ireg=u[j]*iv+hibyte(ireg);
        if (j < p-1) w[j+1]=lobyte(ireg);
    }
    w[0]=hibyte(ireg);
    for (j=n+1;j<p;j++) w[j]=0;
}

void NR::mpsdv(Vec_O_UCHR &w, Vec_I_UCHR &u, const int iv, int &ir)
```

捷除法:以 256 为基数的无符号数 u 被整数 iv 除(而 $0\leq iv\leq 255$),产生商为 w,余数为 ir($0\leq ir\leq 255$)。为了实现全部可获得的精度,数组 w 必须与数组 u 一样长。

```
{
    int i,j,p_min;

    int n=u.size();
    int p=w.size();
    p_min=MIN(n,p);
    ir=0;
    for (j=0;j<p_min;j++) {
        i=256*ir+u[j];
        w[j]=(unsigned char) (i/iv);
        ir=i % iv;
    }
    if (p > p_min)
        for (j=p_min;j<p;j++) w[j]=0;
}

void NR::mpneg(Vec_IO_UCHR &u)
```

对以 256 为基数的无符号数 u 执行一次补码非操作。

```
{
    int j;
    unsigned short ireg=256;

    int n=u.size();
    for (j=n-1;j>=0;j--) {
        ireg=255-u[j]+hibyte(ireg);
        u[j]=lobyte(ireg);
    }
```

```
}

void NR::mpmov(Vec_O_UCHR &u, Vec_I_UCHR &v)
将以 256 为基数的无符号数 v 复制到 u 中，为了实现全部可获得的精度，数组 v 必须与数组 u 一样长。
{
    int j,n_min;

    int n=u.size();
    int m=v.size();
    n_min=MIN(n,m);
    for (j=0;j<n_min;j++) u[j]=v[j];
    if (n > n_min)
        for(j=n_min;j<n-1;j++) u[j]=0;
}

void NR::mplsh(Vec_IO_UCHR &u)
左移以 256 为基数的无符号数 u 的数字。数组中的最后一个元素置为 0
{
    int j;

    int n=u.size();
    for (j=0;j<n-1;j++) u[j]=u[j+1];
    u[n-1]=0;
}
```

如果用传统的手算，两个数字串的全乘法不是快速的运算方法：两个长度均为 N 的数字串相乘，被乘数将依次与乘数的每一个数字位做捷乘，共需 $O(N^2)$ 次操作。然而，我们将看到，对于长度为 N 的数字的所有算术运算，实际上可按 $O(N \times \log N \times \log\log N)$ 次操作内完成。

这种方法是，将乘法实质上视为乘数和被乘数字的卷积（见 13.1 节），然后进行某种进位操作。例如考虑 456×789 的两种计算方法：

```
        4 5 6                          4   5   6
      × 7 8 9                        ×   7   8   9
      -------                        -----------
      4 1 0 4                            36  45  54
    3 6 4 8                          32  40  48
  3 1 9 2                          28  35  42
  -----------                      -----------
  3 5 9 7 8 4                      28  67 118  93  54
                                 -----------------------
                                 3   5   9   7   8   4
```

上图左边显示的是传统的乘法运算，该方法中，对全部被乘数(9,8,7)分别进行捷乘，然后相加得到最后的结果。而图右边显示了不同的运算方法（有时用做心算），此时，所有单个数字进行叉乘（例 $8 \times 6 = 48$），然后按列相加得到没完成的进位结果（这里是 28,67,118,93,54）。最后一步是一个从右到左过程，记下单一的最低有效数字，并且将其他的较高有效数字进位到左边的总数中相加（例如，$93 + 5 = 98$，记录 8，进位 9）。

可立刻看出，在右边的方法中，每一列的和是数字串卷积的组成部分，例如 $118 = 4 \times 9 + 5 \times 8 + 6 \times 7$。在 13.1 节中，我们学习过怎样用快速傅里叶变换（FFT）计算两个向量的卷积：每个向量做 FFT，然后将两个复变换式相乘，得到的结果做逆 FFT。因为这个变换采用了浮点算法，因此需要足够的精度，以使在出现舍入误差时，结果中每个组成部分的准确整数值仍是可辨认的。为了 FFT 的舍入误差，应该允许（连续）增加几倍 $\log_2(\log_2 N)$ 位的二进位。以 256 为基数、长度为 N 字节的数可产生大到 $(256)^2 N$ 阶的卷积分量，所以为了精确存储，需要 $16 + \log_2 N$ 位精度，若数 it 是浮点尾数的二进位数（见 20.1 节），则得条件：

$$16 + \log_2 N + \text{few} \times \log_2 \log_2 N < \text{it} \tag{20.6.3}$$

我们看到，如果 it = 24，则对于任意感兴趣的 N 值，单精度是不合适的；如果 it = 53，则双精度允许 N 大于 10^6，相当于几百万个十进位数。因此在程序 realft（见 12.3 节）和 four1（见 12.2 节）中使用双精度很有必要。

```
#include "nr.h"

void NR::mpmul(Vec_O_UCHR &w, Vec_I_UCHR &u, Vec_I_UCHR &v)
运用快速傅里叶变换计算以 256 为基数的无符号整数 u[0..n-1]与 v[0..m-1]相乘，产生积为 w[0..n+m-1]。
```

```
{
    const DP RX=256.0;
    int j,n_max,nn=1;
    DP cy,t;

    int n=u.size();
    int m=v.size();
    int p=w.size();
    n_max=MAX(m,n);
    while (nn < n_max) nn <<= 1;     为变换找出最小可用的2的幂次。
    nn <<= 1;
    Vec_DP a(0.0,nn),b(0.0,nn);
    for (j=0;j<n;j++) a[j]=u[j];     将u和v移入双精度浮点型数组。
    for (j=0;j<m;j++) b[j]=v[j];
    realft(a,1);                     执行卷积：首先求出二个傅里叶变换。
    realft(b,1);
    b[0] *= a[0];                    复数相乘的结果（实部和虚部）。
    b[1] *= a[1];
    for (j=2;j<nn;j+=2) {
        b[j]=(t=b[j])*a[j]-b[j+1]*a[j+1];
        b[j+1]=t*a[j+1]+b[j+1]*a[j];
    }
    realft(b,-1);                    进行傅氏反变换。
    cy=0.0;                          执行最后完成所有进位的过程。
    for (j=nn-1;j>=0;j--) {
        t=b[j]/(nn >> 1)+cy+0.5;     0.5考虑了舍入误差。
        cy=(unsigned long) (t/RX);
        b[j]=t-cy*RX;
    }
    if (cy >= RX) nrerror("cannot happen in mpmul");
    for (j=0;j<p;j++) w[j]=0;
    w[0]=(unsigned char) cy;         将结果复制输出。
    for (j=1;j<MIN(n+m,p);j++)
        w[j]=(unsigned char) b[j-1];
}
```

若采用这种"快速"过程的乘法，那么除法最好的实现方式就是用除数的倒数乘以被除数。值 V 的倒数用牛顿法的迭代来计算：

$$U_{i+1} = U_i(2 - VU_i) \tag{20.6.4}$$

这导致 U_∞ 二次收敛于 $1/V$，这一点很容易证明(实际上，许多超级计算机和 RISC 机都是用这种迭代法来实现除法的)。现在就可知道上面所提到的运算量 $N \log N \log\log N$ 是在何处产生的了：傅里叶变换中要求 $N \log N$ 运算量，它用牛顿法则的迭代收敛需要另增加一个 $\log\log N$ 的因子。

```
#include "nr.h"

void NR::mpinv(Vec_O_UCHR &u, Vec_I_UCHR &v)
字符串v[0..m-1]是以256为基数的数，基数的小数点在(非零的)v[0]之后；u[0..n-1]设置为它的倒数的最高有效数
字，基数的小数点在u[0]之后。
{
    const int MF=4;
    const DP BI=1.0/256.0;
    int i,j,mm;
    DP fu,fv;

    int n=u.size();
    int m=v.size();
    Vec_UCHR s(n+m),r(2*n+m);
    mm=MIN(MF,m);
    fv=DP(v[mm-1]);                  采用通常的浮点运算以得到一个初始近似值。
    for (j=mm-2;j>=0;j--) {
        fv *= BI;
        fv += v[j];
    }
    fu=1.0/fv;
    for (j=0;j<n;j++) {
        i=int(fu);
        u[j]=(unsigned char) i;
        fu=256.0*(fu-i);
```

```
    }
    for (;;) {                          迭代牛顿法以收敛。
        mpmul(s,u,v);                   在 S 中构造 2-UV
        mplsh(s);
        mpneg(s);
        s[0] += (unsigned char) 2;      将 SU 的乘积，赋于 U。
        mpmul(r,s,u);
        mplsh(r);
        mpmov(u,r);
        for (j=1;j<n-1;j++)             若 S 的小数部分不是零，它就没有收敛到 1。
            if (s[j] != 0) break;
        if (j==n-1) return;
    }
}
```

现在，除法只是一个简单的推论，仅需计算满足一定精度的倒数，从而得到精确的商和余数。

```
#include "nr.h"

void NR::mpdiv(Vec_O_UCHR &q, Vec_O_UCHR &r, Vec_I_UCHR &u, Vec_I_UCHR &v)
以 256 为基数的无符号整数 u[0..n-1]被 v[0..m-1]除(要求 m≤n)，产生商为 q[0..n-m]和余数 r[0..m-1]。
{
    const int MACC=1;
    int i,is,mm;

    int n=u.size();
    int m=v.size();
    int p=r.size();
    int n_min=MIN(m,p);
    if (m > n) nrerror("Divisor longer than dividend in mpdiv");
    mm=m+MACC;
    Vec_UCHR s(mm),rr(mm),ss(mm+1),qq(n-m+1),t(n);
    mpinv(s,v);                         设 S=1/V。
    mpmul(rr,s,u);                      设 Q=SU。
    mpsad(ss,rr,1);
    mplsh(ss);
    mplsh(ss);
    mpmov(qq,ss);
    mpmov(q,qq);
    mpmul(t,qq,v);                      相乘和相减得到余数。
    mplsh(t);
    mpsub(is,t,u,t);
    if (is != 0) nrerror("MACC too small in mpdiv");
    for (i=0;i<n_min;i++) r[i]=t[i+n-m];
    if (p>m)
        for (i=m;i<p;i++) r[i]=0;
}
```

用牛顿法计算平方根与除法类似。若

$$U_{i+1} = \frac{1}{2}U_i(3 - VU_i^2) \tag{20.6.5}$$

则 U_∞ 二次收敛于 $1/\sqrt{V}$，最后乘以 V 得 $\sqrt{V}$。

```
#include <cmath>
#include "nr.h"
using namespace std;

void NR::mpsqrt(Vec_O_UCHR &w, Vec_O_UCHR &u, Vec_I_UCHR &v)
字符串 v[0..m-1]是以 256 为基数的数表示，基数小数点在 v[0]之后；w[0..n-1]设为 v 的平方根(基数小数点在 w[0]之
后)，并且 u[0..n-1]设为它的倒数(基数小数点在 u[0]之前)。在 w 和 u 不需要区别的情况下，它们均可设为平方根。
{
    const int MF=3;
    const DP BI=1.0/256.0;
    int i,ir,j,mm;
    DP fu,fv;

    int n=u.size();
    int m=v.size();
    Vec_UCHR r(2*n),x(n+m),s(2*n+m),t(3*n+m);
```

```
    mm=MIN(m,MF);
    fv=DP(v[mm-1]);                    采用通常的浮点运算得到一个初始近似值。
    for (j=mm-2;j>=0;j--) {
        fv *= BI;
        fv += v[j];
    }
    fu=1.0/sqrt(fv);
    for (j=0;j<n;j++) {
        i=int(fu);
        u[j]=(unsigned char) i;
        fu=256.0*(fu-i);
    }
    for (;;) {                         迭代牛顿法以收敛。
        mpmul(r,u,u);                  构造 S=(3-rU^2)/2。
        mplsh(r);
        mpmul(s,r,v);
        mplsh(s);
        mpneg(s);
        s[0] += (unsigned char) 3;
        mpsdv(s,s,2,ir);
        for (j=1;j<n-1;j++) {          若 S 的小数部分不为零，它还没有收敛到 1。
            if (s[j] != 0) {
                mpmul(t,s,u);          用 SU 替换 U。
                mplsh(t);
                mpmov(u,t);
                break;
            }
        }
        if (j<n-1) continue;
        mpmul(x,u,v);                  从倒数求得平方根并返回。
        mplsh(x);
        mpmov(w,x);
        return;
    }
}
```

前面已经提到基数转换成十进制数只是习惯问题，所以通常可以忽略。将一个小数转换成十进制的最简单方法是反复乘以10，拾取(并减去)结果中的整数部分。然而，因为每次释出十进位数需 $O(N)$ 次运算，所以这个过程共需 $O(N^2)$ 运算量。基数转换也可以采用"分开并分别处理"的策略实现快速运算，其中小数用10的大幂次项来(快速)相乘，足以使所求数字的一半移到基数小数点的左边。现在，再分别处理整数和小数部分，各部分做进一步划分。如果要计算 π 的几十亿位数字，而不是几千位数字，则需实行这个方案。针对当前的目标，采用下面较慢的程序就足够了。

```
#include "nr.h"

void NR::mp2dfr(Vec_IO_UCHR &a, string &s)
将以 256 为基数的小数 a[0..n-1](基数小数点在 a[0]之前)转换成十进位小数，它用 ASCII 字符串 s[0..m-1]表示，其
中 m 是返回值。输入数组 a[0..n-1]将被破坏。注意：为了简便，本程序实现的是慢速算法(∝ N^2)，基数转
换的快速算法的确存在(∝ N ln N)，但较复杂。
{
    const unsigned int IAZ=48,
    int j,m;

    int n=a.size();
    m=int(2.408*n);
    mplsh(a);
    for (j=0;j<m;j++) {
        mpsmu(a,a,10);
        s += a[0]+IAZ;
        mplsh(a);
    }
}
```

最后，得到实现(20.6.1)式和(20.6.2)式的程序：

```
#include <iostream>
#include <iomanip>
#include "nr.h"
using namespace std;

void NR::mppi(const int np)
计算和打印 π 的前 n 位字节的多精度程序。
{
    const unsigned int IAOFF=48,MACC=2;
    int ir,j,n;
    unsigned char mm;
    string s;

    n=np+MACC;
    Vec_UCHR x(n),y(n),sx(n),sxi(n);
    Vec_UCHR z(n),t(n),pi(n);
    Vec_UCHR ss(2*n),tt(2*n);
    t[0]=2;                          设 T = 2。
    for (j=1;j<n;j++) t[j]=0;
    mpsqrt(x,x,t);                   设 X_0 = √2。
    mpadd(pi,t,x);                   设 π_0 = 2 + √2。
    mplsh(pi);
    mpsqrt(sx,sxi,x);                设 Y_0 = 2^(1/4)。
    mpmov(y,sx);
    for (;;) {
        mpadd(z,sx,sxi);             设 X_{i+1} = (X_i^{1/2} + X_i^{-1/2})/2。
        mplsh(z);
        mpsdv(x,z,2,ir);
        mpsqrt(sx,sxi,x);            计算临时值 T = Y_i X_{i+1}^{1/2} + X_{i+1}^{-1/2}。
        mpmul(tt,y,sx);
        mplsh(tt);
        mpadd(tt,tt,sxi);
        mplsh(tt);
        x[0]++;                      X_{i+1} 和 Y_i 增加 1。
        y[0]++;
        mpinv(ss,y);                 设 Y_{i+1} = T/(Y_i + 1)。
        mpmul(y,tt,ss);
        mplsh(y);
        mpmul(tt,x,ss);              计算临时值 T = (X_{i+1} + 1)/(Y_i + 1)。
        mplsh(tt);
        mpmul(ss,pi,tt);             设 π_{i+1} = T π_i。
        mplsh(ss);
        mpmov(pi,ss);
        mm=tt[0]-1;                  若 T = 1 则已收敛。
        for (j=1;j < n-1;j++)
            if (tt[j] != mm) break;
        if (j == n-1) {
            cout << endl << "pi = ";
            s=pi[0]+IAOFF;
            s += '.';
            mp2dfr(pi,s);
            为打印转换成十进制数。注意：这程序是慢速算法（正比于 N^2），只做说明用。快速转换算法
            （正比于 N ln N）存在，但复杂。
            s.erase(2.408*np,s.length());
            cout << setw(64) << left << s << endl;
            return;
        }
    }
}
```

图 20.6.1 给出当 n = 1000 时的计算结果，作为一个练习，读者可能喜欢采用以上程序，针对 Ramanujan 的著名恒等式(参见本节参考文献[3])

$$\frac{1}{\pi} = \frac{\sqrt{8}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!\,(1103+26390n)}{(n!\,396^n)^4} \tag{20.6.6}$$

的前12项来检验图20.6.1中前100位数字。还可以用这程序来证明数$2^{512}+1$不是素数,而是含有因子2 424 833和7 455 602 825 647 884 208 337 395 736 200 454 918 783 366 342 657(它们实际上是素数,剩下的素数因子是7.416×10^{98})[4]。

```
3.14159265358979323846264338327950288419716939937510582097494459230781640 62
8620899862803482534211706798214808651328230664709384460955058223172535940 81
2848111745028410270193852110555964462294895493038196442881097566593344612 84
7564823378678316527120190914564856692346034861045432664821339360726024914 12
7372458700660631558817488152092096282925409171536436789259036001133053054 88
2046652138414695194151160943305727036575959195309218611738193261179310511 85
4807446237996274956735188575272489122793818301194912983367336244065664308 60
2139494639522473719070217986094370277053921717629317675238467481846766940 51
3200056812714526356082778577134275778960917363717872146844090122495343014 65
4958537105079227968925892354201995611212902196086403441815981362977477130 99
6051870721134999999837297804995105973173281609631859502445945534690830264 25
2230825334468503526193118817101000313783875288658753320838142061717766914 73
0359825349042875546873115956286388235378759375195778185778053217122680661 30
0192787661119590921642019893809525720106548586327886593615338182796823030 19
5203530185296899577362259941389124972177528347913151557485724245415069595 08
2953311686172785588907509838175463746493931925506040092770167113909884882 40
1285836160356370766010471018194295559619894676783744944825537977472684710 40
4753464620804668425906949129331367702898915210475216205696602405803815019 35
1125338243003558764024749647326391419927260426992279678235478163600934172 16
4121992458631503028618297455570674983850549458858692699569092721079750930 29
5532116534498720275596023648066549911988183479775356636980742654252786255 18
1841757467289097777279380008164706001614524919217321721477235014144197356 85
4816136115735255213347574184946843852332390739414333454776241686251898356 94
8556209921922218427255025425688767179049460165346680498862723279178608578 43
8382796797668145410095388378636095068006422512520511739298489608412848862 69
4560424196528502221066118630674427862203919494504712371378696095636437191 72
8746776465757396241389086583264599581339047802759009946576407895126946839 83
5259570982582262052248940772671947826848260147699090264013639443745530506 82
0349625245174939965143142980919065925093722169646151570985838741059788595 97
7297549893016175392846813826868386894277415599185592524595395943104997252 46
8084598727364469584865383673622262609912460805124388439045124413654976278 07
9771569143599770012961608944169486855584840635342207222582848864815845602 85
```

图20.6.1 π的前2398位十进位数,它由本节程序计算求得

参考文献和深入阅读

Borwein, J.M., and Borwein, P.B. 1987, *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity* (New York: Wiley). [1]

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §4.3. [2]

Ramanujan, S. 1927, *Collected Papers of Srinivasa Ramanujan*, G.H. Hardy, P.V. Seshu Aiyar, and B.M. Wilson, eds. (Cambridge, U.K.: Cambridge University Press), pp. 23–39. [3]

Kolata, G. 1990, June 20, *The New York Times*. [4]

Kronsjö, L. 1987, *Algorithms: Their Complexity and Efficiency*, 2nd ed. (New York: Wiley).

附录 A　函数声明表

我们在此列出本书中所有程序的函数说明。对包含或者引用本书中任何程序的源文件单独地进行编译时，应当用指令 #include 将本表列入。你必须直接地或者间接地应用辖域分解操作符 NR::以从名称空间 NR 访问一个程序(参见 1.2 节)。

在机器可读的 Numerical Recipes 版本中(例如，CD-ROM 或者从互联网下载)，本附录在文件 nr.h 之中。注意，它们自动地包含文件 nrutil.h 和 nrtypes.h，后者在附录 B 中细述，以使它们更容易加入到你的程序中。

以下是文件 nr.h 的列表：

```
#ifndef _NR_H_
#define _NR_H_
#include <fstream>
#include <complex>
#include "nrutil.h"
#include "nrtypes.h"
using namespace std;

namespace NR {

void addint(Mat_O_DP &uf, Mat_I_DP &uc, Mat_O_DP &res);
void airy(const DP x, DP &ai, DP &bi, DP &aip, DP &bip);
void amebsa(Mat_IO_DP &p, Vec_IO_DP &y, Vec_O_DP &pb, DP &yb, const DP ftol,
    DP funk(Vec_I_DP &), int &iter, const DP temptr);
void amoeba(Mat_IO_DP &p, Vec_IO_DP &y, const DP ftol, DP funk(Vec_I_DP &),
    int &nfunk);
DP amotry(Mat_IO_DP &p, Vec_O_DP &y, Vec_IO_DP &psum, DP funk(Vec_I_DP &),
    const int ihi, const DP fac);
DP amotsa(Mat_IO_DP &p, Vec_O_DP &y, Vec_IO_DP &psum, Vec_O_DP &pb, DP &yb,
    DP funk(Vec_I_DP &), const int ihi, DP &yhi, const DP fac);
void anneal(Vec_I_DP &x, Vec_I_DP &y, Vec_IO_INT &iorder);
DP anorm2(Mat_I_DP &a);
void arcmak(Vec_I_ULNG &nfreq, unsigned long nchh, unsigned long nradd,
    arithcode &acode);
void arcode(unsigned long &ich, string &code, unsigned long &lcd,
    const int isign, arithcode &acode);
void arcsum(Vec_I_ULNG &iin, Vec_O_ULNG &iout, unsigned long ja,
    const int nwk, const unsigned long nrad, const unsigned long nc);
void asolve(Vec_I_DP &b, Vec_O_DP &x, const int itrnsp);
void atimes(Vec_I_DP &x, Vec_O_DP &r, const int itrnsp);
void avevar(Vec_I_DP &data, DP &ave, DP &var);
void balanc(Mat_IO_DP &a);
void banbks(Mat_I_DP &a, const int m1, const int m2, Mat_I_DP &al,
    Vec_I_INT &indx, Vec_IO_DP &b);
void bandec(Mat_IO_DP &a, const int m1, const int m2, Mat_O_DP &al,
    Vec_O_INT &indx, DP &d);
void banmul(Mat_I_DP &a, const int m1, const int m2, Vec_I_DP &x,
    Vec_O_DP &b);
void bcucof(Vec_I_DP &y, Vec_I_DP &y1, Vec_I_DP &y2, Vec_I_DP &y12,
    const DP d1, const DP d2, Mat_O_DP &c);
void bcuint(Vec_I_DP &y, Vec_I_DP &y1, Vec_I_DP &y2, Vec_I_DP &y12,
    const DP x1l, const DP x1u, const DP x2l, const DP x2u,
    const DP x1, const DP x2, DP &ansy, DP &ansy1, DP &ansy2);
void beschb(const DP x, DP &gam1, DP &gam2, DP &gampl, DP &gammi);
DP bessi(const int n, const DP x);
DP bessi0(const DP x);
DP bessi1(const DP x);
void bessik(const DP x, const DP xnu, DP &ri, DP &rk, DP &rip, DP &rkp);
DP bessj(const int n, const DP x);
DP bessj0(const DP x);
```

```
DP bessj1(const DP x);
void bessjy(const DP x, const DP xnu, DP &rj, DP &ry, DP &rjp, DP &ryp);
DP bessk(const int n, const DP x);
DP bessk0(const DP x);
DP bessk1(const DP x);
DP bessy(const int n, const DP x);
DP bessy0(const DP x);
DP bessy1(const DP x);
DP beta(const DP z, const DP w);
DP betacf(const DP a, const DP b, const DP x);
DP betai(const DP a, const DP b, const DP x);
DP bico(const int n, const int k);
void bksub(const int ne, const int nb, const int jf, const int k1,
    const int k2, Mat3D_IO_DP &c);
DP bnldev(const DP pp, const int n, int &idum);
DP brent(const DP ax, const DP bx, const DP cx, DP f(const DP),
    const DP tol, DP &xmin);
void broydn(Vec_IO_DP &x, bool &check, void vecfunc(Vec_I_DP &, Vec_O_DP &));
void bsstep(Vec_IO_DP &y, Vec_IO_DP &dydx, DP &xx, const DP htry,
    const DP eps, Vec_I_DP &yscal, DP &hdid, DP &hnext,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &));
void caldat(const int julian, int &mm, int &id, int &iyyy);
void chder(const DP a, const DP b, Vec_I_DP &c, Vec_O_DP &cder, const int n);
DP chebev(const DP a, const DP b, Vec_I_DP &c, const int m, const DP x);
void chebft(const DP a, const DP b, Vec_O_DP &c, DP func(const DP));
void chebpc(Vec_I_DP &c, Vec_O_DP &d);
void chint(const DP a, const DP b, Vec_I_DP &c, Vec_O_DP &cint, const int n);
DP chixy(const DP bang);
void choldc(Mat_IO_DP &a, Vec_O_DP &p);
void cholsl(Mat_I_DP &a, Vec_I_DP &p, Vec_I_DP &b, Vec_O_DP &x);
void chsone(Vec_I_DP &bins, Vec_I_DP &ebins, const int knstrn, DP &df,
    DP &chsq, DP &prob);
void chstwo(Vec_I_DP &bins1, Vec_I_DP &bins2, const int knstrn, DP &df,
    DP &chsq, DP &prob);
void cisi(const DP x, complex<DP> &cs);
void cntab1(Mat_I_INT &nn, DP &chisq, DP &df, DP &prob, DP &cramrv, DP &ccc);
void cntab2(Mat_I_INT &nn, DP &h, DP &hx, DP &hy, DP &hygx, DP &hxgy,
    DP &uygx, DP &uxgy, DP &uxy);
void convlv(Vec_I_DP &data, Vec_I_DP &respns, const int isign,
    Vec_O_DP &ans);
void copy(Mat_O_DP &aout, Mat_I_DP &ain);
void correl(Vec_I_DP &data1, Vec_I_DP &data2, Vec_O_DP &ans);
void cosft1(Vec_IO_DP &y);
void cosft2(Vec_IO_DP &y, const int isign);
void covsrt(Mat_IO_DP &covar, Vec_I_BOOL &ia, const int mfit);
void crank(Vec_IO_DP &w, DP &s);
void cyclic(Vec_I_DP &a, Vec_I_DP &b, Vec_I_DP &c, const DP alpha,
    const DP beta, Vec_I_DP &r, Vec_O_DP &x);
void daub4(Vec_IO_DP &a, const int n, const int isign);
DP dawson(const DP x);
DP dbrent(const DP ax, const DP bx, const DP cx, DP f(const DP),
    DP df(const DP), const DP tol, DP &xmin);
void ddpoly(Vec_I_DP &c, const DP x, Vec_O_DP &pd);
bool decchk(string str, char &ch);
void derivs_s(const DP x, Vec_I_DP &y, Vec_O_DP &dydx);
DP df1dim(const DP x);
void dfpmin(Vec_IO_DP &p, const DP gtol, int &iter, DP &fret,
    DP func(Vec_I_DP &), void dfunc(Vec_I_DP &, Vec_O_DP &));
DP dfridr(DP func(const DP), const DP x, const DP h, DP &err);
void dftcor(const DP w, const DP delta, const DP a, const DP b,
    Vec_I_DP &endpts, DP &corre, DP &corim, DP &corfac);
void dftint(DP func(const DP), const DP a, const DP b, const DP w,
    DP &cosint, DP &sinint);
void difeq(const int k, const int k1, const int k2, const int jsf,
    const int is1, const int isf, Vec_I_INT &indexv, Mat_O_DP &s,
    Mat_I_DP &y);
void dlinmin(Vec_IO_DP &p, Vec_IO_DP &xi, DP &fret, DP func(Vec_I_DP &),
    void dfunc(Vec_I_DP &, Vec_O_DP &));
void eclass(Vec_O_INT &nf, Vec_I_INT &lista, Vec_I_INT &listb);
void eclazz(Vec_O_INT &nf, bool equiv(const int, const int));
```

```
DP ei(const DP x);
void eigsrt(Vec_IO_DP &d, Mat_IO_DP &v);
DP elle(const DP phi, const DP ak);
DP ellf(const DP phi, const DP ak);
DP ellpi(const DP phi, const DP en, const DP ak);
void elmhes(Mat_IO_DP &a);
DP erfcc(const DP x);
DP erff(const DP x);
DP erffc(const DP x);
void eulsum(DP &sum, const DP term, const int jterm, Vec_IO_DP &wksp);
DP evlmem(const DP fdt, Vec_I_DP &d, const DP xms);
DP expdev(int &idum);
DP expint(const int n, const DP x);
DP fidim(const DP x);
DP factln(const int n);
DP factrl(const int n);
void fasper(Vec_I_DP &x, Vec_I_DP &y, const DP ofac, const DP hifac,
    Vec_O_DP &wk1, Vec_O_DP &wk2, int &nout, int &jmax, DP &prob);
void fdjac(Vec_IO_DP &x, Vec_I_DP &fvec, Mat_O_DP &df,
    void vecfunc(Vec_I_DP &, Vec_O_DP &));
void fgauss(const DP x, Vec_I_DP &a, DP &y, Vec_O_DP &dyda);
void fit(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sig, const bool mwt, DP &a,
    DP &b, DP &siga, DP &sigb, DP &chi2, DP &q);
void fitexy(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sigx, Vec_I_DP &sigy,
    DP &a, DP &b, DP &siga, DP &sigb, DP &chi2, DP &q);
void fixrts(Vec_IO_DP &d);
void flegq(const DP x, Vec_O_DP &pl);
void flmoon(const int n, const int nph, int &jd, DP &frac);
DP fmin(Vec_I_DP &x);
void fourl(Vec_IO_DP &data, const int isign);
void fourew(Vec_FSTREAM_p &file, int &na, int &nb, int &nc, int &nd);
void fourfs(Vec_FSTREAM_p &file, Vec_I_INT &nn, const int isign);
void fourn(Vec_IO_DP &data, Vec_I_INT &nn, const int isign);
void fpoly(const DP x, Vec_O_DP &p);
void fred2(const DP a, const DP b, Vec_O_DP &t, Vec_O_DP &f, Vec_O_DP &w,
    DP g(const DP), DP ak(const DP, const DP));
DP fredin(const DP x, const DP a, const DP b, Vec_I_DP &t, Vec_I_DP &f,
    Vec_I_DP &w, DP g(const DP), DP ak(const DP, const DP));
void frenel(const DP x, complex<DP> &cs);
void frprmn(Vec_IO_DP &p, const DP ftol, int &iter, DP &fret,
    DP func(Vec_I_DP &), void dfunc(Vec_I_DP &, Vec_O_DP &));
void ftest(Vec_I_DP &data1, Vec_I_DP &data2, DP &f, DP &prob);
DP gamdev(const int ia, int &idum);
DP gammln(const DP xx);
DP gammp(const DP a, const DP x);
DP gammq(const DP a, const DP x);
DP gasdev(int &idum);
void gaucof(Vec_IO_DP &a, Vec_IO_DP &b, const DP amu0, Vec_O_DP &x,
    Vec_O_DP &w);
void gauher(Vec_O_DP &x, Vec_O_DP &w);
void gaujac(Vec_O_DP &x, Vec_O_DP &w, const DP alf, const DP bet);
void gaulag(Vec_O_DP &x, Vec_O_DP &w, const DP alf);
void gauleg(const DP x1, const DP x2, Vec_O_DP &x, Vec_O_DP &w);
void gaussj(Mat_IO_DP &a, Mat_IO_DP &b);
void gcf(DP &gammcf, const DP a, const DP x, DP &gln);
DP golden(const DP ax, const DP bx, const DP cx, DP f(const DP),
    const DP tol, DP &xmin);
void gser(DP &gamser, const DP a, const DP x, DP &gln);
void hpsel(Vec_I_DP &arr, Vec_O_DP &heap);
void hpsort(Vec_IO_DP &ra);
void hqr(Mat_IO_DP &a, Vec_O_CPLX_DP &wri);
void hufapp(Vec_IO_ULNG &index, Vec_I_ULNG &nprob, const unsigned long n,
    const unsigned long m);
void hufdec(unsigned long &ich, string &code, const unsigned long lcode,
    unsigned long &nb, huffcode &hcode);
void hufenc(const unsigned long ich, string &code, unsigned long &nb,
    huffcode &hcode);
void hufmak(Vec_I_ULNG &nfreq, const unsigned long nchin,
    unsigned long &ilong, unsigned long &nlong, huffcode &hcode);
void hunt(Vec_I_DP &xx, const DP x, int &jlo);
```

```
void hypdrv(const DP s, Vec_I_DP &yy, Vec_O_DP &dyyds);
complex<DP> hypgeo(const complex<DP> &a, const complex<DP> &b,
    const complex<DP> &c, const complex<DP> &z);
void hypser(const complex<DP> &a, const complex<DP> &b,
    const complex<DP> &c, const complex<DP> &z,
    complex<DP> &series, complex<DP> &deriv);
unsigned short icrc(const unsigned short crc, const string &bufptr,
    const short jinit, const int jrev);
unsigned short icrc1(const unsigned short crc, const unsigned char onech);
unsigned long igray(const unsigned long n, const int is);
void indexx(Vec_I_DP &arr, Vec_O_INT &indx);
void indexx(Vec_I_INT &arr, Vec_O_INT &indx);
void interp(Mat_O_DP &uf, Mat_I_DP &uc);
int irbit1(unsigned long &iseed);
int irbit2(unsigned long &iseed);
void jacobi(Mat_IO_DP &a, Vec_O_DP &d, Mat_O_DP &v, int &nrot);
void jacobn_s(const DP x, Vec_I_DP &y, Vec_O_DP &dfdx, Mat_O_DP &dfdy);
int julday(const int mm, const int id, const int iyyy);
void kendl1(Vec_I_DP &data1, Vec_I_DP &data2, DP &tau, DP &z, DP &prob);
void kendl2(Mat_I_DP &tab, DP &tau, DP &z, DP &prob);
void kermom(Vec_O_DP &w, const DP y);
void ks2d1s(Vec_I_DP &x1, Vec_I_DP &y1, void quadvl(const DP, const DP,
    DP &, DP &, DP &, DP &), DP &d1, DP &prob);
void ks2d2s(Vec_I_DP &x1, Vec_I_DP &y1, Vec_I_DP &x2, Vec_I_DP &y2, DP &d,
    DP &prob);
void ksone(Vec_IO_DP &data, DP func(const DP), DP &d, DP &prob);
void kstwo(Vec_IO_DP &data1, Vec_IO_DP &data2, DP &d, DP &prob);
void laguer(Vec_I_CPLX_DP &a, complex<DP> &x, int &its);
void lfit(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sig, Vec_IO_DP &a,
    Vec_I_BOOL &ia, Mat_O_DP &covar, DP &chisq,
    void funcs(const DP, Vec_O_DP &));
void linbcg(Vec_I_DP &b, Vec_IO_DP &x, const int itol, const DP tol,
    const int itmax, int &iter, DP &err);
void linmin(Vec_IO_DP &p, Vec_IO_DP &xi, DP &fret, DP func(Vec_I_DP &));
void lnsrch(Vec_I_DP &xold, const DP fold, Vec_I_DP &g, Vec_IO_DP &p,
    Vec_O_DP &x, DP &f, const DP stpmax, bool &check, DP func(Vec_I_DP &));
void locate(Vec_I_DP &xx, const DP x, int &j);
void lop(Mat_O_DP &out, Mat_I_DP &u);
void lubksb(Mat_I_DP &a, Vec_I_INT &indx, Vec_IO_DP &b);
void ludcmp(Mat_IO_DP &a, Vec_O_INT &indx, DP &d);
void machar(int &ibeta, int &it, int &irnd, int &ngrd, int &machep,
    int &negep, int &iexp, int &minexp, int &maxexp, DP &eps, DP &epsneg,
    DP &xmin, DP &xmax);
void matadd(Mat_I_DP &a, Mat_I_DP &b, Mat_O_DP &c);
void matsub(Mat_I_DP &a, Mat_I_DP &b, Mat_O_DP &c);
void medfit(Vec_I_DP &x, Vec_I_DP &y, DP &a, DP &b, DP &abdev);
void memcof(Vec_I_DP &data, DP &xms, Vec_O_DP &d);
bool metrop(const DP de, const DP t);
void mgfas(Mat_IO_DP &u, const int maxcyc);
void mglin(Mat_IO_DP &u, const int ncycle);
DP midexp(DP funk(const DP), const DP aa, const DP bb, const int n);
DP midinf(DP funk(const DP), const DP aa, const DP bb, const int n);
DP midpnt(DP func(const DP), const DP a, const DP b, const int n);
DP midsql(DP funk(const DP), const DP aa, const DP bb, const int n);
DP midsqu(DP funk(const DP), const DP aa, const DP bb, const int n);
void miser(DP func(Vec_I_DP &), Vec_I_DP &regn, const int npts,
    const DP dith, DP &ave, DP &var);
void mmid(Vec_I_DP &y, Vec_I_DP &dydx, const DP xs, const DP htot,
    const int nstep, Vec_O_DP &yout,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &));
void mnbrak(DP &ax, DP &bx, DP &cx, DP &fa, DP &fb, DP &fc,
    DP func(const DP));
void mnewt(const int ntrial, Vec_IO_DP &x, const DP tolx, const DP tolf);
void moment(Vec_I_DP &data, DP &ave, DP &adev, DP &sdev, DP &var, DP &skew,
    DP &curt);
void mp2dfr(Vec_IO_UCHR &a, string &s);
void mpadd(Vec_O_UCHR &w, Vec_I_UCHR &u, Vec_I_UCHR &v);
void mpdiv(Vec_O_UCHR &q, Vec_O_UCHR &r, Vec_I_UCHR &u, Vec_I_UCHR &v);
void mpinv(Vec_O_UCHR &u, Vec_I_UCHR &v);
void mplsh(Vec_IO_UCHR &u);
```

```
void mpmov(Vec_O_UCHR &u, Vec_I_UCHR &v);
void mpmul(Vec_O_UCHR &w, Vec_I_UCHR &u, Vec_I_UCHR &v);
void mpneg(Vec_IO_UCHR &u);
void mppi(const int np);
void mprove(Mat_I_DP &a, Mat_I_DP &alud, Vec_I_INT &indx, Vec_I_DP &b,
    Vec_IO_DP &x);
void mpsad(Vec_O_UCHR &w, Vec_I_UCHR &u, const int iv);
void mpsdv(Vec_O_UCHR &w, Vec_I_UCHR &u, const int iv, int &ir);
void mpsmu(Vec_O_UCHR &w, Vec_I_UCHR &u, const int iv);
void mpsqrt(Vec_O_UCHR &w, Vec_O_UCHR &u, Vec_I_UCHR &v);
void mpsub(int &is, Vec_O_UCHR &w, Vec_I_UCHR &u, Vec_I_UCHR &v);
void mrqcof(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sig, Vec_I_DP &a,
    Vec_I_BOOL &ia, Mat_O_DP &alpha, Vec_O_DP &beta, DP &chisq,
    void funcs(const DP, Vec_I_DP &,DP &, Vec_O_DP &));
void mrqmin(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sig, Vec_IO_DP &a,
    Vec_I_BOOL &ia, Mat_O_DP &covar, Mat_O_DP &alpha, DP &chisq,
    void funcs(const DP, Vec_I_DP &, DP &, Vec_O_DP &), DP &alamda);
void newt(Vec_IO_DP &x, bool &check, void vecfunc(Vec_I_DP &, Vec_O_DP &));
void odeint(Vec_IO_DP &ystart, const DP x1, const DP x2, const DP eps,
    const DP h1, const DP hmin, int &nok, int &nbad,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &),
    void rkqs(Vec_IO_DP &, Vec_IO_DP &, DP &, const DP, const DP,
    Vec_I_DP &, DP &, DP &, void (*)(const DP, Vec_I_DP &, Vec_O_DP &)));
void orthog(Vec_I_DP &anu, Vec_I_DP &alpha, Vec_I_DP &beta, Vec_O_DP &a,
    Vec_O_DP &b);
void pade(Vec_IO_DP &cof, DP &resid);
void pccheb(Vec_I_DP &d, Vec_O_DP &c);
void pcshft(const DP a, const DP b, Vec_IO_DP &d);
void pearsn(Vec_I_DP &x, Vec_I_DP &y, DP &r, DP &prob, DP &z);
void period(Vec_I_DP &x, Vec_I_DP &y, const DP ofac, const DP hifac,
    Vec_O_DP &px, Vec_O_DP &py, int &nout, int &jmax, DP &prob);
void piksr2(Vec_IO_DP &arr, Vec_IO_DP &brr);
void piksrt(Vec_IO_DP &arr);
void pinvs(const int ie1, const int ie2, const int je1, const int jsf,
    const int jc1, const int k, Mat3D_O_DP &c, Mat_IO_DP &s);
DP plgndr(const int l, const int m, const DP x);
DP poidev(const DP xm, int &idum);
void polcoe(Vec_I_DP &x, Vec_I_DP &y, Vec_O_DP &cof);
void polcof(Vec_I_DP &xa, Vec_I_DP &ya, Vec_O_DP &cof);
void poldiv(Vec_I_DP &u, Vec_I_DP &v, Vec_O_DP &q, Vec_O_DP &r);
void polin2(Vec_I_DP &x1a, Vec_I_DP &x2a, Mat_I_DP &ya, const DP x1,
    const DP x2, DP &y, DP &dy);
void polint(Vec_I_DP &xa, Vec_I_DP &ya, const DP x, DP &y, DP &dy);
void powell(Vec_IO_DP &p, Mat_IO_DP &xi, const DP ftol, int &iter,
    DP &fret, DP func(Vec_I_DP &));
void predic(Vec_I_DP &data, Vec_I_DP &d, Vec_O_DP &future);
DP probks(const DP alam);
void psdes(unsigned long &lword, unsigned long &irword);
void pwt(Vec_IO_DP &a, const int n, const int isign);
void pwtset(const int n);
DP pythag(const DP a, const DP b);
void pzextr(const int iest, const DP xest, Vec_I_DP &yest, Vec_O_DP &yz,
    Vec_O_DP &dy);
DP qgaus(DP func(const DP), const DP a, const DP b);
void qrdcmp(Mat_IO_DP &a, Vec_O_DP &c, Vec_O_DP &d, bool &sing);
DP qromb(DP func(const DP), DP a, DP b);
DP qromo(DP func(const DP), const DP a, const DP b,
    DP choose(DP (*)(const DP), const DP, const DP, const int));
void qroot(Vec_I_DP &p, DP &b, DP &c, const DP eps);
void qrsolv(Mat_I_DP &a, Vec_I_DP &c, Vec_I_DP &d, Vec_IO_DP &b);
void qrupdt(Mat_IO_DP &r, Mat_IO_DP &qt, Vec_IO_DP &u, Vec_I_DP &v);
DP qsimp(DP func(const DP), const DP a, const DP b);
DP qtrap(DP func(const DP), const DP a, const DP b);
DP quad3d(DP func(const DP, const DP, const DP), const DP x1, const DP x2);
void quadct(const DP x, const DP y, Vec_I_DP &xx, Vec_I_DP &yy, DP &fa,
    DP &fb, DP &fc, DP &fd);
void quadmx(Mat_O_DP &a);
void quadvl(const DP x, const DP y, DP &fa, DP &fb, DP &fc, DP &fd);
DP ran0(int &idum);
DP ran1(int &idum);
```

```
DP ran2(int &idum);
DP ran3(int &idum);
DP ran4(int &idum);
void rank(Vec_I_INT &indx, Vec_O_INT &irank);
void ranpt(Vec_O_DP &pt, Vec_I_DP &regn);
void ratint(Vec_I_DP &xa, Vec_I_DP &ya, const DP x, DP &y, DP &dy);
void ratlsq(DP fn(const DP), const DP a, const DP b, const int mm,
    const int kk, Vec_O_DP &cof, DP &dev);
DP ratval(const DP x, Vec_I_DP &cof, const int mm, const int kk);
DP rc(const DP x, const DP y);
DP rd(const DP x, const DP y, const DP z);
void realft(Vec_IO_DP &data, const int isign);
void rebin(const DP rc, const int nd, Vec_I_DP &r, Vec_O_DP &xin,
    Mat_IO_DP &xi, const int j);
void red(const int iz1, const int iz2, const int jz1, const int jz2,
    const int jm1, const int jm2, const int jmf, const int ic1,
    const int jc1, const int jcf, const int kc, Mat3D_I_DP &c,
    Mat_IO_DP &s);
void relax(Mat_IO_DP &u, Mat_I_DP &rhs);
void relax2(Mat_IO_DP &u, Mat_I_DP &rhs);
void resid(Mat_O_DP &res, Mat_I_DP &u, Mat_I_DP &rhs);
DP revcst(Vec_I_DP &x, Vec_I_DP &y, Vec_I_INT &iorder, Vec_IO_INT &n);
void reverse(Vec_IO_INT &iorder, Vec_I_INT &n);
DP rf(const DP x, const DP y, const DP z);
DP rj(const DP x, const DP y, const DP z, const DP p);
void rk4(Vec_I_DP &y, Vec_I_DP &dydx, const DP x, const DP h,
    Vec_O_DP &yout, void derivs(const DP, Vec_I_DP &, Vec_O_DP &));
void rkck(Vec_I_DP &y, Vec_I_DP &dydx, const DP x,
    const DP h, Vec_O_DP &yout, Vec_O_DP &yerr,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &));
void rkdumb(Vec_I_DP &vstart, const DP x1, const DP x2,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &));
void rkqs(Vec_IO_DP &y, Vec_IO_DP &dydx, DP &x, const DP htry,
    const DP eps, Vec_I_DP &yscal, DP &hdid, DP &hnext,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &));
void rlft3(Mat3D_IO_DP &data, Mat_IO_DP &speq, const int isign);
DP rofunc(const DP b);
void rotate(Mat_IO_DP &r, Mat_IO_DP &qt, const int i, const DP a,
    const DP b);
void rsolv(Mat_I_DP &a, Vec_I_DP &d, Vec_IO_DP &b);
void rstrct(Mat_O_DP &uc, Mat_I_DP &uf);
DP rtbis(DP func(const DP), const DP x1, const DP x2, const DP xacc);
DP rtflsp(DP func(const DP), const DP x1, const DP x2, const DP xacc);
DP rtnewt(void funcd(const DP, DP &, DP &), const DP x1, const DP x2,
    const DP xacc);
DP rtsafe(void funcd(const DP, DP &, DP &), const DP x1, const DP x2,
    const DP xacc);
DP rtsec(DP func(const DP), const DP x1, const DP x2, const DP xacc);
void rzextr(const int iest, const DP xest, Vec_I_DP &yest, Vec_O_DP &yz,
    Vec_O_DP &dy);
void savgol(Vec_O_DP &c, const int np, const int nl, const int nr,
    const int ld, const int m);
void scrsho(DP fx(const DP));
DP select(const int k, Vec_IO_DP &arr);
DP selip(const int k, Vec_I_DP &arr);
void shell(const int n, Vec_IO_DP &a);
void shoot(Vec_I_DP &v, Vec_O_DP &f);
void shootf(Vec_I_DP &v, Vec_O_DP &f);
void simp1(Mat_I_DP &a, const int mm, Vec_I_INT &ll, const int nll,
    const int iabf, int &kp, DP &bmax);
void simp2(Mat_I_DP &a, const int m, const int n, int &ip, const int kp);
void simp3(Mat_IO_DP &a, const int i1, const int k1, const int ip,
    const int kp);
void simplx(Mat_IO_DP &a, const int m1, const int m2, const int m3,
    int &icase, Vec_O_INT &izrov, Vec_O_INT &iposv);
void simpr(Vec_I_DP &y, Vec_I_DP &dydx, Vec_I_DP &dfdx, Mat_I_DP &dfdy,
    const DP xs, const DP htot, const int nstep, Vec_O_DP &yout,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &));
void sinft(Vec_IO_DP &y);
void slvsm2(Mat_O_DP &u, Mat_I_DP &rhs);
```

```
void slvsml(Mat_O_DP &u, Mat_I_DP &rhs);
void sncndn(const DP uu, const DP emmc, DP &sn, DP &cn, DP &dn);
DP snrm(Vec_I_DP &sx, const int itol);
void sobseq(const int n, Vec_O_DP &x);
void solvde(const int itmax, const DP conv, const DP slowc,
    Vec_I_DP &scalv, Vec_I_INT &indexv, const int nb, Mat_IO_DP &y);
void sor(Mat_I_DP &a, Mat_I_DP &b, Mat_I_DP &c, Mat_I_DP &d, Mat_I_DP &e,
    Mat_I_DP &f, Mat_IO_DP &u, const DP rjac);
void sort(Vec_IO_DP &arr);
void sort2(Vec_IO_DP &arr, Vec_IO_DP &brr);
void sort3(Vec_IO_DP &ra, Vec_IO_DP &rb, Vec_IO_DP &rc);
void spctrm(ifstream &fp, Vec_O_DP &p, const int k, const bool ovrlap);
void spear(Vec_I_DP &data1, Vec_I_DP &data2, DP &d, DP &zd, DP &probd,
    DP &rs, DP &probrs);
void sphbes(const int n, const DP x, DP &sj, DP &sy, DP &sjp, DP &syp);
void splie2(Vec_I_DP &x1a, Vec_I_DP &x2a, Mat_I_DP &ya, Mat_O_DP &y2a);
void splin2(Vec_I_DP &x1a, Vec_I_DP &x2a, Mat_I_DP &ya, Mat_I_DP &y2a,
    const DP x1, const DP x2, DP &y);
void spline(Vec_I_DP &x, Vec_I_DP &y, const DP yp1, const DP ypn,
    Vec_O_DP &y2);
void splint(Vec_I_DP &xa, Vec_I_DP &ya, Vec_I_DP &y2a, const DP x, DP &y);
void spread(const DP y, Vec_IO_DP &yy, const DP x, const int m);
void sprsax(Vec_I_DP &sa, Vec_I_INT &ija, Vec_I_DP &x, Vec_O_DP &b);
void sprsin(Mat_I_DP &a, const DP thresh, Vec_O_DP &sa, Vec_O_INT &ija);
void sprspm(Vec_I_DP &sa, Vec_I_INT &ija, Vec_I_DP &sb, Vec_I_INT &ijb,
    Vec_O_DP &sc, Vec_I_INT &ijc);
void sprstm(Vec_I_DP &sa, Vec_I_INT &ija, Vec_I_DP &sb, Vec_I_INT &ijb,
    const DP thresh, Vec_O_DP &sc, Vec_O_INT &ijc);
void sprstp(Vec_I_DP &sa, Vec_I_INT &ija, Vec_O_DP &sb, Vec_O_INT &ijb);
void sprstx(Vec_I_DP &sa, Vec_I_INT &ija, Vec_I_DP &x, Vec_O_DP &b);
void stifbs(Vec_IO_DP &y, Vec_IO_DP &dydx, DP &xx, const DP htry,
    const DP eps, Vec_I_DP &yscal, DP &hdid, DP &hnext,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &));
void stiff(Vec_IO_DP &y, Vec_IO_DP &dydx, DP &x, const DP htry,
    const DP eps, Vec_I_DP &yscal, DP &hdid, DP &hnext,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &));
void stoerm(Vec_I_DP &y, Vec_I_DP &d2y, const DP xs,
    const DP htot, const int nstep, Vec_O_DP &yout,
    void derivs(const DP, Vec_I_DP &, Vec_O_DP &));
void svbksb(Mat_I_DP &u, Vec_I_DP &w, Mat_I_DP &v, Vec_I_DP &b, Vec_O_DP &x);
void svdcmp(Mat_IO_DP &a, Vec_O_DP &w, Mat_O_DP &v);
void svdfit(Vec_I_DP &x, Vec_I_DP &y, Vec_I_DP &sig, Vec_O_DP &a,
    Mat_O_DP &u, Mat_O_DP &v, Vec_O_DP &w, DP &chisq,
    void funcs(const DP, Vec_O_DP &));
void svdvar(Mat_I_DP &v, Vec_I_DP &w, Mat_O_DP &cvm);
void toeplz(Vec_I_DP &r, Vec_O_DP &x, Vec_I_DP &y);
void tptest(Vec_I_DP &data1, Vec_I_DP &data2, DP &t, DP &prob);
void tqli(Vec_IO_DP &d, Vec_IO_DP &e, Mat_IO_DP &z);
DP trapzd(DP func(const DP), const DP a, const DP b, const int n);
void tred2(Mat_IO_DP &a, Vec_O_DP &d, Vec_O_DP &e);
void tridag(Vec_I_DP &a, Vec_I_DP &b, Vec_I_DP &c, Vec_I_DP &r, Vec_O_DP &u);
DP trncst(Vec_I_DP &x, Vec_I_DP &y, Vec_I_INT &iorder, Vec_IO_INT &n);
void trnspt(Vec_IO_INT &iorder, Vec_I_INT &n);
void ttest(Vec_I_DP &data1, Vec_I_DP &data2, DP &t, DP &prob);
void tutest(Vec_I_DP &data1, Vec_I_DP &data2, DP &t, DP &prob);
void twofft(Vec_I_DP &data1, Vec_I_DP &data2, Vec_O_DP &fft1,
    Vec_O_DP &fft2);
void vander(Vec_I_DP &x, Vec_O_DP &w, Vec_I_DP &q);
void vegas(Vec_I_DP &regn, DP fxn(Vec_I_DP &, const DP), const int init,
    const int ncall, const int itmx, const int nprn, DP &tgral, DP &sd,
    DP &chi2a);
void voltra(const DP t0, const DP h, Vec_O_DP &t, Mat_O_DP &f,
    DP g(const int, const DP),
    DP ak(const int, const int, const DP, const DP));
void wt1(Vec_IO_DP &a, const int isign,
    void wtstep(Vec_IO_DP &, const int, const int));
void wtn(Vec_IO_DP &a, Vec_I_INT &nn, const int isign,
    void wtstep(Vec_IO_DP &, const int, const int));
void wwghts(Vec_O_DP &wghts, const DP h,
    void kermom(Vec_O_DP &w, const DP y));
```

```
bool zbrac(DP func(const DP), DP &x1, DP &x2);
void zbrak(DP fx(const DP), const DP x1, const DP x2, const int n,
    Vec_O_DP &xb1, Vec_O_DP &xb2, int &nroot);
DP zbrent(DP func(const DP), const DP x1, const DP x2, const DP tol);
void zrhqr(Vec_I_DP &a, Vec_O_CPLX_DP &rt);
DP zriddr(DP func(const DP), const DP x1, const DP x2, const DP xacc);
void zroots(Vec_I_CPLX_DP &a, Vec_O_CPLX_DP &roots, const bool &polish);
}
#endif /* _NR_H_ */
```

附录 B　实用例程和类

无版权说明：本附录和它的实用例程都属于公共领域。任何人可以为任何目的自由地复制它们。当然，我们不承担任何这类用途的责任。

下面所列出的程序和类基本上用于本书的所有算法。首先，我们给出 typedefs 的默认集（参见 1.2 节）。在 Numerical Recipes 机器可读的版本中，就是文件 nrtypes_nr.h。不过在实际中，我们通过语句

```
#include "nrtypes.h"
```

包含这些定义。文件 nrtypes.h 是一个中间文件，它自身含有一个单独的包含语句，以调用一个"nrtypes"文件，后者由你所使用的矩阵/向量类库具体确定。nrtypes.h 的默认内容是

```
#include "nrtypes_nr.h"
```

它具有包括以下文件的效用：

```
#ifndef _NR_TYPES_H_
#define _NR_TYPES_H_

#include <complex>
#include <fstream>
#include "nrutil.h"
using namespace std;

typedef double DP;

// Vector Types

typedef const NRVec<bool> Vec_I_BOOL;
typedef NRVec<bool> Vec_BOOL, Vec_O_BOOL, Vec_IO_BOOL;

typedef const NRVec<char> Vec_I_CHR;
typedef NRVec<char> Vec_CHR, Vec_O_CHR, Vec_IO_CHR;

typedef const NRVec<unsigned char> Vec_I_UCHR;
typedef NRVec<unsigned char> Vec_UCHR, Vec_O_UCHR, Vec_IO_UCHR;

typedef const NRVec<int> Vec_I_INT;
typedef NRVec<int> Vec_INT, Vec_O_INT, Vec_IO_INT;

typedef const NRVec<unsigned int> Vec_I_UINT;
typedef NRVec<unsigned int> Vec_UINT, Vec_O_UINT, Vec_IO_UINT;

typedef const NRVec<long> Vec_I_LNG;
typedef NRVec<long> Vec_LNG, Vec_O_LNG, Vec_IO_LNG;

typedef const NRVec<unsigned long> Vec_I_ULNG;
typedef NRVec<unsigned long> Vec_ULNG, Vec_O_ULNG, Vec_IO_ULNG;

typedef const NRVec<float> Vec_I_SP;
typedef NRVec<float> Vec_SP, Vec_O_SP, Vec_IO_SP;

typedef const NRVec<DP> Vec_I_DP;
typedef NRVec<DP> Vec_DP, Vec_O_DP, Vec_IO_DP;

typedef const NRVec<complex<float> > Vec_I_CPLX_SP;
typedef NRVec<complex<float> > Vec_CPLX_SP, Vec_O_CPLX_SP, Vec_IO_CPLX_SP;

typedef const NRVec<complex<DP> > Vec_I_CPLX_DP;
typedef NRVec<complex<DP> > Vec_CPLX_DP, Vec_O_CPLX_DP, Vec_IO_CPLX_DP;

// Matrix Types
```

```
typedef const NRMat<bool> Mat_I_BOOL;
typedef NRMat<bool> Mat_BOOL, Mat_O_BOOL, Mat_IO_BOOL;

typedef const NRMat<char> Mat_I_CHR;
typedef NRMat<char> Mat_CHR, Mat_O_CHR, Mat_IO_CHR;

typedef const NRMat<unsigned char> Mat_I_UCHR;
typedef NRMat<unsigned char> Mat_UCHR, Mat_O_UCHR, Mat_IO_UCHR;

typedef const NRMat<int> Mat_I_INT;
typedef NRMat<int> Mat_INT, Mat_O_INT, Mat_IO_INT;

typedef const NRMat<unsigned int> Mat_I_UINT;
typedef NRMat<unsigned int> Mat_UINT, Mat_O_UINT, Mat_IO_UINT;

typedef const NRMat<long> Mat_I_LNG;
typedef NRMat<long> Mat_LNG, Mat_O_LNG, Mat_IO_LNG;

typedef const NRVec<unsigned long> Mat_I_ULNG;
typedef NRMat<unsigned long> Mat_ULNG, Mat_O_ULNG, Mat_IO_ULNG;

typedef const NRMat<float> Mat_I_SP;
typedef NRMat<float> Mat_SP, Mat_O_SP, Mat_IO_SP;

typedef const NRMat<DP> Mat_I_DP;
typedef NRMat<DP> Mat_DP, Mat_O_DP, Mat_IO_DP;

typedef const NRMat<complex<float> > Mat_I_CPLX_SP;
typedef NRMat<complex<float> > Mat_CPLX_SP, Mat_O_CPLX_SP, Mat_IO_CPLX_SP;

typedef const NRMat<complex<DP> > Mat_I_CPLX_DP;
typedef NRMat<complex<DP> > Mat_CPLX_DP, Mat_O_CPLX_DP, Mat_IO_CPLX_DP;

// 3D Matrix Types

typedef const NRMat3d<DP> Mat3D_I_DP;
typedef NRMat3d<DP> Mat3D_DP, Mat3D_O_DP, Mat3D_IO_DP;

// Miscellaneous Types
typedef NRVec<unsigned long *> Vec_ULNG_p;
typedef NRVec<NRMat<DP> *> Vec_Mat_DP_p;
typedef NRVec<fstream *> Vec_FSTREAM_p;

#endif /* _NR_TYPES_H_ */
```

下面列出默认的向量和矩阵类 NRVec 和 NRMat,包括其具体实施(参见 1.3 节)。

需要指出的是,我们不用 C 风格的数组作为向量或矩阵的专用数据变量。对于向量,我们用一个指针指向我们通过 new 明确分派的存储块。对二维数组,我们用一个指向指针数组的指针,在矩阵的每行(指针)数组元素指向第一个元素(参见图 B.1)。这种方式比固定尺寸的数组要灵活得多。

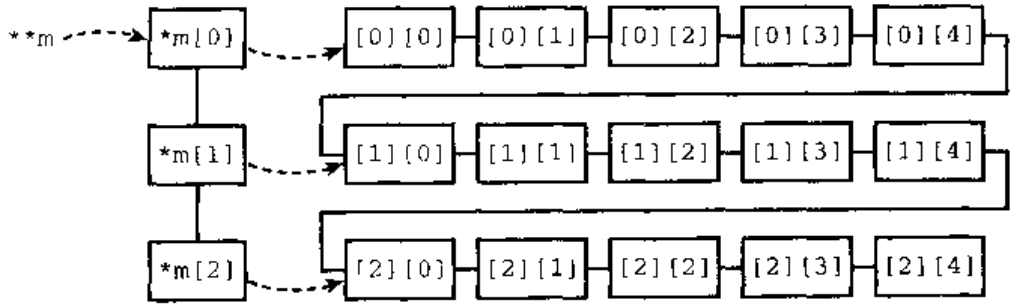


图 B.1 矩阵 m 的存储方式,其实施中作为指向指针数组之行的指针。点划线表示起始地址,实线连接了连续存储单元

实用程序 SQR,MAX,MIN,SIGN 和 SWAP 以及错误程序 nrerror 也在下面列出,当一个不可恢复的错误出现时,后者会发出适当的信息,然后关闭运行程序。

在 Numerical Recipes 的机器可读版本中，就是文件 nrutil_nr.h。然而，在实际中，我们通过语句

```
#include "nrutil.h"
```

包括这些定义。与 nrtypes.h 类似，文件 nrutil.h 是一个中间文件，它自身含有一个单独的包含语句，以调用一个"nrutil"文件，后者由你所使用的矩阵/向量类库具体确定。nrutil.h 的默认内容是

```
#include "nrutil_nr.h"
```

它有包含以下文件的效用：

```
#ifndef _NR_UTIL_H_
#define _NR_UTIL_H_

#include <string>
#include <cmath>
#include <complex>
#include <iostream>
using namespace std;

typedef double DP;
template<class T>
inline const T SQR(const T a) {return a*a;}

template<class T>
inline const T MAX(const T &a, const T &b)
{return b > a ? (b) : (a);}

inline float MAX(const double &a, const float &b)
{return b > a ? (b) : float(a);}

inline float MAX(const float &a, const double &b)
{return b > a ? float(b) : (a);}

template<class T>
inline const T MIN(const T &a, const T &b)
{return b < a ? (b) : (a);}

inline float MIN(const double &a, const float &b)
{return b < a ? (b) : float(a);}

inline float MIN(const float &a, const double &b)
{return b < a ? float(b) : (a);}

template<class T>
inline const T SIGN(const T &a, const T &b)
    {return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}

inline float SIGN(const float &a, const double &b)
    {return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}

inline float SIGN(const double &a, const float &b)
    {return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}

template<class T>
inline void SWAP(T &a, T &b)
    {T dum=a; a=b; b=dum;}

namespace NR {
    inline void nrerror(const string error_text)
    // Numerical Recipes standard error handler
    {
        cerr << "Numerical Recipes run-time error..." << endl;
        cerr << error_text << endl;
        cerr << "...now exiting to system..." << endl;
        exit(1);
    }
}

template <class T>
class NRVec {
```

```
private:
    int nn;    // size of array. upper index is nn-1
    T *v;
public:
    NRVec();
    explicit NRVec(int n);        // Zero-based array
    NRVec(const T &a, int n);    //initialize to constant value
    NRVec(const T *a, int n);    // Initialize to array
    NRVec(const NRVec &rhs);    // Copy constructor
    NRVec & operator=(const NRVec &rhs);    //assignment
    NRVec & operator=(const T &a);    //assign a to every element
    inline T & operator[](const int i);    //i'th element
    inline const T & operator[](const int i) const;
    inline int size() const;
    ~NRVec();
};

template <class T>
NRVec<T>::NRVec() : nn(0), v(0) {}

template <class T>
NRVec<T>::NRVec(int n) : nn(n), v(new T[n]) {}

template <class T>
NRVec<T>::NRVec(const T& a, int n) : nn(n), v(new T[n])
{
    for(int i=0; i<n; i++)
        v[i] = a;
}

template <class T>
NRVec<T>::NRVec(const T *a, int n) : nn(n), v(new T[n])
{
    for(int i=0; i<n; i++)
        v[i] = *a++;
}

template <class T>
NRVec<T>::NRVec(const NRVec<T> &rhs) : nn(rhs.nn), v(new T[nn])
{
    for(int i=0; i<nn; i++)
        v[i] = rhs[i];
}

template <class T>
NRVec<T> & NRVec<T>::operator=(const NRVec<T> &rhs)
// postcondition: normal assignment via copying has been performed;
//        if vector and rhs were different sizes, vector
//        has been resized to match the size of rhs
{
    if (this != &rhs)
    {
        if (nn != rhs.nn) {
            if (v != 0) delete [] (v);
            nn=rhs.nn;
            v= new T[nn];
        }
        for (int i=0; i<nn; i++)
            v[i]=rhs[i];
    }
    return *this;
}

template <class T>
NRVec<T> & NRVec<T>::operator=(const T &a)    //assign a to every element
{
    for (int i=0; i<nn; i++)
        v[i]=a;
    return *this;
}
```

```
template <class T>
inline T & NRVec<T>::operator[](const int i)    //subscripting
{
    return v[i];
}

template <class T>
inline const T & NRVec<T>::operator[](const int i) const    //subscripting
{
    return v[i];
}

template <class T>
inline int NRVec<T>::size() const
{
    return nn;
}

template <class T>
NRVec<T>::~NRVec()
{
    if (v != 0)
        delete[] (v);
}

template <class T>
class NRMat {
private:
    int nn;
    int mm;
    T **v;
public:
    NRMat();
    NRMat(int n, int m);            // Zero-based array
    NRMat(const T &a, int n, int m);    //Initialize to constant
    NRMat(const T *a, int n, int m);    // Initialize to array
    NRMat(const NRMat &rhs);        // Copy constructor
    NRMat & operator=(const NRMat &rhs);    //assignment
    NRMat & operator=(const T &a);        //assign a to every element
    inline T* operator[](const int i);    //subscripting: pointer to row i
    inline const T* operator[](const int i) const;
    inline int nrows() const;
    inline int ncols() const;
    ~NRMat();
};

template <class T>
NRMat<T>::NRMat() : nn(0), mm(0), v(0) {}

template <class T>
NRMat<T>::NRMat(int n, int m) : nn(n), mm(m), v(new T*[n])
{
    v[0] = new T[m*n];
    for (int i=1; i< n; i++)
        v[i] = v[i-1] + m;
}

template <class T>
NRMat<T>::NRMat(const T &a, int n, int m) : nn(n), mm(m), v(new T*[n])
{
    int i,j;
    v[0] = new T[m*n];
    for (i=1; i< n; i++)
        v[i] = v[i-1] + m;
    for (i=0; i< n; i++)
        for (j=0; j<m; j++)
            v[i][j] = a;
}

template <class T>
NRMat<T>::NRMat(const T *a, int n, int m) : nn(n), mm(m), v(new T*[n])
```

```
{
    int i,j;
    v[0] = new T[m*n];
    for (i=1; i< n; i++)
        v[i] = v[i-1] + m;
    for (i=0; i< n; i++)
        for (j=0; j<m; j++)
            v[i][j] = *a++;
}

template <class T>
NRMat<T>::NRMat(const NRMat &rhs) : nn(rhs.nn), mm(rhs.mm), v(new T*[nn])
{
    int i,j;
    v[0] = new T[mm*nn];
    for (i=1; i< nn; i++)
        v[i] = v[i-1] + mm;
    for (i=0; i< nn; i++)
        for (j=0; j<mm; j++)
            v[i][j] = rhs[i][j];
}

template <class T>
NRMat<T> & NRMat<T>::operator=(const NRMat<T> &rhs)
// postcondition: normal assignment via copying has been performed;
//        if matrix and rhs were different sizes, matrix
//        has been resized to match the size of rhs
{
    if (this != &rhs) {
        int i,j;
        if (nn != rhs.nn || mm != rhs.mm) {
            if (v != 0) {
                delete[] (v[0]);
                delete[] (v);
            }
            nn=rhs.nn;
            mm=rhs.mm;
            v = new T*[nn];
            v[0] = new T[mm*nn];
        }
        for (i=1; i< nn; i++)
            v[i] = v[i-1] + mm;
        for (i=0; i< nn; i++)
            for (j=0; j<mm; j++)
                v[i][j] = rhs[i][j];
    }
    return *this;
}

template <class T>
NRMat<T> & NRMat<T>::operator=(const T &a)    //assign a to every element
{
    for (int i=0; i< nn; i++)
        for (int j=0; j<mm; j++)
            v[i][j] = a;
    return *this;
}

template <class T>
inline T* NRMat<T>::operator[](const int i)    //subscripting: pointer to row i
{
    return v[i];
}
template <class T>
inline const T* NRMat<T>::operator[](const int i) const
{
    return v[i];
}

template <class T>
```

```
inline int NRMat<T>::nrows() const
{
    return nn;
}

template <class T>
inline int NRMat<T>::ncols() const
{
    return mm;
}

template <class T>
NRMat<T>::~NRMat()
{
    if (v != 0) {
        delete[] (v[0]);
        delete[] (v);
    }
}

template <class T>
class NRMat3d {
private:
    int nn;
    int mm;
    int kk;
    T ***v;
public:
    NRMat3d();
    NRMat3d(int n, int m, int k);
    inline T** operator[](const int i);    //subscripting: pointer to row i
    inline const T* const * operator[](const int i) const;
    inline int dim1() const;
    inline int dim2() const;
    inline int dim3() const;
    ~NRMat3d();
};

template <class T>
NRMat3d<T>::NRMat3d(): nn(0), mm(0), kk(0), v(0) {}

template <class T>
NRMat3d<T>::NRMat3d(int n, int m, int k) : nn(n), mm(m), kk(k), v(new T**[n])
{
    int i,j;
    v[0] = new T*[n*m];
    v[0][0] = new T[n*m*k];
    for(j=1; j<m; j++)
        v[0][j] = v[0][j-1] + k;
    for(i=1; i<n; i++) {
        v[i] = v[i-1] + m;
        v[i][0] = v[i-1][0] + m*k;
        for(j=1; j<m; j++)
            v[i][j] = v[i][j-1] + k;
    }
}
template <class T>
inline T** NRMat3d<T>::operator[](const int i) //subscripting: pointer to row i
{
    return v[i];
}

template <class T>
inline const T* const * NRMat3d<T>::operator[](const int i) const
{
    return v[i];
}

template <class T>
inline int NRMat3d<T>::dim1() const
```

```
{
    return nn;
}

template <class T>
inline int NRMat3d<T>::dim2() const
{
    return mm;
}

template <class T>
inline int NRMat3d<T>::dim3() const
{
    return kk;
}

template <class T>
NRMat3d<T>::~NRMat3d()
{
    if (v != 0) {
        delete[] (v[0][0]);
        delete[] (v[0]);
        delete[] (v);
    }
}

//The next 3 classes are used in artihmetic coding, Huffman coding, and
//wavelet transforms respectively. This is as good a place as any to put them!

class arithcode {
private:
    NRVec<unsigned long> *ilob_p,*iupb_p,*ncumfq_p;
public:
    NRVec<unsigned long> &ilob,&iupb,&ncumfq;
    unsigned long jdif,nc,minint,nch,ncum,nrad;
    arithcode(unsigned long n1, unsigned long n2, unsigned long n3)
        : ilob_p(new NRVec<unsigned long>(n1)),
        iupb_p(new NRVec<unsigned long>(n2)),
        ncumfq_p(new NRVec<unsigned long>(n3)),
        ilob(*ilob_p),iupb(*iupb_p),ncumfq(*ncumfq_p) {}
    ~arithcode() {
        if (ilob_p != 0) delete ilob_p;
        if (iupb_p != 0) delete iupb_p;
        if (ncumfq_p != 0) delete ncumfq_p;
    }
};

class huffcode {
private:
    NRVec<unsigned long> *icod_p,*ncod_p,*left_p,*right_p;
public:
    NRVec<unsigned long> &icod,&ncod,&left,&right;
    int nch,nodemax;
    huffcode(unsigned long n1, unsigned long n2, unsigned long n3,
        unsigned long n4) :
        icod_p(new NRVec<unsigned long>(n1)),
        ncod_p(new NRVec<unsigned long>(n2)),
        left_p(new NRVec<unsigned long>(n3)),
        right_p(new NRVec<unsigned long>(n4)),
        icod(*icod_p),ncod(*ncod_p),left(*left_p),right(*right_p) {}
    ~huffcode() {
        if (icod_p != 0) delete icod_p;
        if (ncod_p != 0) delete ncod_p;
        if (left_p != 0) delete left_p;
        if (right_p != 0) delete right_p;
    }
};

class wavefilt {
```

```
private:
    NRVec<DP> *cc_p,*cr_p;
public:
    int ncof,ioff,joff;
    NRVec<DP> &cc,&cr;
    wavefilt() : cc(*cc_p),cr(*cr_p) {}
    wavefilt(const DP *a, const int n) : //initialize to array
        cc_p(new NRVec<DP>(n)),cr_p(new NRVec<DP>(n)),
        ncof(n),ioff(-(n >> 1)),joff(-(n >> 1)),cc(*cc_p),cr(*cr_p) {
            int i;
            for (i=0; i<n; i++)
                cc[i] = *a++;
            DP sig = -1.0;
            for (i=0; i<n; i++) {
                cr[n-1-i]=sig*cc[i];
                sig = -sig;
            }
    }
    ~wavefilt() {
        if (cc_p != 0) delete cc_p;
        if (cr_p != 0) delete cr_p;
    }
};

//Overloaded complex operations to handle mixed float and double
//This takes care of e.g. 1.0/z, z complex<float>

inline const complex<float> operator+(const double &a,
    const complex<float> &b) { return float(a)+b; }

inline const complex<float> operator+(const complex<float> &a,
    const double &b) { return a+float(b); }

inline const complex<float> operator-(const double &a,
    const complex<float> &b) { return float(a)-b; }

inline const complex<float> operator-(const complex<float> &a,
    const double &b) { return a-float(b); }

inline const complex<float> operator*(const double &a,
    const complex<float> &b) { return float(a)*b; }

inline const complex<float> operator*(const complex<float> &a,
    const double &b) { return a*float(b); }
inline const complex<float> operator/(const double &a,
    const complex<float> &b) { return float(a)/b; }

inline const complex<float> operator/(const complex<float> &a,
    const double &b) { return a/float(b); }

//some compilers choke on pow(float,double) in single precision. also atan2

inline float pow (float x, double y) {return pow(double(x),y);}
inline float pow (double x, float y) {return pow(x,double(y));}
inline float atan2 (float x, double y) {return atan2(double(x),y);}
inline float atan2 (double x, float y) {return atan2(x,double(y));}
#endif /* _NR_UTIL_H_ */
```

如果你想使用不同于上述默认的矩阵/向量类库，你需要做两件事。首先，如同在 1.3 节中所述，要改变类型说明的所有 const 属性。以下是文件 nrtypes_nr.h 的列表，用它将包含在 nrtypes.h 中的文件从 nrtypes_nr.h 改变到 nrtypes_lib.h 。

```
#ifndef _NR_TYPES_H_
#define _NR_TYPES_H_

#include <complex>
#include <fstream>
#include "nrutil.h"
using namespace std;

typedef double DP;
```

```
// Vector Types

typedef const NRVec<bool> Vec_I_BOOL;
typedef const NRVec<bool> Vec_BOOL, Vec_O_BOOL, Vec_IO_BOOL;

typedef const NRVec<char> Vec_I_CHR;
typedef const NRVec<char> Vec_CHR, Vec_O_CHR, Vec_IO_CHR;

typedef const NRVec<unsigned char> Vec_I_UCHR;
typedef const NRVec<unsigned char> Vec_UCHR, Vec_O_UCHR, Vec_IO_UCHR;

typedef const NRVec<int> Vec_I_INT;
typedef const NRVec<int> Vec_INT, Vec_O_INT, Vec_IO_INT;

typedef const NRVec<unsigned int> Vec_I_UINT;
typedef const NRVec<unsigned int> Vec_UINT, Vec_O_UINT, Vec_IO_UINT;

typedef const NRVec<long> Vec_I_LNG;
typedef const NRVec<long> Vec_LNG, Vec_O_LNG, Vec_IO_LNG;

typedef const NRVec<unsigned long> Vec_I_ULNG;
typedef const NRVec<unsigned long> Vec_ULNG, Vec_O_ULNG, Vec_IO_ULNG;

typedef const NRVec<float> Vec_I_SP;
typedef const NRVec<float> Vec_SP, Vec_O_SP, Vec_IO_SP;

typedef const NRVec<DP> Vec_I_DP;
typedef const NRVec<DP> Vec_DP, Vec_O_DP, Vec_IO_DP;

typedef const NRVec<complex<float> > Vec_I_CPLX_SP;
typedef const NRVec<complex<float> > Vec_CPLX_SP, Vec_O_CPLX_SP, Vec_IO_CPLX_SP;
typedef const NRVec<complex<DP> > Vec_I_CPLX_DP;
typedef const NRVec<complex<DP> > Vec_CPLX_DP, Vec_O_CPLX_DP, Vec_IO_CPLX_DP;

// Matrix Types

typedef const NRMat<bool> Mat_I_BOOL;
typedef const NRMat<bool> Mat_BOOL, Mat_O_BOOL, Mat_IO_BOOL;

typedef const NRMat<char> Mat_I_CHR;
typedef const NRMat<char> Mat_CHR, Mat_O_CHR, Mat_IO_CHR;

typedef const NRMat<unsigned char> Mat_I_UCHR;
typedef const NRMat<unsigned char> Mat_UCHR, Mat_O_UCHR, Mat_IO_UCHR;

typedef const NRMat<int> Mat_I_INT;
typedef const NRMat<int> Mat_INT, Mat_O_INT, Mat_IO_INT;

typedef const NRMat<unsigned int> Mat_I_UINT;
typedef const NRMat<unsigned int> Mat_UINT, Mat_O_UINT, Mat_IO_UINT;

typedef const NRMat<long> Mat_I_LNG;
typedef const NRMat<long> Mat_LNG, Mat_O_LNG, Mat_IO_LNG;

typedef const NRVec<unsigned long> Mat_I_ULNG;
typedef const NRMat<unsigned long> Mat_ULNG, Mat_O_ULNG, Mat_IO_ULNG;

typedef const NRMat<float> Mat_I_SP;
typedef const NRMat<float> Mat_SP, Mat_O_SP, Mat_IO_SP;

typedef const NRMat<DP> Mat_I_DP;
typedef const NRMat<DP> Mat_DP, Mat_O_DP, Mat_IO_DP;

typedef const NRMat<complex<float> > Mat_I_CPLX_SP;
typedef const NRMat<complex<float> > Mat_CPLX_SP, Mat_O_CPLX_SP, Mat_IO_CPLX_SP;

typedef const NRMat<complex<DP> > Mat_I_CPLX_DP;
typedef const NRMat<complex<DP> > Mat_CPLX_DP, Mat_O_CPLX_DP, Mat_IO_CPLX_DP;
```

```
// 3D Matrix Types

typedef const NRMat3d<DP> Mat3D_I_DP;
typedef NRMat3d<DP> Mat3D_DP, Mat3D_O_DP, Mat3D_IO_DP;

// Miscellaneous Types

typedef NRVec<unsigned long *> Vec_ULNG_p;
typedef const NRVec<const NRMat<DP> *> Vec_Mat_DP_p;
typedef NRVec<fstream *> Vec_FSTREAM_p;

#endif /*  NR TYPES H  */
```

你需要做的第二件事，就是用一个不同的矩阵/向量库，以提供对 nrutil.h 的一个包装类(wrapper-class)替代。例如，我们在此列出文件 nrutil_tnt.h 以便读者能够用 TNT 矩阵/向量库[1]代替我们的默认值。利用它将包含在 nrutil.h 中的文件从 nrutil_nr.h 改变为 nrutil_tnt.h。

```
#ifndef _NR_UTIL_H_
#define _NR_UTIL_H_

#include <string>
#include <cmath>
#include <complex>
#include <iostream>
using namespace std;

typedef double DP;

template<class T>
inline const T SQR(const T a) {return a*a;}

template<class T>
inline const T MAX(const T &a, const T &b)
{return b > a ? (b) : (a);}

inline float MAX(const double &a, const float &b)
{return b > a ? (b) : float(a);}

inline float MAX(const float &a, const double &b)
{return b > a ? float(b) : (a);}

template<class T>
inline const T MIN(const T &a, const T &b)
{return b < a ? (b) : (a);}

inline float MIN(const double &a, const float &b)
{return b < a ? (b) : float(a);}

inline float MIN(const float &a, const double &b)
{return b < a ? float(b) : (a);}

template<class T>
inline const T SIGN(const T &a, const T &b)
    {return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}

inline float SIGN(const float &a, const double &b)
    {return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}

inline float SIGN(const double &a, const float &b)
    {return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}

template<class T>
inline void SWAP(T &a, T &b)
    {T dum=a; a=b; b=dum;}

namespace NR {
```

```
    inline void nrerror(const string error_text)
    // Numerical Recipes standard error handler
    {
        cerr << "Numerical Recipes run-time error..." << endl;
        cerr << error_text << endl;
        cerr << "...now exiting to system..." << endl;
        exit(1);
    }
}

#include "tnt/tnt.h"
#include "tnt/vec.h"
#include "tnt/cmat.h"

// TNT Wrapper File
// This is the file that "joins" the TNT Vector<> and Matrix<> classes
// to the NRVec and NRMat classes by the Wrapper Class Method

// NRVec contains a Vector and a &Vector. All its constructors, except the
// conversion constructor, create the Vector and point the &Vector to it.
// The conversion constructor only points the &Vector. All operations
// (size, subscript) are through the &Vector, which as a reference
// (not pointer) has no indirection overhead.

template<class T>
class NRVec {
private:
    TNT::Vector<T> myvec;
    TNT::Vector<T> &myref;
public:
    NRVec<T>() : myvec(), myref(myvec) {}
    explicit NRVec<T>(const int n) : myvec(n), myref(myvec) {}
    NRVec<T>(const T &a, int n) : myvec(n,a), myref(myvec) {}
    NRVec<T>(const T *a, int n) : myvec(n,a), myref(myvec) {}
    NRVec<T>(TNT::Vector<T> &rhs) : myref(rhs) {}
    // conversion constructor makes a special NRVec pointing to Vector's data
    // this handles Vector actual args sent to NRVec formal args in functions
    NRVec(const NRVec<T>& rhs) : myvec(rhs.myref), myref(myvec) {}
    // copy constructor calls Vector copy constructor
    inline NRVec& operator=(const NRVec& rhs) { myref=rhs.myref; return *this;}
    // assignment operator calls Vector assignment operator
    inline NRVec& operator=(const T& rhs) { myvec=rhs; return *this;}
    // scalar assignment calls Vector assignment operator
    inline int size() const {return myref.size();}
    inline T & operator[](const int i) const {return myref[i];}
    // return element i
    inline operator TNT::Vector<T>() const {return myref;}
    // conversion operator to Vector
    // this handles NRVec function return types when used in Vector expressions
    ~NRVec() {}
};

template <class T>
class NRMat {
private:
    TNT::Matrix<T> mymat;
    TNT::Matrix<T> &myref;
public:
    NRMat() : mymat(), myref(mymat) {}
    NRMat(int n, int m) : mymat(n,m), myref(mymat) {}
    NRMat(const T& a, int n, int m) : mymat(n,m,a), myref(mymat) {}
    //Initialize to constant
    NRMat(const T* a, int n, int m) : mymat(n,m,a), myref(mymat) {}
    //Initialize to array
    NRMat<T>(TNT::Matrix<T> &rhs) : myref(rhs) {}
    // conversion constructor from Matrix
    NRMat(const NRMat& rhs) : mymat(rhs.myref), myref(mymat) {}
    // copy constructor
    inline NRMat& operator=(const NRMat& rhs) { myref=rhs.myref; return *this;}
```

```
    // assignment operator
    inline NRMat& operator=(const T& rhs) { mymat=rhs; return *this;}
    // scalar assignment calls Matrix assignment operator
    inline T* operator[](const int i) const {return myref[i];}
    //subscripting: pointer to row i
    //return type is whatever Matrix returns for a single [] dereference
    inline int nrows() const {return myref.num_rows();}
    inline int ncols() const {return myref.num_cols();}
    inline operator TNT::Matrix<T>() const {return myref;}
    // conversion operator to Matrix
    ~NRMat() {}
};

template <class T>
class NRMat3d {
The rest of the code is identical to nrutil_nr.h and is omitted.
```

最后，我们列出文件 nrutil_mtl.h 以便读者能够用 MTL 矩阵/向量库[2]。同样，还要将包含在 nrutil.h 中的文件从 nrutil_nr.h 改变为 nrutil_mtl.h。

要注意的是，MTL 库用类 std::vector 定义它的基本向量类型。这个类是专门对 vector < bool > 的，后者在我们的包装类方式下不起作用。所以，我们提供 NRVec < bool > 作为我们自己的 NRVec 专用化，以便由此导出 NRVec < int > 类(我们知道，如果需要，变量 bool 会自动转换为 int)。如此进行只有一个潜在的问题：如果轻率地将类型 mtl::Vector < bool > 的对象与类型 NRVec < bool > 的对象混合使用！

```
#ifndef _NR_UTIL_H_
#define _NR_UTIL_H_

#include <string>
#include <cmath>
#include <complex>
#include <iostream>
using namespace std;

typedef double DP;

template<class T>
inline const T SQR(const T a) {return a*a;}

template<class T>
inline const T MAX(const T &a, const T &b)
{return b > a ? (b) : (a);}

inline float MAX(const double &a, const float &b)
{return b > a ? (b) : float(a);}

inline float MAX(const float &a, const double &b)
{return b > a ? float(b) : (a);}

template<class T>
inline const T MIN(const T &a, const T &b)
{return b < a ? (b) : (a);}

inline float MIN(const double &a, const float &b)
{return b < a ? (b) : float(a);}

inline float MIN(const float &a, const double &b)
{return b < a ? float(b) : (a);}

template<class T>
inline const T SIGN(const T &a, const T &b)
    {return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}

inline float SIGN(const float &a, const double &b)
    {return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}

inline float SIGN(const double &a, const float &b)
    {return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}
```

```
template<class T>
inline void SWAP(T &a, T &b)
    {T dum=a; a=b; b=dum;}

namespace NR {
    inline void nrerror(const string error_text)
    // Numerical Recipes standard error handler
    {
        cerr << "Numerical Recipes run-time error..." << endl;
        cerr << error_text << endl;
        cerr << "...now exiting to system..." << endl;
        exit(1);
    }
}

#include "mtl/mtl.h"
using namespace mtl;

// mtl Wrapper File
// This is the file that "joins" the mtl Vector<> and Matrix<> classes
// to the NRVec and NRMat classes by the Wrapper Class Method

// NRVec contains a Vector and a &Vector. All its constructors, except the
// conversion constructor, create the Vector and point the &Vector to it.
// The conversion constructor only points the &Vector. All operations
// (size, subscript) are through the &Vector, which as a reference
// (not pointer) has no indirection overhead.

template<class T>
class NRVec {
// Use the std::vector based dense1D for our Vector type
typedef dense1D<T> Vector;
protected:        //access required in NRVec<bool> below
    Vector myvec;
    Vector &myref;
public:
    NRVec<T>() : myvec(), myref(myvec) {}
    explicit NRVec<T>(const int n) : myvec(n), myref(myvec) {}
    NRVec<T>(const T &a, int n) : myvec(n), myref(myvec) {
        for (int i=0; i<n; i++) myvec[i] = a;}
    NRVec<T>(const T *a, int n) : myvec(n), myref(myvec) {
        for (int i=0; i<n; i++) myvec[i] = *a++;}
    NRVec<T>(Vector &rhs) : myref(rhs) {}
    // conversion constructor makes a special NRVec pointing to Vector's data
    // this handles Vector actual args sent to NRVec formal args in functions
    NRVec(const NRVec<T>& rhs) : myvec(rhs.myref.size()), myref(myvec)
        {copy(rhs.myref,myref);}
    // copy constructor. mtl copy constructor
    // does shallow copy only. so use copy() instead
    inline NRVec& operator=(const NRVec& rhs) {
        if (myref.size() != rhs.myref.size())
            myref.resize(rhs.myref.size());
        copy(rhs.myref,myref); return *this;}
    inline int size() const {return myref.size();}
    inline T & operator[](const int i) const {return myref[i];}
    inline operator Vector() const {return myref;}
    // conversion operator to Vector
    // handles NRVec function return types when used in Vector expressions
    ~NRVec() {}
};

//The std:vector class has a specialization for vector<bool> that doesn't
//work with the above wrapper class scheme. So implement our own
//specialization as a derived class of NRVec<int>. This could cause
//problems if you mix mtl::Vector<bool> with NRVec<bool>!

template <> class NRVec<bool> : public NRVec<int> {
public:
    NRVec() : NRVec<int>() {}
    explicit NRVec(const int n) : NRVec<int>(n) {}
```

```
    NRVec(const bool &a, int n) : NRVec<int>(int(a),n) {}
    NRVec(const bool *a, int n) : NRVec<int>(n) {
        for (int i=0; i<n; i++) myvec[i] = *a++;}
//note: defaults OK for copy constructor and assignment
};

template <class T>
class NRMat {
// Use the matrix generator to select a matrix type
typedef matrix< T,
    rectangle<>,
    dense<>,
    row_major>::type Matrix;
protected:
    Matrix mymat;
    Matrix &myref;
public:
    NRMat() : mymat(), myref(mymat) {}
    NRMat(int n, int m) : mymat(n,m), myref(mymat) {}
    NRMat(const T& a, int n, int m) : mymat(n,m), myref(mymat) {
        for (int i=0; i< n; i++)
            for (int j=0; j<m; j++)
                mymat[i][j] = a;}
    NRMat(const T* a, int n, int m) : mymat(n,m), myref(mymat) {
        for (int i=0; i< n; i++)
            for (int j=0; j<m; j++)
                mymat[i][j] = *a++;}
    NRMat<T>(Matrix &rhs) : myref(rhs) {}
    NRMat(const NRMat& rhs) :
        mymat(rhs.myref.nrows(),rhs.myref.ncols()), myref(mymat)
        {copy(rhs.myref,myref);}
    inline NRMat& operator=(const NRMat& rhs) {
        if (myref.nrows() != rhs.myref.nrows() && myref.ncols() !=
            rhs.myref.ncols()) {
                cerr << "assignment with incompatible matrix sizes\n";
                abort();
        }
        copy(rhs.myref,myref); return *this;
    }
    typename Matrix::OneD operator[](const int i) const {return myref[i];}
    //return type is whatever Matrix returns for a single [] dereference
    inline int nrows() const {return myref.nrows();}
    inline int ncols() const {return myref.ncols();}
    inline operator Matrix() const {return myref;}
    ~NRMat() {}
};

template <> class NRMat<bool> : public NRMat<int> {
public:
    NRMat() : NRMat<int>() {}
    explicit NRMat(int n, int m) : NRMat<int>(n,m) {}
    NRMat(const bool &a, int n, int m) : NRMat<int>(int(a),n,m) {}
    NRMat(const bool *a, int n, int m) : NRMat<int>(n,m) {
        for (int i=0; i< n; i++)
            for (int j=0; j<m; j++)
                mymat[i][j] = *a++;}
//note: defaults OK for copy constructor and assignment
};

template <class T>
class NRMat3d {
The rest of the code is identical to nrutil_nr.h and is omitted.
```

参考文献和深入阅读

Pozo, R., *Template Numerical Toolkit*, http://math.nist.gov/tnt. [1]

Lumsdaine, A., and Siek, J. 1998, *The Matrix Template Library*, http://www.lsc.nd.edu/research/mtl. [2]

附录 C　转换为单精度

正如 1.2 节所论述，最好使用我们提供的默认模式，即双精度使用本书算法。不过，如果想将它们都改变为单精度，可以按以下步骤进行：

- 在 nrtypes.h 和 nrutil.h 两者中将定义

  ```
  typedef double DP;
  ```

  改变为

  ```
  typedef float DP;
  ```

  （我们知道，这些程序基本上恰好包含默认程序 nrtypes_nr.h 和 nrutil_nr.h，它们是将要真正进行改变之处。）
- 改变下面所附表格内列出的程序中的"精度参数"（注意，这些仅仅是建议值。有些时候需要凭经验处理问题）。为了易于进行这些改变，我们在机器可读的版本中提供了一个 Perl 脚本文件 nrtosp.pl。从文件 nrtosp.dat 中它读取表格信息，如果想做任何其他改变，你可以很方便地编辑该文件。

最后要提醒读者：一些算法对舍入误差特别敏感，随之在单精度下给出不精确的结果。对这类易受影响的程序，我们已经在注释中提及。

程序名	改变	成为
beschb	NUSE1=7,NUSE2=8	NUSE1=5,NUSE2=5
bessi	ACC=200	ACC=40
bessj	ACC=160	ACC=40
broydn	TOLF=1.0e-8,TOLMIN=1.0e-12	TOLF=1.0e-4,TOLMIN=1.0e-6
dlinmin	TOL=2.0e-8	TOL=1.0e-4
fdjac	EPS=1.0e-8	EPS=1.0e-4
frprmn	EPS=1.0e-18	EPS=1.0e-10
gauher	EPS=1.0e-14	EPS=4.0e-7
gaujac	EPS=1.0e-14	EPS=1.0e-7
gaulag	EPS=1.0e-14	EPS=1.0e-6
gauleg	EPS=1.0e-14	EPS=1.0e-7
hypgeo	EPS=1.0e-14	EPS=1.0e-6
linmin	TOL=1.0e-8	TOL=1.0e-4
newt	TOLF=1.0e-8,TOLMIN=1.0e-12	TOLF=1.0e-4,TOLMIN=1.0e-6
probks	EPS1=1.0e-6,EPS2=1.0e-16	EPS1=0.001,EPS2=1.0e-8
qromb	EPS=1.0e-10	EPS=1.0e-6
qromo	EPS=3.0e-9	EPS=1.0e-6
qroot	TINY=1.0e-14	TINY=1.0e-6
qsimp	EPS=1.0e-10	EPS=1.0e-6
qtrap	EPS=1.0e-10	EPS=1.0e-6
ran1	EPS=3.0e-16	EPS=1.2e-7
ran2	EPS=3.0e-16	EPS=1.2e-7
rc	ERRTOL=0.0012	ERRTOL=0.04
rd	ERRTOL=0.0015	ERRTOL=0.05
rf	ERRTOL=0.0025	ERRTOL=0.08
rj	ERRTOL=0.0015	ERRTOL=0.05

程序名	改变	成为
sfroid	conv=1.0e-14	conv=5.0e-6
shoot	EPS=1.0e-14	EPS=1.0e-6
shootf	EPS=1.0e-14	EPS=1.0e-6
simp2	EPS=1.0e-14	EPS=1.0e-6
simplx	EPS=1.0e-14	EPS=1.0e-6
sncndn	CA=1.0e-8	CA=.0003
sor	EPS=1.0e-13	EPS=1.0e-4
sphfpt	DXX=1.0e-8	DXX=1.0e-4
sphoot	dx=1.0e-8	dx=1.0e-4
svdfit	TOL=1.0e-13	TOL=1.0e-5
zroots	EPS=1.0e-14	EPS=1.0e-6

参考文献

此处所列的参考书都有广泛的用途，通常在本书中不只一次引用。一些更专业的资料，往往只在本书某一节中被引用，故在此不再列出。

首先列出少量向读者推荐的书籍，它们是数值方法和数值分析方面值得参考的核心著作，也是我们学习数值方法所必备的书籍。

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York)

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America)

Ames, W.F. 1977, *Numerical Methods for Partial Differential Equations*, 2nd ed. (New York: Academic Press)

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, *A Guide to Simulation*, 2nd ed. (New York: Springer-Verlag)

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall)

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press)

Dennis, J.E., and Schnabel, R.B. 1983, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*; reprinted 1996 (Philadelphia: S.I.A.M.)

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley)

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press)

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall)

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed.; reprinted 2001 (New York: Dover)

Sedgewick, R. 1998, *Algorithms in C*, 3rd ed. (Reading, MA: Addison-Wesley)

Stoer, J., and Bulirsch, R. 1993, *Introduction to Numerical Analysis*, 2nd ed. (New York: Springer-Verlag)

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag)

其次列出的大量书籍，我们认为在任何一个正式研究计算、数值方法或数值分析的图书馆中都应收藏。

Anderson, E., et al. 2000, LAPACK User's Guide, 3rd ed. (Philadelphia: S.I.A.M.)

Bevington, P.R., and Robinson, D.K. 1992, *Data Reduction and Error Analysis for the Physical Sciences*, 2nd ed. (New York: McGraw-Hill)

Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley)

Bowers, R.L., and Wilson, J.R. 1991, *Numerical Modeling in Applied Physics and Astrophysics* (Boston: Jones & Bartlett)

Brent, R.P. 1973, *Algorithms for Minimization without Derivatives* (Englewood Cliffs, NJ: Prentice-Hall)

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall)

Brownlee, K.A. 1965, *Statistical Theory and Methodology*, 2nd ed. (New York: Wiley)

Bunch, J.R., and Rose, D.J. (eds.) 1976, *Sparse Matrix Computations* (New York: Academic Press)

Canuto, C., Hussaini, M.Y., Quarteroni, A., and Zang, T.A. 1988, *Spectral Methods in Fluid Dynamics* (New York: Springer-Verlag)

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley)

Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press)

Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press)

Cooper, L., and Steinberg, D. 1970, *Introduction to Methods of Optimization* (Philadelphia: Saunders)

Dantzig, G.B. 1963, *Linear Programming and Extensions* (Princeton, NJ: Princeton University Press)

Devroye, L. 1986, *Non-Uniform Random Variate Generation* (New York: Springer-Verlag)

Downie, N.M., and Heath, R.W. 1965, *Basic Statistical Methods*, 2nd ed. (New York: Harper & Row)

Duff, I.S., and Stewart, G.W. (eds.) 1979, *Sparse Matrix Proceedings 1978* (Philadelphia: S.I.A.M.)

Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press)

Fike, C.T. 1968, *Computer Evaluation of Mathematical Functions* (Englewood Cliffs, NJ: Prentice-Hall)

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall)

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall)

Gass, S.I. 1985, *Linear Programming*, 5th ed. (New York: McGraw-Hill)

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall)

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library)

Gottlieb, D. and Orszag, S.A. 1977, *Numerical Analysis of Spectral Methods: Theory and Applications* (Philadelphia: S.I.A.M.)

Hackbusch, W. 1985, *Multi-Grid Methods and Applications* (New York: Springer-Verlag)

Hamming, R.W. 1962, *Numerical Methods for Engineers and Scientists*; reprinted 1986 (New York: Dover)

Hart, J.F., et al. 1968, *Computer Approximations* (New York: Wiley)

Hastings, C. 1955, *Approximations for Digital Computers* (Princeton: Princeton University Press)

Hildebrand, F.B. 1974, *Introduction to Numerical Analysis*, 2nd ed.; reprinted 1987 (New York: Dover)

Hoel, P.G. 1971, *Introduction to Mathematical Statistics*, 4th ed. (New York: Wiley)

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press)

Householder, A.S. 1970, *The Numerical Treatment of a Single Nonlinear Equation* (New York: McGraw-Hill)

Huber, P.J. 1981, *Robust Statistics* (New York: Wiley)

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods*; reprinted 1994 (New York: Dover)

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press)

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley)

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall)

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems*; reprinted 1991 (New York: Dover)

Knuth, D.E. 1997, *Fundamental Algorithms*, 3rd ed., vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley)

Knuth, D.E. 1997, *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley)

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley)

Koonin, S.E., and Meredith, D.C. 1990, *Computational Physics, Fortran Version* (Redwood City, CA: Addison-Wesley)

Kuenzi, H.P., Tzschach, H.G., and Zehnder, C.A. 1971, *Numerical Methods of Mathematical Optimization* (New York: Academic Press)

Lanczos, C. 1956, *Applied Analysis*; reprinted 1988 (New York: Dover)

Land, A.H., and Powell, S. 1973, *Fortran Codes for Mathematical Programming* (London: Wiley-Interscience)

Lawson, C.L., and Hanson, R. 1974, *Solving Least Squares Problems* (Englewood Cliffs, NJ: Prentice-Hall)

Lehmann, E.L. 1975, *Nonparametrics: Statistical Methods Based on Ranks* (San Francisco: Holden-Day)

Luke, Y.L. 1975, *Mathematical Functions and Their Approximations* (New York: Academic Press)

Magnus, W., and Oberhettinger, F. 1949, *Formulas and Theorems for the Functions of Mathematical Physics* (New York: Chelsea)

Martin, B.R. 1971, *Statistics for Physicists* (New York: Academic Press)

Mathews, J., and Walker, R.L. 1970, *Mathematical Methods of Physics*, 2nd ed. (Reading, MA: W.A. Benjamin/Addison-Wesley)

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press)

Murty, K.G. 1976, *Linear and Combinatorial Programming* (New York: Wiley)

Norusis, M.J. 1999, *SPSS 9.0 Guide to Data Analysis* (Englewood Cliffs, NJ: Prentice-Hall)

Nussbaumer, H.J. 1982, *Fast Fourier Transform and Convolution Algorithms* (New York: Springer-Verlag)

Ortega, J., and Rheinboldt, W. 1970, *Iterative Solution of Nonlinear Equations in Several Variables* (New York: Academic Press)

Ostrowski, A.M. 1966, *Solutions of Equations and Systems of Equations*, 2nd ed. (New York: Academic Press)

Polak, E. 1971, *Computational Methods in Optimization* (New York: Academic Press)

Rice, J.R. 1983, *Numerical Methods, Software, and Analysis* (New York: McGraw-Hill)

Richtmyer, R.D., and Morton, K.W. 1967, *Difference Methods for Initial Value Problems*, 2nd ed. (New York: Wiley-Interscience)

Roache, P.J. 1976, *Computational Fluid Dynamics* (Albuquerque: Hermosa)

Robinson, E.A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, NJ: Prentice-Hall)

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag)

Strang, G. 1988, *Linear Algebra and its Applications*, 3rd ed. (New York: Harcourt Brace and Co.)

Stuart, A., and Ord, J.K. 1994, *Kendall's Advanced Theory of Statistics*, 6th ed. (London: Edward Arnold) [previous eds. published as Kendall, M., and Stuart, A., *The Advanced Theory of Statistics*]

Stroustrup, B. 1997, *The C++ Programming Language*, 3rd ed. (Reading, MA: Addison-Wesley)

Tewarson, R.P. 1973, *Sparse Matrices* (New York: Academic Press)

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley)

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press)

Young, D.M., and Gregory, R.T. 1973, *A Survey of Numerical Mathematics*, 2 vols.; reprinted 1988 (New York: Dover)

程序从属表

下面，按拉丁字母顺序列出本书中所有的程序。当某一程序需用到其他子程序时，无论是从本书中取得或用户自己提供，其依从关系用从属树图表示：一个程序直接调用的所有程序列在紧靠它的右边一列中，用直实线连接；间接调用的程序列在右边其他列中，用直实线将它与相关程序连接起来。印刷符号的约定为：书中的程序用正体字（如 eulsum 和 gammln），而斜体字用来表示在单个从属树中第二级和随后出现的程序（当你从 Numerical Recipes 磁盘或者它们的档案文件中获取程序时，只需用正体字）。用户自己需提供的程序用正体字和方括号指出，例如[*funcv*]。为了便于查阅这些程序的正文说明，列表最右端列出每个程序所处的章节。

```
addint ── interp ..................................... § 19.6
airy ─┬─ bessik ─┐
      └─ bessjy ── beschb ── chebev ..................... § 6.7
amebsa ─┬─ ran1 ..................................... § 10.9
        ├─ amotsa ─┬─ [funk]
        │          └─ ran1
        └─ [funk]
amoeba ─┬─ amotry ── [funk] ............................. § 10.4
        └─ [funk]
amotry ── [funk] ............................ § 10.4
amotsa ─┬─ [funk] ..................................... § 10.9
        └─ ran1
anneal ─┬─ ran3 ..................................... § 10.9
        │
        ├─ irbit1
        ├─ trncst
        ├─ metrop ─ ran3
        ├─ trnspt
        ├─ revcst
        └─ reverse
anorm2 ..................................... § 19.6
arcmak ..................................... § 20.5
arcode ── arcsum ........................... § 20.5
arcsum ..................................... § 20.5
avevar ..................................... § 14.2
badluk ─┬─ julday ..................................... § 1.1
        └─ flmoon
balanc ..................................... § 11.5
banbks ..................................... § 2.4
bandec ..................................... § 2.4
banmul ..................................... § 2.4
beucof ..................................... § 3.6
```

bcuint —— bcucof …… § 3.6
beschb —— chebev …… § 6.7
bessi —— bessi0 …… § 6.6
bessi0 …… § 6.6
bessi1 …… § 6.6
bessik —— beschb —— chebev …… § 6.7
bessj ┬ bessj0 …… § 6.5
└ bessj1
bessj0 …… § 6.5
bessj1 …… § 6.5
bessjy —— beschb —— chebev …… § 6.7
bessk ┬ bessk0 —— bessi0 …… § 6.6
└ bessk1 —— bessi1
bessk0 —— bessi0 …… § 6.6
bessk1 —— bessi1 …… § 6.6
bessy ┬ bessy1 —— bessj1 …… § 6.5
└ bessy0 —— bessj0
bessy0 —— bessj0 …… § 6.5
bessy1 —— bessj1 …… § 6.5
beta —— gammln …… § 6.1
betacf …… § 6.4
betai ┬ gammln …… § 6.4
└ betacf
bico —— factln —— gammln …… § 6.1
bksub …… § 17.3
bnldev ┬ ran1 …… § 7.3
└ gammln
brent —— [func] …… § 10.2
broydn ┬ fmin ┬ …… § 9.7
│ └ fdjac ┴ [funcv]
├ qrdcmp
├ qrupdt —— rotate
└ rsolv
└ lnsrch —— *fmin* —— [funcv]
bsstep ┬ mmid —— [derivs] …… § 16.4
└ pzextr
caldat …… § 1.1
chder …… § 5.9
chebev …… § 5.8
chebft —— [func] …… § 5.8
chebpc …… § 5.10
chint …… § 5.9
chixy …… § 15.3

```
cholde ····· § 2.9
cholsl ····· § 2.9
chsone ── gammq ┬─ gser ┐
                │       ├─ gammln
                └─ gcf ┘
                                ····· § 14.3
chstwo ── gammq ┬─ gser ┐
                │       ├─ gammln
                └─ gcf ┘
                                ····· § 14.3
cisi ····· § 6.9
cntab1 ── gammq ┬─ gser ┐
                │       ├─ gammln
                └─ gcf ┘
                                ····· § 14.4
cntab2 ····· § 14.4
convlv ── realft ── four1 ····· § 13.1
copy ····· § 19.6
correl ── realft ── four1 ····· § 13.2
cosft1 ── realft ── four1 ····· § 12.3
cosft2 ── realft ── four1 ····· § 12.3
covsrt ····· § 15.4
crank ····· § 14.6
cyclic ── tridag ····· § 2.7
daub4 ····· § 13.10
dawson ····· § 6.10
dbrent ── [func]
       └─ [dfunc]
                                ····· § 10.3
ddpoly ····· § 5.3
decchk ····· § 20.3
dfldim ── [dfunc] ····· § 10.6
dfpmin ┬─ [func]
       ├─ [dfunc]
       └─ lnsrch ── [func]
                                ····· § 10.7
dfridr ── [func] ····· § 5.7
dftcor ····· § 13.9
dftint ┬─ [func]
       ├─ realft ── four1
       ├─ polint
       └─ dftcor
                                ····· § 13.9
difeq ····· § 17.4
dlinmin ── mnbrak ── fldim ····· § 10.6
       │
       └─ dbrent ── dfldim ── [func]
eclass ····· § 8.6
eclazz ── [equiv] ····· § 8.6
ei ····· § 6.3
eigsrt ····· § 11.1
elle ┬─ rf
     └─ rd
                                ····· § 6.11
```

ellf — rf ····· § 6.11

ellpi — rf ····· § 6.11
- rj — rc
- *rf*

elmhes ····· § 11.5

erff — gammp — gser — gammln
- gcf — gammln ····· § 6.2

erffc — gammp — gser — gammln
- gcf — gammln ····· § 6.2
- gammq — *gser* — *gammln*
- *gcf* — *gammln*

erfcc ····· § 6.2

eulsum ····· § 5.1

evlmem ····· § 13.7

expdev — ran1 ····· § 7.2

expint ····· § 6.3

f1dim — [func] ····· § 10.5

factln — gammln ····· § 6.1

factrl — gammln ····· § 6.1

fasper — avevar ····· § 13.8
- spread
- realft — four1

fdjac — [funcv] ····· § 9.7

fgauss ····· § 15.5

fit — gammq — gser — gammln
- gcf — gammln ····· § 15.2

fitexy — avevar ····· § 15.3
- fit — gammq — gser — gammln
- gcf — gammln
- chixy
- mnbrak
- brent
- *gammq* — *gser* — *gammln*
- *gcf* — *gammln*
- zbrent — *chixy*

fixrts — zroots — laguer ····· § 13.6

fleg ····· § 15.4

flmoon ····· § 1.0

fmin — [funcv] ····· § 9.7

four1 ····· § 12.2

fourew ····· § 12.6

fourfs — fourew ····· § 12.6

fourn ····· § 12.4

fpoly ····· § 15.4

fred2 — gauleg ..... § 18.1
- [ak]
- [g]
- ludcmp
- lubksb

fredex — quadmx — wwghts — kermom ..... § 18.3
- ludcmp
- lubksb

fredin — [ak] ..... § 18.1
- [g]

frenel ..... § 6.9

frprmn — [func] ..... § 10.6
- [dfunc]
- linmin — mnbrak
  - brent — f1dim — [func]

ftest — avevar ..... § 14.2
- betai — gammln
  - betacf

gamdev — ran1 ..... § 7.3

gammln ..... § 6.1

gammp — gser ..... § 6.2
- gcf — gammln

gammq — gser ..... § 6.2
- gcf — gammln

gasdev — ran1 ..... § 7.2

gauccof — tqli — pythag ..... § 4.5
- eigsrt

gauher ..... § 4.5

gaujac — gammln ..... § 4.5

gaulag — gammln ..... § 4.5

gauleg ..... § 4.5

gaussj ..... § 2.1

gcf — gammln ..... § 6.2

golden — [func] ..... § 10.1

gser — gammln ..... § 6.2

hpsel — sort ..... § 8.5

hpsort ..... § 8.3

hqr ..... § 11.6

hufapp ..... § 20.4

hufdec ..... § 20.4

hufenc ..... § 20.4

hufmak — hufapp ..... § 20.4

hunt ..... § 3.4

hypdrv …… § 6.12

hypgeo ─┬─ hypser …… § 6.12
  │ odeint ┬ bsstep ─ mmid
  │ │ └─ pzextr
  │ └─ hypdrv

hypser …… § 6.12

icrc ── icrc1 …… § 20.3

icrc1 …… § 20.3

igray …… § 20.2

iindexx …… indexx 的长整型版,见该项

indexx …… § 8.4

interp …… § 19.6

irbit1 …… § 7.4

irbit2 …… § 7.4

jacobi …… § 11.1

jacobn …… § 16.6

julday …… § 1.1

kendl1 ── erfcc …… § 14.6

kendl2 ── erfcc …… § 14.6

kermom …… § 18.3

ks2d1s ─┬─ quadct …… § 14.7
  ├─ quadvl
  ├─ pearsn ── betai ─┬─ gammln
  │ └─ betacf
  └─ probks

ks2d2s ─┬─ quadct …… § 14.7
  ├─ pearsn ─ betai ─┬─ gammln
  │ └─ betacf
  └─ probks

ksone ─┬─ sort …… § 14.3
  ├─ [func]
  └─ probks

kstwo ─┬─ sort …… § 14.3
  └─ probks

laguer …… § 9.5

lfit ─┬─ [funcs] …… § 15.4
  ├─ gaussj
  └─ covsrt

linbcg ─┬─ atimes …… § 2.7
  ├─ snrm
  └─ asolve

linmin ─┬─ mnbrak ─ …… § 10.5
  └─ brent ── f1dim ─ [func]

lnsrch —— [func] …… § 9.7
locate …… § 3.4
lop …… § 19.6
lubksb …… § 2.3
ludcmp …… § 2.3
machar …… § 20.1
matadd …… § 19.6
matsub …… § 19.6
medfit —— rofunc —— select …… § 15.7
memcof …… § 13.6
metrop —— ran3 …… § 10.9
mgfas ┬ rstrct …… § 19.6
│ ├ slvsm2 — fill0
│ ├ interp
│ └ copy
├ relax2
├ lop
├ matsub
├ anorm2
└ matadd

mglin ┬ rstrct …… § 19.6
│ ├ slvsml —— fill0
│ ├ interp
│ └ copy
├ relax
├ resid
├ *fill0*
└ addint —— *interp*

midinf —— [func] …… § 4.4
midpnt —— [func] …… § 4.4
miser ┬ ranpt —— ran1 …… § 7.8
└ [func]

mmid —— [derivs] …… § 16.3
mnbrak —— [func] …… § 10.1
mnewt ┬ [usrfun] …… § 9.6
├ ludcmp
└ lubksb

moment …… § 14.1
mp2dfr —— mpops …… § 20.6
mpdiv ┬ mpinv ┬ mpmul —— drealft —— dfour1 …… § 20.6
│ │ └ mpops
│ ├ *mpmul* —— *drealft* —— *dfour1*
└ mpops

```
mpinv ─┬─ mpmul ── drealft ── dfour1 ····· § 20.6
       └─ mpops
mpmul ── drealft ── dfour1 ····· § 20.6
mpops ····· § 20.6
mppi ─┬─ mpsqrt ─┬─ mpmul ── drealft ── dfour1 ····· § 20.6
      │          └─ mpops
      ├─ mpops
      ├─ mpmul ── drealft ── dfour1
      ├─ mpinv ── mpmul ── drealft ── dfour1
      └─ mp2dfr ── mpops
mprove ── lubksb ····· § 2.5
mpsqrt ─┬─ mpmul ── drealft ── dfour1 ····· § 20.6
        └─ mpops
mrqcof ── [funcs] ····· § 15.5
mrqmin ─┬─ mrqcof ── [funcs] ····· § 15.5
        ├─ gaussj
        └─ covsrt
newt ─┬─ fmin ─┬─ [funcv] ····· § 9.7
      │ fdjac ─┘
      └─ ludcmp
      ├─ lubksb
      └─ lnsrch ── fmin ── [funcv]
odeint ─┬─ [derivs] ····· § 16.2
        └─ rkqs ─┬─ [derivs]
                 └─ rkck ── [derivs]
orthog ····· § 4.5
pade ─┬─ ludcmp ····· § 5.12
      ├─ lubksb
      └─ mprove ── lubksb
pccheb ····· § 5.11
pcshft ····· § 5.10
pearsn ── betai ─┬─ gammln ····· § 14.5
                 └─ betacf
period ── avevar ····· § 13.8
piksr2 ····· § 8.1
piksrt ····· § 8.1
pinvs ····· § 17.3
plgndr ····· § 6.8
poidev ─┬─ ran1 ····· § 7.3
        └─ gammln
polcoe ····· § 3.5
polcof ── polint ····· § 3.5
```

poldiv …… § 5.3
polin2 —— polint …… § 3.6
polint …… § 3.1
powell [func] …… § 10.5
— linmin —— mnbrak —
— brent — — f1dim —— [func]
predic …… § 13.6
probks …… § 14.3
psdes …… § 7.5
pwt …… § 13.10
pwtset …… § 13.10
pythag …… § 2.6
pzextr …… § 16.4
qgaus —— [func] …… § 4.5
qrdcmp …… § 2.10
qromb —— trapzd —— [func] …… § 4.3
— polint
qromo —— midpnt —— [func] …… § 4.4
— polint
qroot —— poldiv …… § 9.5
qrsolv —— rsolv …… § 2.10
qrupdt —— rotate …… § 2.10
qsimp —— trapzd —— [func] …… § 4.2
qtrap —— trapzd —— [func] …… § 4.2
quad3d —— qgaus — [func] …… § 4.6
[y1]
— [y2]
— [z1]
— [z2]
quadct …… § 14.7
quadmx —— wwghts —— kermom …… § 18.3
quadvl …… § 14.7
ran0 …… § 7.1
ran1 …… § 7.1
ran2 …… § 7.1
ran3 …… § 7.1
ran4 —— psdes …… § 7.5
rank …… § 8.4
ranpt —— ran1 …… § 7.8
ratint …… § 3.2
ratlsq —— [fn] …… § 5.13
— dsvdcmp —— dpythag
— dsvbksb
— ratval

ratval …… § 5.3
rc …… § 6.11
rd …… § 6.11
realft —— four1 …… § 6.11
rebin …… § 12.3
red …… § 7.8
relax …… § 17.3
relax2 …… § 19.6
resid …… § 19.6
revcst …… § 19.6
reverse …… § 10.9
rf …… § 10.9
rj — rc …… § 6.11
: rf …… § 6.11
rk4 —— [derivs] …… § 16.1
rkck —— [derivs] …… § 16.2
rkdumb — [derivs] …… § 16.1
— rk4 — [derivs]
rkqs —— rkck —— [derivs] …… § 16.2
rlft3 —— fourn …… § 12.5
rofunc —— select …… § 15.7
rotate …… § 2.10
rsolv …… § 2.10
rstrct …… § 19.6
rtbis —— [func] …… § 9.1
rtflsp —— [func] …… § 9.2
rtnewt —— [funcd] …… § 9.4
rtsafe —— [funcd] …… § 9.4
rtsec —— [func] …… § 9.2
rzextr …… § 16.4
savgol — ludcmp …… § 14.8
| — lubksb
scrsho —— [func] …… § 9.0
select …… § 8.5
selip —— shell …… § 8.5
sfroid — plgndr …… § 17.4
: solvde — difeq
— pinvs
— red
bksub
shell …… § 8.1
shoot — [load] …… § 17.1
— odeint —— [derivs]
└ rkqs — rkck — [derivs]
└ [score]

shootf — [load1] ····· §17.2
odeint — [derivs]
rkqs — rkck — [derivs]
[score]
[load2]

simp1 ····· §10.8
simp2 ····· §10.8
simp3 ····· §10.8
simplx — simp1 ····· §10.8
simp2
simp3

simpr — ludcmp ····· §16.6
lubksb
[derivs]

sinft — realft — four1 ····· §12.3
slvsm2 ····· §19.6
slvsml ····· §19.6
sncndn ····· §6.11
snrm ····· §2.7
sobseq ····· §7.7
solvde — difeq ····· §17.3
pinvs
red
bksub

sor ····· §19.5
sort ····· §8.2
sort2 ····· §8.2
sort3 — indexx ····· §8.4
spctrm — four1 ····· §13.4
spear — sort2 ····· §14.6
crank
erfcc
betai — gammln
betacf

sphbes — bessjy — beschb — chebev ····· §6.7
sphfpt — newt — fdjac — shootf(q, v) ····· §17.4
lnsrch
fmin — *shootf*(q, v)
ludcmp
lubksb

```
sphoot ── newt ── fdjac ── shoot (q, v) ..... § 17.4
                  │ lnsrch ─┐
                  └ fmin ── shoot (q, v)
                  │ ludcmp
                  │ lubksb
splie2 ── spline ..... § 3.6
splin2    splint ..... § 3.6
          spline
spline ..... § 3.3
splint ..... § 3.3
spread ..... § 13.8
sprsax ..... § 2.7
sprsin ..... § 2.7
sprspm ..... § 2.7
sprstm ..... § 2.7
sprstp ── indexx ..... § 2.7
sprstx ..... § 2.7
stifbs ── jacobn ..... § 16.6
       ── simpr ── ludcmp
                  ── lubksb
                  ── [derivs]
       ── pzextr
stiff     jacobn ..... § 16.6
          ludcmp
          lubksb
          [derivs]
stoerm ── [derivs] ..... § 16.5
svbksb ..... § 2.6
svdcmp ── pythag ..... § 2.6
svdfit ── [funcs] ..... § 15.4
       ── svdcmp ── pythag
          svbksb
svdvar ..... § 15.4
toeplz ..... § 2.8
tptest ── avevar ..... § 14.2
          betai ── gammln
                ── betacf
tqli ── pythag ..... § 11.3
trapzd ── [func] ..... § 4.2
tred2 ..... § 11.2
tridag ..... § 2.4
trncst ..... § 10.9
trnspt ..... § 10.9
```

```
ttest ──┬── avevar ..................................... § 14.2
        └── betai ──┬── gammln
                    └── betacf

tutest ──┬── avevar ..................................... § 14.2
         └── betai ──┬── gammln
                     └── betacf

twofft ── four1 ...................................... § 12.3
vander ...............................................  § 2.8
vegas ──┬── rebin ..................................... § 7.8
        │   ran2
        └── [fxn]

voltra ──┬── [g_ ....................................... § 18.2
         ├── [_ak]
         ├── ludcmp
         └── lubksb

wt1 ── daub4 ......................................... § 13.10
wtn ── daub4 ......................................... § 13.10
wwghts ── kermom ..................................... § 18.3
zbrac ── [func] ...................................... § 9.1
zbrak ── [func] ...................................... § 9.1
zbrent ── [func] ..................................... § 9.3
zrhqr ──┬── balanc ................................... § 9.5
        └── hqr
zriddr ── [func] ..................................... § 9.2
zroots ── laguer ..................................... § 9.5
```

各章节的计算机程序

1.0	flmoon	按日期计算月相
1.1	julday	从历书日期计算儒略历日数
1.1	badluk	13日正好是星期五、且又为满月
1.1	caldat	从儒略历日数计算历书日期
2.1	gaussj	Gauss-Jordan矩阵求逆和线性方程求解
2.3	ludcmp	线性方程求解，LU分解
2.3	lubksb	线性方程求解，回代过程
2.4	tridag	三对角方程组求解
2.4	banmul	带状对角矩阵与向量相乘
2.4	bandec	带状对角方程组，分解
2.4	banbks	带状对角方程组，回代
2.5	mprove	线性方程求解，迭代改进
2.6	svbksb	奇异值回代过程
2.6	svdcmp	矩阵奇异值分解
2.6	pythag	计算$(a^2+b^2)^{1/2}$，不发生溢出
2.7	cyclic	循环三对角方程组求解
2.7	sprsin	把矩阵转换成稀疏方式
2.7	sprsax	稀疏矩阵和向量的乘积
2.7	sprstx	稀疏矩阵转置和向量的乘积
2.7	sprstp	稀疏矩阵的转置
2.7	sprspm	两个稀疏矩阵的模式乘
2.7	sprstm	两个稀疏矩阵的阈值乘
2.7	linbcg	稀疏线性方程的双共轭梯度法求解
2.7	snrm	计算向量模，用于程序linbcg
2.7	atimes	计算稀疏矩阵的乘积，用于程序linbcg
2.7	asolve	预条件器，用于程序linbcg
2.8	vander	求解Vandermonde方程组
2.8	toeplz	求解Toeplitz方程组
2.9	choldc	Cholesky分解
2.9	cholsl	Cholesky回代
2.10	qrdcmp	QR分解
2.10	qrsolv	QR回代
2.10	rsolv	右三角回代
2.10	qrupdt	校正QR分解
2.10	rotate	用于程序qrupdt的雅可比旋转
3.1	polint	多项式插值
3.2	ratint	有理函数插值
3.3	spline	构造三次样条
3.3	splint	三次样条插值
3.4	locate	用二分法搜索有序表
3.4	hunt	当调用是相关时搜索有序表
3.5	polcoe	从列表值求多项式系数
3.5	polcof	从列表值求多项式系数
3.6	polin2	二维多项式插值
3.6	bcucol	构造二维双三次式
3.6	bcuint	二维双三次插值
3.6	splie2	构造二维样条
3.6	splin2	二维样条插值
4.2	trapzd	梯形法则
4.2	qtrap	用梯形法则求积分
4.2	qsimp	用Simpson法则求积分
4.4	qromb	用龙贝格自适应法求积分
4.2	midpnt	扩展中心点法则
4.2	qromo	用开型龙贝格自适应法求积分
4.2	midinf	在半无穷区间上对函数积分
4.4	midsql	对具有下限平方根奇异点的函数积分
4.4	midsqu	对具有上限平方根奇异点的函数积分
4.4	midexp	对呈指数衰减的函数积分
4.5	qgaus	高斯求积法对函数积分
4.5	gauleg	高斯－勒让德的权和横坐标
4.5	gaulag	高斯－拉盖尔的权和横坐标
4.5	gauher	高斯－埃尔米特的权和横坐标
4.5	gaujac	高斯－雅可比的权和横坐标
4.5	gaucof	从正交多项式求积分的权
4.5	orthog	构造非经典正交多项式
4.6	quad3d	在三维空间上对函数求积分
5.1	eulsum	Euler-van Wijngaarden算法求级数和
5.3	ddpoly	计算多项式和它的导数
5.3	poldiv	用一多项式除另一多项式
5.3	ratval	有理函数求值
5.7	dfridr	Ridders方法的数值求导
5.8	chebft	用切比雪夫多项式拟合一个函数
5.8	chebev	切比雪夫多项式求值
5.9	chder	对已用切比雪夫多项式拟合的函数求导
5.9	chint	对已用切比雪夫多项式拟合的函数求积
5.10	chebpc	切比雪夫拟合的多项式系数
5.10	pcshft	位移多项式的多项式系数

5.11	pccheb	程序 chebpc 的逆过程,用于幂级数经济化
5.12	pade	从幂级数系数作帕德逼近
5.13	ratlsq	最小二乘法的有理多项式拟合
6.1	gammln	Γ函数的对数
6.1	factrl	阶乘函数
6.1	bico	二项式系数函数
6.1	factln	阶乘函数的对数
6.1	beta	B 函数
6.2	gammp	不完全 Γ 函数
6.2	gammq	互补不完全 Γ 函数
6.2	gser	级数,用于程序gammp和程序gammq
6.2	gcf	连分式,用于程序gammp和程序gammq
6.2	erff	误差函数
6.2	erffc	互补误差函数
6.2	erfcc	互补误差函数,简洁的程序
6.3	expint	指数积分 E_n
6.3	ei	指数积分 Ei
6.4	betai	不完全 B 函数
6.4	betacf	连分式,用于程序 betai
6.5	bessj0	贝塞尔函数 J_0
6.5	bessy0	贝塞尔函数 Y_0
6.5	bessj1	贝塞尔函数 J_1
6.5	bessy1	贝塞尔函数 Y_1
6.5	bessy	整数阶贝塞尔函数 Y
6.5	bessj	整数阶贝塞尔函数 J
6.6	bessi0	修正的贝塞尔函数 I_0
6.6	bessk0	修正的贝塞尔函数 K_0
6.6	bessi1	修正的贝塞尔函数 I_1
6.6	bessk1	修正的贝塞尔函数 K_1
6.6	bessk	修正的整数阶贝塞尔函数 K
6.6	bessi	修正的整数阶贝塞尔函数 I
6.7	bessjy	分数阶贝塞尔函数
6.7	beschb	用于程序 bessjy 的切比雪夫展开式
6.7	bessik	修正的分数阶贝塞尔函数
6.7	airy	艾里函数
6.7	sphbes	球面贝塞尔函数 j_n 和 y_n
6.8	plgndr	(与球面调和函数)连带的勒让德多项式
6.9	frcnel	Fresnel 积分 $S(s)$和 $C(x)$
6.9	cisi	余弦和正弦积分 Ci 和 Si
6.10	dawson	Dawson 积分
6.11	rf	第一类 Carlson 椭圆积分
6.11	rd	第二类 Carlson 椭圆积分
6.11	rj	第三类 Carlson 椭圆积分
6.11	rc	Carlson 退化椭圆积分
6.11	ellf	第一类勒让德椭圆积分
6.11	elle	第二类勒让德椭圆积分
6.11	ellpi	第三类勒让德椭圆积分
6.11	sncndn	雅可比椭圆函数
6.12	hypgeo	复超几何函数
6.12	hypser	复超几何函数,级数求值
6.12	hypdrv	复超几何函数的导数
7.1	ran0	用 Park 和 Miller 最低标准生成随机偏离
7.1	ran1	随机偏离,最低标准加混洗
7.1	ran2	随机偏离,L'Ecuyer 长周期加混洗
7.1	ran3	随机偏离,Knuth 相减法
7.2	expdev	指数随机偏离
7.2	gasdev	正态分布的随机偏离
7.3	gamdev	Γ 分布的随机偏离
7.3	poidev	泊松分布的随机偏离
7.3	bnldev	二项分布的随机偏离
7.4	irbit1	随机位序列
7.4	irbit2	随机位序列
7.5	psdes	64 位混洗的"伪数据加密标准"
7.5	ran4	使用数据加密标准混洗生成的随机偏离
7.7	sobseq	Sobols'拟随机序列
7.8	vegas	自适应多维蒙特卡罗积分
7.8	rebin	用于程序 vegas 中取样重新分级
7.8	miser	递归多维蒙特卡罗积分
7.8	ranpt	获取随机点,用于miser程序
8.1	piksrt	用直接插入法对数组排序
8.1	piksr2	用直接插入法对两数组排序
8.1	shell	用 Shell 法对数组排序
8.2	sort	用快速排序法对数组排序
8.2	sort2	用快速排序法对两个数组排序
8.3	hpsort	用堆积排序法对数组排序
8.4	indexx	构造一个数组的索引表
8.4	sort3	排序,使用索引表对 3 个或多个数组排序
8.4	rank	构造数组的秩表
8.5	select	寻找数组中第 N 大的元素
8.5	selip	找出数组中第 N 大的元素,而不变动原数组
8.5	hpsel	找出前 M 个大值,而不变动原数组
8.6	eclass	从表确定等价类

8.6	eclazz	由过程确定等价类
9.0	scrsho	画函数图形以搜索根
9.1	zbrac	为求根向外搜索划界
9.1	zbrak	为求根向内搜索划界
9.1	rtbis	用二分法寻找函数的根
9.2	rtflsp	用试位法寻找函数的根
9.2	rtsec	用弦截法寻找函数的根
9.2	zriddr	用 Ridders 方法寻找函数的根
9.3	zbrent	用 Brent 方法寻找函数的根
9.4	rtnewt	用 Newton-Raphson 算法寻找函数的根
9.4	rtsafe	用 Newton-Raphson 和二分法寻找函数的根
9.5	laguer	用拉盖尔方法找多项式的根
9.5	zroots	用具有降阶的拉盖尔方法求多项式的根
9.5	zrhqr	特征值法求多项式的根
9.5	qroot	多项式的复根或重根，Bairstow
9.6	mnewt	牛顿法解方程组
9.7	lnsrch	用于程序 newt 的沿直线搜索
9.7	newt	全局收敛的多维牛顿法
9.7	fdjac	用于程序 newt 的有限差分雅可比行列式
9.7	fmin	用于程序 newt 的向量函数的范数
9.7	broydn	方程组的弦截法
10.1	mnbrak	函数最小值的划界
10.1	golden	用黄金分割搜索法寻找函数极小值
10.2	brent	用 Brent 方法寻找函数极小值
10.3	dbrent	利用导数信息寻找函数极小值
10.4	amoeba	用下降单纯形法求 N 维极小化
10.4	amotry	求试探点，用于程序 amoeba
10.5	powell	用 Powell 的方法求 N 维极小化
10.5	linmin	沿 N 维空间一条射线求函数极小值
10.5	f1dim	用于程序 linmin 中的函数
10.6	frprmn	用共轭梯度法求 N 维极小化
10.6	dlinmin	利用导数信息沿一条射线求函数极小值
10.6	df1dim	用于程序 dlinmin 中的函数
10.7	dfpmin	用变尺度法求 N 维极小化
10.8	simplx	线性函数的线性规划极大化
10.8	simp1	用于程序 simplx 中的线性规划
10.8	simp2	用于程序 simplx 中的线性规划
10.8	simp3	用于程序 simplx 中的线性规划
10.9	anneal	用模拟退火法求解旅行销售员问题
10.9	revcst	反转的代价，用于程序anneal
10.9	reverse	执行反转过程，用于程序anneal
10.9	trncst	传送的代价，用于程序anneal
10.9	trnspt	执行传送，用于程序 anneal
10.9	metrop	Metropolis 算法，用于程序anneal
10.9	amebsa	连续空间中的模拟退火法
10.9	amotsa	计算一个试验点，用于程序amebsa
11.1	jacobi	对称矩阵的特征值和特征向量
11.1	eigsrt	特征向量，按特征值顺序而排序
11.2	tred2	实对称矩阵的 Householder 约化
11.3	tqli	对称三对角矩阵的特征解
11.5	balanc	配平一个非对称矩阵
11.5	elmhes	将一般矩阵约化成 Hessenberg 型式
11.6	hqr	Hessenberg 矩阵的特征值
12.2	four1	一维块速傅里叶变换(FFT)
12.3	twofft	两个实函数的快速傅里叶变换
12.3	realft	单个实函数的快速傅里叶变换
12.3	sinft	快速正弦变换
12.3	cosft1	带端点的快速余弦变换
12.3	cosft2	"交错"快速余弦变换
12.4	fourn	多维快速傅里叶变换
12.5	rlft3	二维或三维实数据的 FFT
12.6	fourfs	存储的外部介质的大数据集的 FFT
12.6	fourew	用于程序 fourfs 的重绕和置换文件
13.1	convlv	使用 FFT 作数据的卷积和解卷积
13.2	correl	使用 FTT 作数据的相关和自相关
13.4	spctrm	使用 FFT 作功率谱估计
13.6	memcof	对最大熵(MEM)系数求值
13.6	fixrts	反射多项式的根到单位圆内
13.6	predic	使用 MEM 系数的线性预测
13.7	evlmem	由 MEM 系数的功率谱估计
13.8	period	非均匀取样数据的功率谱
13.8	fasper	非均匀取样大数据集的功率谱
13.8	spread	对数组进行倒插值，用于程序fasper
13.9	dftcor	为傅里叶积分计算端点校正
13.9	dftint	高精度的傅里叶积分
13.10	wt1	一维离散小波变换
13.10	daub4	DAUb4 系数的小波滤波器
13.10	pwtset	为程序 pwt 初始化系数
13.10	pwt	部分小波变换
13.10	wtn	多维离散小波变换
14.1	moment	计算数据集的矩
14.2	ttest	均值差的学生 t 检验
14.2	avevar	计算数据集的均值和方差
14.2	tutest	均值的学生 t 检验，方差不相等的情况
14.2	tptest	均值的学生 t 检验，配对数据情况
14.2	ftest	对方差之差的 F 检验

14.3	chsone	数据和模型之间差异的χ^2检验
14.3	chstwo	两数据集之间差异的χ^2检验
14.3	ksone	数据相对模型的K-S检验
14.3	kstwo	两数据集之间的K-S检验
14.3	probks	K-S概率函数
14.4	cntab1	使用χ^2的列联表分析
14.4	cntab2	使用熵度量的列联表分析
14.5	pearsn	两数据集之间的Pearson相关
14.6	spear	两数据集之间的Spearman秩相关
14.6	crank	用数组的秩替换数组元素
14.6	kendl1	两数据集之间的相关,Kendall τ
14.6	kendl2	使用Kendall τ的列联表分析
14.7	ks2d1s	数据对模型的二维K-S检验
14.7	quadct	计算各象限中的点,用于程序ks2d1s
14.7	quadvl	象限的概率,用于程序ks2d1s
14.7	ks2d2s	数据对数据的二维K-S检验
14.8	savgol	S-G平滑系数
15.2	fit	最小二乘法将数据拟合成直线
15.3	fitexy	拟合数据成直线,x和y两者都有误差
15.3	chixy	计算χ^2,用于程序fitexy
15.4	lfit	用正规方程的一般线性最小二乘拟合
15.4	covsrt	重排协方差矩阵,用于程序lfit
15.4	svdfit	用奇异值分解法的线性最小二乘拟合
15.4	svdvar	来自奇异值分解的方差
15.4	fpoly	用程序lfit或svdfit拟合多项式
15.4	fleg	用程序lfit或svdfit拟合勒让德多项式
15.5	mrqmin	Marquardt方法,非线性最小二乘拟合
15.5	mrqcof	用于程序mrqmin中计算系数
15.5	fgauss	使用程序mrqmin拟合高斯分布的和
15.7	medfit	稳健地将数据拟合成直线,最小绝对偏差
15.7	rofunc	用于程序medfit中稳健地拟合数据
16.1	rk4	用四阶Runge-Kutta法计算ODE的一步积分
16.1	rkdumb	用四阶Runge-Kutta法对ODE积分求解
16.2	rkqs	具有精确度监控的ODE的一步积分
16.2	rkck	用于程序rkqs的Cash-Karp-Runge-Kutta步
16.2	odeint	具有精确度监控的ODE积分
16.3	mmid	用修正中点法的ODE积分
16.4	bsstep	用Bulirsch-Stoer步积分ODE
16.4	pzextr	用于程序bsstep的多项式外推
16.4	rzextr	用于程序bsstep的有理函数外推
16.5	stoerm	积分二阶守恒ODE
16.6	stiff	用四阶Rosenbrock积分刚性的ODE
16.6	jacobn	用于程序stiff的求雅可比行列式程序的范例
16.6	derivs	用于程序stiff的求导数程序的范例
16.6	simpr	用半隐式中点法积分刚性的ODE
16.6	stifbs	用Bulirsch-Stoer积分刚性ODE
17.1	shoot	打靶法求解两点边界值问题
17.2	shootf	对拟合点打靶求解两点边界值问题
17.3	solvde	用松驰法求解两点边界值问题
17.3	bksub	回代,用于程序solvde
17.3	pinvs	对角化子块,用于程序solvde
17.3	red	约化矩阵的列,用于程序slovde
17.4	sfroid	用程序solvde计算球体函数
17.4	difeq	球体矩阵系数,用于程序sfroid
17.4	sphoot	用程序shoot计算球体函数
17.4	sphfpt	用程序shootf计算球体函数
18.1	fred2	求解第二类线性Fredholm方程
18.1	fredin	和程序fred2一起得到内插解
18.2	voltra	第二类线性Volterra方程
18.3	wwghts	任意奇异核的积分权
18.3	kermom	对于一个奇异核的矩的样本程序
18.3	quadmx	积分矩阵的样本程序
18.3	fredex	求解奇异Fredholm方程的范例
19.5	sor	用逐次超松驰法求解椭圆PDE
19.6	mglin	用多重网格法求解线性椭圆PDE
19.6	rstrct	半加权限制,用于程序mglin和mgfas
19.6	interp	双线性拓展,用于程序mglin和mgfas
19.6	addint	插值和相加,用于程序mglin
19.6	slvsm1	最粗网格上求解,用于程序mglin
19.6	relax	Gauss-Seidel松驰,用于程序mglin
19.6	resid	计算残差,用于程序mglin
19.6	copy	用于程序mglin和mgfas的实用程序
19.6	fill0	用于程序mglin的实用程序
19.6	mgfas	用多重网格法求解非线性椭圆PDE
19.6	relax2	Gauss-Seidel松驰,用于程序mgfas
19.6	slvsm2	最粗网格上求解,用于程序mgfas
19.6	lop	应用非线性算子,用于程序mgfas
19.6	matadd	用于程序mgfas的实用程序
19.6	matsub	用于程序mgfas的实用程序
19.6	anorm2	用于程序mgfas的实用程序
20.1	machar	诊断计算机的浮点运算
20.2	igray	格雷码和它的逆

20.3	icrc1	用于程序 icrc 的循环冗余度校验和式
20.3	icrc	循环冗余度校验和式
20.3	decchk	十进制校验位的计算或证实
20.4	hufmak	构造一个霍夫曼码
20.4	hufapp	追加位到霍夫曼码，用于hufmak
20.4	hufenc	用霍夫曼码编码和压缩一个字符
20.4	hufdec	用霍夫曼码译码和解压缩一个字符
20.5	arcmak	构造一个算术码
20.5	arcode	用算术码编码或译码一个字符
20.5	arcsum	将整数加入到字节串中，用于程序 arcode
20.6	mpops	多精度算术运算，简单的运算
20.6	mpmul	多精度乘法，采用 FFT 方法
20.6	mpinv	多精度的倒数
20.6	mpdiv	多精度的除法和余数
20.6	mpsqrt	多精度的平方根
20.6	mp2dfr	多精度转换成十进制数
20.6	mppi	多精度的示例，计算 π 的多位数字