

Maple 手册

0.1	Maple 的工作环境.....	10
0.1.1	可执行块.....	10
0.1.2	表格.....	11
0.1.3	段落和文本.....	11
0.1.4	小节.....	12
0.1.1	超链接.....	12
0.2	输入输出方式.....	13
0.3	在线帮助系统.....	15
0.4	命令行工作环境.....	15
1.1	Maple 中的整数和有理数运算.....	19
1.1.1	Maple 命令的输入格式.....	19
1.1.2	整数和有理数计算.....	20
1.2	无理数、复数和浮点运算.....	22
1.2.1	无理数和浮点数.....	22
1.2.2	代数数 (Algebraic Numbers)	23
1.2.3	复数运算.....	24
1.3	Maple 中的变量和常量.....	25
1.3.1	赋值.....	25
1.3.2	变量名.....	27
1.3.3	基本数据类型.....	29
1.3.4	Maple 中的常数.....	31
1.4	函数和表达式.....	32
1.4.1	Maple 中的数学函数.....	32
1.4.2	多项式.....	32
1.4.3	有理式.....	36
1.4.4	有理式的转换.....	36
1.5	内部数据结构和变量代换.....	37
1.5.1	多项式和分式的内部表示法.....	37
1.5.1	分式的内部数据结构.....	40
1.5.2	变量代换.....	42
2.1	极限和连续性.....	47
2.1.1	函数或表达式的极限.....	47
2.1.2	函数的连续性.....	48
2.2	Maple 中的求导和微分运算.....	49
2.2.1	符号表达式求导.....	49
2.2.2	隐函数的导数.....	51

2.3	积分运算.....	52
2.3.1	不定积分.....	52
2.3.2	定积分.....	53
2.3.3	数值积分.....	54
2.3.4	重积分和线积分.....	55
2.3.5	利用辅助手段积分.....	56
2.4	级数.....	59
2.4.1	数值级数和函数项级数求和.....	59
2.4.2	幂级数.....	59
2.4.3	泰勒级数和劳朗级数.....	61
2.4.4	切比雪夫级数和渐进级数.....	64
2.5	积分变换.....	65
2.5.1	拉普拉斯变换.....	65
2.5.2	富利叶变换.....	66
2.5.3	快速富利叶变换.....	67
2.5.4	其他积分变换.....	69
3.1	序列.....	71
3.2	集合.....	73
3.3	有序表.....	75
3.4	数组.....	77
3.5	类型转换和元素运算.....	80
4.1	矩阵的基本运算.....	83
4.2	矩阵求值.....	85
4.3	矩阵和线性方程组.....	87
4.3.1	矩阵基本运算.....	88
4.3.2	分块矩阵.....	88
4.3.3	初等变换.....	89
4.3.4	线性方程组的解.....	91
4.3.5	正定矩阵.....	93
4.3.6	特殊矩阵.....	93
4.4	线性空间基本理论.....	94
4.4.1	基本子空间.....	94
4.4.2	正交基和 Schmidt 正交化.....	94
4.4.3	线性方程组的最小二乘解.....	96
4.5	特征值、特征向量和相似标准型.....	96
4.5.1	矩阵的相似.....	96
4.5.2	特征值和特征向量.....	97
5.1	箭头操作符.....	100
5.2	最简单的子程序.....	101

5.3	局部变量和全局变量	103
5.4	基本程序结构	104
5.4.1	for 循环	104
5.4.2	分支结构	105
5.4.3	while 循环	107
5.5	递归子程序	108
5.6	子程序中的求值	110
5.6.1	参数	110
5.6.2	局部变量	112
5.6.3	全局变量	112
5.6.4	特例	112
5.7	嵌套子程序	114
5.8	记忆表	114
5.8.1	remember 选项	114
5.8.2	在记忆表中加入项	115
5.8.3	在记忆表中删除项	115
6.1	二维基本图形绘制	118
6.2	plot 函数的可选参数	120
6.3	二维图形对象的结构	125
6.4	特殊二维图形的绘制	128
6.4.1	参数曲线的绘制	128
6.4.2	极坐标下的绘图	129
6.4.3	平面代数曲线的绘制	130
6.4.4	对数坐标下的绘图	131
6.4.5	共形映射的图形绘制	132
6.5	二维绘图的注意事项	133
6.5.1	图形走样	133
6.5.2	常见的错误	133
6.6	基本三维图形的绘制	134
6.7	三维绘图的选项	135
6.7.1	style 选项	135
6.7.2	着色选项	136
6.7.3	坐标轴选项	136
6.7.4	空间朝向和投影	136
6.7.5	透视投影	137
6.7.6	网格大小	137
6.7.7	观察区域	138
6.7.8	光照模型	138
6.8	三维图形对象的结构	139

6.9	特殊三维图形的绘制	140
6.9.1	空间参数曲线和参数曲面	140
6.9.2	球坐标和柱坐标下的绘图	141
6.9.3	管状图形的绘制	142
6.9.4	等高线图绘制	143
6.9.5	隐式曲面的绘制	144
6.9.6	多面体的绘制	144
6.10	图形动画的制作	145
7.1	代数方程	148
7.1.1	单未知数的方程	148
7.1.2	solve 函数的缩略形式	149
7.1.3	一些困难	150
7.1.4	方程组的解	151
7.2	其他求解工具	152
7.2.1	数值求解	152
7.2.2	求方程的整数解	153
7.2.3	\mathbf{Z}_n 中的方程求解	155
7.2.4	递归方程的求解	155
7.3	常微分方程的求解	156
7.3.1	常微分方程的解析解	156
7.3.2	利用积分变换求解微分方程	158
7.3.3	常微分方程组的求解	160
7.4	常微分方程的级数解法	161
7.4.1	泰勒级数法	161
7.4.2	幂级数解法	162
7.5	常微分方程数值解法	164
7.5.1	变步长龙格库塔法	164
7.5.2	刚性方程和吉尔法	166
7.5.3	经典数值方法	167
7.6	非线性常微分方程的扰动法	168
7.6.1	庞加莱法 (Poincaré-Lindstedt Method)	168
7.6.2	多尺度法	173
7.7	偏微分方程简介	179
7.7.1	偏微分方程解析解	179
7.7.2	偏微分方程的形式转换	182
7.7.3	偏微分方程解的图形绘制	183
7.7.4	李对称 (Lie Symmetry) 工具包	186
8.1	返回子程序的子程序	190
8.1.1	牛顿迭代法	190

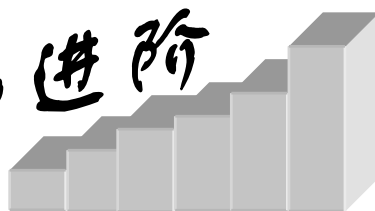
8.1.2	函数的平移	191
8.2	局部变量的保持	192
8.2.1	“越界”的局部变量	193
8.2.2	集合的笛卡尔积	194
8.3	交互式输入	197
8.3.1	从终端读入字符串	197
8.3.2	从终端读入表达式	197
8.3.3	把字符串转化为表达式	199
8.4	扩展 Maple 命令	199
8.4.1	自定义数据类型	200
8.4.2	自定义操作符	200
8.4.3	扩展 Maple 命令	203
8.5	编写自己的工具包	205
10.1.1	工具包的结构	205
10.1.2	工具包的初始化	207
10.1.3	建立自己的程序库	208
9.1	输入输出的例子	212
9.2	文件类型和打开方式	214
9.2.1	有缓冲文件和无缓冲文件	214
9.2.2	文本文件和二进制文件	214
9.2.3	读模式和写模式	215
9.2.4	默认文件 default 和终端文件 terminal	215
9.2.5	文件名和文件描述符	215
9.3	文件控制命令	216
9.3.1	文件的打开和关闭	216
9.3.2	查询和设定文件的当前位置	217
9.3.3	检测文件尾	217
9.3.4	检测文件状态	217
9.3.5	删除文件	218
9.4	输入命令	218
9.4.1	从文件中读入文本	218
9.4.2	从文件中读入任意多字节	219
9.4.3	格式化输入	219
9.4.4	读入 Maple 语句	222
9.4.5	读入表格式数据	222
9.5	输出命令	222
9.5.1	利用 interface 命令设置输出参数	222
9.5.2	一维表达式输出	223
9.5.3	二维表达式输出	223

9.5.4	输出 Maple 字符串	224
9.5.5	向文件输出任意多字节	224
9.5.6	格式化输出	224
9.5.7	输出表格式数据	226
9.5.8	写透文件缓冲	227
9.5.9	默认输出流的重定向	227
9.6	转换命令	228
9.6.1	C 语言、Fortran 语言代码生成	228
9.6.2	生成 LATEX 或 eqn	230
9.6.3	字符串和整数之间的转换	230
9.6.4	从字符串中获得 Maple 表达式	231
9.6.5	对于字符串的格式化输入和输出	231
9.7	调用 Matlab 函数	232
10.1	调试的例子	234
10.2	使用 Maple 的调试器	239
10.2.1	显示程序的语句	240
10.2.2	断点	240
10.2.3	显式的断点	241
10.2.4	监视断点	242
10.2.5	出错断点	242
10.3	系统状态的检查和设置	244
10.3.1	变量值的显示和修改	244
10.3.2	程序运行状态的察看	245
10.3.3	显示断点信息	247
10.3.4	程序执行的控制	248
11.1	平面几何对象	251
11.1.1	点、线段和直线	251
11.1.2	三角形、正方形和圆	253
11.1.3	二次曲线	255
11.1.4	正多边形对象	258
11.2	平面几何对象的相互关系	259
11.2.1	点和直线的位置关系	259
11.2.2	与三角形相关的函数	261
11.2.3	与圆相关的函数	262
11.2.4	其他函数	265
11.3	平面上的变换	265
11.3.1	正交变换	265
11.3.2	其他类型的变换	267
11.4	空间几何对象	268

11.4.1	点、线、面	268
11.4.2	球和多面体	270
11.5	空间几何对象的关系	271
11.5.1	有关点的函数	271
11.5.2	有关直线和平面的函数	272
11.5.3	与球相关的函数	274
11.6	三维空间中的几何变换	275
11.6.1	空间几何对象的变换	275
11.6.2	几何变换的谓词运算	278
12.1	图论工具包 networks	281
12.1.1	图对象及其建立	281
12.1.2	有关图的基本概念	284
12.1.3	图的连通性	288
12.1.4	树	290
12.1.5	图的矩阵表示	292
12.1.6	平面图的判断	292
12.1.7	图的着色	293
12.2	布尔运算和数理逻辑	293
12.2.1	基本的布尔运算	293
12.2.2	其他逻辑函数	294
12.3	群论	296
12.3.1	群的表示	296
12.3.2	有关群的概念	297
12.4	组合数学	299
12.4.1	组合数学的基本数据结构	299
12.4.2	组合结构元素的获取	300
12.4.3	组合类	300
12.4.4	生成函数	304
12.4.5	combinat 工具包	305
12.4.6	Stirling 数和拉丁方	307
13.1	张量数据类型	310
13.1.1	张量数据类型及其建立	310
13.1.2	度量张量的输入	311
13.2	张量的代数运算	313
13.2.1	张量的比较	313
13.2.2	升降指标	314
13.2.3	张量的线性组合	315
13.2.4	张量的内积、外积和缩并	316
13.2.5	张量的转置	317

13.2.6	张量的对称化和反对称化.....	318
13.3	张量场函数的导数.....	319
13.3.1	张量分量对坐标的偏导数.....	319
13.3.2	Christoffel 符号.....	320
13.3.3	张量分量对坐标的协变导数.....	322
13.3.4	标量场的方向导数.....	323
13.3.5	Riemann-Christoffel 张量.....	323
13.4	坐标变换.....	324
13.4.1	坐标变换的 Jacobi 矩阵.....	324
13.4.2	张量的坐标变换.....	325
13.5	张量对象的信息.....	326
13.5.1	张量信息的获取.....	326
13.5.2	张量的指标函数.....	326

起步与进阶



第

操作界面

0

章

本章将 Windows 95/98 下的 Maple V Release 5 为例，简要地介绍 Maple 软件的操作界面。Maple V Release 5 有窗口和命令行两个工作环境，这里将主要介绍窗口工作环境的使用；由于考虑到不同用户的需求，也会对命令行工作环境作附带性的介绍。

本章具体包括以下内容：

- 🕒 如何在 Maple 中输入命令和其他辅助信息
- 🕒 如何在设置 Maple 的输入输出格式
- 🕒 如何在 Maple 中设置和使用各种排版格式
- 🕒 如何打印 Maple 的文档
- 🕒 如何使用 Maple 的在线帮助系统

当一群来自世界不同角落的人聚在一起时，他们将无法谈论社会问题或者哲学问题，但是他们人可以毫无障碍的探讨数学问题，这就是数学语言的魅力。我们每个人从小就接触数学语言，小学乃至初中的数学教师，也不止一次的叮嘱我们注意数学表述的规范性，因为数学语言也许是唯一可以被称作“世界语”的语言吧。

Maple，经历了从 Maple V Release 1 到 Maple V Release 5 的不断改进，从纯文本命令行操作，到图形化的窗口界面，直到现在的超文本界面，越来越贴近我们所耳熟能详的数学语言。直观，漂亮的输出格式，方便的排版功能，能使你的专业报告条理更清晰。

这一章里将主要介绍 Maple 的用户环境，如果您以前接触过 Maple 软件，或者您急于用 Maple 来解决实际问题，您完全可以跳过这一部分内容，直接进入后面的学习。这一章与后面的内容没有直接的联系。

0.1 Maple 的工作环境

Maple V Release 5 的文档包括以下几种形式：可执行块、表格、段落、小节、超链接。可执行块和表格是用户与 Maple 的计算引擎间交互的纽带，是用户进行计算解决问题，显示输入与输出最直接的方式。而段落、小节和超链接则可以方便用户整理计算结果。下面将对这些格式逐一进行介绍。

0.1.1 可执行块

可执行块是 Maple 工作环境中基本的计算元素，它的主要功能是把一个或者多个 Maple 命令组成一个可以重复执行的单元。在 Maple 窗口环境中，你可以很容易地找到可执行块的所在，它的左边是一个大的方括弧 “[” 和一个命令提示符 “>”，参见图 0.1。

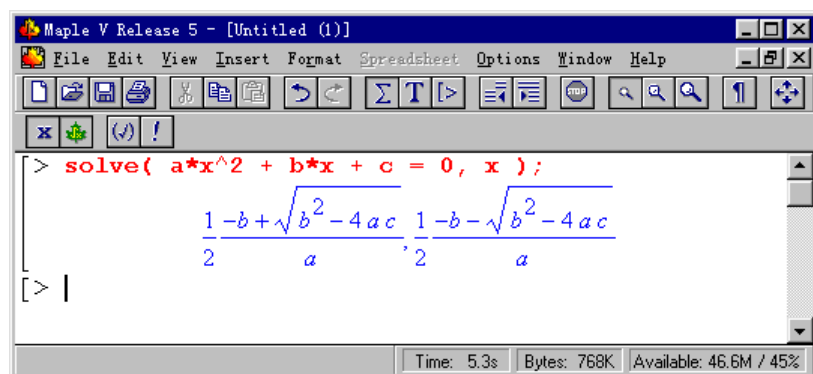


图 0.1 可执行块

图 0.1 中命令提示符所在行中，以分号 “;” 结尾的部分就是一个 Maple 命令，由于数学语言的简明易懂，聪明的读者朋友一定已经看出来它的用途了。是的，solve 命令是用来解代数方程的，图中所示是一个标准的一元二次方程，关于 solve 等命令的具体用法我们在后面章节中会详细介绍。同样，也很容易看出，在同一个可执行块中，位于命令下面的文字，是 Maple 输出的结果。

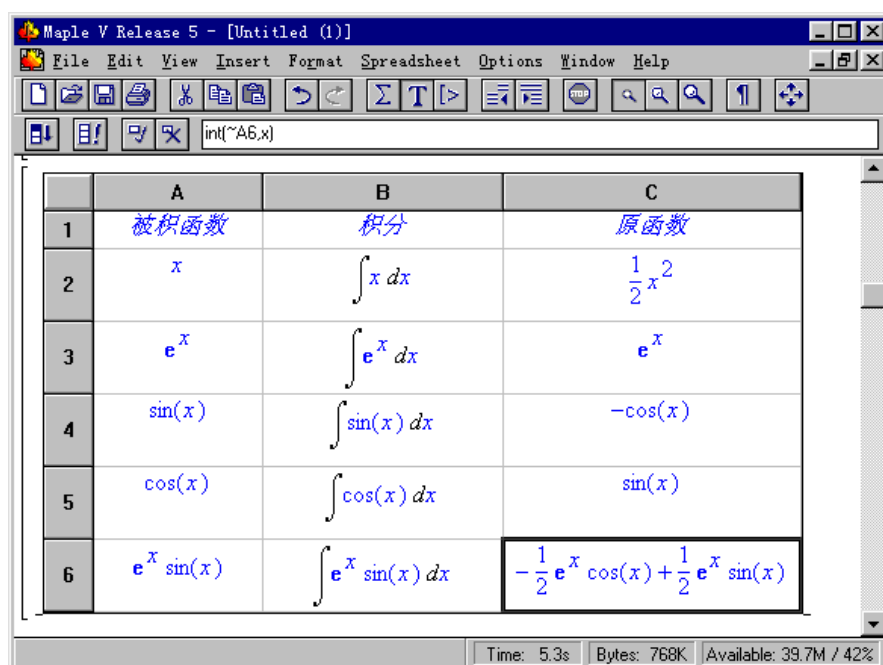
Maple 的可执行块是可以重复执行的，这就是说如果用户第一次的输入有误，或者用户

需要改变一些参数重新计算，用户可以按动键盘上的光标键，或者用鼠标直接定位，将当前光标位置移动到前面的命令上，直接进行修改，按回车键，光标所在的可执行块中的所有命令就会被再次执行。

启动 Maple 环境时，默认的状态就是可执行块。用户也可以从菜单 Insert | Execution Group 中选择 Before Cursor（或者 After Cursor），在当前光标所在行的前面（或者后面）插入一个可执行块。

0.1.2 表格

Maple V Release 5 的工作区中可以加入表格，这是一种把符号计算功能融入到有如 Microsoft Excel 的传统电子表格的功能。需要说明的是，此项功能目前只在 Maple 的 Windows 95、Windows NT 和 Macintosh 版本中有。用户可以利用 Maple 的表格工具，生成公式表。图 0.2 就是一张用 Maple 生成的简单积分公式表，表中第二、三列的表达式由第一列中的表达式决定。



	A	B	C
1	被积函数	积分	原函数
2	x	$\int x dx$	$\frac{1}{2}x^2$
3	e^x	$\int e^x dx$	e^x
4	$\sin(x)$	$\int \sin(x) dx$	$-\cos(x)$
5	$\cos(x)$	$\int \cos(x) dx$	$\sin(x)$
6	$e^x \sin(x)$	$\int e^x \sin(x) dx$	$-\frac{1}{2}e^x \cos(x) + \frac{1}{2}e^x \sin(x)$

图 0.2 表格

用户要在工作区插入表格，可以选择菜单项 Insert | Spread Sheet。然后，将光标移到表各种相应位置，就可以输入或修改单元格的内容了。如果要引用表中单元格的内容，可以使用以波浪线“~”开头的名称，比如~A6 就表示第 A 列第 6 行的单元格中的内容。和可执行块一样，表格可以在工作区的任意位置插入。



0.1.3 段落和文本

Maple 中的段落与大家所熟悉的 Microsoft Word 中的段落非常相似。在一个段落中，可以包含有格式化的文本、数学表达式、图形（包括由 Maple 中的绘图命令生成的图形）等等。而一个段落本身又可以被包含在一个可执行块中。和 Word 中一样，在段落中，你可以让文本居中、居左、或是居右，对应地选择 Format 菜单下的 Center、Left Justify、或是 Right Justify

即可；也可以为文本选择斜体、粗体、或是下划线的修饰，只需选择 **Format** 菜单下的 **Italic**、**Bold** 或 **Underline**。段落中的文本在默认情况下采用的是 **Times New Roman** 字体。当然，你也可以方便的改变字体，和字体的大小。上面的这些段落排版功能在工具栏上都有快捷工具按钮，其用途与外观都与通用文字处理软件（如 **Word** 等）相类似，由于这些软件已经非常普及，在这里就不再对 **Maple** 的工具栏详细介绍了。

选择菜单 **Insert | Paragraph**，就可以在当前位置插入一个段落。**Maple V Release 5** 支持行间的运算，这就是说，你可以在文本中间加入一个可执行块，并且运行它，得到结果。**Maple V Release 5** 还支持拖曳操作，你可以用鼠标方便地将任何一个对象、一段文本选中并拖动到所需要的地方。所见即所得，一切都是那么自然。

0.1.4 小节

为了使你的计算报告更加条理化，结构层次更加分明，**Maple V Release 5** 提供了一个方便的文本格式——小节。在每一个小节的开头，都有一个图标。用鼠标左键单击该图标，小节就可以被打开——显示小节中所有的内容，或是被关闭——只显示小节的标题。在小节关闭时，其图标为；而当小节被打开时，其图标变成。

小节之间相互可以嵌套，通过使用这一格式，你就可以轻松地编写出重点突出，同时条理分明的计算报告来了。

选择菜单 **Insert | Section**，就可以在当前位置插入一个小节。

0.1.1 超链接

人类业已步入 21 世纪，户连网络已经家喻户晓，可以毫不夸张地说，只要是使用过微机的人，就一定接触过网页，也一定为网页中方便的超链接所吸引。超链接是一段文字或者一个图片，当你用鼠标点击它时，浏览器就会自动转到另一处文档或者同一文档的不同位置。由于它的直观性很容易为人所接受，因此，越来越多的文本编辑软件将它纳入到自己的格式中。**Maple V Release 5** 也毫不落伍，你可以在 **Maple** 文当中方便地加入超链接。只要选择菜单 **Insert | HyperLink...**，就会弹出一个对话框（参见图 0.3）；所需做的只是填妥对话框中的内容，然后按 **OK** 按钮这么简单。

在对话框上部，**Link Text** 对应的编辑框中，填写超链接文本。接下来，在下方的选择按钮上选择是链接到文档（**Work Sheet**）还是链接到 **Maple** 的帮助（**Help Topic**）。如果选择链接到文档，就在右边的编辑框中填入所链接的文档文件名，也可以按 **Browse** 按钮在已有的文件中选择；如果选择链接到帮助，就填入帮助的关键字。当然，也可以指定链接到文档的特定位置，可以在下方的下拉选框中选择该特定位置对应的书签（**Book Mark**）。如果链接到本文档中的位置，可以不填写文档名称，默认值就是当前文档。

书签是 **Maple** 文档中特定位置的标志，用来方便阅读时的定位。在菜单 **View | Book Marks** 下选择一个书签，就可以转到该书签所对应的位置。

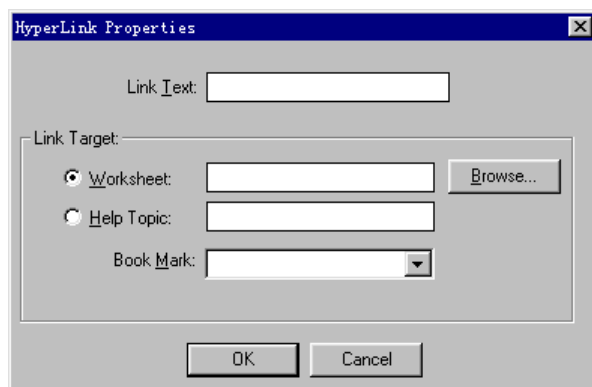


图 0.3 插入超链接对话框

不仅在文档中可以加入超链接，Maple V Release 5 还支持将文档存成超文本格式（HTML），只需选择菜单 File | Save As...，再选择 HTML source 格式即可；或者可以直接选择菜单 File | Export As...。

前面一一介绍了 Maple 文档中几种文本格式，也许你会觉得它们对于解决数学问题毫无必要，而且使用不方便——那就把它们抛开吧，因为 Maple 毕竟不是一个文本编辑器，我们完全有理由把眼光集中到它的主要功能上。

我们在这里没有提供 Maple 多格式文本的例子，因为在 Maple 的帮助系统中，到处都是这样的例子，Maple 的整个帮助系统都是用这种方式生成的。

0.2 输入输出方式

为了满足不同用户的需要，Maple 中可以更换输入输出的格式。从菜单 Option | Input Display 和 Output Display 下，可以选择所需的输入/输出格式。

Maple V Release 5 有两种不同的输入格式：Maple 语言（Maple Notation）和标准数学表示法（Standard Math）。Maple 有一套完善的内建语言，不亚于任何一种高级编程语言，它的语法和 Pascal 或者 C 在一定程度上有些类似，但还是有很大差别的，在阅读了本书之后，读者朋友一定会掌握它的。标准数学表示法就是我们常用的数学语言，这是 Maple 为了方便初学者的设计，用户不用掌握 Maple 的严格语法规则，Maple 自动予以补充，比如乘号可以省略不写等等。但对于熟练使用者来说，这种方法输入速度较慢，不太适用。本书中将统一使用 Maple 语言输入，因为这种输入方式在 Maple 的所有版本中都是适用的。

Maple V Release 5 有 4 种不同的输出格式：Maple 语言、格式化文本（Character Notation）、固定格式记法（Typeset Notation）、可编辑数学记法（Editable Math）。Maple 语言与输入时所用的 Maple 语言完全相同。格式化文本和我们平时所用的数学表示法很相似，只不过它是纯文本的，在一些需要纯文本结果的情况下（比如用于 BBS 上发表的结果）可以使用。固定格式记法和可编辑数学记法在外观上完全一样，只不过可编辑记法可以选择、拷贝、或者拖曳结果的一部分，而固定格式记法只能对整体作这些操作。图 0.4 是几种不同格式的输出结果。在窗口工作环境下，建议读者选用可编辑数学记法作为输出方式，本书中后面的例子

也都是采用可编辑数学记法的。

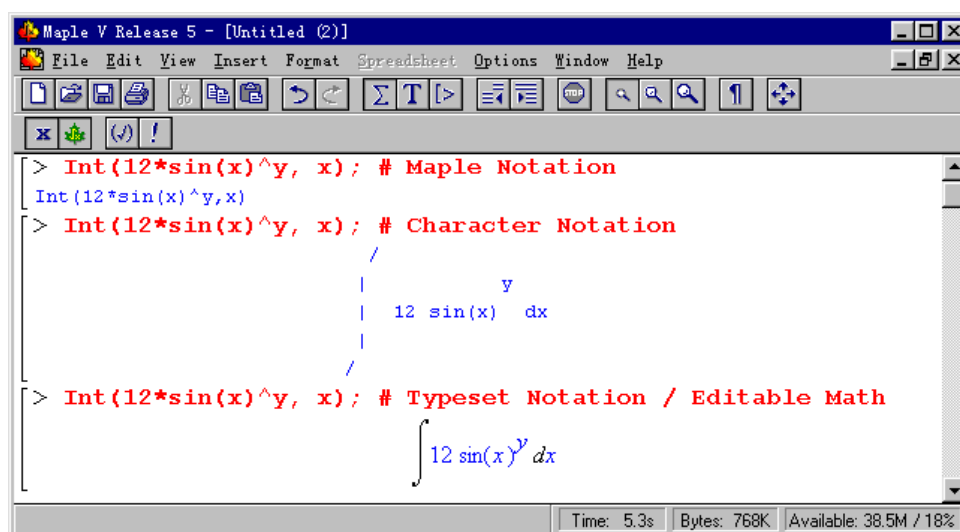


图 0.4 几种不同格式的输出结果

有一点需要说明的是，这些输出方式只在表观上有不同，其内容是相同的，如果拷贝输出结果或者结果的一部分粘贴到输入上，Maple 会自动将其转换成对应的输入格式，使用起来是很方便的。

Maple 会认识一些输入的变量名称，比如我们在数学中经常使用的希腊字母等等。在菜单 Options | Assumed Variables 中有三个选项：No Annotation、Trailing Tildes、Phrase。默认选项是 Trailing Tildes，它会将以这些变量名开头的变量名的相应部分替换成 Maple 的漂亮字体，比如 delta12 会显示成 $\delta 12$ ；如果选择 Phrase，则 Maple 只替换整个的变量；若选择 No Annotation，则 Maple 不作任何替换。为了方便读者输入希腊字母，这里将它们的名称列成一张表，供读者参考。

表 0.1 希腊字母名称表

字母	名称	字母	名称	字母	名称
α	alpha	ι	iota	ρ	rho
β	beta	κ	kappa	σ	sigma
γ	gamma	λ	lambda	τ	tau
δ	delta	μ	mu	υ	upsilon
ϵ	epsilon	ν	nu	ϕ	phi
ζ	zeta	ξ	xi	χ	chi
η	eta	\omicron	omicron	ψ	psi
θ	theta	π	pi	ω	omega

表中仅列出了小写字母，如果要输入大写字母，只需把名称的首字母大写即可。

0.3 在线帮助系统

Maple V Release 5 具有一个使用十分方便的在线帮助系统。选择菜单 Help | Introduction 就可以打开这个帮助系统。图 0.5 就是帮助系统的外观。

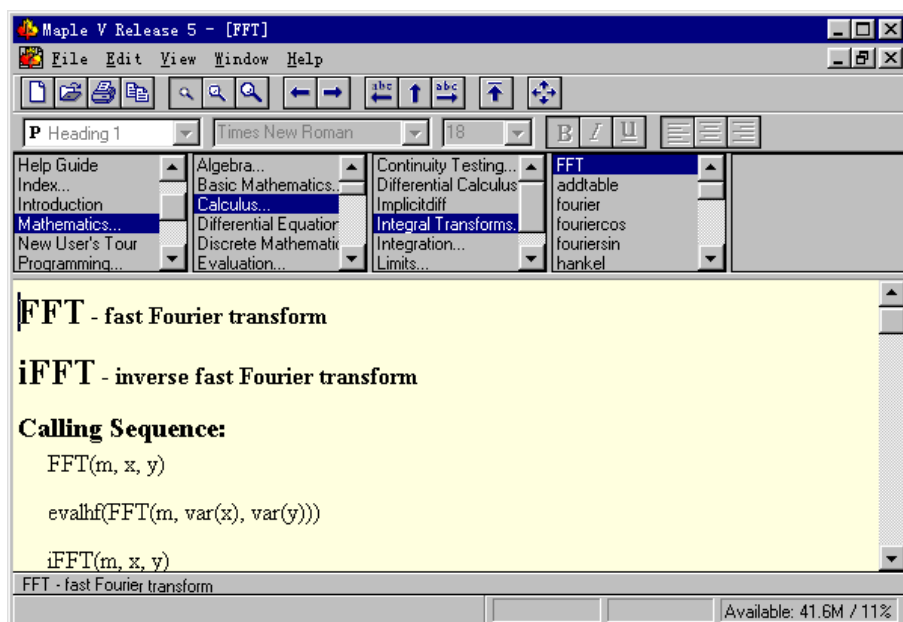


图 0.5 Maple 的在线帮助系统

帮助窗口的上半部分是分类的索引表，从左至右选择索引中的项目，就可以找到帮助主题。分类是依据使用和功能的，十分详尽，只要用户对数学术语熟悉，就可以快速的找到帮助的内容。这以帮助形式在 Maple V Release 4 中曾一度被取消，也许是软件编写者发现这种帮助形式确实必不可少，在新版本中又出现了。

Maple 也支持上下文相关的帮助，用户只要降光标移至所需查找的关键字上，按 F1 键，就可以获得有关该关键字的帮助。

0.4 命令行工作环境

在有些特殊情况下，比如机器资源有限，而计算工作庞大时，我们需要将尽可能多的机器资源用到计算上，这时，我们可以使用 Maple V Release 5 的命令行工作环境。命令行工作环境是一个纯文本界面的环境，一个类似于 Maple V 以前的 Dos 版本的工作环境，Maple 所有的函数和子程序（包括绘图）在命令行环境下均可以使用。所不同的是，命令行工作环境的输入输出方式都是文本格式的，这将节省一定的机器资源。由于是文本界面，命令行环境中不能进行格式文档的编辑，只是用来完成计算任务的。

在 Maple V Release 5 程序组中选择 Command Line Maple 运行，就可以启动命令行环境。

顾名思义，命令行中命令执行以行为单位，输入一行（即键入一次回车），Maple 计算引擎就工作一次。当然，一行中可以包括多个命令，须以命令结束符——分号或冒号间隔开，最后也必须以命令结束符结尾。

和窗口环境一样，命令行环境中也可以存储和读入工作任务进程，用“save 文件名”和“read 文件名”就可以完成存储和读入；与窗口环境不同，命令行环境是单任务环境，也就是说同时只能进行一个工作任务。输入 quit 命令，就可以退出命令行工作环境。

命令行环境中也有文本格式的帮助系统，使用时输入“? 关键字”就可以查到相同或相关的帮助资料。

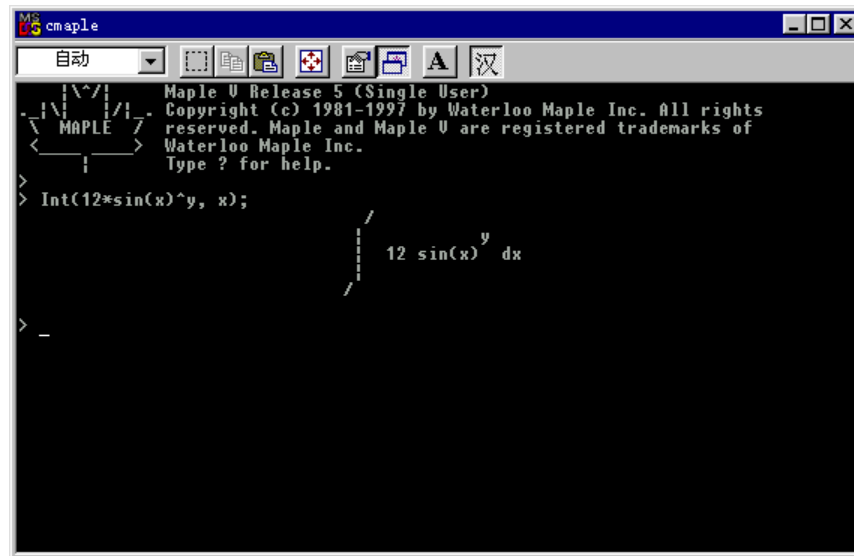
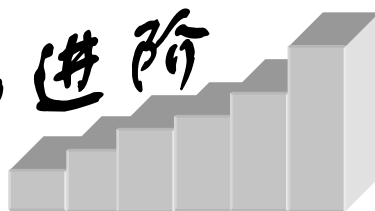


图 0.6 Maple V Release 5 命令行工作环境

起步与进阶



第

基本代数运算

一

章

本章将简要地介绍 Maple 软件中的基本代数运算，读者可以借此熟悉 Maple 软件中的变量、常量、基本代数运算符和一些常用的命令。

本章具体包括以下内容：

- 🕒 Maple 的命令格式
- 🕒 如何在 Maple 中进行整数和有理数的初等运算
- 🕒 Maple 中的整数、有理数的内部数据结构
- 🕒 如何在 Maple 中进行无理数、复数和浮点数运算
- 🕒 如何在 Maple 中进行符号代数运算和化简
- 🕒 Maple 中多项式的内部数据结构
- 🕒 Maple 中分式的内部数据结构

读者朋友一定对袖珍计算器非常熟悉，在这一章里，您将看到把 Maple 作为一个计算器来使用的一些例子。当然，这不是一个一般的计算器，而可以称其为一个超级计算器，它可已精确地计算非常巨大的整数，可以用分数精确地表示有理数计算结果，可以用字母符号代替数字进行运算……；而同时，一切操作又是那么地简单，正如使用你手中的计算器一样。也许你原来对 Maple 有点陌生，而觉得它有点高深莫测，经过这一章的学习，相信你一定会喜欢上这个工具的。

说了这么多，也许你已经等得不耐烦了，好了，让我们马上进入到 Maple 的神奇世界里去吧！

1.1 Maple 中的整数和有理数运算

1.1.1 Maple 命令的输入格式

启动 Maple V Release 5，你会看到新建文档中的 “[>]” 提示符，这是 Maple 中可执行块的标志（参见第 0 章），你可以在 “[>]” 的后面键入你的命令。比如：

```
[> 9 + 1000 / 5^3 * 3;
                                     33
> 8! / (5!+10);
                                     4032
                                     13
```

和在其他常用的高级编程语言中一样，Maple 中有着你所熟悉的运算符：加号 (+)、减号 (-)、乘号 (*)、除号 (/)、指数运算符 (^或**); Maple 更有编程语言所没有的运算符，比如阶乘 (!)，运算符的优先级和初等数学中规定的一样，当然，需要改变运算顺序时，或者为了表达得更清楚，你可以使用圆括弧 “(” 和 “)”。

正如你所看到的，Maple 的使用和计算器极为相似，但又有所不同。这里需要指出不同点来。其一，Maple 中的每一指令都必须由一个分号 “;” 或是冒号 “:” 作为结束标志。千万别忘了打结束标志，否则，系统是不会对这句命令做任何运算的。分号表示命令结束，并且提示系统在计算完毕时显示计算结果；而冒号表示让系统只计算，而不显示计算结果，这往往用在中间计算步骤上，或者你不需要知道计算结果的时候。

其二，Maple 只在你输入行结束符（键盘上的回车）时才进行运算。这并不是说 Maple 的每一行只能输入一句命令，而是在一个完整的可执行块（以结束标志结尾）上键入回车之后，Maple 将执行当前可执行块中所有命令（可能不止一句）。如果可执行块不完整，Maple 将给出如下的提示：

```
[> 1
>
Warning, premature end of input
```

如果要输入的命令很长，不能在一行输完，你完全可以换一行继续输入（此时换行用

Shift+Enter，就不会出现命令未完的警告了），而在最后一行加上命令结束标志；不过需要注意的是，并不是命令的任何一部分都可以拆开输入的，比如，你不能把数字 1000000 从中间断开，而成为 1 000 000；如果你必须将这样的部分分行输入时，你可以在分割处行末尾加上反斜杠“\”，这个符号不是 Maple 的运算符，仅仅表示“此处未完，下行继续”。

这实际上是一件非常简单而且好理解的事，相信你在看了下面的例子后就会完全掌握 Maple 的命令输入格式的。

```
> 4*5; 200/4; 3^5; 1234
;
20
243
1234

> 1\
234\
56789;
123456789
```

1.1.2 整数和有理数计算

作为一个符号代数系统，Maple 中可以绝对避免算术运算的舍入误差。和计算器不同，Maple 从来不自作主张地把算术表达式近似成浮点数。在 Maple 中，两个有公因数的整数的商只会被化简，也即消去其分子和分母的最大公因数。为了能够准确的表示整数和有理数，一个符号代数系统必须具有处理大整数的能力，Maple 便如此。

```
> 3246863/21314349837918478802821336554244217897
+12321321465437812093221894/49464878716123213;
262620956179795903911552862956772845423948171513503962870767737
1054311729645658016863353740801439618660239673071743061

> big_number := 4^(4^4);
big_number := 134078079299425970995740249982058461274793658205923\
933777235614437217640300735469768018742981669034276900318581864\
86050853753882811946569946433649006084096

> length(big_number);
155
```

上面例子中，使用了一个变量 big_number，并用“:=”将其赋值。和 Pascal 语言一样，Maple 中为一个变量赋值用的是“:=”。上面例子中还用到了一个函数 length()，这个函数用在整数上时，返回的是该整数的十进制位数，也就是数字的长度。有关变量和函数的详细使用方法，我们会在随后的章节中学习。

Maple 中可以表达的整数范围是相当大的，但它也有一个上限，在 Maple 中，一个整数的十进制位数的上限是 $2^{19} - 9 = 524279$ 。很奇怪，是吧？这个上限实际上是由 Maple 中的整数存储的数据结构所决定的。在 Maple 中，一个整数 $i_0 + i_1B + i_2B^2 + i_3B^3 + \cdots + i_nB^n$ ，是用一个静态表存储的，如下图所示：

整数	符号	i_0	i_1	
----	----	-------	-------	-------	--

其中， B 是 10 的整数次幂，满足 B^2 可以被机器用单精度整数表示的条件（以保证即使做乘法也不发生溢出）；这样，在 32 位计算机中， $B = 10^4$ 。在静态表的头节点中，除了该整数的正负号外，还保存有该表长度的信息，Maple 用 17 个二进制位表示静态表的长度，这就是那个“奇怪”的数的由来： $2^{19} - 9 = 4((2^{17} - 1) - 1) - 1$ 。

不过你尽可以放心，Maple 足够聪明，它自己有能力分辨一个整数是否可以表示，当整数太大而不能表示时，会给出出错信息：

```
> 123456789^987654321;
Error, object too large
```

Maple 中还有许多整数运算的函数和子程序, 作为例子, 我们来计算一个大数的质因数。首先判断该数是否为质数:

```
> big_number := 10^29 - 10^14 - 1;
      big_number = 99999999999999899999999999999
> isprime(big_number);
      false
```

`isprime()` 是一个布尔返回值的函数，用来判断一个整数是否为素数。接下来，我们来求这个数的质因子。因数分解是一个相当费时的计算过程，我们用 Maple 的系统函数 `time()` 来获取计算过程中所耗费的时间。我们将这三个命令打在同一个可执行块中（同一行，中间不要加回车），以使其连续执行。

```
> settime:=time(): ifactor(big_number);
  caltime:=(time()-settime)*seconds;
      (61) (223) (13166701) (97660768252549) (5717)
              caltime := 2.857 seconds
```

由于计算环境不同，例子中的时间很可能与你的运行结果不同，但无论如何，分解而得的质因数一定是相同的。

```
[> nextprime(big_number);  
9999999999999900000000000157  
> isqrt(big_number);  
316227766016838
```

函数 `nextprime()` 求得的是比参数 `big_number` 大的下一个素数。而函数 `isqrt()` 求得的是和 `big_number` 的平方根最近的整数。

Maple 中也可以进行带余数的除法, 如:

```
[> a := 1234: b := 56:
> q := iquo(a, b); #整数商      q := 22
> r := irem(a, b); #余数      r := 2
```

有了大整数的表示法，理论上讲，一切有理数也都可以精确的表示了，让我们来简单地看一下 Maple 中有理数（分数）的存储结构吧。在 Maple 中，每个分数结构中都存有 两个指针，分别指向分子和分母（大整数）。由于大整数的存储很费空间，所以在 Maple 中所

有的整数至多只存储一次。例如，Maple 工作空间中有三个分数 $-\frac{1}{2}$ ， $\frac{2}{3}$ 和 $\frac{3}{5}$ ，在系统内部，存储结构如图 1.1 所示。

关于 Maple 的内部机制，本书会在相应的地方给出一些定性的描述，因为这些内容对于应用 Maple 没有多大的关系，只是作一个一般性介绍。如果读者具有数据结构方面的基础知识，通过阅读这些内容，会对 Maple 的内部机制有一定的了解；如果你觉得这些内容没有必要，完全可以跳过去不看。

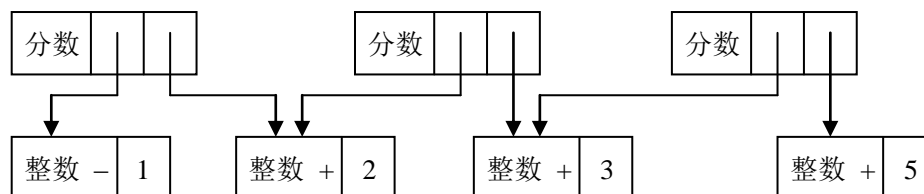


图 1.1 分数的内部存储结构

1.2 无理数、复数和浮点运算

1.2.1 无理数和浮点数

在介绍 Maple 的浮点运算之前，我们先来看一个例子：

```
> 8^(1/6);
      8 (1/6)
> simplify(%);
      √2
```

噢，这么显然的式子，Maple 怎么不自动进行化简呢？先别急着下结论，Maple 可是非常严格的哦。这样的化简，可不是在任何时候都能成立的呀，特别是在复数域中，如果一个无理式是一个中间步骤，如果进行这样的化简，很有可能造成失根。为了避免在这方面出现漏洞，Maple 对无理式不自动化简，除非用户人为地要求化简，比如使用 `simplify()` 函数强迫 Maple 对式子进行化简。

提示：Maple 中可以用百分号“%”代替上一计算的结果，自然而然地，两个连续百分号“%%”表示前一次计算的结果；三个则表示再前一次计算的结果。

下面再来看看浮点运算。接着前面的例子，我们将结果化成浮点形式：

```
> evalf(%%);
      1.414213562
> convert(%%, 'float');
      1.414213562
```

利用 `evalf()` 函数，我们可以把表达式化成浮点形式。而 `convert()` 函数是用来改变表达式类型的通用函数，`convert(exp, 'float')` 起到的效果与 `evalf(exp)` 完全一样。引入了浮点数，自然就引入了舍入误差，“精确”计算也就不复存在了：

```
[> %^6;
                                     7.999999987
```

好在 Maple 中可以设定浮点数的有效位数。`Digits` 是一个内部变量，表示当前的浮点有效位数，你可以通过对 `Digits` 赋值来改变浮点运算的有效位数。默认情况下，`Digits = 10`。

```
[> Digits := 20: evalf(sqrt(2));
                                     1.4142135623730950488
```

在 Maple 中，一个带小数点“.”的数字将被自动地处理为浮点数；而且，和 C、Fortran 等高级语言一样，Maple 可以自动地完成从整数到浮点数的类型转换：

```
[> 100023*0.22;
                                     22005.06
```

除了一般的表示法以外，在 Maple 中还支持另外两种浮点数的表示法。比如浮点数 0.000001，也可以写作 `1E-6`，或者 `Float(1, -6)`。后一种写法与浮点数在 Maple 内部的存储方式很相近，有三部分：结构的头（float 标志）、尾数（1）、指数（-6）。由于浮点数的尾数部分是一个整数，所以它也一样受上一节所讲的最大整数的限制。也就是说，Maple 中浮点数的有效位数最多只能到 $2^{19} - 9 = 524279$ 。

由于用上述结构实现的浮点数运算速度很慢，而有时对浮点数的精度要求不高，而要求有较快的速度（比如画图过程中），这时可以使用机器浮点运算函数——`evalhf()` 来达到目的。机器浮点函数 `evalhf()` 的使用与 `evalf()` 相同，只是其精度不受 `Digits` 变量的影响，一般在 12~16 位有效数字之间。

1.2.2 代数数 (Algebraic Numbers)

正如我们在上一段开头时所看到的，Maple 在处理整数的方根时会遇到一些问题。在这里，我们将更深入地考察这一类问题。

如果对实分析有所了解的话，你一定知道整数的方根是所谓的“代数数”的特例。代数数，一个不太严格的定义是，整系数单变量多项式的实根。其范围较有理数要大，又包含在实数域中。比如 $\sqrt{3}$ 就是多项式 $x^2 - 3$ 的一个根。不要错误地以为实数都是代数数，圆周率 π 就不是任何一个整系数多项式的根。另一方面，代数数也不是都可以表示成为根式的，比如多项式 $x^5 + x + 1$ 的根就不能表示成为根式的形式。

代数数的计算，算法复杂，而且相当耗费时间。当然，用 Maple 的内建方法已经可以很方便的处理它了。在 Maple 中，代数数用函数 `RootOf()` 来表示。例如， $\sqrt{3}$ 作为一个代数数，可以表示为：

```

[> alpha := RootOf(x^2-3, x);
                                      $\alpha := \text{RootOf}(\_Z^2 - 3)$ 
[> simplify(alpha^2);
                                     3

```

在 Maple 的内部，代数数 α 不再被表示成根式；而在化简时，仅仅利用到 $\alpha^2 = 3$ 这样的事实。这里 Maple 用到了一个内部变量 $_Z$ ，有关变量的内容，我们在下一节中将详细介绍。上面用的是赋值的方法；通过下面的例子，您将看到，利用 Maple 的别名方法 `alias()`，可以使这样的计算更清楚明了。

```

[> alias(beta = RootOf(x^2-3, x)):
    1/(1+beta) + 1/(1-beta);
                                      $\frac{1}{1+\beta} + \frac{1}{1-\beta}$ 
[> simplify(%);
                                     -1

```

在可以相互转换的前提下，你也可以使用 `convert()` 函数在根式和代数数之间相互转换。例如：

```

[> convert((-8)^(1/3), 'RootOf');
                                      $\text{RootOf}(\_Z^3 + 8)$ 
[> convert(%, 'radical');
                                      $\left(\frac{1}{3}\right)$ 
                                     (-8)

```

或者，你可以使用 `allvalues()` 函数，以获得代数数所有的值：

```

[> allvalues(beta);
                                      $\sqrt{3}, -\sqrt{3}$ 

```

1.2.3 复数运算

和代数数不同，复数在 Maple 中是基本的数据类型。虚数单位 i ($\sqrt{-1}$) 在 Maple 中用 `I` 表示。在运算中，数值类型转化成复数类型是自动的，所有的算术运算符对复数类型均适用。另外，还可以用 `Re()`、`Im()`、`conjugate()` 和 `argument()` 等函数分别实现取复数的实部、虚部、共轭和幅角等运算。请看下面的例子：

```

[> complex_number := (1+2*I) * (3+4*I);
                                      $\text{complex\_number} := -5 + 10 I$ 
[> Re(%); Im(%); conjugate(%%);
                                     -5
                                     10
                                      $-5 - 10 I$ 
[> argument( complex_number );
                                      $-\arctan(2) + \pi$ 
[> 1 / complex_number;
                                      $-\frac{1}{25} - \frac{2}{25} I$ 

```


有时候,为了在符号表达式中使用 Maple 的复数运算,可以用函数 `evalc()`。函数 `evalc()` 把表达式中所有的符号变量都当成实值的,也就是认为所有的复变量都写成 $a + bI$ 的形式,其中 a 、 b 都是实变量。

```
[> 1 / (3 + a - b*I);
      1
      3 + a - I b
[> evalc(%);
      3 + a      I b
      (3 + a)^2 + b^2 + (3 + a)^2 + b^2
[> abs(%);
      1
      |3 + a - I b|
[> evalc(%);
      1
      sqrt(9 + 6 a + a^2 + b^2)
[> sqrt(a + b*I);
      sqrt(a + I b)
[> evalc(%);
      1/2 sqrt(2 sqrt(a^2 + b^2) + 2 a) + 1/2 I csgn(b - I a) sqrt(2 sqrt(a^2 + b^2) - 2 a)
```

在这里,复数域的符号函数 `csgn()` 是这样定义的:

$$\text{csgn}(z) = \begin{cases} 1 & \text{Im}(z) > 0 \text{ 或 } (\text{Im}(z) = 0 \text{ 且 } \text{Re}(z) > 0) \text{ 时} \\ 1 & z = 0 \text{ 时} \\ -1 & \text{其余} \end{cases}$$

也就是说在复平面的上半平面及实轴的正半轴上为 1。不过,有时为了使表达式简单,我们可以用 `assume` 预先设定变量的取值范围:

```
[> assume(a>0): assume(b>0):
[> evalc(sqrt(a + b*I));
      1/2 sqrt(2 sqrt(a^2 + b^2) + 2 a) + 1/2 I sqrt(2 sqrt(a^2 + b^2) - 2 a)
```

1.3 Maple 中的变量和常量

1.3.1 赋值

从前面的学习中,相信聪明的你一定已经发觉 Maple 与其它编程语言,比如 C、Pascal、Fortran 等等的有一个极大的区别。在 Maple 中,不需要申明变量的类型,甚至在使用变量前并不需要将它赋值。这也正是符号演算的魅力之所在。这个特性是由 Maple 与众不同的赋值

方法决定的，在这一段里，我们将揭开 Maple 赋值的神秘面纱……。

为了理解 Maple 的赋值机制，我们先来看下面这个例子：

```
> polynomial := 9*x^3 - 37*x^2 + 47*x - 19;
      polynomial := 9 x3 - 37 x2 + 47 x - 19
> roots(polynomial);
      [1, 2], [19/9, 1]
> subs(x=19/9, polynomial);
      0
```

在这个例子中，第一条语句是一个赋值语句，它的作用是把变量 `polynomial` 和多项式 $9x^3 - 37x^2 + 47x - 19$ 相关联。在赋值之后，每当 Maple 遇到变量 `polynomial` 时，就取与之唯一关联的“值”。比如随后的语句 `roots(polynomial)`，就被 Maple 理解为 `roots(9*x^3 - 37*x^2 + 47*x - 19)` 了。现在，变量 `x` 还没被赋值，只有它的名字“`x`”。通过语句 `subs(x=19/9, polynomial)`，我们得到将 `polynomial` 所关联的多项式中的所有 `x` 都替换成 $19/9$ 之后的结果——0，这就是我们在数学中常用的方法——变量替换。但此时，无论是 `x` 还是 `polynomial` 的值都没有被改变：

```
> x; polynomial;
      x
      9 x3 - 37 x2 + 47 x - 19
```

当然，如果仅仅是为了检验计算的结果，我们完全可以采用另一种方法：

```
> x := 19/9;
      x := 19/9
> polynomial;
      0
```

我们可以这样来理解 Maple 执行上面后一句语句的过程：

- ✧ Maple 读到变量 `polynomial` 并且把它换成与之关联的多项式 $9x^3 - 37x^2 + 47x - 19$
- ✧ 接着，它又在该多项式中发现了已赋值的变量 `x`，并将其替换成 `x` 的值 $19/9$
- ✧ 最后，Maple 将结果进行算术化简，得到 0

通过赋值而非代换来检验计算结果缺点是，现在变量 `x` 已经指向了一个确定的值。如果你还想让它恢复成未定义的变量，也就是除了它的名字“`x`”以外，什么值也没有的变量，你可以这样做：

```
> x := 'x';
      x := x
```

从这里，我们可以知道，Maple 对用一对单引号“`'`”括起来的部分不进行值的代入，而只取其变量名。为了取消变量的赋值，我们还可以使用函数 `evaln()`。

```
> x := 10; x := evaln(x);
      x := 10
      x := x
```

在上面的例子中，使用函数 `evaln()` 与使用单引号的结果是一样的。但要是你想取消带

指标的变量（如 $A[i]$ ）或联合变量（如 $A.i$ ）的赋值，其中 i 也是一个变量，你就不能再使用单引号了。

```
> i := 1; A[i] := 2; A.i := 3;
      i := 1
      A1 := 2
      A1 := 3
> A[i] := evaln(A[i]); A[i];
      A1 := A1
      A1
```

上面的例子成功的将 A_i 取消了赋值，我们可以看一看如果用单引号会出现什么后果。

```
> A.i := 'A.i';
      A1 := Ai
> A.i;
Error, too many levels of recursion
> i := 2; A1;
      i := 2
      A2
```

很显然，使用单引号的结果是使 A_1 指向了一个暂时没有计算的表达式 $A.i$ ，而这个表达式在此时的计算结果是 A_1 。这样， A_1 绕了一个圈子又指向了自己；事实上，这是一个循环定义。接着，在求 $A.i$ 时，Maple 无法避免地引起了递归次数过多的错误。如果这时把 i 赋为 2，则不出我们的意料， A_1 果然指向了 A_2 。这就是 Maple 的变量赋值的机制，说的通俗一点，就是尽可能的“代入”，对应着数据结构中树的遍历。这种方法我们看起来虽然不怎么“聪明”，但对于计算机来讲，确是一种行之有效的方法。

在实际计算过程中，变量繁多，我们往往会忘记哪些变量已经赋了值，而哪些变量还没被赋值，Maple 提供了以下几个函数可以帮助我们确切地了解工作空间中的变量。

表 1.1 变量赋值相关函数

函数名称	用途
anames	显示所有已赋值的变量名
unames	显示所有未赋值的变量名
assigned	检查一个变量是否已经被赋值

最后，我们介绍两个也许并不十分必要的函数，`assign()` 和 `unassign()`。`assign` 是用来给变量赋值的，`assign(name, expression)` 的作用和 `name := expression` 完全一样。`unassign` 用来取消赋值，和前面介绍的 `evaln` 不同，`unassign` 可以同时取消一批变量，使用起来也许会方便一些吧。不幸的是，`unassign` 是库函数，使用前要用 `readlib(unassign)` 调入库，哎，还是不方便，不是吗？

1.3.2 变量名

在 Maple 中，最简单的变量或常量名是字符串，规则和一般的高级语言差不多，必须是一个由字母、数码或下划线组成的序列，其中第一个字符必须是字母或是下划线。名字的长度限制是 499 个字符，相信这个限制对于任何用户来讲都是形同虚设的了。下面是一些合法

的变量名: $x, y, a_long_name_containing_underscores, H_2O$ unknown, UNKNOWN, UnKnown……

需要注意的是, 在 Maple 中是区分大小写的。在使用变量、常量和函数时都要牢记这一点。数学常量 π 在 Maple 中用 Pi 表示, 而 pi 却与之毫无干系; -1 的平方根是 I 而不是 i。

以下划线开头的变量名是 Maple 内部使用的, 也许你还记得上一节中 RootOf 的例子, 应该尽可能的避免使用, 否则有时会出现意想不到的结果。

```
> _Z := 2;
                                     _Z := 2
> RootOf(x^2-x, x);
Error, (in RootOf) expression independent of, 2
```

除此以外, Maple 中还有一些保留字, 也不可以被用作变量名。

表 1.2 Maple 保留字

and	by	do	done	elif
else	end	fi	for	from
if	in	intersect	local	minus
mod	not	od	option	options
or	proc	quit	read	save
stop	then	to	union	while

另外, Maple 的内部函数, 如 sin, cos, exp, sqrt……, 也不可以被用作变量名。

如果为了增强可读性或是别的原因, 你觉得有必要在变量名中加入空格, 或是其他的特殊字符比如 “.”、“-”、“/” 等等, 你可以使用反向的撇号 “`” (位于键盘左上角) 将其括起来; 如果要加入这个撇号本身, (虽然这也许没有必要), 可以加入两个这样的撇号。下面也是一些合法的 Maple 变量名称:

```
> `exercise 1`, `C:\windows`, `a`b`;
                                     exercise 1, C.\windows, a`b
> ``; #空串名
> `中文名字`;
                                     中文名字
```

注意, 不要把 Maple 中用反向撇号括起来的部分和其他高级语言中的字符串混淆起来。

下面的例子将有助于你理解它们的区别:

```
> a_variant := `apparent_text`;
                                     a_variant := apparent_text
> apparent_text := sin(Pi);
                                     apparent_text := 0
> a_variant, `a_variant`;
                                     0, 0
```

正如你所见到的, 在这里, 加不加反向撇号毫无作用; 反向撇号仅仅是为了输入特殊字符而用的。上面, 我们介绍了两个方向的单引号, 那么双引号有没有用呢? 一试便知。

```
> a_string := "This is a string enclosed by double
quotes";
                                     a_string := "This is a string enclosed by double quotes"
> "a_name" := another;
Error, invalid lhs of assignment
```

和其他常用的编程语言一样, 一对双引号括起来的部分表示字符串, 和变量名不同, 它

不能被赋值。需要特别指出的是，直到 Maple V Release 4 为止，一个双引号都表示上一次计算的结果；而 Maple V Release 5 新加了字符串这个数据类型，同时把引用上一次计算结果的符号变换为 “%”。至此为止，我们已经分别介绍了 Maple 中的双引号、单引号、反向撇号，下面将其用途总结成一个表，希望读者不要把他们搞混。

表 1.3 Maple 中的引号

符号	用途
` `	界定一个包含特殊字符的名字
' '	界定一个暂时不求值的表达式
" "	界定一个字符串

在解决实际问题，常常需要处理一批相关联的数据，这时，我们希望把它们表示成字母加数字的形式，如 A1, A2, A3……。由于 Maple 是一个符号演算系统，我们往往提出这样的要求，即变量的指标（上面的 1, 2, 3……）也要用符号表示；这也是合情合理的，几乎任何一本数学教科书中都会有这样的符号： A^i , B_j , C_{ijk} 等等。在 Maple 中，我们可以用函数 cat 或运算符 “.” 表示联合变量名，来完成这样的功能。

```

[> X.Y, X.1;
                                XY, X1
[> X.(2..10);
                                X2, X3, X4, X5, X6, X7, X8, X9, X10
[> `.`.(X, Y).(1..4);
                                X1, X2, X3, X4, Y1, Y2, Y3, Y4
[> libname, cat(libname, `/cat.m`);
"C:\PROGRAM FILES\MAPLE V RELEASE 5\lib",
"C:\PROGRAM FILES\MAPLE V RELEASE 5\lib/cat.m"
[> i:=4: X.i, i.X;
                                X4, iX

```

从上面最后一个例子中我们可以看出，在适用联合操作符 “.” 时，最左边的一个变量名总不会被计算，比如 x.1 的结果一定是 x1，而无论这时 x 的值究竟是什么。

1.3.3 基本数据类型

在 Maple 中，有着很严格的数据类型，数据的类型决定着它的内部数据结构，同时也决定着在这个数据上可以进行的运算。通常的基本数据类型，比如整型，浮点型，字符串等等，在 Maple 中都有；而 Maple 是被设计作数学工具的，它当然更有着其他的数据类型。在 Maple 中，你可以用命令 whattype 来查看一个变量或一个表达式的类型。

```

[> whattype(3.14);
                                float
[> whattype(Pi);
                                symbol
[> whattype({a, b, c});
                                set
[> whattype(a, b, c);
                                exprseq

```

命令 whattype 得到的结果实际上是 Maple 中数据结构头节点中的类型描述符，也就是

说，这种数据的分类方法是以它们在 Maple 中的存储结构分的。因此，在 Maple 中把这种数据类型叫做“表观数据类型”(surface data types)。下面的表是 Maple 中常用的表观数据类型的一个总结，如果需要获得更为全面的类型，请查看在线帮助 type,surface。

表 1.4 Maple 中的表观数据类型

类别	数据	类型	举例
数字和字符串	整数	integer	1
	分数	fraction	1/2
	浮点数	float	0.33333
	复数	complex	$1 + 3*I$
	符号	symbol	xvalue
	字符串	string	"Beijing, China"
算术式	和	<code>`+`</code>	$x + y$
	积	<code>`*`</code>	$x * y$
	幂	<code>`^`</code> 或 <code>`**`</code>	$x ^ y$
关系式	等式	<code>`=`, equation</code>	$x+1 = 1+x$
	不相等关系式	<code>`<>`</code>	$Pi <> pi$
	小于关系式	<code>`<`</code>	$2 < 3$
	小于等于关系式	<code>`<= `</code>	$x <= y$
逻辑式	与	<code>`and`</code>	P and Q
	或	<code>`or`</code>	P or Q
	非	<code>`not`</code>	not P
复合表达式	表达式序列	expseq	a, b, c
	集合	set	{a, b, c}
	有序表	list	[a, b, c]
	映射表	table	table([a, b, c])
	带指标的变量	indexed	X[1]
	函数	function	f(x) (其中 f 尚未定义)
其他	未求值的联合名	<code>`..`</code>	a.(1..n) (其中 n 尚未赋值)
	域	<code>`..`, range</code>	1..3
	幂级数	series	series(sin(x), x=0,6)
	子程序	procedure	proc(x) x^3 end
	未求值的表达式	uneval	'x + y'

在 Maple 中，还有着另外两个工具可以测试数据类型——type 和 hastype，与 whattype 不同的时，它们不仅可以测试表观数据类型，还可以测试更为一般的数学类型。下面是使用这两个函数的简单例子，你也可以查看在线帮助以获得更为详细的使用方法。

```

[> type(x + y, `+`);
                                     true
[> hastype(x + y, `+`);
                                     true
[> type(x^2+x+1, polynomial);
                                     true
[> type(x^2+x+Pi, polynomial(integer, x)); #x的整系数多项式
                                     false

[> type(x^2+x+1, quadratic(x)); #x的二次式
                                     true

```

在常用的程序设计语言，如 C、Fortran 中，你必须事先声明变量的类型，而且，在使用过程中，类型是不可以改变的。但是在 Maple 中，你不会受到这样的限制，变量没有固定的类型。这点从下面这个例子中就可以看出来。

```
> number := 1: whattype(number);
integer
> number := 0.75: whattype(number);
float
> number := convert(number, fraction);
number := 3/4
```

但是，变量也并不是可以随意转换类型的。比如，Maple 中规定，分数不可以改变数制（比如将上面的 3/4 化成二进制就是不允许的）。

```
> convert(number, binary);
Error, (in convert/binary) invalid argument for convert
```

1.3.4 Maple 中的常数

为了解决各种数学问题，一些常用的数学常数是必不可少的。Maple 系统中，已经存储一些数学常数，它们被放在一个表达式序列——constants 中。你也可以在其中加入自己定义的符号常数。

```
> constants;
false, γ, ∞, true, Catalan, FAIL, π
> constants := constants, electron_rest_mass, kg;
constants := false, γ, ∞, true, Catalan, FAIL, π, electron_rest_mass, kg
> type(electron_rest_mass, constant);
true
> electron_rest_mass := 9.109558E-31*kg;
electron_rest_mass := .9109558 10-30 kg
```

这里需要说明的是，在以前的版本（Maple V Release 3 以前）中，自然对数的底——E，是作为一个系统常数出现的；但在新版本中，这个常数被取消了。事实上，这个常数还是在的，因为你还可以用 exp(1)来获得它；取消的只是符号 E。确实，这一个大写字母 E 很容易让人产生误会，学工科的读者一定知道，这个字母在工程上也有着相当重要的意义——弹性模量（Young's Module）。

表 1.5 Maple 中的数学常数

常数	名称	近似值
圆周率 π	Pi	3.131592654
Catalan 常数 $C = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)^2}$	Catalan	0.9159655942
Euler-Mascheroni 常数 $\gamma = \lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k} - \ln n \right)$	gamma	0.5772156649

∞	infinity	
----------	----------	--

1.4 函数和表达式

1.4.1 Maple 中的数学函数

作为一个数学工具，基本的数学函数是不可缺少的。即使是一个袖珍计算器，我们也可以用它来计算乘幂，对数，三角函数等等。而 Maple 中提供的数学函数就更多了。由于这些函数不是这里介绍的重点，但又考虑到读者需要尽快地掌握 Maple 的使用，就必须知道一些基本的数学函数，所以，我们在这里将一些常用的函数列成一张表，供读者查阅。

表 1.6 常用的数学函数

函数	Maple 中的名称
e 指数函数	exp
自然对数	ln, log
常用对数	log10
平方根	sqrt
绝对值	abs
三角函数	sin, cos, tan, sec, csc, cot
反三角函数	arcsin, arccos, arctan, arcsec, arccsc, arccot
双曲函数	sinh, cosh, tanh, sech, csch, coth
反双曲函数	arcsinh, arccosh, arctanh, arcsech, arccsch, arccoth
贝赛尔函数	BesselI, BesselJ, BesselK, BesselY
Gamma 函数	GAMMA
误差函数	erf

对于其中的很多函数，Maple 可以求得它们在某些点上的精确值，比如在 $\frac{\pi}{8}$ 、 $\frac{\pi}{10}$ 、 $\frac{\pi}{12}$ 它们的整数倍上的三角函数值等等。当然，你可以用 evalf 获得任意精度的近似值。

1.4.2 多项式

多项式和有理式（即多项式和分式）可以说是 Maple 最为拿手的表达式了。也许你还记得高等数学课上学习不定积分时，习题中有理式的出现是多么的频繁吧。理论上，有理式积分的解析表达式总是存在的；同样，有理式，特别是多项式，在其他各种运算中都有着无与伦比的优势，因此，它在数值逼近等方面都有着广泛的运用。

我们先来看看简单的单变量多项式。多项式的加法、减法和乘法在 Maple 中都可以快速准确地完成。


```

> p1 := -x + 2*x^2 + 3*x^3 + 4*x^4;
      p1 := -x + 2x^2 + 3x^3 + 4x^4
> p2 := 5*x^5 - 6*x^2 + 9;
      p2 := 5x^5 - 6x^2 + 9
> p2*3 - 2*p1 + 2*x;
      15x^5 - 22x^2 + 27 + 4x - 6x^3 - 8x^4
> p1 * p2;
      (-x + 2x^2 + 3x^3 + 4x^4)(5x^5 - 6x^2 + 9)
> expand(%);
      -29x^6 + 33x^3 - 9x + 10x^7 + 24x^4 + 18x^2 + 15x^8 - 18x^5 + 20x^9

```

与数字的乘法不同，在 Maple 中，多项式的乘积不会被自动地展开，而需要调用函数 `expand()`。这看起来象是画蛇添足，其实不然，有时表示成因子的乘积形式会比展开式简单的多。比如式子 $(3x+5)^{10}$ ，你一定希望将其表示成为这样的形式，而不是展开成为 $59049x^{10} + 984150x^9 + 7381125x^8 + \cdots + 58593750x + 9765625$ 。

在上面的例子中，虽然 Maple 计算结果的正确性是无可非议的，但结果的各个项却是打乱的，而不是排列成为变量的升幂序或是降幂序。这样的式子可读性很差，但出于效率上的考虑（计算时间和存储空间），Maple 不自动地将其重新排列。如果需要将多项式按照降幂序排列，你可以使用 `sort` 命令。

```

> sort(%);
      20x^9 + 15x^8 + 10x^7 - 29x^6 - 18x^5 + 24x^4 + 33x^3 + 18x^2 - 9x

```

于其他一些命令不同，`sort` 命令不仅得出排序后的结果，而且，内部的数据结构也会相应改变。从 Maple 的运行机制上来看，这是有必要的。因为每一个表达式或子表达式（表达式中的一部分）在 Maple 中都只存储一次。对于每一个新表达式，Maple 首先用内建的代数系统检查它是否可以化成已有的表达式，或者可以用已经存在的元素或表达式构造出来。改变和式或乘积中的一个元素的形式，则整个式子的表达形式会跟着改变。不要小看了这一点，这在化简过程中对于杜绝重复计算，减少计算量是相当有效的。你也许早有体会，在 Maple 中同样的计算第二次进行时，速度要快得多。从下面的简单例子中我们可以看到这种机制的影子。

```

> p := 1 + x + x^3 + x^2; # 未排序
      p := 1 + x + x^3 + x^2
> x^3 + x^2 + x + 1; # 式子出现过，仍用前式
      1 + x + x^3 + x^2
> q := (x - 1) * (x^3 + x^2 + x + 1); # 式子的部分已存在
      q := (x - 1)(1 + x + x^3 + x^2)
> sort(p);
      x^3 + x^2 + x + 1
> q; # q的子式也跟着改变
      (x - 1)(x^3 + x^2 + x + 1)

```

Maple 中有着许多处理多项式的函数和子程序，下面我们将选择其中一些较常用的作简

单的介绍。对于一个多项式，我们可以用函数 `coeff()` 求得具体一个项的系数；还可以用 `lcoeff()` 和 `tcoeff()` 分别求得首项系数 (**leading coefficient**) 和末项系数 (**trailing coefficient**)，这里的首项、末项都是针对降幂序排列而言的。请看下面的例子 (`p1`、`p2` 都是沿用前面例子中的定义的)

```
> coeff(p2, x^3), coeff(p2, x^2);
                                0, -6
> lcoeff(p1, x), tcoeff(p1, x);
                                4, -1
```

如果要求多项式所有的系数，可以调用函数 `coeffs()`。`coeffs()` 中的第三个参数是一个变量名，Maple 将利用它来返回各个系数所对应的项（它们组成一个表达式序列），它可以是一个未被赋值的变量，也可以是已被赋值的，不过这时必须用单引号 “'” 将它括起来。顺带提一句，如果读者学过 C++ 的话，可以比较一下 Maple 中用单引号括起来的变量和 C++ 中的引用，它们之间有一定的相似性。

```
> coeffs(p2, x, 'powers'); powers;
                                9, -6, 5
                                1, x^2, x^5
```

我们还可以用函数 `degree()` 获得多项式的次数（阶数），但要注意，必须将多项式展开成为和式，否则，很有可能出错（在化简后最高次系数为 0 时）。如下例所示：

```
> degree(x^2-x*(x-1));
                                2
```

代数学中对多项式经常作的运算是除法和求余，在 Maple 中，也有对应的函数 `quo()` 和 `rem()`。这两个函数都有第四个参数，分别用来返回余项和商式。

```
> quo(p2, p1, x, 'r'); r;
                                5      15
                                4 x - 16
                                9 - 23 2 + 5 3 - 15
                                8 x  + 16 x  - 16 x

> rem(p2, p1, x, 'q'); q;
                                9 - 23 2 + 5 3 - 15
                                8 x  + 16 x  - 16 x
                                5      15
                                4 x - 16
```

同样，我们还可以用函数 `gcd()` 来求得两个多项式的最大公因式 (**greatest common divisor**)。求最大公因式在代数学中是一个重要的运算，在数学上已经有很深入的研究；在 Maple 中，这个运算比之其他同类运算有较高的运算速度。相对来说，因式分解是一个复杂而且费时的过程（但是比起手算来还是要快的多了），在 Maple 中函数 `factor()` 可以将多项式在有理数范围内进行因式分解。

```

> gcd(p1, x^2 + x);
> p := expand( x^2 * (2*x-12) * (x^2-5*x+6) );
p := 2x^5 - 22x^4 + 72x^3 - 72x^2
> factor(p);
2x^2(x-6)(x-2)(x-3)

```

上面，我们介绍了一些单变量多项式的计算；在 Maple 中，多变量多项式的计算也是几乎相同的。和单变量多项式一样，Maple 不会自动对多项式的项进行次序上的整理，项之间的先后次序取决于第一次出现式的顺序；我们可以用函数 `sort()` 对多项式进行项排序。

```

> polynomial := 6*x*y^5 + 12*y^4 + 14*y^3*x^3
- 15*x^2*y^2 + 9*x^3*y^2 - 30*x*y^2 - 35*x^4*y
+ 18*y*x^2 + 21*x^5;
polynomial := 6xy^5 + 12y^4 + 14y^3x^3 - 15x^2y^2 + 9x^3y^2 - 30xy^2 - 35x^4y
+ 18yx^2 + 21x^5
> sort(polynomial, [x, y], 'plex');
21x^5 - 35x^4y + 14x^3y^3 + 9x^3y^2 - 15x^2y^2 + 18x^2y + 6xy^5 - 30xy^2
+ 12y^4

```

函数 `sort()` 的第二个参数，是一个有序表，用于输入多项式的变元；第三个参数是字符串，用于输入排序方法，'plex' 表示字典顺序 (**p**ure **l**exicographic **o**rdering)，它是如下定义的：

$x^i y^j < x^{i'} y^{j'} \Leftrightarrow i < i' \text{ 或 } (i = i' \text{ 且 } j < j')$ ，故 $1 < y < y^2 < \cdots < x < xy < x^2$ 。另一种排序

依据 'tdeg'，是按所有变量的总次数 (**t**otal **d**egree) 的降幂排列：

$$1 < y < x < y^2 < xy < x^2 < y^3 < xy^2 < x^2y < x^3 < \cdots$$

默认情况下，Maple 采用的就是这种方法：

```

> sort(polynomial);
6y^5x + 14y^3x^3 + 9y^2x^3 - 35yx^4 + 21x^5 + 12y^4 - 15y^2x^2 - 30y^2x
+ 18yx^2

```

由于 Maple 进行的是符号运算，上面的这个多项式也可以看成是一个变量 x 的多项式，而相应的系数又是 y 的多项式。我们可以用函数 `collect()` 将同类项合并起来：

```

> collect(polynomial, x);
collect(6y^5x + 14y^3x^3 + 9y^2x^3 - 35yx^4 + 21x^5 + 12y^4 - 15y^2x^2 - 30y^2x
+ 18yx^2, x)

```

各种求系数的函数对于多变量多项式也一样适用，此处不再赘述。

1.4.3 有理式

所谓有理式，是指这样的式子，它可以表示成 f/g 的形式，其中 f 、 g 都是多项式，而且 $g \neq 0$ 。对于一个有理式，你可以分别用 `numer()` 和 `denom()` 来获得它的分子（**numerator**）和分母（**denominator**）。

```
> f := x^2 + 3*x + 2: g := x^2 + 5*x + 6: f/g;
      x^2+3x+2
      -----
      x^2+5x+6
> numer(%), denom(%);
      x^2+3x+2, x^2+5x+6
```

可以注意到，和数字的除法不同，Maple 并不自动地化简分式，使分子和分母互不可约。除非 Maple 可以不用计算就发现分子分母的公因式，也就是说分子分母都表示成乘积的形式，并且有公共部分，例如：

```
> ff := (x-1)*f; gg := (x-1)^2*g; ff/gg;
      ff := (x-1)(x^2+3x+2)
      gg := (x-1)^2(x^2+5x+6)
      x^2+3x+2
      -----
      (x-1)(x^2+5x+6)
```

如果需要将分式化成最简形式，你可以调用函数 `normal()`，将分式正则化。

```
> normal(f/g);
      x+1
      ---
      x+3
```

Maple 不自动对分式进行正则化的原因有三：

- ✧ 正则化之后，在形式上往往并不比原来的式子简单，例如 $(x^{10000} - 1) / (x-1)$ ，正则化之后就会成为一个有一万项之多的庞大多项式；
- ✧ 如果对每一个有理式都进行正则化，则太耗费时间；
- ✧ 用户可能需要对式子进行进一步处理，不希望式子消去公因子。

1.4.4 有理式的转换

读者一定知道，利用秦九韶算法可以减少多项式求值的计算量。在 Maple 中，我们可以用函数 `convert()` 将多项式转化为这种形式（Horner form）。在下面的例子中，我们利用库函数 `cost()` 来获得求值所需的计算量。

```
> p1;
      -x+2x^2+3x^3+4x^4
> readlib(cost)(p1);
      3 additions + 9 multiplications
> convert(p1, 'horner');
      (-1+(2+(3+4x)x)x)x
```

```
> cost(%);
3 additions + 4 multiplications
```

因为 `cost()` 是库函数，所以第一次调用时需要用命令 `readlib()` 将其调入。从上面的例子中可以看到，转换后的计算量有了很大程度的减少——从 3 次加法 9 次乘法，减少到 3 次加法 4 次乘法。

同样，把分式化成连分式（**continued fraction**）形式也可以降低求值所需的计算量。

```
> (1 + x + x^2 + x^3)/p1;
      x^3 + x^2 + x + 1
      -x + 2x^2 + 3x^3 + 4x^4

> cost(%);
6 additions + 12 multiplications + divisions

> convert(%, 'confrac', x);
      1
      4
      x - 1/4 - 1/4
      x - 3 + 14
      x + 29/7 - 20/49
      x - 1/7

> cost(%);
7 additions + 4 divisions
```

在某些场合下，（比如求微分、积分时），把分式化成部分分式（**partial fraction**），也就是几个最简分式的和式的形式，可以简化计算。

```
> convert(%, 'parfrac', x);
      -1/x + (3 + 4x + 5x^2) / (4x^3 + 3x^2 + 2x - 1)
```

1.5 内部数据结构和变量代换

1.5.1 多项式和分式的内部表示法

这里，我们将花一定的篇幅来介绍 Maple 中多项式和分式的内部结构。

我们以多项式 $x^4 + x^3 - x^2 - x$ 为例，来看看多项式的数据结构。一个展开的多项式是用如图 1.2 的方式表示的，在图论中，我们称之为有向无环图（DAG）。

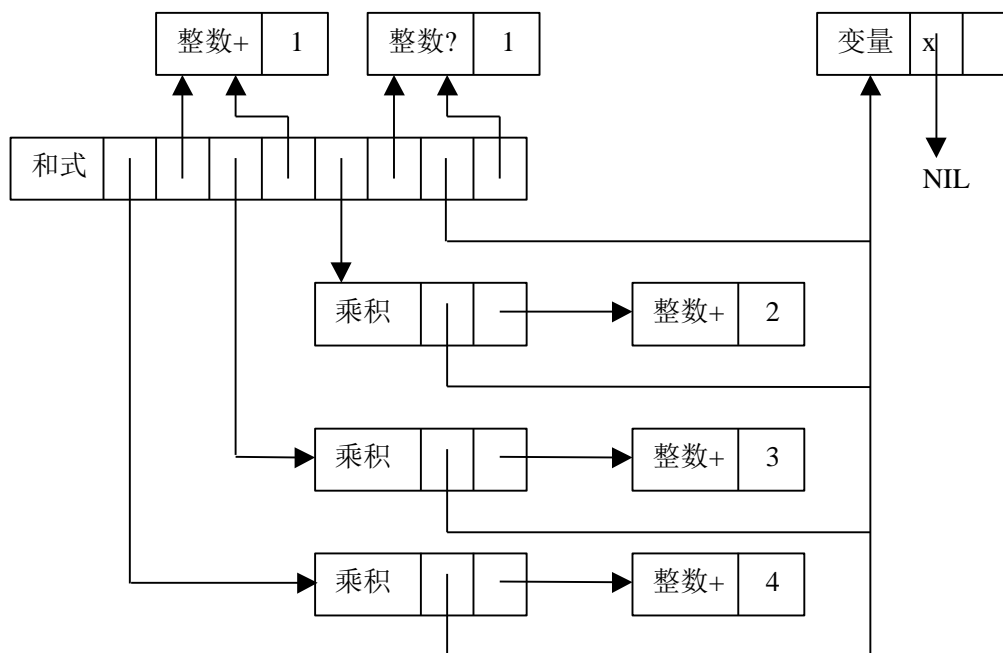
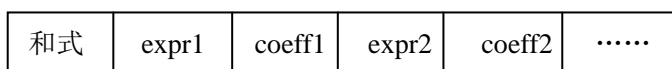


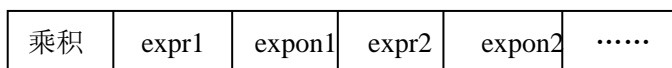
图 1.2 多项式的数据结构

在这里，多项式的数据向量有着这样的形式：



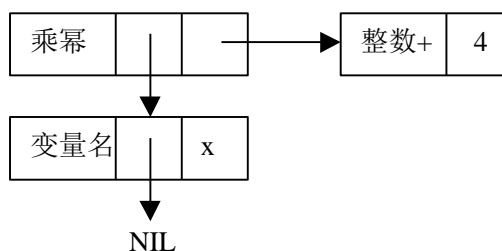
它表示这样的和式： $\text{coeff}_1 \times \text{expression}_1 + \text{coeff}_2 \times \text{expression}_2 + \dots$ 。

同样，Maple 用下面的数据向量表示： $\text{expression}_1^{\text{expon1}} \times \text{expression}_2^{\text{expon2}} \times \dots$ 。



前面的图中，空指针 NIL 表示变量 x 未被赋值。

也许你会觉得奇怪，为什么子表达式 x^4 不用这样的数据结构呢？

图 1.3 x^4 不用这样的数据结构

看起来，这似乎更为合情合理。实际上，这种形式的数据结构在 Maple 中确实存在，但是当指数式数值常量时，Maple 的化简机制就自动地把它转换成乘积形式的了。

通过多项式的数据结构，我们还可以很清楚地看到， x^4 、 x^3 、 x^2 、 $-x^2$ 、 x 、 $-x$ 都是多项式 $x^4 + x^3 - x^2 - x$ 的子式；而式子 $x^4 + x^3$ 或者 $x^4 - x$ 却都不是它的子式，虽然它们在数学上来讲，都是它的一部分。这一机制就可以解释为什么有时 Maple 的变量代换 `subs()`，会得到类似于下面的似乎令人不解的结果了。

```

> subs( 1 = 7, x^4 + x^3 - x^2 - x );
      7x4 + 7x3 - x2 - x
> subs( 1 = 3, Pi*x + x + 1 );
      3π3x3 + 3x + 9

```

让我们来仔细看一看后面的这个例子。正如我们所知道的，式子 $\pi x + x + 1$ 在 Maple 内部是这样表示的：

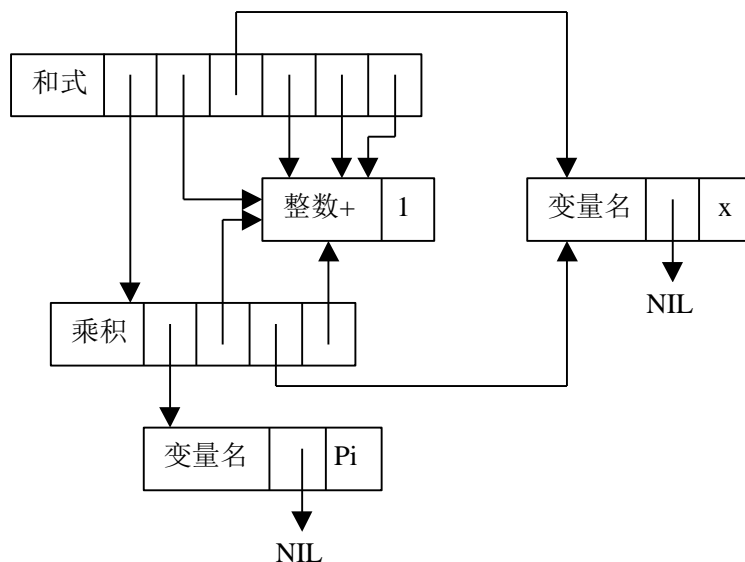


图 1.3 式子 $\pi x + x + 1$ 的数据结构

试一试，在上面的数据结构中将整数 1 替换成 3，你就会明白前面例子中结果的来由了。这两个例子告诉我们，Maple 的内部数据结构会和外部的表示有很大的差别。对于 Maple 来讲，化简无非是把表达式从一个数据结构转换成另一个数据结构而已。

在开始介绍新的内容以前，让我们再来想一想，为什么 Maple 的设计者选择乘积形式，而非乘幂形式来表示多项式的一个项呢？我们以一个多元的项 $x^2y^3z^4$ 为例，来看看两种表示方式的不同。这个项在 Maple 中是这样表示的：

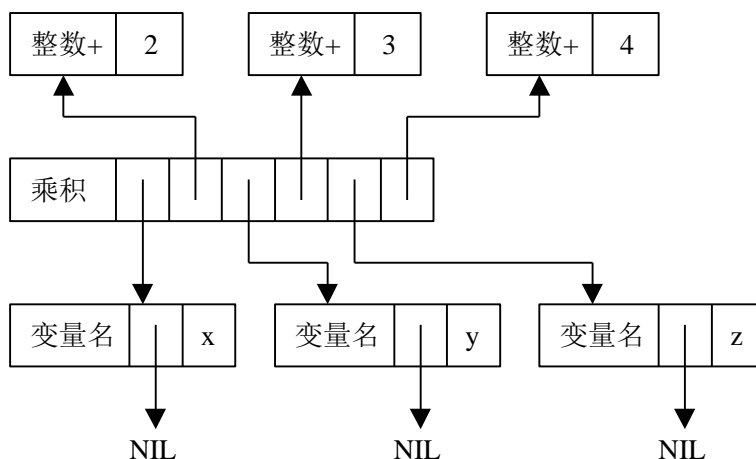
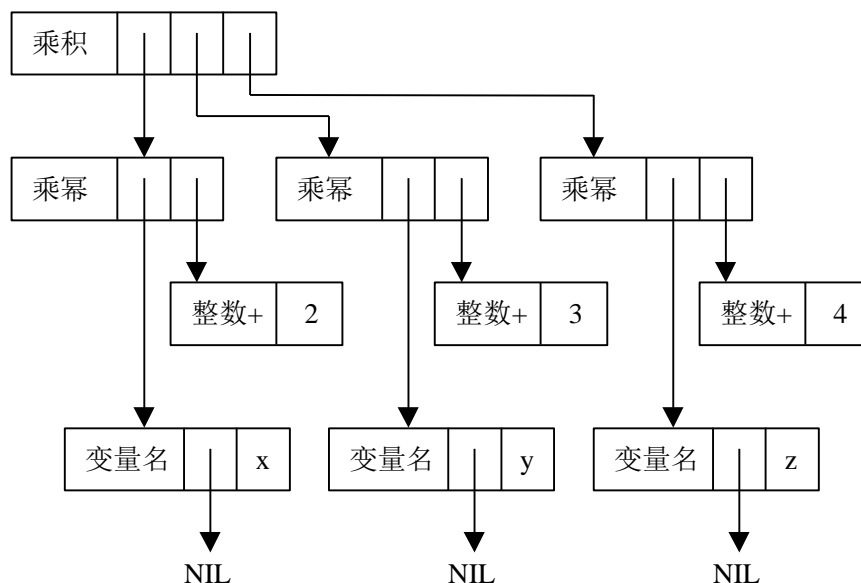


图 1.4 乘积形式下 $x^2y^3z^4$ 的数据结构

如果采用乘幂 x^2 、 y^3 、 z^4 的乘积的形式来表示，数据结构则会是这样：

图 1.5 乘幂形式下 $x^2y^3z^4$ 的数据结构

很明显，后面一种表示法需要更多的存储空间；更重要的是，在进行项之间的乘法时，前一种方法可以更方便地将相同底数的幂的乘积合并。**Maple** 之所以选择前一种表示方法，是对多项式运算——符号运算中最为常用的运算——予以了优化。

也许你已经被这种有向无环图搞得晕头转向了，那好，我们把它简化一下。然我们暂时把 **Maple** 中的鬼条件“一切表达式只存储一次”扔到一边去。这样，我们的“有向无环图”就可以化成为一棵“树”。前面那个例子就可以表示成为图 1.6 的形式：

这儿省略了基本元素，如数值、变量名，的数据向量。这样，数据结构就一目了然了。在后面的介绍中，我们将都采用这种简化了的数据描述。

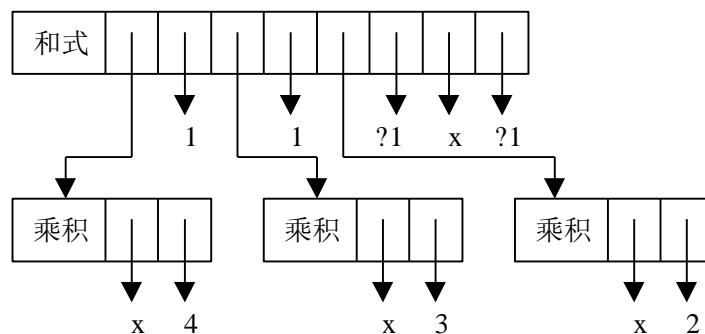


图 1.6 表达式树

1.5.1 分式的内部数据结构

前面，我们简单地对多项式的内部数据结构作了研究；作为多项式的自然延伸，我们在来看一看分式的内部结构。这里，我们通过一个例子来探索 **Maple** 这个黑匣子。我们所用的工具是 **Maple** 的函数 `op()`，它可以获得一个表观数据类型结构中的一个元素。


```

> r := (y^2-1)/(y-1);
                                     
$$r := \frac{y^2-1}{y-1}$$

> type( r, ratpoly );
                                     true
> whattype(r);
                                     *
> op(r);
                                     
$$y^2-1, \frac{1}{y-1}$$

> op(2, r);
                                     
$$\frac{1}{y-1}$$

> whattype(%);
                                     ^
> op(%%);
                                     
$$y-1, -1$$


```

我们看到，分式的表观数据类型是一个乘积，是子式 y^2-1 和 $(y-1)^{-1}$ 的乘积。通过这个例子，我们再一次体会到了 Maple 中内部数据结构和外部表达式之间的差异。上面讲的都是真正的有理式的例子，我们再扩展一下，来看一看下面这种形式的式子：

```

> r := (sin(x)^2 - 1) / (sin(x) - 1);
                                     
$$r := \frac{\sin(x)^2-1}{\sin(x)-1}$$

> type( r, 'ratpoly' );
                                     false

```

Maple 也告诉我们它并不是一个有理式。但你也许已经发现，如果把式子中的 $\sin(x)$ 统统替换成 y ，它就和前一例中的有理式一模一样了。

的确，如果用前一例中相同的手段，我们会发现它们的内部数据表示也是惊人的相似。（参见图 1.6）

我们可以把式子 $\frac{\sin^2 x - 1}{\sin x - 1}$ 看作是以 $\sin x$ 为变元的整系数有理式，Maple 也是这么认为的：

```

> type( r, 'ratpoly'(integer, sin(x)) );
                                     true

```

Maple 把这一类式子当成是广义有理式（generalized rational expression），我们也可以在这一类式子上使用常用的有理式函数，如正则化（约简）、求分子分母等等。但是有一些函数却不可以直接使用。

不过，我们可以使用一个有用的函数 `frontend()`，它可以将一个广义有理式中的函数调用“冻结”起来，成为真正的有理式，然后再调用有理式函数进行处理。它的第一个参数是所要调用的有理式函数，第二个参数是调用函数所需的参数列表。通过这个函数，我们就可以像处理有理式一样方便地处理一些复杂的式子了。

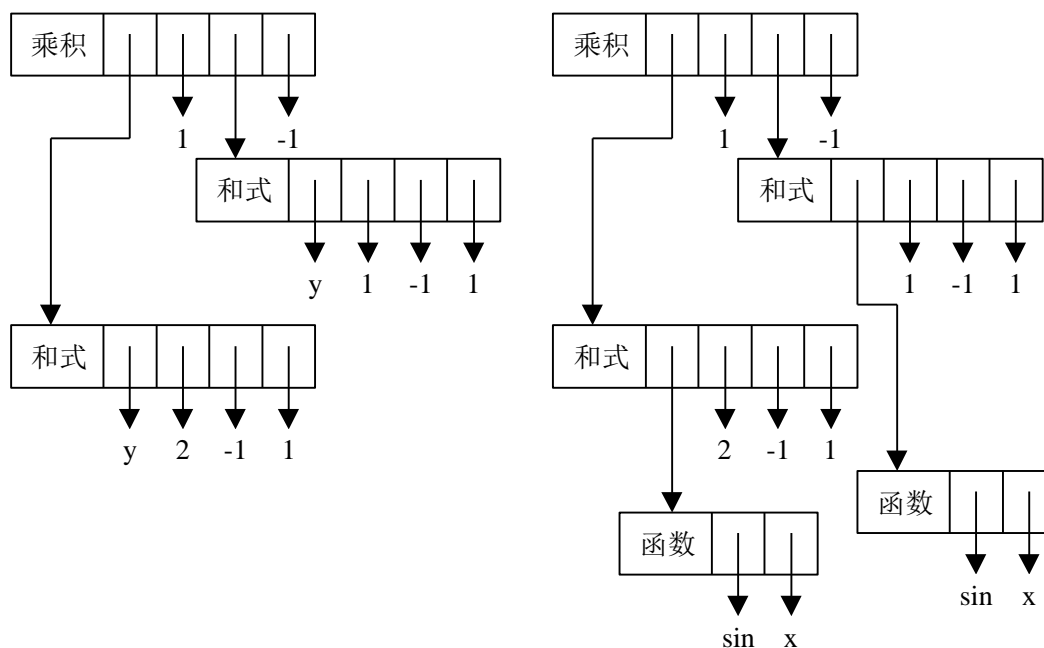


图 1.6 广义有理式和有理式的数据结构

1.5.2 变量代换

在表达式化简中，变量替换是一个得力工具。大的式子，可以通过把其中的子式替换成变量，得到极大的简化。比如矩阵

$$\begin{pmatrix} a & b & e & f & a & b \\ c & d & g & h & c & d \\ -e & -f & a & c & e & f \\ -g & -h & b & d & g & h \\ -a & -b & -e & -f & e & g \\ -c & -d & -g & -h & f & h \end{pmatrix}$$

就可以写成分块矩阵的形式：

$$\begin{pmatrix} \mathbf{X} & \mathbf{Y} & \mathbf{X} \\ -\mathbf{Y} & \mathbf{X}^T & \mathbf{Y} \\ -\mathbf{X} & -\mathbf{Y} & \mathbf{Y}^T \end{pmatrix}$$

其中， $\mathbf{X} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ ， $\mathbf{Y} = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$ 。

在许多时候，用分块矩阵运算将会比整个矩阵计算方便的多。

Maple 在输出结果的时候，会自动地把结果中的一些子式替换成 %1, %2, 等等。

用户也可以利用函数 subs(), 根据自己的意愿进行变量替换。最简单地调用这个函数的形式是这样的：

subs(var = replacement, expression)

调用的结果是将表达式 *expression* 中所有变量 *var* 出现的地方替换成 *replacement*。例如：

```
> kinetic_energy := 1/2 * momentum^2/mass;
      kinetic_energy :=  $\frac{1}{2} \frac{\text{momentum}^2}{\text{mass}}$ 
> subs( momentum = mass*velocity, kinetic_energy );
       $\frac{1}{2} \text{mass velocity}^2$ 
> kinetic_energy;
       $\frac{1}{2} \frac{\text{momentum}^2}{\text{mass}}$ 
```

我们注意到，变量替换函数 **subs()** 只得出替换后的结果，而并不改变原表达式的内容。如果需要改变原表达式，必须再进行赋值。而且，Maple 对变量替换的结果只进行化简，而不进行求值计算，也就是说不会自动调用函数；如果需要，用户可以在变量替换之后，自己调用求值函数 **eval()**。

```
> subs( x=0, cos(x) * ( sin(x) + x^2 + 1 ) );
      cos(0) (sin(0)+1)
> eval(%);
      1
```

变量替换函数 **subs()** 也可以进行多重的变量替换。以两重为例，在 Maple 中，可以以这样的调用形式调用：

subs(var1 = replacement1, var2 = replacement2, expression)

调用的结果和按从左到右的顺序连续两次调用是一样的，也就是先将 *expression* 中的 *var1* 替换成 *replacement1*，再将其结果中的 *var2* 替换成 *replacement2*。我们把这种替换称作顺序替换；与此相对，我们还可以进行同步替换，也就是同时将 *expression* 中的 *var1* 替换成 *replacement1*，而 *var2* 则替换成 *replacement2*。同步替换的调用形式是这样的：

subs({ var1 = replacement1, var2 = replacement2 }, expression)

其中的一个参数是一个集合，用一对大括弧“{}”括起来。我们通过几个简单的例子来比较一下这两种多重替换的差异。

```
> subs( x=y, y=z, x^2*y ); # 顺序替换
      z3
> subs( { x=y, y=z }, x^2*y ); # 同步替换
      zy2
> subs( a=b, b=c, c=a, a + 2*b + 3*c );
      6a
> subs( {a=b, b=c, c=a}, a + 2*b + 3*c ); # 轮换
      b+2c+3a
> subs( { p=q, q=p }, f(p, q) ); # p q互换
      f(q,p)
```

正如我们在前面的章节中介绍过的，**subs** 不仅可以用作替换变量，还可以用来替换子式。

不过这个字是必须是符合 Maple 内部数据结构的，也就是说，必须为表达式树的一棵完整子树。相信读者经过一定的练习，在理解了 Maple 的内部数据表示之后，一定会灵活地掌握变量代换的技巧的。

除了 subs 之外，Maple 还有一些变量替换的函数，比如代数代换 (algebraic substitution) asubs()。它是专门为和式所设计的一个库函数，不依赖于 Maple 的内部数据结构，所以可以完成和式中一个或一些项的替换。

```
[> readlib(asubs):
> asubs( a + b = d, a + b + c, a );
      c+d
```

它有三个参数，第二个参数是有待替换的表达式，第三个参数是替换标志；这个函数检查表达式中的每一个和式，如果有替换标志出现，就将作为第一个参数的代数式代入到和式中去。如果第一个参数的等号左边只有一个未知变量，则可以省略替换标志。这个函数还有一个可选参数——always，如果在最后加上这个参数，例如 asubs(pat = replacement, expression, always)则 Maple 将把表达式 expression 中的每一个和式 s 都改写成为 $s - pat + replacement$ ，而不管和式中是否出现 pat。这样的代换在实际中也是经常用到的，初等代数计算或证明过程中的一个常用技巧就是在一个和式中加上“0”，也就是加上恒等式的两边之差，这就可以用上面的方法做到。

```
[> asubs( a^2 + 1 = d, b + c + 1/(b+c), always );
      b+c+-----
              1
      b+c-a^2-1+d -a^2-1+d
```

同样，我们也可以用函数 powsubs() 替换乘积中的一部分。powsubs 是 student 工具包中的一个库函数，在使用前，我们需要用 with 语句将其调入。

```
[> with( student, powsubs );
> powsubs( x^2*z = p, x^3*y*z^2 );
      yzxp
```

我们都知道，在 Maple 中，任何一个表达式都只存储一次。比如有一个符号变量 x，那么，在一个表达式中的所有出现 x 的地方都是指这个 x。如果用 subs 把表达式中的 x 替换成 y，则表达式中所有的 x 都变成了 y。这有时候并不是我们所需要的。怎样才能随我们所愿地替换表达式中的某一个子式，而同时保持这个式子中另一个相同的子式不变呢？Maple 中

还有另外一种替换方法，利用函数 subsop()，可以替换表达式中特定位置的一个元素或几个元素，这里的元素是相对于表达式的表观数据类型而言的，也就是用 op() 可以得到的元素。subsop 的调用格式是这样的：

subsop(num1 = replacement1, num2 = replacement2, expression)

它能够将表达式 expression 中第 num1 个元素替换成 replacement1，第 num2 个元素替换成 replacement2。使用这种替换方法还有一个好处，就是不须要重新输入可能非常繁琐的被替换子式，而只需用 1, 2, 3……来指代它们就可以了。我们来看一个例子：

```

[> expression := (x^2 + 2*x + 1)^2 + (x^2 - 2*x + 1)^2;
      expression := (x^2 + 2x + 1)^2 + (x^2 - 2x + 1)^2
[> subsop( 1 = factor( op(1, expression) ),
          2 = factor( op(2, expression) ), expression);
      (x+1)^4 + (x-1)^4

```

顺便提一句，上面这个例子用函数 `map()` 其实可以更简单的完成：

```

[> map( factor, expression );
      (x+1)^4 + (x-1)^4

```

函数 `map(procedure, expression)` 的作用是得到令 `expression` 中的每一个元素都分别调用子程序 `procedure`，再和成的结果。我们在后面的章节中会详细地介绍这个函数的其他用法。

初等代数运算的另一个重要技巧是把一个子式看成整体来运算，比如下面的例子中，我们希望把该表达式通分，但同时要保持 $(x+y)^2$ 不展开。显然，用 `normal` 是无能为力了。

```

[> expression := (x+y)^2 + 1/(x+y)^2;
      expression := (x+y)^2 + 1/(x+y)^2
[> normal( expression );
      (x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4 + 1)/(x+y)^2

```

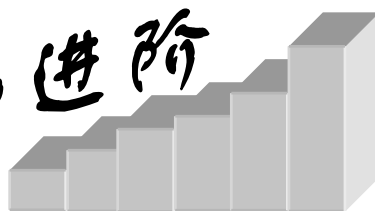
假如我们把 $x+y$ 暂时替换成一个新的变量，在进行正则化之后再将其代回，就可以达到目的了。在 Maple 中，我们可以用 `freeze` 命令将表达式暂时“冻结”，也就是替换成一个临时变量；在需要时，再用 `thaw` 命令将它“解冻”。比如上面例子的要求，就可以这样做到：

```

[> readlib( freeze );
[> subs( x+y = freeze(x+y), expression );
      freeze/R0^2 + 1/(freeze/R0^2)
[> normal(%);
      (freeze/R0^4 + 1)/(freeze/R0^2)
[> subs( `freeze/R0` = thaw(`freeze/R0`), % );
      (x+y)^4 + 1/(x+y)^2

```

起步与进阶



第

微积分运算

二

章

本章将通过例子系统地介绍 Maple 软件中的微积分运算，读者可以学到利用 Maple 软件解决简单的高等数学问题的一些方法和技巧。

本章具体包括以下内容：

- 🕒 如何在 Maple 中计算函数的极限
- 🕒 如何在 Maple 中检验函数的连续性
- 🕒 如何在 Maple 中表示微分运算
- 🕒 如何在 Maple 中进行函数和表达式的微分运算
- 🕒 如何在 Maple 中对隐函数进行微分和求导运算
- 🕒 如何在 Maple 中进行符号积分运算
- 🕒 如何在 Maple 中计算广义积分
- 🕒 如何在 Maple 中计算数值积分
- 🕒 如何在 Maple 中表示和计算数列
- 🕒 如何在 Maple 中求数列的极限
- 🕒 如何在 Maple 中将已知函数展开成级数

Maple 的应用，可以说大多数是用在高等数学的计算上了，微积分运算，也许是 Maple 最为拿手的计算了。任何解析函数，Maple 都可以求出它的导数来；任何理论上可以计算的积分，Maple 也都可以不费吹灰之力地将它计算出来。有了 Maple，你完全可以把积分手册扔到一边去，因为你在也忍受不了它了。不仅如此，Maple 从来不会抱怨表达式太繁，或者太长的。

可以毫不夸张地说，高等数学书上的任何一道计算题，都可以用 Maple 解决。不信？那好，就跟着我用 Maple 重新温习一遍微积分吧，你一定会有新的发现的！

2.1 极限和连续性

2.1.1 函数或表达式的极限

在 Maple 中，我们可以利用函数 `limit` 表示和计算函数和表达式的极限。

```
> 'limit((x^2 - 1)/(x - 1), x=1)';
```

$$\lim_{x \rightarrow 1} \frac{x^2 - 1}{x - 1}$$

```
> %;
```

$$2$$

读者一定还记得，我们用一对单引号表示暂时不作计算的表达式；上面，我们就利用它在 Maple 中写出了一个漂亮的极限式。而后面再次引用它时，Maple 就进行计算，得到了我们所期望的结果。实际上，对于这些常用的“漂亮”计算符号（又比如求导、积分等运算），Maple 中都有一套函数与其一一对应。对应的规则是，把原有函数的首字母改成大写，于是就得到“形式函数”，得到的是一个形式上的表达式。比如上面这个例子，我们就可以写成：

```
> Limit((x^2 - 1)/(x - 1), x=1) =
```

$$\lim_{x \rightarrow 1} \frac{x^2 - 1}{x - 1} = 2$$

```
limit((x^2 - 1)/(x - 1), x=1);
```

顺理成章地，这个函数也可用来求自变量趋于无穷时的极限。无穷，在 Maple 中用 `infinity` 表示。我们来看下面这个经典的极限：

```
> Limit((1 + 1/x)^x, x=infinity) =
```

$$\lim_{x \rightarrow \infty} \left(1 + \frac{1}{x}\right)^x = e$$

```
limit((1 + 1/x)^x, x=infinity);
```

为了使大多数计算能够进行下去，函数 `limit` 假设表达式中所有未被赋值的参数都是非 0 实数。比如在 `a` 未被赋值时， a^2/x 在 x 趋向于 0 时的极限将被认为是正无穷大。

函数的第二个参数表示欲求的极限所在的位置，它是一个等式，等式的左边是自变量，右边是极限点，极限点可以是任意的实数。基于 Maple 的强大符号运算功能，表达式中间完

全可以包含未知参数，绝大多数理论上存在的极限都可以求出来。

```
> Limit({x^n - 1}/{x-1}, x=1) = limit({x^n - 1}/{x-1},
x=1);
```

$$\lim_{x \rightarrow 1} \frac{x^n - 1}{x - 1} = n$$

```
> Limit(({x + ln(x)}/{x - ln(x)})^(x/ln(x)),
x=infinity) = limit(({x + ln(x)}/{x -
ln(x)})^(x/ln(x)), x=infinity);
```

$$\lim_{x \rightarrow \infty} \left(\frac{x + \ln(x)}{x - \ln(x)} \right)^{\left(\frac{x}{\ln(x)} \right)} = e^2$$

该函数不仅可以用来求变量函数的极限，还可以用来求多重极限。这时，函数的第二个参数是一个等式的集合（用一对大括弧“{ }”括起来）。例如：

```
> limit(x+1/y, {x=0, y=infinity});
```

$$0$$

limit 函数的第三个参数是可选参数，利用它可以求单侧极限和复数域极限。在默认情况下，函数求得的是实数域中的双侧极限（除了无穷大处的极限是单侧的外）。如果指定第三个参数为 complex，则函数 limit 在复数域中求极限。在实数域中，我们可以指定 left 或 right，以求得单侧极限。例如：

```
> Limit(x^x, x=0, right) = limit(x^x, x=0, right);
```

$$\lim_{x \rightarrow 0+} x^x = 1$$

2.1.2 函数的连续性

在 Maple 中，你可以用库函数 iscont() 来检验一个函数或者表达式的连续性。由于它是库函数，使用前我们先要用命令 readlib 调入。请看下面的例子：

```
> readlib(iscont):
iscont( 1/x, x=1..2 );
```

$$\text{true}$$

```
> iscont( 1/x, x=-1..1 );
```

$$\text{false}$$

其中第二个参数指定了有待检验的区间，它必须由两个实的常数（或无穷大）界定。默认情况下它指的是一个开区间，在指定了第三个（可选）参数为 'closed' 后，它将检查该闭区间。如果无法判断表达式在该区间上的连续性，函数将返回系统符号常量 FAIL，表示计算失败。

相应的，Maple 中还有另一个库函数 discont()，可以用来找出表达式或函数的间断点。这个函数可以找出所有可能的实间断点，依据 Maple 的算法，它找到的并不一定都是间断点，但一般情况下，也就是函数比较好的情况下，找到的都是真正的间断点。我们会经常遇到间断点周期出现或成对出现的情况，这时，Maple 会利用一些辅助变量予以表达，比如_Zn~、

_NNn~、和_Bn~, 其中 n 是序号, _Zn~表示任意整数, _NNn~表示任意自然数, 而_Bn~则表示一个二进制数 (即可以取 0 或者 1)。

```
> readlib(discont):
   discnt(ln(sin(x)), x);
                                     {π_Zl~}
> discnt({1/x - 1/(x+1)}/{1/(x-1) - 1/x}, x);
                                     {0, -1}
```

利用函数 fdiscont(), 我们可以求得数值上的间断点, 和其他浮点运算一样, 浮点精度由系统变量 Digits 决定。

2.2 Maple 中的求导和微分运算

2.2.1 符号表达式求导

利用 Maple 中的求导函数 diff(), 你可以计算一个表达式的导数或者偏导数; 而利用形式求导函数 Diff(), 你可以获得求导表达式。

```
> Diff(cos(x)/x^4 * ln(1/x), x) = diff(cos(x)/x^4 *
   ln(1/x), x);
```

$$\frac{\partial}{\partial x} \frac{\cos(x) \ln\left(\frac{1}{x}\right)}{x^4} = -\frac{\sin(x) \ln\left(\frac{1}{x}\right)}{x^4} - 4 \frac{\cos(x) \ln\left(\frac{1}{x}\right)}{x^5} - \frac{\cos(x)}{x^5}$$

利用符号\$可以简单地表示多重导数, diff(expr, x\$3)和 diff(expr, x, x, x)是等价的, 它们都表示 expr 对 x 的 3 阶导数。

```
> Diff(exp(x^2), x$5) = diff(exp(x^2), x$5);
```

$$\frac{\partial^5}{\partial x^5} e^{(x^2)} = 120 x e^{(x^2)} + 160 x^3 e^{(x^2)} + 32 x^5 e^{(x^2)}$$

由于 Maple 是一个符号计算软件, 而且在不加特别约束的情况下, 带参数的导数实质上就是偏导数, 所以用 diff()计算偏导数和计算单变量函数的导数在形式上没有任何不同:

```
> Diff(arcsin(x/sqrt(x^2+y^2)), x) =
   diff(arcsin(x/sqrt(x^2+y^2)), x);
```

$$\frac{\partial}{\partial x} \arcsin\left(\frac{x}{\sqrt{x^2+y^2}}\right) = \frac{\frac{1}{\sqrt{x^2+y^2}} - \frac{x^2}{(x^2+y^2)^{\frac{3}{2}}}}{\sqrt{1 - \frac{x^2}{x^2+y^2}}}$$

```
> Diff( sin(x+y)/y^3, x$5, y$2 ) = diff( sin(x+y)/y^3,
      x$5, y$2 );
```

$$\frac{\partial^7}{\partial y^2 \partial x^5} \frac{\sin(x+y)}{y^3} = -\frac{\cos(x+y)}{y^3} + 6 \frac{\sin(x+y)}{y^4} + 12 \frac{\cos(x+y)}{y^5}$$

函数 diff()求得的结果总是一个表达式，如果你需要得到一个函数形式的结果，也就是要求导函数，你可以用 D 运算符。D 运算符作用于一个函数上，得到的结果也是一个函数。我们在这里先用箭头运算符“->”定义一个简单的函数，箭头运算符的左边是函数的自变量，右边是函数表达式。有关函数的定义和箭头运算符的详细情况，我们在后面相应章节会加以详细介绍。

```
> g := x->x^n * exp(sin(x));
```

$$g := x \rightarrow x^n e^{\sin(x)}$$

```
> D(g);
```

$$x \rightarrow \frac{x^n n e^{\sin(x)}}{x} + x^n \cos(x) e^{\sin(x)}$$

```
> Diff(g, x)(Pi/6) = D(g)(Pi/6);
```

$$\left(\frac{\partial}{\partial x} g\right)\left(\frac{1}{6}\pi\right) = 6 \frac{\left(\frac{1}{6}\pi\right)^n n e^{\left(\frac{1}{2}\right)}}{\pi} + \frac{1}{2} \left(\frac{1}{6}\pi\right)^n \sqrt{3} e^{\left(\frac{1}{2}\right)}$$

D 运算符也可以被用作求多重导数，不过这里不是用“\$”而是用两个连续的“@@”。

```
> (D@@2)(cos);
```

$$-\cos$$

D 运算符并不局限于单变量函数，一个带指标的 D 运算符 D[i](f)可以用来求偏导函数。D[i](f)表示函数 f 对第 i 个变量的导函数，而多重导数 D[i, j](f)等价于 D[i](D[j](f))。

```
> f := (x, y, z) -> (x/y)^(1/z);
```

$$f := (x, y, z) \rightarrow \left(\frac{x}{y}\right)^{\left(\frac{1}{z}\right)}$$

```
> Diff(f, y)(1, 1, 1) = D[2](f)(1, 1, 1);
```

$$\left(\frac{\partial}{\partial y} f\right)(1, 1, 1) = -1$$

由于 diff 和 D 这两种运算本质上是一样的，所不同的仅仅是表达形式而已，它们之间也可以用 convert 相互转换。

```
> f := D(y)(x) - a*D(z)(x);
      convert(f, diff, x);
```

$$\left(\frac{\partial}{\partial x} y(x)\right) - a \left(\frac{\partial}{\partial x} z(x)\right)$$

```
> f := diff(y(x), x$2);
      convert(f, D);
```

$$(D^{(2)})(y)(x)$$

2.2.2 隐函数的导数

很多情况下,函数并不能写成显式的解析表达式,而只能通过自变量和函数的方程给出他们之间的关系;有时虽然可以写出显式表达式,但形式上要麻烦的多,所以在高等数学中,我们有直接对隐函数的求导方法。在 Maple 中也有这样的函数——`implicitdiff()`。

```
> implicitdiff( y = x^2/z, y, z );
```

$$-\frac{x^2}{z^2}$$

第一个参数是蕴含着函数关系的方程,它也可以是一个方程组(用一对花括弧扩起来的等式的集合)。第二个参数是函数(应变量),第三个参数是求导的变量,它们也都可以是变量的集合。和 `diff` 中一样,这个函数也可以用来求高阶导数,这时需要提供多个求导变量作为参数,形式和 `diff` 中也一样。

有时隐函数表达式中含有一些字母表示的参数,它们并不是函数的自变量,为了将它们和自变量分开,我们可以人为地加入第二个参数——函数及其自变量的显式形式,比如 $y(x_1, \dots, x_n)$,这样就确定了函数和它的自变量,而方程中其他未知变量就会被认为是常数了。需要注意的是,如果给出函数,必须给出原方程组适定的函数,也就是说从原方程组中可以解出这些函数来。比如下面的例 2.1 中,必须给出 $\{z(x, y), \psi(x, y), \phi(x, y)\}$,若只给出 $z(x, y)$ 是不够的。方程中的变量,除了自变量和函数,其他的都会被看成常数。

如果是对多变量函数求多个偏导数,结果将用偏微分形式给出。我们可以给定最后一个可选参数,来确定结果的表达形式。默认情况下,或者我们给定 `notation = D`,这时结果中的微分用 `D` 运算符表示;否则,我们可以给定 `notation = Diff`,这样给出的结果中的微分运算符和使用 `Diff()`时的相同,也就是用“ ∂ ”来表示。请看下面的例子:

例 2.1 设

$$\begin{cases} x = \cos \varphi \cos \psi \\ y = \cos \varphi \sin \psi, \text{ 求 } \frac{\partial^2 z}{\partial x^2} \\ z = \sin \varphi \end{cases}$$

```
> f := x = cos(phi)*cos(psi);
      f:=x=cos(phi)cos(psi)
> g := y = cos(phi)*sin(psi);
      g:=y=cos(phi)sin(psi)
> h := z = sin(psi);
      h:=z=sin(psi)
> implicitdiff({f, g, h}, {z(x,y), psi(x,y), phi(x,y)},
      {z}, x, x, notation=Diff);
```

$$\left\{ \left(\frac{\partial^2}{\partial x^2} z \right)_y = \frac{\sin(\psi) (-2 + 3 \sin(\psi)^2)}{\sin(\phi)^2 - 1} \right\}$$

例 2.2 设函数 $u = u(x)$ 由方程组

$$u = f(x, y, z), g(x, y, z) = 0, h(x, y, z) = 0$$

定义。求 $\frac{du}{dx}$

```
> eqs := { u=f(x,y,z), g(x,y,z)=0, h(x,y,z)=0 };
          eqs := {u=f(x,y,z), g(x,y,z)=0, h(x,y,z)=0}
> implicitdiff( eqs, {u(x), y(x), z(x)}, u, x );
(D1(f)(x,y,z) D2(g)(x,y,z) D3(h)(x,y,z)
- D1(f)(x,y,z) D3(g)(x,y,z) D2(h)(x,y,z)
+ D2(f)(x,y,z) D1(h)(x,y,z) D3(g)(x,y,z)
- D2(f)(x,y,z) D3(h)(x,y,z) D1(g)(x,y,z)
+ D3(f)(x,y,z) D1(g)(x,y,z) D2(h)(x,y,z)
- D3(f)(x,y,z) D2(g)(x,y,z) D1(h)(x,y,z)) / (
D2(g)(x,y,z) D3(h)(x,y,z) - D3(g)(x,y,z) D2(h)(x,y,z))
```

2.3 积分运算

2.3.1 不定积分

Maple 具有众多的内建积分算法。首先，和我们在大学数学课上学的一样，它会试着使用一些常用方法，比如查表、分部积分、变量代换等等；在这些方法统统失败之后，系统会尝试其他一些特殊方法，比如所谓的 Risch 算法等。

让我们先来看一个用 Maple 中的函数 `int()` 求不定积分的简单例子函数的第一个参数是被积表达式，第二个是积分变量：

```
> Int( (x+1)/sqrt(x), x ) = int( (x+1)/sqrt(x), x );
      
$$\int \frac{x+1}{\sqrt{x}} dx = \frac{2}{3} x^{\frac{3}{2}} + 2\sqrt{x}$$

```

我们可以注意到，和我们在数学课上的要求不同，Maple 求出的不定积分没有积分常数。这是有一定用处的，尤其当我们还有对结果作进一步处理时，由于 Maple 的符号计算特性，引入积分常数相当于引入了一个变量，对于计算会带来不便。但这毕竟和我们通常意义上的不定积分有着一定的距离，它把一族函数变成了一个函数，使用是需要引起注意。好在一般情况下，这对于我们的要求没有多大的影响。

在上面的例子中，我们还用了和求导运算中一样的伎俩，用形式函数 `Int()` 获得了一个漂亮的积分表达式。

在高等数学课上，我们学习过分式的积分方法，即把分式分解成最简真分式的和，再分别求积分。但是这种方法在处理一部分有理函数的时候难免会出现危机，比如在分母不能简单的进行因式分解的时候。下面，我们简单描述一下 Maple 中不定积分的计算过程：

- ✧ 在第一阶段中, Maple 用传统方法处理一些特殊的形式, 有如多项式、有理式、形如 $\left(\sqrt{a+bx+cx^2}\right)^n$ 和 $Q(x)\left(\sqrt{a+bx+cx^2}\right)^n$ 的根式, 以及形如 $(x \text{ 的多项式}) \times \ln x$ 或 $(x \text{ 的有理式}) \times \ln(x \text{ 的有理式})$ 的表达式。
- ✧ 如果传统方法难以奏效, Maple 将应用 Risch-Norman 算法, 以避免在包含三角函数和双曲函数的积分中引入复指数和对数。
- ✧ 如果仍然无法得到答案, Maple 将采用 Risch 算法。这将无法避免地在结果表达式中引入有关积分变量的 RootOf 的表达式。
- ✧ 如果最终还是没有找到解析表达式, Maple 会把积分式作为结果返回。

由于本书的目的在于介绍 Maple 软件的使用, 对于算法将不予详细描述。如果读者对于上面提到的两种算法感兴趣, 可以参考下列相关文献:

K.O. Geddes and L.Y. Stefanus, On the Risch-Norman integration method and its implementation in Maple, In: G.H. Gonnet (ed.), Proceedings of ISSAC '89, ACM Press, New York, 1989, pp. 212-217.

R.H. Risch, The problem of integration in finite terms, Trans. AMS 139 (1969), 167-189.

M. Bronstein, Integration of Elementary Functions, J. Symbolic Computation 9 (1990), 117-174.

2.3.2 定积分

在 Maple 中, 计算定积分也是一样是用函数 `int`, 或者它的别称 `integrate`; 同样, 它的形式函数 `Int` 也可以用来表示定积分。和不定积分一样, 函数 `int` 的第一个参数是被积函数, 第二个是由积分变量和积分区间构成的等式。

$$\begin{aligned} & \text{[> Int(1/(1+x^2), x = -1..1) = int(1/(1+x^2), x} \\ & \quad \text{= -1..1);} \\ & \qquad \qquad \qquad \int_{-1}^1 \frac{1}{1+x^2} dx = \frac{1}{2} \pi \end{aligned}$$

定积分是怎么完成的呢? 一个简单但是并不严格的想法是: 先求初步定积分, 再将积分上下界带入求得定积分。但 Maple 没有这么笨, 它知道这么做会带来错误的。用一个简单的例子就可以说明这一点。我们来求 $1/x^2$ 的积分。

$$\begin{aligned} & \text{[> Int(1/x^2, x) = int(1/x^2, x);} \\ & \qquad \qquad \qquad \int \frac{1}{x^2} dx = -\frac{1}{x} \\ & \text{[> subs(x=1, rhs(%)) - subs(x=-1, rhs(%));} \\ & \qquad \qquad \qquad -2 \end{aligned}$$

这样的方法得出了可笑的结果。问题出在这里：在积分区间 $(-1, 1)$ 内有一个不可去间断点—— $x = 0$ 。在这个例子中，我们还用到了一个函数 `rhs` 来获得等式的右边部分（**right hand side**）。对于这个例子，Maple 知道它的结果是发散的，因此，它得出这样的令人信服的结果：

```
> Int( 1/x^2, x=-1..1 ) = int( 1/x^2, x=-1..1 );
```

$$\int_{-1}^1 \frac{1}{x^2} dx = \infty$$

在大多数情况下，Maple 通过查表和形式匹配，或者利用特殊函数的导数来求定积分。

```
> Int( exp(-sqrt(t)) / ( t^(1/4) * (1-exp(-sqrt(t))) ),
      t = 0..infinity ): % = value(%);
```

$$\int_0^{\infty} \frac{e^{(-\sqrt{t})}}{t^{\left(\frac{1}{4}\right)} (1 - e^{(-\sqrt{t})})} dt = \sqrt{\pi} \zeta\left(\frac{3}{2}\right)$$

```
> evalf(rhs(%));
```

4.630314748

椭圆积分是形如 $\int_a^b R(x, y^{1/2}) dx$ 的积分，其中 R 是有理函数，而 y 是 x 的 3 或 4 阶多项式。这是椭圆积分的代数形式，除此之外，椭圆积分还有三角形式（ \sin 和 \cos 的有理函数与其 4 阶多项式）和双曲三角形式。在 Maple 中，椭圆积分被化成为 Legendre 形式的椭圆函数：EllipticF, EllipticE 和 EllipticPi，以及它们的组合。

```
> Int( 1/sqrt((x^2-1)*(x^2-2)), x = a..b ):
> % = value(%);
```

$$\int_a^b \frac{1}{\sqrt{(x^2-1)(x^2-2)}} dx = -\frac{1}{2} \left(\frac{-\sqrt{1-b^2} \sqrt{-2b^2+4} \operatorname{EllipticF}\left(b, \frac{1}{2}\sqrt{2}\right) \sqrt{a^4-3a^2+2} + \sqrt{1-a^2} \sqrt{-2a^2+4} \operatorname{EllipticF}\left(a, \frac{1}{2}\sqrt{2}\right) \sqrt{b^4-3b^2+2}}{\sqrt{b^4-3b^2+2} \sqrt{a^4-3a^2+2}} \right) / ($$

2.3.3 数值积分

作为符号演算软件，目的是得到精确的解析结果；但作为一个实用的数学工具，Maple 同时具有许多数值处理的功能，数值积分也是其一。和 Maple 中其他函数和表达式的数值计

算一样，可以用 evalf 获得积分的数值结果。

```
[> int( exp( arcsin(x) ), x = 0..1 );
```

$$\int_0^1 e^{\arcsin(x)} dx$$

```
> evalf(%);
```

$$1.905238690$$

Maple 也可以求广义积分的数值；而且，和其他数值计算一样，我们可以指定结果的浮点精度。比如下面的例子：

```
[> Int( exp(-2*t) * t * ln(t), t = 0..infinity );
```

$$\int_0^{\infty} e^{-2t} t \ln(t) dt = -\frac{1}{4} \ln(2) + \frac{1}{4} - \frac{1}{4} \gamma$$

```
> % = value(%);
```

$$-0.6759071136536954251$$

Maple 中默认的数值积分方法是 Clenshaw-Curtis 四阶方法；当收敛很慢（由于存在奇点）时，系统将试着用广义的级数展开和变量代换取消积分的奇异性；如果存在不可去奇点，则改而采用自适应双指数方法。在数值精度要求不高的情况下（比如 Digits ≤ 15），采用自适应的牛顿-柯特斯方法就够了。通过指定 evalf/int 的第 4 个参数，可以选择积分方法。可供选择的是三种方法：_CCquad——Clenshaw-Curtis 四阶方法；_Dexp——自适应双指数法；_NCrue——牛顿-柯特斯方法，如下例：

```
[> evalf(Int(1/sqrt(x), x = 0..2, 15, _Dexp));
```

$$2.82842712474619$$

2.3.4 重积分和线积分

在 Maple 中，有着重积分的形式函数——Doubleint() 和 Trippleint()，它们在 student 工具包中，适用前需要先予以包含。

```
[> with(student):
```

$$\iint f(x,y) dx dy$$

它们适用于定积分和不定积分，我们也可以用 value 来获得积分结果的解析表达式。

```
[> Doubleint( x+y, y=1..exp(x), x=0..1 );
```

$$\int_0^1 \int_1^{e^x} (x+y) dy dx = -\frac{1}{4} + \frac{1}{4} (e)^2$$

```
> % = value(%);
```

在这两个形式函数中，我们还可以加入一个可选的参数，用来表示积分区域（比如我们通常用 S 表示二维区域，用 Ω 表示三维区域），如下例所示。需要注意的是，在 Maple 中，

这个参数仅仅用来做形式上的表示，不可以用来求值。

```
> Tripleint( x^2*y^2*z^2, x, y, z, Omega );
```

$$\iiint_{\Omega} x^2 y^2 z^2 dx dy dz$$

在 Maple 中还有一个可以计算用参数方程形式表示的第一型曲线积分的函数——Lineint()，它也在 student 工具包中。我们通过一个例子来说明这个函数的用法。

例 2.3 计算曲线积分 $\int_C y^2 ds$ ，其中 C 为摆线的一拱：

$$x = a(t - \sin t), y = a(1 - \cos t), 0 \leq t \leq 2\pi$$

```
> with (student):
Lineint( y^2, x = a*(t-sin(t)), y = a*(1-cos(t)),
t = 0..2*Pi );
```

$$\int_0^{2\pi} a^2 (1 - \cos(t))^2 \sqrt{\left(\frac{\partial}{\partial t} a(1 - \cos(t))\right)^2 + \left(\frac{\partial}{\partial t} a(t - \sin(t))\right)^2} dt$$

```
> value(%);
```

$$\frac{128}{15} \sqrt{a^2} a^2 \sqrt{4}$$

可以看到，Lineint 自动地把曲线积分转化成为一般积分的形式。而且，它是一个形式函数，需要用 value 来求值。上面例子中用的是曲线的标准参数方程形式。有时候，曲线往往可以用其他形式表示，这时，调用该函数将没有最后的参变量 t 及其范围；取而代之，将以调用时的最后一个参数为准，其他的参数将作为最后一个参数的函数看待。

2.3.5 利用辅助手段积分

机器终归是机器，再聪明的机器也无论如何替代不了人脑。Maple 也一样，它是我们的得力帮手，但我们不能完全依赖它，只有将它和我们的大脑有机地结合起来，才有可能所向披靡，解决各种问题。这里，我们将看到一些 Maple 不能完全自行解决的积分问题，而我们采取一些辅助的手段，问题就会迎刃而解。这种情况不仅仅局限于积分问题中，实际的问题往往都是如此。通过这些问题，也许您能够体会到 Maple 的真正用处。

第一个问题牵涉到一个满足一定条件的参数，有了这些条件，我们就可以求出积分的解析表达式来。我们的问题是这样的：

例 2.4 求广义积分：

$$\int_0^{\infty} e^{-cx^2} dx, \text{ 其中 } c > 0$$


```
[> Int( exp(-c*x^2), x = 0..infinity ):
> % = value(%);
Definite integration: Can't determine if the integral is convergent.
Need to know the sign of --> c
Will now try indefinite integration and then take limits.
```

$$\int_0^{\infty} e^{(-c x^2)} dx = \lim_{x \rightarrow \infty} \frac{1}{2} \frac{\sqrt{\pi} \operatorname{erf}(\sqrt{c} x)}{\sqrt{c}}$$

这是一个简单的概率积分，但 Maple 却告诉我们，无法确定积分是否收敛，原因是无法确定参数 c 的正负号，因此没有得到我们想要的结果。我们可以通过 `assume` 设定 c 的取值范围，问题迎刃而解：

```
[> assume( c>0 ):
> Int( exp(-c*x^2), x = 0..infinity ): % = value(%);
```

$$\int_0^{\infty} e^{(-c x^2)} dx = \frac{1}{2} \frac{\sqrt{\pi}}{\sqrt{c}}$$

解决这个问题的另一手段是假设 c 是另一个参数 p 的绝对值，这样 c 就自然是一个非负的参数了。

```
[> c := abs(p):
> Int( exp(-c*x^2), x = 0..infinity ): % = value(%);
```

$$\int_0^{\infty} e^{(-|p| x^2)} dx = \frac{1}{2} \frac{\sqrt{\pi}}{\sqrt{|p|}}$$

有时候，为了得到需要的结果，可以指定一种积分方法，比如常用的换元法和分部积分法(**integration by parts**)。这两种方法可以分别通过调用 `student` 工具包中的函数 `changevar()` 和 `intparts()` 来完成。换元法函数的调用格式是 **changevar**(s, f, u)，其中 s 是形如 $h(x) = g(u)$ 的等式，定义 x 为 u 的函数， f 是一个积分式（可以用 `Int()` 获得）， u （可省）是新的积分变量。分部积分的调用格式为 **intparts**(f, u)，其中 f 是积分表达式， u 是被积函数中有待求导的部分；其调用的结果为： $u*v - \operatorname{Int}(du*v, x)$ ，如下面的例子所示。

```
[> with(student):
> int( x*exp(-a^2*x^2)*erf(b*x), x );
```

$$\int x e^{(-a^2 x^2)} \operatorname{erf}(b x) dx$$

```
[> intparts( %, erf(b*x) );
```

$$-\frac{1}{2} \frac{\operatorname{erf}(b x) e^{(-a^2 x^2)}}{a^2} - \int -\frac{e^{(-b^2 x^2)} b e^{(-a^2 x^2)}}{\sqrt{\pi} a^2} dx$$

```
> value(%);
```

$$-\frac{1}{2} \frac{\operatorname{erf}(bx) e^{(-a^2 x^2)}}{a^2} + \frac{1}{2} \frac{b \operatorname{erf}(\sqrt{b^2 + a^2} x)}{a^2 \sqrt{b^2 + a^2}}$$

经过了这一段时间对 Maple 的使用，读者朋友难免会生出一个不太确切的观念，认为 Maple 只能解决数学计算题，而对于令我们头疼的证明题无能为力。为了消除这个错觉，下面我们利用 Maple 来解决一个高等数学或者是复变函数中的证明题。当然，证明的步骤是需要我们人为决定的，Maple 决不会给出它的计算过程的（即使给出了，对于我们也是毫无作用，应为它和我们的“思维方式”有着天壤之别，没有人会容忍它这种“穷举法”的）。

例 2.5 求证

$$\int_0^{2\pi} \frac{1}{(1+3\sin t)^2} dt = \pi$$

学过复变函数的读者一定知道，这个积分可以用围道积分（contour integration）来求解。首先，我们需要把被积函数写成为复变量 $z = e^{it}$ 的有理式。

```
> 1 / ( 1 + 3*sin(t)^2 );
```

$$\frac{1}{1+3\sin(t)^2}$$

```
> convert( %, exp );
```

$$\frac{1}{1-\frac{3}{4}\left(e^{(It)}-\frac{1}{e^{(It)}}\right)^2}$$

这样，题目中的积分就转化成为复平面单位圆上的围道积分，被积函数是：

```
> factor(%) / diff(exp(I*t), t):
```

```
> g := subs(exp(I*t)=z, %);
```

$$g := 4 \frac{Iz}{(3z^2-1)(z^2-3)}$$

我们知道这个积分的结果是 $2\pi i \Sigma$ (单位圆中的留数)，所以，我们先计算上面这个式子的奇点，即分母的零点，再计算它们上的留数。

```
> solve( denom(g), z );
```

$$\frac{1}{3}\sqrt{3}, -\frac{1}{3}\sqrt{3}, \sqrt{3}, -\sqrt{3}$$

```
> readlib(residue):
```

```
> residue( g, z=1/3*sqrt(3) );
```

$$-\frac{1}{4}I$$

```
> residue( g, z=-1/3*sqrt(3) );
```

$$-\frac{1}{4}I$$

```
> 2*Pi*I * ( % + %% );
```

$$\pi$$

我们利用库函数 `residue()` 计算单位圆中的留数。

这样，我们就在 Maple 的帮助下，证明了这个积分式。

2.4 级数

2.4.1 数值级数和函数项级数求和

我们可以利用 Maple 的函数 `sum()` 方便地求得级数的和，无论是有限项还是无穷项，数值级数还是函数项级数；相应地，和式的形式函数是 `Sum()`。

```
[> Sum( 1 / (4*k^2-1), k=1..infinity ):
> % = value(%);
```

$$\sum_{k=1}^{\infty} \frac{1}{4k^2-1} = \frac{1}{2}$$

```
[> Sum( 1/k/(k+1), k=1..n ):
> % = value(%);
```

$$\sum_{k=1}^n \frac{1}{k(k+1)} = -\frac{1}{n+1} + 1$$

Maple 是用如下的方法求级数和的：

◇ 多项式级数求和用的是伯努里级数公式：

$$\sum_{k=0}^{n-1} k^m = \frac{1}{m+1} \sum_{k=0}^m \binom{m+1}{k} B_k n^{m+1-k}$$

其中伯努里数 B_k 是由一个隐式递推公式定义的：

$$B_0 = 1, \quad \sum_{k=0}^m \binom{m+1}{k} B_k = 0$$

- ◇ 有理函数级数求和用的是 Moenck 方法，得到的结果是一个有理函数加上一些关于多伽玛函数（Polygamma function） Ψ 及其导数的项。
- ◇ Gosper 算法是 Risch 算法的离散形式，它被用到求包含级乘和乘幂的级数和上。
- ◇ 计算无穷项级数的和有时会用到超比级数。

2.4.2 幂级数

幂级数是在数学分析中最为常用的级数了，在 Maple 中，有一个专门的幂级数工具包——`powseries`。这个工具包中有着生成和处理幂级数的各种常用工具。

如果我们已知一个幂级数的系数，我们可以用函数 `powcreate()` 来生成它。函数 `powcreate()` 的参数是系数所满足的方程（或方程组）。比如 `powcreate(t(n)=expr, t(0)=epr0,`

$t(1)=\text{expr1}, \dots, t(m)=\text{exprm}$), 将得到一个幂级数, 其系数的通项公式是 expr (可以是递推公式), 而 $\text{expr1}, \dots, \text{exprm}$ 是幂级数的初始项或特殊项, 这些项并没有必要满足通项公式 expr 。我们通过例子来看看它的用法:

```
> with(powseries):
   powcreate(t(n)=3^sqrt(n));
```

结果似乎出于我们的意料, 实际上, `powcreate` 的结果是把该幂级数的系数函数赋给了 `t`, 现在 `t` 已经是一个在非负整数域中取值的函数了:

```
> t(2);
```

$$3^{(\sqrt{2})}$$

当然, 这样的级数并不太直观, 我们也许需要一个截断的幂级数表达式 (**truncated power series form**), 我们可以用该工具包中的另一个函数 `tpsform()`:

```
> tpsform(t, x, 5);
```

$$1 + 3x + 3^{(\sqrt{2})}x^2 + 3^{(\sqrt{3})}x^3 + 3^{(\sqrt{4})}x^4 + O(x^5)$$

这个调用把该幂级数展开到低于 x 的 5 次幂的截断形式。

但大多数情况下, 我们并不知道幂级数的系数, 而只知道幂级数的和的解析表达式, 需要把它展开成和式。对于一些常用的函数, `powseries` 工具包中有一些函数可以生成对应的幂级数, 比如 `sin(p)`, `cos(p)`, `exp(p)`, `ln(p)`, `sqrt(p)` 的幂级数可以分别通过 `powsin(p)`, `powcos(p)`, `powexp(p)`, `powlog(p)`, `powsqrt(p)` 来得到, 其中 p 可以是单变量函数或表达式, 甚至级数。例如 $\ln(1+x+x^2)$ 的幂级数可以这样得到:

```
> t := powlog(1+x+x^2): tpsform(t, x, 5);
```

$$x + \frac{1}{2}x^2 - \frac{2}{3}x^3 + \frac{1}{4}x^4 + O(x^5)$$

对于多项式, 我们可以用 `powpoly(expr, x)` 得到对应的幂级数, 其中 expr 是多项式, x 是幂级数的变量。实际上, 对于任意的函数或表达式, 我们都可以用函数 `evalpow()` 来获得其幂级数形式。

```
> t := evalpow(1/(1-3*x+2*x^2)):
> tpsform(t, x, 5);
```

$$1 + 3x + 7x^2 + 15x^3 + 31x^4 + O(x^5)$$

现在, 我们可以对这些已经生成的级数进行操作了, `powseries` 工具包中具有对幂级数的这些运算: 相加, 相减, 求负, 相乘, 相除, 求倒数, 复合运算, 求逆运算, 求导, 积分; 对应的函数依次为: `powadd()`, `subtract()`, `negative()`, `multiply()`, `quotient()`, `inverse()`, `compose()`, `reversion()`, `powdiff()`, `powint()`。这些函数的用法都很简单, 我们将通过一些例子来学习它们。

首先, 我们通过系数定义两个级数:

```
> powcreate(f(n)=a^n/n!):
> powcreate(g(n)=(-1)^(n+1)/n, g(0)=0):
```

再来求它们的和与差:

```

[> s := powadd(f, g): tpsform(s, x, 4);
      
$$1 + (a+1)x + \left(\frac{1}{2}a^2 - \frac{1}{2}\right)x^2 + \left(\frac{1}{6}a^3 + \frac{1}{3}\right)x^3 + O(x^4)$$

[> d := subtract(f, g): tpsform(d, x, 4);
      
$$1 + (a-1)x + \left(\frac{1}{2}a^2 + \frac{1}{2}\right)x^2 + \left(\frac{1}{6}a^3 - \frac{1}{3}\right)x^3 + O(x^4)$$


```

接着, 我们来求它们级数形式的积与商:

```

[> p := multiply(f, g): tpsform(p, x, 4);
      
$$x + \left(-\frac{1}{2} + a\right)x^2 + \left(\frac{1}{3} - \frac{1}{2}a + \frac{1}{2}a^2\right)x^3 + O(x^4)$$

[> q := quotient(g, f): tpsform(q, x, 4);
      
$$x + \left(-\frac{1}{2} - a\right)x^2 + \left(\frac{1}{3} - a\left(-\frac{1}{2} - a\right) - \frac{1}{2}a^2\right)x^3 + O(x^4)$$


```

我们也可以方便地求得幂级数的导数和积分:

```

[> d := powdiff(f): tpsform(d, x, 4);
      
$$a + a^2x + \frac{1}{2}a^3x^2 + \frac{1}{6}a^4x^3 + O(x^4)$$

[> i := powint(f): tpsform(i, x, 4);
      
$$x + \frac{1}{2}ax^2 + \frac{1}{6}a^2x^3 + O(x^4)$$


```

我们也可以将两个级数进行复合运算和求逆运算。注意, 这里的复合运算指的是幂级数对应的函数之间的运算, 而不是指系数函数 (上面的 $f(n)$ 、 $g(n)$) 之间的复合。如果它们对应的函数分别是 $F(x)$ 、 $G(x)$, 则 $\text{compose}(f, g)$ 得到的结果是复合函数 $F(G(x))$ 的幂级数。

```

[> c := compose(f, g): tpsform(c, x, 4);
      
$$1 + ax + \left(-\frac{1}{2}a + \frac{1}{2}a^2\right)x^2 + \left(\frac{1}{3}a - \frac{1}{2}a^2 + \frac{1}{6}a^3\right)x^3 + O(x^4)$$


```

函数 $\text{reversion}()$ 是 $\text{compose}()$ 的逆运算, 也就是已知 $F(x)$ 和 $F(G(x))$ 的级数形式, 求 $G(x)$ 的级数的运算。但这样的 $G(x)$ 一般情况下是不确定的, $\text{reversion}(a, b)$ 只有在 $a(0) = 0$, $a(1) = 1$ 而且 $b(0) = 0$ 时才可以得到结果, 否则系统会给出出错信息。reversion 的第二个参数为可选参数, 在省略时, 默认为恒等运算, 这时也就是求 a 对应函数的逆函数的幂级数:

```

[> r := reversion(g): tpsform(r, x, 4);
      
$$x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + O(x^4)$$

[> c := compose(g, r): tpsform(c, x, 4);
      
$$x + O(x^4)$$


```

2.4.3 泰勒级数和劳朗级数

在上一段中, 我们介绍了 Maple 中的幂级数。我们已经知道, 幂级数在 Maple 中是用系数函数表示的, 也就是说, 只有系数可以表示成为次数的函数的级数, 才可以用这种方法表示。这种方法有一定的优越性, 比如精确性——没有截断误差; 但同时也有着它的弊端——必然有一些复杂的级数不能够表示, 而且不直观。于是, 我们用 tpsform 将它转化成为截

断级数形式。那么，Maple 中究竟有没有一种方法可以直接得到阶段形式的级数呢？回答是肯定的。

我们先来看看我们所熟悉的泰勒级数(**Taylor series**)。在 Maple 中，你可以用函数 `taylor()` 方便快捷地得到一个函数或表达式在一点的任意阶展开式。作为例子，我们来计算 $\sin(\tan x) - \tan(\sin x)$ 在 $x = 0$ 点的泰勒展开式：

```
> taylor( sin(tan(x))-tan(sin(x)), x=0, 15 );
```

$$-\frac{1}{30}x^7 - \frac{29}{756}x^9 - \frac{1913}{75600}x^{11} - \frac{95}{7392}x^{13} + O(x^{15})$$

在调用 `taylor` 函数时，我们只需要指定有待展开的表达式、展开点、和展开的阶数就可以了。如果不给定展开点，默认情况下就是 0 点。从形式上看，它同截断形式的幂级数完全一样，实际上也是如此，它们在 Maple 内部有着相同的表观数据类型——`series`，读者可以用 `whattype` 命令来检查一下它们的类型是否相同。`series` 同时也是一个函数，它可以用来展开更为一般的截断级数，比如劳朗级数等，它会视具体情况来决定展开成什么级数。它的调用和 `taylor` 完全一样。

```
> series( GAMMA(x), x=0, 3 );
```

$$x^{-1} - \gamma + \left(\frac{1}{12}\pi^2 + \frac{1}{2}\gamma^2 \right)x + \left(-\frac{1}{3}\zeta(3) - \frac{1}{12}\pi^2\gamma - \frac{1}{6}\gamma^3 \right)x^2 + O(x^3)$$

对于一个截断级数，我们可以用函数 `order()` 获得它的展开阶数：

```
> order(%);
```

3

`taylor` 和 `series` 中的第三个参数是可省参数，在省略时，Maple 就会把级数展开到系统变量 `Order` 所指定的阶数。默认情况下，`Order` 的值是 6。可以用赋值改变 `Order` 的值，来改变默认情况下的级数展开阶数。

```
> Order := 3: series( f(x), x=a );
```

$$f(a) + D(f)(a)(x-a) + \frac{1}{2}(D^{(2)})(f)(a)(x-a)^2 + O((x-a)^3)$$

```
> series( f(x)/(x-a)^2, x=a );
```

$$f(a)(x-a)^{-2} + D(f)(a)(x-a)^{-1} + \frac{1}{2}(D^{(2)})(f)(a) + O(x-a)$$

```
> series( 1/(cos(x)-sec(x)), x=0 );
```

$$-x^{-2} + O(x^{-1})$$

从最后一个例子中，我们可以看到，对于劳朗级数而言，截断阶数仅仅指 Maple 再计算过程中用到的阶数，最后结果的阶数往往只含有更少的项。这在某种意义上来说，有一点像数值计算中的精度损失。这是结果阶数小于给定值的情况，而有时，结果的阶数会反过来大于给定的阶数。

```
> series( 1/(1-x^2), x=0, 5);
```

$$1 + x^2 + x^4 + O(x^6)$$

我们只要求展开到 5 阶，然而 Maple 知道第 5 阶项为 0，所以它通过结果表示这一点。前面说过，用 `series` 展开的级数类型是由输入表达式的形式而定的。如果需要限制展开为劳朗级数，可以使用 `numapprox` 工具包中的 `laurent` 函数。否则，`series` 有可能会得出一些更为

广义上的级数，比如 Puisseux 级数：

```
> series( 1 / ( x * (1+sqrt(x)) ), x=0, 3 );
```

$$\frac{1}{x} - \frac{1}{\sqrt{x}} + 1 - \sqrt{x} + x - x^{\left(\frac{3}{2}\right)} + x^2 - x^{\left(\frac{5}{2}\right)} + O(x^3)$$

Maple 在计算截断级数上还具有一个有用的特性，即待展开的表达式或函数不一定要有解析表达式，而只要可以求导就可以计算。比如，我们可以计算一个含参变量的积分式对参变量的级数展开式：

```
> Int( ln(1+s*t)/(1+t^2), t=0..infinity );
```

$$\int_0^{\infty} \frac{\ln(1+st)}{1+t^2} dt$$

```
> series( %, s=0, 4 );
```

$$(-\ln(s)+1)s + \frac{1}{4}\pi s^2 + \left(\frac{1}{3}\ln(s) - \frac{1}{9}\right)s^3 + O(s^4)$$

当级数展开在给定的有限项中已经完全时，和我们通常的表示方法一样，Maple 也会去掉末尾的无穷小量符号。比如在我们将一个多项式展开成为泰勒级数时。

```
> polynomial := a + b*x + c*x^2 + d*x^3;
```

$$polynomial := a + b x + c x^2 + d x^3$$

```
> taylor_series := series( %, x );
```

$$taylor_series := a + b x + c x^2 + d x^3$$

虽然此时两者看起来似乎相同，但它们的内部数据结构是完全不同的：

```
> whattype( polynomial );
```

+

```
> op( polynomial );
```

$$a, b x, c x^2, d x^3$$

```
> whattype( taylor_series );
```

series

```
> op( taylor_series );
```

$$a, 0, b, 1, c, 2, d, 3$$

内部的数据结构决定着它们的计算方式，所以，一些多项式的操作和基本运算是不可用的。这在使用中难免会带来一些不便，这可以通过使用库函数 `mtaylor()` 来克服。`mtaylor` 是用来展开多变量泰勒级数（**multivariate Taylor series**）的，它的结果就是一般的多项式而非截断级数，自然就可以进行多项式运算了。

```
> readlib(mttaylor);
```

```
> mttaylor( sin(x^2 + y^2), [x,y], 8 );
```

$$x^2 + y^2 - \frac{1}{6}x^6 - \frac{1}{2}y^2 x^4 - \frac{1}{2}y^4 x^2 - \frac{1}{6}y^6$$

不过，对于级数，我们还是可以用 `normal()` 来对展开式进行正则化，用 `coeff()` 来获得展开式的系数。和多项式不同的是，对于级数，`coeff` 只能获得主变量的幂的系数。实际上，

我们可以利用 Maple 的库函数 `coeftayl`，不进行泰勒展开就获得泰勒级数的系数。调用 `coeftayl(f, x=x0, k)` 可以计算 f 在 $x=x_0$ 点的泰勒展开式中 $(x-x_0)^k$ 的系数。

与我们在前面介绍的系数函数形式表示的幂级数不同，我们可以直接对截断形式的级数进行求导和积分运算。（系数函数表示的幂级数的导数和积分运算分别用 `powdiff` 和 `powint`。）

```
> sin_series := series( sin(x), x );
                               sin_series := x - 1/6 x^3 + 1/120 x^5 + O(x^6)
> diff( sin_series, x );
                               1 - 1/2 x^2 + 1/24 x^4 + O(x^5)
> int( %, x );
                               x - 1/6 x^3 + 1/120 x^5 + O(x^6)
```

截断形式级数的逆可以通过 Maple 的代数方程求解函数 `solve()` 得到。

```
> solve( y = sin_series, x );
                               y + 1/6 y^3 + 3/40 y^5 + O(y^6)
> series( arcsin(y), y );
                               y + 1/6 y^3 + 3/40 y^5 + O(y^6)
```

2.4.4 切比雪夫级数和渐进级数

利用 Maple 还可以将表达式展开成两个比较重要的级数——切比雪夫级数（Chebyshev series）和渐进级数（asymptotic series）。切比雪夫级数是用切比雪夫多项式逼近函数的方法，在 Maple 中有对应的函数 `chebyshev()`。和前面所提到的几种级数展开不同，Maple 中只能展开数值的切比雪夫级数，换句话说，有待展开的函数在自变量确定时，可以求出唯一的数值结果。

```
> chebyshev( cos(x), x );
.7651976865 T(0, x) - .2298069699 T(2, x) + .004953277928 T(4, x)
- .00004187667601 T(6, x) + .1884468834 10^-6 T(8, x)
- .5261230378 10^-9 T(10, x)
```

作为数值逼近，需要给定自变量的变化范围，可以在该函数的第二个参数中用 $x = a..b$ 的形式给出，如果只给定自变量名而没有范围，则默认范围为 $-1..1$ 。展开的项数由自变量变化范围和所需的精度决定，所需精度可以在该函数的第三个参数中给出，默认的精度是 $10^{-\text{Digits}}$ 。结果中的 T 是切比雪夫多项式，它在 `orthopoly` 工具包中有定义，只需包含该工具包，就可以把切比雪夫级数转化为多项式。

```
> with(orthopoly, T): %%;
.9999999999 - .5000000000 x^2 + .04166666582 x^4 - .001388885292 x^6
+ .00002479463857 x^8 - .2693749954 10^-6 x^10
```


渐进级数，实际上可以看成是以 $\frac{1}{x}$ 为变元的幂级数。在 Maple 内部，实际上也是这么处理的。函数 `asympt(eps, x)` 和用 `taylor(eps, x=infinity)` 得到的结果相同，它们所作的运算也一样，相当于 `subs(x=1/x, series(subs(x=1/x, eps), x=0, n))`。由于在得到了级数之后，又作了一次变量代换，所以结果的数据类型不再是级数，而成为一个和式了。

```
> asymt( x/(1-x+x^2), x );
      1 1 1 1
      x x^2 x^4 x^5 + O(1/x^7)
> taylor( x/(1-x+x^2), x=infinity );
      1 1 1 1
      x x^2 x^4 x^5 + O(1/x^7)
> whattype(%), whattype(%);
      +, +
```

2.5 积分变换

在这一节中，将介绍用 Maple 进行一些常用的积分变换的方法，例如拉普拉斯变换 (Laplace transforms)、富利叶变换 (Fourier transforms)、梅林变换 (Mellin transforms)。函数 f 的积分变换 $T(f)$ 可以这样来定义：

$$T(f)(s) = \int_a^b f(t)K(s,t)dt$$

其中 K 成为变换核，不同的变换核对应着不同的积分变换，拉普拉斯变换、富利叶变换、梅林变换的变换核分别是 e^{-ist} 、 e^{-st} 、 t^{s-1} 。它们在 Maple 中都有着对应的函数，这些函数位于 `intrtrans` 工具包中，使用前要用 `with` 命令引入。

表 2.1 常用积分变换

变换	定义	Maple 中的函数
拉普拉斯	$\int_0^{\infty} f(t)e^{-st} dt$	<code>laplace(f(t), t, s)</code>
富利叶	$\int_{-\infty}^{\infty} f(t)e^{-ist} dt$	<code>fourier(f(t), t, s)</code>
梅林	$\int_0^{\infty} f(t)t^{s-1} dt$	<code>mellin(f(t), t, s)</code>

2.5.1 拉普拉斯变换

很多由多项式或者一些特定的函数（如狄拉克函数 (**Dirac** function)、海维西特函数 (**Heaviside** function)、贝塞尔函数等等)组成的有理表达式或它们的和，都可以用 Maple 积分变换工具包 `intrtrans` 中的函数 `laplace()` 求得它们的拉普拉斯变换；相应的逆变换也可以用 `invlaplace()` 求得。

```

[> t^2-exp(t)+sin(a*t);
      
$$t^2 - e^t + \sin(at)$$

[> with(inttrans):
[> laplace(%, t, s);
      
$$2\frac{1}{s^3} - \frac{1}{s-1} + \frac{a}{s^2+a^2}$$

[> invlaplace(%, s, t);
      
$$t^2 - e^t + \frac{a \sin(\sqrt{a^2} t)}{\sqrt{a^2}}$$


```

由于拉普拉斯变换的导数和积分性质，它被大量地应用到微分方程和积分方程的求解上。作为拉普拉斯变换的一个例题，我们来解一个积分方程：

```

[> int_eqn:= int( exp(a*x) * f(t-x), x = 0..t ) + b*f(t)
    = t;
      
$$\text{int\_eqn} := \int_0^t e^{ax} f(t-x) dx + b f(t) = t$$


```

对方程进行拉普拉斯变换，得到 $f(t)$ 的变换 $L(f)(s)$ 的方程：

```

[> laplace(%, t, s);
      
$$\frac{\text{laplace}(f(t), t, s)}{s-a} + b \text{laplace}(f(t), t, s) = \frac{1}{s^2}$$


```

下面要从这个简单的代数方程中解出 $L(f)(s)$ 来，不过要解出来的不是一个变量，而是一个子式。当然，我们可以先用 `subs` 将这个子式替换为一个变量，再用 `solve` 求解。在这里，我们用库函数 `isolate()` 直接从方程中解出这个子式来：

```

[> readlib(isolate)( %, laplace(f(t), t, s) );
      
$$\text{laplace}(f(t), t, s) = \frac{1}{s^2 \left( \frac{1}{s-a} + b \right)}$$


```

解的拉普拉斯变换 $L(f)(s)$ 已经得到，最后，我们将它作逆变换，得到未知函数 $f(t)$ ：

```

[> invlaplace(%, s, t);
      
$$f(t) = \frac{1}{(-1+ba)^2} + \frac{at}{-1+ba} - \frac{e^{\left(\frac{(-1+ba)t}{b}\right)}}{(-1+ba)^2}$$


```

2.5.2 富利叶变换

Maple 工具包 `inttrans` 中的函数 `fourier()` 可以求表达式的富利叶变换：

```

[> 1/(1+t^2);
                                     1
                                     1+t^2
[> with(inttrans):
[> fourier(%%, t, omega);
                                     e^{\omega} \pi \operatorname{Heaviside}(-\omega) + e^{(-\omega)} \pi \operatorname{Heaviside}(\omega)

```

上面的结果中出现了海维西特函数 $\operatorname{Heaviside}()$ ，它的导数是狄拉克函数（脉冲函数），它在负半轴上取 0，正半轴上取 1。一样地，我们有逆富利叶变换函数 $\operatorname{invfourier}()$ 。

```

[> invfourier(%, omega, t);
                                     - \frac{1}{(1+I t)(-1+I t)}
[> normal(evalc(%));
                                     1
                                     1+t^2

```

对于一些特定的函数，比如三角函数、狄拉克函数、海维西特函数和贝塞尔函数等，Maple 会利用卷积定理、查表、定积分等方法求它们的富利叶变换。

```

[> fourier( BesselJ(0,t), t, omega );
                                     -2 \frac{\operatorname{Heaviside}(\omega-1) - \operatorname{Heaviside}(\omega+1)}{\sqrt{1-\omega^2}}

```

你也可以自己扩充 Maple 的富利叶变换表，尽管 Maple 内部的公式表可能比任何一本数学教科书都要完全。例如，我们自定义函数 $f(t)$ 的富利叶变换为 $F(s)/(1+s^2)$ 。我们用 Maple 函数 $\operatorname{addtable}$ 将它加到变换表中去。

```

[> addtable( fourier, f(t), F(s)/(1+s^2), t, s );
[> fourier( exp(3*I*t)*f(2*t), t, omega );
                                     \frac{1}{2} \frac{F\left(\frac{1}{2}\omega - \frac{3}{2}\right)}{\frac{1}{4}(\omega-3)^2 + 1}

```

$\operatorname{addtable}$ 是工具包 $\operatorname{inttrans}$ 中的函数，它不仅可以用来扩充富利叶变换的变换表，对其他积分变换也适用。它的第一个参数是积分变换名，第二个是一个函数，第三个是该函数经过变换后的函数，后两个参数是变换前后函数的自变量。

2.5.3 快速富利叶变换

函数 $\operatorname{fourier}$ 和 $\operatorname{invfourier}$ 是用来计算符号表达式的连续富利叶变换的。对于离散域上的数值富利叶变换，可以调用快速富利叶变换（Fast Fourier Transformation）函数 $\operatorname{FFT}()$ ，相应的逆变换函数是 $\operatorname{iFFT}()$ 。让我们来回忆一下离散富利叶变换的定义，对于一个长度为 N 的离散数值向量 $\mathbf{x} = [x_0, x_1, \dots, x_{N-1}]$ ，其富利叶变换 $\mathbf{X} = [X_0, X_1, \dots, X_{N-1}]$ 定义如下：

$$X_k = \sum_{j=0}^{N-1} x_j e^{-\frac{2\pi i j k}{N}}$$

其中 $0 \leq k \leq N-1$ 。快速富利叶变换的算法决定了 N 必须为 2 的整数次幂。作为例子，我们取 $N = 2^3$ ，来计算实数序列 $[1, 1, 1, 1, -1, -1, -1, -1]$ 的富利叶变换。和 `fourier` 不同，`FFT` 与 `iFFT` 都是库函数，而不是 `inttrans` 工具包中的函数。

`FFT` 是用来对复数序列进行福利叶变换的，需要分别提供数据的实部和虚部。

```
[> readlib( FFT );
> x := array( [ 1, 1, 1, 1, -1, -1, -1, -1 ] );
> y := array( [ 0, 0, 0, 0, 0, 0, 0, 0 ] );
> FFT(3, x, y):
> print(x);
      [0, 2.000000001, 0, 1.999999999, 0, 1.999999999, 0, 2.000000001]
> print(y);
      [0, -4.828427122, 0, -8.28427124, 0, .828427124, 0, 4.828427122]
```

`FFT` 的第一个参数表明了序列的元素个数，上面中的 3 就表示有待变换的元素有 2^3 个；后两个参数分别是变换序列的实部和虚部，它们的数据类型都是数组（`array`），可以用函数 `array()` 从数据列表生成。有关数组的详细用法我们将在下一章中介绍，这里，我们只需要知道要显示一个数组，可以用 `print` 函数。从上面的结果我们可以看到，`FFT` 把变换的结果直接赋给了输入的数组，而不是把结果作为返回值返回（实际上 `FFT` 是有返回值的，它返回的是变换的元素个数，比如上面例子中的返回值是 8）。这是有必要的，因为快速富利叶变换的要旨在于速度，如果另外赋值，将会降低效率。说到速度，我们再提供一种加快富利叶变换的途径——利用机器浮点精度，也许读者朋友还记得，在前一章中，我们曾介绍过 `evalhf`；这里，我们仍可以用它来加快计算速度，`evalhf(FFT(...))` 就可以实现用机器精度快速地计算富利叶变换。

我们再用 `iFFT` 检验变换的正确性：

```
[> iFFT(3, x, y):
> print(x);
      [1.000000000, .9999999990, .9999999995, .9999999985, -1.000000000,
      -.9999999990, -.9999999995, -.9999999985]
> print(y);
      [0, -.2500000000 10-9, 0, .2500000000 10-9, 0, .2500000000 10-9, 0,
      -.2500000000 10-9]
```

这里的结果也是将实部和虚部分开表示，我们可以利用 `zip` 函数把它们合成为更为直观的形式：

```
[> zip( (a, b) -> a+b*I, x, y );
      [1.000000000, .9999999990 - .2500000000 10-9 I, .9999999995,
      .9999999985 + .2500000000 10-9 I, -1.000000000,
      -.9999999990 + .2500000000 10-9 I, -.9999999995,
      -.9999999985 - .2500000000 10-9 I]
```

zip 函数的第一个参数是一个二元函数，它将后两个向量中的每一对元素按此函数结合成一个新的向量。

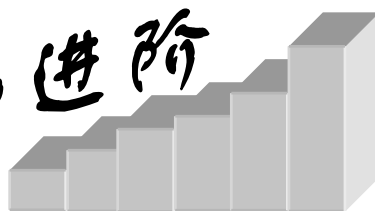
2.5.4 其他积分变换

在 Maple 中还有一些积分变换，它们的调用格式和前面介绍过的一些变换都差不多，而且，因为它们的用途都显得过于专业化，这里将不予一一详细介绍，有兴趣的读者可以参考下表和 Maple 的在线帮助。

表 2.2 其他的积分变换

变换	定义	Maple 中的函数
富利叶余弦变换	$\sqrt{\frac{2}{\pi}} \int_0^{\infty} f(t) \cos(st) dt$	fouriercos(f(t), t, s)
富利叶正弦变换	$\sqrt{\frac{2}{\pi}} \int_0^{\infty} f(t) \sin(st) dt$	fouriersin(f(t), t, s)
汉克尔变换 (Hankel transform)	$\int_0^{\infty} f(t) \sqrt{s t} \text{BesselJ}(\nu, s t) dt$	hankel(f(t), t, s, nu)
希尔伯特 (Hilbert transform)	$\frac{1}{\pi} \int_{-\infty}^{\infty} \frac{f(t)}{t-s} dt$	hilbert(f(t), t, s)

起步与进阶



第

复合数据类型

三

章

本章将介绍 Maple 软件中常用的复合数据类型，这些复合数据类型是进一步学习和灵活掌握 Maple 的基础，也是学习用 Maple 进行程序设计的必要知识。

本章具体包括以下内容：

- 🕒 Maple 中的序列
- 🕒 Maple 中的集合
- 🕒 Maple 中的有序表
- 🕒 Maple 中的数组
- 🕒 复合数据类型的定义和结构
- 🕒 复合数据类型的基本运算
- 🕒 复合数据类型的相互转换

通过前面几章的学习,读者朋友一定已经初步地掌握了在 Maple 中对单个数据元素的运算;同时,您也一定早已发觉,要灵活的应用 Maple 进行符号演算,光靠数据元素是不够的。在数学中,我们也有着各种各样的抽象复合数据类型,比如矩阵、集合等等。为了更好地理解 Maple 中的复合数据类型,我们在开始新的内容之前,有必要先花一点时间系统地介绍一下 Maple 中常用的复合数据类型。

元素级的数据类型,我们已经接触过许多种,如多项式 (polynom)、有理式 (ratpoly) 等;复合数据类型是若干个元素级数据类型按照一定规则的组合,例如序列 (sequence)、集合 (set)、有序表 (list)、数组 (array) 等,它们的作用是把些元素作为整体进行运算或者相关的操作。

3.1 序列

在前面,我们曾经不止一次地见到过表达式组成的序列。实际上,表达式序列 (exprseq) 作为一种表观数据类型,就是我们所最常见的序列。例如,取元素函数 op() 的结果就往往是一个序列——什么是序列,我们有了一个直观印象——一组用逗号相互隔开的对象。

```
> polynomial := x + 2*x^2 + 3*x^3 + 4*x^4;
      polynomial = x + 2x^2 + 3x^3 + 4x^4
> sequence := op( polynomial );
      sequence = x, 2x^2, 3x^3, 4x^4
> whattype( sequence );
      exprseq
```

我们还在什么地方看到过用逗号相互隔开的对象? 对了,多参数的函数调用的括弧中,每个参数之间都用逗号隔开——是的,函数的参数也构成了一个序列。序列这种数据形式在 Maple 中是如此地常见。在下面的例子中,解方程函数 solve() 的参数和结果都是序列。

```
> arguments := x^3 - 6*x^2 + 11*x - 6, x;
      arguments = x^3 - 6x^2 + 11x - 6, x
> whattype( arguments );
      exprseq
> solve( arguments );
      1, 2, 3
```

需要注意的是,每一个序列本身都是一个完整的对象:它们在系统内部都是由一个数据

序列	↑表达式 1	↑表达式 2	↑表达式 3	……
----	--------	--------	--------	----

向量来表示的。

其中“↑表达式 1”,“↑表达式 2”,……,分别表示指向存储各个表达式的数据结构的指针。一个序列中的各个元素并不一定具有相同的数据类型,两个不同类型的序列完全可以合并成一个序列——只要你觉得有必要。连接两个序列的操作十分简单,只需要把两个序列

写在一起，中间用逗号相隔开就可以了。

```
[> seq1 := M, a, p, l, e;
> seq2 := 77, 97, 112, 108, 101;
> concat_seq := seq1, seq2;
concat_seq := M, a, p, l, e, 77, 97, 112, 108, 101
```

序列是由若干个元素组成的，这个“若干”个，当然也可以是 0 个——者在数学上没问题，Maple 中当然也允许。Maple 中的空序列 (0 个元素的序列)，有一个专门名称——NULL。

```
[> empty_seq := NULL;
empty_seq :=
> 1, 2, empty_seq, 2, 1;
1, 2, 2, 1
```

具有一定规律的序列，也就是可以写出通项的序列，可以用函数 seq() 来生成：

```
[> seq( ithprime(i), i = 1..20 );
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71
```

函数调用 seq(f(i), i = m..n) 可以生成这样的序列：f(m), f(m+1), f(m+2), ..., f(n)。上面例子中用到了函数 ithprime(i)，它是用来获得第 i 个素数的。

Maple 还提供了一个具有和 seq 函数几乎相同功能的操作符\$。我们在介绍高阶导数时曾经遇到过这个操作符，那时，我们用 diff(f, x\$3) 表示 f 对 x 的 3 阶导数；现在，我们知道 x\$3 实际上是生成了一个 3 个元素的序列，函数调用实际上就成了 diff(f, x, x, x)，自然表示 f 对 x 的 3 阶导数了。我们利用\$操作符，可以完成和 seq 函数一样的功能。

```
[> ithprime(i) $ i = 1..20;
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71
```

但是，作为操作符的使用，有时很容易发生错误。比如上面的例子中，假如已经对 i 赋过值了，那得到的结果将会是 20 个相同的数了，因为运算符的优先级决定了运算的次序，而\$操作符的优先级是很低的。所以，除了在 diff 中使用之外，我们不提倡使用\$操作符来生成序列。

函数 seq() 第二个参数的等式右边部分还可以不是一个范围，它可以是一个任意表达式甚至一个复合数据类型——这时，代表元素 (上面例子中的 i) 将遍取该表达式或复合数据类型中的所有元素。例如：

```
[> seq( i^2, i = { 1, 2, 3, 4, 5 } );
1, 4, 9, 16, 25
> seq( i^2, i = x + y + z );
x^2, y^2, z^2
```

复合数据类型的目的是为了把一组数据作为整体来操作，但是，我们关心的往往还是单个的元素。怎样获得一个序列中的一个特定的元素呢？我们可以用选择操作符 “[]”。比如我们要选择上一例中第二个元素：

```
[> %[2];
y^2
```

选择操作符中间不仅可以是单个的数，还可以是一个范围，比如 a..b，这时，选择的结果将仍旧是一个序列：


```
[> sequence := a, b, c, d, e, f, g:
[> sequence[ 2..5 ];
                                     b, c, d, e
```

如果你学过 Matlab 或者 Fortran90 程序设计语言，你会发觉 Maple 表示范围的方式和它们是如此的相似。实际上，Maple 中对于这种表示方式的应用要灵活得多。Maple 中范围类型（range）的表达式具有这样的标准形式：

$$\text{left_expression}..\text{right_expression}$$

在前面介绍积分和级数时，我们曾经使用过这样的数据类型，用它来表示积分区间和求和的范围。我们知道，它的左表达式 *left_expression* 和右表达式 *right_expression* 都可以是表达式或者未赋值的变量，仅这一点，就是 Matlab 和 Fortran90 所望尘莫及的。当然，在用来生成序列时，它们就必须是数值型的，而且必须是整数了。顺便提一句，范围表达式的中间可以是两点，也可以是更多的点——只要你愿意——当然，一点表示变量名的联合，不要搞混了。

3.2 集合

在 Maple 中，集合类型（set）最常用的场合是解方程组。Maple 的代数方程组求解函数 *solve* 的参数是一个方程的集合和一个未知变量的集合；而它的结果，通常情况下也是一个由集合（一组解）组成的一个序列。

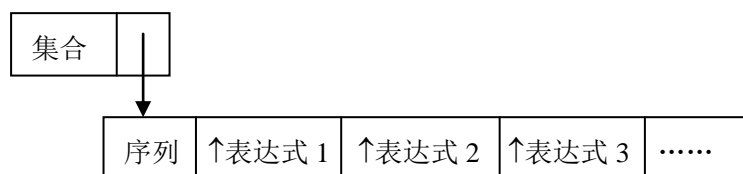
Maple 中表实际和的方法和我们的数学语言完全相同——用一对花括弧“{ }”把一个序列括在当中。和我们在数学中集合的定义相同，Maple 中，集合中的元素不能重复——系统会将重复的元素去掉，集合中的元素相互之间也没有先后次序——显示是按照内部存储地址顺序显示的。

```
[> { 3, 2, 1, 4, 5 };
                                     {1, 2, 3, 4, 5}
[> { x, y, x, x*y, x*(x-1), y*x, -x+x^2 };
                                     {y, x, x*y, x*(x-1), -x+x^2}
```

从上面后一个例子中我们可以看到，对于经过形式变换（实际上是内部存储结构变换）后相等的表达式，Maple 认为是不同的表达式，它们在内存中存储在不同的位置，所以没有把它们从集合中去掉。Maple 的集合类型当然也包括空集。

```
[> empty_set := {};
                                     empty_set := { }
```

没有元素的集合，自然就是空集；但要注意，Maple 并没有定义 ϕ 为空集。系统内部，集合的数据结构是这样的：



数学中集合的基本运算——交、并、差——在 Maple 中都有对应的操作符，分别为 intersect、union、minus。

```
[> {0, 1, 2, 3} intersect {0, 2, 4, 6};
                                {0, 2}
[> {0, 1, 2, 3} union {0, 2, 4, 6};
                                {0, 1, 2, 3, 4, 6}
[> {0, 1, 2, 3} minus {0, 2, 4, 6};
                                {1, 3}
```

利用函数 member()，可以判断一个特定的元素是否属于一个集合，如果是的话，还可以确定它在集合中的位置。

```
[> member( 1, {0, 1, 2, 3}, pos );
                                true
[> pos;
                                2
```

位置的数值对于集合来说是没有意义的，它只表示 Maple 中存储的地址顺序，但是，利用它可以引用集合中的元素。和序列一样，我们可以用选择操作符 “[]”，当然也可以使用我们早已知道的函数 op()。

```
[> # 生成{1, 2, 3}的所有子集
[> collection := combinat[powerset](3);
      collection := ({ }, {1, 3}, {1, 2, 3}, {2, 3}, {3}, {1}, {2}, {1, 2})
[> ops( % ); # 元素个数
      ops(({ }, {1, 3}, {1, 2, 3}, {2, 3}, {3}, {1}, {2}, {1, 2}))
[> collection[4..6];
                                {{2, 3}, {3}, {1}}
[> op( 8, collection );
                                {1, 2}
```

上面的例子中，我们用到了组合数学工具包 combinat 中的函数 powerset，顾名思义，它是用来生成幂集的函数。

前面，我们用元素在集合中的位置来选择该元素。有时，我们须要选择一个集合中满足某种条件的元素，这时，使用 Maple 函数 select()就会很有用。它的调用格式是这样的：

select(criterion, set, extra_arguments)

其中 *criterion* 是一个返回值为布尔型的函数，它是我们用来决定选择与否的准则，*set* 是有待筛选的集合，*extra_arguments* 是调用函数 *criterion* 所需的附加参数。

```
[> die := rand(-20..20):
> number := { seq( die(), i=1..10 ) };
               number := {1, 9, -10, -15, -16, 17, -18, 19}
> select( isprime, number ); # 选择其中的素数
               {17, 19}
> select( type, number, nonnegint ); # 选择非负整数
               {1, 9, 17, 19}
```

3.3 有序表

有序表 (list) 在 Maple 中的表示形式是一对方括弧 “[]” 把一个序列括在其中。在多变量多项式的合并同类项函数 `collect()` 中, 可以把这种数据类型作为参数, 表示合并同类项的变量。

```
[> 1 + x + y^2 + z^3 + x^2*y^2 + x^2*z^3 + y^3*z^2;
               1+x+y^2+z^3+x^2*y^2+x^2*z^3+y^3*z^2
> collect( %, [x, y, z] );
               (y^2+z^3)x^2+x+1+y^2+z^3+y^3*z^2
> collect( %, [z, x, y] );
               (x^2+1)z^3+y^3*z^2+x^2*y^2+x+1+y^2
```

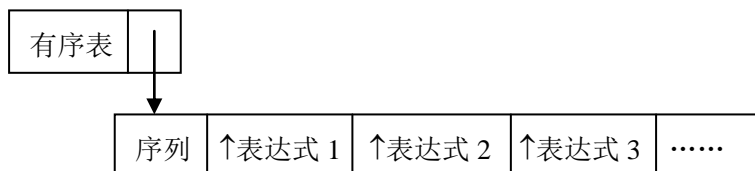
我们看到, 当元素出现的顺序不同时, 有序表也不同。也正因为此, 我们称其为有序表。因为元素在有序表中具有先后顺序关系, 所以有序表中的元素是可以重复的, 这是有序表和集合之间的最大区别了。我们也可以人为地对有序表进行排序, 和集合中的顺序一样, 这里的排序也是相对于存储地址而言的。

```
[> [ x, y, x, x*y, x*(x-1), y*x, -x+x^2 ];
               [x,y,x,x*y,x(x-1),x*y,-x+x^2]
> sort( % );
               [y,x,x,x*y,x*y,x(x-1),-x+x^2]
```

和空集相对应, 有序表也有空表的概念:

```
[> empty_list := [];
               empty_list := [ ]
```

有序表的内部存储结构和集合极为相似:



下面, 我们将有序表的常用操作依次列出, 并附以简单的使用例子:

◇ 求有序表的长度。

```
[> c1 := [ black, red, green, yellow, blue, white ]:
> nops(c1);
6
```

✧ 在有序表中查找一个元素。

```
[> member( brown, c1 );
false
> member( yellow, c1, 'pos' );
true
> pos;
4
```

✧ 选择有序表中的一个或多个元素。

```
[> c1[4], op(4, c1);
yellow, yellow
> c1[2..5];
[red, green, yellow, blue]
```

✧ 在有序表中加入元素。

```
[> [ brown, op(c1) ];
[brown, black, red, green, yellow, blue, white]
> [ op(c1), brown ];
[black, red, green, yellow, blue, white, brown]
> [ op(1..3, c1), brown, op(4..nops(c1), c1) ];
[black, red, green, brown, yellow, blue, white]
```

✧ 联接有序表。

```
[> c12 := [ brown, cyan ]:
> [ op(c1), op(c12) ];
[black, red, green, yellow, blue, white, brown, cyan]
```

✧ 替换有序表中的一个元素。

```
[> subsop( 4 = brown, c1 );
[black, red, green, brown, blue, white]
```

✧ 替换有序表中所有指定的元素。

```
[> subs( yellow = brown, [ yellow, op(c1) ] );
[brown, black, red, green, brown, blue, white]
```

✧ 有序表重排序。

```
[> sort( c1, lexorder ); # 按字母顺序排序
[black, blue, green, red, white, yellow]
> [ op(2..nops(c1), c1), c1[1] ]; # 循环左移
[red, green, yellow, blue, white, black]
> [ c1[nops(c1)], op(1..nops(c1)-1, c1) ]; # 循环右移
[white, black, red, green, yellow, blue]
```

3.4 数组

在这一节中，我们将简要的介绍 Maple 中的数组类型（array），以及一些相关的对象，比如向量（vector）和矩阵（matrix）。在下一章中，我们将进一步对它们作详细介绍。在这里，我们暂时局限于讨论这些数据结构的生成。

和其他大多数常规编程语言一样，在 Maple 中有着一维、二维和三维的数组。下面的例子中，就生成了一个具有数组类型的对象：

```
> array( -1..1, 0..1,
  [ (-1,0)=a, (-1,1)=b, (0,0)=c, (1,0)=d ] );
array(-1..1,0..1,[
  (-1,0)=a
  (-1,1)=b
  (0,0)=c
  (0,1)=?0,1
  (1,0)=d
  (1,1)=?1,1
])
```

在构造数组的函数 `array()` 中，我们首先指定的是数组在两个维度上的指标范围。Maple 中的数组的指标范围可以任意指定，在一定程度上和 Pascal 语言中的数组有点相似。然后，我们以一个有序表的形式指定数组中的一些元素（并不一定要指定所有元素）。在这里，尤其需要注意方括弧和圆括弧的用法。在生成一个数组时，我们用圆括弧括起来的数对指定数组的各个元素；而在引用一个数组的某个元素时，我们还是使用前面曾经用过多次的选择操作符“`[]`”。

数组数据类型，实际上是另一种数据类型——查找表（table）的特例。查找表这种数据结构在 Maple 的内部很常用，例如我们上一章中曾经提到过的积分变换公式表。顺便提一句，Maple 的内部查找表是用数据结构中称为哈希表的结构实现的。

乍一看，数组合我们学过的数据类型——有序表极为相象。而实际上，数组可以由有序表直接转化而来。

```
> v := array( [ a, b, c ] );
                                     v := [a, b, c]
> type(v, list);
                                     false
> type(v, array);
                                     true
```

我们看到，特别是在输出的表达式上，一维数组和有序表几乎一模一样，但它们的内部存储结构是不同的，我们必须加以区分。二维、三维的数组以同样可以由有序表直接生成。用有序表直接转化生成数组，可以省去许多不必要的输入——每个元素的指标。另外，如果省略每一维的指标范围不写，则默认的指标范围是从 1 开始的。

```

> M := array( [ [p, q], [r, s] ] );
                                     
$$M := \begin{bmatrix} p & q \\ r & s \end{bmatrix}$$


> P := array( [ [ [1,2], [3,4] ], [ [5,6], [7,8] ],
               [ [9, 10], [11, 12] ] ] );
P := array(1..3, 1..2, 1..2, [
  (1, 1, 1) = 1
  (1, 1, 2) = 2
  (1, 2, 1) = 3
  (1, 2, 2) = 4
  (2, 1, 1) = 5
  (2, 1, 2) = 6
  (2, 2, 1) = 7
  (2, 2, 2) = 8
  (3, 1, 1) = 9
  (3, 1, 2) = 10
  (3, 2, 1) = 11
  (3, 2, 2) = 12
])

```

从上面的结果中，我们不难发现，Maple 中的 2、3 维数组都是按照指标从后向前变化排列的，从这种意义上来说，它的数组和 C、Pascal 都比较相似，但和 Fortran 中的数组不同。

在数学中，我们常常把二维和三维数组分别称为向量和矩阵；在 Maple 中也是，它们的类型名称分别是 vector 和 matrix。

```

> type( v, vector ), type( M, matrix );
                                     true, true

```

和我们通常在数学中的概念有所不同，Maple 中的 vector 仅仅指行向量，而对于列向量，也就是 $n \times 1$ 的矩阵，Maple 不把它认作是 vector。

在 Maple 的线性代数工具包 linalg 中，有函数 vector() 和 matrix() 可以直接生成向量和矩阵。

```

> with( linalg ):
Warning, new definition for norm
Warning, new definition for trace
> v := vector( [ a, b, c ] );
                                     
$$v := [a, b, c]$$

> M := matrix( [ [p, q], [r, s] ] );
                                     
$$M := \begin{bmatrix} p & q \\ r & s \end{bmatrix}$$


```

在必要的时候，你可以仅仅指定一个矩阵或向量的大小，而不确定它的元素。

```

[> v := vector(100);
      v := array(1..100,[ ])
[> M := array(-10..10, -10..10, []);
      M := array(-10..10, -10..10,[ ])

```

当我们知道矩阵种的元素和它所在位置的关系时，矩阵还可以由指标函数方便地生成。作为例子，我们构造一个 3×3 的希尔伯特阵（Hilbert matrix）。

```

[> h := (i,j) -> 1/(i+j-x); # 指标函数
      h := (i,j) -> \frac{1}{i+j-x}
[> matrix( 3, 3, h );
      \begin{bmatrix} \frac{1}{2-x} & \frac{1}{3-x} & \frac{1}{4-x} \\ \frac{1}{3-x} & \frac{1}{4-x} & \frac{1}{5-x} \\ \frac{1}{4-x} & \frac{1}{5-x} & \frac{1}{6-x} \end{bmatrix}

```

当然了，对于希尔伯特阵，我们完全可以用 linalg 中的函数 hilbert 来生成。利用指标函数，我们还可以方便地生成一些特殊矩阵，例如零矩阵：

```

[> matrix( 3, 3, 0 );
      \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}

```

在这个函数调用中，0 也是一个函数，它可以理解为取 0 值的常函数。Maple 预先定义了一些常用的指标函数，可以通过它们生成一些特殊形式的矩阵：

```

[> M := array( symmetric, 1..2, 1..2 ): M[1,2]:=1:
[> print(M);
      \begin{bmatrix} M_{1,1} & 1 \\ 1 & M_{2,2} \end{bmatrix}

```

注意，只可以附加一个指标函数，而且只有在函数 array()中可以附加这些指标函数，在 vector 和 matrix 中都不可以。这里将这些内建的指标函数列成下表，供读者参考：

表 3.1 特殊矩阵的指标函数

内建指标函数	对应的矩阵
symmetric	对称矩阵
antisymmetric	反对称矩阵
sparse	稀疏矩阵
diagonal	对角阵
identity	单位阵

由于存储结构的原因，数组的整体只显示数组名，而不显示每个元素。为了显示数组的每个元素，需要调用 eval 或者 print。

对数组的元素的操作非常简单直观，和前面介绍的几种复合数据类型一样，可以用选择操作符 “[]” 引用和修改任何一个元素。

3.5 类型转换和元素运算

关于复合数据类型之间的转换，其实我们在无意之中已经用了很多次。比如你一定知道怎样把序列转化成集合或者有序表——在外边加上一对花括弧或者方括弧就可以了，反过来就更简单了，op 函数的结果就是一个序列。除了这些以外，Maple 的类型转换函数 convert() 可以用来显式地在除了序列外的复合数据类型之间相互转换。

```
[> sequence := a, b, c, d;
      sequence := a, b, c, d
[> convert( [sequence], set );
      {a, b, c, d}
[> convert( %, list );
      [a, b, c, d]
[> convert( %, array );
      [a, b, c, d]
```

当然，也可以转换成向量和矩阵：

```
[> convert( [ a,b,c], vector );
      [a, b, c]
[> convert( [ [a,b], [c,d] ], matrix );
      [ a  b ]
      [ c  d ]
```

如果你用过 Fortran90，你一定会为它特别为数组操作设计的函数调用功能所吸引。任何一个定义成以单个元素为参数的函数，不加任何说明，就可以作用在整个数组上，调用的结果是每一个元素分别调用函数所得结果所组成的数组。遗憾的是，Maple 不具备这样的功能。但是，我们可以通过函数 map()，起到相同的效果。

```
[> cos(%);
Error, cos expects its 1st argument, x, to be of type algebraic, but
received array(1 .. 2, 1 .. 2, [(2, 1)=c, (2, 2)=d, (1, 2)=b, (1, 1)=a])
[> map(cos, %%);
      [cos(a) cos(b)]
      [cos(c) cos(d)]
```

但是，需要注意的是，对于用内建指标函数构成的特殊矩阵，这样的操作有时会起不了作用。因为特殊矩阵的内部结构和一般矩阵是不同的，简单地说，就是特殊矩阵的结构元素（就是 op 返回的结果，也是 map 作用的对象）并不是和实际的矩阵元素相对应的。

```
[> sp_m := array( sparse, 1..5, 1..5 );
[> sp_m[1,1] := Pi/4;
[> sp_m[2,2] := Pi/3;
```



```
> map(cos, sp_m);
```

$$\begin{bmatrix} \frac{1}{2}\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

这个例子中，我们期望的结果是一个大部分为 1 ($\cos(0)$) 的矩阵，然而结果却仍是一个稀疏阵，因为 0 并不是稀疏阵的一个结构元素，`map` 对它不起作用。

起步与进阶



第

线性代数

四

章

本章将介绍 Maple 软件中的线性代数工具, 以及利用 Maple 软件解决线性代数问题的方法和技巧。

本章具体包括以下内容:

- 🕒 Maple 中的矩阵运算
- 🕒 Maple 中的行列式计算
- 🕒 利用 Maple 计算特征值
- 🕒 Maple 中数组和表的求值机制
- 🕒 Maple 线性代数工具包中的常用函数
- 🕒 Maple 中的特殊矩阵
- 🕒 利用 Maple 线性代数问题的实例

线性代数是讨论有限维空间线性理论的基础数学分支。由于线性问题广泛地存在于自然科学和计数科学的各个领域,而且某些非线性问题在一定条件下也可以转化为线性问题来处理,线性代数中的概念和方法,已经被广泛地运用到各个学科领域中。在计算机技术高度发展的今天,线性代数更是如虎添翼,成为了从事自然科学和工程计数工作不可缺少的数学工具之一。正因为此,Maple 也对线性代数工具有专门的实现,它对于解决小规模线性代数问题,尤其是符号线性代数问题,是极为实用的。不过,如果要处理较大规模的数值线性代数问题,还是推荐您使用专门的线性代数工具——Matlab,它在数值矩阵运算方面是首屈一指的。

在上一章中的数组部分中,我们曾经简要地介绍过矩阵和向量,我们还提到了 Maple 的线性代数工具包。这一章将为您更为详尽地介绍 Maple 中矩阵和向量的使用和运算,以及 Maple 线性代数工具包中的主要函数和它们的用法,我们将通过例子介绍 Maple 解决实际线性代数问题的方法。我们还会顺带介绍 Maple 中矩阵求值的一些内部机制,使您能更好的理解 Maple 的复合数据类型,为利用 Maple 自己编制计算程序打下基础。

4.1 矩阵的基本运算

在开始用 Maple 进行矩阵运算之前,需要载入 Maple 的线性代数工具包——linalg,Maple 中大部分矩阵运算以及和矩阵相关的函数都是在这个工具包中定义的。

```
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace
```

在载入这个工具包时产生了两个警告,不用为此担心,这是因为 Maple 系统中已有的两个函数——norm 和 trace 被 linalg 中的同名函数所覆盖了。原来系统中这两个函数分别是用于计算多项式的范数(norm)和程序调试跟踪的,一般情况下,在解决线性代数问题时用不到。如果确实要用系统中的库函数,可以再用 readlib 将其重新载入进来。

linalg 中的主要函数的详细介绍我们将留到本章第三节中完成,在这一节中,我们主要着眼于矩阵的基本运算,比如加法、乘法和乘幂等。我们先用上一章中介绍过的方法来定义一些矩阵:

```
> A := matrix( 2, 2, [a, b, c, d] );
      A :=  $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ 
> B := matrix( [ [alpha, beta], [beta, alpha] ] );
      B :=  $\begin{bmatrix} \alpha & \beta \\ \beta & \alpha \end{bmatrix}$ 
> C := matrix( 2, 3, (i, j) -> i+j-1 );
      C :=  $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}$ 
```

矩阵的代数运算最直接而且直观的方法莫过于使用函数 evalm()了,只要把矩阵的代数

计算表达式作为它的参数，就可以得到结果的矩阵。

```
[> evalm( A + B );
```

$$\begin{bmatrix} a+\alpha & b+\beta \\ c+\beta & d+\alpha \end{bmatrix}$$

```
> evalm( 2*A + k*B/7 );
```

$$\begin{bmatrix} 2a+\frac{1}{7}k\alpha & 2b+\frac{1}{7}k\beta \\ 2c+\frac{1}{7}k\beta & 2d+\frac{1}{7}k\alpha \end{bmatrix}$$

矩阵和标量也可以直接作和差。但要注意的是，和 Matlab 以及 Fortran90 中的相应的运算定义不同，在 Maple 中，矩阵和标量相加（相减），被定义成矩阵的对角元和标量分别相加（相减），也就是矩阵加上（减去）和它相同形状的数量阵。请看下面的例子：

```
[> evalm( A - 1 );
```

$$\begin{bmatrix} a-1 & b \\ c & d-1 \end{bmatrix}$$

```
> evalm( C + 10 );
```

$$\begin{bmatrix} 11 & 2 & 3 \\ 2 & 13 & 4 \end{bmatrix}$$

还需要说明的是，运算符“*”仅仅用来表示数乘，对于矩阵乘法，绝对不能使用。这是因为，运算符“*”在 Maple 中被定义成满足交换律的乘法，而我们知道，矩阵乘法是不满足交换律的。我们可以通过简单的例子来说明这个事实。

```
[> 2*A*B + B*A;
```

$$3AB$$

```
> A*A + C*B*C;
```

$$A^2 + C^2B$$

可以看到，Maple 在化简表达式时，丝毫没有顾及它们的类型，只把它们作为一般的符号变量来计算，甚至不考虑这些矩阵是否可以相乘或相加。如果我们用 evalm 来计算，也就得不到正确的结果了。在 Maple V Release 4 及以后的版本中，程序设计者考虑到用户初次使用时可能对此不了解，特别给除了错误信息。

```
[> evalm( B*A );
```

Error, (in evalm/evaluate) use the &* operator for matrix/vector multiplication

但如果你使用的是早期版本，就不但没有错误信息，还会给出错误的结果，使用时一定要特别注意。上面的错误信息告诉我们，矩阵/向量运算必须使用&*作为乘法运算符。让我们再来试一试。

```
[> evalm( B &* A );
```

$$\begin{bmatrix} \alpha a + \beta c & \alpha b + \beta d \\ \beta a + \alpha c & \beta b + \alpha d \end{bmatrix}$$

```
> evalm( A &* B );
```

$$\begin{bmatrix} \alpha a + \beta b & \beta a + \alpha b \\ \alpha c + \beta d & \beta c + \alpha d \end{bmatrix}$$

这次，我们得到了期望的结果。

矩阵也可以进行乘幂运算，矩阵的乘幂运算符和单个表达式的相同——“^”，当然，也需要用 evalm 来计算得到结果矩阵。

```
> evalm( B^2 );
```

$$\begin{bmatrix} \alpha^2 + \beta^2 & 2\alpha\beta \\ 2\alpha\beta & \alpha^2 + \beta^2 \end{bmatrix}$$

Maple 不仅可以计算矩阵的自然数次幂，同样也可以计算矩阵的负整数次幂。

```
> evalm( B^(-1) );
```

$$\begin{bmatrix} -\frac{\alpha}{-\alpha^2 + \beta^2} & \frac{\beta}{-\alpha^2 + \beta^2} \\ \frac{\beta}{-\alpha^2 + \beta^2} & -\frac{\alpha}{-\alpha^2 + \beta^2} \end{bmatrix}$$

在计算矩阵的 0 次幂时，却不怎么令人满意，它的结果令人啼笑皆非：

```
> evalm( B^0 );
```

$$1$$

```
> whattype(%);
```

integer

这是由于 Maple 的表达式自动化简机制造成的，B^0 在还没有进入 evalm 函数之前，就已经被系统化简成了 1！

矩阵乘法运算符“&*”有着和数乘“*”、除法“/”相同的优先级，在输入表达式是需要注意，必要时使用括弧。

```
> evalm( A &* 1/A );
```

Error, (in evalm/ampersstar) &* is reserved for matrix multiplication

```
> evalm( A &* (1/A) );
```

&*()

还好，Maple 没有再次给出 1，Maple 用“&*()”来表示单位阵。

4.2 矩阵求值

我们早在上一章中就已经注意到，数组和单个的表达式不同，具有数组类型的变量不会自动求值，而需要用 eval() 等函数的显式调用来强制地求值。作为数组的特例，矩阵和向量也是一样。更令人吃惊的是，当我们用 whattype 来查看数组变量的表观数据类型时，返回的居然是符号类型 (symbol)，和被赋值之前一样！

```

[> Q := matrix( [ [cos(alpha), sin(alpha)],
                  [-sin(alpha), cos(alpha)] ] );
                  Q :=  $\begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$ 
[> Q;
                  Q
[> whattype(Q);
                  symbol
[> eval(Q);
                   $\begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$ 

```

然而，通过进一步的试验你还会发现，即使用 `eval()`，还是没办法让矩阵的每一个元素都求值。只有通过我们在上一章中用过的函数 `map`，才可以完成最终的求值。

```

[> alpha := 0:
[> eval(Q);
                   $\begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$ 
[> map( eval, Q );
                   $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ 

```

同样的情况也发生在以查找表（table）和函数或子程序（procedure）为类型的对象上：

```

[> int_table := table( [ sin(x) = cos(x) + C,
                       cos(x) = -sin(x) + C ] ):
[> int_table;
                  int_table
[> int_table[ cos(x) ];
                  -sin(x) + C
[> C := C1: eval(int_table);
table([
  sin(x) = cos(x) + C
  cos(x) = -sin(x) + C
])
[> map( eval, int_table );
table([
  sin(x) = cos(x) + C1
  cos(x) = -sin(x) + C1
])

```

在处理这些类型的对象时，Maple 采取的是按名求值的方法，而不是像对于单个表达式时采取的完全求值的方法。所谓“按名求值”，是指系统在计算是只将变量的名称代入进行计算，而不读取该变量名所指向的数据。如果需要得到变量名所指的的值，则必须强制使用 `eval` 甚至是 `map` 和 `eval` 并用。通过一个简单的例子，也许可以更容易地理解这种求值方式。

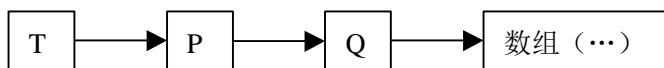
```

[> T := P:
[> P := Q:
[> eval(T, 1); # T的值
[
[> eval(T, 2); # P的值
[
[> eval(T, 3); # Q的值
[

$$\begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$$


```

我们通过对 T 的不同层次的求值，获得了不同的对象，很明显，在 Maple 内部，数据结构是用类似于下图的方式组织的：



如果我们用系统的自动求值，则只能到 Q 的一层为止，就好像从 Q 指向数组的指针上有一道难以逾越的屏障：

```

[> P, T;
[
Q, Q

```

不管我们怎么赋值，T、P、Q 这些变量总是直接或间接地指向同一个数据对象，因为在赋值时，用的也是“按名求值”。用面向对象的观点来看，它们都是这个矩阵变量的引用，而不是它的一个拷贝。当我们修改了其中任意一个变量的元素时，其他变量的元素也会跟着改变。在这上面，Maple 和有一点向一些面向对象的编程语言，比如 Java，但和 C++ 是不同的。

```

[> T[1,2] := changed:
[> eval(T), eval(P), eval(Q);
[

$$\begin{bmatrix} \cos(\alpha) & \text{changed} \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}, \begin{bmatrix} \cos(\alpha) & \text{changed} \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}, \begin{bmatrix} \cos(\alpha) & \text{changed} \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$$


```

当然，有时候我们要的是一个矩阵的拷贝，我们将改变这个拷贝的原素值，而同时保持原来的矩阵不变；这时，我们需要用 Maple 的拷贝函数 `copy()`，它可以用来拷贝这些“按名求值”的对象。

```

[> S := copy(Q): S[2,1] := 0:
[> eval(S), eval(Q);
[

$$\begin{bmatrix} \cos(\alpha) & \text{changed} \\ 0 & \cos(\alpha) \end{bmatrix}, \begin{bmatrix} \cos(\alpha) & \text{changed} \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$$


```

4.3 矩阵和线性方程组

`linalg` 是 Maple 中的线性代数工具包，有着很全面的线性代数计算工具。那它里面到底有些什么工具呢？你可以试试在载入它时，不用冒号而是用分号结尾，那样，它的所有函数

就会被一一打印出来。这可是一个长长的有序表，因为 `linalg` 中的函数多达 114 个。有一定数学基础的读者，也许看看它的函数名称，就可以大概地知道 `linalg` 中间都有些什么工具了。自然，矩阵的基本运算是少不了的，还有矩阵的行、列操作，高斯消元，矩阵的行列式、秩、逆、迹，特征多项式、特征矩阵，矩阵的特征值和特征向量……，举不胜举。从这一节起，我们分类将一些常用的函数作一下简单介绍；其他的一些函数，请读者在使用时参考在线帮助。

4.3.1 矩阵基本运算

在上一节中，我们已经介绍过矩阵的基本运算，那时，我们用的是函数 `evalm()` 配合运算符。作为函数式的编程语言，Maple 同时也提供了一套函数来完成这些矩阵基本运算，以使不同的场合下可以选用合适的方式进行运算。我们不准备详细的介绍它们的使用，仅将它们列成下表，供读者在使用时参考。

表 4.1 基本矩阵运算

运算	函数调用	等效的运算符运算
加法	<code>matadd(A, B)</code>	<code>evalm(A + B)</code>
数乘	<code>scalarmul(A, expr)</code>	<code>evalm(A * expr)</code>
乘法	<code>multiply(A, B, ...)</code>	<code>evalm(A &* B &* ...)</code>
逆矩阵	<code>inverse(A)</code>	<code>evalm(1/A)</code> 或 <code>evalm(A^(-1))</code>
转置	<code>transpose(A)</code>	无
行列式	<code>det(A)</code>	无
秩	<code>rank(A)</code>	无
迹	<code>trace(A)</code>	无

4.3.2 分块矩阵

有时在处理大矩阵时，常常可以把它们视为是由一些小矩阵组成的，这样，大矩阵的运算就转化成小矩阵间的运算，往往可以带来极大的方便。Maple 是一个符号演算软件，在这方面应该具有一定的优势。令人遗憾的是，直到 Maple V Release 5 为止，Maple 中还无法表示一个抽象的矩阵（只有矩阵名称，而不知道矩阵的大小）。因此，严格说来，在 Maple 中还无法作符号表示的抽象分块矩阵运算。

但是，在 Maple 中对于分块矩阵在一定程度上还是支持的。Maple 对于分块矩阵的支持，主要表现在输入和表示某些大而且部分重复的矩阵上。或者，我们可以用已有的矩阵拼合成一个新的矩阵。

```
[> A := matrix( [ [1,0], [-1,2] ] );
[> B := matrix( [ [1,2], [3,2] ] );
[> Z := matrix( 2, 2, 0 );
[> blockmatrix(2,4, [A,B,Z,A,B,Z,A,B]);
```

$$\begin{bmatrix} 1 & 0 & 1 & 2 & 0 & 0 & 1 & 0 \\ -1 & 2 & 3 & 2 & 0 & 0 & -1 & 2 \\ 1 & 2 & 0 & 0 & 1 & 0 & 1 & 2 \\ 3 & 2 & 0 & 0 & -1 & 2 & 3 & 2 \end{bmatrix}$$

对于除了对角元附近，其他大部分元素为 0 的矩阵，可以用 `diag` 函数更为便捷地生成，

它可以将一些方阵（并不需要具有相同大小）或者标量排列在矩阵的对角线上，生成块状对角阵。

```
> diag( B, 2, A );
```

$$\begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 3 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

4.3.3 初等变换

矩阵的初等变换是各种消去法的基础，是解线性方程组的基础。对于符号演算，有时候我们不仅要求最后的求解结果，而且要求中间的求解步骤。这时候我们可以调用 Maple 线性代数工具包中的初等变换函数。对于初学线性代数的理工科低年级学生，还可以用这些函数透彻、直观地理解各种消元法的实质。

表 4.2 矩阵的初等变换

	函数调用	对应的初等变换
行变换	<code>mulrow(A, r, expr)</code>	用标量 <code>expr</code> 乘以矩阵 <code>A</code> 的第 <code>r</code> 行
	<code>addrow(A, r1, r2, m)</code>	将矩阵 <code>A</code> 的第 <code>r1</code> 行的 <code>m</code> 倍加到第 <code>r2</code> 行上
	<code>swaprow(A, r1, r2)</code>	互换矩阵 <code>A</code> 的第 <code>r1</code> 行和 <code>r2</code> 行
列变换	<code>mulcol(A, c, expr)</code>	用标量 <code>expr</code> 乘以矩阵 <code>A</code> 的第 <code>c</code> 列
	<code>addcol(A, c1, c2, m)</code>	将矩阵 <code>A</code> 的第 <code>c1</code> 列的 <code>m</code> 倍加到第 <code>c2</code> 列上
	<code>swapcol(A, c1, c2)</code>	互换矩阵 <code>A</code> 的第 <code>c1</code> 列和 <code>c2</code> 列

除了这些初等变换外，Maple 在 `linalg` 工具包中还提供了一些矩阵的形状变换函数，可以在利用初等变换解决问题时起到辅助作用，我们也将它们一并列出：

表 4.3 辅助变换函数

函数调用	所作的变换或操作
<code>concat(A, B, ...)</code>	将矩阵 <code>A</code> , <code>B</code> , ... 在水平方向上合并成一个矩阵
<code>stackmatrix(A, B, ...)</code>	将矩阵 <code>A</code> , <code>B</code> , ... 在竖直方向上合并成一个矩阵
<code>row(A, i)</code>	取矩阵 <code>A</code> 的第 <code>i</code> 行
<code>col(A, i)</code>	取矩阵 <code>A</code> 的第 <code>i</code> 列
<code>row(A, i..k)</code>	取矩阵 <code>A</code> 由第 <code>i</code> 行到第 <code>k</code> 行
<code>col(A, i..k)</code>	取矩阵 <code>A</code> 由第 <code>i</code> 列到第 <code>k</code> 列
<code>delrows(A, r..s)</code>	删除矩阵 <code>A</code> 中第 <code>r</code> 行到第 <code>s</code> 行后剩下的子矩阵
<code>delcols(A, r..s)</code>	删除矩阵 <code>A</code> 中第 <code>r</code> 列到第 <code>s</code> 列后剩下的子矩阵
<code>extend(A, m, n, x)</code>	扩展矩阵 <code>m</code> 行 <code>n</code> 列，并用 <code>x</code> 填充
<code>submatrix(A, m..n, r..s)</code>	取矩阵 <code>A</code> 的 <code>m-n</code> 行， <code>r-s</code> 列组成的子矩阵

我们通过一个简单的例题来演示这些函数的用法。

例 4.1 利用初等变换判断矩阵 `A` 是否可逆，如果 `A` 可逆，求 A^{-1}

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 3 \\ 0 & -1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

首先，我们将 \mathbf{A} 和单位阵合成一个大矩阵 \mathbf{AI} ：

```
[> A := matrix( [ [-2, 1, 3], [0, -1, 1], [1, 2, 0] ] );
> AI := concat( A, array(identity, 1..3, 1..3) );
```

$$AI = \begin{bmatrix} -2 & 1 & 3 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 & 0 \\ 1 & 2 & 0 & 0 & 0 & 1 \end{bmatrix}$$

接下来，通过初等变换，把 \mathbf{AI} 的左边部分变成单位阵。先用第一行消去其他两行的第一个元素：

```
[> mulrow( AI, 1, -1/2 );
> addrow( %, 1, 3, -1 );
```

$$\begin{bmatrix} 1 & -\frac{1}{2} & -\frac{3}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 & 0 \\ 0 & \frac{5}{2} & \frac{3}{2} & \frac{1}{2} & 0 & 1 \end{bmatrix}$$

再用第二行消去其他两行的第二个元素：

```
[> mulrow( %, 2, -1 );
> addrow( %, 2, 1, 1/2 );
> addrow( %, 2, 3, -5/2 );
```

$$\begin{bmatrix} 1 & 0 & -2 & -\frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & 1 & -1 & 0 & -1 & 0 \\ 0 & 0 & 4 & \frac{1}{2} & \frac{5}{2} & 1 \end{bmatrix}$$

最后用第三行消去其他两行的第三个元素：

```
[> mulrow( %, 3, 1/4 );
> addrow( %, 3, 1, 2 );
> addrow( %, 3, 2, 1 );
```

$$\begin{bmatrix} 1 & 0 & 0 & -\frac{1}{4} & \frac{3}{4} & \frac{1}{2} \\ 0 & 1 & 0 & \frac{1}{8} & -\frac{3}{8} & \frac{1}{4} \\ 0 & 0 & 1 & \frac{1}{8} & \frac{5}{8} & \frac{1}{4} \end{bmatrix}$$

于是，这个过程说明了 \mathbf{A} 是可逆的，同时也求出了 \mathbf{A} 的逆矩阵——上面最后一个矩阵的右边部分。我们可以加以验证：

```
> multiply( A, submatrix( %, 1..3, 4..6 ) );
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4.3.4 线性方程组的解

线性方程组也是一般代数方程组的一个特例，所以，我们当然也可以用曾经多次接触过的代数方程求解函数 `solve()` 来对它们进行求解。由于线性方程组的特殊性，在 Maple 的线性代数工具包中也有很多专门用于处理线性方程组的函数。

在线性代数中，我们通常是把线性方程组转化为与其等价的矩阵问题来解决的。考虑到这一点，Maple 软件的编制者在 `linalg` 工具包中也提供了在线性代数方程组与矩阵之间相互转换的函数：

```
> eqs := { x + 2*y + 3*z = a,
           8*x + 9*y + 4*z = b,
           7*x + 6*y + 5*z = c };
> A := genmatrix( eqs, [ x, y, z ], b );
```

$$A := \begin{bmatrix} 8 & 9 & 4 \\ 7 & 6 & 5 \\ 1 & 2 & 3 \end{bmatrix}$$

```
> print(b);
```

$$[b, c, a]$$

同样，用函数 `geneqns()` 可以把矩阵转换成指定未知数的线性方程的集合。

```
> geneqns( A, x, b );
```

$$\{8x_1 + 9x_2 + 4x_3 = b, 7x_1 + 6x_2 + 5x_3 = c, x_1 + 2x_2 + 3x_3 = a\}$$

解线性方程组的最简单而且最基本方法是高斯消去法（**Gaussian elimination**），在 `linalg` 工具包中，有与此方法对应的函数 `gausselim()`。作为例子，我们把上面得到的矩阵利用高斯消去法转化成相抵的上三角阵：

```
> concat( A, b );
```

$$\begin{bmatrix} 8 & 9 & 4 & b \\ 7 & 6 & 5 & c \\ 1 & 2 & 3 & a \end{bmatrix}$$

```
> gausselim( % );
```

$$\begin{bmatrix} 8 & 9 & 4 & b \\ 0 & \frac{7}{8} & \frac{5}{2} & a - \frac{1}{8}b \\ 0 & 0 & \frac{48}{7} & c - \frac{8}{7}b + \frac{15}{7}a \end{bmatrix}$$

该函数还可以有两个可选参数，可以用来返回作为高斯消元的副产品的矩阵的秩，以及矩阵的行列式（仅当矩阵为方阵时有效）。

我们发现，在高斯消元的结果矩阵中出现了许多分数，这使表达式看起来不太美观。而且，通过进一步的试验，你会发现随着矩阵的增大，最后的分数的分母将会愈来愈大；不仅如此，如果系数矩阵中包含有符号变量，那么结果矩阵中将出现分式。使用一般高斯消去法时，这些都将是无法避免的。实际上，我们可以在作初等变换时，避免作除法，而是将被消去的行乘以相应的因子。这种方法称为无分式高斯消去法（**fraction-free Gaussian elimination**），Maple 中也有这样的函数——`ffgausselim()`。

```
> ffgausselim( %% );
```

$$\begin{bmatrix} 8 & 9 & 4 & b \\ 0 & 7 & 20 & 8a-b \\ 0 & 0 & 48 & 7c-8b+15a \end{bmatrix}$$

在把系数矩阵转化成相抵的上三角阵后，我们就可以用回代（**back substitution**）的方法求出各个未知数的值。

```
> backsub( % );
```

$$\left[-\frac{1}{6}b - \frac{7}{16}a + \frac{19}{48}c, \frac{1}{4}a + \frac{1}{3}b - \frac{5}{12}c, \frac{7}{48}c - \frac{1}{6}b + \frac{5}{16}a \right]$$

和回代相对，我们还有前代（**forward substitution**）的概念。我们用前代函数 `forwardsub()` 求解下三角矩阵。有了这一对函数，我们就可以利用 LU 分解来解线性方程组了。在 `linalg` 工具包中，LU 分解函数是 `LUdecomp()`。上面，我们使用的是高斯消去法，得到的是上三角阵，需要通过回代得到最终结果。我们也可以利用高斯-约当消去法（**Gauss-Jordan elimination**），把系数矩阵变换成单位阵，就可以直接得到结果了。

```
> gaussjord( concat(A, b) );
```

$$\begin{bmatrix} 1 & 0 & 0 & -\frac{1}{6}b - \frac{7}{16}a + \frac{19}{48}c \\ 0 & 1 & 0 & \frac{1}{4}a + \frac{1}{3}b - \frac{5}{12}c \\ 0 & 0 & 1 & \frac{7}{48}c - \frac{1}{6}b + \frac{5}{16}a \end{bmatrix}$$

上面介绍的实际上都是线性方程组求解的中间步骤，我们当然也可以一步到位，不管它到底用的什么方法，而只要求得到最终的解。这时，我们可以使用线性方程组求解函数 `linsolve()`。更重要的是，这个函数不仅可以用来求解具有唯一解的线性方程组，而且可以在方程组有多解时给出通解，如下例所示。

例 4.4 解方程组

$$\begin{cases} x_1 + 3x_2 + 3x_3 + 2x_4 = -1 \\ 2x_1 + 6x_2 + 9x_3 + 5x_4 = 4 \\ -x_1 - 3x_2 + 3x_3 = 13 \end{cases}$$

```

> A := matrix( [ [1,3,3,2], [2,6,9,5], [-1,-3,3,0] ] );
                                     A :=  $\begin{bmatrix} 1 & 3 & 3 & 2 \\ 2 & 6 & 9 & 5 \\ -1 & -3 & 3 & 0 \end{bmatrix}$ 
> b := vector( [ -1, 4, 13 ] );
> linsolve( A, b );
                                      $[-13-3\_t_1+3\_t_2, -t_1, -t_2, 6-3\_t_2]$ 

```

可以看到，Maple 用辅助变量 $_t_1$, $_t_2$ 给出了方程组的通解。

4.3.5 正定矩阵

在线性代数中，我们把相应的二次型为正定二次型的矩阵称作是正定矩阵，相应的，我们有许多判定矩阵正定性的方法。但是在 Maple 中，判定一个矩阵是否正（负）定十分简单，只需要调用函数 `definite()` 即可。`definite` 可以用来判定数值矩阵的正（负）定性，也可以求出符号矩阵的正（负）定条件。

`definite` 的第一个参数需要判定的矩阵，第二个参数是 'positive_def', 'positive_semidef', 'negative_def', 或者 'negative_semidef' 之一，分别表示正定、半正定、负定和半负定。在判定数值矩阵时，它返回布尔值 `true` / `false`；判定符号矩阵时，它返回一个布尔表达式，表示正/负定的条件，例如：

```

> A := array(1..2,1..2, 'symmetric');
definite(A, 'negative_def');
                                      $A_{1,1} < 0 \text{ and } -A_{1,1}A_{2,2} + A_{1,2}^2 < 0$ 

```

4.3.6 特殊矩阵

利用 `linalg` 工具包中的一些函数，可以直接生成一些特殊的矩阵，而不需要提供指标函数，使用非常方便。我们将它们列成下表，供读者参考。

表 4.4 `linalg` 工具包中的特殊矩阵

函数调用	所生成的矩阵
<code>wronskian(f, v)</code>	由变量 v 的向量函数 f 生成的朗斯基矩阵
<code>bezout(p, q, x)</code>	由变量 x 的多项式 p 、 q 生成的 Bezout 阵
<code>hessian(expr, vars)</code>	由变量向量 $vars$ 的表达式 $expr$ 生成的海赛阵
<code>hilbert(n)</code>	生成 n 阶希尔伯特阵
<code>sylvester(p, q, x)</code>	由变量 x 的多项式 p 、 q 生成的西尔维斯特阵
<code>toeplitz(L)</code>	由表达式有序表 L 生成的 Toeplitz 阵

4.4 线性空间基本理论

4.4.1 基本子空间

我们从线性代数中知道与矩阵 \mathbf{A} 相关的 \mathbf{R}^n 的四个基本子空间——行空间、列空间、化零空间和左零空间。化零空间，实际上就是齐次线性方程组 $\mathbf{Ax} = \mathbf{0}$ 的解空间。我们可以利用上一节中介绍的函数 `linsolve()` 来求得它的通解，在将通解中的辅助变量依次替换成 1，就可以获得它的解空间的一组基。对于左零空间，即方程组 $\mathbf{A}^T \mathbf{x} = \mathbf{0}$ 的解空间，亦可以用同样的方法得到。

另两个基本子空间，行空间和列空间，在 `linalg` 工具包中有专门的函数 `rowspace()` 和 `colspace()`，可以获得它们的基以及维数。

```
> A := matrix(3,2, [2,0,3,4,0,5]);
                                     A :=  $\begin{bmatrix} 2 & 0 \\ 3 & 4 \\ 0 & 5 \end{bmatrix}$ 
> rowspace( A, 'dim' );
                                     {[1,0],[0,1]}
> dim;
                                     2
> colspace( A );
                                      $\left\{ \begin{bmatrix} 1, 0, -\frac{15}{8} \end{bmatrix}, \begin{bmatrix} 0, 1, \frac{5}{4} \end{bmatrix} \right\}$ 
```

函数 `rowspace` 和 `colspace` 的第二个参数是可选参数，用来返回行空间或列空间的维数，它必须是一个变量名——可以是一个未赋值的新变量，也可以是已有值的变量，但要加上延迟求值符 “`'`”（参见第 1 章）。

4.4.2 正交基和 Schmidt 正交化

在欧氏空间中，我们可以定义两个向量的内积（**inner product**），在此基础上，我们还可以定义两个向量的夹角。如向量 α 、 β 的夹角 θ 可以定义为 $\theta = \arccos \frac{(\alpha, \beta)}{\|\alpha\| \|\beta\|}$ 。在 Maple 的 `linalg` 工具包中，有相应的函数可以计算向量的内积和夹角。下面将通过例子来说明它们的用法。

```

[> alpha := vector( [1, 2, -1, 1] );
[> beta := vector( [2, 3, 1, -1] );
[> innerprod( alpha, beta );
                                     6
[> angle( alpha, beta );
                                      $\arccos\left(\frac{2}{35}\sqrt{7}\sqrt{15}\right)$ 

```

在线性代数中, 我们定义内积为零向量相互之间正交; 并且, 我们利用 Schmidt 正交化方法 (Gram-Schmidt orthogonalization process), 由欧氏空间中一组普通基得到两两正交的基。Maple 中相应的函数是 GramSchmidt(), 它的输入参数是由一组向量组成的集合 (或有序表), 它将给出 Schmidt 正交化后的向量集合 (或有序表)。输入的向量必须是线性无关的, 否则, 结果向量间也将线性相关。函数并不对向量进行单位化, 如果需要得到一组正交标准化基, 还需要用 map 方法对这些向量使用单位化函数 normalize()。

例 4.5 用 Schmidt 正交化方法, 由下列向量组构造出一组标准正交向量组

$$(1, 1, -1, -2)^T, (5, 8, -2, -3)^T, (3, 9, 3, 8)^T$$

```

[> { vector([1, 1, -1, -2]),
      vector([5, 8, -2, -3]),
      vector([3, 9, 3, 9]) };
[> map( normalize, GramSchmidt(%) );
{  $\left[\frac{1}{7}\sqrt{7}, \frac{1}{7}\sqrt{7}, -\frac{1}{7}\sqrt{7}, -\frac{2}{7}\sqrt{7}\right], \left[\frac{2}{39}\sqrt{39}, \frac{5}{39}\sqrt{39}, \frac{1}{39}\sqrt{39}, \frac{1}{13}\sqrt{39}\right],$ 
   $\left[\frac{2}{91}\sqrt{2}\sqrt{91}, -\frac{3}{182}\sqrt{2}\sqrt{91}, -\frac{11}{182}\sqrt{2}\sqrt{91}, \frac{3}{91}\sqrt{2}\sqrt{91}\right]$  }

```

向量的 Schmidt 正交化过程实际上就给出了矩阵的 QR 分解: $A = QR$, 其中 Q 是正交矩阵, R 是上三角阵, 这个分解是存在而且唯一的。在 Maple 中, 有 QR 分解函数 QRdecomp(), 利用它可以完成数值或符号矩阵的 QR 分解。

```

[> A := matrix(3, 3, [1, 1, 1, 2, 3, 0, 2, 1, 1]);
                                      $A := \begin{bmatrix} 1 & 1 & 1 \\ 2 & 3 & 0 \\ 2 & 1 & 1 \end{bmatrix}$ 
[> R := QRdecomp(A, Q='Q', rank='r');
                                      $R := \begin{bmatrix} 3 & 3 & 1 \\ 0 & \sqrt{2} & -\frac{1}{2}\sqrt{2} \\ 0 & 0 & \frac{1}{2}\sqrt{2} \end{bmatrix}$ 

```

QRdecomp 的返回值是上三角阵 R , 正交阵 Q 和矩阵的秩可以通过函数的可选参数返回, 但这里这两个参数必须指明它们的形参名—— Q 和 $rank$, 这和 Fortran 语言的函数调用有一定的相似之处。有关这方面的知识, 我们将在程序设计章节中专门介绍。

```
> print(Q);
```

$$\begin{bmatrix} \frac{1}{3} & 0 & \frac{2}{3}\sqrt{2} \\ \frac{2}{3} & \frac{1}{2}\sqrt{2} & -\frac{1}{6}\sqrt{2} \\ \frac{2}{3} & -\frac{1}{2}\sqrt{2} & -\frac{1}{6}\sqrt{2} \end{bmatrix}$$

那么这里的 Q 到底是不是一个正交阵呢？可以使用函数 `orthog()` 来判定它的正交性：

```
> orthog(Q);
```

true

4.4.3 线性方程组的最小二乘解

在实际问题中，由于误差或者其他各方面的原因，很容易出现无解的方程组。但问题还是要解决的，我们必须给出一个“最优”的近似解。在线性代数中，我们通常采用最小二乘解（**least-squares solution**）。`linalg` 中对应的函数是 `leastsqrs()`，它有两种调用格式，可以用来求解矩阵形式的方程组，和 `linsolve` 类似；也可以直接求借用解析表达式给出的方程组，这时调用的格式和 `solve` 函数一样。

例 4.6 求不相容方程组

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

在最小二乘意义下的最优解。

```
> A := matrix( 3, 2, [1,0,0,1,1,1] );
> b := vector( [1, 1, 0] );
> leastsqrs( A, b );
```

$$\begin{bmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

4.5 特征值、特征向量和相似标准型

4.5.1 矩阵的相似

我们知道，对于方阵 A , B ，如果存在一个可逆的方阵 P 满足 $B = P^{-1}AP$ ，则称 A , B 相似。在 Maple 中，我们可以利用 `linalg` 中的函数 `issimilar()` 判断两个矩阵是否相似。作为例子，我们来验证一个简单的定理——在 A 可逆时， AB 和 BA 相似。


```
[> A := matrix(2,2): B := matrix(2,2):
> issimilar( A&*B, B&*A );
true
```

这里，Maple 作了一定的假设，由于 \mathbf{A} 的行列式是一个符号表达式，Maple 在判定时假设它为非零常数。不过更多的情况下，这个函数是用在数值矩阵的相似判定上，同时，它还可以求出可逆矩阵 \mathbf{P} 。

```
[> A := matrix( 3, 3, [0,1,0,0,0,1,0,0,0] );
A :=  $\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ 
> issimilar( A, transpose(A), 'P' );
true
> print(P);
 $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$ 
```

4.5.2 特征值和特征向量

特征值问题可以说是最常见的线性代数问题了，许多数学问题（比如常微分方程组的求解、二阶张量的主分量）和工程问题（比如振动模态辨识、系统辨识）都会涉及到特征值问题。求方阵的特征值，实际上是求与之相似的对角阵。在线性代数中，求解矩阵 \mathbf{A} 的特征值，我们用求解方程 $\det(\lambda \mathbf{I} - \mathbf{A}) = 0$ 方法。矩阵 $\lambda \mathbf{I} - \mathbf{A}$ 称为 \mathbf{A} 的特征矩阵（**characteristic matrix**），而行列式 $\det(\lambda \mathbf{I} - \mathbf{A})$ 展开得到的多项式称为特征多项式（**characteristic polynomial**），在 linalg 工具包中，分别有函数 charmat() 和 charpoly() 与它们相对应。

```
[> A := matrix( 3, 3, [1,2,2,2,1,2,2,2,1] );
A :=  $\begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix}$ 
> charmat( A, lambda );
 $\begin{bmatrix} \lambda-1 & -2 & -2 \\ -2 & \lambda-1 & -2 \\ -2 & -2 & \lambda-1 \end{bmatrix}$ 
> charpoly( A, lambda );
 $\lambda^3 - 3\lambda^2 - 9\lambda - 5$ 
```

得到了特征多项式以后，我们就可以利用 solve 求出矩阵的特征值了。不过，在 linalg 中也有可以直接求出矩阵的特征值和特征向量的函数 eigenvalues() 和 eigenvectors()。

```

[> eigenvals(A);
                                     5, -1, -1
[> eigenvectors(A);
                                     [5, 1, {[1, 1, 1]}], [-1, 2, {[0, 1, -1], [1, 0, -1]}]

```

我们看到，特征向量函数 **eigenvectors** 在给出特征向量的同时，还给出了对应的特征值和特征值的重数。一般情况下，如果同时需要得到矩阵的特征值和特征向量，可以直接调用函数 **eigenvectors**。这两个函数不仅可以用来求解数值矩阵的特征值问题，也可以处理符号矩阵；在处理符号矩阵时，它们通常都会以根式的形式给出结果，如果加上可选参数 'implicit'，结果就以 **RootOf** 的形式给出。

函数 **eigenvals()** 不仅可以求解普通特征值问题，还可以求解广义特征值问题，也就是求解方程 $\det(\lambda \mathbf{C} - \mathbf{A}) = 0$ 。广义特征值问题的调用格式是 **eigenvals(A, C)**。

由特征值理论，任何一个实对称阵，都可以用求特征值的方法将它对角化；但对于非对称矩阵，或者是复矩阵，就没有这样的保证了。不过对于任意复矩阵 **A**，还是可以把它化成相似的约当标准型 **J**，使得 $\mathbf{P}^{-1}\mathbf{A}\mathbf{P} = \mathbf{J}$ 。linalg 中的约当标准型函数是 **jordan()**。它在求得矩阵的约当标准型的同时，还可以给出转换矩阵 **P**。

```

[> A := matrix( 3, 3, [2,-1,1,2,2,-1,1,2,-1] );
                                     A =  $\begin{bmatrix} 2 & -1 & 1 \\ 2 & 2 & -1 \\ 1 & 2 & -1 \end{bmatrix}$ 
[> jordan( A, 'P' );
                                      $\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$ 
[> print(P);
                                      $\begin{bmatrix} 0 & 1 & 1 \\ 3 & 2 & 0 \\ 3 & 1 & 0 \end{bmatrix}$ 

```

约当标准型的定义是：由若干约当块组成的块状对角阵，即

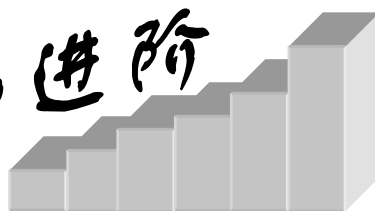
$$\mathbf{J} = \text{diag}(\mathbf{J}(\lambda_1, p_{11}), \mathbf{J}(\lambda_1, p_{12}), \dots, \mathbf{J}(\lambda_1, p_{l_1}), \dots, \mathbf{J}(\lambda_s, p_{s1}), \mathbf{J}(\lambda_s, p_{s2}), \dots, \mathbf{J}(\lambda_s, p_{st_s}))$$

其中，约当块 $\mathbf{J}(\lambda, k)$ 的定义如下：

$$\mathbf{J}(\lambda, k) = \begin{pmatrix} \lambda & 1 & & & \\ & \lambda & \ddots & & \\ & & \ddots & \ddots & \\ & & & \ddots & 1 \\ & & & & \lambda \end{pmatrix}_{k \times k}$$

在 Maple 中，可以调用函数 **JordanBlock(λ, k)** 可以构造约当块 $\mathbf{J}(\lambda, k)$ ，利用它和 **diag** 函数连用，就可以生成具有约当标准型的矩阵。

起步与进阶



第

编程初步

五

章

本章将介绍在 Maple 中进行程序设计的基础知识,以便使大家能够迅速地掌握 Maple 程序的主要成分和基本结构。通过这一章的学习,可以对 Maple 子程序的语法,结构有初步的认识,为进一步学习 Maple 种的程序设计打下基础。而且,必要的 Maple 编程基础也是灵活地利用各种工具函数来解决问题的先决条件。

本章具体包括以下内容:

- 🕒 用箭头操作符定义函数
- 🕒 Maple 的程序构成
- 🕒 循环和分支结构
- 🕒 局部变量和全局变量
- 🕒 过程的递归调用

在前面的几章中，我们一直使用的是 Maple 的交互式命令环境。所谓交互式命令环境，就是一次输入一条或几条命令，然后一按回车，这些命令就被执行了，执行的结果被显示在同一个可执行块中。也许对很大一部分用户来说，利用一个交互式的命令环境解决问题已经足够了；但如果要充分地利用 Maple 的强大功能，仅仅通过这一手段是不够的，无论是从大规模问题的计算效率上考虑，还是从对 Maple 工具函数库的扩展上考虑，都需要掌握一定的程序设计知识。可别小看了 Maple 语言，Maple 中的大部分库函数和工具包中的函数都是用 Maple 语言写成的。

实际上，在前面的介绍中，我们已经介绍过简单的 Maple 程序了，我们曾经用箭头操作符 “->” 定义过最简单的函数。而 Maple 中的函数，都是 Maple 子程序（procedure）的特里，它们是具有返回值的子程序。下面，我们就详细地来介绍 Maple 中的函数和子程序，它们的定义和结构。

5.1 箭头操作符

Maple 是一个符号演算软件，带未知或者已知字母变量的表达式是它的基本数据形式，经过前面几章的学习，你一定已经对 Maple 中的表达式运算烂熟于心了；但同时，你也许会生出一个不太正确的观念——既然表达式中可以包含未知变量，那它不久已经是函数了吗？的确，在数学上我们可以这样理解，但对于 Maple 这个计算机软件来说，两个对象如果具有不同的数据结构，那它们就不应该是相同的对象。

在学习箭头操作符以前，你也许尝试过用这样的方法定义一个函数：

```
> f(x) := a * x * exp(-x);
```

$$f(x) = a x e^{(-x)}$$

Maple 接受了这样的赋值语句，一切都很正常，看起来如我们所料。但我们是不是真的定义了一个函数呢？

```
> f(x), f(0), f(1/a);
```

$$a x e^{(-x)}, f(0), f\left(\frac{1}{a}\right)$$

我们发现，这样的方法定义的不是一个函数——我们不能把我们所定义的“自变量”，或者说“参数”换成别的变量或表达式，一换它就不认识了。F 的确确是一个函数，只不过是一个没有具体定义的函数。

```
> print(f);
```

```
proc() option remember; 'procname(args)' end
```

我们所作的赋值运算，只不过在函数 f 的记忆表（remember table）中加入了函数在 x 上的值，当我们把自变量换作 0 或者 1/a 时，f 的记忆表中就没有对应的表项，所以 Maple 只能给出抽象的函数表达式。

当然，也许在前面接触到箭头操作符时你就已经注意它了，那你一定不会反上面的错误。

因为箭头操作符同样是直观易懂，和我们所熟知的数学语言几乎完全一样。例如，我们可以用箭头操作符定义上面这个函数：

```
> f := x -> a*x*exp(-x);
```

$$f := x \rightarrow a x e^{(-x)}$$

```
> f(x), f(0), f(1/a);
```

$$a x e^{(-x)}, 0, e^{-\frac{1}{a}}$$

多变量的函数也可以同样予以定义，只不过我们把所有的自变量写成一个序列，并用一对括号“()”将它们括起来。

```
> f := (x, y) -> a*x*y * exp( x^2 + y^2 );
```

$$f := (x, y) \rightarrow a x y e^{(x^2 + y^2)}$$

这里的括号是必须的，因为箭头运算符作为一个运算符，优先级高于分隔符“,”，所以Maple会这样来理解它：

```
> f := x , (y -> a*x*y * exp( x^2 + y^2 ));
```

$$f := x, y \rightarrow a x y e^{(x^2 + y^2)}$$

```
> f;
```

$$x, y \rightarrow a x y e^{(x^2 + y^2)}$$

f成了由一个变量和一个但变量函数组成的序列了，这当然不是我们想要的。对于单变量函数，箭头操作符的一般用法是：

参数 -> 函数表达式

而对于无参数的函数或者多变量函数，箭头操作符的用法是：

(参数序列) -> 函数表达式

无参数函数也许不好理解，它可以用来定义常值函数，我们通过一个例子一看就可以明白：

```
> E := ( ) -> exp(1);
```

$$E := () \rightarrow e$$

```
> E();
```

$$e$$

```
> E(x);
```

$$e$$

还可以看出，Maple对于多余的参数采取的是置之不理的态度，而不像通常的编程语言一样有严格的检查，因此需要格外引起注意。

5.2 最简单的子程序

在前一节中，我们介绍了箭头操作符，利用它可以生成最简单的函数——单个语句的函

数。严格意义上来讲，它还不能被称作程序设计，但它所生成的数据对象是子程序（procedure）。简单地说，一个 Maple 子程序就是一组预先编排好的命令。我们先由一个最简单的子程序来看看 Maple 子程序的结构：

```
> half := proc(x)
    evalf(x/2);
end;

half:=proc(x) evalf(1/2*x) end
```

这个子程序只有一个参数，在子程序内部它的名称是 x 。在这个程序中，我们计算了 $x/2$ 的浮点近似值，由于这条语句就是该子程序的最后一条语句，所以该子程序就返回算得的近似值。上面的子程序是 Maple 最简单的子程序结构，仅仅在 `proc(…)` 和 `end` 中加入我们在预算步骤中需要的一条或者多条命令就可以了。Maple 会把最后一个语句的结果作为整个子程序的返回结果。

我们再来实现稍稍复杂一点的功能，例如，我们要完成下面的计算任务：

```
> a := 10809832/2431;
> evalf(a/2);
```

我们需要把它编制成 Maple 子程序（虽然这简直是杀鸡用牛刀），还是用上面的方法，把它们放在 `proc` 和 `end` 的中间。

```
> f := proc()
    a := 10809832/2431;
    evalf(a/2);
end;
Warning, `a` is implicitly declared local
f:=proc() local a; a:=982712/221; evalf(1/2*a) end
```

我们解剖麻雀，从这个简单的例子中来看看 Maple 子程序中都有些什么：

- ✧ 我们把定义的子程序赋值给 f ，以后我们就可以用子程序名 f 来调用它了。
- ✧ 子程序一律都以 `proc()` 开头。括弧中什么也没有，表示这个子程序没有任何输入参数。
- ✧ 子程序中的每一个语句都用分号隔开（当然也可以用冒号，结果是相同的）。
- ✧ 在定义完子程序之后，Maple 会显示它对于该子程序的解释（除非在 `end` 后用冒号结尾），它的解释和你的定义是等价的，但并不一定完全相同。
- ✧ Maple 会自动地把除了参数以外的变量都作为局部变量（local variable），这就是说，它们仅仅在这个子程序的定义中有效，和子程序以外的任何同名变量毫不相干。有关局部变量和全局变量，我们将在下一节中详细介绍。

由于子程序的定义比较长，一般不能在一行之内输入完，换行是，为了避免出现“语句未完”的警告，可以使用 `Shift + Enter` 换行。

在定义了一个子程序之后，我们当然要执行它，执行一个子程序的方法和执行任何 Maple 的系统子程序完全一样——程序名再加上一对括弧，其中包含着调用的参数。即使子程序没有参数，括弧还是不能省略的。

我们曾经说过，和 Maple 中的数组、映射表一样，Maple 的子程序也不能自动地求值。如果需要察看一个已经定义好的子程序的具体定义，可以使用我们接触过多次的 `eval` 命令：

```
[> eval(f);
      proc() local a; a := 982712 / 221; evalf(1 / 2*a) end
```

5.3 局部变量和全局变量

Maple 中的全局变量，是指那些在交互式命令环境中定义和使用的变量，比如我们在前面几章中用到的所有变量都是全局变量。而在编写子程序时，你会需要定义一些变量，这些变量只在子程序内部使用，我们称其为局部变量。当 Maple 执行子程序时，所有和局部变量同名的全局变量都保持不变，而不管在子程序中给局部变量赋予了何值。前面我们已经知道，在子程序中，变量默认情况下都是局部变量。如果要把它定义成全局变量，需要用关键字 `global` 在程序最开始加以申明；对应的，局部变量也可以用 `local` 加以申明，虽然这不是必要的，但申明变量是一个好习惯。

所谓变量的作用域，是指所有可以对该变量操作的子程序和语句的集合，对于简单子程序（无嵌套的子程序）来说，作用域无外乎两种——要么在一切地方都有效（全局变量），要么仅仅在一个子程序中有效（局部变量）。对于嵌套子程序，情况要稍稍复杂一点，我们将在后面详细介绍。

我们通过两个例子来演示局部变量和全局变量的不同。首先，为了观察子程序对全局变量的影响，我们在子程序外设定一个变量的值：

```
[> a := 1;
      a := 1
```

接着，我们分别定义两个子程序。第一个，定义 `a` 为其局部变量：

```
[> f := proc()
      local a;
      a := 10809832/2431;
      evalf(a/2);
    end;
```

然后执行它，再看看外面的变量 `a` 的值。可以看到，`a` 没有受子程序中同名局部变量的干扰：

```
[> f();
      2223.330317
[> a;
      1
```

然后再试试把程序中的 `a` 定义成全局变量：

```
[> g := proc()
      global a;
      a := 10809832/2431;
      evalf(a/2);
    end;
[> g();
      2223.330317
```

```
> a;
          982712
          ----
          221
```

可以看到，外面的 a 也跟着程序中的 a 变了。

不过，像这样地使用全局变量当然没有什么实际意义；而在实际程序设计中，全局变量有时还是必不可少的，例如在子程序中需要输入或输出大量的数据时，仅依靠参数和返回只有时显得力不从心，这是使用全局变量就可以在一定程度上带来方便。

在子程序设计中更重要的一类变量是子程序的输入参数，它既不是全局的，又不是局部的。对于子程序内部，它是形式参数；也就是说，它的具体取值尚未被确定，它的出现，将来在调用时都会被替换成真正的参数值。而对于子程序外部，它们仅仅表示该子程序接受的参数的多少，而对于具体的参数值毫无关系。

5.4 基本程序结构

几乎任何一种高级语言都或多或少地具有程序结构。因为为了完成一项复杂的任务，仅仅按照语句顺序依次地执行是远远不够的。我们需要程序在特定的地方能够跳转、分叉、循环……，与此同时，程序结构就应运而生了。这一节中，我们将依次介绍 Maple 语言中的基本程序结构。

5.4.1 for 循环

在程序设计中，我们常会遇到这种情况，需要把相同或者类似的语句连续执行多次。举个例子来说吧，比如我们可以这样计算前 5 个自然数的和：

```
> total := 0:
   total := total + 1:
   total := total + 2:
   total := total + 3:
   total := total + 4:
   total := total + 5:
                                     total := 15
```

显然这是一个笨办法，我们可以通过 for 循环结构更便捷地编写这段程序：

```
> total := 0:
   for i from 1 to 5 do
       total := total + i:
   od:
```

Maple 的 for 循环结构很接近于英语语法，所以我们很容易看懂上面例子的意思。例子中的 i ，我们称它为循环变量，是用于循环计数的。上面的例子中 i 从 1 变到 5，总共执行了 5 次。Do 后面的部分，称为循环体，它就是需要反复执行的语句，可以是一条语句，也可以是多条语句。在循环体中，我们也可以引用循环变量。循环体的结束标志，也就是整个 for 循环结构的结束标志，是两个字母——od——很奇怪，它不是一个英语单词——但它比

英语单词更好理解，既然循环体的开始标志是 **do**，那么它的结束标志就是把 **do** 反一反，**od** 是也；就像 C 语言中的一对花括弧，对称地将循环体包在中间。类似于这样界定程序块的方法，我们在其他的结构中也能看到，这也许是 Maple 的一个特色吧。

我们可以把 for 循环结构的语法总结如下：

```
for 循环变量 [from 初始值] to 终了值 do
    循环体
od
```

for 循环的初始值和终了值必须都是数值型的，而且必须是实数。如果循环初始值是 1，那么就可以省略不写。

对于这 5 次的重复，循环似乎可有可无；但实际的科学计算中，重复次数往往成千上万，或者重复的次数尚未确定，循环就必不可少了。比如，我们可以编制一个求前 n 个自然数的和的子程序：

```
> SUM := proc(n)
    local i, total;
    total := 0;
    for i from 1 to n do
        total := total + i;
    od;
end;
SUM := proc(n) local i, total; total := 0; for i to n do total := total + i od end
```

作为实验，我们用这个子程序来计算一下前 100 个自然数的和：

```
> SUM(100);
5050
```

完全正确。当然了，对于这样的简单问题，没有必要兴师动众编制程序，完全可以利用 Maple 函数更简单地完成：

```
> add(n, n=1..100);
5050
```

5.4.2 分支结构

另一个重要的基本程序结构是分支结构，分支结构是程序在执行时，依据条件的不同，分别执行两个或多个不同的程序块，所以我们又常常把它称作条件语句。这种结构在数学中更是屡见不鲜，比如我们要计算一个变量的绝对值，就必须对它作判断，而依据判断的结果分别处理。作为分支结构的简单例子，我们把这个函数简单地实现一下，为了防止和 Maple 的内部函数混淆，我们将它命名为 ABS：

```
> ABS := proc(x)
    if x < 0 then
        -x;
    else
        x;
    fi;
end;
ABS := proc(x) if x < 0 then -x else x fi end
```

我们的绝对值函数的简单实现，对于计算实数范围内的数值型变量是没有问题的：

```
[> ABS(1), ABS(-2.5);
                                     1, 2.5
```

但是，这个函数对于非数值型的参数却是无能为力了：

```
[> ABS(a);
Error, (in ABS) cannot evaluate boolean
```

从错误信息中我们看到，Maple 在执行子程序 ABS 时发生了错误——无法对分支结构中的布尔表达式“ $x < 0$ ”求值，所以 Maple 无法决定下面执行哪个程序段，程序被迫中断了。这不是我们编制程序的本意，特别是对于 Maple 这个符号演算系统的编程来说，这种情况不是我们想看到的。对于符号变量 a ，我们期望的结果是 $ABS(a)$ 。我们知道，可以用延迟求值操作符“ $'$ ”得到这样的表达式结果。

我们还可以再用一次条件语句，来判断参数是否为数值的变量：

```
[> ABS := proc(x)
      if type(x, numeric) then
        if x < 0 then
          -x;
        else
          x;
        fi;
      else
        'ABS'(x);
      fi;
    end;
```

这里，我们用到了一个 if 语句的嵌套，也就是一个 if 语句处于另一个当中；这样的结构可以用来判断复杂的条件。我们还可以利用多重嵌套的条件结构来构造更为复杂的函数，例如下面的分段函数：

```
[> HAT := proc(x)
      if type(x, numeric) then
        if x <= 0 then
          0;
        else
          if x <= 1 then
            x;
          else
            0;
          fi;
        fi;
      else
        'HAT'(x);
      fi;
    end;
```

尽管我们用了不同的缩进来表示不同的层次关系，这段程序还是很难读懂。对于这种多分支结构，我们可以用另一种方式来书写。我们可以在第二层的 if 语句中使用 **elif (else if)**，这样，就可以把多个分支形式上写在同一个层次中，以便于阅读。例如上面这个程序，就可以改写为更明了的形式：

```

> HAT := proc(x)
    if type(x, numeric) then
        if x<=0 then 0;
        elif x<=1 then x;
        else 0;
        fi;
    else
        'HAT'(x);
    fi;
end:

```

和许多高级语言一样，这种多重分支结构理论上可以多到任意多重。

我们再回过头去，看看我们在前面定义的绝对值函数，它似乎已经很完美了。但是，在作进一步的试验，我们又发现了它的缺陷，比如我们用它来求一个乘积的绝对值：

```

> ABS(a*b);
ABS(a b)

```

但我们知道，乘积的绝对值可以化成绝对值的积，即 $|a b| = |a| |b|$ 。根据前面学过的，我们知道，可以用 `map` 函数来实现这一点：

```

> map(ABS, a*b);
ABS(a) ABS(b)

```

我们可以用 `type(..., `*`)` 来判断一个表达式是否为乘积，进而将其化简。利用多分支结构，我们把 `ABS` 子程序扩展如下：

```

> ABS := proc(x)
    if type(x, numeric) then
        if x<0 then -x else x fi;
    elif type(x, `*`) then
        map(ABS, x);
    else
        'ABS'(x);
    fi;
end:

```

这一改变对于计算含有数值乘积的绝对值特别有用：

```

> ABS(-1/2*b);
1/2 ABS(b)

```

5.4.3 while 循环

前面我们已经介绍过 `for` 循环，`for` 循环在那些已知循环次数，或者循环次数可以用简单表达式计算的情况下比较适用；但有时循环次数并不能简单地给出，我们要通过计算，判断一个条件决定是否继续循环，这时，可以使用 `while` 循环。`While` 循环有以下形式的标准结构：

while 条件表达式 **do** 循环体 **od**

Maple 首先判断条件是否成立，如果成立，就一遍又一遍地执行循环体，直到条件不成立为

止。

作为 while 循环的例子，我们来编制一个简单的程序，用辗转相除法计算两个自然数的最大公约数。

```
> GCD := proc(a::posint, b::posint)
    local p, q, r;
    p := max(a, b);
    q := min(a, b);
    r := irem(p, q);
    while r <> 0 do
        p := q;
        q := r;
        r := irem(p, q);
    od;
    q;
end;
```

在上面这个程序的参数 a、b 后面，我们用两个冒号指定了输入的参数类型。这样，Maple 就会自动检查调用时的参数类型，并在类型不匹配时返回错误信息。

```
> GCD(a, 21);
Error, GCD expects its 1st argument, a, to be of type posint, but
received a
```

5.5 递归子程序

正如在一个子程序中我们可以调用其他的子程序一样（比如系统内部函数，或者我们已经定义好的子程序），一个子程序也可以在它的内部调用它自己，这样的子程序我们称为递归子程序。在数学中，我们常常可以看到递归定义的例子，比如菲波那契（Fibonacci）数列，就是用递归的方式定义的： $f_0 = 0$ ， $f_1 = 1$ ， $f_n = f_{n-1} + f_{n-2}$ ($n \geq 2$)。同样，在 Maple 中，我们可以利用递归子程序求菲波那契数列的第 n 项：

```
> Fibonacci := proc(n::nonnegint)
    if n <= 1 then
        n;
    else
        Fibonacci(n-1) + Fibonacci(n-2);
    fi;
end;
```

我们用它来求菲波那契数列的前 20 项：

```
> seq( Fibonacci(i), i=0..19 );
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181
```

不管你用的是什么样的计算机，上面的计算一定化了相当长的时间（当然是对计算机而言的，1~2 秒已经是非常长的了）。不妨用 time 来看一下计算 f_{20} 所用的时间：

```
> time( Fibonacci(20) );
.504
```

这是因为上面的程序把相同的计算进行了多次，比如计算 f_{19} 时，需要计算 f_{18} 和 f_{17} ；而计算 f_{18} 时，又要计算 f_{17} 和 f_{16} ……。这样，就等于计算了两次 f_{17} ，三次 f_{16} ，……。如何才能避免这样的重复运算呢，我们可以让 Fibonacci 子程序记住已经计算过的结果。可以用在子程序定义时在最前面加入 `option remember`，这样，Maple 就会把计算过的结果记入该子程序的记忆表中，重新计算就会大大加快，就像大部分内部函数一样。有关记忆表的内容，我们将在后面专门讨论。下面是修改后的 Fibonacci 子程序：

```
[> Fibonacci := proc(n::nonnegint)
      option remember;
      if n<=1 then
          n;
      else
          Fibonacci(n-1) + Fibonacci(n-2);
      fi;
  end;
```

这个版本的 Fibonacci 子程序速度比原来的提高了很多：

```
[> time( Fibonacci(20) );
                                     .347
```

但是，没有事物是十全十美的，在速度上提高了，就一定会在其他方面有所付出，记忆表会占去宝贵的系统资源。递归程序也是一样，它可以简化算法，使程序简明易懂；但它也有不足之处，调用递归子程序受堆栈的限制，不可能无止境地调用下去。比如，我们在 Microsoft Windows98 环境下就不能用上面的子程序求得 f_{2000} 。

```
[> Fibonacci(2000);
[Error, (in type/nonnegint) too many levels of recursion
```

所以，我们在编写递归子程序时一定要注意，不能无限制地递归，也就是说，递归必须要有一个结束；否则，必然会耗尽系统的堆栈空间而导致错误。

从数据结构课上我们知道，递归程序理论上都可以用循环实现，比如我们可以用循环改写上面的 Fibonacci 程序。

```
[> Fibonacci := proc(n::nonnegint)
      local temp, fnew, fold, i;
      if n<=1 then
          n;
      else
          fold := 0;
          fnew := 1;
          for i from 2 to n do
              temp := fnew + fold;
              fold := fnew;
              fnew := temp;
          od;
      fi;
  end;
```

利用循环，程序不如以前那么易懂了；但同时带来的好处也是不可忽视的，循环结构的程序不仅不会受到堆栈的限制，而且计算速度也有了很大的提高：

```
[> time( Fibonacci(2000) );
                                     .577
```

我们知道，Maple 子程序默认情况下把最后一条语句的结果作为子程序的结果返回；然而我们也可以利用 RETURN 命令来显式的返回结果，比如前面的递归形式的程序，可以等价地写成：

```
[> Fibonacci := proc(n::nonnegint)
    option remember;
    if n<=1 then RETURN(n); fi;
    Fibonacci(n-1) + Fibonacci(n-2);
end:
```

程序进入 $n \leq 1$ 的分支中，执行到 RETURN 命令就跳出程序，而不会接着执行后面的语句了。使用 RETURN 命令在一定程度上可以增加程序的可读性。

5.6 子程序中的求值

在 Maple 子程序中的语句，求值的方式和交互式环境中的求值有所不同。这在某种程度上是为了提高子程序的执行效率而考虑的。在交互式环境中，Maple 对于一般的变量和表达式都进行完全求值（除了数组、映射表等数据结构外）。比如你先将 b 赋给 a，再将 c 赋给 b，则最终 a 的值也指向 c。

```
[> a := b;
[> b := c;
[> a + 1;
                                     c + 1
```

但在子程序内部，情况就不同了，我们将上面的语句写到一个子程序中：

```
[> f := proc()
    local a, b;
    a := b;
    b := c;
    a + 1;
end:
```

运行的结果似乎令人吃惊：

```
[> f();
                                     b + 1
```

这是因为 a 和 b 都是局部变量，Maple 对于子程序中的局部变量只进行一层的求值。下面，我们对于子程序中不同的变量，系统地介绍子程序中的求值机制

5.6.1 参数

子程序的参数，就是那些在 proc() 的括弧中的变量，我们前面说过，参数是一类特殊的变量，在调用子程序时，会把它替换成实际的参数。

我们来看下面的程序，它把第一个参数的平方赋给第二个参数（第二个参数的类型是 name，也就是说必须是未被赋值的变量）。

```

[> sqr1 := proc(x::anything, y::name)
    y := x^2;
end:
[> sqr1(d, ans);
                                      $d^2$ 
[> ans;
                                      $d^2$ 

```

我们再来试试别的参数，比如用我们在前面赋值过的 a 作为第一个参数；第二个参数还用 ans ，但这次需要加上单引号了（想想为什么）。

```

[> sqr1(a, 'ans');
                                      $c^2$ 
[> ans;
                                      $c^2$ 

```

看来，Maple 清楚地记得 a 被赋值为 b ，而 b 被赋值为 c 。这里，显然 Maple 对参数 a 进行了求值，但是，这个求值究竟发生再生么时候呢？我们用调试工具 `stopat` 让程序在进入 `sqr1` 时停止。

```

[> stopat(sqr1);
                                     [sqr1]
[> sqr1(a, 'ans');
sqr1:
    1*   y := x^2

```

这时候程序中的形式参数 x 已经是 c 了。我们可以在调试提示符 `DBG>` 下键入需要察看的变量名 x ：

```

[DBG> x
c
sqr1:
    1*   y := x^2
[DBG> cont
                                      $c^2$ 
[> unstopat(sqr1):

```

看来，Maple 在进入子程序以前就已经对参数进行了求值了。因为是在子程序外进行求值，所以求值规则服从调用是所在的环境。上面是在交互式环境下调用 `sqr1`，所以得到的是 c ；作为对比，我们来看看如果在程序内部调用上面的 `sqr1` 子程序会有什么不同。

```

[> g := proc()
    local a, b, ans;
    a := b;
    b := c;
    sqr1(a, ans);
end:
[> g();
                                      $b^2$ 

```

因为这次调用是在程序内部，所以只进行一层求值，进入 `sqr1` 时参数值为 `b`。

Maple 对于子程序的参数，都只在调用之前进行一次求值，而在子程序内部出现的地方都用这次求值的结果替换，而不再重新进行求值。比如我们这样修改前面的子程序 `sqr1`：

```
[> sqr1 := proc(x::anything, y::name)
      y := x^2;
      y + 1;
end:
[> sqr1(d, 'ans');
                                     ans + 1
```

可见，对参数赋值的作用只有一个——返回一定的信息——因为在子程序内部，永远不会对参数求值。我们可以认为，参数是一个 0 层求值的变量。

5.6.2 局部变量

正如我们在前面所见，Maple 对于局部变量只进行一层的求值，也就相当于用 `eval(a, 1)` 得到的结果。

这种求值机制不仅可以提高效率，而且还有着更重要的作用。比如在程序中，我们往往需要把两个变量的值交换，一般我们使用一个中间变量来完成这样的交换。但如果求值机制和交互式环境中一样，将达不到我们的目的。不妨试一试，在交互式环境下，我们企图用这样的方法交换两个未被赋值的变量会有什么后果：

```
[> temp := p;
[> p := q;
[> q := temp;
                                     q := q
```

5.6.3 全局变量

不管是在交互式环境下，还是在程序中，Maple 对于每一个全局变量都进行完全求值，除非它是数组，映射表，或者子程序。对于数组，映射表和子程序，Maple 采用赋值链中的上一名称来求值。

除了上面这些特殊数据对象还有下面将有介绍的两个特例外，总结起来，Maple 对子程序的参数进行 0 层求值；对于局部变量进行 1 层求值；而对于全局变量，则进行完全求值。

5.6.4 特例

对于上面说的求值规则，在 Maple 中还有两个特例，需要引起注意。

其一是“同上”操作符“%”，就其作用域来说，它应该属于局部变量。在进入一个子程序时，Maple 会自动地把该子程序中的%设置成 NULL（空）。但是，对于这个“局部变量”，Maple 不遵循上面的规则，无论在哪儿，都会将其完全求值。在交互式环境中当然无可非议了，我们下面通过一个例子说明它在子程序中的求值机制：


```

> f := proc()
    local a, b, c, d;
    print("At the begining, [%] is", %);
    a := b;
    b := c;
    c := d;
    a + 1;
    print("Now [%] is", %);
end:
> f();

```

"At the begining, [%] is"
"Now [%] is", $d + 1$

我们看到，尽管在子程序内部，局部变量 a 仅进行一层求值，但指代 $a + 1$ 的同上操作符却进行了完全求值，得到 $d + 1$ 的结果。

第二个特例是环境变量。所谓环境变量就是像 `Digits` 这样的变量。从作用域来看，它们应该算作全局变量。在求值上，它们也像其他全局变量一样，始终都是完全求值。但是如果在子程序中间对环境变量进行了设置，那么在退出改子程序时，系统会自动地将其恢复成进入子程序时的状态，以保证当前运行时环境。正因为此，我们称其为环境变量。我们通过一个简单的例子来说明这一点：

```

> f := proc()
    print( "Entering f.  Digits is", Digits );
    Digits := 20;
    print( "Now Digits has become", Digits );
end:
> g := proc()
    print( "Entering g.  Digits is", Digits );
    Digits := 100;
    print( "Now Digits is", Digits );
    f();
    print( "Back in g from f. Digits is", Digits );
end:

```

我们知道，默认情况下 `Digits` 的值是 10：

```

> Digits;
10
> g();
"Entering g. Digits is", 10
"Now Digits is", 100
"Entering f. Digits is", 100
"Now Digits has become", 20
"Back in g from f. Digits is", 100
> Digits;
10

```

而从子程序 `g` 中返回时，`Digits` 又被自动地设成了原来的值。如果你需要自己定义一些具有这样的特性的环境变量，也十分简单——Maple 会把一切以 `_Env` 开头的变量认作是环境变量。

5.7 嵌套子程序

在很多情况下，我们可以在一个子程序的内部定义另一个子程序；实际上，我们在写这样的程序时常常没有意识到它是嵌套子程序。用于对每一个元素操作的 `map` 命令我们在交互式环境下已经非常熟悉了。再程序设计中，这一命令也相当有用。比如我们要编写一个子程序，它返回有序表中的每一个元素都被第一个元素除的结果。

```
> nest := proc(x::list)
    local v;
    v := x[1];
    map( y -> y/v, x );
end:
> lst := [2, 4, 8, 16, 32]:
> nest(lst);
[1, 2, 4, 8, 16]
```

我们不知不觉中已经在子程序 `nest` 中间定义了另一个子程序：`y -> y/v`。这个子程序中有一个变量 `v`，Maple 根据有效域的范围，认为它就是外面的子程序 `nest` 中的同名变量 `v`。那这是一个全局变量呢，还是一个局部变量？显然，两者都不是。如果把上面例子中内部子程序的 `v` 申明成 `local` 或者 `global`，都无法达到我们的目的。

那么，Maple 究竟是怎样来判定一个变量的作用域的呢？首先，它自里向外，一层一层地寻找显式地用 `global`、`local` 申明的同名变量，或者是子程序的参数。如果找到了，它就将其绑定在外层的名变量之上，实际上就是将两个变量视为同一，就像上面例子中的情况。如果没有找到，就遵循下面的原则：如果变量位于赋值运算符“`:=`”的左边，就视其为局部变量；否则均认为它是全局变量。

5.8 记忆表

有时候，一个子程序会被多次调用，而且调用的参数也相同；那么，Maple 也会不厌其烦地一边有一边地计算同一个答案。正如前面所介绍过的，可以利用记忆表来改进这一点。每一个 Maple 子程序都可以有记忆表，记忆表的目的是为了提高计算效率，它把每一个计算过的结果存储在一个映射表中，所以在下一次用相同的参数调用的时候可以避免重新计算。Maple 的记忆表是哈希表，索引速度非常快。但由于多种不同的参数调用可能导致记忆表变得很大，所以记忆表适合于那些常常需要计算相同结果，而就算过程有相当复杂的子程序。

记忆表的用法有几种，我们在下面分别加以介绍：

5.8.1 remember 选项

在定义子程序时，我们可以加入 `remember` 选项，来建立该子程序的记忆表。从本章第 5 节中递归形式的菲波那契数列子程序中，我们早已体会到了加不加这个选项有什么差别。

为了对记忆表加以说明，我们在这里再建立一个简单版本的 Fibonacci 子程序：

```
[> Fibonacci := proc(n::nonnegint)
      option remember;
      if n<=1 then RETURN(n) fi;
      Fibonacci(n-1) + Fibonacci(n-2);
end:
```

在我们用它计算了 f_3 之后，它的记忆表中就有了 4 项。我们可以用 `op` 命令来检查一个子程序的记忆表：

```
[> Fibonacci(3);
2
[> op(4,eval(Fibonacci));
table([
  0 = 0
  1 = 1
  2 = 1
  3 = 2
])
```

记忆表是子程序的第 4 个元素——我们用 `op(4, ...)` 来获得；这里顺便题一句，一个子程序的前三个元素分别是它的参数、局部变量、和选项（比如 `option`）。

5.8.2 在记忆表中加入项

通常，调用具有记忆表的子程序并且计算得到结果，Maple 会自动地把结果加入到记忆表中去；但我们也可以手动地在记忆表中添加内容。您也许还记得，在这一章的开头，我们曾经试图用 `f(x) := expr` 的形式定义函数，但得到的却是函数的记忆表。这就是记忆表项的添加方法。我们可以利用记忆表，再次简化我们的 Fibonacci 子程序——甚至用不着分支结构：

```
[> Fibonacci := proc(n::nonnegint)
      option remember;
      Fibonacci(n-1) + Fibonacci(n-2);
end:
```

当然，在使用之前，我们还需要在它的记忆表中添加菲波那契数列的初始项：

```
[> Fibonacci(0) := 0:
[> Fibonacci(1) := 1:
```

在适当的时候，巧妙地利用记忆表把我们已知的结果“告诉”Maple，可以大大地提高解决问题的效率。

5.8.3 在记忆表中删除项

记忆表是映射表的一种，可以方便地在其中添加或者删除特定的项。和一般变量一样，删除一个表项只需将其名称用 `evaln` 赋给它本身就可以了。例如，因为输入错误，我们在 Fibonacci 子程序的记忆表中加入了一个错误的项：

```
[> Fibonacci(2) := 2:
```

由于映射表和数组一样，在赋值时不产生新的拷贝（参见第四章）。我们可以将 Fibonacci 的记忆表先取出来，再将相应的项加以删除：

```
[> T := op(4, eval(Fibonacci));
      T:=table([
          0 = 0
          1 = 1
          2 = 2
          ])
[> T[2] := evaln( T[2] );
                                     T2:=T2
```

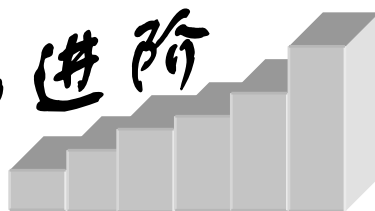
现在，Fibonacci 子程序的记忆表也同时被改变了：

```
[> op(4, eval(Fibonacci));
      table([
          0 = 0
          1 = 1
          ])
```

Maple 也可以自动地删除子程序的记忆表，如果我们在子程序定义时申明 `system` 选项，系统将会在回收无用内存时将记忆表删除，就如同大多数系统内部子程序一样。但对于类似于上面的 Fibonacci 的这一类依赖记忆表来作为递归结束标志的子程序，就不可以使用 `system` 选项。

对于记忆表的使用范围，还有需再加以说明，它只适用于对于确定的参数具有确定的结果的程序；而对于那些结果不确定的程序，比如用到环境变量、全局变量、或者时间函数等等的程序，就不能使用，否则将会导致错误的结果。

起步与进阶



第

Maple 绘图

六

章

这一章将围绕着 Maple 中的绘图功能进行初步的介绍。本章的介绍将以 Windows95 系统中的 Maple V Release 5 为例，在其他版本的 Maple 中，大部分的例子都可以实现。但是，和各种系统相应，这些绘图功能的输出是有所不同的。我们把介绍的重点放在各类函数的使用方法上，对于不同系统下的界面差异将不作介绍。

本章具体包括以下内容：

- 🕒 单变量函数曲线绘制
- 🕒 参数曲线的绘制
- 🕒 隐函数确定的曲线
- 🕒 根据数据绘制曲线
- 🕒 二维动画的绘制
- 🕒 双变量函数曲面绘制
- 🕒 参数曲面的绘制
- 🕒 隐函数确定的曲面
- 🕒 由数据生成曲面
- 🕒 三维动画的绘制

和别的语言不同，当我们利用 Maple 进行二维或者三维绘图时，Maple 可以自动地决定所需的点数、坐标轴的位置、标尺的数字、图形的颜色等等繁杂的设置，在默认状态下就可以绘制出令人满意的图形。当然，你也可以自己设定各种不同的绘图设置，比如更改绘图的坐标系（以画出极坐标、球坐标、或柱坐标下的图形），或者绘图的点数。

下面，我们就由浅入深地介绍 Maple 中的绘图方法。

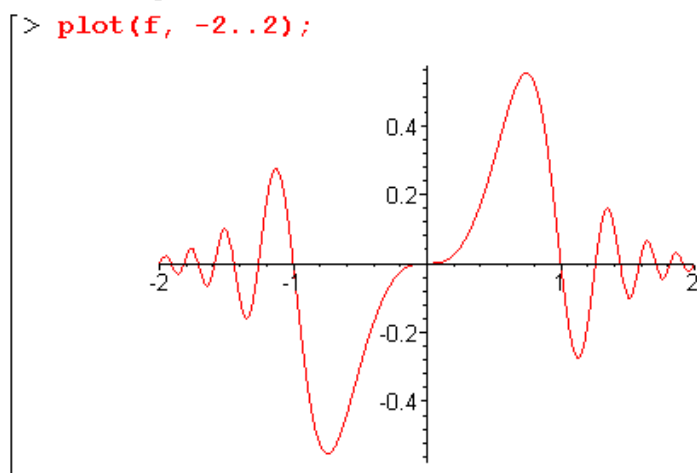
6.1 二维基本图形绘制

在 Maple 中，单变量函数曲线的绘制可以使用函数 `plot`。例如，我们需要绘制函数 $f(x) = e^{-x^2} \sin(\pi x^3)$ 在区间 $[-2, 2]$ 上的图形，我们可以这样来实现。首先用箭头操作符定义函数：

```
> f := x -> exp(-x^2) * sin(Pi*x^3);
```

$$f := x \rightarrow e^{(-x^2)} \sin(\pi x^3)$$

然后，调用 `plot` 函数。



键入命令后，所绘制的图形会立即出现在同一个可执行块中。Maple V Release 5 也支持把图形单独绘制在一个窗口中，如图 6.1 所示，只需要在菜单 `Option | Plot Display` 选择 `Window` 即可。

一般地，函数 `plot` 的调用格式为 `plot(f, a..b, options)`。其中， f 是需要绘制的函数， $a..b$ 是自变量的变化范围，*options* 是可选参数，用它可以控制图形的绘制，我们将在下一节中详细介绍。除了可以绘制函数的图形外，`plot` 也可以绘制表达式表示的函数图形，调用格式为 `plot(expr, x = a..b, options)`。其中，*expr* 是表达式（相信读者一定知道表达式和函数间的区别了）， x 是表达式中的自变量，因为表达式中没有自变量的信息（甚至可以是多变量的表达式），所以必须指定自变量，并用等式形式给出自变量的变化范围。

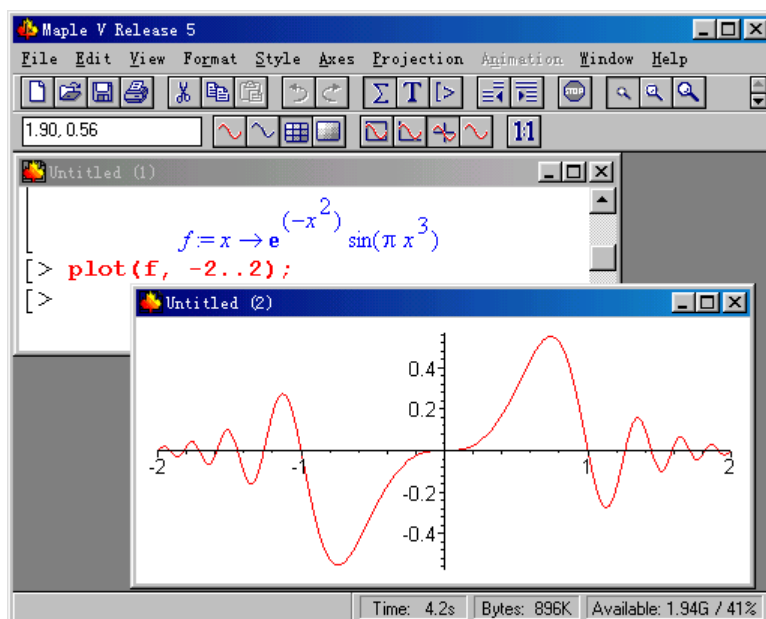
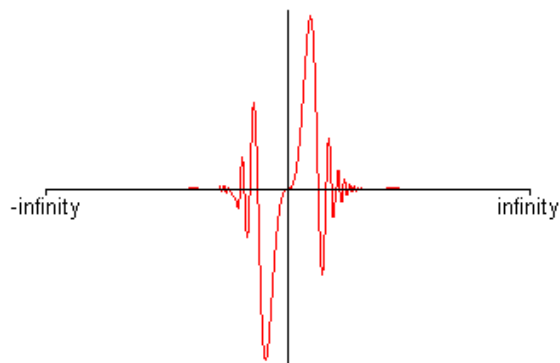


图 6.1 在窗口中绘图

这些基本的绘图功能，Maple 和其他的树脂绘图工具（比如 Matlab、Origin 等）并没有什么区别。但是，作为符号代数系统，Maple 具有其更为强大的功能，它甚至可以在无穷区间上绘图。

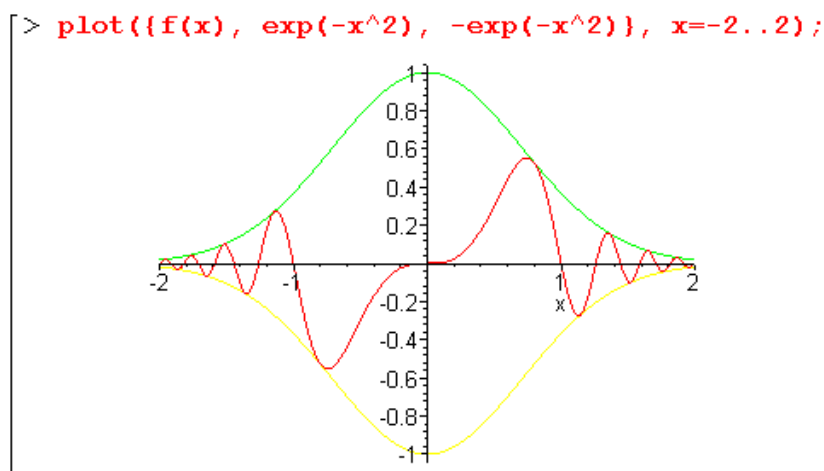
例如，我们希望在实数轴上绘制上面的函数 f 的图形：

```
> plot(f, -infinity..infinity);
```



显然，现在的 x 轴并不具有均匀的标尺（否则我们将什么也看不到）。为了显示所有实数轴上的图形，Maple 作了一个映射，它将整个实数轴映射到 $(-1, 1)$ 区间上。这个映射可以近似地表示为 $x \mapsto \frac{2}{\pi} \arctan(\frac{x}{2\pi})$ 。我们可以用 Maple 在整个实数域上绘制这个映射函数，得到的结果接近于一条直线，可见，Maple 绘图中使用的从 $(-\infty, \infty)$ 到 $(-1, 1)$ 的映射和这个表达式相近。

在 Maple 中，可以同时绘制几个图形，也就是把几个函数图形绘制在同一张图上。对于每一个不同的函数或表达式，Maple 会自动地选择一种不同的颜色来绘制图形，以加以区别。绘制多个函数的 plot 的用法和前面的相近，只是第一个参数是一个函数或表达式的集合（用一对花括弧 “{ }” 括起来）。



Maple 的绘图不仅可以在屏幕上进行，还可以输出到不同的设备。例如，可以输出到打印机文件——PostScript 格式的描述文件。它可以用来直接打印，也可以插入到其他支持 PostScript 格式的程序中去，例如 TEX。

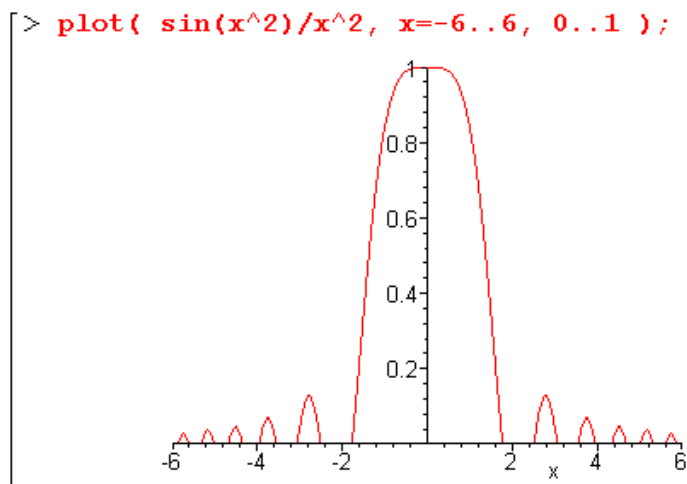
```
> interface( plotdevice = postscript,
              plotoutput = "plotfile" );
```

用以上的形式调用 interface 函数后，图形便会以 PostScript 格式输出到文件名为 plotfile 的文件中去了。如果需要将图形输出重新定向为默认的格式，可以用 plotfile = default 或者 plotfile = inline 为参数调用 interface。

```
> interface( plotdevice = inline );
```

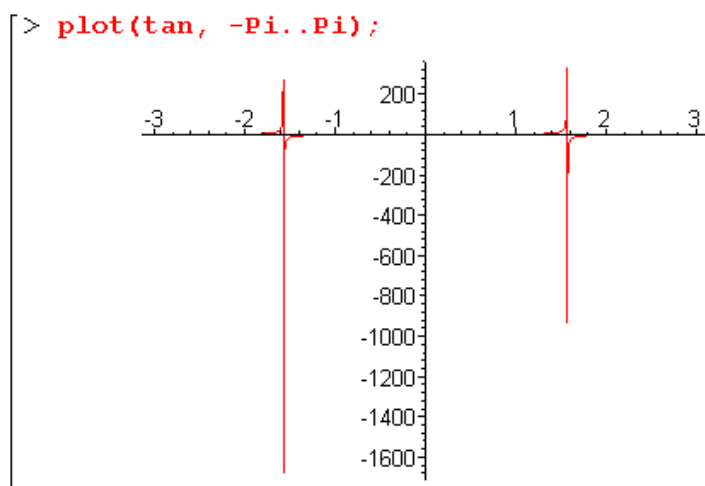
6.2 plot 函数的可选参数

在默认情况下，Maple 的 plot 函数会自动地选择各种绘图参数。例如，它可以自动选择绘图的区间，采样点数，标尺的刻度等等。Maple 选择这些参数的依据是，使最常用的图形绘制尽可能简单易行，而且具有可以认可的质量。当然，为了绘制某些特殊要求的图形，我们也可以人为地确定这些绘图参数。

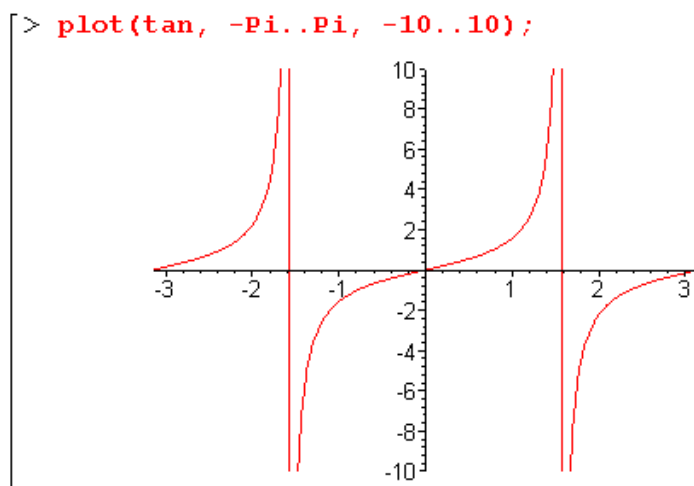


我们已经知道，对于单变量函数曲线的绘制，`plot` 的第二个参数是横坐标（自变量）的取值范围；如果要限制纵坐标的取值范围，就需要在调用 `plot` 时将它作为第三个参数。在上面这个例子中，我们在自变量取值区间 $(-6, 6)$ 上绘制表达式 $\frac{\sin^2 x}{x^2}$ 的图像，而且设定了纵坐标的范围 $[0, 1]$ ，这样，图像在 $[0, 1]$ 以外的部分就被切除了。

通常情况下这一参数好像并没有必要，但有的时候，函数在我们关心的区域内有奇点，或者有远大于其他地方函数值的极值点。这时在使用默认的参数绘图，就会得到一些具有“尖峰”的函数图形，而其他地方的函数特征都被掩盖掉了。比如我们要绘制正切函数的图像，在默认情况下得到的曲线是这样的：

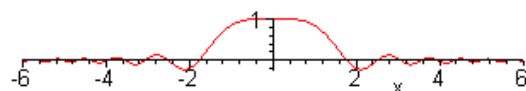


为了得到可以恰当地反映函数特征的曲线，这时就必须限制纵坐标的显示区间。比如对于正切函数的图像，我们在 $[-10, 10]$ 的纵坐标范围中对其进行绘制，得到的曲线就比较令人满意：



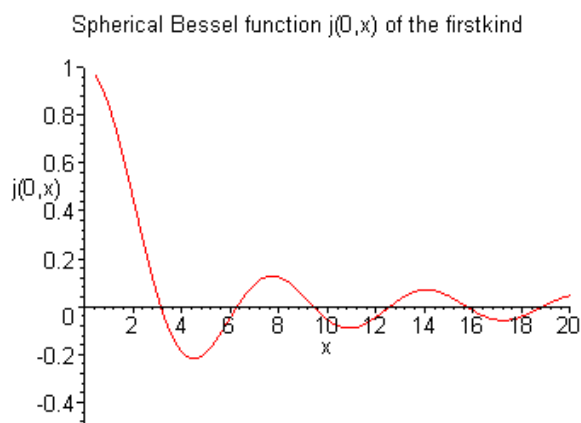
关于绘图的比例问题，默认情况下，Maple 自动调整横纵坐标的比例，使横纵坐标上的最值都可以被正常显示（如果函数在给定区间里有最值的话）。这样，横纵坐标的显示比例往往是不相同的，如果要使横纵坐标以相同的比例显示，可以加入控制参数 `scaling = constrained`。

```
> plot( sin(x^2)/x^2, x=-6..6, scaling =
constrained );
```



如果所要绘制的函数具有单变量的子程序形式，那么第二个参数可以略写为自变量的取值范围，而无须指定自变量名称，如前例中正切函数的曲线绘制所示。这样，横纵坐标上都只有刻度，而没有名称。如果在限制取值范围的同时，还指定了自变量和函数名称，那么坐标轴上也将分别标上它们的名称。而且，在 `plot` 中，我们还可以指定图像的标题，只需要在 `plot` 中加入可选参数 `title = 标题`，当然标题当中一般都含有特殊字符，例如空格、括弧等等，所以需要反向撇号 “```” 将它括起来（也可以用一对双引号括起来，作为字符串）。

```
> j := (n,x) ->
sqrt(Pi/(2*x)) * BesselJ(n+1/2,x):
> plot( j(0, x), x=0..20, `j(0,x)`=-0.5..1,
xtickmarks=8, ytickmarks=4, title =
`Spherical Bessel function j(0,x) of the
firstkind`);
```



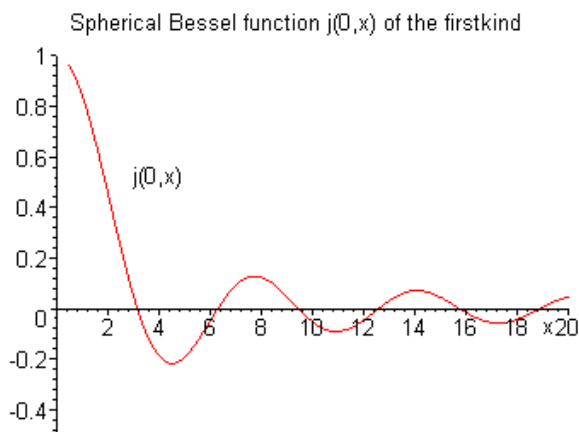
在上面的例子中，我们还使用了两个附加参数，`xtickmarks=8` 和 `ytickmarks=4`，它们分别指定了横、纵坐标上刻度值的个数。

上面这种绘图方法，一气呵成，把所有要加的东西都一次加上，这样当然方便；但有时我们需要逐步的在图像中加入各种元素，然后一同显示。这时，我们可以把图形对象赋值给一个变量（这个变量的类型是函数型的）。把我们需要的图形元素依次赋给不同的变量，然后调用 `plots` 工具包中的函数 `display` 就可以将它们显示在一张图上了。我们还可以用 `textplot` 在图形的任意位置进行标注。

```

> plot1 := plot( x->j(0, x), 0..20, -0.5..1,
  title = "Spherical Bessel function j(0,x) of
  the firstkind"):
> plot2 := plots[textplot]( { [3, 0.5,
  "j(0,x)"], [19, -0.1, "x"]}, align={ABOVE,
  RIGHT}):
> plots[display]( {plot1, plot2} );

```



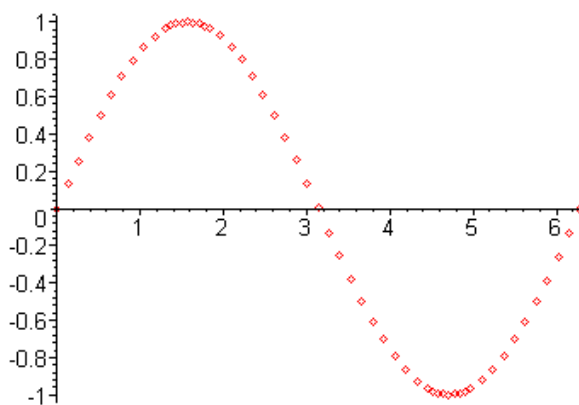
`plots[textplot]`函数的第一个参数是一个有序表的集合，其中每个有序表表示一个标注。有序表由三个元素组成，前两个表示标注点的坐标，第三个是需要标注的数值、变量或者字符串（不管什么类型，输出时都会转换成为字符串输出）。`plots[textplot]`的第二个参数 `align = t` 是可选参数，它确定标注的字符串相对于给定坐标的对齐方式，其中 `t` 可以是 `BELOW`, `RIGHT`, `ABOVE`, `LEFT` 中的一个或几个的集合，在默认情况下，`plots[textplot]`将字符串的中心位置和给定坐标对齐。`plots[display]`函数可以用来显示图形对象，它可以显示一个图形或者几个图形的集合。

在绘图时，Maple 首先计算曲线上的一些点的坐标（点的个数可以用参数“`numpoints = 点数`”人为地设定），然后，在默认情况下，Maple 用直线段把这些离散的点连成一条曲线。但有时需要绘制离散的点构成的图形，可以设置参数 `style = point`（默认情况下为 `style = line`）。对于多边形，还可以将绘图方式 `style` 设成为 `patch`，这时绘制的是经过填充的多边形。在用 `point` 方式绘制曲线时，Maple 会用一些以函数点为圆心的小圆圈来绘图，例如：

```

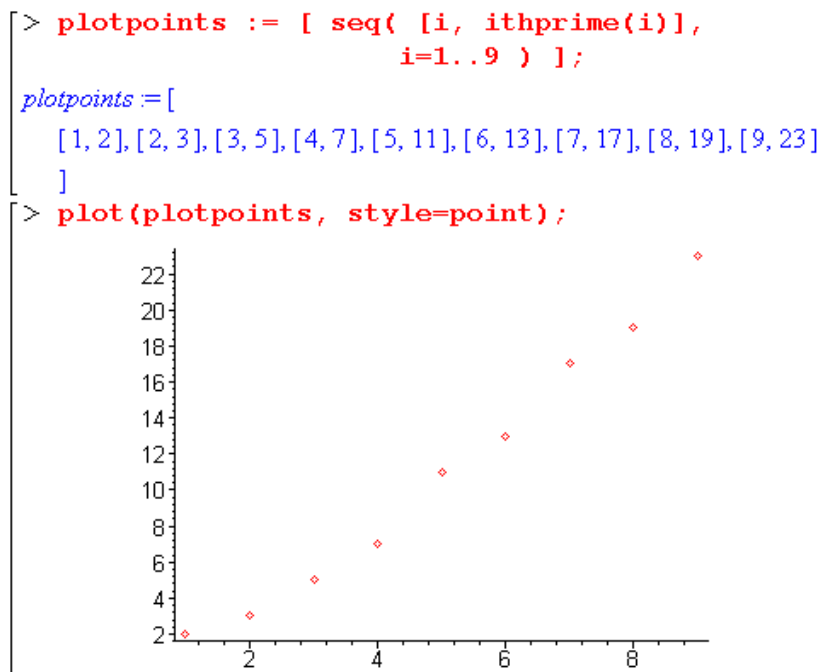
> plot( sin, 0..2*Pi, style = point );

```



除了绘制函数或者表达式的曲线外，`plot` 还可以用于对给定的离散数据绘图，这时同样

也可以选择 `point` 的绘图方式。例如，我们要在图上绘制前 9 个素数，可以把这 9 个素数及其序号组成的有序表作为第一个参数传给 `plot`：



用已知数据绘图往往用在图形不能够显式地写成解析表达式的情况，比如在下面这个例子中，我们需要绘制以 1.0 为初值对余弦函数进行迭代的图形。首先，需要生成一个点的有序表：

[[1.0, cos(1.0)], [cos(1.0), cos(1.0)], [cos(1.0), cos(cos(1.0))], [cos(cos(1.0)), cos(cos(1.0))], [cos(cos(1.0)), cos(cos(cos(1.0)))], ...]。

这样的有序表，最简便的生成方法是利用函数符合操作符@：

```
> plotpoints := [ seq( [
  (cos@@(trunc((2*i+1)/4)))(1.0),
  (cos@@(trunc((2*i+2)/4)))(1.0) ],
  i=1..30 ) ]:
```

这里，`trunc` 函数是向下取整函数，它是用来获得 `cos` 的复合次数的；而 `cos@@(num)` 则表示 `cos` 函数复合运算 `num` 次。

得到了这些数据对之后，就可以绘图形了，为了使整个迭代过程更加清楚，我们同时还要绘制两个函数的图形—— $y = x$ 和 $y = \cos(x)$ 。我们将它们分别保存在不同的变量中。

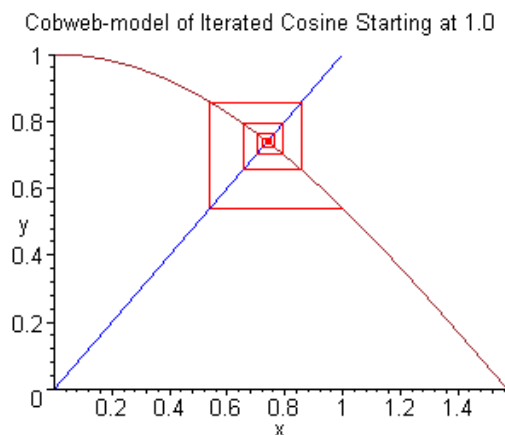
```
> plotpoints := [ seq( [
  (cos@@(trunc((2*i+1)/4)))(1.0),
  (cos@@(trunc((2*i+2)/4)))(1.0) ],
  i=1..30 ) ]:
> plot1 := plot(plotpoints, x=0..Pi/2, y=0..1,
  style=line, colour=red):
> plot2 := plot( x, x=0..Pi/2, y=0..1,
  colour=blue ):
> plot3 := plot( cos(x), x=0..Pi/2, y=0..1,
  colour=brown ):
```

由于图形是分别绘制的，所以 Maple 不对它们取不同的颜色；为了加以区别，我们用

colour 参数设置绘图的颜色。Maple 中对于常用的颜色都有相应的名称,请参考再线帮助 plot, colour。

最后,我们用 plots 工具包中的函数 display 绘制这些图形,并且加上标题。

```
> plots[display]( { plot1, plot2, plot3 },
  title = "Cobweb-model of Iterated Cosine\
  Starting at 1.0" );
```

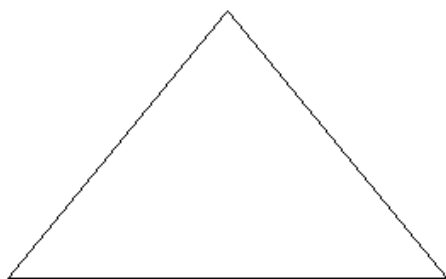


6.3 二维图形对象的结构

为了能清楚地知道 Maple 绘图的可靠性,我们必须了解它的绘图机制。Maple 中的绘图是分两步走的:其一,计算出所有的绘图点,再将它存入到图形对象——PLOT 中;其二,在屏幕或者其他输出设备中进行图形的绘制。在这一小节中,我们把介绍的重点放在第一步中。

Maple 的图形对象实际上是一个函数调用, plot 函数实际上就是生成了一个这样的函数——PLOT。PLOT 函数有多个参数,分别确定了绘图所需的点的坐标,绘图的方式,坐标轴的模式等等。我们也可以直接调用 PLOT 函数生成图形对象,例如,可以画一个一致三个顶点的三角形:

```
> PLOT( CURVES([ [1,1], [2,2], [3,1], [1,1] ]),
  AXESSTYLE(NONE) );
```



我们提供了该函数的部分参数——CURVES(...)和 AXESSTYLE(...),分别表示曲线的点,坐标轴的模式。其余的参数,例如 COLOUR (绘图的颜色)、VIEW (绘图范围)、AXESTICKS (标尺刻度)等,Maple 都自动选择了默认的情况。

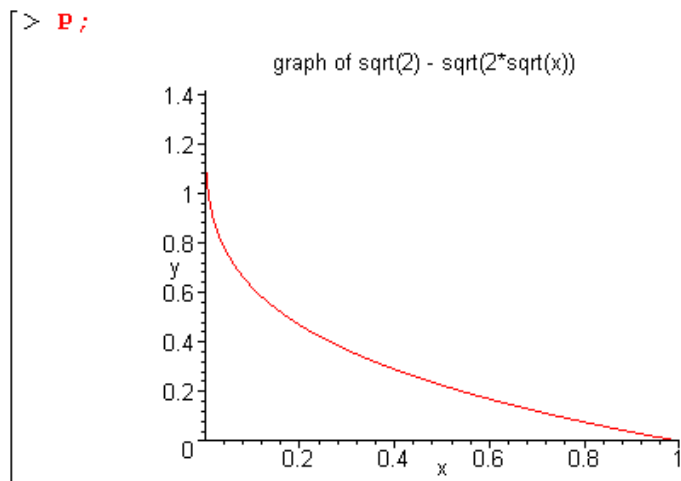
那用 plot 函数所生成的图形对象是不是和它有所不同呢？作为例子，我们来看一看函数

$f(x) = \sqrt{2} - \sqrt{2\sqrt{x}}$ 的图像在 Maple 中的结构。

```
> f := x -> sqrt(2) - sqrt(2*sqrt(x)):
> P := plot( f(x), x = 0..1, y = 0..sqrt(2),
            title = "graph of sqrt(2) -
            sqrt(2*sqrt(x))" );
P:=PLOT(CURVES([[0, 1.414213562373095],
               [.000681161067708333, 1.185744472197018],
               [.9782721018750000, .007745369848915162], [1., 0]],
        COLOUR(RGB, 1.0, 0, 0)), TITLE("graph of sqrt(2) - sqrt(2*sqrt(x))"),
        AXESLABELS("x", "y"), VIEW(0..1, 0..1.414213562))
```

(由于点的数据过多，在这里予以省略)

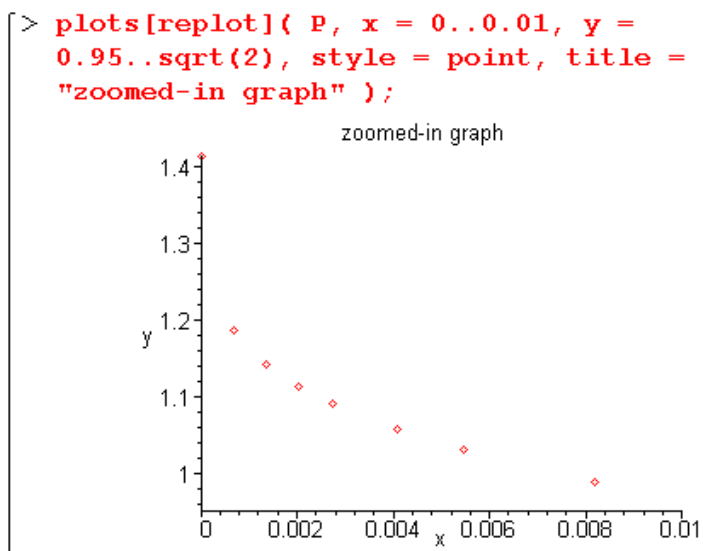
不难发现，除了特殊的绘图要求需要有较多参数外，这个数据结构和前面的是相同的。在 Maple 中只要再次输入它，就可以得到它所表示的图形：



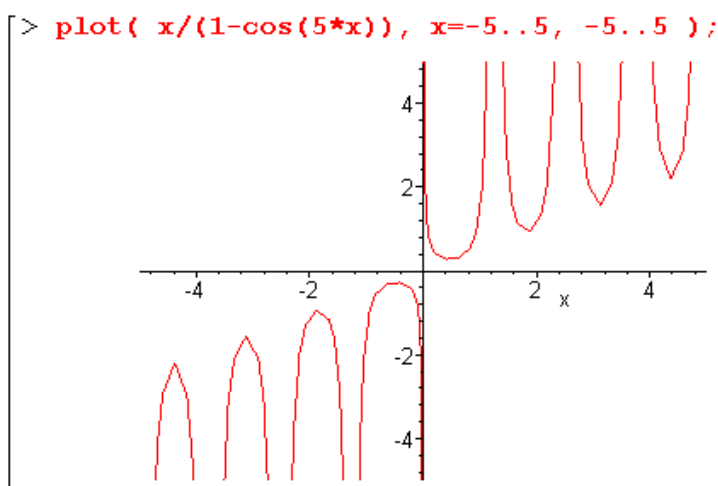
调用绘图函数 plot 时，Maple 首先对表达式或函数求值，然后在采样点上求它的数值结果。为了提高运算速度，Maple 在这里使用的是硬件浮点（详见第一章）。

为了提高图形的质量，Maple 在选择采样点的时候采用的是逐步求精的方法。首先，计算绘图区间中几乎等距的 49 个采样点上的函数值；然后，Maple 会检验这些点之间的连续程度，检验时用的算法很好理解，想象这些点依次用直线段相连，相邻的两条直线段之间会形成一定的夹角，如果夹角过大，也就是在这里的连续性不好，Maple 会适当增加附近的采样点数，以提高图形的质量。但作为一个成熟的计算机软件，Maple 决不会无限制地把采样点数增加下去。图形的最高分辨率由选项 resolution 确定，resolution 的默认值是 200。也就是说，默认情况下，采样点数总是介于 49~200 之间。

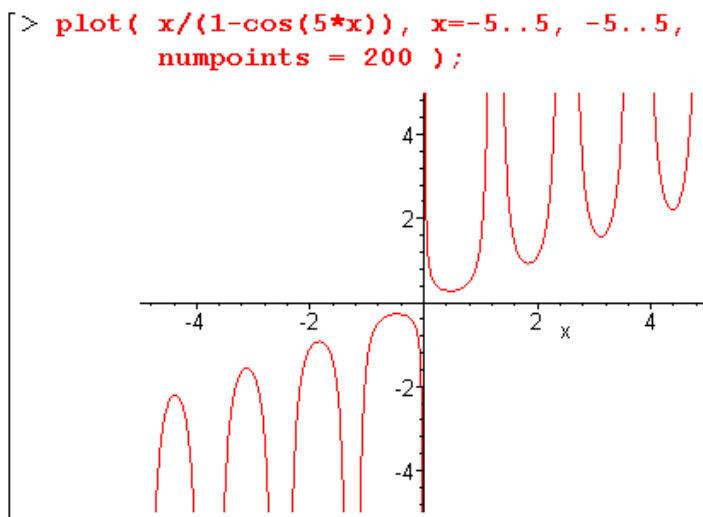
实际上，在上一个例子的图形中，Maple 已经在局部增加了采样点数。这在完整的图形上几乎看不出来。但是可以利用 plots 工具包中的函数 replot 将图形在局部放大，就可以清楚地看到这一点——图形的许多个采样点之间间距不同（如下图所示），这就是利用了上面的逐步求精算法的结果。



但有时, 对于一些局部连续性不是太好的函数, Maple 自动取的采样点仍不能使我们满意。例如, 表达式 $\frac{x}{1-\cos 5x}$ 在区间 $(-5, 5)$ 上的图像:



这时, 人为地增加采样点数, 可以提高图形的质量。采样点数可以在 plot 函数中用参数 “numpoints = 预定的采样点数” 给出。



前面我们所作的绘图参数的改变，都是对于单个图形对象而言的，如果需要改变默认的绘图参数，可以调用函数 `setoptions`。

6.4 特殊二维图形的绘制

6.4.1 参数曲线的绘制

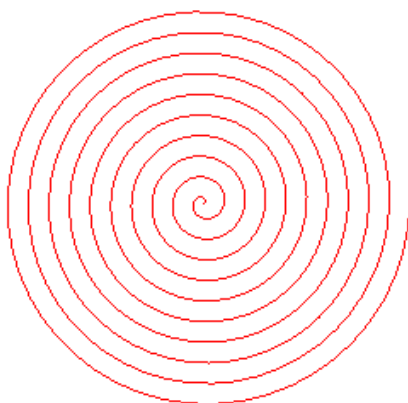
在 Maple 中，对于笛卡尔系下的参数曲线

$$\begin{cases} x = f(t) \\ y = g(t) \end{cases} \quad a < t < b,$$

调用 `plot` 函数的标准格式是 `plot([f(t), g(t), t = a..b], options)`，其中 `a..b` 是绘图的范围，`options` 是可选参数，它的可选参数和绘制一般平面图形式相同的。

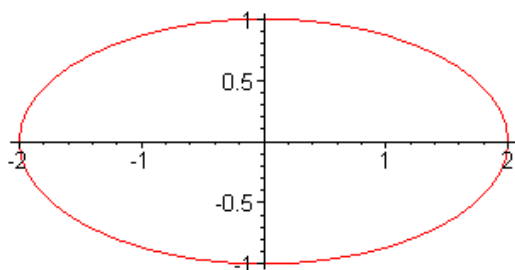
作为例子，我们来画一条平面螺线：

```
> plot( [ t*cos(2*Pi*t), t*sin(2*Pi*t), t=0..10 ],
        numpoints=500, scaling=constrained, axes=none );
```



当所需绘制的图形是用函数而非表达式给定时，就不能再给定它的自变量了，例如：

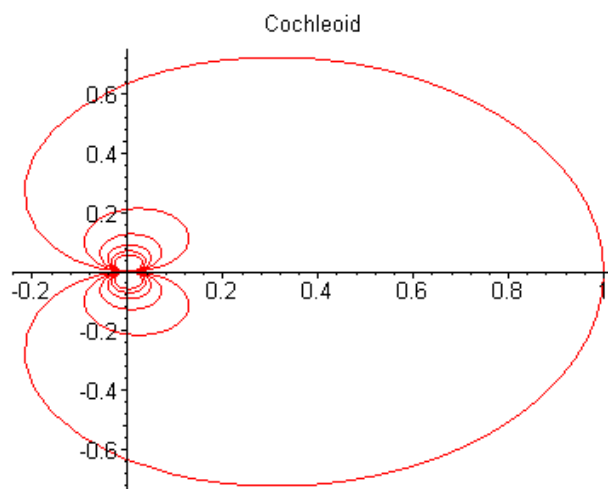
```
> plot( [ 2*cos, sin, 0..2*Pi ], scaling=constrained );
```



6.4.2 极坐标下的绘图

Maple 的极坐标绘图和二维参数曲线的绘制极其相似，函数调用格式相同，所不同的是需要增加一个控制参数 `coords = polar`，极坐标绘图的一般格式为：**plot**([$r(t)$, $\phi(t)$, $t = t_0..t_1$], `coords = polar`, *options*)。其中 $r(t)$ 和 $\phi(t)$ 分别是向径和辐角的函数。例如，我们用它在极坐标下绘制蜗线 (cochleoid)：

```
> plot( [ sin(t)/t, t, t=-6*Pi..6*Pi ], coords=polar,
        numpoints=250, title = "Cochleoid" );
```



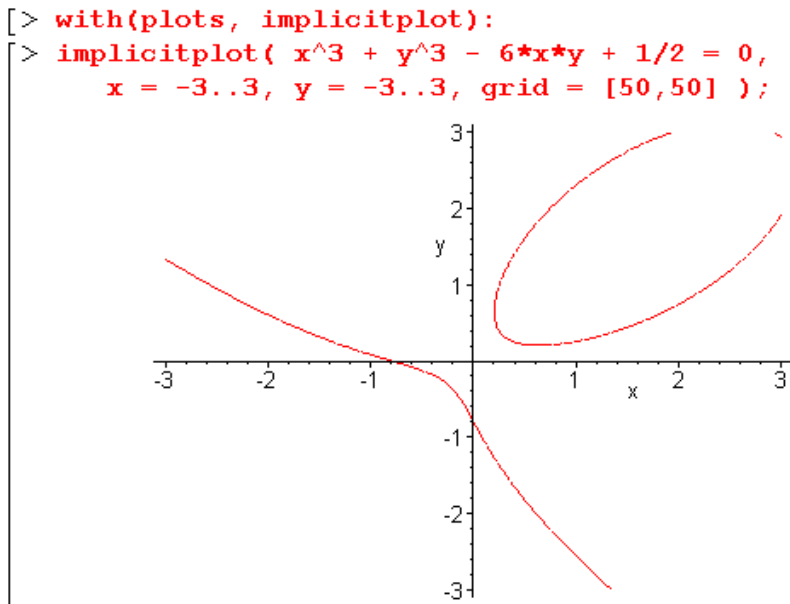
绘制极坐标图形，不仅可以通过在 `plot` 中加参数的方法实现，还可以直接调用 `plots` 工具包中的函数 `polarplot`，它的调用格式和输出结果都是和上面的调用相同的。作为例子，我们来看一个有趣的图形，它看起来有点像 Maple 的商标图案——一片枫叶。

```
> R := t -> 100 / ( 100 + (t-Pi/2)^8 )
        * ( 2 - sin(7*t) - cos(30*t)/2 );
> plots[polarplot]( [ R, t->t, -Pi/2..3/2*Pi ],
        axes = none, numpoints = 800 );
```



6.4.3 平面代数曲线的绘制

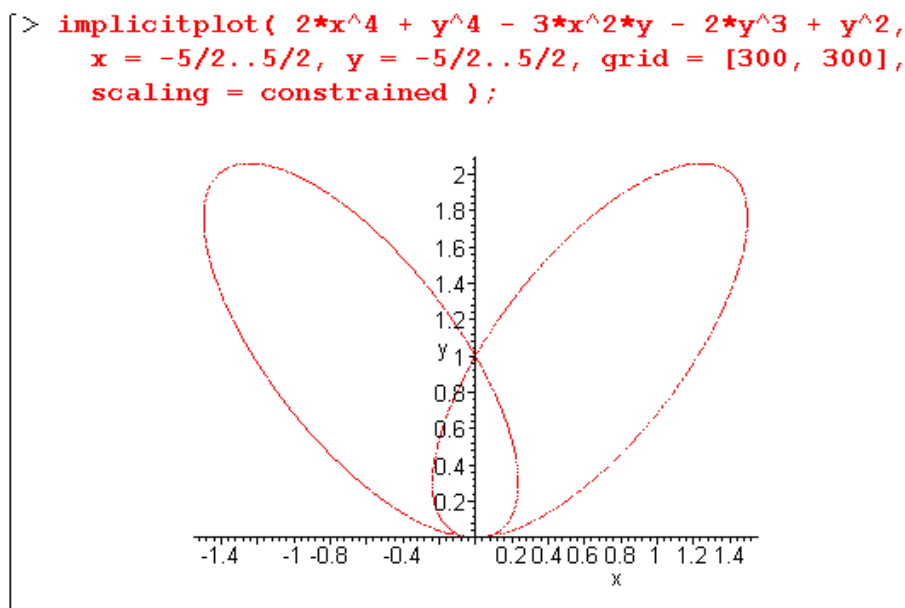
在 Maple 的 plots 工具包中, 还有一个函数 `implicitplot`, 利用它可以绘制用一个二元方程隐式定义的二维平面曲线。例如:



在 Maple 内部, 这样的隐函数 $f(x, y) = 0$ 的图形被视为一个空间曲面 (由方程所定义的二维函数 $f(x, y)$) 和 z 平面的交线, 函数 `implicitplot` 的算法就是基于这一点的。由于求交算法会用到二维曲面的梯度, 所以这一算法在函数的奇点附近, 也就是 $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = 0$ 的点附近, 会遇到无法逾越的障碍。最典型的一个例子是由方程:

$$2x^4 + y^4 - 3x^2y - 2y^3 + y^2 = 0$$

所定义的曲线, 它有两个奇点, $(0, 0)$ 和 $(0, 1)$ 。即使试用很高的分辨率 (用 `grid` 参数设置), `implicitplot` 仍然不能正确地画出它的图形, 而且计算所需的时间和资源是令人无法忍受的。



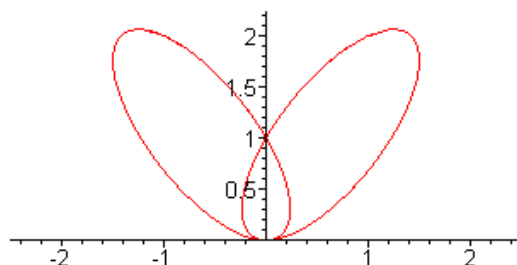
这个算例是在 Pentium 200 上进行的，它运行了将近两分钟，还占据了 20M 的内存（可以在窗口的状态条上看出）。然而，得到的图形还是不理想，尤其在原点附近。

所以，我们不能过分依赖于 Maple 的 `implicitplot`；如果可以将表达式写成显式的或者参数形式的，就不要直接用隐式表达式画图。例如，上面这个例子就可以在极坐标中写成为显式的表达式。

```
[> subs( x=r*cos(phi), y=r*sin(phi),
          2*x^4 + y^4 - 3*x^2*y - 2*y^3 + y^2 );
> factor(%);
r^2 (2 r^2 cos(phi)^4 + r^2 sin(phi)^4 - 3 r cos(phi)^2 sin(phi) - 2 r sin(phi)^3 + sin(phi)^2)
> solve(op(2, %), r);
1 (3 - sin(phi)^2 + sqrt(1 + 10 sin(phi)^2 - 11 sin(phi)^4)) sin(phi)
2 -4 sin(phi)^2 + 2 + 3 sin(phi)^4,
1 (3 - sin(phi)^2 - sqrt(1 + 10 sin(phi)^2 - 11 sin(phi)^4)) sin(phi)
2 -4 sin(phi)^2 + 2 + 3 sin(phi)^4
```

有了显式表达式，就可以用 `polarplot` 在极坐标中绘制图形了：

```
[> sols := map( unapply, {%, phi } );
> plots[polarplot]( { [sols[1], t->t, 0..2*Pi],
                      [sols[2], t->t, 0..2*Pi] }, view=[-5/2..5/2,
                      0..9/4], scaling=constrained, color=red );
```



相比之下，这个图形比前面的要好得多，而且绘图速度也有极大的提高。

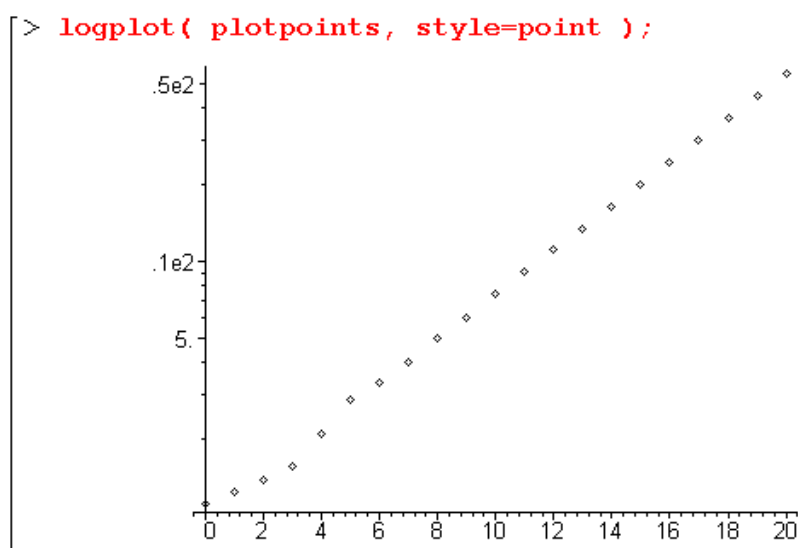
6.4.4 对数坐标下的绘图

在 Maple 的 `plots` 工具包中，还有对数坐标下的绘图函数 `logplot` 和双对数坐标下的绘图函数 `loglogplot`。它们的用法都和 `plot` 类似，也都可以用于绘制表达式，函数和离散数据。在这里，就不一一详细介绍了，以一个简单的例子来说明它们的使用。

对数坐标绘图往往用在数据处理上，因为在对数坐标上可以看出函数的指数关系。作为例子，我们用统计工具包中的随机函数 `random` 生成一组具有正态扰动的数据。

```
[> with( plots );
> rnd := stats[random, normald];
> plotpoints := [ seq( [i, exp(0.2*i) + 0.1*rnd() ],
                      i = 0..20 ) ];
```

然后，用 `logplot` 来绘制图形：



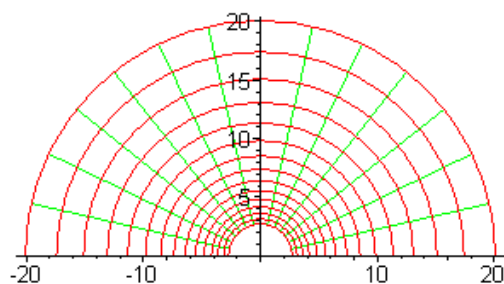
双对数坐标绘图函数 `loglogplot` 的使用也与此类似，这里不再赘述。

6.4.5 共形映射的图形绘制

Maple 的绘图工具包 `plots` 中还有一个共形映射（保角变换，**conformal mapping**）的绘图函数 `conformal`。它可以绘制复平面上一个矩形区域经过给定变换后的图形，还可以在中间绘制网格。

作为例子，我们来看一个常用的变换 e^z ，它把复平面上一个矩形区域映射成一个半环形的区域。可以用 `conformal` 获得变换后的网格图形：

```
> with( plots, conformal );
> conformal( exp(z), z=1..3+Pi*I, grid=[15, 15],
    scaling = constrained );
```

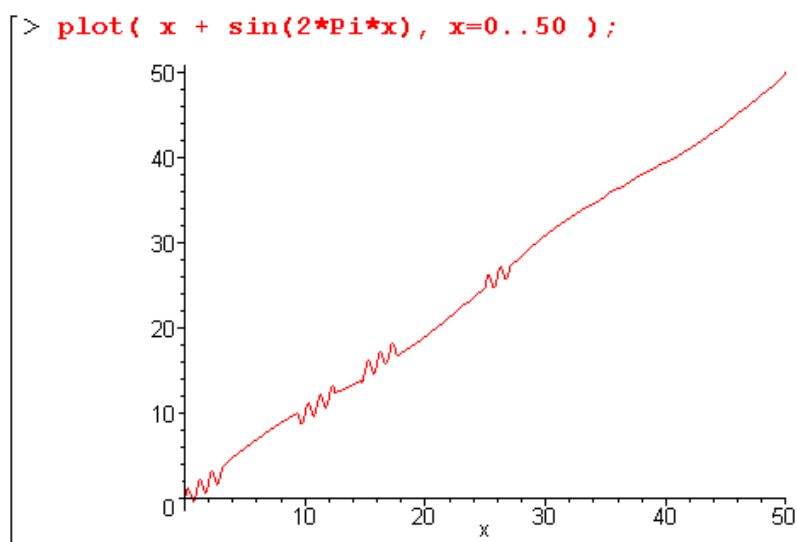


6.5 二维绘图的注意事项

6.5.1 图形走样

前面曾经说过，Maple 在绘图采用的是逐步求精的算法，逐渐增加采样点数。但它仅在采样点明显不足时——也就是图形不连续时才增加采样点数。这样，有时难免会损失绘图的局部精度。因为根据采样定理，采样率必须高于最高频率的两倍，否则会发生频率混淆。而如果 Maple 第一次采样就低于最高频率，而且因为频率混淆是图形呈现出虚假的“连续”，这时走样就难于避免了。

例如，我们用默认采样点数绘制 $x + \sin(2\pi x)$ 在区间(0, 50)上的图形，就会损失大部分的局部细节：



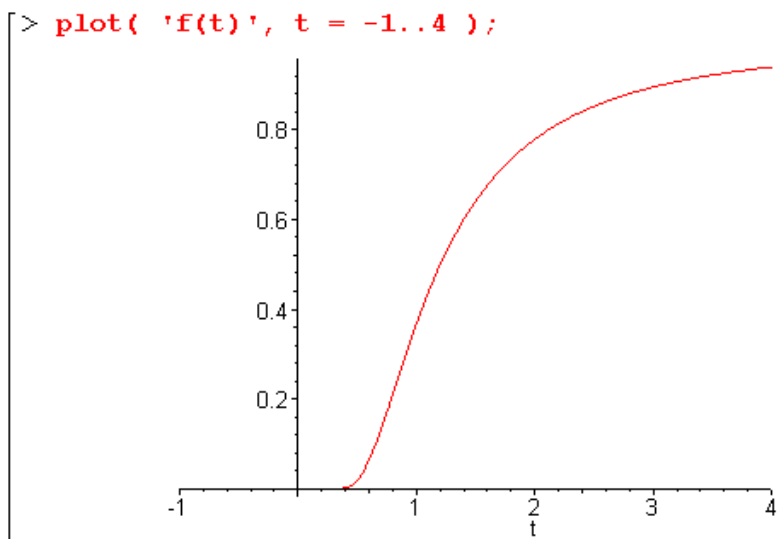
当然，这里我们知道图形的本来面目，所以发生了走样可以察觉；但有时也并不清楚所绘制图形的特征，这时就需要格外小心，必要时可以适当增加采样点数（用 `numpoints` 作为附加参数）。

6.5.2 常见的错误

对于用 `plot(f(x), x = x_min .. x_max)` 形式调用的函数 f 的图形绘制，Maple 首先把 $f(x)$ 化成为 x 的表达式，然后才对其进行数值计算。在一般情况下，这样的调用是不会有问题的，但如果 f 是一个分段函数，或者数值函数，这样的调用就会发生问题。例如下面这个分段函数：

```
[> f := t -> if t>0 then exp(-1/t^2) else 0 fi:
> plot( f(t), t = -1..4 );
Error, (in f) cannot evaluate boolean
```

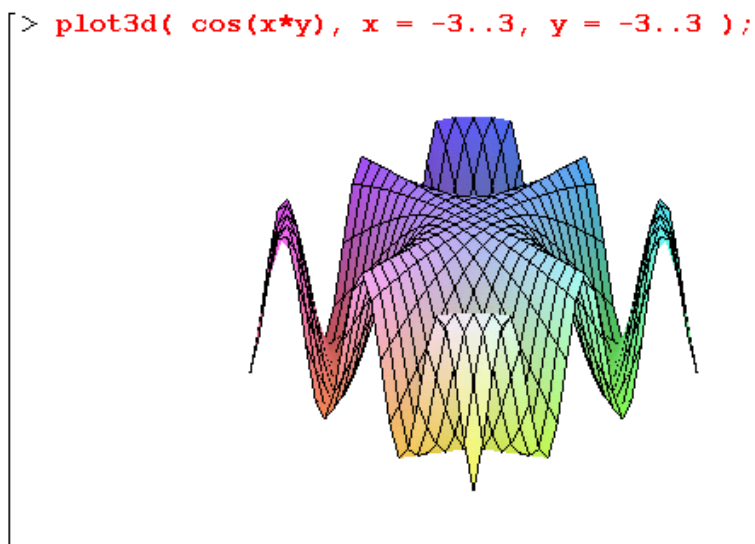
因为函数中有分之结构存在，Maple 不能把它化成为符号表达式，所以无法继续进行下去。解决问题的方法之一是在它外面加上单引号——延迟它的求值：



不过，对于单变量函数，最方便的方法还是用函数名的直接调用：`plot(f, xmin .. xmax)`。

6.6 基本三维图形的绘制

在 Maple 中，绘制由二元函数确定的三维空间中的曲面，和绘制一元函数的图形同样方便，只需要调用 `plot3d`，并且确定变量的范围就可以了。例如，绘制曲面 $\cos(x, y)$ 在 $-3 < x < 3$ ， $-3 < y < 3$ 中的部分，可以这样调用 `plot3d`：



上面的命令是以下的标准调用格式的一个例子。一般地，对于二元函数或者表达式 $f(x, y)$ ，绘制它所确定的曲面，`plot3d` 的调用格式如下：

`plot3d(f(x, y), x = a .. b, y = c .. d, options);`

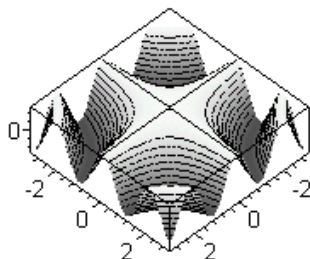
其中， $a .. b$ 和 $c .. d$ 分别是自变量 x 和 y 的绘图取值范围，而 *options* 是可选参数，它指定绘图选项，对它的详细介绍将在下一节中进行。

和 `plot` 一样，`plot3d` 也有函数的调用形式。对于函数 f 确定的曲面，标准调用形式为：

plot3d(*f, a .. b, c .. d, options*);

下面，我们再用函数形式的调用绘制前面的图形，而且在调用时提供了一些绘图参数：

```
> plot3d( f, -3..3, -3..3, grid=[49,49], axes=boxed,
          scaling = constrained, style = patchcontour,
          shading = zgrayscale );
```



6.7 三维绘图的选项

如同二维绘图一样，三维绘图也有许多的选项，使用户能够自定义输出图形的形式。所有这些选项，都可以通过在 `plot3d` 中附加绘图参数实现。在 `Maple V Release 5 for Windows` 中，也可以用鼠标选中绘制好的图形，然后在窗口的菜单中相应的选项中进行改动；在图形上单击鼠标右键弹出的菜单中也可作相同的改动。

在这一节中，我们将对于常用的绘图选项分别加以介绍。

6.7.1 style 选项

`style` 选项决定着图形的样式，可以在函数调用中用 “`style = 样式名称`” 指定，也可以在菜单 `Style` 中选择相应的样式。对于三维绘图，默认的样式是 `PATCH`，也就是经过填充后的曲面图形。对于三维曲面，它表现为有一些四边形的小面片组成的近似曲面。对于其他的绘图样式，我们列表加以说明：

表 6.1 三维图形的样式

样式名称	对应的菜单项	对应的样式
PATCH	Patch	填充后的面片
PATCHNOGRID	Patch w/o grid	无线框的填充面图
PATCHCONTOUR	Patch and contour	带等高线的填充面图
POINT	Point	离散点图
WIREFRAME 或 LINE	Wireframe	线框图
HIDDEN	Hidden line	消隐后的线框图
CONTOUR	Contour	等高线图

在菜单 Style 下,还可以设置点、线的样式,线宽,和网格形状,分别对应着菜单项 Symbol, Line Style, Line Width, Grid Style。其中,点的样式和线的样式分别在绘制点图和线图时才会起作用。

6.7.2 着色选项

在 Maple 的曲面图形中,着色方式可以用“shading = 着色方式”在调用 plot3d 时加入;对于已经绘制好的图形,也可以在菜单 Color 中选择。着色方式的含义,及对应的菜单项请见下表:

表 6.2 三维图形的着色方式

着色方式	对应的菜单项	方式的具体含义
XYZ	XYZ	对 $X + Y + Z$ 相等的点采用相同的色彩
XY	XY	对 $X + Y$ 相等的点采用相同的色彩
Z	Z	根据高度 (Z) 不同着色 (红蓝色)
ZGREYSCALE	Z (Grayscale)	根据高度 (Z) 用灰度着色
ZHUE	Z (Hue)	根据高度 (Z) 用全彩色着色
NONE	No Coloring	不着色

注: shading = NONE 和 style = PATCH 联合使用可以生成单色的线框图。

6.7.3 坐标轴选项

使用 axes 选项或者菜单 Axes 可以确定绘图时坐标轴的绘制方式。Maple 中有四种坐标轴绘制方式: BOXED, NORMAL, FRAME, NONE。对于三维图形,默认的选项是 NONE,也就是不绘制坐标轴。BOXED 是绘制框式的坐标轴,将图形绘制在一个立方体框中;NORMAL 是通常我们在数学中使用的坐标轴,它绘制于图形的中部,接近原点的地方,但对于三维图形,这样的坐标轴会影响图形的显示。一般而言,FRAME 是比较好的选择,它只在图形边缘绘制坐标轴,完全不影响图形本身。

6.7.4 空间朝向和投影

在 Maple 中,三维图形的空间朝向是用视线和空间笛卡尔坐标系的两个夹角 θ 和 ϕ 确定的,如图 6.2 所示。可以在调用 plot3d 时用选项 orientation = $[\theta, \phi]$ 设置。

角 θ 控制着图形在水平方向的转动,而角 ϕ 控制着图形的俯仰角度, ϕ 越大,视线相对于图形就越低。 $\phi = 90$ (度) 时,就意味着从水平方向观察图形。

图形的朝向控制没有菜单选项,但是可以利用鼠标直接进行调整。只需在图形

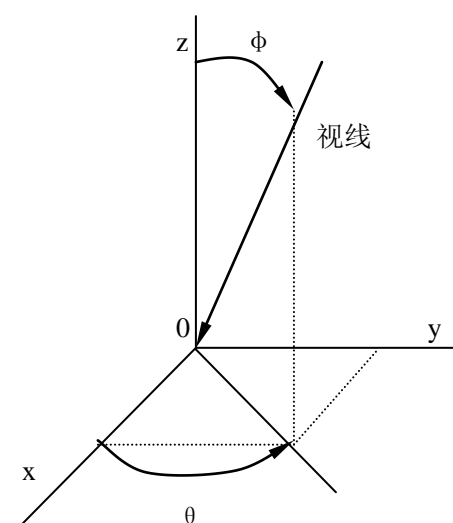


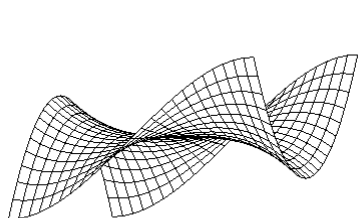
图 6.1 空间朝向的确定

上按住鼠标左键，朝所需转动的方向拖动，图形的朝向就会相应地改变，操作十分方便。

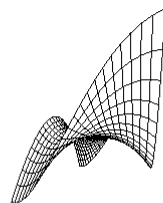
6.7.5 透视投影

三维空间的图形，要显示在平面上，必须采用一定的投影方式。在 Maple 中，我们可以用绘图选项 `projection` 确定图形的投影方式。`Projection` 必须设定成一个介于 0 和 1 之间的数值，数值的大小表示视点距离物体的远近，0 表示非常贴近被观察物体，绘图的效果类似于鱼眼镜头的摄影效果，透视效果非常明显；1 表示距离被观察物体无穷远，也就是平行投影，没有透视效果。Maple 中有三个预设的投影效果 `FISHEYE`，`NORMAL`，`ORTHOGONAL`，分别对应着取 0，0.5，1 的情况。在菜单 `Projection` 中也有相应的选项 `No Perspective`（无透视），`Near Perspective`（近距透视），`Medium Perspective`（中距透视），`Far Perspective`（远透视）对应着不同的投影方式。

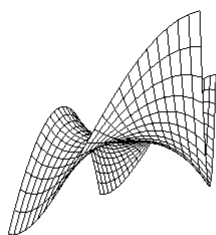
为了体会不同投影方式的效果，我们来看同一个曲面在不同透视距离时的图形（注意图形朝向也就是视角并未发生改变）：



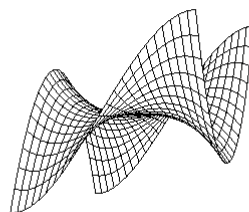
(a) 无透视 (No Perspective)



(b) 近距透视 (Near Perspective)



(c) 中距透视 (Medium Perspective)



(d) 远透视 (Far Perspective)

图 6.2 曲面的不同透视效果

6.7.6 网格大小

和二维绘图不同，在三维绘图中 Maple 没有相应的逐步求精采样算法，采样点数或者说网格数是确定的。默认情况下，Maple 采用 25×25 的网格绘制曲面。如果需要人为加以限制，可以用 `grid = [m, n]` 在绘图时确定网格数为 $m \times n$ ，也就是在 x 方向分为 m 等分， y 方向分为 n 等分。

和前面所讲的选项不同，网格划分时必须在绘图之前确定的选项。也就是说，图形一旦绘制完成，这一选项就无法再改动，否则必须重新计算；而前面介绍的选项都是图形的外观选项，无需改动图形的数据点。这也就是网格数没有对应的菜单选项的原因。

6.7.7 观察区域

和二维图形一样，在 Maple 中，不仅可以设置绘图的自变量变化范围，还可以设置观察的区域，也就是输出图形的区域。可以只设置 z 方向的范围（另两个方向的范围和自变量的变化范围是相同的），当然也可以设置三个方向的范围。设置观察区域的范围需要在绘图时在 `plot3d` 中加入可选参数，加参数的格式可以使下面两种中的任何一种：

$$\text{view} = z_{\min} \dots z_{\max}$$

$$\text{view} = [x_{\min} \dots x_{\max}, y_{\min} \dots y_{\max}, z_{\min} \dots z_{\max}]$$

6.7.8 光照模型

为了使所绘制的三维图形具有立体感，仅仅使用着色是不够的，采用一定的光照模型可以极大地增强图形的表现效果。在 Maple 中用 `plot3d` 等函数绘制三维图形的时候，可以用 `light = [ϕ, θ, r, g, b]` 设置光源。其中 ϕ 和 θ 是光源的方位，定义和图形朝向中的视线角度定义相同， r 、 g 、 b 是设置光源的颜色的，分别表示光源的红、绿、蓝色分量，它们在 0~1 之间取值。

但是，这样定义光源的位置和属性不十分方便，往往经过多次的调整，还打不到满意的效果。Maple 中预设了几种光照模型，我们可以用 `lightmodel` 选项加以设置，预设的值有 `light1`, `light2`, `light3`, `light4` 四种，默认的情况是 `none`，表示不使用光照模型。这些光照模型可以通过菜单 `Color` 中直接予以选择。下图所示是对前面的例子采用光照模型 `light4` 后的效果。

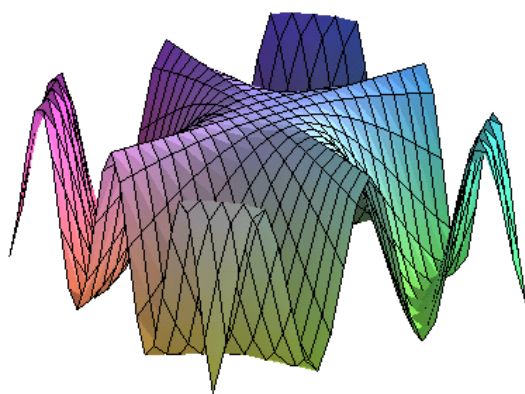


图 6.3 光照模型的使用

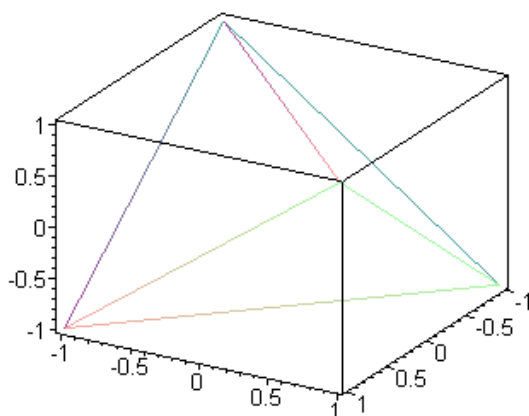
在这一节中，我们介绍了三维绘图中的常用选项。其他还有一些不常用的选项，这里不再一一介绍，如果需要，可以察看 Maple 的在线帮助 `plot3d`, `options`。我们介绍的方法都是针对单个图形而言的，如果需要改动系统的默认绘图选项，可以调用函数 `setoptions3d` 进行

设置。

6.8 三维图形对象的结构

和二维图形一样，Maple 绘制三维图形的过程也可以分为两步。其一，Maple 计算各个网格点上的函数值，并将它们和绘图参数一起存入三维图形对象——PLOT3D 中；第二步，Maple 将 PLOT3D 对象绘制到屏幕或其他设备上。先来看一个简单的例子：

```
> PLOT3D( POLYGONS( [ [1,1,1], [-1,-1,1], [-1,1,-1] ],
  [ [1,1,1], [-1,-1,1], [1,-1,-1] ],
  [ [-1,1,-1], [1,-1,-1], [-1,-1,1] ],
  [ [-1,1,-1], [1,-1,-1], [1,1,1] ] ),
  STYLE(LINE), AXESSTYLE(BOX), ORIENTATION(30, 60) );
```



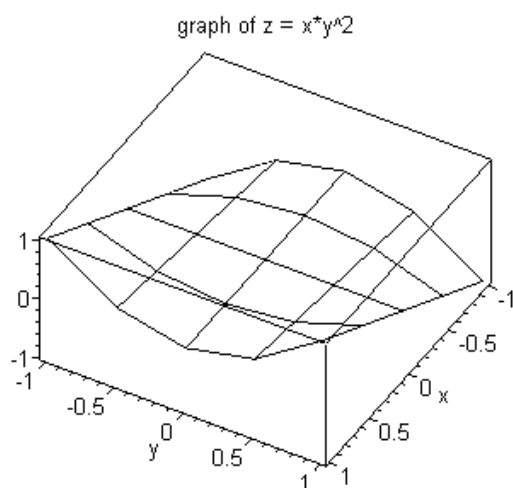
有了前一节中绘图选项的知识，相信读者不难理解上面例子中图形对象参数的意义。这是一个以点 $(1, 1, 1)$, $(-1, -1, 1)$, $(-1, 1, -1)$, $(1, -1, -1)$ 为顶点的四面体，以线框样式绘图，坐标轴采用框式的，观察方向为 $(30^\circ, 60^\circ)$ 。

用 Maple 三维绘图函数 `plot3d` 得到的三维图形对象也具有这样的形式。只不过如果采用矩形网格，那么 PLOT3D 中的第一个数据结构将不再是 POLYGONS，而是一个形式类似于矩阵的数据结构 GRID。例如，我们用默认的矩形网格绘制二元函数 $f(x, y) = x y^2$ 确定的空间曲面：

```
> P := plot3d( x*y^2, x=-1..1, y=-1..1, grid=[5,5],
  axes=boxed, orientation=[30,30], style=patch,
  shading=none, title="graph of z = x*y^2" );
P:=PLOT3D(GRID(-1..1, -1..1, [[-1, -0.25, 0, -0.25, -1], [-0.5, -0.125, 0,
-0.125, -0.5], [0, 0, 0, 0, 0], [0.5, 0.125, 0, 0.125, 0.5], [1, 0.25, 0, 0.25, 1]]),
  COLOUR(NONE), AXESSTYLE(BOX), STYLE(PATCH),
  TITLE("graph of z = x*y^2"), AXESLABELS(x, y, ), PROJECTION(30., 30., 1))
```

有了这些数据点，Maple 下一步就可以用线性插值的方法绘制整个曲面了。

```
> P;
```



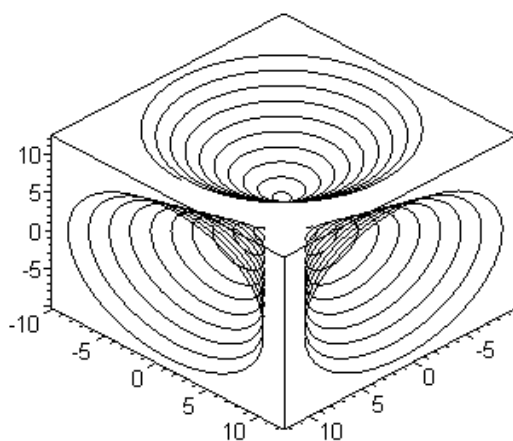
6.9 特殊三维图形的绘制

在本章第四节中，已经介绍了一些特殊的二维图形的绘制。同样，Maple 也提供了一些特殊三维图形的绘制函数。它们中的大部分位于 Maple 的绘图工具包 `plots` 中。在这一节中，我们将对其中一些常用的特殊图形绘制函数作一简单介绍。

6.9.1 空间参数曲线和参数曲面

在 Maple 的 `plots` 工具包中有绘制空间参数曲线的函数 `spacecurve`。它标准的调用格式为：`spacecurve(L, options)`。

```
> with(plots):
> spacecurve( {[t*cos(2*Pi*t), t*sin(2*Pi*t), 2+t],
               [2+t, t*cos(2*Pi*t), t*sin(2*Pi*t)],
               [t*cos(2*Pi*t), 2+t, t*sin(2*Pi*t)] },
             t=0..10, shading=none,
             numpoints=400, style=line, axes=boxed );
```

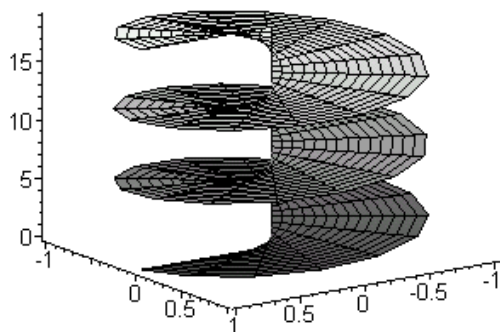


其中，第一个参数 L 是一个数据的有序表或者一个表示参数曲线的有序表，表示参数曲线的有序表必须具有三个或四个元素，前三个分别表示 x, y, z 的参数表达式，最后一个（可选）为参数的变化范围。 $options$ 是绘图的选项。在上面的例子中，我们在同一张图上绘制了三条参数曲线。

要绘制参数曲面，只需调用 `plot3d` 函数。绘制由参数方程 $x = f(s, t), y = g(s, t), z = h(s, t)$ 所确定的曲面，标准的调用格式为：`plot3d([$f(s, t), g(s, t), h(s, t)$], $s = a .. b, t = c .. d$], $options$)`。其中 $a .. b$ 和 $c .. d$ 是参数 s 和 t 的取值范围， $options$ 是绘图选项。

作为例子，我们绘制一个螺旋面：

```
> plot3d( [ r*cos(phi), r*sin(phi), phi ],
  r = 0..1, phi = 0..6*Pi, grid = [ 15,45 ],
  style = patch, orientation = [ 55, 70 ],
  shading = zgrayscale, axes = framed );
```

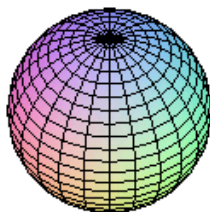


6.9.2 球坐标和柱坐标下的绘图

Maple 也支持球坐标和柱坐标下的三维图形绘制，它们分别是 `plots` 工具包中的 `sphereplot` 和 `cylinderplot` 函数。下面通过两个简单的例子来说明它们的使用：

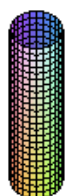
首先通过一个常函数 1 绘制球坐标中的单位球面：

```
> sphereplot( 1, theta=0..2*Pi, phi=0..Pi,
  style=patch, scaling=constrained );
```



接下来用常函数 $1/2$ 在柱坐标下绘制一个圆柱面：

```
> cylinderplot( 1/2, theta=0..2*Pi, z=-2..2,
               style=patch, scaling=constrained );
```



用 `plots` 中的函数 `display3d` 可以将多个三维图形对象绘制在同一张图中，这样，可以通过多个简单图形合成比较复杂的图形。例如，我们可以这样将上面两个图形合成起来：

```
display3d( {%, %%}, ... )
```

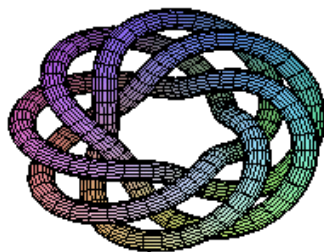
6.9.3 管状图形的绘制

在 `plots` 工具包中有一个函数 `tubeplot`，它专门用于绘制由一条或者多条空间曲线定义的管状图形。

作为例子，我们用它绘制一个较为复杂的三维管状图：

```
> r := 2 + 4/5*cos(7*t): z := sin(7*t):
> curve := [ r*cos(4*t), r*sin(4*t), z ]:
> tubeplot( curve, t=0..2*Pi, radius=1/4,
            numpoints=200, tubepoints=20,
            orientation=[45, 10], style=patch,
            title="torusknot of type 4,7");
```

torusknot of type 4,7



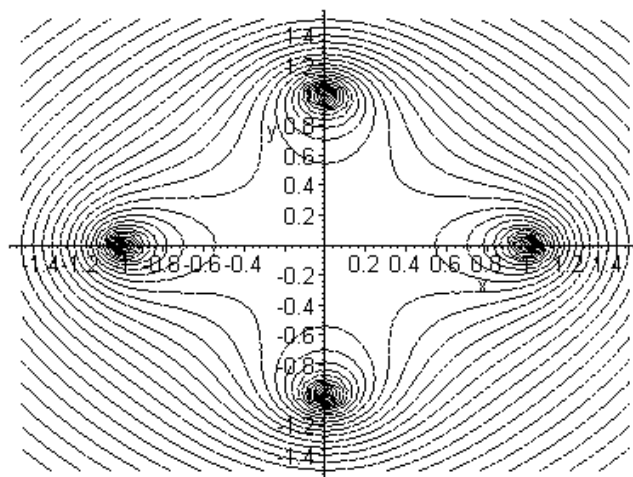
6.9.4 等高线图绘制

等高线图在工程和科学领域的许多方面都有应用,例如二维数据场的可视化等等。Maple 中也提供了绘制等高线图的函数 `contourplot`。等高线的条数可以在函数调用时用 “`contour = 条数`” 设置,默认情况下是 20 条。

作为例子,我们来绘制平面上四个同性点电荷 $(-1, 0)$, $(1, 0)$, $(0, 1)$, $(0, -1)$ 产生的电场的等势线。从物理学中知道,一组在 \mathbf{P}_i 的点电荷 q_i 在 \mathbf{P} 点的电势有计算公式:

$$U = \sum_i q_i \ln(|\mathbf{P} - \mathbf{P}_i|)$$

```
> U := log(sqrt((x+1)^2+y^2))+log(sqrt((x-1)^2+y^2))
+ log(sqrt((y+1)^2+x^2))+log(sqrt((y-1)^2+x^2));
> contourplot( U, x=-3/2..3/2, y=-3/2..3/2,
contours=30, numpoints=2500, color=black );
```



平面的数据场还有一个表现方法,就是平面密度图,它用平面上个点点的不同灰度或者不同颜色来表示不同的数值分布。在 `plots` 工具包中,对应的函数是 `densityplot`:

```
> densityplot( U, x=-3/2..3/2, y=-3/2..3/2,
numpoints=1225, axes=boxed );
```

在大多数情况下,将两者结合起来使用会有较好的表现效果。(只需用 `display` 同时显示

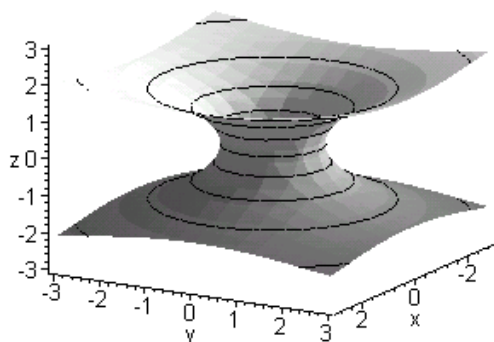
两个图形对象就可以将两者绘制在同一张图上。)

6.9.5 隐式曲面的绘制

在 Maple 中, 可以绘制笛卡尔坐标系下由隐函数确定的曲面, 所需的函数是 `plots` 中的 `implicitplot3d`。不过, 和二维情况下一样, 这个函数只能处理非奇异的曲面。

作为例子, 我们绘制由隐函数 $\cosh z = \sqrt{x^2 + y^2}$ 定义的旋转悬链曲面 (Catenoid):

```
> implicitplot3d( cosh(z) = sqrt( x^2 + y^2 ),
  x=-3..3, y=-3..3, z=-3..3, grid=[15,15,20],
  style=patchcontour, axes=framed,
  shading=zgrayscale, lightmodel=light4,
  orientation=[30, 70]);
```

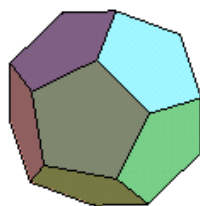


6.9.6 多面体的绘制

多面体是立体几何中经常研究的对象, Maple 的 `plots` 工具包中的函数 `polyhedraplot` 可以直接绘制多种特殊的多面体。

作为例子, 用函数 `polyhedraplot` 绘制正十二面体 (dodecahedron)。

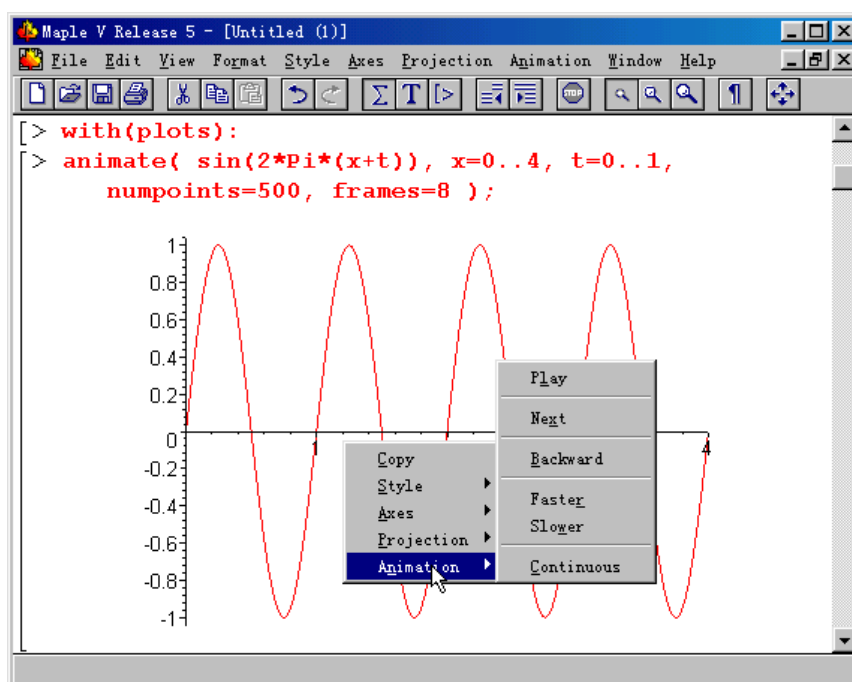
```
> polyhedraplot( [0,0,0], polytype=dodecahedron,
  scaling=constrained, orientation=[70, 65]);
```



函数 `polyhedraplot` 的第一个参数是多面体的中心坐标，其他大部分选项和 `plot3d` 等三维绘图函数相同，它所独有的是 `polytype` 选项，它用来确定所绘制的多面体类型。它可以绘制的特殊多面体种类多达 123 种，这里无法一一列举，读者在需要的时候可以调用函数 `polyhedra_supported()` 来获得所有它能绘制的所有的多面体名称。

6.10 图形动画的制作

在本章的最后一节中，将介绍 Maple 中的图形动画制作。Maple 中的动画是由一系列的静止图形（帧）快速地连续播放形成的。最简单的制作二维图形动画的方法是使用 `plots` 工具包中的函数 `animate`，如图 6.4 所示。



刚生成的动画对象是静止的，要播放动画，只需要在动画对象上单击鼠标右键，在弹出菜单中选择 `Animation | Play` 就可以播放（也可在选中动画对象后选择主菜单 `Animation | Play`），选择 `Animation | Next` 可以单帧播放。如果需要调整播放的速度，可以选择 `Animation | Faster` 或 `Slower`，它们分别可以加快和减慢播放速度。在默认情况下，动画只播放一遍，选择 `Animation | Continuous` 可以进行循环播放。

与前面介绍过的大多数绘图函数类似，函数 `animate` 的标准调用格式是：

`animate (F(x, t), x = xmin .. xmax, t = tmin .. tmax, options)`

其中，第一个变量 $x = x_{\min} \dots x_{\max}$ 指定了空间的变化范围，第二个变量 $t = t_{\min} \dots t_{\max}$ 指定了时间的变化范围。*options* 是绘图选项，它支持大多数的二维绘图选项，利用它们可以绘制特殊图形的动画，例如极坐标下的动画等（参见 6.4 特殊图形的绘制）。除了通用的绘图选项外，还有动画的专用选项 `frames`，利用它可以指定所绘动画的帧数，也就是时间坐标上的采样点数，默认情况下将生成一个 16 帧的动画。

除了 `animate` 外, `plots` 中还有一个二维绘图函数 `animatecurve`, 它是用来演示二维曲线的绘制过程的。也就是生成从没有曲线开始, 逐步绘制曲线, 直到画完整条曲线的动画过程。这个函数在编写教学或者演示程序中有一定的用途。

Maple 不仅可以绘制二维动画, 也可以绘制三维的图形动画。和 `animate` 相对应, 绘制三维曲面形成的动画可以用 `plots` 工具包中的函数 `animate3d`, 它的标准调用格式为:

`animate3d` ($F(x, y, t)$, $x = x_{\min} \dots x_{\max}$, $y = y_{\min} \dots y_{\max}$, $t = t_{\min} \dots t_{\max}$, *options*)

其中, 前两个变量 $x = x_{\min} \dots x_{\max}$ 和 $y = y_{\min} \dots y_{\max}$ 指定了空间的变化范围, 第三个变量 $t = t_{\min} \dots t_{\max}$ 指定了时间的变化范围。*options* 是三维绘图选项, 包括动画选项 `frames`。

在简单介绍了动画的制作方法后, 我们对动画对象的数据结构作简要的说明。

先用 `implicitplot` 生成一个由二维曲线组成的数组:

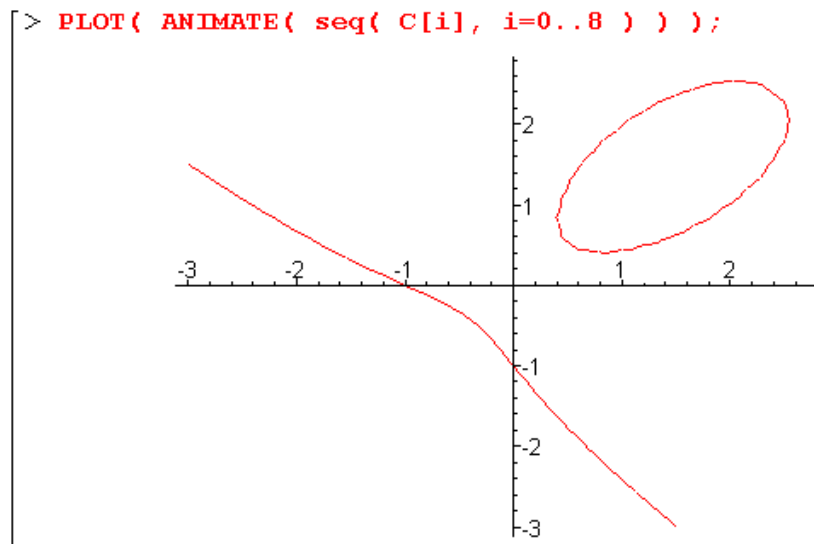
```
> for i from 0 to 8 do
    P[i] := implicitplot( x^3 + y^3 - 5*x*y = 1-i/4,
                        x = -3..3, y = -3..3 )
od:
```

在前面我们已知道二维图形的数据结构, 其中第一部分是图形的数据 `CURVE(...)`, 我们将它们单独提取出来, 存储在数组 `C` 中:

```
> for i from 0 to 8 do
    C[i] := [ op(1, P[i]) ]
od:
```

从这些曲线数据出发, 我们可以构成一个动画对象, 它有着这样的结构:

`PLOT(ANIMATE([图形数据1], ..., [图形数据n]))`

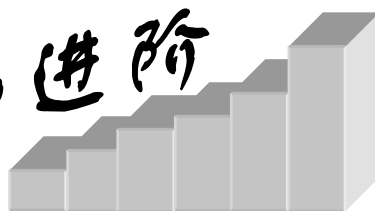


这样的方式生成的动画对象, 和用 `animate` 直接生成的具有相同的属性。

当然, 如果仅仅是为了生成隐函数定义的动画, 或者把已有的图形对象连接成动画, 不需要这样麻烦。可以直接调用 `plots` 工具包中的 `display` 函数, 加以参数 `insequence = true`, 就可以得到所需的动画对象了。如下例所示, 由于结果和上面的一样, 这里将输出略去了。

```
> display( [ seq( P[i], i=0..8 ) ], insequence=true );
```

起步与进阶



第

七

章

方程求解

本章将介绍用 Maple 求解代数方程和方程组，常微分方程和偏微分方程的方法。将涉及到利用 Maple 求得方程的精确解（解析解）和数值解的不同方法，以及对于各种特殊方程的求解技巧。

本章具体包括以下内容：

- 🕒 代数方程的求解
- 🕒 代数方程组的求解
- 🕒 求解代数方程的特殊方法
- 🕒 常微分方程的解析解
- 🕒 常微分方程的数值解法
- 🕒 常微分方程的常用近似解法
- 🕒 偏微分方程简介

在这一章中，我们将学习在 Maple 中实现的各种求解方程和方程组的方法。再学习方程的精确解法的同时，也将介绍各种数值解法，并通过例子使读者能够快速的掌握它们的用法。对于许多常微分方程（组），Maple 都可以显式或隐式地给出它的们解析解。而在 Maple 中，许多常用的近似解法也都予以了实现，例如级数方法，渐进解法等等。我们还将介绍利用 Maple 解决一部分偏微分方程问题的方法。

7.1 代数方程

7.1.1 单未知数的方程

在前面的学习过程中，已经遇到过一些简单的代数方程，我们已经知道，Maple 的函数 solve 可以用来解决代数方程的求解问题。例如，我们先来看一个简单的代数方程：

```
> eqn := ( x^2 + x + 2 ) * ( x - 1 );
                               eqn := (x^2 + x + 2)(x - 1)
> solve( eqn, x );
                               -1/2 + 1/2 I sqrt(7), -1/2 - 1/2 I sqrt(7), 1
```

方程中自然也可以包含有其他的未知参数，solve 将会把待求的未知数用其他未知参数来表示：

```
> eqn := x^3 - 3*a*x^2 + x = 1;
                               eqn := x^3 - 3ax^2 + x = 1
> solve( eqn, x );
1/6 %1 (1/3) - 1/3 a^2
- 1/12 %1 (1/3) + 3 (1/3 - a^2) / (1/3) + a + 1/2 I sqrt(3) (1/6 %1 (1/3) + 1/3 - a^2) / (1/3)
- 1/12 %1 (1/3) + 3 (1/3 - a^2) / (1/3) + a - 1/2 I sqrt(3) (1/6 %1 (1/3) + 1/3 - a^2) / (1/3)
%1 := -108a + 108 + 216a^3 + 12 sqrt(93 - 27a^2 - 162a + 324a^3)
```

Maple 找到了这个三次方程的三个解，包括两个虚数解，结果是用一个表达式序列的形式给出的。在这里，我们选用的输出方式是固定格式输出，即菜单 Option | Output Display |

Typeset Notation (参见第0章)。这样,在表达式比较复杂而且有重复部分的情况下,Maple会自动地把其中的重复部分用临时的全局变量%1, %2等替换,以简化表达式。对于这些临时,用户一直可以引用,直到它们被赋予了新的值。但实际上,只有在它们刚出现时的引用才是可靠的,因为系统可能随时将这些变量重新赋值。而对于这一系列%起始的变量,只要其中有一个被重新赋值,其他所有的也就不复存在了。例如:

```
> (%2)^2 + 1;

$$\frac{\left(\frac{1}{3} - \frac{25}{9}a^2\right)^2}{\left(\frac{2}{3}\right)^2} + 1$$


$$(-180a + 108 + 1000a^3 + 12\sqrt{93 - 75a^2 - 270a + 1500a^3})$$

> solve( x^3 + x = 1, x );

$$\frac{1}{6}\%1\left(\frac{1}{3}\right) - 2\frac{1}{\left(\frac{1}{3}\right)}, -\frac{1}{12}\%1\left(\frac{1}{3}\right) + \frac{1}{\left(\frac{1}{3}\right)} + \frac{1}{2}I\sqrt{3}\left(\frac{1}{6}\%1\left(\frac{1}{3}\right) + 2\frac{1}{\left(\frac{1}{3}\right)}\right),$$


$$-\frac{1}{12}\%1\left(\frac{1}{3}\right) + \frac{1}{\left(\frac{1}{3}\right)} - \frac{1}{2}I\sqrt{3}\left(\frac{1}{6}\%1\left(\frac{1}{3}\right) + 2\frac{1}{\left(\frac{1}{3}\right)}\right)$$

%1 := 108 + 12√93
> %2;
undefined label
```

7.1.2 solve 函数的缩略形式

solve 函数的第一个参数是有待求解的方程或方程的集合,然而,Maple 相当友好,它也可以接受单个的表达式或者表达式的集合(没有等号的式子)。默认情况下,它可以省略方程中的“=0”不写。

```
> solve( a + ln(x-3) - ln(x), x );

$$3\frac{e^a}{-1+e^a}$$

```

对于第二个参数,标准的格式是未知变量或者变量的集合。在它被省略时,Maple 调用 indets 函数获得未知变量, indet (eqns, 'name') minus {constants}, 以它作为函数的第二个参数。在方程(组)没有未知参数的情况下,这个缩略形式时非常方便的,但如果方程具有未知的参数,有时会得到意想不到的结果:

```
> solve( a + ln(x-3) - ln(x) );
{a = -ln(x-3) + ln(x), x = x}
```

这里,Maple 把{a, x}作为未知数集合,而 x = x 表示 x 可以取任何值。

上面说的式输入的简化格式，输出的简化我们在前面已经遇到过——系统用 %n 形式的临时变量简化结果。对于多项式方程的解，Maple 还常常使用另一种简化输出：

```
> x^7 - 2*x^6 - 4*x^5 - x^3 + x^2 + 6*x + 4;
      7      6      5      3      2
x  - 2x  - 4x  - x  + x  + 6x + 4
> solve(%);
      1+sqrt(5), 1-sqrt(5), RootOf(_Z^5 - _Z - 1)
```

这里，Maple 的结果表示，它找到了这个方程的两个实根—— $1+\sqrt{5}$ 和 $1-\sqrt{5}$ ，而其他的五个根，用 RootOf 表示成方程 $_Z^5 - _Z - 1 = 0$ 的解。

7.1.3 一些困难

很多情况下，我们知道一类方程或者方程组有解，但却没有解决这类方程的一般解法，或者说没有解析解。比如，众所周知，对于一般的五次和五次以上的多项式，它的根就不能用够写成解析表达式了。对于 Maple，也不能有太多的期望，它所具备的仅仅是用所有可用的一般算法来尝试。实际上，有一些问题不能用一般方法解决，Maple 就不能给出它的结果，但由于它的特殊性，往往是可以特殊方法解决的。

在遇到找不到解的时候，Maple 将用 RootOf 给出形式解：

```
> solve( cos(x)=x, x );
      RootOf(_Z - cos(_Z))
```

有时，Maple 甚至对于“显而易见”的结果置之不理，例如：

```
> solve( sin(x) = 3*x/Pi , x );
      RootOf(3 _Z - sin(_Z) pi)
```

这个方程的解我们很清楚，稍稍画一下函数的简图，就可以得到方程所有的根： $\pm \frac{\pi}{6}$ 和

0。然而 Maple 对于这个超越方程却无能为力，即使用 allvalues 来获取这个形式解的具体根，也只能得到一个解——0（对于 0 解，Maple 似乎情有独钟，无论什么时候，只要存在 0 解，Maple 一定会给出它来的）。

```
> allvalues(%);
      0
```

在求解方程之前，Maple 会对所有的方程表达式进行化简，而不管表达式的类型。例如，我们知道，Maple 不会主动对分式化简。但在 solve 中就不同了：

```
> (x-1)^2 / (x^2-1);
      (x-1)^2
      x^2 - 1
> solve(%);
      1
```

化简 $\frac{(x-1)^2}{x^2-1}$ 得到 $\frac{x-1}{x+1}$ ，Maple 就自然得出了 1 这样的解。

总而言之, `solve` 的确是一个实用的工具。但是, 切不可盲目相信它给出的结果, 特别是对于非线性方程而言, 对于给出的结果需要加以检验。

通常情况下, 我们遇到的方程解的个数不会太多。但有时, 却会因为方程解的个数受到约束, 不能获得所有的解。例如下面这个方程组, 每一个方程都独立地有两个解 $\{0, 1\}$, 所以方程组的解的组数是 $2^7 = 128$ 组。但是, 默认情况下却不能获得所有的解:

```
[> eqns := { seq( x[i]^2 = x[i], i=1..7 ) };
      eqns := {x4^2 = x4, x3^2 = x3, x1^2 = x1, x2^2 = x2, x6^2 = x6, x7^2 = x7, x5^2 = x5}
[> nops( {solve(eqns)} );
      100
```

获得的解的组数是 100 组, 这是由于 Maple 的环境变量 `_MaxSols` 限制了方程的解的个数。我们可以人为地改变 `_MaxSols` 的值, 来求得所有的解:

```
[> _MaxSols := 200:
[> nops( {solve(eqns)} );
      128
```

7.1.4 方程组的解

在上一小节的最后一个例子中, 我们已经看到了用 `solve` 解方程组的方法, 和求解单个方程不同的是方程和未知数都是以集合的形式给出的。

关于线性方程组的求解, 我们在线性代数一章中已经作了介绍, 这里不再重复。在这里, 我们再看几个非线性方程组的例子。

例 7.1 求解方程组

$$\begin{cases} x^2 + y^2 = 25 \\ x^2 - 9 = y \end{cases}$$

```
[> eqns := { x^2 + y^2 = 25, y = x^2 - 5 };
      eqns := {x^2 + y^2 = 25, y = x^2 - 5}
[> vars := {x, y}:
[> solve( eqns, vars );
      {y = -5, x = 0}, {y = -5, x = 0}, {y = 4, x = 3}, {y = 4, x = -3}
```

这个问题是非常简单的, 但我们通常遇到的非线性问题不是这么简单。比如说, 我们要解更复杂一点方程组: $x^2 + y^2 = 1, \sqrt{x+y} = x - y$ 。

```
[> eqns := { x^2 + y^2 = 1, sqrt( x+y ) = x - y };
      eqns := {x^2 + y^2 = 1, sqrt(x+y) = x - y}
[> sols := solve( eqns, vars );
      sols := {y = 0, x = 1}, {
      x = -2 - RootOf(2 _Z^2 + 4 _Z + 3, -1.000000000 - .7071067812 I),
      y = RootOf(2 _Z^2 + 4 _Z + 3, -1.000000000 - .7071067812 I)}
```

方程解的形式是以集合的序列给出的，序列中的每一个集合是方程的一组解。这样就很利于我们用 subs 把解代入到原方程组进行检验：

```
> subs( sols[1], eqns );
{1=1}
```

对于用 RootOf 表示的解，我们也可以用 allvalues 获得它们的表达式进行检验：

```
> sols2 := allvalues(sols[2]);
sols2 := {x=-1-1/2*I*sqrt(2), y=-1+1/2*I*sqrt(2)}, {x=-1+1/2*I*sqrt(2), y=-1-1/2*I*sqrt(2)}
> simplify( subs( sols2[1], eqns ) );
{1=1, I*sqrt(2)=-I*sqrt(2)}
> simplify( subs( sols2[2], eqns ) );
{1=1, I*sqrt(2)=I*sqrt(2)}
```

我们又一次地发现 Maple 给出了错误的解。实际上，在求解非线性方程过程中，增根是难以避免的，而 Maple 又不主动对每一个根进行检验。所以，和求解单个方程一样，在获得解之后，必须用 subs 检验。

7.2 其他求解工具

7.2.1 数值求解

对于求代数方程的数值解问题，Maple 提供了函数 fsolve。fsolve 的使用方法和 solve 很相似。例如：

```
> x^7 - 2*x^5 + 3*x^3 - 4*x + 5;
x^7 - 2x^5 + 3x^3 - 4x + 5
> fsolve( % );
-1.476154724
```

除此之外，fsolve 还有一些与 solve 不同的可选参数。比如，我们可以加入第三个参数 complex，指定 fsolve 在复数域中求解方程。比如上面这个例子中，由代数学知识可以知道该方程在复数域中有 7 个根：

```
> fsolve( %, x, complex );
-1.476154724, -1.019075478 - .9587951646 I, -1.019075478 + .9587951646 I,
.5606858790 - .8548132750 I, .5606858790 + .8548132750 I,
1.196466961 - .4732220981 I, 1.196466961 + .4732220981 I
```

除了指定在复数域中求解以外，在 fsolve 中还可以指定在实数域的某一个区间中求解：

```
> fsolve( x^2 - 3*x + 1, x, 1..5 );
2.618033989
```

对于多项式的根，fsolve 在默认情况下可以给出所有的实数解，如果附加参数 complex，就可以给出所有的解。但对于更一般的其他形式的方程，fsolve 却往往只满足于得到一个解。

例如:

```
> eqn := sin(x) = x/2;
                                
$$eqn := \sin(x) = \frac{1}{2}x$$

> fsolve( eqn );
                                0
> fsolve( eqn, x, 0.1 .. infinity );
                                1.895494267
> fsolve( eqn, x, -infinity .. -0.1 );
                                -1.895494267
```

函数 fsolve 主要基于两个算法——通常使用牛顿法, 如果牛顿法无效, 它就改而使用切线法。为了使 fsolve 可以求得所有的实根, 我们通常需要确定这些根所在的区间。对于单变量多项式, Maple 有一个库函数 realroot, 可以获得多项式的所有实根所在的区间。

```
> readlib( realroot );
> 4 + 6*x + x^2 - x^3 - 4*x^5 - 2*x^6 + x^7;
                                
$$4 + 6x + x^2 - x^3 - 4x^5 - 2x^6 + x^7$$

> realroot(%);
                                [[0, 2], [2, 4], [-2, -1]]
```

函数 realroot 还有一个可选参数, 它是用来限制区间的最大长度的, 为了保证使用数值求解方法时收敛, 我们可以用它限制区间的最大长度。

```
> realroot( %, 1/1000 );
                                
$$\left[ \left[ \frac{1195}{1024}, \frac{299}{256} \right], \left[ \frac{3313}{1024}, \frac{1657}{512} \right], \left[ \frac{-633}{512}, \frac{-1265}{1024} \right] \right]$$

```

7.2.2 求方程的整数解

Maple 的函数 isolve 是用来求解方程或方程组的整数解的, 它常常被用来求解不定方程。作为它的一个简单应用, 我们来看一个量纲分析的小例子。

例 7.2 已知一个在流体中快速运动的球型物体的阻力 F 由以下物理量决定: 运动的速度 V , 物体的直径 d , 流体的密度 ρ , 该流体中的声速 c , 动力粘度系数 ν 。用量纲分析的方法分析 F 和 V, d, ρ, c, ν 的关系。

我们的目的是得到用 F, V, d, ρ, c, ν 的幂的乘积表示的无量纲量:

```
> nondimensional := force^f * speed^v * diameter^d
    * density^r * acoustic_velocity^c
    * kinematic_viscosity^m;
nondimensional :=
    
$$force^f speed^v diameter^d density^r acoustic\_velocity^c kinematic\_viscosity^m$$

```

我们知道, 这些物理量都是由长度 L 、质量 M 、时间 T 这三个基本量纲组成的。接着, 我们把它都用基本量纲表示, 把它们各自的量纲表示式都代入到上面定义的量纲量中去:

```
> subs( { force = M*L/T^2, speed = L/T, diameter = L,
          density = M/L^3, acoustic_velocity = L/T,
          kinematic_viscosity = L^2/T }, nondimensional );
```

$$\left(\frac{ML}{T^2}\right)^f \left(\frac{L}{T}\right)^v L^d \left(\frac{M}{L^3}\right)^r \left(\frac{L}{T}\right)^c \left(\frac{L^2}{T}\right)^m$$

为了将同一基本量纲的幂合并进行化简，需要设定这些基本量纲为正实数。

```
> assume(M>0, L>0, T>0);
> simplify(%);
```

$$M^{\sim(f+r)} L^{\sim(f+v+d-3r+c+2m)} T^{\sim(-2f-v-c-m)}$$

要获得无量纲量，所有的基本量纲的指数都需要等于零。由此，可以得到方程组：

```
> eqns := { seq( op(2,i)=0, i=% ) };
          eqns := {f+r=0, f+v+d-3r+c+2m=0, -2f-v-c-m=0}
```

这是一个不定方程组，我们用 `isolve` 来得到它的整数解：

```
> isolve( eqns );
{
  f=-_N3, v=-_N1-_N2+2_N3, d=-_N2+2_N3, r=_N3, c=_N1, m=_N2
}
```

整数解中以下划线开始的变量可以取任意整数。然后，我们将结果代入到我们所需要的无量纲量中去：

```
> subs( %, nondimensional );
```

$$\begin{matrix} force^{(-N3)} & speed^{(-N1-N2+2N3)} & diameter^{(-N2+2N3)} & density^{-N3} \\ acoustic_velocity^{-N1} & kinematic_viscosity^{-N2} & & \end{matrix}$$

Maple 对于幂的连乘积的化简规则是把相同底数的项合并起来，而在这里，为了得到相互无关的无量纲量，需要将相同指数的项合并起来（因为 $_N1$, $_N2$, $_N3$ 是相互独立的整数）。可以编写一小段程序实现这个功能。我们知道，对连乘积求对数后，原来的指数都变成了对数项的系数，然后，我们就可以用函数 `collect` 将相同系数的项合并。最后，再用 `exp` 函数获得我们所需要的形式。

不过，再进行这些运算以前，需要限定这些变量的范围，否则，有一些变换将无法进行下去，例如将指数提出对数之外等等。

```
> assume(force>0, speed>0, diameter>0, density>0,
          acoustic_velocity>0, kinematic_viscosity>0);
> assume(_N1, even);
> assume(_N2, even);
> assume(_N3, even);
```

然后将整个整理过程编写成一个子程序：

```
> extract_powers := proc( expression, exponents )
    simplify( ln(expression) );
    combine( collect( %, exponents ), ln );
    simplify( exp(%), exp );
end;
```

利用这个子程序，就可以将具有指数_N1, _N2, _N3 的项合并起来：

$$\left[\begin{array}{l} > \text{extract_powers}(t, \{ _N1, _N2, _N3 \}); \\ \left(\frac{\text{acoustic_velocity}}{\text{speed}} \right)^{_N1} \left(\frac{\text{kinematic_viscosity}}{\text{speed} \cdot \text{diameter}} \right)^{_N2} \\ \left(\frac{\text{speed}^2 \cdot \text{diameter}^2 \cdot \text{density}}{\text{force}} \right)^{_N3} \end{array} \right]$$

至此，我们已经找到了用这些物理量表示的三个无量纲量。它们都是流体力学中重要的无量纲参数。

7.2.3 \mathbf{Z}_n 中的方程求解

利用 Maple 的函数 `msolve` (`eqns, vars, n`)，可以在模 n 的整数环中求解方程（组）`eqns`。

例 7.3 在 \mathbf{Z}_7 中求解 Pell 方程：

$$y^2 = x^3 - 28$$

$$\left[\begin{array}{l} > \text{msolve}(y^2 = x^3 - 28, 7); \\ \{x=0, y=0\}, \{x=1, y=1\}, \{x=1, y=6\}, \{y=1, x=2\}, \{y=6, x=2\}, \\ \{y=1, x=4\}, \{y=6, x=4\} \end{array} \right]$$

`msolve` 在 \mathbf{Z}_7 范围内得到了该方程的七组解。

7.2.4 递归方程的求解

在 Maple 中，也可以求解递归形式的函数方程，所需调用的函数是 `rsolve`。函数 `rsolve` 使用的是一些比较通用的方法，例如产生函数法， z 变换法，以及一些基于变量替换和特征方程的方法。作为例子，我们来求解广义菲波那契多项式：

$$\left[\begin{array}{l} > \text{rsolve}(\{ f(n+2) = x \cdot f(n+1) + y \cdot f(n), f(0)=0, \\ f(1)=1 \}, f(n)); \\ \frac{(x^2 + 4y - x\sqrt{x^2 + 4y}) \left(-2 \frac{y}{x - \sqrt{x^2 + 4y}} \right)^n}{(x^2 + 4y)(x - \sqrt{x^2 + 4y})} \\ - \frac{(x\sqrt{x^2 + 4y} + x^2 + 4y) \left(-2 \frac{y}{x + \sqrt{x^2 + 4y}} \right)^n}{(x^2 + 4y)(x + \sqrt{x^2 + 4y})} \end{array} \right]$$

当然，并不是所有的递归形式的函数方程的解都可以写成解析形式的。如果不能写成解析表达式，Maple 将保留原来的调用形式。但是，我们还是可以用 `asympt` 函数获得它的渐进表达式（**asymptotic expression**），也就是 $1/n$ 的级数解。例如，对于一个具有超越形式的递归函数方程，仍然可以得到解的渐进形式。

```

> rsolve( u(n+1) = ln(u(n)+1), u(n) );
      rsolve(u(n+1)=ln(u(n)+1), u(n))
> asympt( %, n, 5 );

$$2\frac{1}{n} + \frac{-C + \frac{2}{3}\ln(n)}{n^2} + \frac{\frac{1}{9} - \frac{1}{3}C + \frac{1}{2}C^2 - \left(-\frac{2}{3}C + \frac{2}{9}\right)\ln(n) + \frac{2}{9}\ln(n)^2}{n^3} + O\left(\frac{1}{n^4}\right)$$


```

7.3 常微分方程的求解

在数学中，常微分方程是这样定义的，它具有一般形式： $F(y, y', y'', \dots, y^{(n)}, x) = 0$ ，其中 $y, y', y'', \dots, y^{(n)}$ 是一元函数 $y(x)$ 对其自变量 x 的各阶导数， F 是定义在 \mathbf{R}^{n+2} 上的实值函数。称该常微分方程为线性的，当且仅当 F 对其前 $n+1$ 个变元而言是线性函数，亦即可以表示成为如下的形式：

$$a_n(x)y^{(n)} + a_{n-1}(x)y^{(n-1)} + \cdots + a_1(x)y' + a_0(x)y + a(x) = 0$$

关于常微分方程，有两个不同的概念，常微分方程的阶数是指方程中出现的最高阶导数的阶数，而次数是指该最高阶项的指数。例如方程 $y'' - x^2y = 0$ 是一个二阶一次常微分方程；而 $(y''')^2 + y' + y = 0$ 是一个三阶二次常微分方程。

在数学中，常微分方程是一个比较成熟的领域，有着多种较为通用的求解方法，Maple 中对一些常用的常微分方程求解方法进行了实现。对于一般的常微分方程，用户可以让 Maple 决定具体用什么方法，也可以指定求解的方法，例如求级数解或者利用拉普拉斯变换求解等。但是，需要说明的是，和其他的微积分领域相比，Maple 求解微分方程的能力是较为不足的。对于特殊的微分方程，Maple 起到的作用往往是辅助性质的。我们将通过例子来说明这一点。

7.3.1 常微分方程的解析解

如果你对微分方程的求解方法所知甚少，那么，对付它的最简单方法就是利用 Maple 的常微分方程求解函数 `dsolve`。例如，对于一阶一次常微分方程 $xy' = y \ln(xy) - y$ ，可以用 `dsolve` 函数直接获得它的解析解：

```

> ODE := x*diff(y(x), x) = y(x)*ln(x*y(x)) - y(x);
      ODE := x \left( \frac{\partial}{\partial x} y(x) \right) = y(x) \ln(x y(x)) - y(x)
> dsolve( ODE, y(x) );

$$y(x) = \frac{e^{(x e^{-Cl})}}{x}$$


```

函数 `dolve` 的第一个参数是一个等式，也就是待求的常微分方程，第二个参数是未知函数。需要注意的是，无论在方程中还是作为第二个参数，未知函数必须用函数的形式给出（也就是必须加括弧，并在其中确定自变量）。这一规定是必须的，否则 Maple 将无法区分方程中的函数、自变量和参变量；但这毕竟和我们通常的微分方程书写方式不一致。为了使它符合我们的书写习惯，并且简化输入，可以用 `alias` 将函数用函数名作为别称表示。

```
[> alias( y = y(x) );
> ODE := x*diff(y, x) = y*ln(x*y) - y;
                                 $ODE := x \left( \frac{\partial}{\partial x} y \right) = y \ln(xy) - y$ 
> dsolve( ODE, y );
                                 $y = \frac{e^{(x e^{(-C1)})}}{x}$ 
```

函数 `dsolve` 给出的是微分方程的通解，其中的任意常数是用下划线起始的内部变量（`_C1`, `_C2`, ……）表示的。

在 Maple 中，微分方程的解是很容易验证的，只需要将解代入到原方程就可以了。和代数方程一样，希望读者养成检验微分方程解的良好习惯。例如对于上一个例子的结果，如果需要代入检验，为了让 Maple 可以将形如 $\ln(e^x)$ 的式子化简，需要用 `assume` 指定 `x` 和 `_C1` 的范围。

```
[> check := subs( %, ODE );
check :=
                                 $x \left( \frac{\partial}{\partial x} \frac{e^{(x e^{(-C1)})}}{x} \right) = \frac{e^{(x e^{(-C1)})}}{x} \ln \left( e^{(x e^{(-C1)})} \right) - \frac{e^{(x e^{(-C1)})}}{x}$ 
> assume(x, real): assume( _C1, real );
> expand( check );
                                 $\frac{e^{\left( \frac{x \sim}{e^{-C1 \sim}} \right)}}{e^{-C1 \sim}} - \frac{e^{\left( \frac{x \sim}{e^{-C1 \sim}} \right)}}{x \sim} = \frac{e^{\left( \frac{x \sim}{e^{-C1 \sim}} \right)}}{e^{-C1 \sim}} - \frac{e^{\left( \frac{x \sim}{e^{-C1 \sim}} \right)}}{x \sim}$ 
> evalb(%);
                                true
```

我们再来看一个常微分方程的例子，这里，要求解方程 $y' = \sqrt{y^2 + 1}$ 。为了取消前面指定的 `x` 的范围，需要对 `x` 取消赋值。

```
[> x := 'x':
> alias( y = y(x) );
> ODE := diff(y, x) = sqrt(y^2 + 1);
                                 $ODE := \frac{\partial}{\partial x} y = \sqrt{y^2 + 1}$ 
> dsolve( ODE, y );
                                 $y = \sinh(x + _C1)$ 
```

函数 `dsolve` 对于求解含有未知参变量的常微分方程也完全可以胜任。例如，我们可以

用它来求解一阶常微分方程 $y' = -\frac{y}{\sqrt{a^2 - y^2}}$ 。

```
[> alias( y=y(x) );
> ODE := diff(y, x) = -y / sqrt(a^2 - y^2);
```

$$ODE := \frac{\partial}{\partial x} y = -\frac{y}{\sqrt{a^2 - y^2}}$$

```
[> sol := dsolve( ODE, y );
```

$$sol := x + \sqrt{a^2 - y^2} - \frac{a^2 \operatorname{arctanh}\left(\frac{\sqrt{a^2 - y^2}}{\sqrt{a^2}}\right)}{\sqrt{a^2}} + _C1 = 0$$

就象在这个例子中一样，对于不能表示成显式结果的微分方程解，Maple 会尽可能地将结果表示成为隐式解。从这个例子中还可以看到，对于微分方程的平凡解 $y \equiv 0$ ，Maple 常常是对其忽略的。

在 Maple 中，求解常微分方程的初值问题也是十分方便的。

例如，我们可以用 dsolve 求解数学摆的自由振动方程： $u'' + \omega^2 u = 0$ 。与前面介绍的唯一不同点是，需要将方程和初值组成一个集合，作为第一个参数提供给 dsolve。

```
[> ODE := diff(u(t), t$2) + omega^2 * u(t) = 0;
```

$$ODE := \left(\frac{\partial^2}{\partial t^2} u(t) \right) + \omega^2 u(t) = 0$$

```
[> dsolve( { ODE, u(0)=u0, D(u)(0)=v0 }, u(t) );
```

$$u(t) = \frac{v0 \sin(\omega t)}{\omega} + u0 \cos(\omega t)$$

在这里， $u(t)$ 一阶导数在 $t = 0$ 点的初值是用 $D(u)(0)$ 表示的，同样高阶的导数 $u'(0)$ ， $u''(0)$ 等分别可以用 $(D@@2)(u)(0)$ ， $(D@@3)(u)(0)$ 表示。（关于 D 操作符，请参考第三章；关于函数复合操作符 @，请参考第一章。）

7.3.2 利用积分变换求解微分方程

对于特殊的常微分方程，我们还可以指定 dsolve 利用积分变换方法进行求解。只需要在 dsolve 中加入可选参数 `method = transform` 就可以了，其中 `transform` 是特定的积分变换，可以是 `laplace`，`fourier`，`fouriercos` 或者 `fouriersin`，分别表示用拉普拉斯变换，富利叶变换和富立叶正弦、余弦变换进行求解。

作为例子，我们来看一个具有线性阻尼的振子在阶跃冲击（Heaviside 函数）下的响应。

$$u''(t) + 2c\omega u'(t) + \omega^2 u(t) = \operatorname{Heaviside}(t)$$

$$u(0) = 0, \quad u'(0) = 0$$

首先在 Maple 中建立方程和初始条件：

```
[> ODE := diff(u(t), t$2) + 2*damp*omega*diff(u(t), t)
      + omega^2*u(t) = Heaviside(t);
      ODE :=  $\left(\frac{\partial^2}{\partial t^2} u(t)\right) + 2 \text{damp} \omega \left(\frac{\partial}{\partial t} u(t)\right) + \omega^2 u(t) = \text{Heaviside}(t)$ 
[> initvals := {u(0)=0, D(u)(0)=0}:
```

然后用 dsolve 求解，并限制求解方法为拉普拉斯变换法：

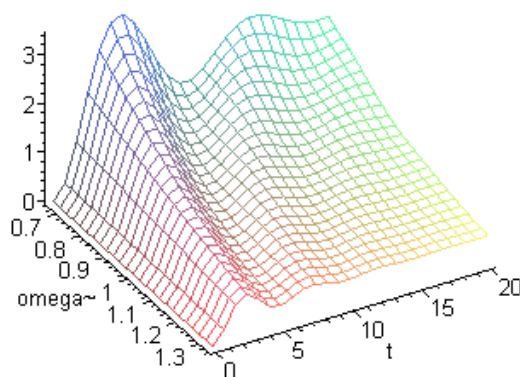
```
[> solution := dsolve( {ODE, initvals}, u(t),
      method=laplace );
      solution :=  $u(t) = \frac{1}{\omega^2} + \frac{e^{\left(\left(-\text{damp} \omega + \frac{1}{2} \sqrt{4 \text{damp}^2 \omega^2 - 4 \omega^2}\right) t\right)}}{-\text{damp} \omega + \frac{1}{2} \sqrt{4 \text{damp}^2 \omega^2 - 4 \omega^2}} - \frac{e^{\left(\left(-\text{damp} \omega - \frac{1}{2} \sqrt{4 \text{damp}^2 \omega^2 - 4 \omega^2}\right) t\right)}}{-\text{damp} \omega - \frac{1}{2} \sqrt{4 \text{damp}^2 \omega^2 - 4 \omega^2}} \Bigg/ \sqrt{4 \text{damp}^2 \omega^2 - 4 \omega^2}$ 
```

Maple 给出了问题的通解，它没有区分自由振动（damp = 0），欠阻尼（0 < damp < 1），临界阻尼（damp = 1）和过阻尼（damp > 1）的情况。如果我们加以区分，分别求解，结果将会得到简化：

```
[> assume(omega>0):
[> simplify(subs(damp=0, solution));
       $u(t) = -\frac{\cos(\omega \sim t) - 1}{\omega^2}$ 
```

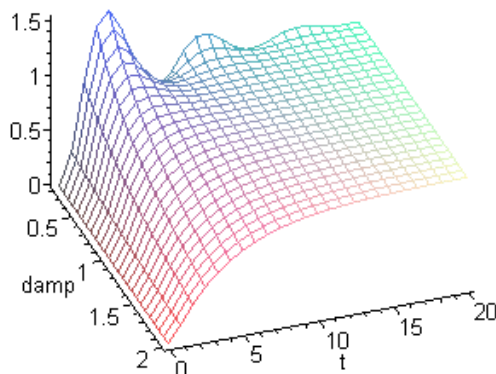
我们还可以将结果绘制成图形，例如，对于上面的解在欠阻尼（damp = 1/5）时的情况，可以对一定的频率范围绘制解的曲面图：

```
[> subs( damp = 1/5, solution ):
[> plot3d( rhs(%), omega=2/3..4/3, t=0..20,
      style=hidden, orientation=[-30, 45],
      axes=framed );
```



当然，也可以绘制给定固有频率，而阻尼作为参变量的曲面：

```
[> omega := 1:
> plot3d( rhs(solution), 'damp'=1/5..2, t=0..20,
        style=hidden, orientation=[-20, 45],
        axes=framed );
```



对于临界阻尼的情况，由于分母中会有 0 出现，不能直接代入。不过，我们还可以用 `limit` 求极限来获得结果：

```
[> limit(rhs(solution), damp=1);
      (e^(omega~t) - omega~t - 1) e^(-omega~t)
      -----
      omega~^2
```

7.3.3 常微分方程组的求解

函数 `dsolve` 不仅可以用来求解单个的常微分方程，也可以求接连立的常微分方程组。特别是对于线性常微分方程组，由于数学上具有成熟的理论，Maple 的求解也是得心应手。例如下面的常系数线性常微分方程组：

```
[> ODE := diff(f(t), t$2) - 6*diff(g(t), t) = 6*sin(t),
        6*diff(g(t), t$2) + c^2*diff(f(t), t) = 6*cos(t);
      ODE := (frac{partial^2 f(t)}{partial t^2}) - 6 (frac{partial g(t)}{partial t}) = 6 sin(t), 6 (frac{partial^2 g(t)}{partial t^2}) + c^2 (frac{partial f(t)}{partial t}) = 6 cos(t)
[> initvals := f(0)=0, g(0)=1, D(f)(0)=0, D(g)(0)=1:
[> funcs := {f(t), g(t)}:
```

求解常微分方程组的调用格式和单个方程一样，所不同的是把所有的方程（包括方程和定解条件）都写在一个集合中，作为第一个参数传递给 `dsolve`；而所有的未知函数也都写在一个集合中，作为 `dsolve` 的第二个参数。

其他的附加参数（例如用 `method` 确定求解的方法等）也和求解单个方程式一样。


```
> dsolve( {ODE, initvals}, funcs );
```

$$\left\{ \begin{aligned} f(t) &= \left(-12 \frac{\sin(ct) c^3}{c^2 - 1} - 6 + 12 \frac{c \sin(ct)}{c^2 - 1} - 6 \cos(ct) c^2 + 12 \sin(t) c^2 + 6 c^2 \right. \\ &\quad \left. + 6 \cos(ct) \right) / (c^2 (c - 1) (c + 1)), g(t) = \frac{1}{6} \left(-12 \frac{\cos(ct) c^3}{c^2 - 1} + 6 \sin(ct) c^2 \right. \\ &\quad \left. + 12 \frac{\cos(ct) c}{c^2 - 1} + 6 c^3 \cos(t) + 6 c \cos(t) - 6 \sin(ct) \right) / (c (c - 1) (c + 1)) \end{aligned} \right\}$$

这个结果是用 c 的有理式的形式给出的，看起来比较复杂。我们可以利用多项式函数 `collect`（参见第二章）合并其中的同类项，并且用 `collect` 的附加参数 `normal` 消去结果中分子和分母的公因子。

```
> collect( %, {sin(t), cos(t)}, normal );
```

$$\left\{ \begin{aligned} f(t) &= 12 \frac{\sin(t)}{(c-1)(c+1)} - 6 \frac{-c^2 + \cos(ct) c^2 + 2c \sin(ct) + 1 - \cos(ct)}{(c+1)(c-1)c^2}, \\ g(t) &= \frac{(c^2+1)\cos(t)}{(c-1)(c+1)} + \frac{\sin(ct) c^2 - 2\cos(ct)c - \sin(ct)}{c(c-1)(c+1)} \end{aligned} \right\}$$

7.4 常微分方程的级数解法

7.4.1 泰勒级数法

当一个常微分方程的解析解难以求得时，我们仍可以用 Maple 求得方程解的泰勒级数。这在大多数情况下，这还是一种非常好的近似结果。

作为例子，我们用泰勒级数法求解物理摆的角振动方程：

$$l\theta'' = -g \sin \theta$$

其中 l 是物理摆的摆长， θ 是摆角， g 是当地的重力加速度。我们要求解的是没有激励的情况下，在 v_0 初速度下的自由振动。

首先，还是在 Maple 中先定义要解的微分方程和定解条件：

```
> ODE := l*diff(theta(t), t$2) = -g*sin(theta(t));
```

$$ODE := l \left(\frac{\partial^2}{\partial t^2} \theta(t) \right) = -g \sin(\theta(t))$$

```
> initvals := theta(0)=0, D(theta)(0)=v0/l;
```

$$initvals := \theta(0) = 0, D(\theta)(0) = \frac{v_0}{l}$$

用泰勒级数法解微分方程，用的还是函数 `dsolve`，只不过需要加上参数 `type = series`。我们在第二章中介绍级数时讲过，级数的默认展开阶数是由环境变量 `Order` 决定的，在这里

也如此。默认情况下，Order 的值是 6。如果需要改变，可以先对它赋值。例如，我们对物理摆问题求它的 9 阶泰勒级数解：

```
> sol := dsolve( { ODE, initvals }, theta(t),
  type = series );
```

$$\text{sol} := \theta(t) = \frac{v_0 t}{l} - \frac{1}{6} \frac{g v_0 t^3}{l^2} + \frac{1}{120} \frac{g v_0 (g l + v_0^2) t^5}{l^4} - \frac{1}{5040} \frac{g v_0 (g^2 l^2 + 11 g l v_0^2 + v_0^4) t^7}{l^6}$$

7.4.2 幂级数解法

对于一个符号代数系统来说，幂级数是必不可少的微分方程求解工具。在 Maple 的幂级数工具包 powseries 中就有幂级数求解函数 powsolve。不过，这一工具的使用范围很有限，它只可以用来解决多项式系数的线性常微分方程（或方程组）。

我们首先来看一个贝赛尔方程的简单例子：

$$xy'' + y' + 4x^2y = 0$$

对于它，Maple 可以给出用贝赛尔函数表示的解析结果：

```
> ODE := x*diff(y(x), x$2) + diff(y(x), x)
  + 4*x^2*y(x) = 0;
```

$$ODE = x \left(\frac{\partial^2}{\partial x^2} y(x) \right) + \left(\frac{\partial}{\partial x} y(x) \right) + 4x^2 y(x) = 0$$

```
> dsolve( ODE, y(x) );
```

$$y(x) = _C1 \text{BesselJ}\left(0, \frac{4}{3}x^{\frac{3}{2}}\right) + _C2 \text{BesselY}\left(0, \frac{4}{3}x^{\frac{3}{2}}\right)$$

下面，我们用函数 powsolve 来求解在初值 $y(0) = 1$ ， $y'(0) = 0$ 下的幂级数解：

```
> initvals := y(0)=1, D(y)(0)=0;
> with( powseries );
> sol := powsolve( { ODE, initvals } );
```

$$\text{sol} := \text{proc}(\text{powparm}) \dots \text{end}$$

可以看到，powsolve 给出的是子程序形式的幂级数，为了得到截断形式的近似解，可以用 tpsform 获得幂级数的截断形式（参见第二章）：

```
> tpsform( sol, x, 16 );
```

$$1 - \frac{4}{9}x^3 + \frac{4}{81}x^6 - \frac{16}{6561}x^9 + \frac{4}{59049}x^{12} - \frac{16}{13286025}x^{15} + O(x^{16})$$

也可以用 powsolve 给出的函数直接获得用递归形式定义的幂级数系数，不过，参数必须用 _k，因为这是 powsolve 使用的临时变量。

```
[> sol(_k);
```

$$-4 \frac{a(_k-3)}{_k^2}$$

上面这个式子表示该幂级数具有递归形式的系数： $a_k = -4 \frac{a_{k-3}}{k^2}$ 。

幂级数解法的第二个例子是量子力学中的一个经典例子——一维谐振子的解。薛定谔方程在无量纲化后具有这样的形式：

$$y'' + (\varepsilon - x^2)y = 0$$

```
[> alias( y=y(x), h=h(x) );
> ODE := diff(y, x$2) + (epsilon-x^2)*y = 0;
```

$$ODE := \left(\frac{\partial^2}{\partial x^2} y \right) + (\varepsilon - x^2)y = 0$$

然后，做变量替换 $y(x) = h(x)e^{-x^2/2}$ ，方程可以整理为：

```
[> subs( y=exp(-x^2/2)*h, ODE );
> collect( %, exp(-x^2/2) ) / exp(-x^2/2);
```

$$-h + x^2 h - 2x \left(\frac{\partial}{\partial x} h \right) + \left(\frac{\partial^2}{\partial x^2} h \right) + (\varepsilon - x^2)h = 0$$

```
[> ODE := collect( %, [diff(h, x$2), diff(h, x), h] );
```

$$ODE := \left(\frac{\partial^2}{\partial x^2} h \right) - 2x \left(\frac{\partial}{\partial x} h \right) + (-1 + \varepsilon)h = 0$$

我们再用 powsolve 求解这个变换后的方程：

```
[> H := powsolve( ODE );
> h := tpsform( H, x, 8 );
```

$$h = C0 + C1x - \frac{1}{2}(-1 + \varepsilon)C0x^2 - \frac{1}{6}(-3 + \varepsilon)C1x^3 + \frac{1}{24}(-5 + \varepsilon)(-1 + \varepsilon)C0x^4 + \frac{1}{120}(-7 + \varepsilon)(-3 + \varepsilon)C1x^5 - \frac{1}{720}(-9 + \varepsilon)(-5 + \varepsilon)(-1 + \varepsilon)C0x^6 - \frac{1}{5040}(-11 + \varepsilon)(-7 + \varepsilon)(-3 + \varepsilon)C1x^7 + O(x^8)$$

还可以得到递归形式的幂级数系数：

```
[> H(_k);
```

$$-\frac{(3 + \varepsilon - 2_k)a(_k-2)}{_k(_k-1)}$$

当然，这个递归表达式还可以写成： $(k+1)(k+2)a_{k+2} = (2k+1-\varepsilon)a_k$ 。如果存在整数 k ，使 $\varepsilon = 2k+1$ 成立，那么，这个级数就是一个有限项级数。

7.5 常微分方程数值解法

在 Maple 中，我们还可以求得常微分方程初值问题的数值解，调用的函数还是 `dsolve`，只需要加入参数 `type = numeric`（有时可以省略“`type =`”）。

对于常微分方程初值问题的数值解，Maple 中可选用的方法很多。我们可以在 `dsolve` 中用参数“`method = 方法参数`”进行指定。可供选择的方法见表 7.1。在默认情况下，`dsolve` 将选择 `method = rkf45`，也就是用 4-5 阶龙格库塔法进行求解。

表 7.1 常微分方程的数值解法

方法参数	具体方法
rkf45	4-5 阶变步长 Runge-Kutta-Fehlberg 法
dverk78	7-8 阶变步长 Runge-Kutta-Fehlberg 法
classical	经典方法（包括向前欧拉法，改进欧拉法，2、3、4 阶龙格库塔法，Adams-Bashford 方法等等）
gear	吉尔单步法
mgear	吉尔多步法
lsode	

7.5.1 变步长龙格库塔法

作为例子，我们用 4-5 阶 Runge-Kutta-Fehlberg 法求解 Van de Pol 方程：

$$y'' - (1 - y^2)y' + y = 0$$

在初始条件 $y(0) = 0$, $y'(0) = -0.1$ 下的数值解。

```
> ODE := diff(y(t), t$2) - (1-y(t)^2)*diff(y(t), t) +
    y(t) = 0;

$$ODE = \left( \frac{\partial^2}{\partial t^2} y(t) \right) - (1 - y(t)^2) \left( \frac{\partial}{\partial t} y(t) \right) + y(t) = 0$$

> initvals := y(0)=0, D(y)(0)=-0.1:
> F := dsolve( {ODE, initvals}, y(t), type=numeric );
    F := proc(rkf45_x) ... end
```

函数 `dsolve` 返回的是一个函数，我们可以在给定的数值点上对它求值。

```
> F(0);

$$\left[ t = 0, y(t) = 0, \frac{\partial}{\partial t} y(t) = -0.1 \right]$$

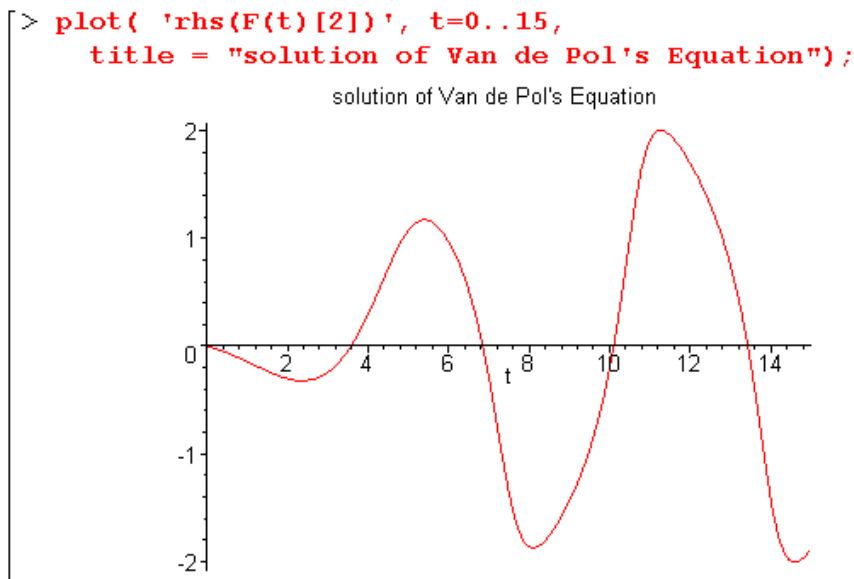
> F(1);

$$\left[ t = 1, y(t) = -.1447686096006437, \frac{\partial}{\partial t} y(t) = -.1781040958088073 \right]$$

```

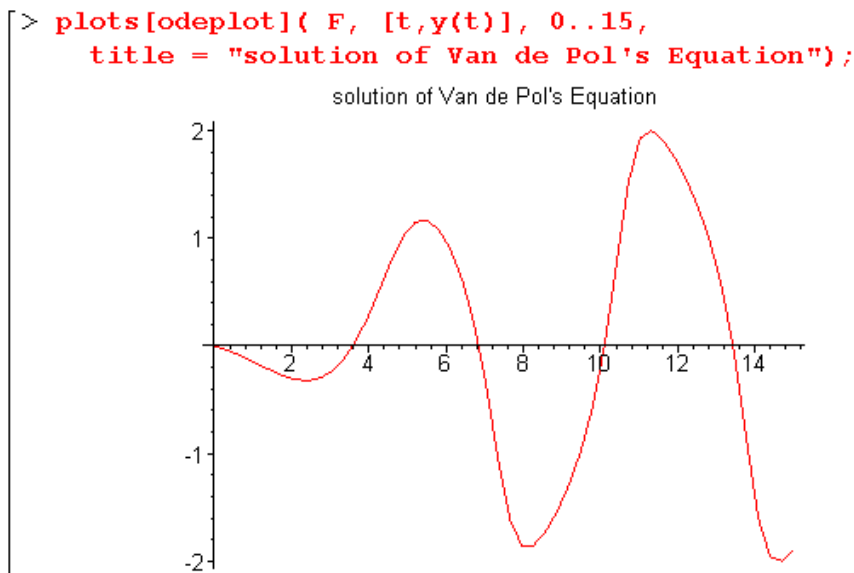
可以看到，F 给出的是一个包括 t , $y(t)$, $y'(t)$ 在内的有序表，它对于每一个时间点可以给出一组数值表达式。有序表的每一项都是一个等式，如果我们需要对数值解画图，则要用

rhs 取等式的右边部分。(由于每一点的计算都要分别进行,所以在作为 plot 的参数时,必须用一对单引号 “'” 将函数扩起来,以延迟求值时间。



我们知道,常微分方程的数值解法是一个递推方法,也就是计算每一个函数值,需要前一个时间点(或者前几个点)的函数值(或导数值)。而在上面的绘图过程中,并没有考虑到这一特点,所以计算每一个点时,都需要从初值开始从头算起,有很大部分的重复运算,有时会需要很长的计算时间。

如果不需要很高的绘图精度,可以直接用 plots 工具包中的常微分方程解的绘图工具 odeplot, 如下面的例子所示。它充分利用了数值解法的特点,依次地求出每一个值并绘制图形。但是,不可避免地,它的图形精度不如 plot 所绘制的高。



很明显,由于没有用逐步求精的绘图算法,在图形的转折点处光滑性不是太好。

7.5.2 刚性方程和吉尔法

在科学和工程计算中，常常会遇到这样一类常微分方程问题。它可以表示成方程组：

$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}), \mathbf{y}(t_0) = \mathbf{y}^{(0)}$ ，之所以称之为刚性方程组，是因为解的分量数量级相差很大，分量的变化速度也相差很大。如果用常规方法求解，为了使快变分量有应有的精度，必须取很小的步长，而为了使慢变分量达到近似的稳态解，则需要很长的时间。这样用小步长计算大时间跨度，必定造成庞大的计算量，而且会使误差不断积累。

吉尔法（Gear method）是专门用来求解刚性方程的一种数值方法。

作为例子，我们用它来求解一个刚性方程组：

$$\begin{cases} \frac{du}{dt} = -2000u + 999.75v + 1000.25 \\ \frac{dv}{dt} = u - v \end{cases}$$

$u(0) = 0, v(0) = -2$

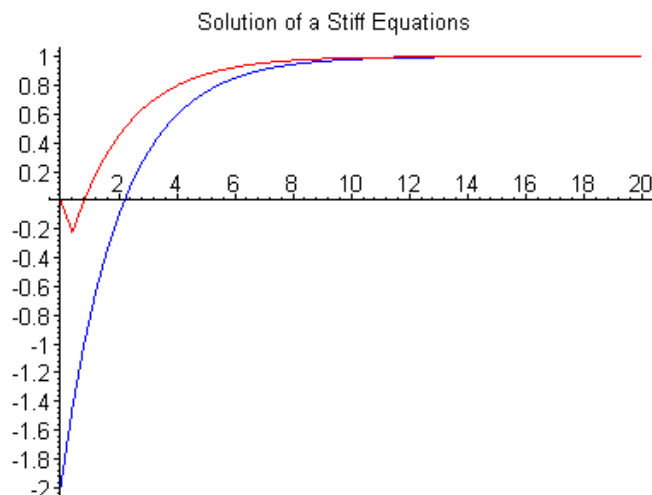
```

> ODE := diff(u(t), t) = - 2000*u(t) + 999.75*v(t)
      + 1000.25, diff(v(t), t) = u(t) - v(t);
      ODE :=  $\frac{\partial}{\partial t} u(t) = -2000 u(t) + 999.75 v(t) + 1000.25, \frac{\partial}{\partial t} v(t) = u(t) - v(t)$ 
> initvals := u(0) = 0, v(0) = -2;
      initvals := u(0) = 0, v(0) = -2
> ans1 := dsolve( {ODE, initvals}, {u(t), v(t)},
      type=numeric, method=gear );
      ans1 := proc(x_gear) ... end
> ans1(10.0);
      [t = 10.0, u(t) = .9898939191610572, v(t) = .9797878427705208]
  
```

对于这个结果，我们可以用 `plots[odeplot]` 绘制解曲线来观察方程解的稳定情况：

```

> p1 := plots[odeplot](ans1, [t, u(t)], 0..20, color=red);
> p2 := plots[odeplot](ans1, [t, v(t)], 0..20, color=blue);
> plots[display]( {p1, p2},
      title = "Solution of a Stiff Equations" );
  
```



7.5.3 经典数值方法

Maple 中常微分方程数值解法中有一类被称作是“经典”(classical)方法。当然,称它们为“经典”方法,不是因为它们常用,或是精度高,是因为它们的形式简单,经常被用于计算方法课上的教学内容。它们是一些常见的固定步长方法,在 `dsolve` 中用参数 `method = classical[方法名称]`。如果不予指定,它将默认采用向前欧拉法。

我们把这些方法列成下表。

表 7.2 经典数值方法

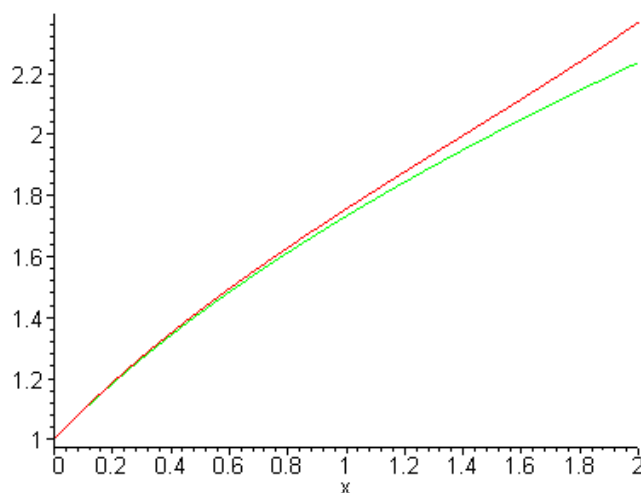
方法名称	具体方法
foreuler	向前欧拉法(默认)
heunform	Heun 公式法(梯形方法,改进欧拉法)
impoly	改进多项式法
rk2	二阶龙格库塔法
rk3	三阶龙格库塔法
rk4	四阶龙格库塔法
adambash	Adams-Bashford 方法(预测法)
abmoulton	Adams-Bashford-Moulton 方法(预测-矫正法)

作为例子,我们用欧拉法解一个简单的初值问题,并且就这个例子再介绍一些常微分方程数值求解的附加参数。待求解的方程是: $y' = y - 2\frac{x}{y}$, 初值: $y(0) = 1$ 。

```
> sol1 := dsolve( diff(y(x), x) = y(x) - 2*x/y(x),
  y(x), numeric, method=classical, initial=array([1]),
  stepsize = 0.1, start=0 );
sol1 := proc(x_classical) ... end
```

为了将这个数值解和精确解作比较,我们再用 `dsolve` 求出这个方程的解析解来,将数值结果绘制在同一图中进行比较:

```
> sol2 := dsolve( {diff(y(x), x) = y(x) - 2*x/y(x),
  y(0)=1}, y(x) );
sol2 := y(x) = sqrt(2*x+1)
> plot({rhs(sol2), 'rhs(sol1(x)[2])'}, x=0..2);
```



从图中可以清楚地看到，在经过一段时间之后，欧拉法的数值结果就有了很大的误差。

我们在这个例子中，用到了一些附加参数。例如，用 `initial = array([1])` 指定初值，用 `stepsize = 0.1` 指定步长为 0.1，用 `start = 0` 指定初始的时刻等等。其他的参数请参考表 7.3。

表 7.2 微分方程数值方法的参数

参数名	参数类型	参数用途	参数用法
<code>initial</code>	浮点数的一维数组	指定初值向量	
<code>number</code>	正整数	指定方程个数	
<code>output</code>	'procedurelist' (默认) 或 'listprocedure'	指定生成单个函数或 多个函数的有序表	'procedurelist': 单个函数，返回有序表 'listprocedure': 函数的有序表
<code>procedure</code>	子程序名	用子程序形式指定第 一类常微分方程组的等 号右边部分。	参数 1: 未知函数个数 参数 2: 自变量 参数 3: 函数向量 参数 4: 导函数向量
<code>start</code>	浮点数	自变量起始值	
<code>startinit</code>	布尔量 (默认 FALSE)	指定数值积分是否总 是从起始值开始	对 <code>dverk78</code> 不适用
<code>value</code>	浮点数向量 (一维数组)	指定需要输出函数值 的自变量数值点	如果给定，结果是一个 2×1 的矩阵。 元素 [1, 1] 是一个向量，包含自变量名 和函数名称；元素 [2, 1] 是一个数值矩 阵，其中第一列 <code>value</code> 的输入相同，其 他列中是相应的函数值

对于这一节中介绍的经典方法，还有一些特殊的附加参数，这里将它们列成表 7.3。

表 7.3 经典方法的参数

参数名	参数类型	参数用途	参数用法
<code>maxfun</code>	整数	最大的函数值数量	默认值 50000，为负数时表示无限制
<code>corrections</code>	正整数	指定每步修正值数量	在 <code>abmoulton</code> 中使用，建议值 ≤ 4
<code>stepsize</code>	浮点数值	指定步长	默认值：向前， $\min((T_{out}-T)/3, 0.005)$ ； 向后， $\max((T_{out}-T)/3, -0.005)$

7.6 非线性常微分方程的扰动法

如果需要用扰动法求常微分方程的近似解，那么，符号代数系统将是最好的助手。再这一节中，我们将简单介绍两种经典的扰动方法——庞加莱法和多尺度法。并且，我们将利用这两种方法分别求解 van der Pol 方程。

7.6.1 庞加莱法 (Poincaré-Lindstedt Method)

在这一节中，我们将主要围绕 van der Pol 方程：

$$y'' - \varepsilon(1 - y^2)y' + y = 0$$

展开讨论。当 $\varepsilon = 0$ 时，问题就简化为一个数学摆的振动方程。对于非 0 的 ε ，方程的解都会趋向于一个稳态的周期解，我们称之为极限环（limit cycle）。

需要解决的问题是对于较小的 ε ，找到一个近似的极限环，亦即近似的稳态解。由于方程中没有显含时间 t 的项，所以，不失一般性，可以假设在 0 点的初始值为 $y(0) = 0$ 。在庞加莱法中，需要先将时间无量纲化，我们作变换：

$$\tau = \omega t$$

其中，

$$\omega = 1 + \omega_1 \varepsilon + \omega_2 \varepsilon^2 + \omega_3 \varepsilon^3 + \dots$$

作了这样的变换后，van der Pol 方程可以写成为关于 $y(\tau)$ 的方程：

$$\omega^2 y'' - \omega \varepsilon (1 - y^2) y' + y = 0$$

同样，我们也假设 $y(\tau)$ 可以展开成 ε 的泰勒级数：

$$y(\tau) = y_0(\tau) + y_1(\tau)\varepsilon + y_2(\tau)\varepsilon^2 + y_3(\tau)\varepsilon^3 + \dots$$

接下来，将 $y(\tau)$ 和 $\omega(\varepsilon)$ 的展开式带入到 van der Pol 方程中去，将 ε 不同次数的项合并起来，就可以得到 ε 的低次项系数的方程：

$$y_0'' + y_0 = 0$$

$$y_1'' + y_1 = y_0'(1 - y_0^2) - 2\omega_1 y_0''$$

$$y_2'' + y_2 = (1 - y_0^2)y_1' - 2y_0 y_1 y_0' - 2\omega_1 y_1'' - (2\omega_2 + \omega_1^2)y_0'' + \omega_1(1 - y_0)x_0'$$

原来的初值 $y(0) = 0$ 经过变换，成为一组初值：

$$y_0(0) = 0, y_1(0) = 0, y_2(0) = 0, y_3(0) = 0, \dots$$

下面，我们用 Maple 在推导一下前面的过程，一方面检验其结果，另一方面也为进一步的计算定义一些变量和表达式。

为了输入简便，避免每次都输入长长的希腊字母名，我们用 w, e, T 分别代替 $\omega, \varepsilon, \tau$ ，首先利用 alias 设置他们为等同的别称：

```
[> alias( omega=w, epsilon=e, tau=T ):]
```

然后，我们定义变换以后的 van der Pol 方程方程：

```
[> ODE := w^2*diff(y(T), T$2) -  
      w*e*(1-y(T)^2) * diff(y(T), T) + y(T) = 0;  
      ODE = w^2*(d^2 y(T)/d tau^2) - w*e*(1-y(T)^2)*(d y(T)/d tau) + y(T) = 0
```

接下来，将 y_1, y_2, y_3, \dots 定义为 τ 的函数：

```
[> e_order := 6:  
> for i from 0 to e_order do  
      y.i := T->y.i(T)  
od:
```

再把 ω 和 w 都定义为 ε 的截断幂级数的形式：

```

[> w := 1 + sum( 'w.i*e^i', 'i'=1..e_order );
      w = 1 + w1 ε + w2 ε2 + w3 ε3 + w4 ε4 + w5 ε5 + w6 ε6
[> y := sum( 'y.i*e^i', 'i'=0..e_order );
      y = y0 + y1 ε + y2 ε2 + y3 ε3 + y4 ε4 + y5 ε5 + y6 ε6

```

由于 ε 是与 τ 无关的常数，需要显式地加以说明。我们将代入的结果化简，并且略去 ε 的高次项。

```

[> e := {}->e:
[> deqn := simplify( collect( ODE, e ),
      {e^(e_order+1)=0} );

```

至此，就可以将 ε 不同次项的系数方程分离出来了。

```

[> for i from 0 to e_order do
      eqn.i := coeff( lhs(deqn), e, i ) = 0
    od:

```

为了检验得到的结果，我们将前面的几项打印出来：

```

[> eqn0;
      
$$\left( \frac{\partial^2}{\partial \tau^2} y_0(\tau) \right) + y_0(\tau) = 0$$

[> eqn1;
      
$$\left( \frac{\partial^2}{\partial \tau^2} y_1(\tau) \right) + 2w_1 \left( \frac{\partial^2}{\partial \tau^2} y_0(\tau) \right) - \left( \frac{\partial}{\partial \tau} y_0(\tau) \right) + \left( \frac{\partial}{\partial \tau} y_0(\tau) \right) y_0(\tau)^2 + y_1(\tau) = 0$$


```

这样，一个非线性的常微分方程就近似成了一系列相互独立的线性常微分方程 eqn0, eqn1, eqn2, ……。首先，利用 dsolve 可以方便地求出 eqn0 在初值 $y_0(0)$ 下的解：

```

[> dsolve( { eqn0, y0(0)=0 }, y0(T) );
      y0(τ) = _C1 sin(τ)

```

我们再将这个结果反解出来，赋给 y0，一边进行下一步 y1(τ) 的求解。

```

[> y0 := unapply( rhs(%), T );
      y0 := T → _C1 sin(T)
[> eqn1;
      
$$\left( \frac{\partial^2}{\partial \tau^2} y_1(\tau) \right) - 2w_1 \_C1 \sin(\tau) - \_C1 \cos(\tau) + \_C1^3 \cos(\tau) \sin(\tau)^2 + y_1(\tau) = 0$$


```

再继续求解前，先将这个表达式进行一定的化简，将它化成为我们所熟悉的有限项三角级数的形式。由于化简是对各个系数进行的，所以需要借用 map 工具。

```

[> map( combine, eqn1, trig );
[> eqn1 := map( collect, %, [sin(T), cos(T)] );
      eqn1 := -2w1 _C1 sin(τ) +  $\left( -\_C1 + \frac{1}{4} \_C1^3 \right) \cos(\tau) + \left( \frac{\partial^2}{\partial \tau^2} y_1(\tau) \right) + y_1(\tau)$ 
      
$$- \frac{1}{4} \_C1^3 \cos(3\tau) = 0$$


```

在数学中，我们把上面方程中包含有 $\sin\tau$ 和 $\cos\tau$ 的项成为久期项（secular terms）或共振项（resonant terms），它们是方程近似解中非周期成分的来源，正如我们在下面的解中可以看到。

对于未消除久期项的方程 eqn1，可以用拉普拉斯变换法进行求解：

```
[> dsolve( {eqn1, y1(0)=0}, y1(T), method = laplace );
```

$$y1(\tau) = -\frac{1}{32}C1^3 \cos(3\tau) + \frac{1}{32}C1^3 \cos(\tau) + D(y1)(0) \sin(\tau) + w1_C1 \sin(\tau) \\ + \frac{1}{2}C1 \tau \sin(\tau) - \frac{1}{8}\tau_C1^3 \sin(\tau) - C1 \tau w1 \cos(\tau)$$

```
[> map( collect, %, [sin(T), cos(T), T] );
```

$$y1(\tau) = \left(\left(\frac{1}{2}C1 - \frac{1}{8}C1^3 \right) \tau + w1_C1 + D(y1)(0) \right) \sin(\tau) \\ + \left(\frac{1}{32}C1^3 - C1 \tau w1 \right) \cos(\tau) - \frac{1}{32}C1^3 \cos(3\tau)$$

可以取定一组常数 $C1$ 和 $w1$ ，以消除方程 eqn1 中的久期项：

```
[> solve( { coeff( lhs(eqn1), sin(T) ) = 0,
             coeff( lhs(eqn1), cos(T) ) = 0 }, {_C1, w1} );
```

$$(_C1 = 2, w1 = 0), (_C1 = -2, w1 = 0), (_C1 = 0, w1 = w1)$$

为了和前一节中的数值解比较，不妨取有负值 $C1$ 的一组结果。我们用 assign 函数将它赋值。这样，方程 eqn1 就变成了一个简单的线性常微分方程了：

```
[> assign( %[2] );
```

$$\left(\frac{\partial^2}{\partial \tau^2} y1(\tau) \right) + y1(\tau) + 2 \cos(3\tau) = 0$$

```
[> eqn1;
```

为了避免系统自动出现难以控制的临时变量，我们给定 $y1$ 导数的初值为 $C2$ 。

```
[> dsolve( {eqn1, y1(0)=0, D(y1)(0)=C2}, y1(T),
            method = laplace );
```

$$y1(\tau) = \frac{1}{4} \cos(3\tau) - \frac{1}{4} \cos(\tau) + C2 \sin(\tau)$$

```
[> y1 := unapply( rhs(%), T );
```

$$y1 := T \rightarrow \frac{1}{4} \cos(3T) - \frac{1}{4} \cos(T) + C2 \sin(T)$$

接下来，进行对 eqn2 的求解：

```
[> eqn2;
```

$$\frac{3}{4} \sin(3\tau) - \frac{1}{4} \sin(\tau) - C2 \cos(\tau) + 4 \left(-\frac{3}{4} \sin(3\tau) + \frac{1}{4} \sin(\tau) + C2 \cos(\tau) \right) \sin(\tau)^2 \\ + 8 \cos(\tau) \sin(\tau) \left(\frac{1}{4} \cos(3\tau) - \frac{1}{4} \cos(\tau) + C2 \sin(\tau) \right) + \left(\frac{\partial^2}{\partial \tau^2} y2(\tau) \right) + 4 \sin(\tau) w2 \\ + y2(\tau) = 0$$

需要对 $C2$ 和 $w2$ 作和前面类似的处理，以消去久期项。

```
[> map( combine, eqn2, trig );
```

```

[> eqn2 := map( collect, %, [sin(T), sin(3*T), cos(T),
    cos(3*T)] ):
[> solve( { coeff( lhs(eqn2), sin(T) ) = 0,
    coeff( lhs(eqn2), cos(T) ) = 0 }, {C2, w2} );
                                
$$\{C2 = 0, w2 = \frac{-1}{16}\}$$

[> assign(%):

```

然后, 求解 $y_2(\tau)$:

```

[> dsolve( { eqn2, y2(0)=0, D(y2)(0)=C3 }, y2(T),
    method = laplace ):
[> collect( %, [ sin(T), sin(3*T), sin(5*T), cos(T),
    cos(3*T), cos(5*T) ] ):
[> y2 := unapply( rhs(%), T );
                                
$$y2 = T \rightarrow \left(\frac{29}{96} + C3\right) \sin(T) + \frac{5}{96} \sin(5T) - \frac{3}{16} \sin(3T)$$


```

通过前面几步的运算, 相信您对于这个过程一定已经熟悉了。我们可以将它简单地描述如下:

1. 整理方程为三角级数形式;
2. 消去久期项 ($\sin\tau$, $\cos\tau$ 的系数);
3. 求解无久期项的线性常微分方程;
4. 将结果反求代入下一方程;
5. 对下一方程, 重复过程 1~4。

在清楚了求解过程后, 对于下面的计算, 我们通过一小段循环程序来完成:

```

[> for i from 3 to e_order do
    map( combine, eqn.i, trig ):
    eqn.i := map( collect, %,
        [ seq(sin((2*j+1)*T), j=0..i),
          seq(cos((2*j+1)*T), j=0..i) ] ):
    solve( { coeff( lhs(eqn.i), sin(T) ) = 0,
        coeff( lhs(eqn.i), cos(T) ) = 0 },
        { C.i, w.i } ):
    assign(%):
    dsolve( { eqn.i, y.i(0)=0, D(y.i)(0)=C.(i+1) },
        y.i(T), method = laplace ):
    collect( %, [ seq(sin((2*j+1)*T), j=0..i),
        seq(cos((2*j+1)*T), j=0..i) ] ):
    y.i := unapply( rhs(%), T )
od:

```

至此, 我们已经得到了方程对于小扰动 ε 的幂级数解, 并且将解的频率也写成了 ε 的幂级数形式:

```

[> w;
                                
$$1 - \frac{1}{16} \varepsilon^2 + \frac{17}{3072} \varepsilon^4 + \frac{35}{884736} \varepsilon^6$$

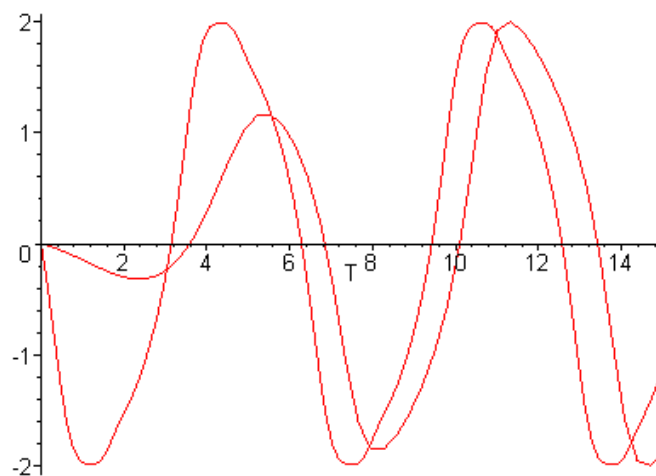

```

$y(\tau)$ 解的表达式中还有一个代定常数 $C7$, 如果继续下一步求解, 就可以确定这一代定常数。不过在这里, 我们只需要展开到 ε^5 的结果。所以, 可以用 $\varepsilon^6 = 0$ 对解再进行一次化简:

```
> y := unapply( simplify( y(T), {e^e_order=0} ), T );
y = T → -2 sin(T) + (-cos(T) + cos(T)3) ε
+ ( - $\frac{11}{8}$  sin(T) cos(T)2 +  $\frac{5}{6}$  sin(T) cos(T)4 +  $\frac{23}{96}$  sin(T) ) ε2
+ ( - $\frac{7}{9}$  cos(T)7 -  $\frac{1381}{576}$  cos(T)3 +  $\frac{89}{36}$  cos(T)5 +  $\frac{45}{64}$  cos(T) ) ε3 + ( - $\frac{64487}{552960}$  sin(T)
+  $\frac{1519}{540}$  sin(T) cos(T)6 -  $\frac{323}{96}$  sin(T) cos(T)4 +  $\frac{365}{256}$  sin(T) cos(T)2
-  $\frac{61}{80}$  sin(T) cos(T)8 ) ε4 + ( - $\frac{471671}{1105920}$  cos(T) +  $\frac{1012987}{129600}$  cos(T)7
-  $\frac{1853173}{259200}$  cos(T)5 +  $\frac{9894757}{3317760}$  cos(T)3 -  $\frac{114941}{28800}$  cos(T)9 +  $\frac{5533}{7200}$  cos(T)11 ) ε5
```

下面，我们取 $\varepsilon = 1$ ，和前一节中直接用数值方法求解 van der Pol 方程的结果进行比较：

```
> e := 1:
> p1 := plot( y(T), T=0..15 ):
> y := 'y':      # 取消赋值
> F := dsolve( { diff(y(t), t$2) - ( 1-y(t)^2 ) *
diff(y(t), t) + y(t) = 0 , y(0) = 0,
D(y)(0) = -0.1 }, y(t), numeric ):
> p2 := plots[odeplot]( F, [t, y(t)], 0..15 ):
> plots[display]( {p1, p2} );
```



7.6.2 多尺度法

从上一节的结果中我们已经看到，庞加莱方法可以很好地求得方程的稳态解；但对于在方程极限环的临域中的解，它就无能为力了。

在多尺度方法（Method of multiple scales）中，方程的解不再被看作单一变量 t 的函数，它是一系列变量 t_1, t_2, t_3, \dots 的函数。其中，自变量 t_1, t_2, t_3, \dots 定义如下：

$$t_1 = t, t_2 = \varepsilon t, t_3 = \varepsilon^2 t, \dots$$

这样，对 t 的导数就可以用对这一系列自变量的导数表示：

$$\frac{d}{dt} = \frac{\partial}{\partial t_1} + \varepsilon \frac{\partial}{\partial t_2} + \varepsilon^2 \frac{\partial}{\partial t_3} + \dots$$

$$\frac{d^2}{dt^2} = \frac{\partial^2}{\partial t_1^2} + 2\varepsilon \frac{\partial^2}{\partial t_1 \partial t_2} + \varepsilon^2 \left(\frac{\partial^2}{\partial t_2^2} + 2 \frac{\partial^2}{\partial t_1 \partial t_3} \right) + \dots$$

我们可以用 Maple 检验这个结果。

```
[> alias( epsilon = e ):
> e_order := 2:
> e := subs( variables = seq(t.j, j=0..e_order),
    body = e, (variables -> body) );
    epsilon := (t0,t1,t2) -> epsilon
> subs( D = sum( 'e^(i-1)*D[i]', 'i'=1..e_order+1 ),
    (D@@2)(y) ):
> simplify( collect(%, e), {e^(e_order+1)=0} );
    D1,1(y)+2*epsilon*D1,2(y)+(2*D1,3(y)+D2,2(y))*epsilon^2
```

再多尺度法中，我们把解展开为 ε 的幂级数，它的每一项系数都是我们设定的时间尺度 t_1, t_2, t_3, \dots 的函数：

$$y = y_0(t_1, t_2, t_3, \dots) + \varepsilon y_1(t_1, t_2, t_3, \dots) + \varepsilon^2 y_2(t_1, t_2, t_3, \dots) + \dots$$

将它代入到方程中，就得到关于 y_0, y_1, y_2, \dots 的微分方程。

下面，我们就用具有三个时间尺度的方法来求解 van der Pol 方程：

$$y'' - \varepsilon(1 - y^2)y' + y = 0$$

在初始条件 $y(0) = 0, y'(0) = -0.1$ 下的近似解。

首先，我们用 Maple 来求得关于 $y_0(t_1, t_2, t_3), y_1(t_1, t_2, t_3), y_2(t_1, t_2, t_3)$ 的微分方程：

```
[> ODE := (D@@2)(y) - e*(1-y^2)*D(y) + y = 0;
    ODE = (D@@2)(y) - epsilon*(1-y^2)*D(y) + y = 0
> subs( D = sum( 'e^(i-1)*D[i]', 'i'=1..e_order+1 ),
    ODE ):
> y := sum( 'y.i*e^i', 'i'=0..e_order );
    y = y0 + y1*epsilon + y2*epsilon^2
> diffeqn := simplify( collect( %, e ),
    {e^(e_order+1) = 0} );
> for i from 0 to e_order do
    eqn.i := coeff( lhs(diffeqn), e, i ) = 0 ;
od;
    eqn0 := D1,1(y0) + y0 = 0
    eqn1 := D1,1(y1) + 2*D1,2(y0) + y1 - D1(y0) + D1(y0)*y0^2 = 0
    eqn2 := -D1(y1) - D2(y0) + y0^2*D1(y1) + y0^2*D2(y0) + 2*y0*y1*D1(y0)
    + 2*D1,2(y1) + y2 + D2,2(y0) + D1,1(y2) + 2*D1,3(y0) = 0
```

这里，得到了一组微分方程，我们可以将它们写成通常的形式：

$$\frac{\partial^2 y_0}{\partial t_1^2} + y_0 = 0$$

$$\frac{\partial^2 y_0}{\partial t_1^2} + y_1 = -2 \frac{\partial^2 y_0}{\partial t_1 \partial t_2} + (1 - y_0^2) \frac{\partial^2 y_0}{\partial t_1}$$

$$\frac{\partial^2 y_2}{\partial t_1^2} + y_2 = -2 \frac{\partial^2 y_0}{\partial t_1 \partial t_2} - \frac{\partial^2 y_0}{\partial t_2^2} - 2 \frac{\partial^2 y_0}{\partial t_1 \partial t_3} + (1 - y_0^2) \left(\frac{\partial y_1}{\partial t_1} + \frac{\partial y_0}{\partial t_2} \right) - 2 y_0 y_1 \frac{\partial y_0}{\partial t_1}$$

其中第一个方程有如下形式的通解:

$$y_0(t_1, t_2, t_3) = A(t_2, t_3) \sin(t_1 + B(t_2, t_3))$$

再将它代入到后面的方程中, 消去久期项, 就得到如下关于 $A(t_2, t_3)$ 和 $B(t_2, t_3)$ 的方程:

$$\frac{\partial B(t_2, t_3)}{\partial t_2} = 0$$

$$\frac{\partial A(t_2, t_3)}{\partial t_2} = \frac{1}{2} A(t_2, t_3) - \frac{1}{8} A(t_2, t_3)^3$$

下面, 我们用 Maple 来完成这一步。首先, 将方程 eqn1 化成导数的形式:

```
> eqn1 := convert( eqn1(t1, t2, t3), diff );
```

$$\text{eqn1} := \left(\frac{\partial^2}{\partial t_1^2} y_1(t_1, t_2, t_3) \right) + 2 \left(\frac{\partial^2}{\partial t_2 \partial t_1} y_0(t_1, t_2, t_3) \right) + y_1(t_1, t_2, t_3) - \left(\frac{\partial}{\partial t_1} y_0(t_1, t_2, t_3) \right) + \left(\frac{\partial}{\partial t_1} y_0(t_1, t_2, t_3) \right) y_0(t_1, t_2, t_3)^2 = 0$$

再将方程的通解形式代入, 并进行整理:

```
> y0 := (t1, t2, t3) -> A(t2, t3)*sin(t1 + B(t2, t3));
```

$$y_0(t_1, t_2, t_3) \rightarrow A(t_2, t_3) \sin(t_1 + B(t_2, t_3))$$

```
> combine( eqn1, trig );
```

然后将久期项合并起来:

```
> eqn1 := collect( %, [ sin( t1 + B(t2, t3) ),
```

$$\text{eqn1} := -2 A(t_2, t_3) \sin(t_1 + B(t_2, t_3)) \left(\frac{\partial}{\partial t_2} B(t_2, t_3) \right) + \left(2 \left(\frac{\partial}{\partial t_2} A(t_2, t_3) \right) - A(t_2, t_3) + \frac{1}{4} A(t_2, t_3)^3 \right) \cos(t_1 + B(t_2, t_3)) + \left(\frac{\partial^2}{\partial t_1^2} y_1(t_1, t_2, t_3) \right) - \frac{1}{4} A(t_2, t_3)^3 \cos(3 t_1 + 3 B(t_2, t_3)) + y_1(t_1, t_2, t_3) = 0$$

这里, 久期项是 $\sin(t_1 + B(t_2, t_3))$ 和 $\cos(t_1 + B(t_2, t_3))$ 。再令它们等于 0, 就得到下面的方程组:

```
> restriction := {
    coeff( lhs(eqn1), sin( t1+B(t2,t3) ) ) = 0,
    coeff( lhs(eqn1), cos( t1+B(t2,t3) ) ) = 0 };
restriction := {
    2  $\left(\frac{\partial}{\partial t_2} A(t_2, t_3)\right) - A(t_2, t_3) + \frac{1}{4} A(t_2, t_3)^3 = 0, -2 A(t_2, t_3) \left(\frac{\partial}{\partial t_2} B(t_2, t_3)\right) = 0$ 
```

虽然表面上看起来是偏微分方程组，但由于方程不相耦合；而且，从第一个方程中可以直接解出 A 关于 t_2 的表达式来，实际上，它等价于一个常微分方程：

```
> 2*diff(F(t), t) - F(t) + 1/4*F(t)^3 = 0;
    2  $\left(\frac{\partial}{\partial t} F(t)\right) - F(t) + \frac{1}{4} F(t)^3 = 0$ 
> dsolve( %, F(t) );
     $F(t) = 2 \frac{\sqrt{(e^t + 4\_C1) e^t}}{e^t + 4\_C1}, F(t) = -2 \frac{\sqrt{(e^t + 4\_C1) e^t}}{e^t + 4\_C1}$ 
```

因为初值为负，我们不妨取 A 为这样的形式：

$$A(t_2, t_3) = -\frac{2}{\sqrt{1 + C(t_3)e^{-t_2}}}$$

对于 B，通过第二个方程，用同样的方法可以得到它仅仅是 t_3 的函数。在方程 eqn1 中除去久期项，得到如下形式的方程：

```
> subsop(1=0, 2=0, lhs(eqn1)) = 0;
     $\left(\frac{\partial^2}{\partial t_1^2} y_1(t_1, t_2, t_3)\right) - \frac{1}{4} A(t_2, t_3)^3 \cos(3 t_1 + 3 B(t_2, t_3)) + y_1(t_1, t_2, t_3) = 0$ 
```

对于这个方程，我们略去它的齐次解，而选择如下形式的一个特解：

```
> y1 := (t1, t2, t3) -> 1/32 * A(t2, t3)^3
    * sin( 3*t1 + 3*B(t2, t3) + 3/2*Pi );
     $y_1 = (t_1, t_2, t_3) \rightarrow \frac{1}{32} A(t_2, t_3)^3 \sin\left(3 t_1 + 3 B(t_2, t_3) + \frac{3}{2} \pi\right)$ 
> %%;
    0 = 0
```

接下来，我们需要将 y_1 代入到方程 eqn2 中去。首先，将 eqn2 转化成微分形式。为此，需要取消 y_0, y_1 的赋值：

```
> y0 := 'y0': y1 := 'y1':
> eqn2 := convert( eqn2(t1, t2, t3), diff );
```

再代入 y_0 和 y_1 ，并且引入 B 只是 t_2 的函数的条件：

```
> y0 := (t1, t2, t3) -> A(t2, t3) * sin(t1 + B(t3));
     $y_0 = (t_1, t_2, t_3) \rightarrow A(t_2, t_3) \sin(t_1 + B(t_3))$ 
> y1 := (t1, t2, t3) -> 1/32 * A(t2, t3)^3
    * sin( 3*t1 + 3*B(t3) + 3/2*Pi );
     $y_1 = (t_1, t_2, t_3) \rightarrow \frac{1}{32} A(t_2, t_3)^3 \sin\left(3 t_1 + 3 B(t_3) + \frac{3}{2} \pi\right)$ 
```


整理方程 eqn2, 并且另其中的久期项为 0:

```
[> combine( eqn2, trig );
> eqn2 := collect(%, [ sin(t1+B(t3)), cos(t1+B(t3)) ]):
> conditions := { coeff(lhs(eqn2), sin(t1+B(t3))) = 0,
                  coeff(lhs(eqn2), cos(t1+B(t3))) = 0 };
conditions := { -1/128 A(t2, t3)^5 + (d^2 A(t2, t3)/dt2^2) - (d A(t2, t3)/dt2)
                + 3/4 A(t2, t3)^2 (d A(t2, t3)/dt2) - 2 A(t2, t3) (d B(t3)/dt3) = 0,
                2 (d A(t2, t3)/dt3) = 0 }
```

其中, 第二个条件表示 $A(t_2, t_3)$ 和 t_3 无关, 也就是说 $C(t_3)$ 为一常数。从前面 eqn1 的久

期项结果中, 我们还可以得到关于 $\frac{\partial^2 A(t_2, t_3)}{\partial t_2^2}$ 的条件:

```
[> restriction[2];
                2 (d A(t2, t3)/dt2) - A(t2, t3) + 1/4 A(t2, t3)^3 = 0
> diff( %, t2 );
                2 (d^2 A(t2, t3)/dt2^2) - (d A(t2, t3)/dt2) + 3/4 A(t2, t3)^2 (d A(t2, t3)/dt2) = 0
```

注意, 由于 restriction 和 conditions 都是集合类型的符合数据对象, 元素之间没有固定的先后次序, 要视环境和具体情况而定。所以这里对它们取的指标可能与读者的运行结果不同, 请读者根据自己的结果取相应的元素指标。

利用这两个条件, 可以将 condition[2]进行简化:

```
[> simplify( conditions[2], { %, %% },
            convert( [ D[1,1](A)(t2, t3), D[1](A)(t2, t3),
                      A(t2, t3) ], diff ) );
                -7/128 A(t2, t3)^5 + (-2 (d B(t3)/dt3) - 1/4) A(t2, t3) + 1/4 A(t2, t3)^3 = 0
```

利用库函数 isolate, 可以将 B 的微分式分离出来:

```
[> readlib(isolate)( %, diff(B(t3), t3) );
                d B(t3)/dt3 = -1/2 (7/128 A(t2, t3)^5 - 1/4 A(t2, t3)^3) / A(t2, t3) - 1/8
> simplify(%);
                d B(t3)/dt3 = -7/256 A(t2, t3)^4 + 1/8 A(t2, t3)^2 - 1/8
```

由于 A 和 t_3 无关, 我们很容易得到 B 的表达式:

$$B(t_3) = -\frac{1}{8} \left[1 - A(t_2, t_3)^2 + \frac{7}{32} A(t_2, t_3)^4 \right] t_3 + B_0$$

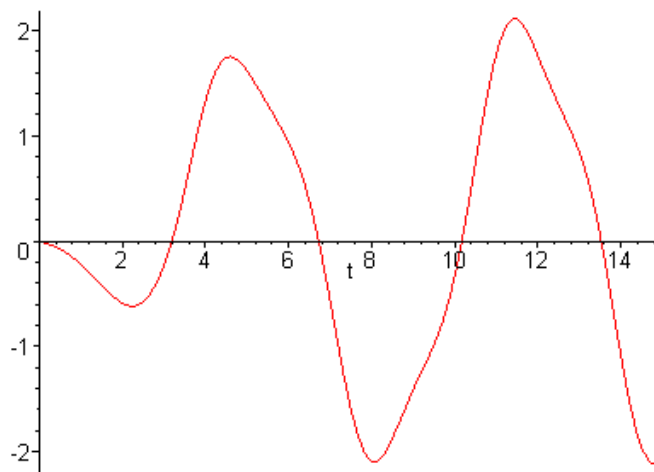
其中 B_0 是一个常数。严格地说，这里得到的 B 的表达式和前面的假设有一定的矛盾。前面曾经假设 B 与 t_2 无关，但这里， B 却和 A 有关，而 A 又是 t_2 的函数。实际上，从 A 的表达式可以看出， A 对于 t 变化很慢，对于较小的 t 来说， A 可以看成是一个常数。如果读者需要对此作深入了解，可以查阅多尺度方法的相关文献。

为了和数值计算结果以及庞加莱方法进行比较，我们再用这里得到的结果计算 van der Pol 方程在初始条件 $y(0)=0$, $y'(0)=-0.1$ 下的数值解：

```
> y := y0 + e*y1;
      y := y0 + y1 ε
> y0 := t -> A(t)*sin(t+B(t));
      y0 := t -> A(t) sin(t + B(t))
> y1 := t -> -1/32*A(t)^3*cos(3*t+3*B(t));
      y1 := t -> -1/32 A(t)^3 cos(3 t + 3 B(t))
> A := t -> -2/(1+c*exp(-e*t));
      A := t -> -2 -----
                      1
                    1 + c e(-ε t)
> B := t -> -1/8*(1 - A(t)^2 + 7/32*A(t)^4)*e^2*t + b;
      B := t -> -1/8 (1 - A(t)^2 + 7/32 A(t)^4) ε2 t + b
```

现在，初值问题就成为了代定常数 b 和 c 的代数方程组问题了，我们用 `fsolve` 来解决：

```
> fsolve( { y(0)=0, D(y)(0)=-0.1 }, {b, c},
      {b=-0.1..0.1} );
      {b = .0004073638406, c = 16.51715658}
> assign(%);
> plot( y(t), t=0..15 );
```



从结果图形可以看出，多尺度法在解趋向于极限环的过程中是和方程的真解符合得相当好的。

7.7 偏微分方程简介

7.7.1 偏微分方程解析解

在 Maple V Release 5 以上的版本中有一个偏微分方程工具包 PDEtools，其中包括求偏微分方程解析解的求解工具（**partial differential equation solver**）——pdsolve。给定一个偏微分方程，pdsolve 的目标就是求出它的解析解来。它对于方程的具体类型、自变量数、和方程阶数都没有具体的要求。但是，在 Maple V Release 5 中，这一工具还有待完善，它目前只能用于求解包含单个未知函数的单个偏微分方程。

函数 pdsolve 可以识别出用通用方法可以解决的标准形式的偏微分方程，如果方程非标准形式，pdsolve 会试图用分离变量等方法将它转化成标准形式再进行求解。如果求解成功，pdsolve 将得到以下几种可能的结果：

- ◇ 方程的通解；
- ◇ 拟通解（包含有任意函数，但不足以构造出通解）；
- ◇ 一些解耦的常微分方程的集合

在最好的情况下，pdsolve 将返回方程的通解，例如：

```
> PDE := x*diff(f(x,y),y)-y*diff(f(x,y),x) = 0;
      PDE := x \left( \frac{\partial}{\partial y} f(x,y) \right) - y \left( \frac{\partial}{\partial x} f(x,y) \right) = 0
> pdsolve(PDE);
      f(x,y) = _F1(x^2+y^2)
```

结果中用到了一个任意函数 _F1，pdsolve 的结果中用到的所有任意函数都以 _F 开头，并在后面附加一个不同的数字，以加以区别。

我们再来看一个第一类偏微分方程的例子：

```
> PDE := x * diff( f(x,y), y ) - diff( f(x,y), x ) =
      f(x,y)^2 * g(x) / h(y);
      PDE := x \left( \frac{\partial}{\partial y} f(x,y) \right) - \left( \frac{\partial}{\partial x} f(x,y) \right) = \frac{f(x,y)^2 g(x)}{h(y)}
> ans := pdsolve(PDE);
      ans := f(x,y) = \frac{1}{\int \frac{g(_a)}{h\left(-\frac{1}{2}_a^2 + y + \frac{1}{2}x^2\right)} d_a + _F1\left(y + \frac{1}{2}x^2\right)}
```

方程的解也是用显式表达式给出的，如果需要检验方程的结果，可以调用另一个函数 pdetest。在结果正确时，pdetest 的返回值为 0；如果结果可能有误，将返回一个代数表达式（由结果代入方程中化简而得）。

```
[> pdetest( ans, PDE );
0
```

如果 Maple 不能找到最一般形式的通解,但还是有结果,它会将结果用函数 `PDESolStruc` 的结构给出,显示成带有 `&where` 的形式。该函数的第一个参数是待求的未知函数的表达形式,其中包括一些单变量的函数;第二个参数中是这些单变量函数所满足的常微分方程组。这样形式给出的不是方程的通解,而是具有一定形式的特解。

例如,我们用 `pdsolve` 求解球坐标下的拉普拉斯方程:

```
[> PDE := Diff( r^2 * diff( F(r,t,p), r ), r )
+ 1/sin(t) * Diff( sin(t)*diff( F(r,t,p), t ), t )
+ 1/sin(t)^2 * diff( diff( F(r,t,p), p ), p ) = 0;


$$PDE = \left( \frac{\partial}{\partial r} r^2 \left( \frac{\partial}{\partial r} F(r, t, p) \right) \right) + \frac{\frac{\partial}{\partial t} \sin(t) \left( \frac{\partial}{\partial t} F(r, t, p) \right)}{\sin(t)} + \frac{\frac{\partial^2}{\partial p^2} F(r, t, p)}{\sin(t)^2} = 0$$


[> ans := pdsolve(PDE);

ans := (F(r, t, p) = _F1(r) _F2(t) _F3(p)) &where  $\left[ \begin{array}{l} \frac{\partial^2}{\partial t^2} \_F2(t) = -\_F2(t) \_c1 - \frac{\left( \frac{\partial}{\partial t} \_F2(t) \right) \cos(t)}{\sin(t)} - \frac{\_c3 \_F2(t)}{\sin(t)^2}, \\ \frac{\partial^2}{\partial r^2} \_F1(r) = \frac{\_F1(r) \_c1}{r^2} - 2 \frac{\frac{\partial}{\partial r} \_F1(r)}{r}, \frac{\partial^2}{\partial p^2} \_F3(p) = \_c3 \_F3(p) \end{array} \right]$ 
```

Maple 之所以不直接给出显式的结果,是为了让用户清楚地看到特解所具有的形式。我们可以用 `PDEtools` 中的 `build` 函数从这样的结果中获得显式的解:(由于结果过于繁琐,此处予以略去)

```
[> with(PDEtools):
[> build(ans):
```

对于这种形式的结果,也可以用 `pdetest` 进行检验:

```
[> pdetest( ans, PDE );
0
```

如果 Maple 求解失败,将返回符号常数 `NULL`。

实际上,由于偏微分方程的复杂性,大多数情况下这样的盲目求解常常会得不到结果。我们可以通过附加参数提供给 Maple 一些求解的信息,以得到需要的结果。

函数 `pdsolve` 完整的调用格式是这样的:

pdsolve(PDE, f, HINT = ..., INTEGRATE, build)

其中, `PDE` 是待求的偏微分方程, `f` 是未知函数,其余的可选参数在下面一一进行介绍。

- **HINT = ...** 用户输入的提示信息

可以有以下几种形式:

- ◇ $\text{HINT} = '+'$: 提示 pdsolve 用和式的形式分离变量, 求解偏微分方程;
- ◇ $\text{HINT} = '*'$: 提示 pdsolve 用积的形式分离变量, 求解偏微分方程;
- ◇ $\text{HINT} =$ 代数表达式: 提示 pdsolve 寻找表达式所制定的形式的解, 例如 $\text{HINT} = f_1(x) / f_2(y) f_2(z)$, 也可以包括多变量函数形式, 如 $\text{HINT} = f_1(x, y) f_2(y, z)$;
- ◇ $\text{HINT} = \text{strip}$: 仅用于一阶偏微分方程, 提示用特征线 (characteristic strip) 求解。
- INTEGRATE 使 pdsolve 自动给出常微分方程的积分结果
- build 使 pdsolve 给出显式的表达式, 无论是否通解 (相当于应用 build 函数的结果)

我们来看一个二阶偏微分方程的例子:

```
> PDE := S(x, y) * diff( S(x, y), y, x) + diff( S(x,
y), x) * diff( S(x, y), y ) = 1;

$$PDE := S(x, y) \left( \frac{\partial^2}{\partial y \partial x} S(x, y) \right) + \left( \frac{\partial}{\partial x} S(x, y) \right) \left( \frac{\partial}{\partial y} S(x, y) \right) = 1$$

> struc := pdsolve( PDE, HINT = f(x) * g(y) );

$$\text{struc} := (S(x, y) = f(x) g(y)) \&\text{where} \left[ \left\{ \frac{\partial}{\partial x} f(x) = \frac{-c_1}{f(x)}, \frac{\partial}{\partial y} g(y) = \frac{1}{2} \frac{1}{g(y) - c_1} \right\} \right]$$

```

我们用 build 把它写成显式的形式:

```
> build(struc);

$$S(x, y) = - \frac{\sqrt{2 - c_1 x + -C1} \sqrt{-c_1 y + -C2 - c_1^2}}{-c_1}$$

```

从这个结果, 我们得到启示, 是不是可以把通解写成根式的形式呢:

```
> pdsolve(PDE, HINT=P(x, y)^(1/2));

$$S(x, y) = \sqrt{-F2(x) + -F1(y) + 2xy}$$

```

实践证明这是正确的, Maple 求解成功, 直接给出了显式的通解表达式。

对于非线性一阶偏微分方程, 可以利用特征线方法求解, 如下例:

```
> PDE := diff(f(x, y, z), x) + diff(f(x, y, z), y)^2
= f(x, y, z) + z;

$$PDE := \left( \frac{\partial}{\partial x} f(x, y, z) \right) + \left( \frac{\partial}{\partial y} f(x, y, z) \right)^2 = f(x, y, z) + z$$

> pdsolve(PDE, HINT=strip);

$$\left( \left( \frac{\partial}{\partial x} f(x, y, z) \right) + \left( \frac{\partial}{\partial y} f(x, y, z) \right)^2 - f(x, y, z) - z = 0 \right) \&\text{where} \left[ \left\{ \left( \begin{aligned} x(_s) &= -C1 + _s, z(_s) = -C4, _p1(_s) = e^{-s} - C5, \\ f(_s) &= e^{(2-s)} - C3 - C4 + C4 e^{(2-s)} - C5 e^{(2-s)} + e^{-s} - C5, \\ _p2(_s) &= e^{-s} - C6, y(_s) = -C2 + 2 e^{-s} - C6 - 2 - C6 \end{aligned} \right\} \right. \right. \\ \left. \left. \&\text{and} \left( \left\{ _p1 = \frac{\partial}{\partial x} f(x, y, z), _p2 = \frac{\partial}{\partial y} f(x, y, z) \right\} \right) \right] \right]$$

```

在 PDEtools 工具包中, 也有函数可以直接求出一阶偏微分方程的特征线来, 调用 charstrip 或者 splitstrip。函数 charstrip 返回的是特征线的常微分方程组。如下例:

```
> PDE := x * diff( f(x, y, z), z ) - f(x, y, z) + y^2 *
      diff( f(x, y, z), y ) = 0;
      
$$PDE := x \left( \frac{\partial}{\partial z} f(x, y, z) \right) - f(x, y, z) + y^2 \left( \frac{\partial}{\partial y} f(x, y, z) \right) = 0$$

> sys0 := charstrip(PDE, f(x, y, z));
      
$$\text{sys0} := \left\{ \frac{\partial}{\partial s} f(s) = f(s), \frac{\partial}{\partial s} y(s) = y(s)^2, \frac{\partial}{\partial s} z(s) = x(s), \frac{\partial}{\partial s} x(s) = 0 \right\}$$

```

特征线方程可以用常微分方程求解函数 dsolve 求解而得到显式的表达式:

```
> dsolve(sys0, {f(s), x(s), y(s), z(s)}, explicit);
      
$$\{f(s) = \_C1 e^{-s}, x(s) = \_C3, y(s) = \frac{1}{-s + \_C4}, z(s) = \_C3 s + \_C2\}$$

```

使用函数 splitstrip 可以获得几组相互之间解耦的常微分方程组——是以子集的形式给出的:

```
> sys1 := splitstrip( PDE, f(x, y, z) );
      
$$\text{sys1} := \left\{ \left\{ \frac{\partial}{\partial s} z(s) = x(s), \frac{\partial}{\partial s} x(s) = 0 \right\}, \left\{ \frac{\partial}{\partial s} y(s) = y(s)^2 \right\}, \left\{ \frac{\partial}{\partial s} f(s) = f(s) \right\} \right\}$$

```

对于这一组常微分方程组, 需要借助 map 的方法进行求解:

```
> map( u -> dsolve( u, indets(u, Function) ), sys1 );
      
$$\{ \{x(s) = \_C1, z(s) = \_s \_C1 + \_C2\}, y(s) = \frac{1}{-s + \_C1}, f(s) = \_C1 e^{-s} \}$$

```

7.7.2 偏微分方程的形式转换

Maple 的 PDEtools 工具包中还有一个函数——mapde, 它可以进行偏微分方程的形式转换。给定一个偏微分方程, mapde 的目的是将该偏微分方程转换成其他的形式, 这有时会使方程更易于求解。

函数 mapde 的标准调用格式是这样的:

mapde (PDE, into, f)

其中 PDE 是有待转换的偏微分方程, into 是转换的形式说明 (见表 7.4), f 是偏微分方程的未知函数 (可选参数, 只需在必要时提供)。

表 7.4 偏微分方程转换形式

参数 into	转换成的方程形式
noF	未知函数的隐式表达式
homo *	各向同性偏微分方程
ccoeff *	高阶导数项系数为常数的偏微分方程
canom	只含有一个混合导数的经典形式
canop	只含单纯导数 (没有混合导数) 的经典形式

注: 1. 带*号的转换在 Release 5 中尚未实现;

2. canom 和 canop 仅对二阶线性常微分方程有效。对于常系数问题, 总能得到结果, 对于符号系数的情况, 有可能找不到相应的转换。

我们举一个例子来说明它的用法:

```
[> with(PDEtools):
> PDE := diff( f(x, y, z), x )^3 = f(x, y, z)
      * diff( f(x, y, z), y, y ) * diff( f(x, y, z), z);
```

$$PDE = \left(\frac{\partial}{\partial x} f(x, y, z) \right)^3 = f(x, y, z) \left(\frac{\partial^2}{\partial y^2} f(x, y, z) \right) \left(\frac{\partial}{\partial z} f(x, y, z) \right)$$

首先, 我们试着用 pdsolve 直接求解:

```
[> pdsolve(PDE);
```

Maple 什么结果也没有给, 直接解法失败了。我们再试着用 mapde 转换它的形式, 然后再进行求解:

```
[> PDE1 := mapde(PDE, noF);
```

$$PDE1 = \left(\left(\frac{\partial}{\partial x} _F1(x, y, z) \right)^3 - \left(\frac{\partial}{\partial z} _F1(x, y, z) \right) \left(\frac{\partial^2}{\partial y^2} _F1(x, y, z) \right) - \left(\frac{\partial}{\partial z} _F1(x, y, z) \right) \left(\frac{\partial}{\partial y} _F1(x, y, z) \right)^2 = 0 \right) \&\text{where } \{ _F1(x, y, z) = \ln(f(x, y, z)) \}$$

```
[> pdsolve( op(1, PDE1) );
( \_F1(x, y, z) = \_F2(x) + \_F3(y) + \_F4(z) ) &where
[ \left\{ \frac{\partial}{\partial x} \_F2(x) = \_c1, \frac{\partial}{\partial z} \_F4(z) = \_c3, \frac{\partial^2}{\partial y^2} \_F3(y) = \frac{\_c1^3}{\_c3} - \left( \frac{\partial}{\partial y} \_F3(y) \right)^2 \right\} ]
```

```
[> build(%);
```

$$_F1(x, y, z) = _c1 x + _C1 + _c1 \sqrt{_c1 _c3} \int \text{RootOf}(y _c1 \sqrt{_c1 _c3} - _c3 \operatorname{arctanh}(_Z) + _C2 _c1 \sqrt{_c1 _c3}) dy / _c3 + _C3 + _c3 z + _C4$$

问题得到了相当好的解决, 最后的结果的 e 指数就是原方程的解。

7.7.3 偏微分方程解的图形绘制

利用 PDEtools 工具包中的函数 PDEplot 可以进行一阶偏微分方程解曲线(曲面)的绘制。该函数的调用格式如下:

PDEplot (PDE, inits, srangle, options)

其中的参数意义如下:

- ✧ PDE: 一阶偏微分方程, 可以是线性或者非线性的, 但必须只含有一个 n 元未知函数;
- ✧ inits 是一个 $n+1$ 个表达式或者方程组成的有序表, 它们表示初值, 用参数方式给出, 包括 $n-1$ 个参数, 在 $n=3$ 时(曲面), 一般用 s 和 t 作为参数变量;
- ✧ srangle 是初值中参数的取值范围;
- ✧ options 是辅助的参数, 用“参数名=辅助参数”的形式给出。

我们用例子来说明它的用法：

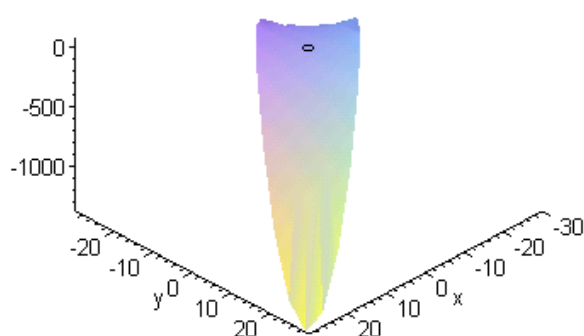
```
[> with(PDEtools):
> diff(u(x,y), x) * diff(u(x,y), y) - x*y + u(x, y)=0;

$$\left(\frac{\partial}{\partial x}u(x,y)\right)\left(\frac{\partial}{\partial y}u(x,y)\right) - xy + u(x,y) = 0$$

```

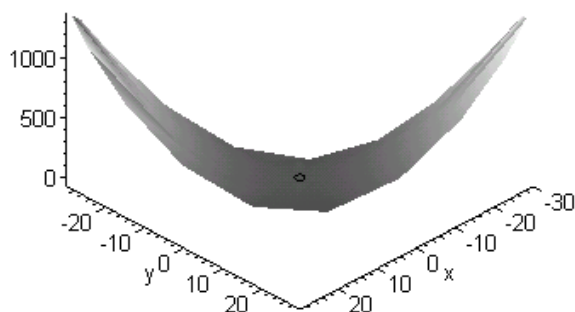
用 x-y 平面上的单位圆作为初值曲线，绘制偏微分方程解的积分曲面：

```
> PDEplot(%, [cos(t), sin(t), 0], t=-Pi..Pi,
ic_assumptions = [diff(u(x,y),x) = -cos(t)]);
```



由于该偏微分方程的非线性性，在给定的初始条件下有两个可能的解，取决于初始曲线上的导数值。在 PDEplot 中，可以用附加参数 ic_assumption 给定初始的导数值。可以改变初始的导数值，获得另一个解的曲面：

```
> PDEplot(%%, [cos(t), sin(t), 0], t=-Pi..Pi,
ic_assumptions = [diff(u(x,y),x) = cos(t)]);
```



除了 ic_assumption 之外，函数 PDEplot 还有很多辅助参数，需要用“参数名=参数值”的等式形式给出。各参数的用途请读者参考表 7.5。

表 7.5 PDEplot 的辅助参数

参数名称	参数类型	默认值	参数用途
iterations	整数	1	图形上单个点的计算点数（为提高精度）
stepsize	实数	0.25 (最大值)	每条特征曲线上两个相邻数据点的距离
numsteps	整数区间 或单个整数	[-10, 10]	每条特征曲线上的数据点数（两个方向） 如果给定单个整数，只绘制一个方向
numchar	整数有序表	20	从初始值曲线出发的特征曲线数 最小值为 4
scene	三个表达式的 有序表	前两个未知 函数和自变量	所要绘制的函数（或自变量）
xi	区间	整个解曲面	例如: $x1 = x1_min..x1_max$ 制定绘制的曲面区间
obsrange	布尔值	true	是否对于不可见曲面不予计算 true: 不予计算
method	名称	rkf45	同数值求解常微分方程, dsolve 的 method
animate	true, false, 或 only	n = 2 时 true n > 2 时 only	是否用动画显示解的超曲面 only 表示只显示初值曲面的动画
ic_assumptions	等式的有序表		给定非线性偏微分方程的附加初值假设
basechar	true, false, 或 only	false	是否显示基特征曲线(初始条件演化过程在 x-y 平面上的投影) only 只用于 animate = false 时
colour	plot[color]中的颜色名称		以指定颜色绘制曲面
	COLOUR(HUE, 实数)		在制定数值范围内用不同色调绘制曲面
	COLOUR(RGB, 实数, 实数, 实数)		在三个坐标上, 分别在各自的制定数值范围内 着色
	二变量函数或表达式		数据点的数值作为函数或表达式的自变量, 更 具在[0, 1]之间的不同返回值来着色
	3 个二变量函数或表达式的有 序表		数据点的数值作为函数或表达式的自变量, 更 具三个坐标方向在[0, 1]之间的不同返回值来着色
initcolour	基本同 colour, 表达式和函数为 单变量		设置初值曲线的色调

另外, 大部分三维图形绘制的参数在这里也可以使用, 例如 style, orientation 等等, 请读者参阅第六章。

这里通过一个例子来简单说明 PDEplot 的附加参数的试用, 该例子所得到的图形就是 Maple V Release 5 启动是的封面图形。它是这个偏微分方程的解。

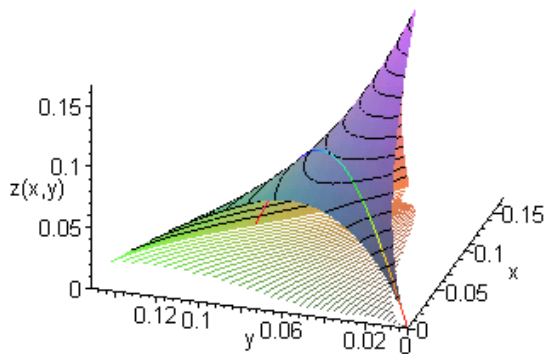
$$\begin{aligned}
 & \text{PDE} := (y^2 + z(x,y)^2 + x^2) * \text{diff}(z(x,y), x) \\
 & \quad - 2*x*y*\text{diff}(z(x,y), y) - 2*z(x,y)*x = 0; \\
 & \text{PDE} = (y^2 + z(x,y)^2 + x^2) \left(\frac{\partial}{\partial x} z(x,y) \right) - 2xy \left(\frac{\partial}{\partial y} z(x,y) \right) - 2z(x,y)x = 0
 \end{aligned}$$

下面绘制这个偏微分方程的解 $z(x, y)$ 的图形。初值曲线选择参数方程:

$$\begin{cases} x = t \\ y = t \\ z = \frac{1}{10} \sin(10\pi t) \end{cases} \quad (0 \leq t \leq 0.1)$$

所确定的空间半圆周；特征曲线采用 40 条；视角方向 $[-163^\circ, 56^\circ]$ ；绘制基特征曲线；每条特征曲线上计算 20 个点；特征曲线步长 0.15；初值曲线色调根据表达式 $t \cos t$ 着色；不生成动画；用带等高线的面图绘制。

```
> PDEplot(PDE, z(x, y), [t, t, sin(Pi*t/0.1)/10],
  t=0..0.1, numchar=40, orientation=[-163,56],
  basechar=true, numsteps=[20,20], stepsize=.15,
  initcolour=cos(t)*t, animate=false,
  style=PATCHCONTOUR);
```



7.7.4 李对称 (Lie Symmetry) 工具包

李对称方法是研究微分方程的利器之一，Maple 中有李对称方法的专门工具包 `liesymm`。在这一小节中，将利用这一工具包，找出 Korteweg-de Vries 方程

$$u_t + uu_x + u_{xxx} = 0$$

的李对称基。

首先，载入该工具包，并且定义方程：

```
> with(liesymm):
Warning, new definition for close
> KdV_eqn := Diff(u(t, x), t) + u(t, x)*Diff(u(t, x), x)
  + Diff(u(t, x), x$3) = 0;
```

$$KdV_eqn = \left(\frac{\partial}{\partial t} u(t, x) \right) + u(t, x) \left(\frac{\partial}{\partial x} u(t, x) \right) + \left(\frac{\partial^3}{\partial x^3} u(t, x) \right) = 0$$

在这里，先简单介绍一下李对称的基本思想。对于一个偏微分方程：

$$\omega(t, x, u, u_t, u_x, u_{tt}, u_{tx}, u_{xx}, \dots) = 0$$

而言，所谓点对称 (Lie point symmetry)，是指这样的映射：

$$t \rightarrow \bar{t}(t, x, u), \quad x \rightarrow \bar{x}(t, x, u), \quad u \rightarrow \bar{u}(t, x, u),$$

它们满足原来的方程。通常考虑的是单参数的点对称，这时，相应的无限小映射为：

$$t \rightarrow t + \varepsilon \tau, \quad x \rightarrow x + \varepsilon \xi, \quad u \rightarrow u + \varepsilon \eta.$$

在这样的映射下，如果对于微分算子 $X = \tau \partial_t + \xi \partial_x + \eta \partial_u + \dots$ ，满足 $X\omega = 0$ ，则原来的偏

微分方程保持不变。这一性质就引出了一组各项同性的线性偏微分方程，未知函数是 τ, ξ, η 。这个方程组称为决定系统 (determining system)。在 Maple 中，可以用函数 `determine` 得到它。

```
[> eqns1 := determine( KdV_eqn, V, u(t, x), w );
```

其中， τ, ξ, η 分别用 $V1, V2, V3$ 表示了。

在 Maple 中也提供了化简决定系统的函数 `autosimp`，而且，在方程比较好的情况下，还能直接给出解析解来。

```
[> eqns2 := autosimp( eqns1 );
eqns2 := { } &where { V1_2(t) = C9 - 3/2 t^2 C7 + 3 C6 t, V2_3(t) = t C3 + C4,
V3_3(t) = t C7 + C8, t C7 - 2 C6 - C8 = 0,
V2_1(t, x) = x (-t C7 + C6) + t C3 + C4, V3(t, x, u) = u (t C7 + C8) - x C7 + C3,
V3_1(t, x) = t C7 + C8, V2(t, x, u) = x (-t C7 + C6) + t C3 + C4,
V3_2(t, x) = -x C7 + C3, V3_4(t) = -C7, V3_5(t) = C3, V2_2(t) = -t C7 + C6,
V1(t, x, u) = C9 - 3/2 t^2 C7 + 3 C6 t }
```

正如我们所看到的，Maple 已经解出了这个方程组，只不过中间还含有一个约束方程： $t C7 - 2C6 - C8 = 0$ 。为了使所有的 t 都满足这一方程，只能取 $C7 = 0, C8 = -2C6$ 。我们将它代入到结果中去：

```
[> eqns := subs( C7 = 0, C8 = -2*C6, eqns2 );
[> select( has, op(2, eqns), {V1, V2, V3} );
{ V2(t, x, u) = x C6 + t C3 + C4, V1(t, x, u) = C9 + 3 C6 t,
V3(t, x, u) = -2 u C6 + C3 }
```

至此，我们找到了决定系统的通解：

$$\tau = C_9 + 3C_6 t, \quad \xi = C_6 x + C_3 t + C_4, \quad \eta = -2C_6 u + C_3$$

其中， C_3, C_4, C_6, C_9 是任意常数。将它们代入到前面的微分算子 X 中去，可以得到四个对称变换基。所以，Korteweg-de Vries 方程的解对于一些四个基本变换都是对称的：

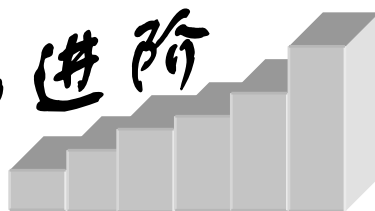
$$\partial_t, \quad \partial_x, \quad t\partial_x + \partial_u, \quad x\partial_x + 3t\partial_t - 2u\partial_u$$

这也就是说，对于该方程的任意一个解 $u = f(t, x)$ ，对应地，对于任意实数 ε ，以下的函数也是方程的解。

$$\begin{aligned}u_1 &= f(t - \varepsilon, x) \\u_2 &= f(t, x - \varepsilon) \\u_3 &= f(t, x - \varepsilon t) + \varepsilon \\u_4 &= e^{-2\varepsilon} f(e^{-3\varepsilon} t, e^{-\varepsilon} x)\end{aligned}$$

Maple 的 `liesymm` 工具包中还有一些函数，由于专业性较强，这里不再一一介绍。如果需要，可以查阅相应的在线帮助。

起步与进阶



第

编程进阶

八

章

本章将进一步介绍 Maple 语言的编程。这一章的内容是在第五章的基础之上的一个提高，目的在于解决实际编程中可能遇到的问题。通过这一章的学习，一方面可以更加熟练地编写实用的程序，另一方面可以更为透彻地掌握用 Maple 解决实际问题的技巧。

本章具体包括以下内容：

- 🕒 返回子程序的子程序
- 🕒 子程序作为参数的输入
- 🕒 子程序参数序列的表示
- 🕒 交互式输入方法
- 🕒 扩展 Maple 功能
- 🕒 在 Maple 中加入工具包

在用 Maple 进行越来越复杂的实际问题编程中，你一定会发现，单单使用在编程初步一章中介绍的方法是不够的。为了解决这些问题，在这一章中，将介绍较为高级的编程技术。这些技术也都是建立在基本的 Maple 机制之上的，相信你通过前面几章的学习，已经具备了这一章所需的基础知识，可以理解这一章的内容了。

首先，我们将接触到一个在介绍嵌套子程序时遗漏的问题，就是将子程序作为返回值的问题。在这样的程序当中，局部变量可以在子程序运行完之后仍然存在。然后，我们会涉及到交互式输入的问题。这是解决实际问题乃至推广我们编制的程序所必不可少的。接着，将介绍扩展 Maple 的现有功能的方法。最后，我们将把自己编制的程序制作成工具包，以使之可以像 Maple 的自带工具包那样方便的使用。

8.1 返回子程序的子程序

在所有的子程序中，编写返回值为一个子程序的程序将遇到的困难也许是最多的了。编写这样的程序，需要对 Maple 的各种变量的求值规则、有效语句有透彻的理解。

Maple 一些内部子程序就有这样的机制，比如随机函数 `rand`，返回的就是一个子程序，它会在给定的范围内随机地取值。你也许想在你自己的子程序中应用这样的机制，那么，让我们从最简单的例子开始吧。

8.1.1 牛顿迭代法

牛顿迭代法 (Newton Iteration) 是用来求解非线性方程数值解的常用方法之一。首先，我们选择一个接近于精确解的初值点。接着，在曲线上该初值点处作切线，求出切线与 x 轴的焦点。对于一大部分函数，这个交点比初值点要更接近精确解。所以，用这个点代替初值点，重复上面的过程，就可以得到更精确的点。

我们用数学语言来描述上面的过程。对于方程 $f(x) = 0$ ，选定初值 x_0 ；然后利用递推公式：
$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$
，来逐步获得精确解。对于这个算法，我们可以用多种不同的途径加以实现。在下面的程序 `MakeIteration` 中，我们根据已有的表达式 `expr`，来获得递推函数。其中，我们用到了 Maple 命令 `unapply`，从表达式中得到相应的函数，作为子程序的返回值予以返回。

```
> MakeIteration := proc( expr::algebraic, x::name )
    local iteration;
    iteration := x - expr/diff(expr, x);
    unapply(iteration, x);
end;
```

我们试着用它来得到方程 $x - 2\sqrt{x} = 0$ 的牛顿迭代函数：

```
> expr := x - 2*sqrt(x);
      expr := x - 2*sqrt(x)
```

```
[> Newton := MakeIteration( expr, x );
```

$$Newton := x \rightarrow x - \frac{x - 2\sqrt{x}}{1 - \frac{1}{\sqrt{x}}}$$

然后，我们利用已经得到的迭代递推函数 `Newton`，可以用来求解这个超越方程。首先选定初值点 $x = 2.0$ ，（为了使 Maple 对其进行浮点运算，需要设置 x 为浮点数）。

```
[> x0 := 3.0;
```

$$x0 := 3.0$$

```
[> to 4 do x0 := Newton(x0); od;
```

$$x0 := 4.098076213$$

$$x0 := 4.000579795$$

$$x0 := 4.000000023$$

$$x0 := 4.000000001$$

可以看到，不过进行了 5 次迭代，结果已经相当精确了。

上面的子程序 `MakeIteration` 的输入参数类型是代数表达式，我们也可以编制用函数作为参数的子程序。由于函数的求值具有特殊性，在程序中需要用 `eval` 对其进行完全求值。

```
[> MakeIteration := proc( f::procedure )
    (x->x) - eval(f) / D(eval(f));
end;
```

由于输入参数是一个子程序，就不需要再显式地指出自变量了，我们用 `D` 运算符求它的导函数，并且作运算。这里， $(x \rightarrow x)$ 表示恒等函数 $f(x) = x$ 。从这里，我们也可以知道，在 Maple 中，不仅代数表达式可以进行运算，具有相同自变量的函数也可以相互进行运算。我们利用它来求解超越方程 $x^2 - \cos(x) = 0$ ：

```
[> g := x -> x^2 - cos(x);
```

$$g := x \rightarrow x^2 - \cos(x)$$

```
[> Newton := MakeIteration( g );
```

$$Newton := (x \rightarrow x) - \frac{x \rightarrow x^2 - \cos(x)}{x \rightarrow 2x + \sin(x)}$$

该函数的收敛也相当快，只需要 4 次迭代就可以达到 8 位有效数字的精度：

```
[> x0 := 1.0;
```

$$x0 := 1.0$$

```
[> to 4 do x0 := Newton(x0); od;
```

$$x0 := .8382184099$$

$$x0 := .8242418682$$

$$x0 := .8241323190$$

$$x0 := .8241323123$$

8.1.2 函数的平移

考虑这样一个简单的问题，已知一个函数 f ，我们需要得到另一个函数 g ，满足 $g(x) = f(x + 1)$ 。在数学中，我们称这样的操作为函数的平移。在 Maple 中，我们可以编程实现这样的

操作。它的输入和输出都是子程序。由于程序的表达式简单，可以用箭头操作符简单地定义：

```
[> shift := (f::procedure) -> ( x->f(x+1) );
```

我们用它来将 $\cos(x)$ 向左平移一个单位：

```
[> shift(cos);  
x → cos(x+1)
```

在这个程序中，Maple 的变量寻找机制自动地把内部嵌套子程序中的 f 和外部的参数 f 绑定起来。所以，程序 `shift` 可以如我们所愿地工作。

但是，这个程序仅仅对于单变量函数有效；对于多变量函数，却不能有效地完成平移的操作了。

```
[> h := (x,y) -> x*y;  
h := (x,y) → x y  
[> hh := shift(h);  
hh := x → h(x+1)  
[> hh(x, y);  
Error, (in h) h uses a 2nd argument, y, which is missing
```

它得到的仍然是一个单变量函数 `hh`，当我们用参数 x, y 对其进行调用时，就会发生错误。对于一个不知道参数个数的子程序，我们可以用 `args` 表示它的所有参数组成的序列，从而，`args[2..-1]` 就表示从第 2 个开始的所有参数序列。

```
[> shift := (f::procedure) -> ( x->f(x+1,args[2..-1]) );  
[> hh := shift(h);  
hh := x → h(x+1, args_2 .. _1)  
[> hh(x, y);  
(x+1)y
```

这里，子程序 `hh` 和 `h` 直接相关，（实际上，`hh` 内部调用了 `h`）。如果改变了 `h` 的表达式，那么，`hh` 的形式也会跟着变化：

```
[> h := (x, y, z) -> z^2/x/y;  
h := (x,y,z) →  $\frac{z^2}{xy}$   
[> hh(x, y, z);  
 $\frac{z^2}{(x+1)y}$ 
```

8.2 局部变量的保持

在第五章中，我们已经知道，局部变量的作用域是当前的自程序；而且，对于每一次该子程序的运行，局部变量都是不同的变量。简单来说，每一次调用子程序，都产生一些新的变量；如果你两次调用相同的子程序，那么，第二次所用的局部变量和第一次的局部变量是不同的。

有时候，在退出一个子程序时，局部变量并不消失。比如在子程序中将局部变量作为结果返回了，那么在子程序结束后，这些局部变量仍然存在。对于这些变量，你也许会觉得它们难以捉摸，因为它们可以和全局变量同名，但却是不同的变量；更极端的情况，几次从同一个程序返回的局部变量名字都相同，但都是不同的变量——它们在系统内存中占有不同的位置，修改其中一个的值不会影响到其他的值。

8.2.1 “越界”的局部变量

为了对这种情况有一定的了解，我们首先定义一个返回局部变量的子程序：

```
[> make_a := proc()
      local a;
      a;
    end;
```

现在，我们有了一个产生这样的变量的子程序。那么，怎样检查所产生的变量是否相同呢？也许您还记得集合这种数据结构，它中间的元素都具有唯一性，它会自动删除相同的元素。

```
[> test := { a, a, a };
                                     test := {a}
```

我们用它来检查 make_a 所生成的变量：

```
[> test := test union { make_a() };
                                     test := {a, a}
[> test := test union { 'make_a'()$5 };
                                     test := {a, a, a, a, a, a}
```

我们看到，6次用子程序 make_a 生成了6个同名变量，它们是互不相同的，而且，它们和全局变量 a 也不相同。从这里，我们知道，Maple 并不是只依据变量名来区别变量的。当我们在交互式环境中键入变量 a 时，Maple 认为它是全局变量。所以，我们可从上面的同名变量集合 test 中很容易地找到全局变量。

```
[> member( a, test, num );
                                     true
[> num;
                                     1
```

那么，这些同名变量有什么用呢？它们可以用来写出一些通常用 Maple 很难得到的表达式。举个例子来说，Maple 会自动地把表达式 $a + a$ 化简成 $2a$ ，要得到表达式 $a + a = 2a$ 是一件不容易的事。现在，我们可以利用上面的子程序 make_a 来轻松地写出这样的式子。

```
[> a + make_a() = 2*a;
                                     a + a = 2 a
```

对于 Maple 来说，等号左边的两个 a 是不同的，虽然它们具有相同的变量名；所以，它没有自动地把它化简成 $2a$ 。对于全局变量，可以用变量名直接引用，但是对于另一个用 make_a 得到的变量呢？引用它就需要费一番周折。我们可以通过 remove 命令去掉等式左边的全局变量 a 来得到另一个 a。

```
[> eqn := %;
                                eqn := a + a = 2 a
[> another_a := remove( x->eval(x=a), lhs(eqn) );
                                another_a := a
```

现在，全局变量 `another_a` 指向了那个难以捉摸的 `a`，我们要做的是把那个 `a` 赋成全局变量 `a`，用赋值语句显然是不行的，我们用 `assign` 命令将其赋值。`assign` 命令和赋值语句的不同在于被赋值的变量是作为参数传给 `assign` 的，所以 Maple 会自动地将它求值为它所指的 `a`，这样，就可以为 `a` 赋值了。

```
[> assign( another_a = a );
[> eqn;
                                2 a = 2 a
```

这时，等式两边都是作为全局变量的 `a` 了。我们用 `evalb` 可以检验等式的正确性（虽然这是显然的，但对于复杂的等式，这却是必要的）。

```
[> evalb(%);
                                true
```

在这一节中，我们引入了令人费解的“越界”的局部变量。而实际上，你也许早就已经碰到过这样的变量了，只是那时没有引起你的注意而已。我们曾经接触过 `assume` 命令，它将一个变量赋予另一个有确定范围的变量——新的变量只是在原来的变量后面加上一个“~”。这个具有波浪线的变量就是一个“越界”的局部变量——如果你在交互式环境中输入它（当然需要用一对反向撇号“`”括起来），Maple 将不能识别，因为它误以为你输入的是全局变量。

```
[> assume(b>0);
[> x := b + 1;
                                x := b~ + 1
[> subs( `b~`=c, x );
                                b~ + 1
```

Maple 所做的是把局部变量 `b~` 赋给全局变量 `b`。如果我们将 `b` 另外赋值，`b~` 仍然存在，由它所生成的表达式也仍然存在。

```
[> b := 'b';
                                b := b
[> x;
                                b~ + 1
```

8.2.2 集合的笛卡尔积

将局部变量保持的一个重要应用是在返回的对象是一个子程序时。如果写一个返回子程序的程序，你常常会觉得有必要建立一个变量，它可以保存仅供该子程序使用的信息。这也提供了一个在多个子程序之间交换信息的手段。这一小节中介绍的程序就用到了这个思想。它的输入参数是一个集合的序列，它的输出结果是一个子程序。该子程序每一次产生原集合的笛卡尔积的一个不同的项。具体输出哪一项由保持的局部变量所决定。

所谓集合的笛卡尔积，是指这样的有序表的集合，集合中每一个有序表的第 i 个元素都是原来第 i 个集合中的元素。例如，集合 $\{\alpha, \beta, \gamma\}$ 和 $\{x, y\}$ 的笛卡尔积为：

$$\{\alpha, \beta, \gamma\} \times \{x, y\} = \{[\alpha, x], [\beta, x], [\gamma, x], [\alpha, y], [\beta, y], [\gamma, y]\}$$

当集合增加或者集合中的元素个数增加时，集合的笛卡尔积中的元素个数会急剧增加，如果直接计算所有的元素，会占用大量存储空间。一个解决的方法是生成一个子程序，它每次产生笛卡尔积中的一个新元素，只要连续调用这个子程序，就可以获得笛卡尔积中的所有元素了。

下面的程序是一个简单的实现，它将每次返回笛卡尔积中的下一个元素。我们用数组 a 来计数，以确定下一个元素，例如， $a[1]=2$ ， $a[2]=1$ 就对应着第一个集合中的第 2 个元素和第 2 个集合中的第 1 个元素组成的有序表。

```
[> s := [ {alpha, beta, gamma}, {x, y} ];
      s := [{gamma, beta, alpha}, {y, x}]
> a := array( 1..2, [2, 1] );
      a := [2, 1]
> [ seq( s[j][a[j]], j=1..2 ) ];
      [beta, y]
```

在开始调用程序获得第一个元素前，需要将 a 初始化成 $[1, 1]$ 。

```
[> a := array( [0, 1] );
      a := [0, 1]
```

在下面的程序中，首先检查计数数组 a 是否超过了计数范围 ($nops(s[i])$)，如果未超过，则返回该元素，否则，就设置 a 的该项为 1。当所有的元素都输出完毕时，程序把 a 重新设为 $[0, 1]$ ，并输出 FAIL 作为标志。

```
[> element := proc(s::list(set),
      a::array(1, nonnegint))
    local i, j;
    for i to nops(s) do
      a[i] := a[i] + 1;
      if a[i] <= nops(s[i]) then
        RETURN( [ seq(s[j][a[j]], j=1..nops(s)) ] )
      fi;
      a[i] := 1;
    od;
    a[1] := 0;
    FAIL;
end;
```

利用它，就可以一次求得集合的笛卡尔积的各个元素了：

```
[> element(s, a); element(s, a); element(s, a);
      [gamma, y]
      [beta, y]
      [alpha, y]
```

但是，这样的程序不具有通用性，因为如果我们变更一下原来的基集，那么程序也要被重写。下面，我们试着来写一个更通用的程序，它将返回一个子程序，完成上面的功能。首先，我们用程序的参数生成一个有序表 s ，它的每一项都是一个集合。然后，它初始化我们

作为计数器的数组 a ，当然也定义了每一步返回笛卡尔积的一项的子程序，并把用 s 和 a 作为参数对它的调用作为结果返回。

```
> CartesianProduct := proc()
    local s, a, element;
    s := [args];
    if not type(s, list(set)) then
        ERROR( "expected a sequence of sets, but
received", args );
    fi;
    a := array( [0, 1$(nops(s)-1)] );

    element:=proc(s::list(set),a::array(1,nonnegint))
        local i, j;
        for i to nops(s) do
            a[i] := a[i] + 1;
            if a[i] <= nops( s[i] ) then
                RETURN( [seq(s[j][a[j]], j=1..nops(s))] );
            fi;
            a[i] := 1;
        od;
        a[1] := 0;
        FAIL;
    end;

    proc()
        element(s, a);
    end;
end;
```

同样，我们可以利用它所生成的子程序，获得 $\{\alpha, \beta, \gamma\} \times \{x, y\}$ 的6个元素：

```
> f:=CartesianProduct( {alpha, beta, gamma}, {x, y} );
      f:=proc() element(s,a) end
> to 7 do f() od;

      [γ, x]
      [α, x]
      [β, x]
      [γ, y]
      [α, y]
      [β, y]
      FAIL
```

这个例子中，返回的程序中的子程序 `element` 也是函数 `CartesianProduct` 中的局部变量，我们可以在交互式环境中重新定义另一个 `element`，而它却不会改变。

```
> element := x -> x;
      element := x → x
> f();
      [γ, x]
```

这些例子不仅仅是为了说明局部变量可以越出它的有效域之外，更重要的是，我们可以利用这一特性，编制我们所需要的特殊的函数。

8.3 交互式输入

一般情况下，Maple 程序和外部的数据交换方式主要是通过参数的传递。但某些时候，我们需要编写交互式的程序，比如用 Maple 编写数学教学软件，就需要用户输入结果，再与正确的结果比较；更多的时候，我们可以编写这样的程序，即使对于 Maple 语法不了解的用户，也可以方便地使用。Maple 中最常用的交互式输入命令是 `readline` 和 `readstat`。

8.3.1 从终端读入字符串

`readline` 命令可以从文件或者键盘读入一行字符串，它的格式是 `readline(filename)`，如果需从键盘读入字符串，可以用特殊的文件名 `terminal` 作为参数。`readline` 命令把所读入的文本以字符串形式返回。

```
[> s := readline( terminal );
> Maple V Release 5 起步与进阶
s := "Maple V Release 5 起步与进阶"
```

下面是 `readline` 的一个简单应用，我们用 `print` 函数输出提示信息，等待用户输入。

```
> DetermineSign := proc()
    local s, n;
    n := rand(-100..100)();
    print(`Is the sign of`,n,`positive?`);
    s := readline(terminal);
    evalb( (s="yes" and n>0) or (s="no" and n<=0) );
end:
> DetermineSign(a);
Is the sign of, 90, positive?
> yes
true
```

程序中，我们用到了函数 `rand`，它返回的是一个子程序，所以为了得到一个随机数，必须再调用一次（加上一对括弧）。在获得输入后，我们用 `evalb` 检验了输入的结果是否正确。

不过，`readline` 只能从终端读入一行的字符串，如果需要读入数据，我们可以使用另一个命令 `readstat`。

8.3.2 从终端读入表达式

由于 Maple 是一个符号代数系统，所以大多数情况下，我们需要输入的是表达式，而不是字符串。对于表达式，我们有另一个命令 `readstat`。它的命令格式是 `readstat(prompt)`。其中 `prompt` 是可选参数，它是等待输入时显示的提示。

```
[> deg := readstat("Enter degree: ");
Enter degree: n+1;
deg := n + 1
```

需要注意的是，`readstat` 命令读入的表达式必须用分号（或冒号）结尾。和 `readline` 不同，`readstat` 的输入不局限在一行之内。在输入时，和在交互式环境中一样，可以分行输入。使用 `readstat` 的另一个好处是如果用户在输入时语法有问题，Maple 会自动地给出出错信息，并且提示用户重新输入。

```
[> readstat("Enter a number: ");
Enter a number: 5^-8;
syntax error, '-' unexpected:
5^-8;
^
Enter a number: 5^(-8);

1
-----
390625
```

以下是 `readstat` 命令的一个简单应用，我们用它来实现一个更方便的求极限子程序。首先，如果输入的表达式中只有一个变量，那就默认这个变量是自变量；否则，提示用户输入自变量名。然后，再向用户询问要求极限的点。最后，调用系统子程序 `limit` 求得极限。

```
> GetLimit := proc(f::algebraic)
    local x, a, v;
    # 选择f中所有的变量
    v := select(type, indets(f), name);
    if nops(v) = 1 then
        x := v[1];
    else
        x := readstat("Input limit variable: ");
        while not type(x, name) do
            printf("A variable is required: but received
            %a\n", x);
            x := readstat("Please re-input limit
            variable: ");
        od;
    fi;
    a := readstat("Input limit point: ");
    Limit(f, x = a) = limit(f, x = a);
end;
```

在上面的程序中，调用了系统子程序 `indets`，它可以返回一个表达式中所有的未知变量（`indeterminates`）。在输出错误信息时，调用了系统函数 `printf`，它的用法和 C 语言中的同名函数几乎完全一样，它也用 % 表示数据的类型，和 C 语言不同的是，它可以用 %a 表示所有的 Maple 对象。比如程序中的 `x`，我们无法知道用户输入的 `x` 是什么类型的（当然也可以用 `type`，再用分支结构分别输出，不过那样就麻烦了），所以就用通用标识符 %a 表示它的类型。

首先，我们用它来求表达式 $\sin(x)/x$ 在 $x=0$ 点的极限。由于它只包含一个变量 `x`，所以程序就默认 `x` 为自变量，而不再询问极限变量了。

```
[> GetLimit( sin(x)/x );
Input limit point: 0;

lim sin(x)
x → 0 x = 1
```

我们再来看看包含有参变量的表达式。程序会先询问极限变量名，再询问求极限的点。

```
[> GetLimit( (exp(u*x)-1) / x );
Input limit variable: 0;
A variable is required: but received 0
Please re-input limit variable: x;
Input limit point: 0;


$$\lim_{x \rightarrow 0} \frac{e^{(u x)} - 1}{x} = u$$

```

在 readstat 命令中, 还可以加入许多的选项, 我们将在后续的章节中作更为详细的介绍。

8.3.3 把字符串转化为表达式

由于 Maple 的符号运算特性, 它可以随时随地地把字符串转化成为相应的表达式。我们在编写程序时, 也可以充分地利用这一特性, 编写出输入输出形式更为灵活的程序。比如, 我们可以用 readline 把用户的输入当作字符串读取, 而在必要的时候分析用户的输入, 再将其转化成可用的表达式。

利用 Maple 的 parse 命令就可以做到这一点, 它的功能相当于 Matlab 中的 eval 函数, 可以将一个字符串转化为相应的表达式。不过, 字符串所表示的必须是一个完整的表达式。

```
[> s := "a*x^2 + b*x + c";
      s := "a*x^2+b*x+c"
[> y := parse( s );
      y := a x^2 + b x + c
```

parse 命令并不对返回的表达式求值, 如果需要对它求值, 必须显式地调用求值函数。

```
[> a := 2;
      a := 2
[> z := parse( s );
      z := a x^2 + b x + c
[> eval( z );
      2 x^2 + b x + c
```

这一节中介绍的例子都是非常简单的, 但是, 你完全可以利用这些例子的方法构筑起实用的应用程序来, 比如 Maple 的教学软件, 交互式的数学教学软件等等。

8.4 扩展 Maple 命令

虽然你可以通过自己编写程序来满足各种不同的需要; 但有时, 扩展 Maple 原有的命令显得更为方便。许多已有的 Maple 命令都可以被扩展。在这一节中, 我们将介绍一些较为常用的命令的扩展方法, 包括用户自定义数据类型和操作符, 改变 Maple 的表达式显示形式, 还有例如 simplify 和 expand 等的常用命令的功能扩展。

8.4.1 自定义数据类型

在大多数现代编程语言中，都支持自定义的数据结构和数据类型。在 Maple 中也具有这样的功能，你只要把一个结构类型赋值给 ``type / TypeName``，以后 `TypeName` 就可以作为一个类型使用了。这样，你对于结构本身只需要写一次，降低了出错的可能性，也减少了工作量。

```
[> `type/Variables` := { name, list(name), set(name) };
> type( x, Variables );
true
> type( { x[1], x[2] }, Variables );
true
```

在这个例子中，我们把几个类型的集合赋给了 ``type / Variables``，也就是说 `Variables` 这种我们定义的数据类型，是这几个类型的集合。所以，不管单个变量还是变量的集合，都是 `Variables` 类型的。

如果觉得 Maple 判断的数据类型不够全面，你也可以将一个子程序赋给 ``type / TypeName``。在你测试一个数据对象是否具有 `TypeName` 类型时，Maple 会自动调用该子程序。不过，你的子程序必须返回布尔值，也就是 `true` 或者 `false`。

作为例子，下面定义一个全排列 (permutation) 的数据类型，也就是检测一个有序表是否含有从 1 到 n 的所有自然数，而且每一个只出现一次。提取有序表中所有元素的集合，再与 1 到 n 的自然数集比较，就可以实现这一要求。

```
[> `type/permutation` := proc(p)
    local i;
    type(p, list) and {op(p)}={seq(i, i=1..nops(p))};
end:
> type( [2,3,1,4], permutation );
true
> type( [1,2,3,1], permutation );
false
```

自定义的类型检测函数可以具有多余的一个参数。比如，要检测一个表达式 `expr` 是否具有类型 `TypeName(parameters)`，Maple 就会用如下形式调用检测函数 `TypeName(expr, parameters)`。比如我们可以定义一元线性表达式类型 `LINEAR`，它是一个未知变量的一次多项式。

```
[> `type/LINEAR` := proc(f, V::name)
    type(f, polynom(anything, V)) and degree(f, V)=1;
end:
> type( x^2, LINEAR(x) );
false
> type( a*x + b, LINEAR(x) );
true
```

8.4.2 自定义操作符

我们已经知道，Maple 中有着许多的操作符，比如 `+`，`*`，`^`，`and`，`not`，`union` 等等。所有这些操作符对于 Maple 来说，都有着特殊的意义：例如它们表示代数运算，集合运算等。

在 Maple 中还有着这样一类操作符，Maple 把它们当作操作符，但没有赋予它们实际的意义，我们称其为中性操作符（Neutral Operator）。这是留给用户的一个接口，用户可以用中性操作符来定义特殊的运算。中性操作符都以符号“&”开始，例如：

```
[> 7 &^ 8 &^ 9;
                                     (7 &^ 8) &^ 9
```

注意，Maple 对于中性操作符并没有赋予任何信息，它们不满足交换律和结合律：

```
[> evalb( 7 &^ 8 = 8 &^ 7 );
                                     false
[> evalb( (7&^8)&^9 = 7&^(8&^9) );
                                     false
```

在 Maple 内部，中性操作符是用子程序来表示的。例如， $7 \&^ 8$ 就可以用子程序调用形式写成： $\&^ (7, 8)$ 。

```
[> &^ (7, 8);
                                     7 &^ 8
```

Maple 自动地把运算符写在中间，一般情况下，这样比较容易读懂。但是对于参数个数不是两个的情况下，例如只有一个参数，或者有多于两个的参数，Maple 就会把表达式表示成为函数的形式了。

```
[> &^ (1), &^ (2, 3), &^ (4, 5, 6);
                                     &^ (1), 2 &^ 3, &^ (4, 5, 6)
```

中性操作符实际上是一种子程序，所以定义中性操作符的方法和定义子程序一样。作为例子，我们用一个中性操作符来实现两个哈密顿数（Hamiltonian）的乘法。哈密顿数又称为四元数（Quaternion），它是复数的一个扩充，就如同复数对实数的扩充一样。每一个哈密顿数都具有这样的形式： $a + bi + cj + dk$ ，其中 a, b, c, d 都是实数；而符号 i, j, k, l 具有下面的乘法性质： $i^2 = -1, j^2 = -1, k^2 = -1, ij = k, ji = -k, ik = -j, ki = j, jk = i, kj = -i$ 。

在哈密顿数的乘法程序 $\&^$ 中，我们把符号 I, J, K 作为三个特殊符号来用，所以，我们必须把原来作为虚数单位别称（alias）的 I 取消。

```
[> alias( I=I );
```

为了定义哈密顿数的乘法，首先我们定义哈密顿数的数据类型 Hamiltonian。

```
[> `type/Hamiltonian` := { `+`, `*`, name, realcons,
                             specfunc(anything, `&^`) };
                             type/Hamiltonian := {name, *, +, realcons, specfunc(anything, &^)}
```

下面的 $\&^$ 程序计算两个哈密顿数 x, y 的乘积。如果 x 或者 y 是实数，那么乘积就是普通的乘积——Maple 中的“*”。如果 x 或者 y 是一个和式，就将分配律作用其上： $x(u+v) = xu + xv$ 与 $(u+v)x = xu + xv$ 。如果 x 或者 y 是乘积，那就将所有的实数因子合并到一起；在这个时候，需要防止无穷递归的发生。如果在上述情况之外的话，我们就用“&^”返回不求值的形式。下面就是哈密顿积函数的源程序：（由于源程序较长，没有用 Maple 中的形式编辑，具体的格式可能稍有不同）

```
`&^` := proc( x::Hamiltonian, y::Hamiltonian )
```

```

local Real, unReal, isReal;
isReal := z -> evalb( is(z, real) = true );

if isReal(x) or isReal(y) then
    # x 或 y 是实数
    x * y;

elif type(x, `+`) then
    # x 是和式, u + v, 故  $x^y = u^y + v^y$ 
    map(`&^`, x, y);

elif type(y, `+`) then
    # y 是和式, u + v, 故  $x^y = x^u + x^v$ 
    map2(`&^`, x, y);

elif type(x, `*`) then
    # 选择 x 的实数因子
    Real := select(isReal, x);
    unReal := remove(isReal, x);
    #  $x^y = \text{Real} * (\text{unReal}^y)$ 
    if Real=1 then
        # 无实数因子
        if type(y, `*`) then
            Real := select(isReal, y);
            unReal := remove(isReal, y);
            Real := '`&^`'(x, unReal);
        else
            '`&^`'(x, y);
        fi;
    else
        # 实数因子*递归调用自己
        Real * `&^`(unReal, y);
    fi;

elif type(y, `*`) then
    # y 为乘积, 与 x 的情况相似, 但简单一些, x 此处不可能为乘积
    Real := select(isReal, y);
    unReal := remove(isReal, y);
    if Real=1 then

```

```

        '&^' (x, y);
    else
        Real * '&^' (x, unReal);
    fi;

    else
        '&^' (x, y);
    fi;
end:

```

图 8.1 哈密顿数乘法源程序

上面程序中，我们规定了特定的乘法规则，比如分配律等等；但对于哈密顿数基本单位符号 I, J, K 的乘法规则我们还没有给出定义。我们可以利用该程序的记忆表加入这些运算规则。

```

[> '&^' (I, I) := -1: '&^' (J, J) := -1: '&^' (K, K) := -1:
[> '&^' (I, J) := K: '&^' (J, I) := -K:
[> '&^' (J, K) := I: '&^' (K, J) := -I:
[> '&^' (K, I) := J: '&^' (I, K) := -J:

```

现在，你就可以用中性操作符 $\&^$ 进行哈密顿数的乘法了：

```

[> (1 + 2*I + 3*J + 4*K) &^ (5 + 6*I - 7*K);
                                     21-5 I-5 K+53 J
[> (5 + 6*I - 7*K) &^ (1 + 2*I + 3*J + 4*K);
                                     21+37 I-23 J+31 K
[> 123 &^ (4*I);
                                     492 I

```

它也可以用来做符号运算，但对于符号表示的实数，必须用 `assume` 进行设定。

```

[> a &^ K;
                                     a &^ K
[> assume(a, real):
[> a &^ K;
                                     a~ K

```

8.4.3 扩展 Maple 命令

在 8.4.1 小节中，我们已经学习了自定义数据结构的方法，对于自定义的数据类型，Maple 并不知道应该怎样处理它们。当然，你可以自己编写程序完成对它们的运算；但有时，扩展 Maple 的内置命令显得更为便捷。Maple 中可以进行扩展的常用命令有 `expand`, `simplify`, `diff`, `seires`, `evalf` 等。这些命令的用法我们在前面的章节中都已经学习过，现在，我们主要学习将它们适用范围进行扩展的方法。

例如，我们自己定义了多项式数据结构，用 `POLYNOM(u, a_0, a_1, ..., a_n)` 表示多项式 $a_0 + a_1 u + \dots + a_n u^n$ 。现在，需要扩展 Maple 的内置命令 `diff` 来使之适用于多项式类型 `POLYNOM`。

定义适用于特定数据类型 `F` 的 `diff` 命令的方法很简单，与我们定义一个新的数据类型类

似，只需要定义一个子程序`diff / F`就可以了。特别地，如果我们需要定义形如 $F(\text{arguments})$ 的函数对于变量 x 的求导运算，所定义的`diff / F`就具有这样的形式：

`diff / F` (arguments, x)

下面的程序段定义了一个以 u 作为基本变量的常系数多项式 POLYNOM 对于 x 求导的子程序：

```
> `diff/POLYNOM` := proc(u)
    local i, s, x;
    x := args[-1];
    s := seq( i*args[i+2], i=1..nargs-3 );
    'POLYNOM'(u, s) * diff(u, x);
end:
> diff( POLYNOM(x, 1, 1, 1, 1, 1, 1, 1), x );
      POLYNOM(x, 1, 2, 3, 4, 5, 6)
> diff( POLYNOM(x*y, 12, 34, 56), x );
      POLYNOM(x y, 34, 112) y
```

在 8.4.2 小节中，我们用中性操作符实现了哈密顿数的乘法，并且已经可以利用分配律进行简单的化简。但对于结合律，也就是 $x(yz) = (xy)z$ ，在程序中却没有予以实现。例如：

```
> x &^ I &^ J;
      (x &^ I) &^ J
> x &^ ( I &^ J );
      x &^ K
> evalb( %% = % );
      false
```

我们可以通过扩展 Maple 的化简命令 simplify 来实现这一点。和 diff 一样，对 simplify 进行扩展，也只需要对于` simplify / F`赋予子程序就可以了。可以编写子程序` simplify / &^`来实现哈密顿数的乘法结合律。

为了实现对于`&^`的化简运算，首先需要检测有待化简的表达式是否具有两重（或两重以上）的`&^`相嵌套的形式，Maple 的模式匹配命令 typematch 可以完成这样的检测。例如，它可以检查它的输入参数是否具有 $(a \&^ b) \&^ c$ 的形式，如果是，它还可以把 a, b, c 的具体表达式分离出来：

```
> x &^ y &^ z;
      (x &^ y) &^ z
> typematch( %, `&^` ( `&^` ( a::anything,
      b::anything ), c::anything ) );
      true
> a, b, c;
      x, y, z
```

模式匹配命令是一个非常有用的命令，可以利用它来实现许多不同数据结构的化简。下面，我们就用它来实现对于哈密顿数乘法的化简。

```

> `simplify/&^` := proc( x )
    local a, b, c;
    if typematch( x, '`&^`'({`&^`'(a::anything,
        b::anything), c::anything ) ) then
        a &^ ( b &^ c );
    else
        x;
    fi;
end:

```

现在，就可以利用 `simplify` 对哈密顿数用结合律进行化简了。

```

> x &^ I &^ J &^ K;
((x &^ I) &^ J) &^ K
> simplify(%);
-x

```

其他的命令，如 `expand`, `series`, `evalf` 等的扩展，在这里就不一一介绍了，如果需要的话，可以参考它们的在线帮助。

利用开放式设计，使得用户可以根据它们的具体要求灵活地定义便于使用的命令，这是 Maple 的设计思路。

8.5 编写自己的工具包

在用 Maple 解决复杂的问题时，常常会需要编写一系列相关的程序。在编写了这样的一系列程序之后，在以后研究相同问题时往往还有可能用到。Maple 的设计者也考虑到了这样的问题，他们用工具包将相关的函数和子程序归并成一些组，例如我们前面曾经介绍过的线性代数工具包 `linalg` 和绘图工具包 `plots`。在 Maple 中，用户也可以方便地建立自己的工具包。

10.1.1 工具包的结构

Maple 中的一个工具包，从结构上而言是一个映射表——每一个表项是工具包中的子程序名，对应的映射值是具体的子程序定义。所以，要建立自己的工具包，所需做的也仅仅是定义这样的一个映射表，然后将它保存到一个名为 `*.m` 的文件中。Maple 就可以象对待所有系统的工具包一样处理你编写的工具包了。要使用时，只需用 `with` 命令载入该工具包即可。

作为例子，我们定义一个简单的乘幂工具包。首先建立一个映射表：

```

> powers := table();
powers := table([
])

```

然后，定义表中的各个表项——工具包中的函数或子程序：

```
[> powers[sqr1] := proc(x::anything)
      x^2;
    end:
> powers[cube] := proc(x::anything)
      x^3;
    end:
```

需要引用表中的一个子程序也很方便，例如要引用上面定义的 `cube`，只需这样调用：

```
[> powers[cube](x+y);

      (x+y)^3
```

将子程序定义在映射表中有一个好处——它不会被外部的同名子程序所覆盖。例如，我们甚至可以对 `sqr1` 赋值，仍不改变映射表中的 `sqr1` 以及用它所定义的子程序。

```
[> powers[fourth] := proc(x::anything)
      powers['sqr1']( powers['sqr1'](x) );
    end:
> sqr1 := 3^2;

      sqr1 := 9
> powers[fourth](x);

      x^4
```

为了不影响后面的运算，这里还是将 `sqr1` 取消赋值：

```
[> sqr1 := 'sqr1':
```

除了用指标运算符[]引用映射表中的子程序外，还可以用 `with` 命令把映射表中的子程序调入到全局变量中去，这样就可以对它们直接进行引用了。

```
[> with(powers);

      [cube,fourth,sqr1]
> cube(x-y);

      (x-y)^3
```

和其他所有的 Maple 对象一样，可以用 `save` 函数将映射表保存到文件中去。例如，将上面定义的 `powers` 工具包存储到路径为 “D:\user\mylib” 下的二进制文件 `powers.m` 中（注意路径必须存在），可以这样调用：（字符串中的 “\” 需要用 “\\” 表示）

```
[> save( powers, "D:\\user\\mylib\\powers.m" );
```

为了检验我们建立的工具包 `powers` 是否可以象系统工具包一样使用，首先用 `restart` 命令将 Maple 重新初始化——也就是出去内存中所有用户定义的变量，并将环境变量恢复为默认值，即相当于 Maple 刚刚启动时的状态。

```
[> restart;
```

然后，为了使 Maple 能够找到我们建立的工具包文件 `powers.m`，必须将其路径加入到环境变量 `libname` 中去，`libname` 所设置的是系统搜寻工具包文件的路径。

```
[> libname := "D:\\user\\mylib", libname;

      libname := "D:\\user\\mylib", "D:\\PROGRAM FILES\\MAPLE\\lib"
```

设置完路径之后，就可以用 `with` 命令载入我们定义的工具包了：

```

[> with( powers );
                                     [cube,fourth,sqrt]
[> cube( 5 );
                                     125

```

虽然这个工具包是如此的简单甚至有些幼稚,但是通过它我们了解了在 Maple 中定义一个工具包的基本步骤: 首先定义子程序组成的映射表; 然后将它存储到*.m 文件中去; 再设置 Maple 的搜索路径, 就可以使用了。

10.1.2 工具包的初始化

有的时候, 工具包中含有预先定义的数据结构, 或者在使用工具包中的子程序前都需要执行某些 Maple 命令, 为了完成这些工作, 可以将它们编写成工具包的初始化函数——init。在 Maple 中用 with 命令载入工具包时, 如果该工具包中存在 init 函数, Maple 会自动运行它。

Maple 对于复数运算的支持已经是相当完美的了, 但在这里, 为了演示如何初始化工具包, 我们画蛇添足, 建立一个简单的复数运算工具包 cmplx。在这个工具包中, 定义一个新的数据类型 COMPLEX, 用来表示复数。例如复数 $a + bi$, 其中 a、b 分别是它的实部和虚部, 就可以表示成 COMPLEX(a, b)。

```

[> `type/COMPLEX` := 'COMPLEX'(realcons, realcons);
                                     type/COMPLEX:=COMPLEX(realcons,realcons)
[> z := COMPLEX(3, 4);
                                     z:=COMPLEX(3,4)
[> type(z, COMPLEX);
                                     true

```

在这个复数类型中, 我们可以方便地提取出它的实部和虚部来——分别就是它的第一个和第二个元素:

```

[> cmplx[realpart] := proc(z::COMPLEX)
    op(1, z);
end:
[> cmplx[imagpart] := proc(z::COMPLEX)
    op(2, z);
end:
[> cmplx[imagpart](z);
                                     4

```

同样, 如果给定了复数的实部和虚部, 也可以方便地建立一个复数类型的结构:

```

[> cmplx[makecomplex] := proc(a::realcons, b::realcons)
    'COMPLEX'(a, b);
end:

```

再简单地实现 COMPLEX 类型的加法运算:

```
> cmplx[addition] := proc(z::COMPLEX, w::COMPLEX)
    local x1, x2, y1, y2;
    x1 := cmplx['realpart'](z);
    y1 := cmplx['imagpart'](z);
    x2 := cmplx['realpart'](w);
    y2 := cmplx['imagpart'](w);
    cmplx['makecomplex'](x1+x2, y1+y2);
end;
```

cmplx 中的所有这些子程序都用到了全局变量`type/COMPLEX`，所以，我们必须在工具包 cmplx 的初始化子程序 init 中定义这一全局变量：

```
> cmplx[init] := proc()
    global `type/COMPLEX`;
    `type/COMPLEX` := 'COMPLEX'(realcons, realcons);
end;
```

现在，可以将工具包 cmplx 保存到文件 cmplx.m 中去了：

```
> save( cmplx, "D:\\user\\mylib\\cmplx.m" );
```

重新初始化 Maple，试一试我们的工具包 cmplx：

```
> restart;
> libname := "D:\\user\\mylib", libname;
    libname = "D:\\user\\mylib", "D:\\PROGRAM FILES\\MAPLE\\lib"
> with( cmplx );
    [addition, imagpart, init, makecomplex, realpart]
```

在用 with 载入工具包时，init 函数已经被执行了：

```
> type( makecomplex(3, 4), COMPLEX );
    true
```

10.1.3 建立自己的程序库

除了把 Maple 对象分别存储到文件目录系统中之外，还可以将这些文件的内容合并到一个较大的文件中去——也就是建立 Maple 的程序库（library）。利用程序库，可以使用与操作系统无关的文件名，这样就使不同类型计算机用户的程序相互交流更为方便。比之将每一个对象分别存储，程序库还可以节省存储空间。每一个目录下最多只允许包含一个 Maple 的程序库，而每一个库中可以包含任意多的程序。实际上，Maple 的一个库文件 maple.lib 中包含了所有的库函数。

在建立自己的程序库之前，最好将系统的程序库进行写保护，以免无意中将其破坏。Maple 的程序库文件包含在 libname 所对应的路径中，由于每个系统的安装不同，路径可能会有差异：

```
> restart;
> libname;
    "D:\\PROGRAM FILES\\MAPLE\\lib"
```

在 Maple 中不能建立新的程序库，而必须使用 Maple V Release 5 程序组中所带的程序 March。运行 March，会弹出一个对话框，在其中添加参数，就可以按要求生成新的程序库。例如，要用前面路径“D:\\user\\mylib”中的两个 m 文件建立一个新的程序库，可以加入

参数“-c D:\user\mylib 2”，第一个参数-c 表示建立新的程序库，第二个参数指明路径，最后的参数表示有两个 m 文件。

将新建的程序库的路径加入到 libname 中后，就可以用 readlib 载入程序库中的文件了。调用 readlib(name)，将载入程序库中 name.m 文件中的所有对象。

```
[> restart;
> libname := "D:\\user\\mylib", libname;
      libname := "D:\\user\\mylib", "D:\\PROGRAM FILES\\MAPLE\\lib"
> readlib(powers);
table([
  sqr1 = (proc(x::anything) x^2 end)
  cube = (proc(x::anything) x^3 end)
  fourth = (proc(x::anything) powers['sqr1'](powers['sqr1'](x)) end)
])
```

readlib 不仅载入了 powers 工具包，还将映射表作为返回值返回，所以，可以直接用这样的形式调用库中的子程序：

```
[> readlib(powers) [sqr1];
      proc(x::anything) x^2 end
> readlib(powers) [sqr1] (x);
      x^2
```

Maple 的程序库不仅可以用 March 通过 m 文件生成，还可以在 Maple 中用 savelib 函数在已有的程序库中加入 Maple 对象。savelib 的调用格式是这样的：

savelib (nameseq, "filename")

其中，nameseq 是需要存储的 Maple 对象名的序列，filename 是存储在程序库中的 m 文件名，它可以已经存在，也可以不存在。savelib 一次可以存储多个 Maple 对象，但必须有一个与文件名同名——它就会作为 readlib 的返回值。在不指定具体的对象 nameseq 时，savelib 将存储工作环境中所有已赋值的变量。

环境变量 savelibname 指定 savelib 所存储的程序库路径。例如，我们要在刚才建立的程序库中加入一个函数 mylibfun，可以这样设置：

```
[> savelibname := "D:\\user\\mylib":
> mylibfun := proc()
      "This is a library function!";
end:
> savelib('mylibfun', "mylibfun.m");
```

至此，就可以象调用系统库函数一样调用我们建立的库函数 mylibfun 了。

```
[> readlib(mylibfun) ();
      "This is a library function!"
```

但是，由于我们新建了程序库，如果每次都要用给 libname 赋值的方法重新加入路径，使用不太方便。

Maple 提供了一个初始化环境的接口，可以修改 Maple 的执行文件所在目录（Maple V Release 5 for Windows95 是 Bin.wnt 目录）下的文本格式初始化文件 Maple.ini，如果这个文件不存在，可以创建这个文件。在其中加入用户的初始化命令，就可以使 Maple 每次启动都

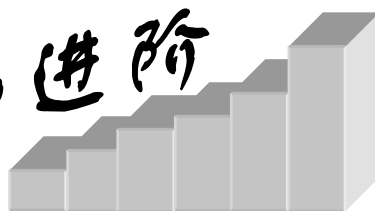
自动执行这些命令了。

例如，对于上面附加的程序库路径，可以在文件 `Maple.ini` 中加入这样的命令：

```
libname := "D:\\user\\mylib", libname:
```

注意，为了在初始化时避免不必要的输出信息，可以用冒号 “:” 作为命令结束以抑制输出。

起步与进阶



第

输入和输出

九

章

本章将围绕着 Maple 的输入输出函数,向大家介绍在 Maple 的命令行环境中或者程序设计中直接输入输出和读写文件的方法。虽然作为符号代数系统,Maple 更多的是直接和用户打交道,很少有大量的数据需要用文件方式和外界交换,但是在进行大型的计算时,往往不能立即给出结果,或者给出的结果繁琐,这是存入文件进一步处理就是必要的了。

本章具体包括以下内容:

- 🕒 Maple 可以处理的文件类型
- 🕒 文件处理的必要操作
- 🕒 格式化文件输入输出
- 🕒 二进制文件输入输出
- 🕒 表达式和数组的输入输出
- 🕒 C 和 Fortran 语言源程序的生成
- 🕒 字符串和数据之间的转换
- 🕒 Matlab 函数的调用

虽然 Maple 是一个处理数学问题的工具，但在很多情况下，用户也需要使用 Maple 外部的数据，或者需要向外部提供数据。另外，作为一门程序设计语言，在用来编写应用程序时，Maple 也需要读入用户的输入并向用户输出信息。为了满足这两方面的需要，和其他通用编程语言一样，Maple 也提供了一整套输入输出命令和函数，我们称其为 Maple 的 I/O 函数库。

这一章中，将主要介绍 Maple 的输入输出命令和函数的用法。除此之外，还将介绍从 Maple 程序向常用语言（例如 C 和 Fortran）的转换方法；并附带介绍在 Maple 中使用部分 Matlab 函数的方法。

9.1 输入输出的例子

这一节将通过例子演示使用 Maple 的 I/O 函数的方法。在例子中，我们将看到如何将一系列数据写入到文件中，以及如何从文件中读取它们的方法。例子中将用到下面这组数据，它的组织形式是一个有序表组成的有序表，表示一些数对 (x, y) ，其中 x 是整数，而 y 是实数：

```
> A := [[ 0, 0 ],
        [ 1, 0.8427007929 ],
        [ 2, 0.9953222650 ],
        [ 3, 0.9999779095 ],
        [ 4, 0.9999999846 ],
        [ 5, 1.0000000000 ]]:
```

在实际情况下，数据往往是用一个或一系列函数或者子程序得到的结果。这里为了演示，直接将它们在交互式环境中键入。

如果要利用另外的工具（例如用 C 语言编写的数据处理程序）处理 Maple 中得到的数据，最便捷的方法莫过于将它们存储到文件中了。为了使后处理程序可以理解这些数据，最简单的方法是将它们用格式化文本的方式写入到文件中，利用 Maple 的 I/O 库，这一操作非常简单：

```
> for xy in A do
    fprintf("myfile", "%d %e\n", xy[1], xy[2])
od:
> fclose("myfile");
```

这里的 for 循环是另一种形式的循环，它的元素 xy 在对象 A 的所有元素中遍取，它相当于这样的循环形式：

```
for i from 1 to nops (A) do
    xy := op ( i, A );
    .....
od:
```

这里用到了格式化输出函数 `fprintf`，它将数据按第二个参数的字符串指定的格式写入到文件 `myfile`（由第一个参数决定）中去，其余的参数都是需要输出的数据。如果用文本处理

软件打开文件 myfile（默认情况下在目录 Bin.wnt 中），就可以看到这样的数据：

```
0 0.000000e-01
1 8.427008e-01
2 9.953223e-01
3 9.999779e-01
4 1.000000e+00
5 1.000000e+00
```

函数 `fprintf` 的第一个参数 "myfile" 指定的是文件名，它用字符串形式给出，可以加入文件的路径。在一个文件名第一次出现在 `fprintf` 中时，如果该文件不存在，则函数建立一个新文件；否则就覆盖原有的文件。当然也可以不采用这两种方法，比如需要在原有文件的最后附加数据，我们在后面两节中会详细介绍。

格式字符串 "%d %e\n"，表示让 Maple 将第一个数据用十进制整数的形式 (%d) 输出，将第二个数据用类似于 Fortran 语言中的科学计数法 (%e) 输出。中间的空格表示在两个数据间用空格分开，最后的 "\n" 表示在两个数据输出后在输出一个回车符（另起一行输出）。在默认情况下，Maple 输出浮点数的时候会将其四舍五入到 6 位有效数字。有关 `fprintf` 函数得更详细的情况，将在后面专门介绍。

在结束了对一个文件的输出后，必须用 `fclose` 将其关闭。否则，由于操作系统对于文件的输出有缓冲的缘故，输出的数据有可能尚未写到文件中去。如果没有关闭文件，Maple 在退出时会将所有打开的文件全部关闭。

对于这一简单的例子，使用函数 `writedata` 将显得更为简单：

```
[> writedata("myfile2", A, [integer, float]);
```

这里，`writedata` 命令进行了一系列操作，包括打开文件，按指定格式写入数据，最后关闭文件。但是，`writedata` 对输出格式的控制不如 `fprintf` 严格，如果为了简便通用，`writedata` 是一个好的选择。

除了将数据输出到文件外，很多时候会需要从外部读入数据用 Maple 进行分析。从文件中读取数据和输出数据同样简单明了：

```
[> A := []:
> do
    xy := fscanf("myfile", "%d %e"):
    if xy = 0 then break fi:
    A := [op(A), xy]:
od:
[> fclose("myfile");
> A;
[[0, 0], [1, .8427008], [2, .9953223], [3, .9999779], [4, 1.000000], [5, 1.000000]]
```

这段小程序首先将 A 初始化为空有序表，然后每一步循环中，从文件中读出一对数据，再加入 A 中去。

函数 `fscanf` 从文件中读取一个字符串，并用格式字符串 "%d %e" 分析字符串从而得到其中的数据。如果读取成功，`fscanf` 将返回读取的数据组成的有序表，如果已经到文件尾，则返回 0。在第一次用一个文件名调用 `fscanf` 时，`fscanf` 将打开这个文件，如果文件不存在，Maple 将给出出错信息。

正如向文件输出数据一样，也可以用 `readdata` 更简便地从文件中读取数据：

```
> A := readdata("myfile", [integer, float]);
A :=
[[0, 0], [1, .8427008], [2, .9953223], [3, .9999779], [4, 1.000000], [5, 1.000000]]
```

函数 `readdata` 也完成了从打开文件，读入数据一直到关闭文件的一系列操作，但是它对于格式的控制也不是很严格。

这一节通过简单的例子介绍了输入输出的基本方法，利用这里介绍的方法，相信你一定已经可以解决许多问题了。但是，要更深入地理解 Maple 输入输出命令的用法和工作机制，还需要进一步的学习，这一章后面的部分将对它们作详细的介绍。

9.2 文件类型和打开方式

9.2.1 有缓冲文件和无缓冲文件

Maple 的输入输出函数可以使用两种类型的文件——有缓冲文件（STREAM）和无缓冲文件（RAW）。对于使用而言，它们没有区别，但在通常情况下，有缓冲文件比无缓冲文件的读写速度要快。在有缓冲文件中，Maple 先将数据写在文件的缓冲区中，在缓冲区满了或者是文件关闭的时候在将数据一次性写入文件。使用无缓冲文件，可以对底层的操作系统和磁盘文件结构更好地控制。不过如果作为一般用途，应尽可能使用有缓冲文件，Maple 中的大部分输入输出函数在默认情况下使用的也是有缓冲文件。

在输入输出命令中可以分别用标识 `STREAM` 或 `RAW` 表示文件的类型。

9.2.2 文本文件和二进制文件

大多数操作系统，包括 DOS/Windows，Macintosh 操作系统，VMS 等，对于包含着字符序列的文件（文本文件）和包含数据字节序列的文件（二进制文件）都有所区分。主要的区别在于对回车符的不同处理上，在不同的系统中，可能还存在着其他的差异，但在使用 Maple 的输入输出函数时，那些差异都是不可见的。

在 Maple 中，另起一行用的是单个的字符。尽管在 Maple 中输入这个标志是用“`\n`”表示，Maple 内部表示用的是 ASCII 码为 10 的字符。很多不同的操作系统对于另起一行的表示不同，例如，在 Dos/Windows 和 VMS 系统中，另起一行用两个字符表示，ASCII 码分别为 13 和 10（分别表示回车和换行）。Macintosh 系统中，另起一行用的是 ASCII 码为 13 的字符（回车）。

Maple 的输入输出函数对于文本文件和二进制文件都可以进行处理。在对文本文件输出时，Maple 自动将另起一行的标志转换成操作系统支持的表示方法。在从文本文件中读入数据时，Maple 自动将操作系统的表示方法转换成 Maple 中使用的单个字符表示的另起一行符号。而对于二进制文件，Maple 不作任何形式的转换。

如果运行在 UNIX 操作系统或者它的一些变体下，Maple 将不区分文本文件和二进制文件，对两种文件都作相同处理，不进行转换。

区分文本文件和二进制文件的标识符分别是 TEXT 和 BINARY。

9.2.3 读模式和写模式

文件打开的模式，可以是为了读取而打开的，也可以是为了在文件中写数据而打开的。在专为读取而打开的文件中，不能写入数据；但是在写模式中，同时也可以读取文件中的数据。如果在 Maple 中试图在读取模式的文件中写入数据，Maple 会关闭该文件，并重新用写模式将其打开。如果用户没有在文件中写入数据的权限，例如文件是只读的，那么 Maple 会给出出错提示。

区分文件的读写模式的标识符是 READ 和 WRITE，分别对应着读模式和写模式。

9.2.4 默认文件 default 和终端文件 terminal

Maple 的输入输出函数将用户的输入输出接口也作为文件来处理。标识符 default 和 terminal 就是指这种文件。标识符 default 是指当前的输入数据流，也就是 Maple 从中读取命令的数据流。标识符 terminal 表示顶层的输入数据流，也就是刚启动 Maple 时的输入数据流。

在交互式命令行环境中，这两个表示符所指的是同一个数据流。仅仅在用 read 从源程序文件中读取命令执行时才有差异。在这种情况下，default 指的是正在读取的文件，而 terminal 是指当前的工作区输入。而如果在 UNIX 系统中，如果 Maple 的输入被重定向为一个文件或者一个管道，那标识符 terminal 指的就是该文件或者管道。

需要注意的是，符号 default 和 terminal 是有特殊意义的，它们专门表示上述文件——用户的输入流，所以在任何时候都不要将它们挪作他用。

9.2.5 文件名和文件描述符

Maple 的输入输出命令可以用两种方式之一引用文件：用文件名或者用文件的描述符。

如我们在上一节中所见到的，用文件名引用文件是两者中较为简单的一种方式。在 Maple 第一次对一个文件进行操作时，它自动将文件打开，打开的方式（读/写，文本/二进制）由所作的操作决定。用文件名引用文件的最大好处是操作的方便。但有利必又弊，这种引用方式在对同一个文件进行熟练很多的小操作时，将会带来一定程度上的不便，也会在一定程度上影响运行的效率。

用文件描述符引用文件比起前一种方法稍微复杂一些，但对于对其他的高级语言有所接触的用户，这也许是更为合理的一种方法。首先给定文件名打开文件，并建立一个文件描述符。这样，描述符和一个打开的文件的关系就建立起来了，接下来，所有的操作都将对这个文件描述符进行。

用描述符引用文件的一个好处是在打开文件时可以有更大的自由度，可以用标识符确定文件是用文本方式，还是二进制方式打开；是用读模式打开，还是用写模式打开，还可以在必要的时候使用无缓冲文件。

具体用哪一种方式，要视情况而定，对于简单的文件操作，直接用文件名会简便易行；

对于较为复杂的任务，用文件描述符会更有利一些。在以后的介绍中，输入输出函数的参数 *fileIdentifier* 即可以指文件名，也可以指文件描述符。

9.3 文件控制命令

9.3.1 文件的打开和关闭

如果使用文件描述符引用文件，则在使用之前必须先显式地用 **Maple** 命令打开文件。打开文件可以用两个命令，**fopen** 打开的是有缓冲文件（STREAM），**open** 打开的是无缓冲文件（RAW）。

函数 **fopen** 的调用格式如下：

fopen (*fileName*, *accessMode*, *fileType*)

其中，*fileName* 是文件名字符串（包括路径）；*accessMode* 是文件的打开方式，可以为 **READ**、**WRITE** 或者 **APPEND**，分别表示文件用读模式、写模式、或者在文件末尾追加（写入）的方式打开；*fileType* 是文件的格式，可以为 **TEXT** 或 **BINARY**，分别表示文本格式和二进制格式。

用写入模式（**WRITE**）打开一个文件，但文件不存在的话，**Maple** 会首先建立这个文件；如果文件已经存在，**Maple** 会首先把该文件内容清空；如果用追加模式（**APPEND**），**Maple** 将在原来文件的末尾写入新的数据。但如果用读取模式打开一个不存在的文件，**Maple** 将产生一个错误。

函数 **open** 的调用格式如下：

open (*fileName*, *accessMode*)

其中的参数意义和 **fopen** 的相应参数一样。但不能指定文件类型，对于无缓冲文件，**Maple** 将以二进制方式打开。

这两个函数的返回值都是一个文件描述符，在接下来文件操作中，引用该文件都可以使用这一描述符，当然还是可以用文件名引用的。

在对一个文件的所有操作结束时，须要用命令显式地关闭文件，以保证 **Maple** 确实已经将缓冲区中的数据写入到磁盘文件中去了。同时也可以释放一部分系统资源，对操作系统而言，打开的文件数是有限制的。关闭文件可以用命令 **fclose** 或者 **close**，这两个命令是等价的，当然，为了在形式上和打开文件的两个不同命令对称，就有了这一对命令。它们的调用格式如下，唯一参数是文件描述符或文件名：

fclose (*fileIdentifier*)

close (*fileIdentifier*)

在关闭了一个文件之后，该文件的文件描述符就不再有效了。如果还需要再使用，就必须重新打开该文件，并重新给文件描述符赋值。同样，再退出 **Maple** 或者用 **restart** 重置 **Maple** 环境时，**Maple** 也将关闭所有的文件，包括用命令打开的文件和输入输出函数自动打开的文件。


```

[> fid := fopen("test.txt", WRITE);
                                fid := 0
[> writeline(fid, "This is a test");
                                15
[> fclose(fid);
[> writeline(fid, "This is another test");
Error, (in fprintf) file descriptor not in use

```

9.3.2 查询和设定文件的当前位置

每一个打开的文件都有一个当前位置的概念，接下来的所有读写操作，都将从这一位置开始。而每一次读写操作，都会将文件的当前位置移动到读写结束的位置上。利用命令 `filepos` 可以查询或者设置文件的当前位置。它的调用格式如下：

filepos (fileIdentifier, position)

其中 *fileIdentifier* 是该文件的描述符或文件名，如果文件尚未被打开，则 `filepos` 将用读取模式打开该文件。

position 是一个可选参数，如果不加以指明，则 `filepos` 返回文件的当前位置；如果给定一个整数，则 `filepos` 将文件的当前位置移动到 *position* 所指定的位置。如果文件的长度小于给定的整数，则 `filepos` 将文件的当前位置移动到文件尾，并返回文件的总长度。和一些编程语言不同，*position* 所指定的位置是文件的绝对位置，也就是从文件头开始计算的字节数（字符数）。

如果要将文件的当前位置移动到文件尾，或者要获得文件的长度，可以给定参数 *position* 的值为 `infinity`，例如：

```

[> filepos("test.txt", infinity);
                                16

```

9.3.3 检测文件尾

命令 `feof` 可以检测文件的当前位置是否已经到达文件尾。注意，它只对有缓冲文件（`STREAM`）适用，也就是用 `fopen` 打开的文件，或者是由输入输出函数自动打开的文件。其调用格式：

feof (fileIdentifier)

其中 *fileIdentifier* 是文件描述符或者文件名，如果给定的是文件名，而且文件尚未被打开，`Maple` 将用二进制方式的读取模式打开该文件。

如果该文件已经到达文件尾，`feof` 将返回 `true`，否则返回 `false`。

9.3.4 检测文件状态

命令 `iostatus` 将返回所有正在使用的文件的详细信息。它的调用格式如下：

iostatus ()

该命令返回的是一个有序表，具体包括以下的元素：

✧ `iostatus ()[1]` `Maple` 的输入输出系统正在使用的文件数，也就是已经打开但尚未被

关闭的文件数。

- ✧ `iostatus ()`[2] `read` 命令活动的递归调用次数（如果 `read` 正在读取的源文件中又有 `read` 命令，将导致 `read` 命令的递归调用）。
- ✧ `iostatus ()`[3] 操作系统所支持的打开文件数，也就是 `iostatus ()`[1] + `iostatus ()`[2] 的上界。
- ✧ `iostatus ()`[*n*] *n*>3，每个打开的文件的信息。

如果有文件被打开，也就是 `iostatus ()`[1]>0，就存在 *n*>3 的表项。其中每一项对应着一个打开的文件，也是一个有序表，它的具体元素如下：

- ✧ `iostatus ()`[*n*][1] 文件描述符（`fopen` 或 `open` 的返回值）。
- ✧ `iostatus ()`[*n*][2] 文件名字符串。
- ✧ `iostatus ()`[*n*][3] 文件类型（`STREAM`、`RAW` 或 `DIRECT`）。
- ✧ `iostatus ()`[*n*][4] 底层操作系统使用的文件指针或文件描述符（分别以 `FP = integer` 或 `FD = integer` 的形式给出）。
- ✧ `iostatus ()`[*n*][5] 文件模式（`READ` 或 `WRITE`）。
- ✧ `iostatus ()`[*n*][6] 文件格式（`TEXT` 或 `BINARY`）。

9.3.5 删除文件

有的计算过程中，需要建立一些临时文件来存储中间数据。在使用结束时，可以用 Maple 命令 `fremove` 删除这些文件，它的调用格式为：

`fremove (fileIdentifier)`

其中 *fileIdentifier* 是需要删除的文件描述符或者文件名。如果文件已经被打开，Maple 在删除前会首先关闭文件。如果文件不存在，这个命令将引发一个错误。

有时需要删除一些文件，但并不知道文件是否存在，这时可以用 `traperror` 捕获可能出现的错误，以免不必要的错误信息输出。例如：

```
[> traperror(fremove("myfile.txt")):
```

9.4 输入命令

9.4.1 从文件中读入文本

用 Maple 的输入命令 `readline` 可以从文件中读入一行文本，将所有字符组成的字符串返回，但不包括行结束符“`\n`”。如果 Maple 无法读取一行（例如已到文件尾），它将返回整数 0 而不是一个字符串。`readline` 的调用格式为：

`readline (fileIdentifier)`

其中 *fileIdentifier* 是文件名或者描述符。和 Maple 以前的版本相兼容，这里的 *fileIdentifier* 可以省略，默认情况下，Maple 从文件 `default` 中读入，也就是用户的输入。这样，`readline ()`

和 `readline (default)` 是等价的。

如果用 `-1` 作为 `readline` 的参数, `readline` 也将从 `default` 文件中读入一行文本, 除了以 “!” 和 “?” 起始的行 (分别表示执行操作系统命令和帮助)。

如果用文件名作为参数, 但该文件尚未被打开, Maple 将以文本格式的读入模式打开该文件。如果已经到达文件的结束, `readline` 将返回 0 并关闭该用文件名打开的文件。

作为例子, 下面的子程序从一个文件中读入文本, 并将其输出在默认的输出流中:

```
> ShowFile := proc( fileName::string )
    local line;
    do
        line := readline(fileName);
        if line = 0 then break fi;
        printf("%s\n", line);
    od;
end;
```

9.4.2 从文件中读入任意多字节

命令 `readbytes` 可以从文件中读入一个或多个字节 (或字符), 如果读取成功, 它将返回一个整数的有序表 (或者字符串), 如果已到文件尾, `readbytes` 将返回 0。它具有以下的调用格式:

`readbytes (fileIdentifier, length, TEXT)`

其中 *fileIdentifier* 是文件名或文件描述符; 可选参数 *length* 表示需要从文件中读取的字节数, 默认情况将读取一个字节; 如果指定 *TEXT*, 则结果将以字符串形式返回, 否则将给出一个整数的有序表。

参数 *length* 也可以给定为 `infinity`, 这时, Maple 将读取从文件的当前位置起直到文件尾的所有字节。

如果指定了 *TEXT*, 而且文件中包含有值为 0 的字节 (字符串结束标志), 则 Maple 将读取到这个字节为止。

如果用一个尚未打开的文件名为参数调用 `readbytes`, Maple 将以读取模式打开文件, 若同时指定了 *TEXT*, 则文件打开格式为文本, 否则就以二进制格式打开。对于一个用文件名调用的文件, 如果已经到文件尾, 则 `readbytes` 返回 0 并关闭该文件。

下面的例子是用 `readbytes` 编写的文件拷贝子程序:

```
> CopyFile := proc( sourceFile::string, destFile::string )
    writebytes(destFile, readbytes(sourceFile, infinity))
end;
```

9.4.3 格式化输入

命令 `fscanf` 和 `scanf` 从文件 (或输入流) 中读入数据, 并根据给定的格式从输入中读取数值或字符串, 然后将所读取到的数值或字符串组成的有序表作为返回值返回。如果已经到达文件尾, 则返回 0。

它们的调用格式如下:

fscanf (*fileIdentifier*, *format*)

scanf (*format*)

其中 *fileIdentifier* 是文件描述符或者文件名。直接调用命令 **scanf** 相当于用 **default** 作为参数 *fileIdentifier* 调用命令 **fscanf**。

如果用文件名作参数调用 **fscanf**，而且文件尚未被打开，Maple 将以文本格式的读取模式打开该文件。如果在用文件名作参数时，**fscanf** 返回 0（表示文件结束），则 Maple 自动关闭该文件。

format 是指定输入格式的字符串，它由一系列格式说明和其他的间隔字符组成。每一个格式说明表示一个数据项，它具有以下的形式，其中方括弧扩起来的部分表示可选部分：

%[*][width]code

每一个格式说明都由“%”起始，可选的星号“*”表示让 Maple 扫描该数据项，但不作为数据返回，也就是略去该输入数据。

可选的 *width* 表示该数据项的最大宽度，利用它可以区分两个紧密连结在一起的数据或无分隔的字符串。

code 表示输入的对象类型，它将决定返回的有序表中的对象类型。它可以为以下的字符之一：

- ✧ **d** 或 **D** 接下去的一组输入的字符表示一个有符号或者无符号的十进制整数，返回 Maple 的整型数据。
- ✧ **o** 或 **O** 接下去的一组输入的字符表示一个有符号或者无符号的八进制整数，返回 Maple 的整型数据（转换为十进制）。
- ✧ **x** 或 **X** 接下去的一组输入的字符表示一个有符号或者无符号的十六进制整数，返回 Maple 的整型数据（转换为十进制）。
- ✧ **e**, **f** 或 **g** 接下来的一组输入的字符表示一个十进制数，可以包含小数点，或者用 **E** 或 **e** 表示 10 的幂次的科学计数法，结果为 Maple 的浮点数。
- ✧ **he**, **hf** 或 **hg** 接下来的输入数据构成一个浮点数（或整数）的一维或二维数组。读入的数据可以有三种：数值、分隔符、终止符。数值的形式是任意的，可以为整数、小数或者用科学计数法表示的浮点数（包括 **E**、**e**、**D**、**d** 作为指数符号）；分隔符可以为空格、逗号或者方括弧，二维的数组，用方括弧作为不同维之间的分隔；任何不成对出现的方括弧，或者方括弧结束后没有紧跟着逗号，以及任何其他字符，都将作为这一输入的终止符。如果输入中包含有反斜杠“\”，则反斜杠之后的一个字符将被忽略。
- ✧ **hx** 接下来的输入数据构成一个 IEEE 标准的十六进制浮点数（每一个数字 16 个字符）的一维或二维数组，数据的分隔和结束与 **he**, **hf** 或 **hg** 的相同。
- ✧ **s** 读入接下来的一系列字符直到（但不包括）空格或者回车，并将其作为 Maple 的字符串返回。
- ✧ **a** Maple 读入接下来的一系列字符直到（但不包括）空格或者回车，并将其作为 Maple 命令加以分析，返回一个未求值的 Maple 表达式。
- ✧ **m** 接下来的数据是用 Maple 的 .m 文件格式编码的表达式，Maple 将读取一个完整的表达式，它将忽略 *width* 指定的宽度，返回的是一个 Maple 表达式。

- ◇ **c** 读取下一个字符（可为空格）并作为 Maple 字符串。如果指定了宽度 *width*，将读取指定的字符数，并作为字符串返回。
- ◇ **[...]** 在方括弧中间的字符为接受的字符，Maple 读取所有在其中出现的字符，直到遇到不是方括弧中的字符，将结果作为字符串返回。如果方括弧中以 “^” 起始，则表示接受所有未列出的字符。如果需要指定字符 “]”，则它必须出现在最开头、或者紧跟在 “^” 的后面。在指定字符时可以利用 “-” 表示字符的范围，例如 “A-Z” 表示所有的大写字母。但如果要指定字符 “-”，必须出现在所有字符的开头或者末尾。
- ◇ **n** 返回从第一个数据直到这一项为止扫描的字符总数。

除了格式说明符以外，Maple 将跳过输入中出现的 *format* 中的间隔字符；但对于 *format* 中的空格将忽略，除非出现在 “%c” 之前，这时 Maple 在读入下面的字符前将跳过所有的空格。

如果在读取任何一个数据项时 Maple 遇到了错误，都将返回一个空的有序表。

下面的例子是一个读取表格式文件的子程序，文件中每一行的数据个数是不同的，每一行的第一个数字表示这一行中的数据个数，每一个数据都由逗号分隔开。

```
> ReadRows := proc( fileName::string )
    local A, count, row, num;
    A := [];
    do
        # 确定该行的数据个数
        count := fscanf(fileName, "%d");
        if count = 0 then break fi;
        if count = [] then
            ERROR("integer expected in the file")
        fi;
        count := count[1];

        # 读入一行中的数据
        row := [];
        while count > 0 do
            num := fscanf(fileName, ",%e");
            if num = 0 then
                ERROR("unexpected end of file")
            fi;
            if num = [] then
                ERROR("number expected in file")
            fi;
            row := [op(row), num[1]];
            count := count - 1
        od;
        # 加入该行数据
        A := [op(A), row]
    od;
    A;
end;
```

9.4.4 读入 Maple 语句

Maple 的命令 `readstat` 可以从终端输入流 `terminal` 中读入一条 Maple 语句。Maple 将分析并对该语句求值，返回结果。它的调用格式如下：

`readstat (prompt, ditto3, ditto2, ditto1)`

其中 *prompt* 是输入的提示，如果省略，Maple 将不给出提示；*ditto3*, *ditto2*, *ditto1* 给定的是输入命令中同上操作符（`%%%`, `%%`, `%`）的对应语句，也可以省略。

对于 `readstat` 命令的输入必须是单一的语句，可以分作几行输入，但不能输入多条语句。如果输入的语句中含有语法错误，Maple 将返回错误信息和出错位置。

下面的简单例子演示了在子程序中用 `readstat` 语句获得用户的表达式输入的方法。

```
> InteractiveDiff := proc()
    local a, b;
    a := readstat("Please enter an expression: ");
    b := readstat("Differentiate with respect to: ");
    printf("The derivative of %a with respect to %a \
is %a\n", a, b, diff(a, b))
end:
> InteractiveDiff();
Please enter an expression: sin(x^x);
Differentiate with respect to: x;
The derivative of sin(x^x) with respect to x is cos(x^x)*x^x*(ln(x)+1)
```

9.4.5 读入表格式数据

Maple 命令 `readdata` 可以输入文本格式的表格式数据。对于简单的数据表格（数据方阵），使用这一命令比起用 `fscanf` 和循环要方便许多。命令 `readdata` 的调用格式如下：

`readdata (fileIdentifier, dataType, numColumns)`

其中 *fileIdentifier* 是文件描述符或文件名；*dataType* 指定表格中每一列的数据类型（用一个有序表形式给出，可以为 `integer` 或 `float`），默认的类型是 `float`；*numColumns* 指定了表格的列数，省略时以 *dataType* 中的元素个数决定表格的列数。

如果表格只有一列，`readdata` 将返回一个有序表，否则，`readdata` 返回一个二重的有序表，每一个子表表示表格中的一行数据。

如果用文件名方式指定文件，而且文件尚未被打开，Maple 用文本格式的读取方式打开文件。另外，如果用文件名指定文件，`readdata` 在结束后将自动关闭文件。

9.5 输出命令

9.5.1 利用 `interface` 命令设置输出参数

Maple 的 `interface` 命令并不是一个输出命令，但可以用它来改变 Maple 的输出语句给出

的结果。用 `interface` 设置接口参数的调用格式如下：

`interface (variable = expression)`

其中 `variable` 是要设置的接口参数，`expression` 为要设置的值，可以在同一个 `interface` 命令中设置多个参数，将几个等式之间用逗号间隔即可。对于具体参数的含义及取值，我们在输出命令中详细介绍，也可以参考 `interface` 的在线帮助。

如果不给定 `expression`，则 `interface` 返回当前的参数设置值。但一次只能查询一个接口参数。

9.5.2 一维表达式输出

Maple 的输出命令 `lprint` 将 Maple 表达式用一维方式输出，所谓一维方式，就是类似于我们输入 Maple 命令的方式。在大多数情况下，`lprint` 的输出可以直接作为 Maple 的输入使用。但也有反例，最简单的就是在输出包含有特殊字符的变量名，在 Maple 的输入中，带有特殊字符的变量名需要用反向撇号 “`” 括起来，但由于历史的原因，`lprint` 对于这些变量名的输出不带有反向撇号。

`lprint` 的调用格式如下：

`lprint (expressionSequence)`

其中，`expressionSequence` 是要输出的表达式序列。`lprint` 将依次输出每一个表达式，表达式之间用三个空格分开。在所有表达式输出完后，Maple 将另起一行。

命令 `lprint` 总是向默认的输出流 `default` 输出，不过，也可以用 `writeto` 和 `appendto` 命令（将在这一节的后面部分介绍）将它的输出暂时重定向到文件。

`interface` 设置的接口参数 `screenwidth` 将影响 `lprint` 的输出。如果可能的话，Maple 在输出时将尽可能在标识符之间分行，也就是不在数字、变量名中间断开。如果数字或变量名过长，`lprint` 将在断开处行末尾加上反斜杠的续行标志 “\”。

9.5.3 二维表达式输出

Maple 的输出命令 `print` 可以将表达式用二维的形式输出。具体的输出形式取决于系统的设置（菜单 `Options | Output Display`，参见第0章相应部分）。`print` 命令的调用格式如下：

`print (expressionSequence)`

其中 `expressionSequence` 是要输出的表达式序列，Maple 将依次输出，并在两个表达式之间用逗号隔开。`print` 的输出总是面向默认的输出流 `default`，当然，也可以使用命令 `writeto` 或 `appendto` 将输出暂时重定向到文件。

`interface` 设置的一些接口参数将影响到 `print` 的输出格式，包括以下几个：

- ✧ `prettyprint` 选择 `print` 输出的格式。如果将 `prettyprint` 设置为 0，`print` 的输出与 `lprint` 相同；如果设置为 1，则 `lprint` 输出带格式的文本形式；如果设置为 2，则输出为图形形式的标准数学格式。默认的 `prettyprint` 设置为 2。
- ✧ `indentamount` 这一参数设置表达式在续行时的缩进。仅对于 `prettyprint` 设置为 1 或者在输出子程序时有效。默认的 `indentamount` 设置为 4。
- ✧ `labelling` 或 `labeling` 可以设置为 `true` 或者 `false`，表示在输出时是否允许用标识

替换表达式中的重复子式。对于复杂的表达式而言，允许替换常常可以将长的表达式缩短，并且便于阅读。默认的 `labelling` 设置为 `true`。

- ✧ `labelwidth` 允许替换的子式大小，仅在 `labelling` 设置为 `true` 是有效。这个大小是子表达式的近似宽度，在 `prettyprint` 设置为 1 时按字符数计算。
- ✧ `screenwidth` 设置输出时的屏幕宽度。按照字符数计算，仅对于 `prettytype` 为 0 或 1 时有效，在 `prettyprint` 为 2 时，输出为图形形式，Maple 自动控制宽度。
- ✧ `verboseproc` 设置 Maple 的子程序输出形式。如果设置为 1，Maple 只输出用户定义的子程序的程序体，对于系统子程序，只打印参数或者还包括子程序的说明；如果设置为 2，Maple 对于所有的子程序都完整输出；如果设置 `verboseproc` 为 3，将完整地打印所有子程序，并同时输出子程序映射表的内容。

在交互式环境中，Maple 自动地显示计算的结果，显示的格式与 `print` 的输出格式相同。所以，上面的这些设置对于自动的输出也是同样有效的。

下面的例子显示了几种 `prettyprint` 设置对于输出的影响：

```
> print( expand((x+y)^6) );
      6 5 4 3 2 1 0
      x + 6 x y + 15 x y + 20 x y + 15 x y + 6 x y + y
> interface( prettyprint = 1 );
> print( expand((x+y)^6) );
      6 5 4 2 3 3 2 4 5 6
      x + 6 x y + 15 x y + 20 x y + 15 x y + 6 x y + y
> interface( prettyprint = 0 );
> print( expand((x+y)^6) );
x^6+6*x^5*y+15*x^4*y^2+20*x^3*y^3+15*x^2*y^4+6*x*y^5+y^6
```

9.5.4 输出 Maple 字符串

Maple 的输出命令 `writeline` 可以向文件输出一个或多个 Maple 字符串。每输出一个字符串，Maple 都将在文件中另起一行。它的调用格式如下：

writeline (*fileIdentifier*, *stringSequence*)

其中 *fileIdentifier* 是文件名或文件描述符，*stringSequence* 是要输出的字符串的序列。如果不指定任何字符串，`writeline` 将输出一个空行。

9.5.5 向文件输出任意多字节

命令 `writebytes` 可以向文件中输出一系列字符或者字节。它的调用格式如下：

writebytes (*fileIdentifier*, *bytes*)

其中 *fileIdentifier* 是文件名或者文件描述符。*bytes* 指定了需要输出的字节，可以是一个字符串，也可以是一个整数的有序表。如果用文件名调用 `writebytes`，而且文件尚未被打开，则 Maple 将用写入模式打开该文件。若在 `writebytes` 中给定的是整数有序表，则 Maple 以二进制格式打开该文件，若给定字符串，则以文本格式打开该文件。

9.5.6 格式化输出

与 `fscanf` 和 `scanf` 相对应，Maple 的输出命令 `fprintf` 和 `printf` 可以格式化地输出 Maple

对象。它们的调用格式如下：

fprintf (*fileIdentifier*, *format*, *expressionSequence*)

printf (*format*, *expressionSequence*)

其中 *fileIdentifier* 是文件名或者文件描述符。调用命令 **printf**，相当于用 **fprintf** 在默认输出文件 **default** 中输出。如果用文件名调用 **fprintf**，而且文件尚未被打开，则 Maple 将用文本格式的写模式打开该文件。

格式描述字符串 *format* 和 **fscanf** 中的有类似之处，但所包含的信息更多。它也是由一系列格式说明和间隔字符所组成的，每一格式说明都以“%”起始，具体的语法如下：

%[*flags*][*width*][.*precision*]*code*

其中可选的 *flags* 标志可以为以下标志中的一个或几个，它们的含义如下：

- ✧ + 使输出的数值总前带有符号——“+”或“-”（默认对于正数不显示符号）。
- ✧ - 使输出左对齐（默认为在宽度范围内右对齐）。
- ✧ 0 在输出的左边用 0 补满（如果已经指定了“-”，则忽略“0”）。

可选项 *width* 指定了该输出域的最小宽度；*precision* 指定的是浮点数据的精度，也就是小数点后的位数，对于字符串，*precision* 指定输出域的最大宽度。也可以指定 *width* 或 *precision* 为“*”，这时输出的宽度和精度将取决于要输出的数据本身的宽度与精度。

code 指定的是要输出的数据类型，可以有以下几种：

- ✧ d 将数据对象以十进制整数格式输出。
- ✧ o 将数据对象以八进制整数格式输出。
- ✧ x 或 X 以十六进制整数格式输出数据对象。对于 10~15 的数码，分别用字母 A~F 或 a~f 表示（分别对应着格式符 X 和 x）。如果输出的对象是一个硬件浮点数组（**harray**），数组的所有元素将以 IEEE 标准的十六进制浮点格式输出（宽度为 16）。每一行将输出一个一维数组，每两个数据间将以一个空格间隔。
- ✧ e 或 E 用科学计数法输出浮点数，对应的指数符号分别为 e 或者 E，指数部分将包括一个符号和至少三位数码。对于基数部分，如果未指定精度，将默认为输出小数点后 6 位。如果有待输出的对象是一个硬件浮点数组，将以科学计数法输出数组中所有的元素，输出的数据排列与“x”格式中介绍的浮点数组类似。
- ✧ f 将数据对象用定点数形式输出。具体的小数精度由 *precision* 决定。如果要输出的是一个硬件浮点数组，数组中所有的元素都以将这样的形式输出，排列和“x”格式中介绍的类似。
- ✧ g 或 G 数据对象将以“d”、“f”、“e”（或者“E”，如果指定的格式为“G”）格式输出，取决于具体对象的值。如果数值中不含小数点，Maple 将用“d”格式输出；如果数值小于 10^{-4} 或者大于 $10^{precision}$ ，则 Maple 用“e”（或“E”）格式输出；对于其他的情况，都将用“f”格式输出。对于硬件浮点数组，数组中的每一个元素都将根据具体情况用这几种方式之一输出，元素的排列与“x”中介绍的相同。
- ✧ c 输出一个 Maple 字符串，其中必须只包含有恰好一个字符。
- ✧ s 输出一个 Maple 字符串，字符数至少为 *precision*，至多为 *width*。
- ✧ a 用 Maple 语法输出一个表达式，宽度至少为 *width*，至多为 *precision*。如果指

定了 *precision*，那将有可能因为表达式的截断，而引起输出不完整。

✧ **m** 用 Maple 的 **m** 文件格式输出一个表达式，宽度至少为 *width*，至多为 *precision*。

如果指定了 *precision*，有可能因为表达式的截断，造成输出不完整。

✧ **%** 输出百分号 “%”。

以上格式中所有的浮点数格式都支持整数、有理数或者浮点数的数据对象，Maple 在输出前会自动进行类型转换。

对于 *format* 中其他的间隔字符，Maple 将全部直接予以输出。在 **fprintf** 或者 **printf** 输出结束时，不会自动另起一行。如果需要，可以在 *format* 字符串的末尾加上 “\n”。

interface 的接口参数对于命令 **fprintf** 和 **printf** 的输出都没有影响。

9.5.7 输出表格式数据

命令 **writedata** 可用于向文本格式的文件输出表格式数据。和 **readdata** 类似，许多时候用它输出数组比用循环和 **fprintf** 更为方便。它的调用格式如下：

writedata (*fileIdentifier*, *data*, *dataType*, *defaultProc*)

其中 *fileIdentifier* 是数据将要向其输出文件名或者文件描述符。如果用文件名指定文件，并且文件尚未被打开，则 Maple 将自动以文本格式的写模式将其打开。更进一步，对于用文件名打开的文件，**writedata** 在输出完毕后将自动将其关闭。

data 是要输出的数据，必须为向量、矩阵、有序表或者二阶有序表。如果要输出向量或有序表，则每一个数据将分别输出在一行上，形式上是一个列向量。如果要输出矩阵或者二阶有序表，则矩阵的每一行、或者每一个子有序表都分别在一行上输出，其中每一个数据之间用制表符间隔开。

可选参数 *dataType* 指定数据的具体输出类型，可以指定为整数、浮点数（默认）或者字符串。如果指定为 **integer**，对应的输出对象必须是数值型的，Maple 将它们转化为十进制整数输出（对于有理数和浮点数作截断处理）。如果指定为 **float**，对应的数值型对象输出都转化为浮点数格式。如果指定 **string**，则对应的输出对象必须为字符串。在输出矩阵或者二阶有序表时，*dataType* 可以为一个类型名称的有序表，分别指定每一列的输出类型。

可选参数 *defaultProc* 指定了在输出对象与指定类型不匹配时的处理函数（用户自己编写）。该处理函数必须由两个参数，第一个参数是输出文件的文件名或者描述符，第二个参数是有待处理的数据（类型不匹配的数据）。例如，一个处理表格中特殊数据的实用的函数可以这样编写：

```
[> UsefulDefaultProc := proc(f, x)
      fprintf(f, "%a", x)
    end;
```

下面的例子计算并向文件用浮点形式输出一个 5×5 的希尔伯特阵：

```
[> writedata("hilbertFile.txt", linalg[hilbert](5));
```

输出文件中的数据如下：

```
1          .5          .3333333333 .25          .2
.5          .3333333333 .25          .2          .1666666667
```

```
.3333333333 .25          .2          .1666666667 .1428571429
.25          .2          .1666666667 .1428571429 .125
.2          .1666666667 .1428571429 .125          .1111111111
```

9.5.8 写透文件缓冲

文件的缓冲区, 会导致从用输出命令输出, 到 Maple 将输出的数据真正写入到磁盘文件中有一定的延时。这一机制的目的在于将小块分散的数据合并成大的数据块写入文件, 以提高执行效率。

一般情况下, Maple 自动地处理何时将缓冲区数据写入到文件的问题。但有的时候, 用户希望确认重要的数据已经安全的写入到文件中, 以免后续计算过程中的意外中断导致数据的丢失。为了达到这一目的, Maple 提供了写透缓冲区的命令 `fflush`。它具有如下的调用格式:

`fflush (fileIdentifier)`

其中 *fileIdentifier* 为文件名或者文件描述符。调用时, Maple 将缓冲区中所有尚未写入到文件中的数据都确实地写入到文件中。

对于即将关闭的文件, 没有必要用 `fflush` 命令, 因为关闭文件时, Maple 会自动地将所有缓冲区中的数据写入到文件中。

9.5.9 默认输出流的重定向

前面的介绍中, 曾经涉及一些只能对默认输出流进行输出的命令, 例如 `print` 或者 `lprint` 等。Maple 命令 `writeto` 和 `appendto` 可以将默认输出流 `default` 重定向到文件, 它们的调用格式如下:

`writeto (fileName)`

`appendto (fileName)`

其中 *fileName* 指定了输出重定向的文件名。如果使用 `writeto`, Maple 将覆盖原有的同名文件; 如果使用 `appendto`, Maple 将在原有文件的后面追加数据。这两个命令使用的文件必须是未被打开的, 如果指明的文件已被打开, 将产生一个错误。

如果指定的 *fileName* 为终端输出流 `terminal`, 则默认输出流又被重新设置回刚启动 Maple 时的情况, 这时使用两个命令是等价的。

不过, 在 Maple 中的交互式环境中使用这两个命令是不推荐的, 因为对于写入文件时出现的错误信息也将输出到文件, 使用户无法看到。所以, 这两个命令一般用在子程序中。

9.6 转换命令

9.6.1 C 语言、Fortran 语言代码生成

Maple 中提供了从 Maple 语言向其他通用编程语言转换的命令，现在所支持的语言有 C 语言和 Fortran 语言。这样的转换在利用 Maple 推倒算法公式时特别有用，Maple 的转换命令可以直接从复杂的 Maple 表达式直接得到 C 语言或者 Fortran 语言的源代码。

这两种转换的命令分别为 C 和 fortran。除了这两个命令之外，Maple 的 codegen 工具包中还提供了其他的一些辅助命令。由于 C 是单个字母，常常会无意中将它作为变量名称，为了避免误用，默认情况下 Maple 并不将其调入，使用前需要先用 readlib 将其载入。这两个命令的调用格式如下：

fortran (*expression*, *options*)

C (*expression*, *options*)

其中，*expression* 可以是以下几种形式：

- ✧ 单个 Maple 表达式：这时，Maple 将生成一组 Fortran 或者 C 语句，计算表达式的结果。
- ✧ 一个由一系列形如 $name = expression$ 的 Maple 等式组成的有序表：这时，Maple 将生成一系列 Fortran 或 C 语句，计算每一个表达式 *expression*，并将其值赋给变量名为 *name* 的变量。
- ✧ 一个表达式数组的名称：这时，Maple 将生成的 Fortran 或 C 语句将计算一个数组，数组的每一个元素是表达式数组中的相应元素的值。
- ✧ 一个 Maple 子程序：Maple 生成一个 Fortran 的子程序或者 C 的函数。

对于 Maple 语句中调用的函数，转换时使用对应的 Fortran 或 C 内建函数，如果没有对应的函数，则转换成同名的函数。对与默认转换，可以用对函数 `fortran/function_name` 赋值的方法加以重新指定（注意使用反向撇号）。该函数有三个参数，其一是 Maple 中的函数，其二是函数的参数个数，最后一个是参数类型（双精度或者单精度）。例如：

```
> `fortran/function_name`(arctan, 1, double) := datan;
      fortran/function_name(arctan, 1, double) := datan
> `fortran/function_name`(arctan, 2, single) := atan2;
      fortran/function_name(arctan, 2, single) := atan2
```

在对数组进行转换时，C 转换命令将所有的数组起始指标都重新设置为 0，使其符合 C 语言中的标准；而 fortran 转换命令则将数组设置为指标从 1 开始。

作为例子，我们计算一个不定积分的解析表达式：

```
> f := unapply( int( 1/(1+x^4), x ), x );
      f := x →  $\frac{1}{8}\sqrt{2} \ln\left(\frac{x^2+x\sqrt{2}+1}{x^2-x\sqrt{2}+1}\right) + \frac{1}{4}\sqrt{2} \arctan(x\sqrt{2}+1) + \frac{1}{4}\sqrt{2} \arctan(x\sqrt{2}-1)$ 
```

用 fortran 命令可以将这个函数转换为 Fortran 子程序:

```
> fortran(f, optimized);
c The options were      : operatorarrow
  real function f(x)
  real x

  real t1
  real t12
  real t16
  real t2
  real t3
  real t8

  t1 = sqrt(2.E0)
  t2 = x**2
  t3 = x*t1
  t8 = alog((t2+t3+1)/(t2-t3+1))
  t12 = atan(t3+1)
  t16 = atan(t3-1)
  f = t1*t8/8+t1*t12/4+t1*t16/4
  return
end
```

用 C 命令则可以将它转换成等价的 C 程序:

```
[> readlib(C):
> C(f, optimized);
/* The options were      : operatorarrow */
#include <math.h>
double f(x)
double x;
{
  double t1;
  double t2;
  double t3;
  double t8;
  double t12;
  double t16;
  {
    t1 = sqrt(2.0);
    t2 = x*x;
    t3 = x*t1;
    t8 = log((t2+t3+1.0)/(t2-t3+1.0));
    t12 = atan(t3+1.0);
    t16 = atan(t3-1.0);
    return(t1*t8/8.0+t1*t12/4.0+t1*t16/4.0);
  }
}
```

调用中的第二个参数 `optimized` 是可选参数, 表示对要转换的表达式进行优化; 如果不给定, 则直接对表达式进行一一对应的转换。

9.6.2 生成 LATEX 或 eqn

LATEX 和 eqn 是两种国际上通用的格式描述语言。如果要在国外期刊上发表论文，将公式转换成为标准的描述语言是必须的。

在 Maple 中，可以用命令 `latex` 和 `eqn` 分别完成从 Maple 表达式到这两种描述语言的转换。由于 `eqn` 是一个常用的符号，所以 Maple 将转换命令 `eqn` 放在库中，调用前需要用 `readlib` 载入。这两个命令的调用格式如下：

latex (*expression*, *fileName*)

eqn (*expression*, *fileName*)

其中，*expression* 是 Maple 表达式，但对于 Maple 所特有的表达式，例如子程序等等，有可能无法转换。可选参数 *fileName* 指定了输出的文件名（必须尚未被打开），如果不提供，将输出到默认的输出流（窗口）中。

Maple 可以自动完成大多数数学表达式的转换，包括积分、极限、求和号、连乘积和矩阵等。如果要加以扩展可以自定义函数 `latex/functionName` 和 `eqn/functionName`，则 Maple 在转换函数 *functionName* 时，将调用它们，它们必须用 `printf` 定义转换后的输出结果。

下面的例子将一个不定积分及其结果的表达式分别转换成这两种描述语言。注意在不要求值时，用于延迟求值的形式函数（Int）的使用。

```
> expression := Int(1/(x^4+1), x) = int(1/(x^4+1), x);
expression := 
$$\int \frac{1}{1+x^4} dx =$$


$$\frac{1}{8}\sqrt{2} \ln\left(\frac{x^2+x\sqrt{2}+1}{x^2-x\sqrt{2}+1}\right) + \frac{1}{4}\sqrt{2} \arctan(x\sqrt{2}+1) + \frac{1}{4}\sqrt{2} \arctan(x\sqrt{2}-1)$$

> latex(expression);
\int \!\! \left( 1+(x)^{4} \right)^{-1} {dx}=1/8\,\sqrt{2}\ln\left(\frac{(x)^{2}+x\sqrt{2}+1}{(x)^{2}-x\sqrt{2}+1}\right)+1/4\,\sqrt{2}\arctan(x\sqrt{2}+1)+1/4\,\sqrt{2}\arctan(x\sqrt{2}-1)
> readlib(eqn);
> eqn(expression);
{{int { (( {1 ^{ "x" sup 4 } }) sup -1 )~d "x" )~~~{ ( {
sqrt 2 }^{ln ( { ( { "x" sup 2 }^{ "x" ^{ sqrt 2 } })^{+1 } over { (
"x" sup 2 }^{-{ "x" ^{ sqrt 2 } })^{+1 } } ) } over 8 }^{+{ ( { sqrt 2 }^{
arctan ( { ( "x" ^{ sqrt 2 } )^{+1 } } ) } over 4 }^{+{ ( { sqrt 2 }^{
arctan ( { ( "x" ^{ sqrt 2 } )^{-1 } } ) } over 4 } ) } }
```

9.6.3 字符串和整数之间的转换

在 9.4.2 和 9.5.5 中，我们介绍过 `readbytes` 和 `writebytes` 命令，它们输入/输出的对象，可以是字符串，也可以是一个整数的有序表。字符串和一系列整数之间是相互对应的，字符串中每一个字符的 ASCII 码分别对应着整数有序表中的一个整数。用 Maple 的 `convert` 命令，可以完成这两者之间的转换，命令的调用格式如下：

convert (*toBeConverted*, *bytes*)

如果第一个参数 *toBeConverted* 为字符串，则返回的是对应的整数有序表；如果给定整数有序表作为输入参数，则返回对应的字符串。

```
[> convert("Test String", bytes);
      [84, 101, 115, 116, 32, 83, 116, 114, 105, 110, 103]
```

不过，整数 0 在字符串中对应的字符，所以，如果一个有序表中包含整数 0，则转换的字符串到 0 截止。这和 C 语言中将 0 作为字符串的结束符有类似之处。

```
[> convert([84, 101, 115, 116, 0, 83, 116, 114, 105, 110,
           103], bytes);
      "Test"
```

9.6.4 从字符串中获得 Maple 表达式

命令 `parse` 可以对 Maple 字符串进行语法分析，并将符合语法的字符串转换成对应的 Maple 表达式。该命令的调用格式如下：

parse (*string*, *options*)

参数 *string* 是有待分析的字符串；*options* 为两个可选参数 `statement` 和 `nosemicolon`。如果指定 `statement`，则 Maple 对于获得的 Maple 语句将作求值运算，否则，在默认情况下，Maple 仅仅进行形式上的化简，而不进行求值。在一般情况下，Maple 对于不完整的字符串（缺少分号或冒号），会在语句末尾加入分号。但如果指定了 `nosemicolon`，则 Maple 不给分析的字符串的末尾加入语句结束符，以便于分析多个语句。

```
[> parse("a+2+b+3");
      a + 5 + b
[> parse("sin(0)");
      sin(0)
[> parse("sin(Pi/2)", statement);
      1
```

9.6.5 对于字符串的格式化输入和输出

Maple 命令 `sscanf` 与 `sprintf` 是用来向字符串进行格式化输入/输出的，它们的使用方法和调用格式和 `fscanf` 与 `fprintf` 相似，唯一不同的是，它们的输入/输出是面向字符串，而不是面向文件的。它们的调用格式如下：

sprintf (*format*, *expressionSequence*)

sscanf (*sourceString*, *format*)

`sprintf` 返回的是将数据 *expressionSequence* 格式化后的字符串，而 `sscanf` 返回扫描所得的数据对象的有序表。

```
[> s := sprintf("%4.2f %a %m", evalf(Pi), sin(3), cos(3));
      s := "3.14 sin(3) -%$cosG6#\\"$
[> sscanf(s, "%f %a %m");
      [3.14, sin(3), cos(3)]
```

9.7 调用 Matlab 函数

Matlab 是目前最为流行的数值分析软件之一，通为数学软件，它和 Maple 各有所长。为了互相补充，它们之间都有借鉴的地方——Matlab 的符号运算用的是 Maple 的计算内核，Maple 中也可以调用 Matlab 函数进行快速的浮点数值处理。

如果你的系统中除了 Maple 之外，还安装了 Matlab 软件，那么就可以在 Maple 中通过 Matlab 软件包调用 Matlab 的函数。由于本书的主要介绍对象是 Maple，所以这里对于 Matlab 函数不作详细的介绍。

首先，要调用 Matlab 函数，需要在 Maple 工作环境中与 Matlab 建立连接。用 Matlab 工具包中的函数 `openlink` 可以完成这一功能，调用成功后，Matlab 就在后台运行了，并且和在前台运行的 Maple 建立了联系。

要调用 Matlab 函数，还需要用 `with` 命令载入 Matlab 工具包。

建立了这样的环境之后，你可以分别在两个系统中定义变量——Maple 和 Matlab。Maple 中定义和引用变量的方法和通常一样。但是 Matlab 的变量，在 Maple 中都必须以变量名的字符串形式进行引用。利用 Matlab 工具包中的函数 `setvar` 可以定义 Matlab 变量——所有的 Matlab 变量都是是硬件浮点数组。

Matlab 的长处在于它强大的矩阵运算功能。在 Maple 中可以调用 Matlab 中常用的函数，这里将这些函数列成表 9.1，具体的用法请参见 Maple 中有关 Matlab 工具包的在线帮助。

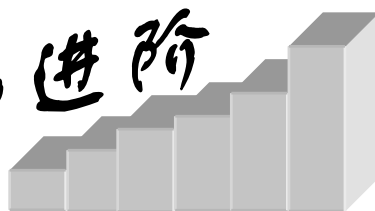
表 9.1 Maple 中可以调用的 Matlab 函数

名称	用途
<code>chol</code>	矩阵的 Cholesky 分解
<code>det</code>	矩阵的行列式
<code>eig</code>	求矩阵的特征值和特征向量
<code>fft</code>	快速富利叶变换
<code>inv</code>	矩阵求逆
<code>lu</code>	矩阵 LU 分解
<code>ode45</code>	常微分方程数值解（4-5 阶龙格-库塔法）
<code>qr</code>	矩阵的 QR 分解
<code>transpose</code>	矩阵转置

这些函数的返回值都是 Maple 的硬件浮点数组。但它们的参数，可以是 Maple 环境中的变量，也可以使用后台的 Matlab 变量（必须用一对双引号将变量名括起来，作为字符串传递给函数）。如果传递的参数是 Maple 数组，Maple 将首先将数组中的元素全部转化为硬件浮点数。

如果和 Matlab 的通讯结束，需要中断联系，可以用命令 `closelink` 关闭 Maple 和 Matlab 之间的连接。

起步与进阶



第

十

章

程序的调试

任何语言编写的程序都不能保证没有错误，Maple 也一样。有时候错误非常隐蔽，仅仅通过检察源程序或者数值试验可能无法排除程序中错误。Maple 中提供了一些实用的调试工具，本章将介绍利用这些调试工具调试 Maple 程序的方法。

本章具体包括以下内容：

- 🕒 显示子程序中的语句
- 🕒 设置断点
- 🕒 在运行中察看、设置变量的值
- 🕒 条件断点的设置
- 🕒 程序运行流程的控制

利用 Maple 内建的调试工具，可以在一个 Maple 程序的内部中断程序的运行，察看和修改局部变量和全局变量的值，并且可以单步运行程序。可以在程序的特定语句中断程序的运行，也可以在某个特定变量被赋值时中断程序，甚至可以在一个特定的错误发生时中断程序。Maple 的调试工具，提供了一个监视程序的内部执行过程的手段，利用它，可以使用户更快地找到并修改程序的错误。

10.1 调试的例子

在这一节中，将通过例子介绍 Maple 中的各个调试工具的用法。这里将接触到许多调试命令，对于它们的详细介绍，将在本章后面几节中进行。这一节将围绕下面的例子进行。

```
> sieve := proc(n::integer)
    local i, k, flags, count, twice_i;
    count := 0;
    for i from 2 to n do flags[i] := true od;
    for i from 2 to n do
        if flags[i] then
            twice_i := 2*i;
            for k from twice_i by i to n do
                flags[k] = false;
            od;
            count := count + 1
        fi;
    od;
    count;
end;
```

Maple 中的大部分调试命令和具体的语句有关。但是，子程序中的语句并不一定与行号一一对应。利用命令 `showstat`，可以显示子程序具体的语句与每一个语句编号的关系。

```
> showstat(sieve);

sieve := proc(n::integer)
local i, k, flags, count, twice_i;
  1   count := 0;
  2   for i from 2 to n do
  3       flags[i] := true
      od;
  4   for i from 2 to n do
  5       if flags[i] then
  6           twice_i := 2*i;
  7           for k from twice_i by i to n do
  8               flags[k] = false
              od;
  9           count := count+1
              fi
          od;
 10   count
end
```

这个程序是 Eratosthenes 筛法的一个实现，给定一个 n 作为参数，程序会返回所有小于 n 的素数个数。这个程序中存在着一定的错误，这里通过 Maple 的调试工具来找出这些错误。

要使 Maple 的调试器工作，首先需要运行有待调试的程序，并且要在程序中间中断程序的运行。运行程序，可以在 Maple 的交互式命令环境中直接调用，也可以是由别的程序调用。在程序内部中断程序的最简单方法是对程序设置断点。利用 `stopat` 命令就可以做到。

```
[> stopat(sieve);
                                     [sieve]
```

这样就在 `sieve` 函数的第一条语句前设置了断点。命令 `stopat` 的返回值是所有设置了断点的子程序名。在调用 `sieve` 函数时，Maple 会自动地在它的第一条语句之前停止执行。然后进入调试状态。

```
[> sieve(10);
sieve:
  1*   count := 0;
[DBG>
```

在 Maple 的调试提示符 “DBG>” 之前，会给出以下几部分的信息：

- ✧ 上一步的计算结果（在这个例子中，断点设在所有的执行语句之前，所以没有给出上一步的结果）。
- ✧ 正在调试的子程序名（这里是 “sieve”）。
- ✧ 从断点开始的下一条语句（以语句编号为准，这里是 “1”）。语句编号后面的星号 “*” 表示该语句之前设有断点。
- ✧ 对于复杂结构，例如循环和判断语句，Maple 不写出执行体中的具体语句，而用 “...” 表示。

在调试状态中，可以对特定的表达式赋值求值，也可以调用 Maple 的调试命令。Maple 对表达式求值时，所在的环境以中断的程序内部为准。所有的参数、局部变量、或者全局变量，都是依据所在的程序而定的。例如，这里调用 `sieve` 所给定的参数是 10，所以这时 n 的值为 10：

```
[DBG> n
10
sieve:
  1*   count := 0;
```

在程序调试环境中，可以用调试命令控制程序的执行。最常用的调试命令应该算是 `next` 了，它将执行下一条语句（就是显示在提示符前的一条语句），并在下一条语句运行后中断，返回调试状态。

```
[DBG> next
0
sieve:
  2   for i from 2 to n do
      ...
  od;
```

这里就可以看到上一步的运算结果 0 (“count := 0;” 的结果)。而且，由于第二条语句

前没有设置断点，所以标号后面没有星号。具体的循环体被省略了，只用省略号表示。

继续用命令 `next`，得到这样的显示：

```
DBG> next
true
sieve:
  4      for i from 2 to n do
          ...
        od;
```

`next` 命令跳过了语句 3（并不是没有执行，而是没有在语句 3 前面中断），因为语句 3 是更深层次的语句，它在语句 2（循环语句）的内部，`next` 命令不进入深层进行调试。但是，调试器还是给出了深层语句的计算结果：`true`。如果要进入深层语句进行调试，可以用命令 `step`：

```
DBG> step
true
sieve:
  5      if flags[i] then
          ...
        fi

DBG> step
true
sieve:
  6      twice_i := 2*i;
```

如果在一个简单语句（没有深层结构的语句）前使用 `step`，其结果与 `next` 相同。

```
DBG> step
4
sieve:
  7      for k from twice_i by i to n do
          ...
        od;
```

利用 `step` 命令进入到程序循环中的最深层语句：

```
DBG> step
4
sieve:
  8      flags[k] = false
```

我们知道，利用 `showstat` 命令，可以显示当前正在调试中的子程序的所有语句。但是，如果程序比较长，显示所有的语句并不利于察看。

这时，可以采用 `list` 命令，它可以显示当前中断位置的前 5 条语句（按照语句编号计算）和下一步要执行的语句。这可以提供源程序信息，并且可以使调试用户清楚地知道当前所处的位置。

和 `showstat` 的显示一样，`list` 显示的程序代码中，对于设有断点的语句号，在旁边附有星号；而对于当前中断的位置，将用感叹号“!”加以标出：

```

DBG> list

sieve := proc(n::integer)
local i, k, flags, count, twice_i;
    ...
3     flags[i] := true
      od;
4     for i from 2 to n do
5         if flags[i] then
6             twice_i := 2*i;
7             for k from twice_i by i to n do
8 !         flags[k] = false
              od;
9             count := count+1
            fi
          od;
        ...
end

```

和 `step` 相对，命令 `outfrom` 可以从当前中断的位置继续执行，直到跳出该语句层到达更浅层的语句为止。

```

DBG> outfrom
true = false
sieve:
9     count := count+1

DBG> outfrom
1
sieve:
5     if flags[i] then
      ...
      fi

```

调试命令 `cont` 可以使当前中断的子程序继续连续执行，直到正常结束或者遇到新的断点为止：

```

DBG> cont

9!

```

由于没有遇到其他的断点，程序执行完毕，但给出的结果显然不是我们所期望的。也许在前面的调试中，你早已经发现错误所在。但在实际的复杂程序中，调试远没有这么简单。所以，为了进一步介绍调试命令，这里假设我们尚未发现错误所在。

在开始新的调试之前，先删除原来设定的 `sieve` 程序开始所定义的断点，删除断点所用的命令是 `unstopat`，可以看到，`unstopat` 执行后断点有序表为空：

```

[> unstopat(sieve);

[ ]

```

从上面的我们发现，程序的错误最直接的来源是对于变量 `count` 赋值的结果，为了进行更细致的调试，我们利用命令 `stopwhen` 对于 `sieve` 的局部变量 `count` 设置监视断点——在 `count` 的值变化时，程序将中断执行。命令 `stopwhen` 的返回值是当前所有被监控的变量的有序表：

```
[> stopwhen([sieve, count]);
                                     [[sieve, count]]
```

再执行 sieve 程序，调试器使程序在第一条语句后中断，因为第一条语句对 count 赋了初值 0。对于监视断点，中断时将显示对应的赋值语句而不是上一步的计算结果：

```
[> sieve(10);
count := 0
sieve:
  2    for i from 2 to n do
      ...
      od;
```

显然，这一赋值语句没有错误，我们继续执行程序：

```
[DBG> cont
count := 1
sieve:
  5    if flags[i] then
      ...
      fi
```

如果不太仔细的话，这里的错误也很难发现，所以我们继续程序的执行：

```
[DBG> cont
count := 2*1
sieve:
  5    if flags[i] then
      ...
      fi
```

到这里，奇怪的结果出现了，我们希望 count 被赋值为 2，可以这里却将它赋成了 2*1，其中的“1”是字母“L”的小写，而不是数字 1。检察源程序，不难发现问题出在行号为 9 的语句 `count := count + 1` 中。既然发现了错误，我们就终止调试，改正错误（修改程序并不需要重新键入，只需在源程序上修改，再按回车确认就可以了）：

```
[DBG> quit
Warning, computation interrupted
```

改正了错误后，首先用 `unstopwhen` 解除监视中断，重新运行程序，可以结果仍然不对：

```
[> unstopwhen();
                                     [ ]
[> sieve(10);
                                     9
```

小于 10 的素数一共有 4 个：2, 3, 5, 7；但 sieve 程序返回的却是 9。让我们再对程序进行一定的调试，由于程序的前面部分没有错误，这次在 sieve 的第 6 条语句设置断点。

```
[> stopat(sieve, 6);
                                     [sieve]
[> sieve(10);
true
sieve:
  6*    twice_i := 2*i;
```

```

DBG> step
4
sieve:
  7      for k from twice_i by i to n do
          ...
          od;

DBG> step
4
sieve:
  8      flags[k] = false

DBG> step
true = false
sieve:
  8      flags[k] = false

```

经过几次单步执行后，我们找到了错误所在。语句 8 的用途是给标志数组 `flags` 的第 `k` 个元素赋值 `false`，但却得到了一个不成立的等式 `true = false`——原来是将赋值语句误写成了一个等式。退出调试环境，将源程序修改为：

```

DBG> quit
Warning, computation interrupted
> sieve := proc(n::integer)
    local i, k, flags, count, twice_i;
    count := 0;
    for i from 2 to n do flags[i] := true od;
    for i from 2 to n do
        if flags[i] then
            twice_i := 2*i;
            for k from twice_i by i to n do
                flags[k] := false;
            od;
            count := count + 1
        fi;
    od;
    count;
end:

```

这次，程序 `sieve` 给出了我们所期望的结果：

```

> sieve(10);
4

```

10.2 使用 Maple 的调试器

在前一节中，我们通过一个例子简单介绍了 Maple 程序的调试方法和调试过程。接下来，将对 Maple 的各种调试命令进行系统的介绍。为使 Maple 的调试器工作，可以有三种途径——设置无条件断点、监视断点、或者出错断点。

10.2.1 显示程序的语句

再设置断点以前,有必要知道具体语句对应的语句编号。命令 `showstat` 可以显示源程序,语句对应的编号,断点的设置,以及当前的中断位置。它的调用格式如下:

`showstat (procedure, number)`

其中 *procedure* 是需要显示的子程序名称。对于无条件断点, `showstat` 将在其语句编号后显示星号 “*”; 对于条件断点, 则显示问号 “?”。

对于命令 `showstat`, 还可以加入可选参数 *number*, 指定语句号或者语句号的范围。这时, 将只显示所指定的语句, 其余的语句都用 “...” 表示。

`showstat` 命令不仅可以在交互式环境中使用, 也可以作为调试命令在调试环境中使用。在调试环境中, 使用的格式为:

`showstat procedure number`

参数的意义和在外使用相同时, 只不过这里参数 *procedure* 如果是指当前正在调试的程序, 则可以省略。对于当前中断的语句编号, 将在数字后面加感叹号 “!” 表示。

需要注意的是, 常用的调试命令, 在调试环境中的使用与在外部交互式环境中的使用略有不同。在调试环境中, 一般无需指出具体的程序名称, 也不用将参数用括弧括起来。这里的 `showstat` 命令就是一个例子, 对于其他的调试命令也一样, 下面出现时就不再予以一一指出了。

10.2.2 断点

设置断点是使程序中断在特定位置, 从而启动调试器的最简单的办法。在 Maple 中, 可以用命令 `stopat` 设置断点, 它的调用格式为:

`stopat (procedure, number, condition)`

其中 *procedure* 是子程序名称; *number* 是需要设置中断的语句编号, 它可以省略, 默认情况下将在该程序的第一条语句之前设置断点; *condition* 是在该断点处中断的条件, 它是一个布尔表达式, 可以包含任意全局变量, 该子程序中的局部变量和参数, 在省略这一参数时, `stopat` 将设置无条件断点。无条件断点的语句号后面将用星号标记, 而条件断点则标记以问号。

`stopat` 命令也可以在调试环境下使用, 使用格式为:

`stopat procedure number condition`

需要注意的是, `stopat` 设置的断点是在语句号为 *number* 的语句之前, 所以, 我们将无法在程序的最后一句语句之后设置断点。对于两个完全相同的程序, Maple 中设置断点是否相同, 将取决于它们是否为相互赋值的结果。换句话说, 如果两个程序各自分别定义, 则它们的断点相互无关; 如果用一个程序为另一个变量赋值, 则它们的断点相同, 例如:

```
[> f := proc(x) x^2 end:
[> g := proc(x) x^2 end:
[> h := op(f):
[> stopat(f);

[f, h]
```


可以用 `unstopat` 来清除程序中的断点，它的调用格式如下：

`unstopat (procedure, number)`

这里的语句编号 *number* 也是可选参数，如果省略，将删除该程序中所有的断点。这个命令也可以在调试环境内部使用：

`unstopat procedure number`

10.2.3 显式的断点

在用户自己编写的源程序中，也可以用调用函数 `DEBUG` 的方法加入一个显式的断点：

`DEBUG (condition)`

其中 *condition* 是一个布尔表达式，表示中断的条件，在它的值为 `true` 时引起中断，如果为 `false` 或者 `FAIL` 时将忽略这一中断语句；它是可选参数，默认情况下将导致无条件中断。

`DEBUG` 的参数也可以是非布尔类型的表达式，这使程序将无条件中断，并且显示这个表达式的值。

显式断点的调用结果和调试断点的结果相同，都是使程序中断，并显示上以执行语句的结果以及下一条将要执行的语句。

```
[> f := proc(x)
      DEBUG( "my breakpoint, current value of x: ", x );
      x^2
    end:
> f(5);
"my breakpoint, current value of x: "
5
f:
  2      x^2
[DBG>
```

它们的不同点是，显式断点在用 `showstat` 显示时，语句编号后面并没有任何标记：

```
[DBG> showstat

f := proc(x)
  1   DEBUG("my breakpoint, current value of x: ",x);
  2   !  x^2
end
```

而且，`unstopat` 命令也不能将显式断点清除：

```
[DBG> unstopat 1
Error, no break point set at specified location
[DBG> quit
Warning, computation interrupted
```

但是，对于用 `stopat` 加入的断点，如果用 `print` 或者 `lprint` 命令进行输出，对应的语句将显示为显式的中断——对 `DEBUG` 的调用：

```
[> f := proc(x) x^2 end:
> stopat(f);

[']
```

```
[> print(f);  
proc(x) DEBUG( ); x^2 end
```

10.2.4 监视断点

顾名思义，监视断点可以对变量进行监视，在对变量进行赋值后，程序将中断运行，进入调试状态。程序中的监视断点可以用命令 `stopwhen` 加入，它有两种调用格式：

stopwhen (globalVariableName)

stopwhen [procedure, variableName]

第一种调用设置了对全局变量 `globalVariableName` 的监视，还可以用它来设置对环境变量的监视；第二种调用则在程序 `procedure` 内部监视变量 `variableName`（可以为全局变量或者局部变量）。

在 Maple 对所监视的变量进行赋值以后，监视断点就中断程序的运行，并显示对应的赋值语句（对于所赋的值已进行化简），而不是显示结果。然后，依照惯例显示下一条语句。需要注意的是，监视断点在赋值之后引起中断，而不是赋值以前。

在调试环境中也可以使用 `stopwhen` 命令设置监视断点：

stopwhen globalVariableName

stopwhen [procedure, variableName]

要清除监视断点，可以调用命令 `unstopwhen`，它的参数和 `stopwhen` 完全一样，也可以在调试环境中调用。如果不提供任何参数，`unstopwhen` 将清除所有的监视断点。

10.2.5 出错断点

出错断点可以截获 Maple 的错误信息，并引起程序中断，进入调试状态。命令 `stoperror` 可以设置出错中断，它的调用格式如下：

stoperror ("errorMessage")

`errorMessage` 指定了需要截获的 Maple 出错信息，也可以使用 `all` 作为参数，以截获所有的错误。`stoperror` 的返回值是所有设置出错断点的错误有序表：

```
[> stoperror("division by zero");  
[division by zero]
```

我们在上一章中曾经涉及到命令 `traperror` 的使用，它可以截获出错信息（但不引发中断），而使 `stoperror` 对错误设置断点无效。为了截获已经被 `traperror` 截获的错误，可以用 `traperror` 作为 `stoperror` 的参数。

命令 `stoperror` 也可以在调试状态下使用，它的调用格式如下（无须使用双引号）：

stoperror errorMessage

除了用 `all` 或者 `traperror` 作为 `stoperror` 的参数外，`traperror` 还支持以下几种出错信息：

- ✧ ``interrupted`` 在用户终止程序运行（用 `Ctrl-C` 或者 `Ctrl-Break`）时引发中断。
- ✧ ``time expired`` 在用函数 `timelimit` 限制一个语句或程序的执行时间时，程序运行超时将引发中断。
- ✧ ``assertion failed`` 在用 `ASSERT` 命令检查断言时，发现断言不正确将引发中断。

- ◇ `invalid arguments` 在调用函数或子程序时，如果参数类型不符或者参数不足将引发中断。

利用命令 `unstoperror` 可以清除出错断点，它的参数与 `stoperror` 相同。如果不给定具体参数，`unstoperror` 将清除所有的出错断点。

在 `unstoperror` 引起的断点中，将不能用 `cont`、`next`、或者 `step` 命令继续程序的执行。例如在下面的例子中，我们将截获一个除零错误`division by zero`。

```
[> unstoperror():
> f := proc(x) 1/x end:
> g := proc(x)
    local r;
    r := traperror(f(x));
    if r = lasterror then infinity
    else r
    fi
end:
```

对于函数 `f`，如果用 0 作为参数将引发除零错误。但是在 `g` 对 `f` 的调用中，它用 `traperror` 截获了错误，使其返回无穷大：

```
[> g(0);
                                     ∞
```

不过，如果直接调用 `f`，还是会引发错误。如果用 `stoperror` 设置了出错断点，还将引起程序的中断：

```
[> stoperror("division by zero"):
> f(0);
Error, division by zero
f:
  1    1/x
```

此时，纵使在调试状态，程序仍然不能继续执行：

```
[DBG> cont
Error, (in f) division by zero
```

但是，如果调用 `g` 则不能引发错误中断。我们再用 `traperror` 作为参数设置出错断点：

```
[> unstoperror("division by zero"):
> stoperror("traperror"):
```

这时，直接调用 `f` 将不触发出错中断，也不会进入调试状态：

```
[> f(0);
Error, (in f) division by zero
```

但如果调用 `g`，将因为 `traperror` 而触发中断，进入调试状态：

```
[> g(0);
Error, division by zero
f:
  1    1/x
```

10.3 系统状态的检查和设置

10.3.1 变量值的显示和修改

在程序在调试状态下中断时，可以检查系统中所有的全局变量，当前的局部变量；还可以对用这些变量组成的表达式求值。

在调试状态下求值，只需要在调试提示符后面直接输入要求值的表达式即可：

```
[> f := proc(x) x^2 end:
[> stopat(f):
[> f(10);
f:
  1*   x^2

DBG> sin(3.0)
.14111200081
f:
  1*   x^2

DBG> x
10
f:
  1*   x^2
```

在中断时，所有的变量都按照当前的系统环境求值，也就是中断前各个变量的值。

如果表达式中恰好包含有与调试命令同名的变量，还是可以对它们进行求值，只需要用一对圆括弧将该变量扩起来就可以了。例如：

```
[> f := proc(step)
  local i;
  for i to 10 by step do i^2 od;
end:
[> stopat(f, 2):
[> f(3);
f:
  2*   i^2

DBG> step
1
f:
  2*   i^2

DBG> (step)
3
f:
  2*   i^2

DBG> quit
Warning, computation interrupted
```

在程序运行中断时，可以用赋值运算符修改全局变量或者局部变量的值。

为了演示在调试环境中对变量的修改，在下面的例子中，我们将在程序的循环中当循环变量 i 的值为 5 时中断程序。

```
[> sumn := proc(n)
    local i, sum;
    sum := 0;
    for i to n do sum := sum + i od;
end:
> showstat(sumn);

sumn := proc(n)
local i, sum;
  1   sum := 0;
  2   for i to n do
  3     sum := sum+i
      od
end
```

用 `stopat` 在第 3 条语句前设置条件断点：

```
[> stopat(sumn, 3, i=5);
                                     [f, sumn]
> sumn(6);
10
sumn:
  3?   sum := sum+i
```

这时，如果我们将循环变量再设置成 3，则当程序重新运行到 $i=5$ 时，又将引发中断。

```
[DBG> i := 3
sumn:
  3?   sum := sum+i

DBG> cont
17
sumn:
  3?   sum := sum+i
```

这时的和 `sum` 值为 17，是 1、2、3、4、3、4 这 6 个数的和，如果再运行下去直到程序结束，加上整数 5、6 则总和应为 28——可以看到，调试过程改变了程序的运行结果。

```
[DBG> cont
                                     28
```

10.3.2 程序运行状态的察看

Maple 提供了两个调试命令可以察看程序运行的状态。命令 `list` 可以显示程序当前运行的源程序和中断所在的位置；而命令 `where` 则可以显示程序的活动堆栈。命令 `list` 的调用格式如下：

list procedure statementNumber

其中 *procedure* 和 *statementNumber* 分别是程序名和需要显示的语句编号。命令 `list` 与 `showstat`

很相似，除了不提供语句号的时候。如果不指定具体的语句编号，list 将显示前 5 条语句，当前语句以及下一条语句。

命令 **where** 将显示程序的活动堆栈，从最顶层的程序开始，它依次显示所有的子程序调用，以及调用时所传递的参数值，直到当前中断所在的程序。也可以指定最多显示的堆栈层数，调用格式如下：

where numLevels

作为例子，下面的程序中调用了前面的子程序 **sumn**。子程序 **sumn** 中包含一个条件断点，在 **check** 调用 **sumn** 并运行到断点时，Maple 进入调试状态。

```
> check := proc(i)
    local p, a, b;
    p := ithprime(i);
    a := sumn(p);
    b := p*(p+1)/2;
    evalb( a=b );
end:
> check(10);
10
sumn:
 3?      sum := sum+i
```

这时，如果调用调试命令 **where**，将显示顶层的调用——**check** 参数为 9，以及 **check** 中对 **sumn** 的调用——参数为 29：

```
DBG> where
TopLevel: check(10)
[10]
check: a := sumn(p)
[29]
sumn:
 3?      sum := sum+i

DBG> cont
true
```

命令 **where** 在调试递归程序时非常有用，下面通过一个简单的例子介绍在递归程序中使用 **where** 命令进行调试的方法。这是一个通过递归求阶乘的程序：

```
> fact := proc(x)
    if x<=1 then 1
    else x * fact(x-1) fi;
end:
> showstat(fact);

fact := proc(x)
 1   if x <= 1 then
 2       1
    else
 3       x*fact(x-1)
    fi
end
```

为了在底层的递归调用中中断，我们在第 2 条语句中设置断点，并运行程序：

```
[> stopat(fact, 2):
> fact(5);
fact:
  2*    1
```

不加参数执行命令 `where`，将显示所有的活动堆栈：

```
DBG> where
TopLevel: fact(5)
[5]
fact: x*fact(x-1)
[4]
fact: x*fact(x-1)
[3]
fact: x*fact(x-1)
[2]
fact: x*fact(x-1)
[1]
fact:
  2*    1
```

如果并不对所有的调用堆栈感兴趣，可以加参数指明需要显示的堆栈层数：

```
DBG> where 3
fact: x*fact(x-1)
[2]
fact: x*fact(x-1)
[1]
fact:
  2*    1

DBG> quit
Warning, computation interrupted
```

10.3.3 显示断点信息

调试命令 `showstop` 可以显示所有已经设置的断点，包括一般断点，监视断点，以及出错断点。在调试环境之外，该命令的调用格式如下：

`showstop()`

如果再调试环境内部调用，则无需括弧。这里通过一个例子程序来说明它的使用方法，以及它显示的结果。首先编写一个简单的程序：

```
[> f := proc(x)
  local y;
  if x<2 then
    y := x;
    print(y^2);
  fi;
  print(-x);
  x^3;
end:
```

再加入一些断点：

```
[> stopat(f):
> stopat(f, 2):
> stopat(int);

[f, fact, int, sumn]
```

设置一些监视断点和出错断点:

```
[> stopwhen(f, y):
> stopwhen(Digits);

[Digits, [f, y]]
> stoperror("division by zero");

[division by zero]
```

命令 `showstop` 将分类显示所有的断点, 包括我们在前面例子中设置的断点:

```
> showstop();

Breakpoints in:
  f
  fact
  int
  sumn

Watched variables:
  Digits
  y in procedure f

Watched errors:
  `division by zero`
```

10.3.4 程序执行的控制

在程序中断并进入调试状态时, 可以用一系列调试命令用不同的方式继续程序的运行。由于在本章第一节中已经有比较详细的例子, 这里就不再展开进行详细介绍了。这一节中介绍的命令都只能在调试环境下使用, 因为只是用于控制程序的执行, 它们不需要附加任何参数。

在程序中断的调试状态下, 命令 `quit` 可以终止当前程序的运行, 并退回到交互式环境中; 命令 `cont` 可以继续当前程序的运行, 直到运行到新的断点或者程序正常结束为止。

命令 `next` 可以执行当前位置以下的一条语句, 然后中断程序。如果该语句是一个程序控制结构 (例如循环结构或者分支结构), `next` 将执行整个结构。同样, 如果该语句是一个子程序或者函数的调用, `next` 语句也将运行完整的一条语句, 而不进入子程序内部调试 (当然, 如果子程序内部有断点, 就另当别论了)。如果当前所在语句是执行过程中最后一条语句, `next` 命令在执行完该语句后, 将退出调试状态。

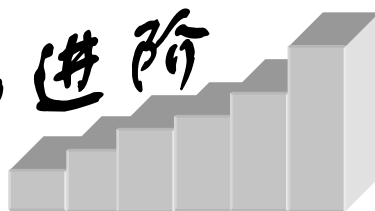
命令 `step` 也将执行当前位置的下一条语句, 和 `next` 不同的是, 如果下一条语句是一个控制结构或者是一个函数调用, `step` 都将进入结构或者子程序内部, 执行结构或子程序中的第一条语句, 然后中断到调试状态。

而 `into` 命令则介于 `step` 和 `next` 之间, 它对于程序控制结构将进入调试, 但对于子程序调用则采取与 `next` 一样的办法, 执行完整的语句, 并不进入调用的子程序内部调试。

命令 `outfrom` 可以说是 `step` 的逆命令，它从当前的中断位置开始执行，直到跳出当前所在的层（子程序调用或者程序结构）为止。如果已经是最顶层，执行完该命令后将退出调试状态。

调试命令 `return` 可以从当前中断为止继续执行，直到跳出正在运行的子程序调用为止。和 `outfrom` 不同，它在跳出当前的控制结构后并不中断，而要等到该子程序执行完后才终止。

起步与进阶



第

十

一

章

欧氏几何学

从这章开始,将分别介绍 Maple 中的几个常用的工具包。本章将围绕着 Maple 中的两个几何工具包,向大家介绍如何利用 Maple 解决欧氏空间中的几何问题。由于 Maple 擅长的是符号演算,所以它所能解决的问题,大多是解析几何问题。

本章具体包括以下内容:

- 🕒 如何构造平面几何对象
- 🕒 如何获取平面几何对象的属性
- 🕒 平面几何对象之间的相互关系
- 🕒 如何对平面几何对象进行几何变换
- 🕒 如何建立空间几何对象
- 🕒 空间几何对象属性的获取
- 🕒 空间几何对象之间的位置关系
- 🕒 空间几何对象的变换
- 🕒 如何进行空间变换的谓词演算

欧氏几何学是数学中的一个重要分支，它是研究欧氏空间中的抽象几何形体的性质，以及形体之间相互关系的学科。由于几何问题具有它的直观性，所以早在远古时代，数字诞生以前，人类就凭着对自然界的感官认识，对几何问题有所研究。代数和微积分的发展，为几何的进一步发展提供了有利的工具。Maple 中也可以处理初等的几何学问题，它采用的方法是解析几何的方法。

在 Maple 中有两个欧氏几何的工具包——`geometry` 和 `geom3d`。其中，`geometry` 是专门用来解决平面几何问题的，而 `geom3d` 是三维空间几何的工具包。我们将在这一章中分别进行详细的介绍。

11.1 平面几何对象

Maple 的平面几何工具包 `geometry` 中，支持 11 种平面几何对象，它们分别是：点(`point`)，线段(`segment`)，有向线段(`directed segment`)，直线(`line`)，三角形(`triangle`)，正方形(`square`)，园(`circle`)，抛物线(`parabola`)，椭圆(`ellipse`)，双曲线(`hyperbola`)，二次曲线(`conic`)。在这一节中将逐一介绍它们的使用。

11.1.1 点、线段和直线

建立一个点对象，需要调用函数 `point`。建立点对象的时候，需要指定点对象的名称，以及点的坐标。坐标可以分别给出，也可以用有序表的形式给出。

```
[> with(geometry):
> point(A, 1, 1);
                                     A
> point(B, [x, y]);
                                     B
```

对于点对象，可以用函数 `coordinates` 获得点的坐标，或者用 `HorizontalCoord` 和 `VerticalCoord` 分别获得横纵坐标。

```
[> coordinates(A);
                                     [1, 1]
> VerticalCoord(B);
                                     y
```

对于所有的几何对象，都可以用 `form` 来获得几何对象的具体形式，以及用 `detail` 获得对象的详细情况。

```
[> form(A);
                                     point2d
> detail(A);
name of the object: A
form of the object: point2d
coordinates of the point: [1, 1]
```

定义了线段的两个端点，就可以定义线段。Maple 中的线段两种，有向线段和无向线段，

分别用函数 `dsegment` 和 `segment` 建立。例如：

```
[> segment(AB, [A, B]);
                                     AB
[> dsegment(dBA, B, A);
                                     dBA
```

对于已经定义的线段，可以用函数 `DefinedAs` 检测是否线段，如果是无向线段，就返回两个点的序列，如果是有向线段，则返回起点和终点构成的有序表。

```
[> DefinedAs(dBA);
                                     [B,A]
[> map(coordinates, %);
                                     [[x,y],[1,1]]
```

用两个点不仅可以定义线段，也可以用 `line` 定义一条直线。但是，和线段不同，由于定义直线需要直线方程，所以点的坐标必须定义为可以确定两个点不重合的条件。例如，前面定义的点 `A`, `B` 就不能确定一条直线：

```
[> line(l1, [A, B]);
geometry/checkline: One of the following conditions must be satisfied
1-x <> 0  1-y <> 0
Error, (in geometry/checkline) not enough information: the line is not
defined
```

但并不是说不能采用符号定义坐标的点定义直线，如果可以确定两个点的关系，同样可以定义直线：

```
[> point(C, [x+1, y+2]);
[> line(l1, [B, C]);
                                     l1
```

定义直线，不仅可以用两个不重合的点，还可以用直线方程直接确定一条直线：

```
[> line(l2, y = 2*x + a, [x, y]);
                                     l2
```

第二个参数是直线方程，`[x, y]` 是横纵坐标的变量，它是可省参数，只在必要时（方程中有未知参数时）才需给出。

对于直线对象，除了可以用 `form` 获得它的类型，用 `detail` 获得它的详细情况之外，还可以用函数 `Equation` 获得直线方程，用 `HorizontalName` 和 `VerticalName` 分别获得直线方程中的横纵坐标变量（这两个函数对于除了点以外的其他平面几何对象也是适用的）。当然，对于用点定义的直线，没有坐标变量，调用 `Equation` 时会提示输入坐标变量，需要注意的是，坐标变量名不可以在点坐标中出现。

```
[> Equation(l1);
[enter name of the horizontal axis > X;
[enter name of the vertical axis > Y;
                                     -2X+Y+(y+2)x-y(x+1)=0
[> HorizontalName(l2);
                                     x
```

11.1.2 三角形、正方形和圆

三角形是平面几何中经常研究的对象，在 Maple 中，可以用三个顶点定义三角形，也可以用三角形的三边定义三角形。调用函数 `triangle`，以三角形名称作为第一个参数，以三个顶点或者三条边所在的直线组成的有序表作为第二个参数，可以建立三角形对象。

```
[> point(A,0,0), point(B,1,1), point(C,1,0):
> triangle( T1, [A, B, C] );
                                     T1
> line( l1, y=0, [x,y] ), line( l2, y=x, [x,y] ),
  line( l3, x+y-2=0, [x,y] ):
> triangle( T2, [l1, l2, l3] );
                                     T2
```

和直线一样，在定义三角形时，还可以用第三个可选参数指明三角形的横纵坐标变量。

生成三角形对象，还可以不从已有的点或直线对象出发，而通过指定三角形的三边长，或者两条边的长度以及它们的夹角来定义三角形。例如：

```
[> triangle( T3, [3, 3, 3] ):
> triangle( T4, [3, angle = Pi/2, 4] ):
```

对于三角形对象，不仅可以用通用的函数 `form`, `detail`, `HorizontalName` 和 `VerticalName` 获得其信息，还可以用函数 `DefinedAs` 获得三角形的定义信息（三个顶点、三条边、三边长度或者两边长度与夹角），或者用 `method` 查询三角形的定义方式（如果用顶点或边定义，它将返回 `points`；如果用三边长度定义，将返回 `sides`；如果用两边和夹角定义，将返回 `angle`）。另外，还有特殊三角形的判定函数：用 `IsEquilateral` 可以检验三角形是否等边，用 `IsRightTriangle` 可以检验三角形是否为直角三角形：

```
[> IsRightTriangle( T4 );
                                     true
> IsEquilateral( T3 );
                                     true
```

`geometry` 工具包中，还有函数 `altitude`、`median` 和 `bisector` 可以分别求得三角形的高、中线和角平分线。它们的调用格式都相近，首先提供返回的直线名作为第一个参数，第二个参数确定高、中线或角平分线对应的三角形的顶点，第三个参数是三角形的名称。

```
[> altitude(hC, C, T1);
                                     hC
> detail(hC);
name of the object: hC
form of the object: line2d
assume that the name of the horizontal and vertical
axis are _x and _y
equation of the line: -1+_x+_y = 0
```

这三个函数还可以返回线段形式的对象，只要在函数调用中加入一个未赋值的变量名，作为第三个可选参数即可。此时，第三个参数返回的是高线（或者中线、角平分线）与对边的交点。

```

[> median(mC, C, T1, p);
                                         mC
[> detail(mC);
    name of the object: mC
    form of the object: segment2d
    two ends of the segment: [[1, 0], [1/2, 1/2]]
[> detail(p);
    name of the object: p
    form of the object: point2d
    coordinates of the point: [1/2, 1/2]

```

正方形也是一类特殊的平面图形，在 Maple 中，可以用函数 `square` 生成正方形对象。所提供的是正方形的四个顶点，不过，这四个顶点必须依次构成一个正方形。

```

[> point(A,0,0), point(B,a,0), point(C,a,a), point(D,0,a):
[> square(Sq, [A, B, C, D]);
                                         Sq
[> detail(Sq);
    name of the object: Sq
    form of the object: square2d
    the four vertices of the square: [[0, 0], [a, 0], [a, a], [0, a]]
    the length of the diagonal:  $2^{1/2} \cdot (a^2)^{1/2}$ 

```

除了可以调用 `square` 函数用正方形的四个顶点定义正方形外，还可以使用函数 `MakeSquare` 建立正方形对象，它的第二个参数是一个有序表，可以为下面三种形式之一：
`[p1, p2, 'diagonal']`——表示用对角线的两个端点 `p1`, `p2` 定义正方形；
`[p1, p2, 'adjacent']`——表示用正方形相邻的两个顶点 `p1`, `p2` 定义正方形（用这种方式定义的正方形不确定，有两个解，所以得到的是两个正方形的有序表）；
`[p1, 'center'=c]`——表示用一个顶点 `p1` 和正方形的中心点 `c` 定义正方形。例如上面的正方形 `Sq`，还可以用这样的方式定义：

```

[> MakeSquare(Sq1, [A, B, adjacent]):
[> MakeSquare(Sq2, [A, B, diagonal]):
[> MakeSquare(Sq3, [A, center = point(center, a/2, a/2)]):

```

对于正方形对象，除了通用的属性函数外，可以调用函数 `diagonal` 获得正方形的对角线长度。对于所有封闭的平面图形，包括三角形、正方形、还有下面将要介绍的圆和椭圆，都可以用函数 `area` 求它们的面积。

```

[> area(Sq);
                                         a2

```

在平面几何中，圆是我们研究最多的对象了。在 Maple 中，可以用函数 `circle` 生成圆对象，它的调用格式有以下四种：

```

circle ( c, [A, B, C], n, 'centername'=m )
circle ( c, [A, B], n, 'centername'=m )
circle ( c, [A, rad], n, 'centername'=m )
circle ( c, eqn, n, 'centername'=m )

```

其中，`c` 是生成的圆对象名，`[A, B, C]` 是圆周上的三个相异的点；`[A, B]` 是圆的一条直径的两个端点；`[A, rad]` 是圆的圆心和半径；`eqn` 是圆方程；`n` 是可选参数，是由两个坐标变量组成的有序表；`'centername'=m` 也是可选参数，是用来指定圆心名称为 `m` 的。

作为例子，首先用圆周上的三点定义一个圆：

```
> circle( c01, [point(A,0,0), point(B,2,0), point(C,1,2)],
  'centername'=O1);
```

利用函数 center, radius, Equation 和 area 可以分别获得圆对象的园心、半径、圆方程和圆的面积：

```
> center(c01), coordinates(O1);
                                 $O1, \left[1, \frac{3}{4}\right]$ 
> radius(c01);
                                 $\frac{1}{16}\sqrt{25}\sqrt{16}$ 
> Equation(c01);
enter name of the horizontal axis > X;
enter name of the vertical axis > Y;
                                 $X^2 + Y^2 - 2X - \frac{3}{2}Y = 0$ 
> area(c01);
                                 $\frac{25}{16}\pi$ 
```

11.1.3 二次曲线

在平面解析几何中，二次曲线（conic）是我们经常讨论的对象，在 Maple 中，可以用函数 parabola、ellipse 和 hyperbola 分别生成抛物线、椭圆和双曲线，也可以用函数 conic 统一定义二次曲线。

用函数 parabola 定义抛物线对象，有四种方式，如表 10.1 所示。

表 11.1 定义抛物线的方式

函数调用格式	参数含义
parabola (p, [A,B,C,E,F], n)	[A,B,C,E,F]——五个相异的点
parabola (p, ['focus'=fou, 'vertex'=ver], n)	fou——抛物线的焦点；ver——抛物线的顶点
parabola (p, ['directrix'=dir, 'focus'=fo, n)	dir——准线；fo——焦点
parabola (p, eqn, n)	eqn——抛物线方程

p——抛物线方程；n——坐标变量（可选）（下同）

对于抛物线对象，不仅可以用 Equation 获得抛物线方程，还可以用函数 vertex、focus 和 directrix 分别获得抛物线的顶点、焦点和准线：

```
> parabola( p1, y^2+12*x-6*y+33=0, [x, y] );
> vertex(p1), coordinates(vertex(p1));
                                vertex_p1, [-2, 3]
> focus(p1), coordinates(focus(p1));
                                focus_p1, [-5, 3]
> directrix(p1), Equation(directrix(p1));
                                directrix_p1, x - 1 = 0
```

同样, 在 Maple 中用函数 `ellipse` 生成椭圆对象, 也有多种方法, 这里将它们列成表 10.2。

表 11.2 定义椭圆的方式

函数调用格式	参数含义
<code>ellipse (p, [A,B,C,E,F], n)</code>	[A,B,C,E,F]——五个相异的点
<code>ellipse (p, ['directrix'=dir, 'focus'=fou, 'eccentricity'=ecc], n)</code>	dir——椭圆的准线; fou——焦点; ecc——偏心率
<code>ellipse (p, ['foci'=foi, 'MajorAxis'=lma], n)</code>	foi——两个焦点的有序表; lma——长轴长
<code>ellipse (p, ['foci'=foi, 'MinorAxis'=lmi], n)</code>	foi——两个焦点的有序表; lmi——短轴长
<code>ellipse (p, ['foci'=foi, 'distance'=dis], n)</code>	foi——两个焦点的有序表; dis——椭圆周上一点到两焦点的距离和
<code>ellipse (p, ['MajorAxis'=ep1, 'MinorAxis'=ep2], n)</code>	ep1——长轴端点的有序表; ep2——短轴端点的有序表;
<code>ellipse (p, eqn, n)</code>	eqn——椭圆方程

关于椭圆的属性, 可以调用函数 `center` 和 `foci` 分别获得椭圆的中心点和一对焦点; 还可以用 `MajorAxis` 和 `MinorAxis` 分别获得椭圆的长轴长和短轴长。例如:

```
[> ellipse( e1, 2*x^2+y^2-4*x+4*y=0, [x, y] );
> center(e1), coordinates(center(e1));
                                center_e1,[1,-2]
> foci(e1), map( coordinates, foci(e1) );
                                [foci_1_e1,foci_2_e1],[[1,-2-√3],[1,-2+√3]]
> MajorAxis(e1), MinorAxis(e1);
                                2√6, 2√3
```

双曲线对象的生成方式和椭圆基本相同, 这里将 `hyperbola` 的调用格式列成表 10.3。

表 11.3 定义双曲线的方式

函数调用格式	参数含义
<code>hyperbola (p, [A,B,C,E,F], n)</code>	[A,B,C,E,F]——五个相异的点
<code>hyperbola (p, ['directrix'=dir, 'focus'=fou, 'eccentricity'=ecc], n)</code>	dir——双曲线的准线; fou——焦点; ecc——偏心率
<code>hyperbola (p, ['foci'=foi, 'vertices'=ver], n)</code>	foi——两个焦点的有序表; ver——两个顶点
<code>hyperbola (p, ['foci'=foi, 'distancev'=disv], n)</code>	foi——两个焦点的有序表; disv——顶点距
<code>hyperbola (p, ['vertices'=ver, 'distancef'=disf], n)</code>	foi——两个焦点的有序表; disf——两焦点的间距
<code>hyperbola (p, eqn, n)</code>	eqn——双曲线方程

除了 `center` 和 `foci` 函数外, 对于双曲线对象, 还可以用 `asymptotes` 函数求得其渐进线; 用函数 `vertices` 求得它的两个顶点。

```
[> hyperbola( h1, 9*y^2-4*x^2=36, [x, y] );
> vertices(h1), map( coordinates, vertices(h1) );
                                [vertex_1_h1,vertex_2_h1],[[0,-2],[0,2]]
> asymptotes(h1), map( Equation, asymptotes(h1) );
                                [asymptote_1_h1,asymptote_2_h1],[y+2/3x=0,y-2/3x=0]
```

除了用 `parabola`、`ellipse` 和 `hyperbola` 分类定义二次曲线外, 在 Maple 中, 还可以用函

数 `conic` 直接定义不同的二次曲线。它的调用形式有三种：

表 11.3 定义二次曲线的方式

函数调用格式	参数含义
<code>conic (p, [A,B,C,E,F], n)</code>	[A,B,C,E,F]——五个相异的点
<code>conic (p, [dir, fou, ecc], n)</code>	dir——准线; fou——焦点; ecc——偏心率
<code>conic (p, eqn, n)</code>	eqn——二次曲线方程

根据输入的不同，函数 `conic` 可以返回不同类型的对象，包括抛物线、圆、椭圆和双曲线。

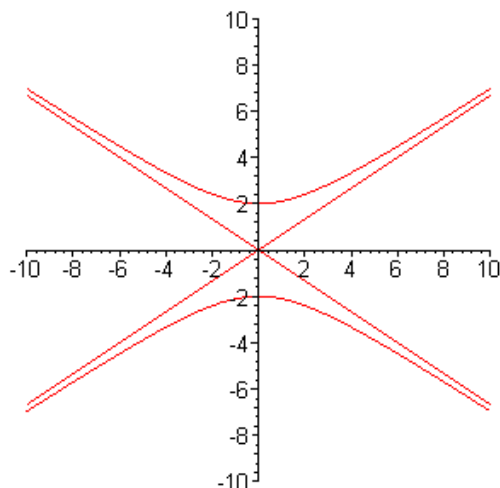
```
[> conic( c1, x^2-2*x*y+y^2-6*x-10*y+9=0, [x, y] ):
> form(c1);
                                     parabola2d
> line( l, x=-2, [x, y] ): point(f, 1, 0): e := 1/2:
   conic( c2, [l, f, e], [c, d]):
> form(c2);
                                     ellipse2d
> assume( R>0 ):
> conic( c3, (x-a)^2+(y-b)^2=R^2, [x, y]):
> form(c3);
                                     circle2d
```

除了这些以外，`conic` 还可以处理各种退化的二次曲线，例如退化的椭圆——单个的点，退化的抛物线——两条平行直线，退化的双曲线——两条相交直线。请看下面的例子。

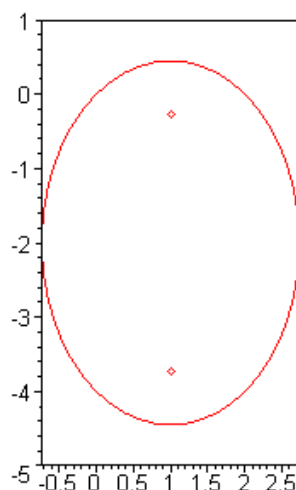
```
[> conic( c4, x^2-2*x*y-4*x+y^2+4*y-77, [x, y]);
geometry/conic/classify: "degenerate case: two parallel lines"
                                     [Line_1_c4, Line_2_c4]
> conic( c5, 11*x^2+24*x*y+4*y^2+26*x+32*y+15=0, [x, y]);
geometry/conic/classify: "degenerate case: two intersecting lines"
                                     [Line_1_c5, Line_2_c5]
```

所有的几何对象，都可以用 `geometry` 中的函数 `draw` 绘制图形——得到的结果是标准的二维图形对象，第六章中介绍的对图形对象属性的操作都仍将有效。

```
[> p1 := draw(h1, view=[-10..10, -10..10]):
> p2 := draw(asymptotes(h1), view=[-10..10, -10..10]):
> plots[display]([p1, p2], axes=normal);
```



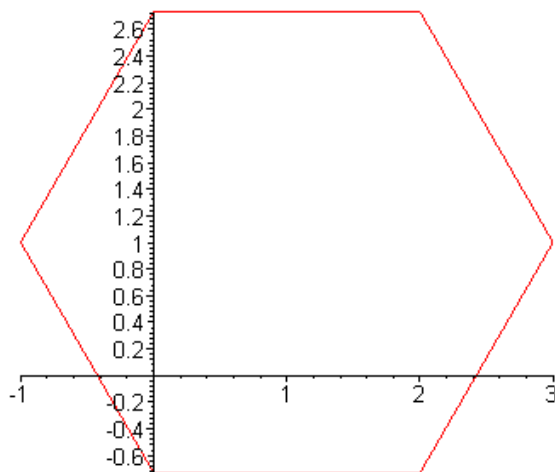
```
> draw([e1, op(foci(e1))]);
```



11.1.4 正多边形对象

在 Maple 中，可以用函数 `RegularPolygon` 生成正多边形对象，它的调用格式是：
RegularPolygon (p, n, cen, rad), 其中 p 是所生成的多边形对象的名称；n 是一个大于等于 3 的整数，表示多边形的边数；cen 是多边形的中心点；rad 是多边形外接圆的半径。

```
> RegularPolygon( gon, 6, point(o,1,1), 2 );
> draw( gon, axes = normal );
```

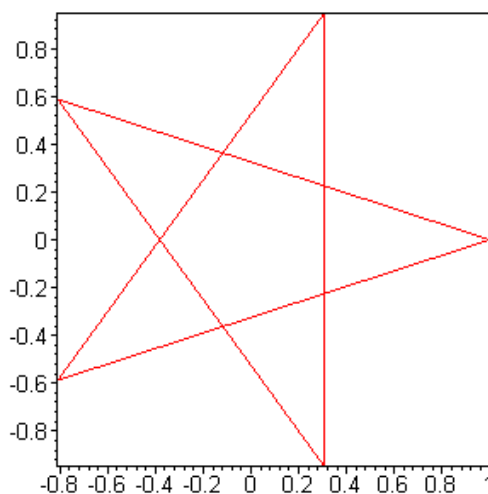


对于正多边形对象，用函数 `DefinedAs`、`sides`、`center` 和 `radius` 分别可以获得其顶点、边、中心和外接圆半径；另外，还可以用函数 `inradius`、`InteriorAngle`、`ExteriorAngle`、`apothem`、`perimeter` 和 `area` 分别求正多边形的内切圆半径、内角、外角、垂辐（边心距，同内切圆半径）、周长和面积。

除了这样的多边形外，还有所谓的星形正多边形，在 Maple 中用函数 `RegularStarPolygon` 生成。它的参数和 `RegularPolygon` 相同，只不过 n 不再限制为一个整数，而是可以为任一个大于 2 的有理数。这样，`RegularStarPolygon` 将生成一组相邻两点之间圆心角差为 $\frac{2\pi}{n}$ 的顶

点，并以它们为顶点生成星形多边形。例如，可以用 $5/2$ 作为参数，生成五角星对象：

```
[> RegularStarPolygon( gon, 5/2, point(o,0,0), 1 );
> draw(gon);
```



对于星形多边形对象，可以调用的属性函数有 DefinedAs、sides、center、radius、InteriorAngle、ExteriorAngle、perimeter 和 area，意义与对应的正多边形对象的函数相同。

11.2 平面几何对象的相互关系

11.2.1 点和直线的位置关系

在欧氏几何学中，两条相异直线之间无外乎两种位置关系——平行和相交。在 Maple 中，可以利用 geometry 中的函数 AreParallel 检测两条直线是否平行。例如：

```
[> with(geometry):
> line( l1, 3*x+5*y-4=0, [x, y] ):
> line( l2, 6*x+10*y+7=0, [x, y] ):
> AreParallel( l1, l2 );
true
```

在两条直线的关系不能确定时，函数将返回 FAIL。此时可以提供第三个可选参数，用它返回直线平行的条件，例如：

```
[> line( l1, a*x-2*y-1=0, [x, y] ):
> line( l2, 6*x-4*y-b=0, [x, y] ):
> AreParallel( l1, l2, cond );
AreParallel: "hint: cannot determine if -4*a+12 is zero"
FAIL
> cond;
-4*a+12=0
```

垂直是相交的特殊情况，在 Maple 中可以用函数 ArePerpendicular 加以判定。同样，在不可确定时，也可以用附加参数返回垂直的条件。

```

[> ArePerpendicular( l1, l2, 'cond' );
ArePerpendicular:  "hint: cannot determine if 6*a+8 is zero"
                     FAIL
[> cond;
                      $6a+8=0$ 

```

一般地，对于两条相交直线，可以用函数 FindAngle 求出两直线的夹角，或者用 intersection 求出两直线的交点。在这里，两条直线的夹角定义为两个夹角中较小的一个。

```

[> line( l1, y = 7*x - 2, [x, y] ),
      line( l2, y = x - sqrt(2), [x, y] );
                     l1, l2
[> FindAngle(l1, l2);
                      $\arctan\left(\frac{3}{4}\right)$ 
[> intersection(P, l1, l2);
                     P
[> coordinates(P);
                      $\left[-\frac{1}{6}\sqrt{2}+\frac{1}{3}, \frac{1}{3}-\frac{7}{6}\sqrt{2}\right]$ 

```

这两个函数不仅对于相交的直线适用，还可以用 intersection 求出直线和圆或者两个圆的交点，或者用 FindAngle 求出两个圆的夹角。对于圆的夹角，可以定义为在圆的交点处的逆时针方向的切线的夹角，它的取值在 $[0, \pi]$ 中变化（内切时为0，外切时为 π ）。

三条直线相交于一点，我们称它们为共点的，在 Maple 中，可以利用函数 AreConcurrent 检验三条直线是否共点。作为例子，我们用 Maple 验证一个特定三角形的三条高线共点：

```

[> line(l1, x+2*y+3=0, [x, y]):
[> line(l2, 3*x-7*y+9=0, [x, y]):
[> line(l3, 5*x-3*y-11=0, [x, y]):
[> triangle(T, [l1, l2, l3]):
[> P := DefinedAs(T);
                     P := [uu1, uu2, uu3]
[> altitude(h1, P[1], T):
[> altitude(h2, P[2], T):
[> altitude(h3, P[3], T):
[> AreConcurrent(h1, h2, h3);
                     true

```

函数 AreConcurrent 中还可以加入附加的可选参数，用来返回共点的条件。例如：

```

[> assume(a>0):
[> line(l1, a*x+b*y+1=0, [x, y]):
[> line(l2, 2*x-3*y+5=0, [x, y]):
[> line(l3, x-1=0, [x, y]):
[> AreConcurrent(l1, l2, l3, 'cond'):
AreConcurrent:  "unble to determine if 3*a+7*b+3 is zero"
[> cond;
                      $3a+7b+3=0$ 

```

三个或三个以上的点在同一条直线上，我们称它们是共线的，在 Maple 中，可以用函数 AreCollinear 判断三个点是否共线。

作为例子，我们验证前面例子中的三角形 T 的重心、垂心和外心三点共线。三角形的

重心可以用函数 `centroid` 获得:

```
[> centroid(G, T):
```

三角形的垂心可以用函数 `orthocenter` 求得:

```
[> orthocenter(H, T):
```

函数 `circumcircle` 可以求得三角形的外接圆, 同时也可以获得三角形的外心:

```
[> circumcircle(cc, T, 'centername'=O):
```

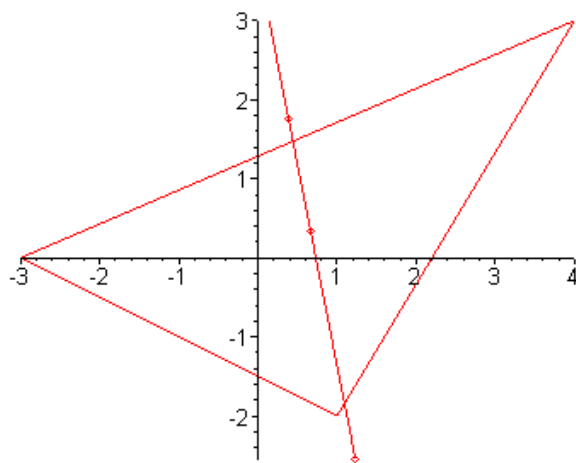
```
[> AreCollinear(G, H, O);
```

```
true
```

在平面几何中, 我们称三角形的重心、垂心和外心所在的直线为三角形的欧拉线, Maple 中对应的函数 `EulerLine` 可以求出给定三角形的欧拉线:

```
[> EulerLine(Ell, T):
```

```
[> draw([T, Ell, G, H, O], axes=normal);
```



11.2.2 与三角形相关的函数

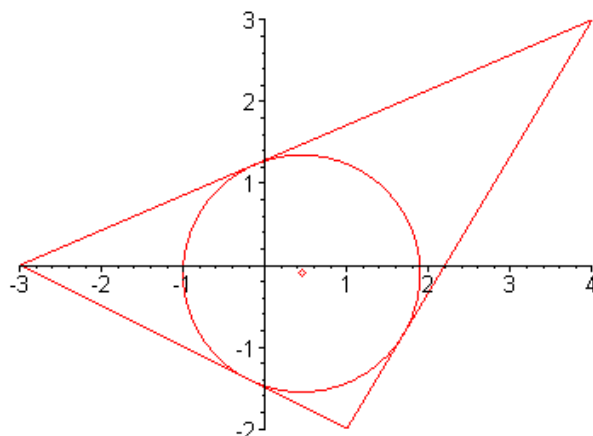
除了三角形的重心、垂心可以分别用函数 `centroid` 和 `orthocenter` 求得外, Maple 中还可以求得另两个平面几何中关心的点。其中之一是三角形的 Gergonne 点, 它定义为三角形的内切圆的三个切点与其相对顶点的连线的交点, Maple 中的函数 `GergonnePoint` 可以求得给定三角形的 Gergonne 点。另一个是 Nagel 点, 它是这样定义的: 设分别由顶点 A、B、C 出发, 在三角形的边上绕行周长的一半所到达的点为 D、E、F, 则可以证明直线 AD、BE、CF 共点, 它们的交点就称为 Nagel 点, 在 Maple 中可以用函数 `NagelPoint` 获得。

在介绍三角形对象时, 我们已经介绍了三角形的高、中线和内角平分线的函数——`altitude`、`median` 和 `bisector`; 除了这三个函数外, 还可以用 `ExternalBisector` 求出三角形的外角平分线。

从上一小节的例子中, 我们已经知道函数 `circumcircle` 可以求得三角形的外接圆, 除此之外, 还可以用函数 `incircle` 求得三角形的内切圆。和 `circumcircle` 一样, `incircle` 也可以用附加参数 “`centername = 圆心名称`” 的形式获得三角形的内心。例如:

```
[> incircle(ic, T, centername=icc):
```

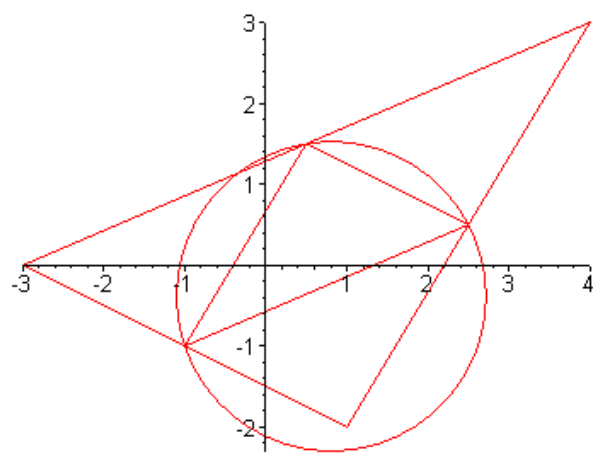
```
> draw([T, ic, icc], axes=normal);
```



函数 `excircle` 则可以求得三角形的旁切圆，和 `incircle` 不同，由于三角形有三个旁切圆，所以这个函数得到的是三个圆对象组成的有序表。

函数 `medial` 可以获得三角形三条中位线构成的三角形，而这个中三角形的外接圆，则称为原来的三角形的欧拉圆，可以用函数 `EulerCircle` 求得：

```
> medial(mT, T):
> EulerCircle(Elc, T):
> draw([T, mT, Elc], axes=normal);
```



相似三角形是平面几何中研究较多的问题，在解析几何中却没有过多的涉及，但是，Maple 中还是提供了判断两个三角形相似的函数 `AreSimilar`。这个函数也可以附加第三个参数，用于在无法判断时返回相似所需满足的条件。

```
> AreSimilar(T, mT);
true
```

11.2.3 与圆相关的函数

函数 `AreConcyclic`，可以判断四个点是否共圆。除了有待判断的四个点作为前四个参数外，它也可以加入第 5 个参数，用于在无法判断时返回共圆的条件。

```

[> point(P1, 0, 0): point(P2, 2, 0): point(P3, 2, 2):
    point(P4, 0, 2):
> AreConcyclic(P1, P2, P3, P4);
                                     true
[> point(P5, a, b):
> AreConcyclic(P1, P2, P3, P5, 'cond');
AreConcyclic:  "unable to determine if 32/45*(-2*a-2*b+a^2+b^2)/(a^2+b^2
+1) is zero"
                                     FAIL
[> cond;
                                     
$$\frac{32-2a-2b+a^2+b^2}{45a^2+b^2+1}=0$$


```

对于给定的圆 A，平面上任意一点 M 关于该圆的幂可以定义为：

$$\sigma(M) = d^2(M, A) - R^2$$

其中， $d(M, A)$ 表示点 M 与圆心 A 的距离，R 为圆的半径。由定义可以知道，当 M 在圆内时，其幂为负；在圆上时，幂为 0；圆外的点对于圆的幂为正。在 Maple 中，可以用函数 `powerpc` 求出一个给定点关于一个给定圆的幂（**power** of a given point with respect to a given circle）。

```

[> circle( c, x^2 + y^2 =1, [x, y] ):
[> point( A, 6/7, 2 ):
[> powerpc( A, c );
                                     
$$\frac{183}{49}$$

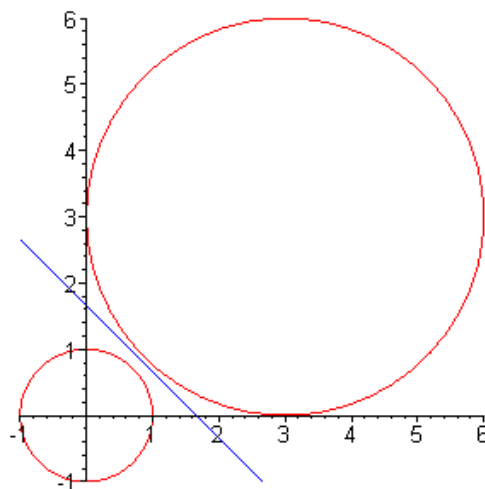

```

对于两个给定的圆，可以证明，所有关于两个圆的幂相等的点都共线，我们称这条直线为这两个圆的幂线（radical axis）。很容易知道，在两个圆相交的时候，它们的幂线就是两个圆交点的连线。Maple 函数 `RadicalAxis` 可以求得两个给定圆的幂线。

```

[> circle( c1, [ point(A,3,3), 3 ], [x, y] ):
[> RadicalAxis(l, c, c1);
                                     l
[> draw( [c, c1, l(color=blue)], axes=normal );

```



对于三个圆心不共线的圆，可以证明，它们两两之间确定的幂线共点，我们称这个点为等幂心（radical center）。Maple 中对应的函数是 `RadicalCenter`。

```

[> circle( c2, (x-2)^2 + y^2 = 9/5, [x,y] );
[> RadicalCenter(rc, c, c1, c2);
                                     rc
[> coordinates(rc);
                                     [ 4 13 ]
                                     [ 5' 15 ]

```

设 $\odot A$ 和 $\odot B$ 是两个非同心圆, 半径分别为 R_1 和 R_2 , 点 I 和 E 分别在线段 AB 上和 BA 的延长线上, 满足 $|AI|/|BI| = |AE|/|BE| = R_1/R_2$, 则分别称 I 、 E 为 $\odot A$ 和 $\odot B$ 的内、外相似中心 (internal and the external centers of **similitude**)。在 Maple 中有对应的函数 `similitude`, 它返回的是两个相似中心组成的有序表。

```

[> similitude( Psm, c, c1 );
                                     [in_similitude_of_c_c1, ex_similitude_of_c_c1]
[> map(coordinates, Psm);
                                     [[ 3 3 ] [ -3 -3 ]
                                     [ 4' 4' ] [ 2' 2 ]

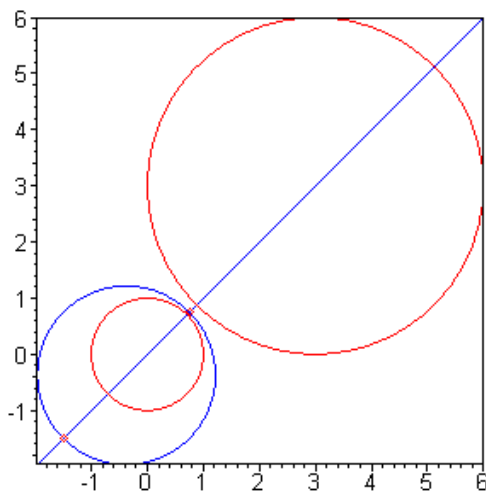
```

设 I 、 E 分别为 $\odot A$ 和 $\odot B$ 的内、外相似中心, 则以 IE 为直径的圆称为 $\odot A$ 和 $\odot B$ 的相似圆, 在 Maple 中可以用函数 `CircleOfSimilitude` 求得:

```

[> CircleOfSimilitude( cs, c, c1 );
[> draw([c, c1, line(l, [center(c), center(c1)])]
        (color=blue), op(Psm), cs(color=blue)]);

```



圆的切线是平面几何中经常研究的对象, 过圆外一点, 可以作圆的两条切线。在 Maple 中, 可以用函数 `TangentLine` 获得过圆外一点的圆的切线。它返回的是两条直线组成的有序表, 不过, 可以用两个变量名组成的有序表作为可选的附加参数, 分别用来命名这两条切线。例如:

```

[> TangentLine( obj, point(A, 1, 1), c, [t11, t12] );
[> Equation(t11), Equation(t12);
                                     -1+x=0, 1-y=0

```

对于圆上的一点, 通过它只能作圆的一条切线。可以用 `tangentpc` 获得过圆上一点的切线, 例如:


```
[> tangentpc(l, point(A, 1, 0), c):
> Equation(l);
-1+x=0
```

如果两个圆在交点处的切线相互垂直，我们称这两个圆相互正交，可以利用函数 AreOrthogonal 加以判断。

11.2.4 其他函数

这里，将前面未涉及到的其他有关平面几何对象相互关系的函数列成表 10.4，以供读者参考。

表 11.4 其他位置关系函数

函数调用格式	函数作用	参数含义
IsOnCircle (f, c, cond)	判断点是否在圆上	f——点；c——圆；cond——（可选） 返回在圆上的条件
IsOnLine (f, l, cond)	判断点是否在直线上	f——点；l——直线；cond——（可选） 返回在直线上的条件
SensedMagnitude (A, B)	返回两点的有向距离	A, B——两个点，B 在 A 的东北方向 为正，其余取负值
distance (P, l)	返回点到直线的距离	P——点；l——直线
projection (Q, P, l)	求点向给定直线的投影点	Q——返回投影点；P——已知点； l——给定直线
OnSegment (C, A, B, k) OnSegment (C, seg, k)	求以给定比例划分线段的点	C——返回划分点；A, B——线段端点； seg——线段；k——比例
randpoint (P, c) randpoint (P, u, v)	在给定范围内、直线或圆上生成一个随机点	P——返回随机点；c——直线或圆；u, v——给定的横纵坐标范围
CrossProduct (ds1, ds2)	计算两条线段的叉积	ds1, ds2——有向线段
slope (l)或 slope (A, B)	计算直线的斜率	l——直线；A, B 直线上两个点
ParallelLine (lp, P, l)	过给定点作已知直线的平行线	lp——返回平行直线；P——给定点； l——已知直线
PerpendicularLine (lp, P, l)	过给定点作已知直线的垂线	lp——返回垂线；P——给定点； l——已知直线
PerpenBisector (l, A, B)	求线段的中垂线	l——返回中垂线；AB——线段端点

11.3 平面上的变换

11.3.1 正交变换

在平面上，关于给定的有向线段 AB 的平移变换是这样定义的，平面上任意一点 P 经过变换后映射为 P'点，有向线段 AB // PP'，并且长度相同。称 AB 为该平移变换的变换向量。

在 Maple 的 geometry 工具包中，可以用函数 translation 完成任意平面几何对象的平移变换。

```
[> with(geometry):
[> dsegment( AB, point(A,0,0), point(B,a,b) ):
[> circle( c, [point(OO,0,0), 1] ):
[> translation( ctran, c, AB ):
[> detail( ctran );
      name of the object: ctran
      form of the object: circle2d
      name of the center: center_ctran
      coordinates of the center: [a, b]
      radius of the circle: 1
      assume that the name of the horizontal and vertical
axis are _x and _y
      equation of the circle: _x^2-1+_y^2-2*a*_x-2*b*_y+b^2+a^2 = 0
```

对于关于指定点 R 的旋转变换，Maple 中是这样定义的：平面上任意一点 P，经过变换映射为 P'，满足 $|PR| = |P'R|$ ， $\angle PRP'$ 为指定的旋转角。函数 rotation 可以完成任意平面几何对象的旋转变换，它的调用格式如下：

rotation (Q, P, g, co, R)

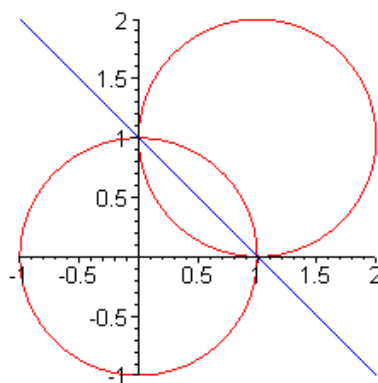
其中 Q 用于返回变换后的对象；P 是原对象；g 是旋转角；co 可以为 counterclockwise 或者 clockwise，分别表示逆时针和顺时针旋转；R 是可选参数，指定旋转中心，默认为坐标原点。

```
[> parabola( p, y^2=x, [x, y] ):
[> rotation( p1, p, Pi/4, 'counterclockwise' ):
[> Equation( p1 );
      
$$\frac{1}{2}y^2 - xy + \frac{1}{2}x^2 - \frac{1}{2}x\sqrt{2} - \frac{1}{2}y\sqrt{2} = 0$$

```

对于平面上的一条给定直线 L，将平面上任意一点 P 映射到 P 关于 L 的对称点 P' 的变换，称为平面上关于直线 L 的反射变换。同样，将任意点 P 映射到其关于给定点 A 的对称点 P' 的变换，称为平面上关于点 A 的反射变换。这两种反射变换，在 Maple 中都可以用函数 reflection 完成。例如：

```
[> line ( l, x + y = 1, [x, y] ):
[> reflection( cr, c, l ):
[> draw([c, l(color=blue), cr], axes=normal);
```



除了这些基本的正交变换外，Maple 还提供了一个组合变换——滑移反射——反射变换以及沿着对称线方向的平移的组合。滑移反射变换的函数调用是 GlideReflection(Q, P, l, AB)，它相当于两步变换的组合 reflection(Q, translation(T, P, AB), l)，其中，有向线段 AB

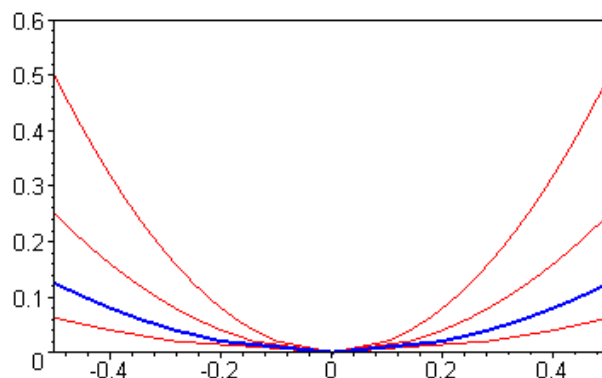
必须在直线 l 上。

11.3.2 其他类型的变换

正交变换保持平面上两点间的距离不变,除了正交变换外,在平面几何中经常研究的还有仿射变换。在 Maple 中这样定义仿射变换:在平面上给定一个点 O (称为变换的中心或不动点),平面上任意一点 P 映射为 P' ,有向线段 OP 与 OP' 方向相同,长度变为原来的 k 倍。有四个完全等价的函数可以完成仿射变换 `dilatation`、`expansion`、`homothety` 和 `stretch`,它们的第一个参数是变换后的对象名,第二个是原几何对象,第三个是变换系数 k ,最后一个是变换的中心点。例如:

```
> parabola( p1, [ 'vertex' = point( 'ver', 0, 0),
    'focus' = point( 'fo', 0, 1/2 ) ]):
> point(oo, 0, 0):
> expansion( p2, p1, 2, oo ):
> expansion( p3, p1, 1/2, oo):
> expansion( p4, p1, 1/4, oo):
> draw( [ p1( color=blue, style=LINE, thickness=2 ),
    p2, p3, p4], view=[-1/2..1/2, 0..3/5],
    title="dilatation of a hyperbola" );
```

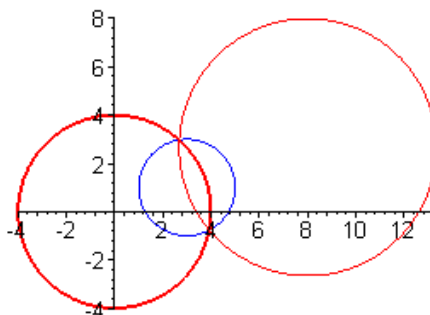
dilatation of a hyperbola



仿射变换也可以和正交变换相组合,得到复合的变换。Maple 中予以实现的有仿射变换与反射变换的合成——`StretchReflection`,仿射变换与旋转变换的合成——`StretchRotation` (或等价的函数 `homology`、`SpiralRotation`)。其中函数调用 `StretchReflection (Q, P, l, O, k)` 完成的变换相当于两步变换的组合 `stretch (Q, reflection (T, P, l), O, k)`,仿射变换的不动点 O 必须位于直线 l 上;函数调用 `StretchRotation (Q, P, O, theta, dir, k)` 的结果也相当于两步变换合成的结果 `stretch (Q, rotation (T, O, theta, dir), O, k)`。

Maple 中还有一个基于圆的变换,它将圆外的部分映射成圆内的部分,而原来位于圆内的部分映射成圆外的部分。给定 $\odot O$,对于平面上任意一点 P ,变换的像点 P' 在直线 OP 上,并且满足 OP 和 OQ 的有向长度 (带符号的长度) 的乘积为圆半径的平方 R^2 。函数 `inversion` 完成的的就是这一变换。不过,这一函数只能对点、直线和圆进行变换——因为只有它们在变换后仍将是 Maple 的基本几何对象。例如:

```
[> circle( c1, x^2+y^2 = 16, [x, y] );
> circle( c2, [point(A, 3, 1), 2], [x, y] );
> inversion( c3, c2, c1 );
> draw([c1(thickness=2), c2(color=blue), c3],
      axes=normal);
```



11.4 空间几何对象

geom3d 是 Maple 中专门处理三维空间欧氏几何问题的工具包。不过，由于没有空间曲面的几何对象，它的解题能力相对较弱一些。在这一章的剩余部分中，我们将围绕 geom3d 工具包，着重介绍用 Maple 解决空间几何问题的方法。

和 geometry 工具包一样，geom3d 中也定义了专门的几何对象类型，包括点、线段、有向线段、直线、三角形、平面、球和多面体。这一节中将对它们逐一介绍。

11.4.1 点、线、面

和平面几何工具包中定义点的函数相类似，在这里，定义点也需要给出点的坐标，所不同的是，三维空间中点有三个坐标。例如：

```
[> with(geom3d):
> point( A, a, b, c );
```

A

三维空间中的几何对象信息，也可以用函数 detail 获得，用 form 可以获得它们的类型。对于三维空间中的点，coordinates 函数可以获得它的坐标，xcoord、ycoord、zcoord 可以分别获得它们在 x、y、z 轴上的坐标。

线段 segment 和有向线段 dsegment 的定义方法和平面几何中的定义方法完全相同，只需要指明线段的两个端点，这里不再重复。对于空间的线段对象，也可以用函数 DefinedAs 获得线段的两个端点，用函数 midpoint 获得线段的中点。

相比来讲，空间几何中定义直线的方式要灵活得多，在 geom3d 中一共有 6 种定义直线的方法，这里将它们列成表 10.5。

表 11.5 定义空间直线的方式

函数调用格式	参数含义
--------	------

<code>line (l, [A, B])</code>	[A,B]——直线上两个相异的点
<code>line (l, [A, v])</code>	A——直线上一个点; v——向量, 表示直线方向
<code>line (l, [A, dseg])</code>	A——直线上一个点; dseg——有向线段
<code>line (l, [A, p1])</code>	定义的直线 l 过点 A 且垂直于平面 p1
<code>line (l, [p1, p2])</code>	定义的直线 l 为平面 p1、p2 的交线
<code>line (l, [a1+b1*t, a2+b2*t, a3+b3*t], t)</code>	[a1+b1*t, a2+b2*t, a3+b3*t]——直线参数方程

```
[> assume(l<>0): # 保证v非0向量
[> v := [1,m,n]:
[> line ( l1, [A, v] ):
[> detail(l1);
name of the object: l1
form of the object: line3d
assume that the name of the parameter          in the parametric
equations is _t
assume that the name of the axis are _x, _y, and _z
equation of the line: [_x = a+_t*1, _y = b+_t*m, _z = c+_t*n]
```

可以看到, 在 Maple 的内部, 空间直线是用参数方程的形式表示的。

geom3d 中三角形的定义只有两种方式——用三个顶点定义或者用三条边所在的三条直线定义。函数 triangle 的调用格式与 geometry 中相应定义方式的调用格式相同。

利用 geom3d 中的函数 plane 可以定义三维空间中的平面对象, 它也有多种定义方式:

表 11.6 定义平面的方式

函数调用格式	参数含义
<code>plane (p, [A, B, C])</code>	[A, B, C]——平面上三个不共线的点
<code>plane (p, [A, v])</code>	A——平面上一个点; v——平面的法向量
<code>plane (p, [A, dseg])</code>	A——平面上一个点; dseg——有向线段, 表示平面的法向量
<code>plane (p, [dseg1, dseg2])</code>	[dseg1, dseg2]——起点相同的两条有向线段
<code>plane (p, [l1, l2])</code>	平面上两条不重合的直线, 相交时 p 过这两条直线, 异面时 p 过 l1 且平行于 l2
<code>plane (p, [A, l1, l2])</code>	定义的平面 p 过点 A, 而且与 l1、l2 平行
<code>plane (p, eqn, n)</code>	eqn——平面方程; n——坐标变量

利用 geom3d 中的函数 Equation 可以获得平面的方程, 而出现在方程中的定义平面的三个坐标变量, 则分别可以用函数 xname、yname 和 zname 获得。利用函数 NormalVector, 还可以获得平面的法向量 (并不进行单位化)。

```
[> plane( p1, [ point(P0, 2, 1, -1), [1, -2, 3] ] ):
[> Equation(p1);
enter name of the x-axis > x;
enter name of the y-axis > y;
enter name of the z-axis > z;
3+x-2y+3z=0
[> NormalVector(p1);
[1, -2, 3]
```

11.4.2 球和多面体

如同圆在平面几何中的地位一样，球在空间几何也占有相当重要的地位。在 Maple 中，可以调用 geom3d 中的函数 sphere 生成球体对象。

表 11.7 定义球体的方式

函数调用格式	参数含义
sphere (s, [A, B, C, D], n, 'centername'=m)	[A, B, C, D]——球面上四个相异的点
sphere (s, [A, B], n, 'centername'=m)	[A, B]——球一条直径的两个端点
sphere (s, [A, rad], n, 'centername'=m)	A——球心；rad——球半径
sphere (s, [A, p], n, 'centername'=m)	A——球心；p——球的切平面
sphere (s, eqn, n, 'centername'=m)	eqn——球方程

s——生成的球对象名；n（可选）——坐标变量；'centername'=m（可选）——返回球心

对于球对象，除了可以使用 form、Equation、xname、yname、zname 和 detail 函数外，还可以用 center 和 radius 分别获得球的球心和半径，或者用 area 和 volume 求得球的表面积和体积，例如：

```
[> sphere(s, x^2+y^2+z^2+7*y-2*z+2=0, [x, y, z]):
> coordinates(center(s));
                                 $\left[0, -\frac{7}{2}, 1\right]$ 
> radius(s);
                                 $\frac{3}{2}\sqrt{5}$ 
> S = area(s), V = volume(s);
                                 $S = 45\pi, V = \frac{45}{2}\pi\sqrt{5}$ 
```

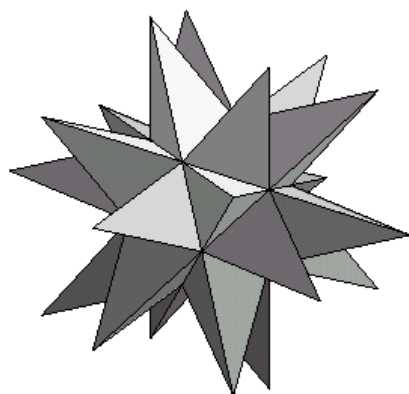
除了球体，Maple 的 geom3d 中还定义了很多多面体几何对象。其中包括 6 种正多面体对象：

表 11.8 geom3d 中的正多面体

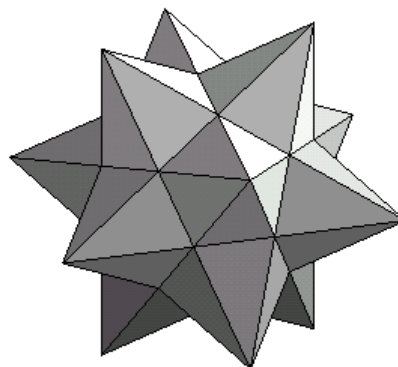
函数	对应的多面体
tetrahedron (gon, o, r)	正四面体
hexahedron (gon, o, r)	正六面体
cube (gon, o, r)	正方体
octahedron (gon, o, r)	正八面体
dodecahedron (gon, o, r)	正十二面体
icosahedron (gon, o, r)	正二十面体

所有函数的参数意义都相同，gon 作为多面体对象名，o 为多面体的中心点，r 为多面体外接球半径。

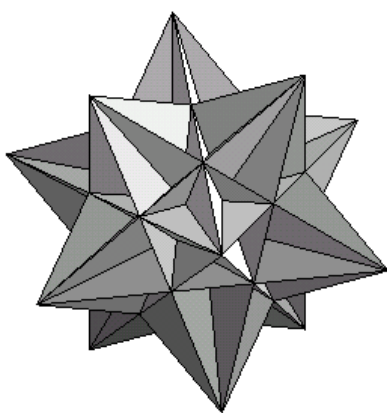
除了这些凸正多面体外，Maple 中还定义了一些星状的（凹）多面体，图 10.1 所示的是它们的图形（可以用 draw 函数绘制）。



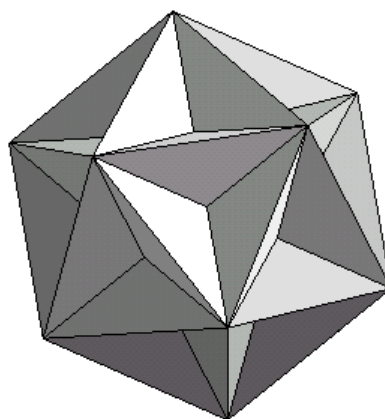
(a) GreatStellatedDodecahedron



(b) SmallStellatedDodecahedron



(c) GreatIcosahedron



(b) GreatDodecahedron

图 11.1 星状正多面体

对于正多面体对象，可以用的属性函数有 `area`（求表面积），`center`（中心），`faces`（表面片，返回每个面片的顶点坐标），`InRadius`（内切球半径），`MidRadius`（中切球半径，指与所有棱相切的球），`radius`（外接球半径），`sides`（棱长），`vertices`（返回所有顶点坐标），`volume`（体积）。

11.5 空间几何对象的关系

11.5.1 有关点的函数

和平面上的点一样，三维空间中的点也有共线关系，调用函数 `AreCollinear` 就可以检验空间三个点是否共线。如果条件不足而无法判断，则可以加入可附加参数返回共线需要满足的条件。与之相类似，函数 `AreCoplanar` 可以判断空间中的四个点是否共面。

另外，下列函数的意义和调用格式均和平面几何中的同名函数相同，这里不再重复：

coordinates (点坐标), randpoint (在指定范围内随机取点), midpoint (线段中点), OnSegment (线段的分划点)。

函数 IsOnObject 可以判断一个点 (或者一系列点) 是否在一条直线、一个平面或球上。

作为例子我们首先作一个平面 p , 它过两个平面 $3x - 4y + 5z = 10$, $2x + 2y - 3z = 4$ 的交线, 并且平行于直线 $x = 2y = 3z$:

```
[> point(o, 0, 0, 0):
[> line(l1, [o, [1, 1/2, 1/3]]):
[> plane(p1, 3*x-4*y+5*z=10, [x,y,z]):
[> plane(p2, 2*x+2*y-3*z=4, [x,y,z]):
[> line(l2, [p1, p2]):
[> plane(p, [l2, l1]):
```

利用 FixedPoint 函数可以获得直线上的一个随机的固定点, 我们用它检查直线 $l1$ 上的点是否在平面 p 上:

```
[> IsOnObject( FixedPoint(l1), p );
false
```

11.5.2 有关直线和平面的函数

直线和平面的关系, 无外乎三种——直线在平面上、直线和平面平行、直线和平面相交。geom3d 中的函数 AreParallel 可以测试三维空间中多种几何对象的平行关系, 例如两条直线或有向线段, 或者两个平面的平行判断, 还可以判断一条直线与一个平面的平行关系。和其他大多数判断函数一样, 在条件不足以判定是否平行时, 还可以利用附加参数给出平行的条件。

例如, 我们可以用它判断两个带参数的平面是否平行:

```
[> assume( l1<>0, l2<>0, l1<>l2, t<>0 ):
[> plane( pp1, l1*x+p1*y+n1*z+p1, [x,y,z] ):
[> plane( pp2, l2*x+p2*y+n2*z+p2, [x,y,z] ):
[> AreParallel( pp1, pp2, 'cond' );
FAIL
[> cond;
&and(p1 n2 - n1 p2 = 0, n1 l2 ~ - l1 ~ n2 = 0, l1 ~ p2 - p1 l2 ~ = 0)
```

再用 additionally 假设平行条件成立: (additionally 的用法与用途和 assume 相似, 所不同的是, 再作了新的假设之后, assume 自动去除同一变量以前的假设, 而 additionally 是保持原有假设前提下的附加假设。)

```
[> additionally( op(cond) );
[> AreParallel( pp1, pp2 );
true
```

不过, 和我们通常意义上的平行概念有所不同, AreParallel 对于重合 (或者是直线在平面上) 的情况也认为是平行的, 例如:

```
[> AreParallel( pp1, pp1 );
true
```


在这种意义下，直线和平面的关系除了平行之外，就是相交；Maple 中可以用函数 `intersection` 求直线和平面的交点。函数 `intersection` 的应用范围比较广，除了求直线和直线、平面、球面的交点以外，还可以求两个平面的交线、三个平面的交点、或两个球面的交线（在两个球重合时返回整个球面）。调用 `intersection` 时，第一个参数是用来返回的空间几何对象名，后面的参数就是有待求教的几何对象。例如，可以这样求三个已知平面的交点：

```
[> point(A, 0, 0, 0): point(B, 1, 0, 0):
    point(C, 0, 1, 0): point(E, 0, 0, 1):
> plane(oxy, [A, B, C]): plane(oyz, [A, C, E]):
    plane(oxz, [A, B, E]):
> intersection(P, oxy, oyz, oxz):
> coordinates(P);
                                [0, 0, 0]
```

直线和平面垂直是相交的一种特殊情形，在 Maple 中，可以用函数 `ArePerpendicular` 判断直线和平面是否垂直。除此之外，`ArePerpendicular` 还可以判断其他空间几何对象的垂直情况。例如两条直线或线段的垂直，两个平面的垂直，还有两个球的垂直，和平面几何中两个圆的垂直类似，两个球的垂直定义为它们在交线处的切平面相互垂直。在给定条件不足以判断对象是否垂直时，函数 `ArePerpendicular` 也可以用附加参数返回垂直的条件。

```
[> ArePerpendicular(oxy, oyz);
                                true
> assume(a<>0):
> line(l, [A, [a, b, c]]):
> ArePerpendicular(l, oyz, 'cond');
                                FAIL
> cond;
                                (c = 0) &and (-b = 0)
```

除了判断垂直之外，Maple 中的函数 `FindAngle` 可以求得两个空间几何对象，例如两条直线（未必相交）、两个平面或两个球的夹角，还可以求直线与平面的夹角。

```
[> FindAngle(l, oyz);
                                 $\arcsin\left(\frac{a}{\sqrt{a^2 + b^2 + c^2}}\right)$ 
```

空间两条直线的关系，除了平行和相交，还有一种情况——异面，在 Maple 中可以用函数 `AreSkewLines` 判断两条直线是否为异面直线：

```
[> line(l1, [point(A, 0, 0, 0), point(B, 1, 1, 0)]):
> line(l2, [point(C, 0, 0, 1), point(D, 1, -1, 1)]):
> AreSkewLines(l1, l2);
                                true
```

空间两个几何对象的距离可以用函数 `distance` 求得，它可以求空间中两个点、两条直线、或者两个平面之间的距离，还可以求点到直线或平面、直线到平面的距离。

例10.1 求直线 $x = 2y = 3z + 6$ 上一点 P ， P 到平面 $2x + y - 2z = 5$ 的距离为 8。

首先定义直线上的一个动点 o 和平面 p ：

```
[> point(o, t, 1/2*t, -2+t/3):
> plane(p, 2*x+y-2*z=5, [x, y, z]):
```

再求动点到平面距离的表达式，并求解方程：

```
[> d := distance( o, p );
                                     
$$d := \frac{1}{3} \left| \frac{11}{6}t - 1 \right|$$

> ans := solve(d = 8, t);
                                     
$$ans := \frac{150}{11}, \frac{-138}{11}$$

```

将参数 t 的结果带入到点 o 的坐标中，就可以得到所求的点的坐标：

```
[> subs( t=ans[1], coordinates(o) );
                                     
$$\left[ \frac{150}{11}, \frac{75}{11}, \frac{28}{11} \right]$$

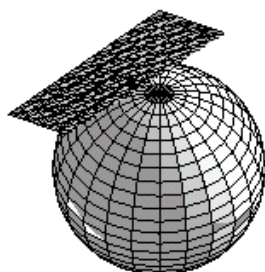
> subs( t=ans[2], coordinates(o) );
                                     
$$\left[ \frac{-138}{11}, \frac{-69}{11}, \frac{-68}{11} \right]$$

```

11.5.3 与球相关的函数

过给定的球面上的一点，可以作该球的一个切面，在 Maple 的 geom3d 工具包中，函数 TangentPlane 可以求得过球面上一点的切面。

```
[> sphere( s, [ point(o, 0, 0, 0), 1 ] );
> randpoint( P, s );
> TangentPlane( tp, P, s );
> draw( [s, tp] );
```



函数 IsTangent 则可以用来检验一个平面是否与给定的球相切。

```
[> IsTangent( tp, s );
                                     true
```

和平面几何中点关于圆的幂一样，在空间中也可以定义点 P 关于球 C 的幂：

$$\sigma(P) = d^2(P, C) - R^2$$

其中 $d(P, C)$ 是点 P 到球心 C 的距离，R 是球半径。在 Maple 中，可以用函数 powerps 求一点关于给定球的幂。

```
[> point(P, 1, 1, 1);
> powerps(P, s);
                                     2
```

关于两个给定球面的幂相等的点的轨迹是一个平面，称之为这两个球的**根面**（radical plane）。在 Maple 中，两个球的根面可以用函数 RadicalPlane 求得：

```
[> sphere(s1, (x-5)^2+(y-3)^2+(z+1)^2=9, [x, y, z]):
[> sphere(s2, (x-7)^2+y^2+(z-2)^2=16, [x, y, z]):
[> RadicalPlane(rp, s1, s2):
[> detail(rp);
      name of the object: rp
      form of the object: plane3d
      equation of the plane: -11+4*x-6*y+6*z = 0
```

可以证明，关于三个球面的幂相等的点的轨迹是空间中的一条直线，可以称其为这三个球的**根线**（radical line）；而对于四个球面，关于它们的等幂点是唯一确定的一个点，称为**根点**（radical center）。在 Maple 中，可以分别用函数 RadicalLine 和 RadicalCenter 求得根线和根点。

```
[> sphere( s1, x^2 + y^2 + z^2 = 1, [x, y, z] ):
[> sphere( s2, [ point(B, [5, 5, 5]), 2 ] ):
[> sphere( s3, [ point(A, 1, 2, 3), 3 ] ):
[> RadicalLine( l, s1, s2, s3 ):
[> Equation(l);
enter name of the independent variable > t;
      
$$\left[ \frac{57}{5} + 20t, -\frac{21}{5} - 40t, 20t \right]$$

```

11.6 三维空间中的几何变换

11.6.1 空间几何对象的变换

三维空间中的几何变换也可以分成正交变换和其他类型的变换。其中，正交变换可以分为平移变换、旋转变换、反射变换以及由这三种基本变换合成的复合变换。

与平面几何中的平移变换一样，三维空间中的平移变换 translation 也需要提供一条有向线段作为变换向量。它可以对任意三维空间几何对象进行变换。

例如，可以对一个星形正二十面体作平移变换。首先，可以通过一个正二十面体生成星形二十面体：

```
[> icosahedron(ico, point(o, 0, 0, 0), 1):
[> stellate(ico1, ico, 20):
```

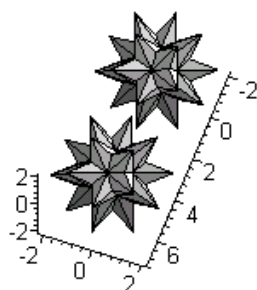
然后对其进行由有向线段定义的平移变换：

```
[> dsegment( AB, point(A,0,0,0), point(B,5,0,0) ):
[> translation(ico2, ico1, AB):
```

可以通过函数 draw 绘制变换前后的星形二十面体的图形来观察变换的结果：

```
> draw([ico1, ico2], axes=framed, orientation=[20,32],
lightmodel=light4, title="Translation of a stellated
icosahedron");
```

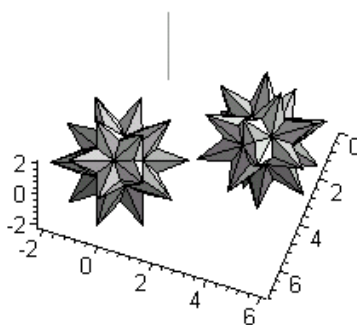
Translation of a stellated icosahedron



三维空间中的旋转变换采用的函数也是 `rotation`，所不同的是，三维空间的旋转变换，旋转中心不再是一个点，而是一条直线。例如：

```
> line(1, [0,0,t], t):
> rotation(ico3, ico2, evalf(Pi/3), 1):
> draw([ico2, ico3, 1], orientation = [20,32],
axes=framed, lightmodel=light4,
title="Rotation of a stellated icosahedron");
```

Rotation of a stellated icosahedron



三维空间中的反射变换有三种，分别是关于点、直线和平面的反射变换。空间中任意一个点在变换以后的像分别是该点关于点、直线和平面的对称点。函数 `reflection` 的第一个参数用于返回变换后的对象，第二个参数是有待变换的几何对象，第三个参数是变换的参考点、参考直线、或者参考平面。例如，我们对一个球作关于 `oxy` 平面的反射变换：

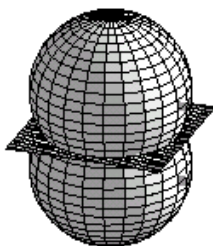
```
> sphere( s, [point(0,1,1,1), 2] ):
> plane( oxy, [ point(0, 0, 0, 0), point(X, 1, 0, 0),
point(Y, 0, 1, 0) ]):
> reflection( s1, s, oxy );
```

由于反射变换的逆变换就本身，所以可以再作一次变换，得到的几何对象应该和原对象相重合。

```
[> reflection( s2, s1, oxy ):
> AreSameObjects( s, s2);
true
```

也可以通过作图对变换结果进行检验:

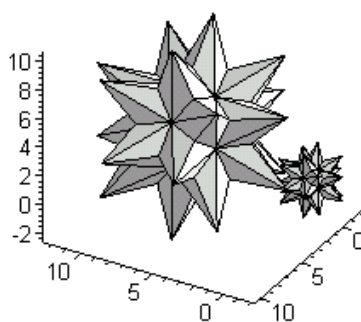
```
> draw( [s1, s, oxy], orientation=[20, 65],
lightmodel=light4 );
```



仿射变换也是三维空间中一类重要的非正交变换, geom3d 中仿射变换的函数是 homothety, 和平面中的仿射变换一样, homothety 也需要指定变换的比例和变换的中心。

```
[> homothety(ico2, ico1, 3, point(P, -3, -2, -2)):
> draw([ico1, ico2], axes=framed, orientation=[120,60],
lightmodel=light4, title="Homothety of a stellated
icosahedron");
```

Homothety of a stellated icosahedron



投影变换是一类特殊的变换, geom3d 中的函数 projection 可以计算点、线段或直线在给平面上的投影, 也可以用来计算点在一条空间直线上的投影点。例如:

```
[> line( l, [point(o,1,-1,3), [2,-1,4]] ):
> plane( p, x+2*y+z=6, [x, y, z] ):
> projection( l1, l, p ):
> IsOnObject( FixedPoint(l1), p );
true
```

```
[> Equation(l1);
enter name of the independent variable > t;

$$\left[ \frac{5}{3} + \frac{4}{3}t, \frac{1}{3} - \frac{7}{3}t, \frac{11}{3} + \frac{10}{3}t \right]$$

```

与在平面中的变换一样，Maple 中也有一些由基本变换复合而成的空间几何变换。这里将它们列成表 10.9。

表 11.9 geom3d 中的复合变换

复合变换函数	对应的变换
GlideReflection (Q, P, p, AB)	以 AB 为变换向量的平移变换和以平面 p 为对称面的反射变换的复合，有向线段 AB 在平面 p 上。
RotatoryReflection (Q, P, p, theta, l)	以直线 l 为轴转角为 theta 的旋转变换和以 p 为对称面的反射变换的复合， $l \perp p$ 。
ScrewDisplacement (Q, P, theta, l, AB)	螺旋变换，以直线 l 为轴转角为 theta 的旋转变换和以 AB 为变换向量的平移的复合，有向线段 AB 在 l 上。
homology (Q, P, K, O, theta, l)	以直线 l 为轴转角为 theta 的旋转变换和以 O 为中心，K 为缩放比例的仿射变换的复合， $O \in l$ 。

11.6.2 几何变换的谓词运算

和平面几何工具包中的变换不同，geom3d 中的变换不仅可以对空间几何对象直接作用，还可以进行变换之间的抽象运算。要定义没有具体几何对象的空间变换，需要用到一些基本变换的变换谓词。geom3d 中预先定义的变换谓词如表 10.10 所列。

表 11.10 geom3d 中的基本变换谓词

谓词	对应的变换	对应的变换函数
rotate	旋转变换	rotation
translate	平移变换	translation
ScrewDisplace	螺旋运动	ScrewDisplacement
reflect	反射变换	reflection
RotatoryReflect	旋转反射	RotatoryReflection
GlideReflect	平移反射	GlideReflection
dilate	仿射变换	homothety
StretchRotate	仿射旋转	homology

用变换谓词定义一个几何变换，和调用函数对特定的几何对象作变换的形式类似，所不同的是不需要指定变换的对象。例如，定义以原点为中心缩放比例为 3 的仿射变换：

```
[> t1 := dilate( 3, point(o, 0, 0, 0) );
t1 := dilate(3,o)
```

定义一个平移反射变换：

```
[> point(A, 1, 0, 0): point(B, 0, 0, 1):
[> line(l1, [o, A]): line(l2, [o, B]): plane(oxz, [l1, l2]):
[> dsegment(AB, [A, B]):
```

```
[> t2 := GlideReflect(oxz, AB);
      t2 := GlideReflect(oxz, AB)
```

再定义一个以直线 AB 为轴，旋转 $\pi/2$ 的螺旋运动：

```
[> t3 := ScrewDisplace( Pi/2, line(l3, [A, B]), AB );
      t3 := ScrewDisplace( $\frac{1}{2}\pi, l3, AB$ )
```

变换谓词之间支持三种基本的运算——幂运算、乘积和求逆，分别用运算符“^”和函数 transprod、inverse 表示。乘积运算就是两个变换的复合，幂运算定义了这样的一种复合运算：例如 $t_1 \wedge t_2$ ，表示 $t_2^{-1} * t_1 * t_2$ ，也即相当于 transprod (inverse (t_2), t_1 , t_2)。

```
[> q1 := transprod( t2^t1, t3 );
      q1 := transprod( $\text{dilate}(\frac{1}{3}, o)$ , reflect(oxz), translate(AB),  $\text{dilate}(3, o)$ ,  $\text{rotate}(\frac{1}{2}\pi, l3$ ,
      translate(AB))
[> q2 := inverse(q1);
      q2 := transprod(translate(_AB),  $\text{rotate}(\frac{3}{2}\pi, l3$ ,  $\text{dilate}(\frac{1}{3}, o)$ , translate(_AB), reflect(oxz),
      dilate(3, o))
```

$q1$ 和 $q2$ 显然是两个互逆的变换，我们可以通过求它们的积来检验：

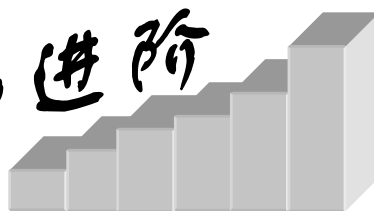
```
[> q := transprod(q1, q2);
      q := transprod( $\text{dilate}(\frac{1}{3}, o)$ , reflect(oxz), translate(AB),  $\text{dilate}(3, o)$ ,  $\text{rotate}(\frac{1}{2}\pi, l3$ ,
      translate(AB), translate(_AB),  $\text{rotate}(\frac{3}{2}\pi, l3$ ,  $\text{dilate}(\frac{1}{3}, o)$ , translate(_AB), reflect(oxz),
      dilate(3, o))
```

对于由变换谓词组成的变换，可以用函数 transform 将其作用到特定的几何对象上去。

```
[> tetrahedron(te, o, 1);
[> transform(te1, te, q);
[> AreDistinct(te, te1);
      false
```

由于 q 实际上是一个幺变换，所以 te 和它的像 $te1$ 是相同的几何对象（函数 AreDistinct 返回 false）。

起步与进阶



第

十

二

章

离散数学

本章将围绕着 Maple 中的五个和离散数学相关的软件包，向大家介绍如何用 Maple 解决离散数学中有关图论、布尔代数、群论和组合数学的问题。在计算机飞速发展的今天，离散数学作为计算机科学的数学基础，其地位也日益提高。希望大家通过本章的学习，能掌握用 Maple 辅助解决离散数学问题的基本方法。

本章具体包括以下内容：

- 🕒 如何在 Maple 中建立图对象
- 🕒 图论的基本概念和运算
- 🕒 图的着色问题
- 🕒 Maple 中的布尔预算和数理逻辑函数
- 🕒 Maple 中的群论运算简介
- 🕒 Maple 中的组合结构
- 🕒 组合数学中的常用工具

离散数学是研究离散量的结构及其相互关系的数学学科，是现代数学的一个重要分支，尤其在计算机科学与技术领域中有无可替代的应用。

在 Maple V Release 5 中，有五个与离散数学相关的工具包，分别是图论工具包 `networks`、布尔代数工具包 `logic`、群论工具包 `group` 和组合数学工具包 `combinat` 与 `combstruct`。在这一章中，将对它们一一加以介绍。

12.1 图论工具包 `networks`

图论的研究对象是离散的事物以及它们之间的二元关系。在图（`graph`）中，用点表示单个的事物，用点之间的连线表示它们之间的关系，这样就建立了图与实际问题的联系。

在 Maple 的图论工具包 `networks` 中，包含着大部分常用的图分析工具，在这一节中，将对它们作一个概要的介绍。

12.1.1 图对象及其建立

在 Maple 的 `networks` 工具包中，图对象是一个子程序（`procedure`），它根据不同的输入返回或设置图的不同信息。我们可以用 `networks` 中的 `new` 函数生成一个空图。

```
[> with(networks):
> new(G);
proc(x)
option GRAPH, 'I';
if x = _Edges then procname(_Edges) := { }
elif x = _EdgeIndex then procname(_EdgeIndex) := table(symmetric)
elif x = _Head then procname(_Head) := table( )
elif x = _Tail then procname(_Tail) := table( )
elif x = _Eweight then procname(_Eweight) := table( )
elif x = _Ends then procname(_Ends) := table( )
elif x = _Vertices then procname(_Vertices) := { }
elif x = _Vweight then procname(_Vweight) := table(sparse)
elif x = _Ancestor then procname(_Ancestor) := table( )
elif x = _Daughter then procname(_Daughter) := table( )
elif x = _Neighbors then procname(_Neighbors) := table( )
elif x = _Status then procname(_Status) := {'SIMPLE'}
elif x = _Emaxname then procname(_Emaxname) := 0
else RETURN('procname(args)')
fi
end
```

在这个新生成的空图中，我们可以用函数 `addvertex` 加入新的顶点（`vertex`）；而相应地，我们还可以用函数 `vertices` 查看图中的顶点：

```
[> addvertex({1, 2, 3, 4, 5}, G);
1, 2, 3, 4, 5
```

```
[> vertices(G);
{1, 2, 3, 4, 5}]
```

函数 `addvertex` 的返回值是加入的新顶点序列。而且图中的顶点不允许重复，如果所加的顶点在图中已经出现，就忽略该顶点，而不会出现重复的顶点。

```
[> addvertex(1, G);
> vertices(G);
{1, 2, 3, 4, 5}]
```

如果同时加入多个顶点，需要将它们写成一个集合或者有序表的形式。顶点的名称，即可以为数字，也可以为符号变量名。

Maple 中图对象的顶点也可以带权，可以在加入顶点得时候用 “`weights = 权值`” 的参数形式加以指明，其中权值也可以是符号变量或表达式。在默认情况下，顶点的权值为 0。

```
[> addvertex( [n1, n2], weights = [1, n2_weight], G );
n1, n2
> vertices(G);
{1, 2, 3, 4, 5, n1, n2}]
```

对于顶点的权值，可以用函数 `vweight` 查看，它可以查看一个图中的一个或多个顶点的权值，查看多个权值时，顶点名称以有序表的形式作为它的参数。

```
[> vweight( [1, n1, n2], G );
[0, 1, n2_weight]]
```

在图中加入边 (edge)，可以使用函数 `addedge`。加入边时，需要指定该边所联系的两个顶点。无向边的两个顶点用集合的形式给出，有向边的顶点则须包含在一个二元有序表中，边的起点是第一个顶点，终点是第二个。图对象中边的名称也可以在函数 `addedge` 中用 “`names = 边名称`” 的参数形式指定。但是，边的名称必须以字母 `e` 起始，默认情况下，Maple 按照次序用 `e1, e2, ……` 作为边的名称。

```
[> addedge( [ {3, 4}, [1, 1] ], G );
e1, e2
> addedge( [1, 3], names = edge13, G );
edge13]
```

可以用 `ends` 函数查看边所联系的两个顶点。对于无向边，返回的是边的端点所组成的集合，而对于有相变，则返回顶点的有序表。如果不指定具体的边，`ends` 将返回图中所有的边的端点对组成的集合。

```
[> ends(e1, G);
{3, 4}
> ends({e2, edge13}, G);
[[1, 1], [1, 3]]]
```

对于有向边，可以用 `head` 函数查看它的指向，也就是边的终点；用 `tail` 函数查看它的起点。如果在函数 `head` (或 `tail`) 中不给定具体的边名称，它们将返回所有有向边的终点 (起点) 所组成的映射表，映射关系是：边名称—边终点 (起点)。

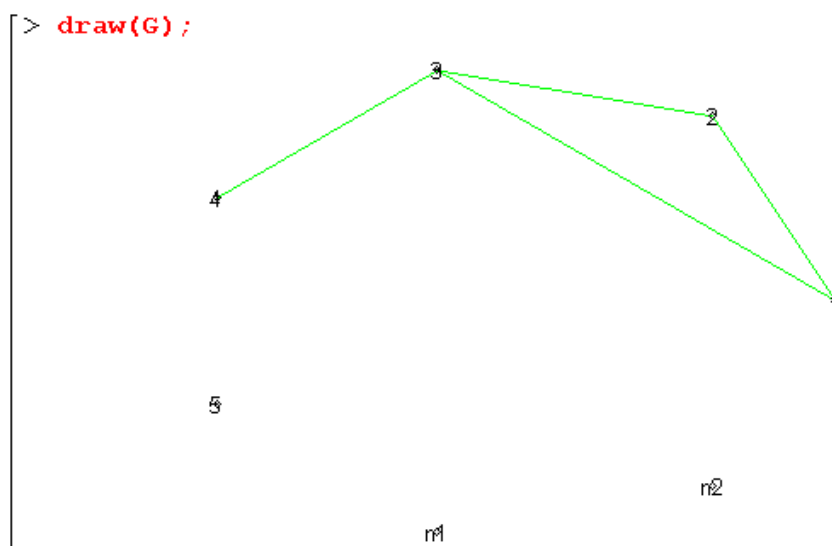
```
[> head(edge13, G);
                                     3
> tail(edge13, G);
                                     1
```

和顶点一样，图对象的边也可以加以权值，加入权值的方法也是在函数 `addedge` 中用 “weights = 权值” 指定。但是，和顶点有所不同，边的权值在默认情况下是 1 而不是 0。我们可以用 `eweight` 函数来查看图对象中一条或者多条边的权值。它的用法和 `vweight` 相似，此处不再赘述。

在图对象中加入边，不仅可以每一条独立的加入，也可以用路径或者环的方式加入，只需要在 `addedge` 中用参数 `Path(v1, ..., vn)` 或者 `Cycle(v1, ..., vn)` 指定路径或圈所经过的顶点名称 `v1, …, vn` 就可以了。不过，它们所加的边都只能是无向边。

```
[> addedge( Cycle(1, 2, 3), G );
                                     e3,e4,e5
```

虽然，我们可以用 `ends`、`head` 或者 `tail` 查看边和顶点的关系，但这毕竟缺乏直观性。Maple 也提供了一个工具 `draw`，可以用来画出图的结构形式。



美中不足的是，`draw` 工具还有很多缺点，首先，它不能显示有向边和无向边的区别，也不能显示边的名称；其次，对于有相同端点的不同边，在图形上不能加以区分；再次，对于两个端点相同的边，在图形上无法表达，特别是无向边，由于采用集合纪录边的顶点，导致该边只有一个顶点，在绘图时会出现错误。由于对两个端点相同的边，有向边和无向边在实际上没有区别，所以建议读者对于这种情况，采用有向边。

对于图对象中的元素——边和顶点，都可以用 `delete` 函数删除，只需要指明所需删除的顶点或者边的名称就可以了。例如：

```
[> delete( n1, G );
> vertices( G );
                                     {1, 2, 3, 4, 5, n2}
```

删除边，对于顶点不会有影响，但是，如果删除顶点，所有和这个顶点相连的边也都将同时被删除，例如：

```

[> ends( G );
      {{1,2},{3,4},{1,1},{1,3},{1,3},{2,3}}
[> delete( 1, G );
[> ends( G );
      {{3,4},{2,3}}
[> edges( G );
      {e1,e4}

```

除了上面介绍的用 `new` 生成空图，再用 `addvertex` 和 `addedge` 加入顶点和边的方法外，也可以调用函数 `graph` 由所有的顶点和边直接生成图对象。在使用熟练后，这种途径往往更为便捷。调用 `graph` 时，第一个参数是顶点的集合，第二个是边的集合，例如下面这个有向图：

```

[> G := graph( {seq(v[i], i=1..4)}, {[v[1], v[2]],
      [v[2], v[3]], [v[1], v[3]], [v[3], v[1]],
      [v[3], v[4]], [v[4], v[4]]} );
[> vertices(G);
      {v1,v2,v3,v4}
[> ends(G);
      {[v1,v2],[v2,v3],[v1,v3],[v3,v1],[v3,v4],[v4,v4]}

```

在图论中，我们称不含有边的图为零图，在 Maple 中，可以用 `void` 生成指定顶点个数的零图。例如：

```

[> N3 := void(3);
[> edges(N3);
      {}

```

由于图对象具有子程序类型，所以，每次赋值都仅仅是建立了该对象的一个引用。也就是说，如果将图 `G` 赋值给 `F` 后，`G` 发生了变化，那 `F` 中的信息也会跟着变化，因为它们所指的是一个对象。如果需要产生一个和原图对象相同的新对象，可以调用函数 `duplicate`，它将返回一个新的对象，是原对象的拷贝，而非引用。

12.1.2 有关图的基本概念

在一个无向图中，如果顶点 v_1 , v_2 为边 e 的端点，则称 e 与 v_1 (或 v_2) 是彼此关联的。在 `networks` 工具包中，有函数 `incident`，可以找出与一个顶点关联的所有边。例如，对于上一小节中生成的图 `G`，将它看成无向图，可以这样求所有与顶点 v_3 相关联的边：

```

[> incident( v[3], G );
      {e2,e3,e4,e5}
[> ends( %, G );
      {[v2,v3],[v1,v3],[v3,v1],[v3,v4]}

```

可以看到，`incident` 将 `G` 当作是无向图，不仅给出了以 v_3 作终点的边，也给出了以 v_3 作终点的边。对于有向图，可以在 `incident` 中加入可选参数 `In` 或 `Out`，求得以给定顶点为终点或起点的边。例如：

```
[> incident( v[3], G, In );
                                     {e2,e3}
```

函数 `incident` 的第一个参数不仅可以是单个的顶点，还可以是一组顶点的集合，这时，返回值将是所有这些顶点与其他顶点相关联的边的集合。

在一个无向图中，如果顶点 v_1, v_2 为边 e 的端点，则称 v_1 和 v_2 是彼此相邻的。在 Maple 中，一个顶点的相邻点集（邻域）可以用函数 `neighbors` 获得：

```
[> neighbors( v[3], G );
                                     {v1,v2,v4}
```

函数 `neighbors` 也可以不对点加以指定，那么，返回的将是所有顶点到其领域的映射表。

对于有向图，如果顶点 v_1, v_2 为边 e 的端点，而且 v_1 是 e 的起点， v_2 是 e 的终点。则称顶点 v_1 邻接到顶点 v_2 ，顶点 v_2 邻接于顶点 v_1 。一个图中所有邻接到顶点 v 的顶点集合称为点 v 的**先驱元集**，邻接于 v 的顶点集合称为 v 的**后继元集**。顶点的先驱元集和后继元集分别可以用函数 `arrivals` 和 `departures` 获得：

```
[> arrivals( v[3], G );
                                     {v1,v2}
[> departures( v[3], G );
                                     {v1,v4}
```

在一个图中，如果两条边的端点（对于有向边，起点和终点）都相同，则称它们为平行边；如果一条边的两个端点相同，称其为环。含平行边的图称为**多重图**（multigraph），称集不含平行边也不含环的图为**简单图**（simple graph）。在 Maple 中，可以用 `gsimp` 把多重图化成为简单图。它消去多余的平行边，去掉所有的环。

```
[> H := duplicate(G):
[> gsimp( H ):
[> ends( H );
                                     {[v1,v3],[v3,v4],[v1,v2],[v3,v1],[v2,v3]}
```

在一个无向图中，一个顶点作为边的端点的次数的和称为该顶点的**度数**；而对于有向图，一个顶点作为边的起点的次数称为**出度**，作为终点的次数称为**入度**。在 `networks` 中，分别有函数 `vdegree`，`outdegree` 和 `indegree` 和它们相对应。（注意，`vdegree` 只计算无向边的端点次数，对于全部都是具有向边的图，将返回 0。）

```
[> indegree( v[3], G );
                                     2
[> outdegree( v[2], G );
                                     1
```

一个图中的所有顶点的度数的最大值和最小值分别称为该图的**最大度**和**最小度**，在 Maple 中分别有对应的函数 `maxdegree` 和 `mindegree`。它们可以有两个参数，第二个参数是可选参数，可以用来返回最大度（或最小度）所在的顶点名称。和 `vdegree` 一样，它们只能用于无向图，也就是没有有向边的图。

我们用在 4 顶点完全图（参见本节后面部分）上加上 4 条环边的无向图作为例子：

```

[> G := complete(4):
  addedge( [{1}, {1}, {2}, {3}], G):
[> maxdegree(G, large);
                                     5
[> large;
                                     1
[> mindegree(G);
                                     3

```

对于一个给定的图，每个顶点的度数所组成的序列是唯一确定的（Maple 中的对应函数 `degreeseq`）；反过来，对于一个给定的非负整数序列 d ，如果存在一个图的度数序列与其相同，则称 d 为可图化的。如果存在以 d 为度数序列的简单图，则称 d 为可简单图化的。可以用 `networks` 中的函数 `graphical` 加以检验。如果是可图化的（可简单图化的），它们返回一个对应的图的所有边，否则则返回 FAIL。

不加参数时，`graphical` 检验的是数列是否可以简单图化。

```

[> graphical( [5, 5, 3, 3, 2, 2, 2] );
[ [ {1,2}, {1,4}, {1,3}, {1,7}, {1,5}, {2,6}, {2,3}, {2,4}, {2,5}, {4,7}, {3,6} ]

```

对这个结果，我们可以生成一个图加以检验：

```

[> G := void(7):
[> addedge( %, G ):
[> degreeseq(G);
                                     [2, 2, 2, 3, 3, 5, 5]

```

如果要检验是否可图化，也就是允许多重图，可以加入参数 `MULTI`。

```

[> graphical( [2, 3, 3, 4, 4, 5], MULTI );
                                     FAIL
[> graphical( [1, 2, 3, 3, 5], MULTI );
[ [ {4,5}, {3,5}, {4,5}, {3,5}, {2,5}, {2,3}, {1,4} ]

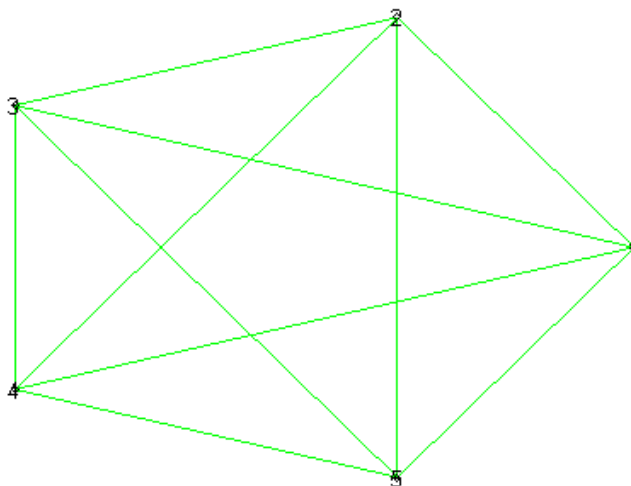
```

在图论中，我们把这样的 n 阶无向简单图称为 n 阶完全图，它的每一个顶点都和剩余的 $n-1$ 个顶点相邻。在 Maple 中，可以用 `networks` 中的函数 `complete` 生成完全图。

```

[> K5 := complete( 5 );
[> draw( K5 );

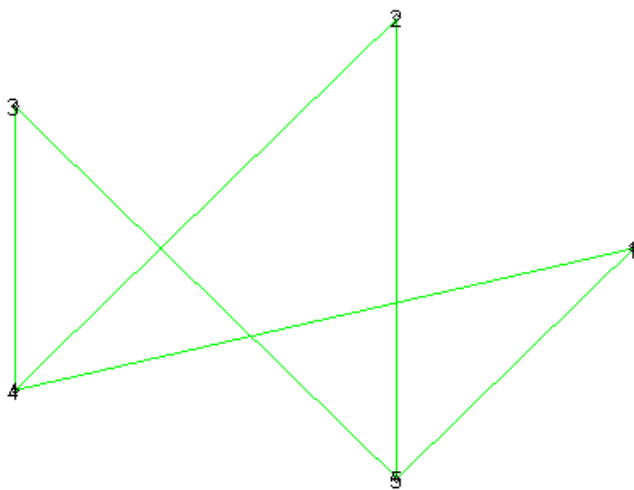
```



函数 `complete` 中, 可以给定单个整数作参数指定完全图的阶数, 也可以给定顶点名称的集合。

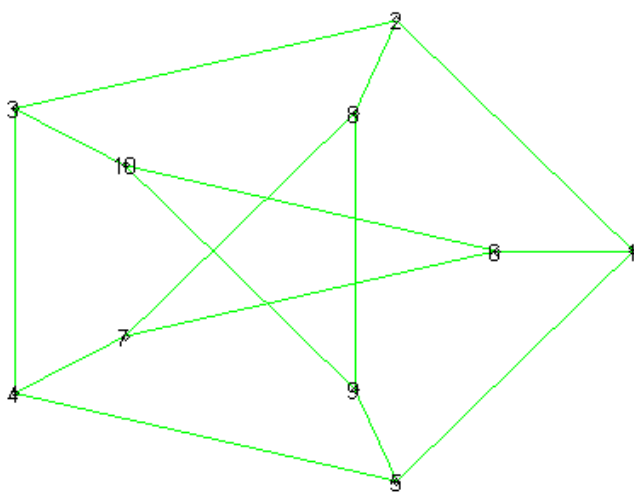
在一个无向图 G 中, 如果顶点集合 V 可以分成 r 个互不相交的子集 V_1, V_2, \dots, V_r , 使得 G 中的每一条边的两个端点都不在同一个 V_i 中, 则称 G 为 **r 部图**。若对于任意的 $i \neq j$, V_i 中任何一个顶点都与 V_j 中所有的顶点相邻, 则称 G 为 **r 部完全图**。函数 `complete` 也可以用来生成 r 部完全图, 只需要指定每一部中的顶点个数。例如:

```
[> K32 := complete( 3, 2 );
> draw( K32 );
```



设 G 为 n 阶 ($n \geq 1$) 无向简单图, 如果 G 中的任意顶点的度数都为 k , 则称 G 为 **k -正则图**。我们知道, 零图总是 0 -正则图, n 阶无向完全图总是 $(n-1)$ -正则图。Maple 中还有一些特殊的正则图, 例如著名的彼得森 (Petersen) 图, 可以用函数 `petersen()` 直接获得, 它是一个 3 -正则图。

```
[> P := petersen();
> draw(P);
```



```
[> degreeseq(P);
```

```
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

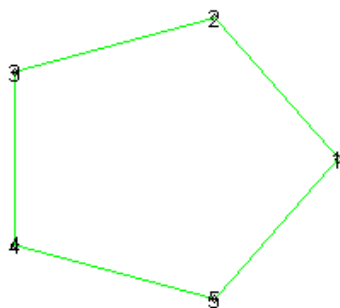
还有柏拉图 (Plato) 图, 在 Maple 中予以实现的有立方体图 `cube`, 八面体图 `octahedron`, 十二面体图 `dodecahedron`, 二十面体图 `icosahedron`。

```
[> Oct := octahedron():
> degreeseq(Oct);
[4, 4, 4, 4, 4, 4]
> Dec := dodecahedron():
> degreeseq(Dec);
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

设图 G 的顶点集为 V , $V_1 \subset V$, 称以 V_1 为顶点集, G 中两个端点都在 V_1 中的边组成的集合为边集组成的图, 为 G 的 V_1 导出的子图; 对于 G 的边集的子集 E_1 , 称以 E_1 为边集, E_1 的端点组成的集合为顶点集的图, 为 G 的 E_1 导出的子图。在 Maple 中, 可以用函数 `induce` 得到由边子集或者顶点子集导出的子图:

作为例子, 我们求前面得到的彼得森图的顶点 $\{1, 2, \dots, 5\}$ 导出的子图:

```
[> P1 := induce({1, 2, 3, 4, 5}, P):
> draw(P1);
```



对于一个无向图 G , 我们称这样的运算为边 e 的**收缩**: 在 G 中除去 e , 将 e 的两个端点 u, v 用一个新的顶点 w 代替, 并使 w 关联除 e 外 u, v 关联的一切边。同样也有关于顶点集 V 的子集 V_1 的**收缩**, 在原图中将所有 V_1 中的顶点用一个新的顶点代替, 并使之与所有 $V \setminus V_1$ 中与 V_1 相邻的顶点也相邻。Maple 中这两种运算分别有对应的函数 `contract` 和 `shrink`。

```
[> contract({4, 5}, P1):
> ends(P1);
{(1, 2), (2, 3), (3, 4), (1, 4)}
> shrink({6, 7, 8, 9, 10}, P):
> ends(P);
{(1, 6), (1, 2), (2, 3), (3, 4), (4, 5), (1, 5), (4, 6), (2, 6), (5, 6), (3, 6)}
```

12.1.3 图的连通性

要刻画无向图的连通度, 首先需要引入**割集** (cutset) 的概念。这里给出一个不严格的定义, 我们把这样的集合称作边割集 (简称割集), 在原图中去掉这些边, 恰好可以使图的连通分支数增加。在 Maple 中, 可以用函数 `countcuts` 求出一个图对象中具有最少元素的割集个数。

作为例子, 我们用 `cycle` 生成一个包含 6 个顶点的圈, 再用函数 `countcuts` 求它的最小割集的个数。直观地, 可以知道, 圈中的任意两条边都可以组成它的割集, 所以, 最小割集个数为 $C_6^2 = 15$ 个, Maple 也给出了相同的结果:

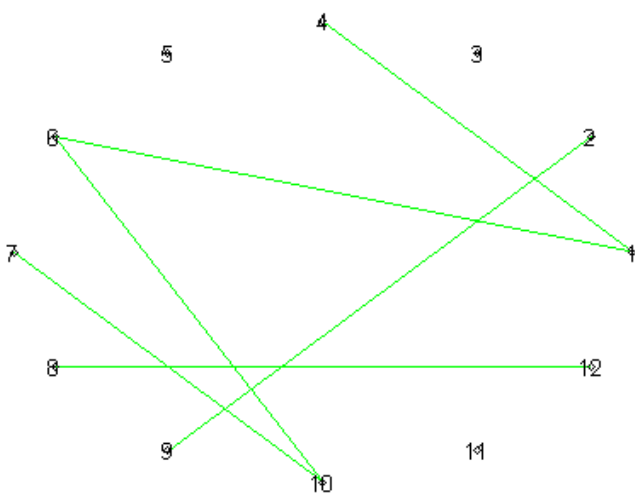

```
[> C := cycle(6):
> ends(C);
      {{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}, {1, 6}}
> countcuts(C);
      15
```

有了边割集，就可以定义无向图的边连通度了。我们把最小割集的元素个数称为一个图的**边连通度**（edge connectivity）。在 Maple 中，对应的函数是 `connectivity`。

```
[> connectivity(C);
      2
```

Maple 的图论工具包中，还有函数 `components`，它可以求出图的连通分支。作为例子，我们先用 `random` 生成一个 12 个顶点，6 条边的随机图。

```
[> G := random(12, 6):
> draw(G);
```



再用 `components` 求它的各连通分支。它返回的是一个集合的集合，每一个小集合是一个连通分支中的顶点集。

```
[> components(G);
      {{5}, {3}, {1, 4, 6, 7, 10}, {2, 9}, {11}, {8, 12}}
```

对于连通度为 1 的图，我们总可以找到一些割集，它们都只含有一条边，并将图分割称为连通度大于 1 的分支，称这些边为**桥**。在 Maple 中，可以利用函数 `bicomponents` 找出图中的桥，并得出剩余的多连通分支。它返回两个集合组成的有序表，其一是所有的桥组成的集合，其二是各连通分支的边集组成的集合。

例如，对于下面这个单连通图：

```
[> G := void(9):
> addedge( Cycle(1, 2, 3), G ):
> addedge( Cycle(4, 5, 6), G ):
> addedge( Cycle(7, 8, 9), G ):
> addedge( {{3, 4}, {6, 7}}, G ):
```

它的图形如下所示。用 `bicomponents` 就可以求得其中的桥 $\{\{3, 4\}, \{6, 7\}\}$ 和三个多连通分支了。

```
> draw(G);
```

```
> bicomponents(G);
```

```
[[{e10,e11}], [{e1,e2,e3}, {e4,e5,e6}, {e7,e8,e9}]]
```

12.1.4 树

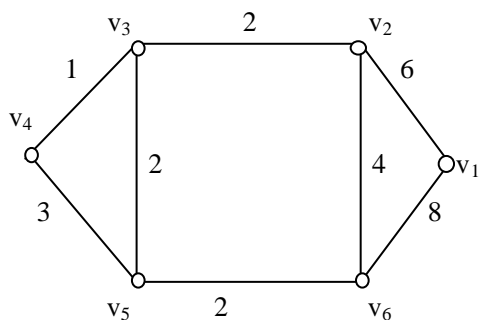
图论中，我们将连通无回路的图称作树。如果图 G 的子图 T 包含 G 中所有顶点，而且为树，则称 T 为 G 的**生成树** (spanning tree)。

图论工具包 `networks` 中的函数 `counttrees` 可以求出一个图的所有生成树的个数。

对于带权图，称其带权最小的生成树为**最小生成树**（minimum weight spanning tree）。最小生成树是有其实际意义的，例如将城市看作图的顶点，城市间的道路看作连接顶点的边，道路的造价就是边的权值，那么最小生成树就是连接这些城市的最低造价公路网。

给定一个图对象，`networks` 中的函数 `spantree` 就可以得到它的最小生成树。结果是作为一个新的图对象给出的。

例 11.1 求下图所示带权图的最小生成树



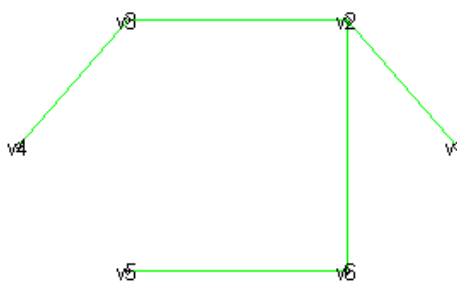
首先建立这个带权图:

```
[> restart;
[> with(networks): new(G):
[> addvertex( {seq(v.i, i=1..6)}, G ):
[> addedge( [{v1, v2}, {v2, v3}, {v3, v4}, {v4, v5},
[>           {v5, v6}, {v6, v1}, {v2, v6}, {v3, v5}],
[>           weights = [6, 2, 1, 3, 2, 8, 2, 4], G ):
```

注意, 由于 `spantree` 需要图的边按照默认的 `e1, e2, …` 的方式命名, 所以, 我们先用 `restart` 重置了环境, 以防止已有的边量的干扰。

然后, 调用 `spantree` 求它的最小生成树:

```
[> T := spantree(G):
> draw(T);
```



可以在 `spantree` 中加入参数, 指定生成树的根, 还可以返回最小生成树的总权值:

```
[> T1 := spantree(G, v3, 'w'):
> w;
```

13

和最小生成树不同, **最短路径生成树** (shortest path spanning tree) 是由图中的一个顶点出发, 到其他各顶点的最短路径所构成的生成树。在 Maple 中, 对应的函数是 `shortpathstree`。它采用的是著名的计算机学家 E. W. Dijkstra 在 1959 年给出的最短路径算法。它的返回值也是一个新的图对象, 其中每个顶点的权值是从根到该顶点的距离, 也就是最短路径的权值和。

```
[> G := petersen():
> T := shortpathstree(G, 1):
> vweight([1, 2, 7, 10], T);
```

[0, 1, 2, 2]

对于有向树, 在 Maple 中, 可以用函数 `ancestor` 和 `daughter` 分别求出它的一个或多个顶点的祖先和后代。例如对于上面所得的彼得森图的最短路径生成树, 可以这样求它的顶点的祖先或后代:

```
[> ancestor([1, 2, 5], T);
[(), {1}, {1}]
> daughter(1, T);
{2, 5, 6}
```

如果不给定具体的顶点名称, 它们将返回一个映射表。

Maple 的函数 `path` 可以在有向树中找到从一个顶点到另一个顶点的路径, 如果没有路径存在, 将返回 `FAIL`。但是, 它依赖于有向树的祖先和后代映射表, 这两个映射表只有在调用 `spantree` 或者 `shortpathstree` 之后才会存在, 而且如果又进行了其他图操作 (例如增删顶点或边), 它们将不复存在。

例如, 我们可以利用上面得到的最短路径生成树, 求出从顶点 1 到顶点 7 的最短路径:

```
[> path([7, 1], T);
[7, 6, 1]
```

由于树是有向的, 所以反过来就没有路径存在了:

```
[> path([1, 7], T);
                                     FAIL
```

12.1.5 图的矩阵表示

图的**关联矩阵** (incidence matrix) 是用来表示图的顶点和边的关系的, 对于无环图, 设其边集为 $E = \{e_1, e_2, \dots, e_m\}$, 顶点集为 $V = \{v_1, v_2, \dots, v_n\}$, 则其关联矩阵的元素 m_{ij} 定义如下:

$$m_{ij} = \begin{cases} 0 & v_i \text{ 与 } e_j \text{ 不关联} \\ -1 & v_i \text{ 是有向边 } e_j \text{ 的起点} \\ 1 & \text{其余情况} \end{cases}$$

在 Maple 中, 可以用 incidence 函数求得图对象的关联矩阵, 例如:

```
[> G := void(4):
> addedge([ [2, 1], [2, 1], [1, 3], [2, 3], [3, 4],
            [4, 3], {1, 4} ], G);
                                     e1, e2, e3, e4, e5, e6, e7
> incidence(G);
                                     [ 1  1 -1  0  0  0  1 ]
                                     [-1 -1  0 -1  0  0  0 ]
                                     [ 0  0  1  1 -1  1  0 ]
                                     [ 0  0  0  0  1 -1  1 ]
```

图的**邻接矩阵** (adjacency matrix) 是这样定义的, 它的元素 a_{ij} 是从 v_i 邻接到 v_j 的边的条数。它对于任何形式的图都适用, 而不限于无环图。在 Maple 中, 相应的函数是 adjacency :

```
[> adjacency(G);
                                     [ 0  0  1  1 ]
                                     [ 2  0  1  0 ]
                                     [ 0  0  0  1 ]
                                     [ 1  0  1  0 ]
```

12.1.6 平面图形的判断

如果图 G 可以这样画在一个平面上, 出了顶点外, 处处无边相交, 则称 G 为**平面图**。在 Maple 中, 可以用函数 isplanar 检验一个图对象是否为平面图。该函数在执行过程中会自动生成一个原图的拷贝, 在其上作简化运算, 所以不会改变原图。

```
[> isplanar(icosahedron());
                                     true
> isplanar(complete(5));
                                     false
```

12.1.7 图的着色

图的着色问题也是图论中一个经常研究的问题。对于一个无环无向图，对其顶点的 k 着色，是指用 k 中颜色给她所有的顶点着色，使相邻的顶点涂有不同的颜色。对一个图的两个 k 着色被认为是不同的，是指至少有一个顶点在两个 k 着色中被涂称不同的颜色。如果用 $f(G, k)$ 表示图 G 的不同 k 着色方式的总数，则称 f 为 G 的**着色多项式**(chromatic polynomial)。

在 Maple 中，无环无向图的的着色多项式，可以用函数 `chrompoly` 获得。它的调用形式是：**chrompoly** (G , λ), 其中 G 是一个无向图， λ 是着色多项式的自边量，也就是颜色数。例如，我们用它求彼得森图的着色多项式：

```
[> G := petersen();
> chrompoly(G, x);
      x(x-1)(x-2)(x7-12x6+67x5-230x4+529x3-814x2+775x-352)
```

12.2 布尔运算和数理逻辑

Maple 中的 `logic` 工具包中包含了常用的逻辑运算和逻辑表达式化简函数，和一般的编程语言不同，它们不仅可以计算和化简逻辑常量所构成的表达式，还可以化简符号变量构成的逻辑表达式。在这一节中将对它们作简要的介绍。

12.2.1 基本的布尔运算

Maple 的 `logic` 工具包中实现了一些布尔运算符，它们都是扩展的运算符，以字符 `&` 起始。需要注意的是，和工具包中的函数一样，是用这些运算符之前也必须载入 `logic` 工具包。这里将这些运算符列成表 11.1。

表 12.1 布尔运算符

运算符	对应的运算	参数个数
<code>&and</code>	与	2 个或多个
<code>&or</code>	或	2 个或多个
<code>&not</code>	非	1 个
<code>&iff</code>	同或	2 个
<code>&nor</code>	或非	2 个或多个
<code>&nand</code>	与非	2 个或多个
<code>&xor</code>	异或	2 个
<code>&implies</code>	蕴含	2 个

对于逻辑表达式，在默认情况下，Maple 不会自动化简。但是，我们可以通过函数 `environ` 设置系统对于逻辑表达式的自动化简程度。它有一个参数，可以设置为 0~3 的整数，对应的化简程度如下：

✧ 0 —— 不进行任何自动化简（系统默认状态）；

- ◇ 1 ——利用结合律，自动消去多余的括弧，化成`&and`，`¬`，`&or`表示的形式；
 - ◇ 2 ——在 1 的基础之上，利用 $a \text{ \&and } a \Rightarrow a$ ， $a \text{ \&or } a \Rightarrow a$ ，以及布尔常量 `true` 和 `false` 进一步化简；
 - ◇ 3 ——将结果化为模 2 的代数运算表达式。
- 通过几个简单的例子，就可以看出它们的区别：

```
[> with(logic):
[> environ(0);
[> a &and b &and a, a &iff a;
                                (a &and b) &and a, a &iff a
[> environ(1);
[> a &and b &and a, a &iff a;
                                &and(b, a, a), (&not(a) &and &not(a)) &or (a &and a)
[> environ(2);
[> a &and b &and a, a &iff a;
                                b &and a, a &or &not(a)
[> environ(3);
[> a &and b &and a, a &iff a;
                                a2 b, 2 a + 1
```

不过，所有这些自动化简都是初步的化简，如果需要对逻辑表达式进行进一步化简（例如，去掉上面例子中的 $a \text{ \&or } \text{not}(a)$ ， $2a + 1$ 一类的表达式），可以调用函数 `bsimp`，它是专门用于布尔表达式的化简的。

为了看清楚结果，我们仍将自动化简程度设为 0。

```
[> environ(0);
[> bsimp( (a &and b &and a) &or &not a );
                                b &or &not(a)
[> bsimp( a &iff a );
                                true
```

12.2.2 其他逻辑函数

在 `logic` 工具包中，还有一些函数可以处理逻辑表达式。

函数 `bequal` 可以用于判别两个逻辑表达式是否相等。例如：

```
[> bequal(a &iff (a &or b), b &implies a);
                                true
```

函数 `bequal` 还可以加入一个未赋值的附加参数，用来返回使两个表达式不相等的一组条件，如果相等，则返回 `NULL`。不过，这有时会明显地降低函数的运行速度。

```
[> bequal(a &implies b, b &implies a, 'p');
                                false
[> p;
                                {b = false, a = true}
```

展开一个逻辑表达式，可以用函数 `bdstrib`，它可以将表达式展开为与或式。这个函数只是简单地利用分配律和摩根律，所以其结果并不能保证是最简形式。`bdstrib` 还可以有一个附

加参数，它可以将结果对于指定的变量化成析取范式（canonical disjunctive normal form）。

```
> distrib(&and(a,b &or c));
      (a &and b) &or (a &and c)
> distrib(&not(a &or b), {a,b,c});
      &and(&not(a), &not(b), c) &or &and(&not(a), &not(b), &not(c))
```

如果要将表达式化成析取或合取范式，可以调用函数 `canon`。它的调用格式是：

canon (b, alpha, form)

其中，b 是一个逻辑表达式，alpha 是变量集合，form 是可选参数，可以用以下几种：

- ✧ MOD2 化成模 2 的代数运算式，此时 alpha 不起作用，但仍须提供；
- ✧ CNF 化成合取范式（conjunctive normal form）
- ✧ DNF 化成析取范式（默认）

```
> canon(a &nor b, {a,b}, MOD2);
      1+a+b+a b
> canon(a &nor b, {a,b}, CNF);
      &and(&not(a) &or &not(b), &not(a) &or b, &not(b) &or a)
> canon(a &nor b, {a,b}, DNF);
      &not(a) &and &not(b)
```

利用 `logic` 中的函数 `randbool`，可以用给定的变量随机生成一个逻辑表达式，也可以指定表达式为上面的几种范式之一。例如：

```
> randbool({a,b});
      &or(b &and a, &not(a) &and &not(b), a &and &not(b))
> randbool([a,b,c], DNF);
      &or(&and(b, a, c), &and(a, &not(b), &not(c)), &and(b, &not(a), c),
      &and(a, &not(b), c), &and(&not(a), &not(b), c))
```

利用函数 `dual`，可以把逻辑表达式化成对偶形式的表达式，所谓对偶形式的表达式，就是把所有 `true` 和 `false` 出现的地方都分别换成 `false` 和 `true`，再把 `&and` 换成 `&or`，`&or` 换成 `&and`，其余的逻辑运算符都保持不变。

```
> dual(a &and (&not a) = false);
      a &or &not(a) = true
> dual(a &implies b);
      a &implies b
```

但是，如果设置了 `environ`，在调用之前会对表达式自动化简。例如：

```
> environ(2):
      dual(a &implies b);
      b &and &not(a)
```

在数理逻辑中，我们把无条件为真的表达式称为重言式（或者永真式，`tautology`）。在 `Maple` 中，可以用函数 `tautology` 检验一个表达式是否为重言式。如果不是的话，它还可以通过附加参数返回使表达式不成立的一组条件，和 `bequal` 一样，带返回参数的函数会明显降低运行速度。

```

[> tautology(&and(a,b) &or (&not a) &or (&not b));
                                     true
[> tautology((a &iff b) &or b, 'p');
                                     false
[> p;
                                     {a = true, b = false}

```

12.3 群论

群是一类重要的代数系统，它在很多领域都有着广泛的应用。在 Maple 中，有一个专门进行与群论相关的符号运算的工具包 `group`，在这一节中将予以简要介绍。

12.3.1 群的表示

我们知道，群是这样的一种代数系统，它包含一个二元运算，该运算满足结合律；对于该二元运算，群中存在单位元；群中的每一个元素都有逆元。

置换群 (permutation group) 是我们在群论中经常研究的一类重要的群。 n 元置换群中的每一个元素都是一个 n 元置换，而在这些元素上的二元运算对应的是置换的合成。在 Maple 中，一个 n 元置换群是用这样的函数调用来表示的：

permgrou (n, gens)

其中， n 必须是一个非负整数；`gens` 是群的生成元的集合。每个生成元表示一个或几个不相交的轮换，而最后得到的群是这些轮换的乘积所张成的置换群。每个轮换都用一个有序表表示，例如 $[a_1, a_2, \dots, a_n]$ 表示这样的轮换：将 a_1 映射成 a_2 ， a_2 映射成 a_3 ， \dots ， a_{n-1} 映射成 a_n ， a_n 映射成 a_1 。表中不出现的代表元素都映射成本身。相应的，置换群中的单位元——恒等映射，就用一个空有序表 `[]` 表示（当然这在表示通常的群时是不需要的，如果要表示只有单位元的平凡群就有必要了）。

```

[> with(group):
[> permgrou(6, {[[1,2], [3,4,6]], [[1,2,3,4,5,6]]});
                                     permgrou(6, {[[1,2],[3,4,6]],[[1,2,3,4,5,6]]})

```

在 `gens` 中，可以为每个生成元或者几个不相交的生成元命名，只需要在 `gens` 中用等式表示即可。但是，生成元之间不可以相互嵌套。

```

[> permgrou(5, {a=[[1,2], [4,5]], b=[[5,4,3,2,1]]});
                                     permgrou(5, {a=[[1,2],[4,5]], b=[[5,4,3,2,1]]})

```

轮换中的代表元素必须是从 1 到 n 的整数，否则会导致错误：

```

[> permgrou(4, {a=[[1,2], [4,5]], b=[[5,3,2,1]]});
Error, (in permgrou) invalid parameters

```

函数 `permgrou` 在成功时只给出形式，而不给出群的其他信息。如果我们要检验一个特定的元素是否在置换群中，可以调用函数 `groupmember`。它的返回值是布尔类型的。


```
[> pg := permgroup( 7, { [[1,2,3]], [[3,4,5,6,7]] } );
      pg := permgroup(7, {[[1, 2, 3]], [[3, 4, 5, 6, 7]]})
[> groupmember( [[1,5], [3,6]], pg );
      true
```

还可以用函数 RandElement 随机地获得置换群中的一个元素。

```
[> RandElement(pg);
      [[1, 6, 3, 7, 4, 5, 2]]
```

置换群中每个元素都是一个轮换，它的逆可以用 invperm 函数获得；如果要求两个轮换的乘积（也就是它们的合成），可以调用函数 mulperms。

12.3.2 有关群的概念

在群论中，我们将满足交换律的群称作**交换群**（abelian group）。在 Maple 中，可以用函数 isabelian 判断一个置换群是否交换群，例如：

```
[> permgroup(8, { [[1,2]], [[1,2,3,4,5,6,7,8]] }):
[> isabelian(%);
      false
[> permgroup(5, { [[1,2], [3,5]] }):
[> isabelian(%);
      true
```

群中的元素个数称为群的**阶**（order），可以用函数 grouporder 求得：

```
[> grouporder(permgroup(7, { [[1,2,3]], [[3,4,5,6,7]] }));
      2520
```

如果一个群的子集也构成群，我们称这个子集为它的**子群**（subgroup）。在 Maple 中，可以用函数 issubgroup 检验一个置换群是否为另一个置换群的子群。

```
[> pg := permgroup(8, { [[1,2]], [[1,2,3,4,5,6,7,8]] }):
[> sg := permgroup(8, { [[1,2,3,4]], [[1,2]], [[5,6,7,8]] }):
[> issubgroup(sg, pg);
      true
```

设 G 是一个群，令 C 为 G 中满足交换律的元素集合，即

$$C = \{a \mid a \in G \text{ 且 } \forall x \in G (ax = xa)\}$$

则 C 为 G 的子群，称 C 为 G 的**中心**（center）。在 Maple 中，可以用函数 center 求得一个置换群的中心。

```
[> pg := permgroup(8, { [[1,2,4], [5,6]], [[5,6,7,8]] }):
[> center(pg);
      permgroup(8, {[[1, 2, 4]]})
```

设 H 是群 G 的子群， $a \in G$ ，称 $Ha = \{ha \mid h \in H\}$ 为 H 在 G 中的一个**右陪集**（right coset）。可以用函数 coset 获得一个子群的所有右陪集：

```
[> pg1 := permgroup(7, { [[1,2]], [[1,2,3,4,5,6,7]] }):
    pg2 := permgroup(7, { [[1,2,3]], [[3,4,5,6,7]] }):
    cosets(pg1, pg2);
      {[ ], [[6, 7]]}
```

由于平凡群(只含有单位元的群)对于群中任何元素的右陪集都是元素本身组成的集合。所以,可以借用 `cosets` 函数,求出置换群中的所有元素:

```
[> pg := permgroup(4, {[[1,2]], [[1,4]]}):
[> identity := permgroup(4, {[[]]}):
[> cosets(pg, identity);
      [[], [[1,2]], [[1,4]], [[1,2,4]], [[2,4]], [[1,4,2]]]
```

如果群 G 的子群 H 满足对于任意 G 中的元素 a 都有 $H a = a H$, 则称 H 为 G 的正规子群 (normal subgroup)。在 Maple 中, 检验一个子群是否正规子群, 可以用函数 `isnormal`。它的第一个参数是一个群, 第二个参数是它的一个子群, 例如:

```
[> pg := permgroup(8, {[[1,2]], [[1,2,3,4,5,6,7,8]]}):
[> sg := permgroup(8, {[[1,2,3]], [[3,4,5,6,7,8]]}):
[> isnormal(pg, sg);
      true
```

如果 G 是一个群, H 为其子群, 那么, 称 $N(H) = \{x | x \in G \text{ 且 } Hx = xH\}$ 为 H 的正规化

子 (Normalizer)。Maple 函数 `normalizer` 可以得到置换群的子群的正规化子:

```
[> pg := permgroup(7, {[[1,2,3]], [[3,4,5,6,7]]}):
[> sg := permgroup(7, {[[1,2,3]], [[3,4,5]]}):
[> normalizer(pg, sg);
      permgroup(7, {[[1,2,3]], [[4,5]], [[6,7]], [[3,4,5]]})
```

一个子群中包含的最大的正规子群称作它的核 (core), 可以用 Maple 函数 `core` 求得。函数的调用格式如下:

core (sg, pg)

其中, `sg` 和 `pg` 是两个置换群; 在这里, `sg` 并不必须是 `pg` 的子群, 但是为了加快运算速度, 推荐使用 `pg` 的子群作为参数。

```
[> pg := permgroup(7, {[[1,2]], [[1,2,3,4,5,6,7]]}):
[> sg := permgroup(7, {[[1,2,3]], [[3,4,5,6,7]]}):
[> core(sg, pg);
      permgroup(7, {[[1,2,3]], [[3,4,5,6,7]]})
```

与核相对应, 称包含一个子群的最小正规子群为其正规闭包 (normal closure), 对应的函数是 `NormalClosure`。例如上面例子中 `pg` 的子群 `sg`, 由于它是正规子群, 所以它的核与正规闭包都是它本身:

```
[> NormalClosure(sg, pg);
      permgroup(7, {[[1,2,3]], [[3,4,5,6,7]]})
```

两个置换群的交 (intersection), 在 Maple 中可以用函数 `inter` 求得:

```
[> pg1 := permgroup(7, {[[2,3,4]], [[3,4,5,6,7]]}):
[> inter(pg, pg1);
      permgroup(7, {[[2,4,3]], [[5,7,6]], [[4,6,5]], [[3,5,4]]})
```

12.4 组合数学

Maple 中有两个与组合数学相关的工具包。其一是 `combstruct`，这个工具包中定义了组合数学中常用的数据结构以及基于这些结构的组合数学函数；其二是 `combinat`，它包括组合数学中一些常用的方法。

12.4.1 组合数学的基本数据结构

在 Maple 的 `combstruct` 工具包中，预定义了以下的四种基本结构，它们是组合数学中常用的研究对象，在调用这个工具包中的其他函数时都将直接或间接用到这些结构。在对这些结构调用函数时，有以下的调用格式：

function (struct (args), size = n)

其中，`function` 是调用的函数名，`struct` 使这里介绍的组合结构，`args` 是结构的参数。可以用可选参数 `size = n` 指定它们的大小，如果不予指定，将采用它们的默认大小。

- ✧ **Combination (或 Subset)** —— 元素的组合，它的参数表示所有的元素，可以是一个有序表，一个集合，或者一个正整数。如果以正整数 `n` 作参数，就表示元素集合为 $\{1, 2, \dots, n\}$ 。默认大小为 `'allsizes'`，表示大小任意。
- ✧ **Permutation** —— 元素的排列，参数可以为一个有序表，一个集合，或者一个正整数。如果以正整数 `n` 作参数，就表示元素集合为 $\{1, 2, \dots, n\}$ 。默认大小为有序表或者集合的元素个数。
- ✧ **Partition** —— 将一个正整数分解为一组正整数的和，不关心它们的先后排列次序。默认大小为 `'allsizes'`。
- ✧ **Composition** —— 将一个正整数分解为一组正整数的和，并按一定次序的排列。默认大小为 `'allsizes'`。

利用这些基本结构和 `combstruct` 中的一些函数，就可以解决简单的排列组合问题了。首先需要载入 `combstruct` 工具包：

```
[> with(combstruct):
```

最简单地，可以用 `count` 函数对于这四种结构进行计数。

例 11.2 求集合 $\{a, b, c, d\}$ 的子集个数

由于要求所有子集的总数，所以大小应该取 `Combination` 结构的默认大小，即对任意大小求和。`Combination` 和 `Subset` 是相同的结构名称。

```
[> count(Subset({a, b, c, d}));
16
```

例 11.3 求从三个元素 a, a, b 中取两个的排列总数

由于这里有重复元素，所以不能再用作参数（否则将略去重复的元素）。而且由于取 2 个元素的排列不是默认大小（默认为全排列），需要用 `size = 2` 指明。

```
[> count(Permutation([a, a, b]), size = 2);
3
```

除了这四个基本结构外，Maple 还允许用户使用自己定义组合数据结构。首先需要定义数据结构的名称。同时，为了使自定义的组合结构为 Maple 的组合数学函数所识别，需要自己定义几个基本的函数，例如，需要定义名为 `Mystruct` 的组合结构，需要定义以下几个函数：``combstruct/count/Mystruct``，``combstruct/draw/Mystruct``，``combstruct/allstructs/Mystruct``，``combstruct/iterstructs/Mystruct``。注意，由于函数名中含有特殊字符“/”，所以需要一对反向撇号“`”将它们括起来。自定义结构和函数的具体定义方法请参见 8.4 中相应部分。

12.4.2 组合结构元素的获取

利用函数 `draw`，可以从组合结构中随机地取出一个元素。例如，我们需要随机地寻找 5 个正整数，它们的和是 20，可以这样做：

```
[> draw(Partition(20), size=5);
      [1, 2, 4, 6, 7]
```

Maple 不仅可以随机地取出一个组合结构元素，还可以获得所有的组合结构元素，只需要调用 `combstruct` 工具包中的函数 `allstructs`。例如，可以这样获得所有和为 3 的正整数的排列：

```
[> allstructs(Composition(3));
      [[1, 2], [1, 1, 1], [2, 1], [3]]
```

更常用的方法是生成一个迭代函数，每次调用这个迭代函数都会返回一个不同的结构元素。您也许还记得在第八章中，作为程序设计例子，我们编写了一个程序，它生成的函数每次返回一组集合的笛卡尔积中的一个元素。这里的迭代函数也是用这种方法实现的。

首先，先要用函数 `iterstructs` 建立一个迭代结构。例如，我们要求 `x`, `y`, `z` 的全排列的所有元素：

```
[> allp := iterstructs(Permutation([x, y, z]));
```

然后，以这个迭代结构为参数调用函数 `nextstruct`，就可以依次获得每一个结构元素：

```
[> nextstruct(allp);
      [x, y, z]
```

可以用函数 `finished` 检查是否已经生成完所有的结构元素，如果生成完毕，它将返回 `true`。在生成完所有的结构元素后，迭代结构也就完成了它的使命，不可以再次被使用了。

```
[> while not finished(allp) do nextstruct(allp) od;
      [y, x, z]
      [z, x, y]
      [x, z, y]
      [y, z, x]
      [z, y, x]
```

12.4.3 组合类

利用我们前面介绍的基本数据结构可以处理比较简单的组合数学问题，对于更复杂的组合数学问题，Maple 中有更完善的数据结构可以进行处理，在 Maple 中，将这些数据结构称为组合类（combinatorial class）。组合类可以是一个基本类——`Epsilon` 或者 `Atom`，也可以

是通过构造函数形成的对象。在 `combstruct` 工具包中，有以下的组合类构造函数：

- ✧ **Epsilon** 空类——大小为 0 的对象；
- ✧ **Atom** 原子类——大小为 1 的对象，变量 `Z` 是一个预定义的原子类；
- ✧ **Union** (`A, B, ...`) 类 `A, B, ...` 的不相交并（表示或者取 `A` 或者取 `B`）；
- ✧ **Prod** (`A, B, ...`) 类 `A, B, ...` 的直积（表示取一个 `A`，一个 `B`，... 的排列）；
- ✧ **Set** (`A`) 所有由 `A` 中元素所组成的可重复集（多重集）；
- ✧ **PowerSet** (`A`) 所有由 `A` 中元素所组成的不可重复集，亦即 `A` 的幂集；
- ✧ **Sequence** (`A`) 所有由 `A` 中元素所组成的序列；
- ✧ **Cycle** (`A`) 所有由 `A` 中元素所组成的有向环；
- ✧ **Subst** (`A, B`) 将 `B` 对象中的原子类替换成 `A` 对象所得的对象

在构造函数 `Set`, `PowerSet`, `Sequence` 和 `Cycle` 中，还可以对集合的基数（cardinality）加以限制。例如，`Set (A, card>=1)` 表示所有由 `A` 中元素组成的非空集合；`Sequence (A, card<=3)` 表示所有由 3 个或 3 个以下 `A` 中的元素组成的序列；`Cycle (A, card=5)` 表示所有由任意 5 个 `A` 中的元素组成的有向环。由于有可能构造出具有无穷多个大小为 0 的对象（例如用除了 `PowerSet` 外的构造函数，构造基数无上限的对象），这 4 个构造函数都不支持用 `Epsilon` 作为构造参数。

在 `Subst` 中，`A` 与 `B` 都不可以为 0 对象 `Epsilon`。

利用构造函数所组成的一些等式的集合，可以描述常见的组合结构。描述的格式为一个有序表：`[A, spec, typ]`，其中，`A` 为类名；`spec` 为描述说明，是一组等式的集合；`typ` 可以为 `labelled` 或者 `unlabelled`，分别表示每个同名类是否具有标记，也就是同名类是否作为不同元素看待。描述说明中的每个等式形式是 `B = rhs`，`B` 是代表该组合类的名称，`rhs` 是由基本类、构造函数和其他类的名称组成的表达式。式子中可以进行递归定义。

这些组合结构描述都比较抽象，这里通过例子来详细说明它们的使用方法。

作为最简单的例子，我们来看二叉树的结构。二叉树的定义是这样的，它可以是一个节点（叶子节点），或者也可以由两棵二叉树分别组成它的两个孩子（子树）。在这里，用 `Union` 表示这里的“或者”关系，用 `Prod` 表示两棵子树的排列关系：

```
[> sys := { B = Union(Z, Prod(B, B)) };
      sys := { B = Union(Z, Prod(B, B)) }
```

建立了结构之后，可以用 `draw` 随机地获得一棵指定大小的二叉树：

```
[> draw( [B, sys], size=6 );
      Prod(Prod(Prod(Z, Prod(Z, Z)), Z), Prod(Z, Z))
```

默认情况下，组合结构被认为是不加标记的（`unlabelled`），所以原子类都被认为是相同的，在一定程度上相当于组合。也可以指定它为加标记的结构，只需加入 `labelled` 标志。由于每个原子类都被认为是不同的，相当于广义的排列。它们的计数结果是不同的：

```
[> count( [B, sys, unlabeled], size=6 );
      42
[> count( [B, sys, labeled], size=6 );
      30240
```

系统预定义的原子类是 `Z`，其他的类名也可以在描述说明中加以指定。例如，我们可以

定义这样的组合结构，它表示一串由三种不同颜色的珠子组成的项链。项链是一个环，所以用 Cycle 表示它的结构，其中每个元素都是三种颜色之一，所以采用 Union 选其一。

```
[> necklace := { N = Cycle( Union(red, blue, green) ),
    red=Atom, blue=Atom, green=Atom }:
> draw( [ N, necklace ], size = 8 );
    Cycle(red, red, blue, blue, blue, blue, green, blue)
> count( [ N, necklace ], size = 8 );
    834
```

和原子类一样，也可以为不同的空类 Epsilon 命名，这常常用在标志组合意义上相同但实际意义不同的结构中。例如，一个简单电路是由串并联子结构混合而成的，一个串联子结构是由两个或两个以上的并联子结构或者电阻串联而成的，而一个并联子结构是由两个或两个以上的串联子结构或者电阻并联而成的。可以这样建立这个结构，分别用 P、S 和 R 表示并联子结构、串联子结构和电阻。

```
[> circuit := { C=Union(P, S, R), P=Set( Union(S, R),
    card>=2 ), S=Set( Union(P, R), card>=2 ), R=Atom }:
```

但是这样的结构中，我们无法判断具体的一个 Set 究竟是并联子结构还是串联子结构：

```
[> draw( [ C, circuit ], size = 6 );
    Set(R, Set(R, R, Set(R, Set(R, R))))
```

要解决这个问题，可以在其中加入一个大小为 0 的空类，并分别命名，加以标记。比如在串联子结构中加入标记 ser，并联子结构中加入 par。

```
[> circuit := { C=Union(P, S, R),
    P=Prod( par, Set( Union(S, R), card>=2) ),
    S=Prod( ser, Set( Union(P, R), card>=2) ),
    R=Atom, par=Epsilon, ser=Epsilon }:
> draw( [ C, circuit ], size = 6 );
    Prod(par, Set(R, R, Prod(ser, Set(Prod(par, Set(R, R)), Prod(par, Set(R, R)))))
```

利用 Subst，我们可以来构造 2-3 树的结构，2-3 树是这样的树，它的所有叶子节点都在同一层上，每一个内部节点都有 2 个或 3 个孩子。用递归的方式来定义它：一棵 2-3 树，或者是一个叶子节点，或者是一棵所有的叶子节点都被替换成具有 2 或 3 个孩子的内部节点的 2-3 树。

```
[> t23 := { T = Union( Z, Subst( Union( Prod(Z, Z),
    Prod(Z, Z, Z) ), T) ) }:
> draw( [ T, t23 ], size = 6 );
    Prod(Prod(Z, Z), Prod(Z, Z), Prod(Z, Z))
```

Maple 的 draw 函数输出的都是用构造函数形式描述的结构，我们可以针对不同的实际情况用代换的方法改变结构的输出形式。例如用递归形式定义的串 C，它是由具有 (aCb) 形式的子串组成的。

```
[> sys := { C=Sequence( Prod(a, C, b)), a=Atom, b=Atom }:
> draw( [ C, sys ], size = 8 );
    Sequence(Prod(a, Sequence(Prod(a, Sequence(Prod(a, E, b), Prod(a, E, b)), b)), b))
```

利用代换函数 subs，可以将这个输出形式改变成所需要的形式：

```
[> eval( subs(Prod={ {}->args }, Sequence={ {}->args },
             Epsilon=NULL, % ) );
          a,a,a,b,a,b,b,b
```

作为组合结构的应用，这里用它来解决一个化学中的同分异构体的实际问题。

例 11.4 求 n 元醇 $C_n H_{2n+1} OH$ 的异构体总数

由有机化学我们知道，烷基 $C_n H_{2n+1}$ 具有三叉树的结构，它由一个碳原子连接着三个氢原子或者烷基构成：

```
[> molecule := { alkyl=Union(H, Prod(C, Set(alkyl,
             card=3))), H=Atom, C=Atom }:
```

利用这个组合结构，就不难求得它的计数了。例如要求己醇的异构体数：

```
[> count( [alkyl, molecule], size=6+2*6+1 );
          17
```

为了便于使用，这里将常用的组合结构列成表 11.2 和表 11.3。

表 12.2 常见的组合结构（有标记）

表达式	对应的组合结构
$A = \text{Prod}(Z, \text{Set}(A))$	非平面树
$B = \text{Union}(Z, \text{Prod}(B, B))$	平面二叉树
$C = \text{Prod}(Z, \text{Sequence}(C))$	广义平面树
$D = \text{Set}(\text{Cycle}(Z))$	全排列
$F = \text{Set}(\text{Set}(Z, \text{card} \geq 1))$	集合的分划
$G = \text{Union}(Z, \text{Prod}(Z, \text{Set}(G, \text{card} = 3)))$	非平面三叉树
$H = \text{Union}(Z, \text{Set}(H, \text{card} \geq 2))$	分层结构
$L = \text{Set}(\text{Set}(\text{Set}(Z, \text{card} \geq 1), \text{card} \geq 1))$	3-平衡层次结构
$M = \text{Sequence}(\text{Set}(Z, \text{card} \geq 1))$	满射
$N = \text{Set}(\text{Cycle}(A)), A = \text{Prod}(Z, \text{Set}(A))$	流程图

表 12.3 常见的组合结构（无标记）

表达式	对应的组合结构
$A = \text{Set}(\text{Sequence}(Z, \text{card} \geq 1))$	整数分划
$B = \text{Sequence}(\text{Union}(Z, Z))$	二叉树
$C = \text{Cycle}(\text{Set}(Z, \text{card} \geq 1))$	项链
$D = \text{Prod}(Z, \text{Set}(D))$	无标记根树
$F = \text{Union}(Z, \text{Set}(F, \text{card} = 2))$	非平面二叉树
$G = \text{Union}(Z, \text{Set}(G, \text{card} = 3))$	非平面三叉树
$H = \text{Union}(Z, \text{Set}(H, \text{card} \geq 2))$	无标注层次结构
$J = \text{Set}(\text{Cycle}(D)), D = \text{Prod}(Z, \text{Set}(D))$	随机映射模式
$K = \text{Union}(Z, \text{Subst}(\text{Union}(\text{Prod}(Z, Z), \text{Prod}(Z, Z, Z)), K))$	2-3 树
$L = \text{PowerSet}(\text{Sequence}(Z, \text{card} \geq 1))$	无重复整数分划

12.4.4 生成函数

comstruct 工具包中还有几个函数可以用于求组合结构对象的生成函数 (generating function)。对象 A 的生成函数是这样的函数项级数:

$$A(z) = \sum_{n=0}^{\infty} a_n z^n$$

其中的系数 a_n 是 A 当其大小为 n 时的计数。

例如上一小节中的三色项链的例子:

```
[> with(comstruct):
> necklace := { N=Cycle(bead), bead=Union(red,blue,green),
    red=Atom, blue=Atom, green=Atom}:
```

可以用函数 gfeqns 获得生成函数的函数方程组, 但它不对方程进行求解。

```
[> gfeqns(necklace, unlabelled, z);
[
bead(z) = red(z) + blue(z) + green(z),
N(z) = \sum_{j_1=1}^{\infty} \frac{\text{numtheory}_{\phi}(j_1) \ln\left(\frac{1}{1 - \text{bead}(z)^{j_1}}\right)}{j_1}, \text{green}(z) = z, \text{blue}(z) = z, \text{red}(z) = z
]
```

利用函数 gfsolve 可以求出生成函数的显式表达式:

```
[> gfsolve(necklace, unlabelled, z);
[
N(z) = \sum_{j_1=1}^{\infty} \frac{\text{numtheory}_{\phi}(j_1) \ln\left(-\frac{1}{-1 + 3z^{j_1}}\right)}{j_1}, \text{bead}(z) = 3z, \text{green}(z) = z,
blue(z) = z, \text{red}(z) = z
]
```

但是, 更多的情况下, 生成函数的显式表达式是无法求得的。这时, gfsolve 将返回 FAIL。

例如考虑非平面二叉树的例子:

```
[> sys := {G = Union(Z, Set(G, card=2))}:
> gfsolve(sys, unlabelled, x);
FAIL
```

但是, 函数 gfeqns 仍将有效。而且, 我们还可以用函数 gfseries 求得生成函数的截断幂级数形式。(幂级数的截断阶数由系统变量 Order 控制。)


```
> gfseries(sys, unlabelled, x);
table([
  G(x)=x+x2+x3+2x4+3x5+O(x6)
  Z(x)=x
])
```

如果引入对空类 Epsilon 的计数，可以定义二元生成函数：

$$A(u, v) = \sum_{n=0}^{\infty} \sum_{m=0}^{\infty} a_{nm} u^n v^m$$

其中 a_{nm} 是大小为 n ，并含有 m 个空类的对象 A 的计数。

作为例子，我们来看一棵以原子类为中间节点，空类为叶子节点的广义树：

```
> tree1 := { T=Union(L, Prod(N, Set(T))), L=Prod(leaf, Atom),
  leaf=Epsilon, N=Atom };
```

同样的，可以调用 gfeqns 求树的二元生成函数所满足的方程；调用 gfsolve 求解生成函数；或者用 gfseries 获得生成函数的截断级数形式。

```
> gfeqns( tree1, unlabelled, z, [[u, leaf]] );
[
  leaf(u, z) = u, L(u, z) = leaf(u, z) z, N(u, z) = z,
  T(u, z) = L(u, z) + N(u, z) e
  ( \sum_{j_1=1}^{\infty} \frac{T(u, z)^{j_1}}{j_1} ) ]
```

12.4.5 combinat 工具包

Maple 的 combinat 工具包是另一个组合数学的研究工具，它是用另一套方法对组合数学问题进行处理，与 combstruct 相互之间没有关联。

```
> with(combinat):
Warning, new definition for Chi
```

最简单的，这个工具包中还是有排列组合的基本函数。求排列，在 combinat 工具包中用的是函数 permute，它的参数可以是一组名称组成的集合或有序表（用于多重集），也可以是一个非负整数 n ，表示从 $\{1, 2, \dots, n\}$ 中选取元素排列。第二个参数是一个整数，表示选取元素的个数，省略时默认为全排列。函数 permute 返回的是一个有序表，有序表中的每一个元素表示一种排列方式。

```
> permute({a, b, c}, 2);
[[a, b], [a, c], [b, a], [b, c], [c, a], [c, b]]
> permute([a, a, b], 2);
[[a, a], [a, b], [b, a]]
> permute(3);
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

对应的，求一组元素的组合，可以调用函数 `choose`。与 `permute` 不同的是，在不给定选取元素的个数时，它将返回所有的组合。

```
[> choose({a, b, c}, 2);
      {(a,c), (a,b), (b,c)}
[> choose([a, a, b], 2);
      [[a,a], [a,b]]
[> choose(3);
      {( ), {1}, {1,2}, {1,3}, {3}, {1,2,3}, {2,3}, {2}}
```

上面的两个函数都是求排列或组合的形式，如果要求排列数或组合数，可以调用函数 `numbperm` 或 `numbcomb`：

```
[> numbperm([a, a, b], 2);
      3
[> numbperm(3);
      6
[> numbcomb({a, b, c}, 2);
      3
[> numbcomb(3);
      8
```

同样，在这个工具包中也有处理整数分划问题的函数——`composition` 和 `numbcomp`。例如，要将 5 表示成两个正整数的和：

```
[> composition(5, 2);
      {[1,4], [4,1], [2,3], [3,2]}
[> numbcomp(5, 2);
      4
```

如果不考虑整数分划之间的先后次序，可以调用函数 `partition`，它将返回整数的所有分划。对应的，它的计数函数是 `numbpart`。

```
[> partition(3);
      [[1,1,1], [1,2], [3]]
```

还有一组函数，可以使我们依次得到一个整数的每一个分划。它的出现顺序是按照通常的编码顺序的。首先，第一个分划总是全 1 的分划，我们用函数 `firstpart` 获得这样的第一个分划。然后，以此以前一个分划为参数，调用 `nextpart` 或 `prevpart`，可以获得编码顺序排列的下一个或前一个分划。也可以用 `lastpart` 获得最后一个分划，也就是整数本身；用函数 `conjpart` 获得共轭的分划。

```
[> firstpart(5);
      [1,1,1,1,1]
[> nextpart(%);
      [1,1,1,2]
[> lastpart(5);
      [5]
[> conjpart([1,2,2]);
      [2,3]
```

利用函数 `decodepart`，还可以获得指定位置的分划；相应的，还可以用 `encodepart` 获得

一个特定分划的序列号。

```
[> decodepart(5, 1);
                                     [1, 1, 1, 1, 1]
[> encodepart(%);
                                     1
```

12.4.6 Stirling 数和拉丁方

将多项式 $x(x-1)(x-2)\cdots(x-n+1)$ 展开成和式 $\sum_{m=1}^n S(n, m)x^m$ ，称系数 $S(n, m)$

为第一类 Stirling 数。

对于第二类 Stirling 数，可以这样定义：将 n 个不同的球放到 m 个相同的盒子里，并保证没有空盒，方案数就是第二类 Stirling 数 $S(n, m)$ 。

在 Maple 中，可以用 `combinat` 工具包中的函数 `stirling1` 和 `stirling2` 分别求两类 Stirling 数。其中第二类 Stirling 数的计算采用的是我们熟知的公式：

$$S(n, m) = \frac{1}{m!} \sum_{k=0}^m (-1)^{m-k} \binom{m}{k} k^n$$

```
[> with(combinat):
Warning, new definition for Chi
[> stirling1(10, 5);
                                     -269325
[> stirling2(10, 5);
                                     42525
```

拉丁方 (Latin square) 是组合数学中经常讨论的区组设计问题。它是由 $0, 1, \dots, n-1$ 组成的 $n \times n$ 的数字方阵，每一行、每一列中每个数字都只出现一次，称 n 为拉丁方的阶数。如果两个同阶的拉丁方 $[a_{ij}]$, $[b_{ij}]$ ，每个位置上的元素组成的有序对 $\langle a_{ij}, b_{ij} \rangle$ 各不相同，则称它们是正交的。Maple 中的库函数 `MOLS`，可以求得一组相互正交的拉丁方。它的调用格式是：

MOLS (p, m, n)

它的第一个参数 p 必须是一个素数，它可以返回 n 个阶数为 p^m 的相互正交的拉丁方。

例如，我们要生成 3 个相互正交的 4 阶拉丁方：

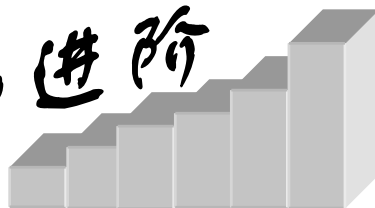
```
[> readlib(MOLS):
[> MOLS(2, 2, 3);
      [[ [0 1 2 3] [0 1 2 3] [0 1 2 3]
         [1 0 3 2] [2 3 0 1] [3 2 1 0]
         [2 3 0 1] [3 2 1 0] [1 0 3 2]
         [3 2 1 0] [1 0 3 2] [2 3 0 1]]
```

在 `combinat` 工具包中，还有几个与组合数学相关的函数，这里将它们列成表 11.4，供读者参考。

表 12.4 combinat 中其他的函数

函数	功能
<code>bell (n)</code>	求 n 阶 Bell 数
<code>fibonacci (n, x)</code>	求多项式的菲波那契数
<code>graycode (n)</code>	返回 n 比特格雷码序列
<code>multinomial (n, k1, k2, ..., km)</code>	求 n 的一个多项展开系数
<code>powerset (s)</code>	求集合 s 的幂集
<code>randcomb (n, m)</code>	n 中取 m 个元素的一个随机组合
<code>randpart (n)</code>	整数 n 的一个随机分划
<code>randperm (n)</code>	n 的一个随机排列
<code>subsets (L)</code>	迭代生成 L 的所有子集

起步与进阶



第

十

三

章

张量分析

本章将主要围绕 Maple 的张量分析工具包，向大家介绍如何用 Maple 解决数学、力学和理论物理中的张量分析问题。将主要介绍 Maple 的张量分析工具包中常用的张量运算函数。

本章具体包括以下内容：

- 🕒 张量数据类型
- 🕒 张量的输入输出
- 🕒 张量的代数运算
- 🕒 张量的导数
- 🕒 坐标变换

自从爱因斯坦 1915 年发表著名的广义相对论以来，张量分析一直在理论物理领域起着无可替代的重要作用。而张量分析在物理学中的应用，又反过来推动着张量分析本身的发展。近几十年来，张量分析更是被广泛地应用到力学和数学的各个方面。

介于张量分析的广泛应用，Maple 中也加入了张量分析软件包 `tensor`，本章将就这个软件包，介绍应用 Maple 解决张量分析问题或者辅助张量分析研究的方法。

13.1 张量数据类型

13.1.1 张量数据类型及其建立

Maple 中的张量分析软件包 `tensor` 中包含张量运算的各种常用函数，它们所使用的数据类型也是一个专门的类型——`tensor_type`。从一般意义的复合数据类型上讲，`tensor_type` 是一个映射表，它包含有两个域，其一是分量域——`compts`，用来存储张量的各个分量；其二是指标域——`index_char`，用来指明对应的指标是协变指标（`covariant index`）或是逆变指标（`contravariant index`）。

举例来说，一个 n 阶张量，它的分量域必须是一个 n 维数组，而且必须是一个“方”的数组，就是说数组的每一维的分量个数必须相同。实际上，数组每一维的分量个数就是该张量所在空间的维数。

而它的指标域则必须是一个长度为 n 的一维数组，数组中的元素不是 1 就是 -1 。如果第 i 个位置上是 1，就表示张量的第 i 个指标是逆变指标；反之，则为协变指标。例如，一个 4 阶张量的指标域是 $[-1, 1, 1, -1]$ ，这就表示张量的第 2 个和第 3 个指标是逆变指标（写成上指标），而第 1 个和第 4 个指标是协变指标（写成下指标）。

特别地，对于 0 阶张量，也就是标量，指标域是一个空数组 $[]$ ，相应的分量域就是不是一个数组而仅仅是一个代数表达式了。

上面介绍的是张量数据类型的构成，在 Maple 中，建立一个张量对象需要调用 `tensor` 工具包中的函数 `create`，它的第一个参数是表明指标类型的数组，第二个参数就是张量分量的数组。

作为例子，我们用 `create` 函数生成一个理论物理中的 Schwarzschild 协变度量张量。它是一个二阶张量，首先载入 `tensor` 工具包：

```
[> with( tensor );
```

再定义它的分量组成的矩阵。由于它是个对角阵，大部分元素是 0，我们用稀疏的对称阵来简化输入（`array` 的参数 `symmetric`, `sparse`）。

```
[> g_compts := array( symmetric, sparse, 1..4, 1..4):
[> g_compts[1, 1] := 1 - 2*m/r:
[> g_compts[2, 2] := - 1 / g_compts[1,1]:
[> g_compts[3, 3] := - r^2:
[> g_compts[4, 4] := - r^2 * sin(th)^2:
```

接下来，用 `create` 生成该张量，同时指定指标的类型，协变度量张量的分量都是逆变

分量，所以需要输入[-1, -1]。由于 Maple 不自动对矩阵求值，所以需要注意用 eval 或 evalm 使其求值。

```
> g := create( [-1, -1], eval(g_compts) );
g:=table([
  index_char=[-1,-1]
  compts=

$$\begin{bmatrix} 1-2\frac{m}{r} & 0 & 0 & 0 \\ 0 & -\frac{1}{1-2\frac{m}{r}} & 0 & 0 \\ 0 & 0 & -r^2 & 0 \\ 0 & 0 & 0 & -r^2 \sin(\theta)^2 \end{bmatrix}$$

])
```

和其他数据类型一样，tensor_type 也可以用 type 来检验其类型：

```
> type(g, tensor_type);
true
```

0 阶张量，也就是标量，也可以用 create 生成。这时，我们用空数组作为第一个参数，标量表达式作为第二个参数：

```
> create( [ ], 1/r );
table([
  index_char=[ ]
  compts= $\frac{1}{r}$ 
])
```

13.1.2 度量张量的输入

度量张量是最常用的张量了，张量的各种微分运算都需要用到度量张量。由于度量张量也是张量，所以自然也可以用 create 来生成。不过在编写交互式程序中，如果需要用户输入度量张量，这是很不方便的。在 tensor 工具包中还提供了交互式输入度量张量的子程序 entermetric。利用它，只需要给出各个指标的线元系数表达式，就可以得到相应坐标系下的度量张量。

函数 entermetric 由两个参数，第一个用于输出度量张量，第二个用于输出坐标符号。由于都是作为输出参数，两个参数都需要是未被赋值的变量（对于已赋值变量，可以用单引号扩起来）。

作为例子，我们用它来输入柱坐标的度量张量：

```
> entermetric ( g, coord );
When entering values, end each line with ";".
```

系统出现提示，等待用户输入空间的维数：

```
[ Input the dimension (integer greater than 1): 3;
  Enter the coordinates:
```

接下来输入的是各个坐标的变量符号（因为要用在输入线元系数表达式中）。

```
[ x1= r;
  x2= theta;
  x3= z;
```

然后系统提示用户，度量张量的矩阵是否为对角阵。由于柱坐标是正交坐标系，所以度量张量的矩阵是对角阵。

```
[ Enter (1) if the metric is diagonal, (2) if otherwise: 1;
  Enter the coefficients of the line element:
```

最后，需要输入的是线元的系数：

```
[ d(r)d(r): 1;
  d(theta)d(theta): r^2;
  d(z)d(z): 1;

                                The coordinates variables are :
                                x1 = r
                                x2 = θ
                                x3 = z
                                The Covariant Metric
                                non-zero components :
                                cov_g11 = 1
                                cov_g22 = r^2
                                cov_g33 = 1
```

至此，柱坐标度量张量的协变分量就输入完成了。同时，我们还得到了坐标符号。

```
> coord;
                                [r, θ, z]
> eval(g);
table([
  compts =  $\begin{bmatrix} 1 & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ 
  index_char = [-1, -1]
])
```


13.2 张量的代数运算

13.2.1 张量的比较

我们称两个张量相等，当且仅当它们在同一个坐标系中的协变（或逆变、混变）分量一一相等。在 Maple 中，可以利用 `tensor` 工具包中的函数 `compare` 比较两个张量。

不过，由于张量在 Maple 中是以分量的形式存放的，所以 `compare` 不能比较两个不同类型的张量（比如同一张量的混变分量和逆变分量，将视为不同）。对于两个张量对象，Maple 首先比较它们的阶数，维数，和指标类型，如果都相同，再对各个分量进行比较。如果各个分量都相同，将返回 `Identical`；否则，则返回不同的分量的指标；如果张量分量的矩阵采用的是不同的指标函数，则返回 `indexfunc`。

函数 `compare` 在比较时会调用张量化简函数 `simp`，所以，对于特殊的张量，最好重载函数 `simp`。对于化简后相同的张量，也将返回 `Identical`。我们通过例子来说明它的使用。

首先比较一个分量由单位矩阵组成的三维空间的二阶张量和他本身：

```
[> with(tensor):
> I3 := array(identity,1..3,1..3):
> U := create( [1,1], op(I3) );
U:=table([
  compts=

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  index_char=[1,1]
])
> compare( U, U );
Identical
```

不出我们的意料，返回的是相等——`Identical`。然后，再将它和一个四维空间的二阶张量比较：

```
[> I4 := array(identity,1..4,1..4):
> V := create( [1,1], op(I4) );
V:=table([
  compts=

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  index_char=[1,1]
])
> compare( U, V );
Dimensions unmatched
```

返回的是维数不匹配——`Dimesions unmatched`。

13.2.2 升降指标

在前一小节中，我们已经知道，不同指标类型的张量是不能比较的。但是，利用 `tensor` 工具包中的函数 `raise` 和 `lower`，可以分别完成张量分量的升、降指标。它们的调用方法是这样的：

raise (contravariant_metric_tensor, A, i1, i2, ...)

lower (covariant_metric_tensor, A, i1, i2, ...)

其中，`contravariant_metric_tensor` 是逆变分量形式的度量张量，而 `covariant_metric_tensor` 是协变分量形式的度量张量；`A` 是有待升/降指标的张量；`i1, i2, …` 是所需升/降的指标，在 `raise` 中，它们对应的 `A` 中的指标必须是协变指标，而 `lower` 中则必须是逆变指标。

例如，对于下面的混变分量张量 `T`：

```
> T := create( [1, -1], array( [ [w, x, 0], [y, z, 0],
    [0, y^2, x*y*w] ] ) );
T:=table([
  compts=
$$\begin{bmatrix} w & x & 0 \\ y & z & 0 \\ 0 & y^2 & xyw \end{bmatrix}$$

  index_char=[1, -1]
])
```

我们利用前一节中输入的柱坐标下的协变形式的度量张量 `g` 对其降指标：

```
> lower( g, T, 1 );
table([
  compts=
$$\begin{bmatrix} w & x & 0 \\ r^2 y & r^2 z & 0 \\ 0 & y^2 & xyw \end{bmatrix}$$

  index_char=[-1, -1]
])
```

这样，就得到了该张量的协变分量形式。但是，如果需要升指标，就必须用到逆变形式的度量张量。对于二阶张量，可以用求逆的方式从度量张量的协变分量求得其逆变分量，`tensor` 工具包中的 `invert` 函数，可以完成二阶张量的逆的运算：

```
> invert( g, detG );
table([
  compts=
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{r^2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  index_char=[1, 1]
])
```

函数 `invert` 在求得二阶张量的逆的同时，还求得了张量的行列式值，通过第二个参数返

回:

```
> detG;
```

$$r^2$$

利用逆变形式的度量张量，就可以进行升指标的运算了：

```
> raise( %, T, 2 );
table([
  compis = 
$$\begin{bmatrix} w & \frac{x}{r^2} & 0 \\ y & \frac{z}{r^2} & 0 \\ 0 & \frac{y^2}{r^2} & x y w \end{bmatrix}$$

  index_char = [1, 1]
])
```

13.2.3 张量的线性组合

我们把同阶张量的数乘和加减运算称为线性组合 (**linear combination**)，利用 `tensor` 工具包中的函数 `lin_com` 可以完成这一类运算。和张量的比较一样，由于 Maple 中用分量方式存储张量，所以，参加运算的每一个张量必须事先化成相同的指标类型。

该函数的调用形式是这样的：

lin_com (c1, T1, c2, T2, ..., cN, TN)

其中，c1, c2, ……是代数表达式；T1, T2, ……是张量对象，它代表这样的运算：

$$c_1 \mathbf{T}_1 + c_2 \mathbf{T}_2 + \cdots + c_N \mathbf{T}_N$$

这个函数的参数数量不限，可以进行任意多个张量的线性组合。函数中的代数表达式参数 c1, c2, ……可以省略，默认情况下，认为它们是 1。

作为例子，我们用它来进行下面几个张量的运算。为了输入的简洁，我们直接用 `table` 构筑张量数据对象，而不采用 `create` 函数生成：

```
> A := array(1..3, 1..3, [[a,0,0], [0,a,0], [0,0,a]]):
  B := array(1..3, 1..3, [[0,b,0], [0,0,0], [0,b,0]]):
  C := array(1..3, 1..3, [[0,0,c], [0,0,0], [c,0,0]]):
> T1 := table([ 'index_char'=[-1,-1], 'compts'=op(A) ]):
  T2 := table([ 'index_char'=[-1,-1], 'compts'=op(B) ]):
  T3 := table([ 'index_char'=[-1,-1], 'compts'=op(C) ]):
```

下面用 `lin_com` 函数求这三个张量的线性组合：

$$x\mathbf{T}_1 + \mathbf{T}_2 + z\mathbf{T}_3$$

由于第二个张量的系数是 1，所以将其省略。

```
> lin_com( x, T1, T2, z, T3 );
table([
  compts = 
$$\begin{bmatrix} x a & b & z c \\ 0 & x a & 0 \\ z c & b & x a \end{bmatrix}$$

  index_char = [-1, -1]
])
```

13.2.4 张量的内积、外积和缩并

在 Maple 的张量分析工具包中，张量的内、外积（inner and outer **product**）都可以用一个函数——**prod** 来完成。它的调用格式是这样的：

prod (A, B, [a1, b1], [a2, b2], ...)

其中，A、B 是有待作内/外积的张量对象；[a_i, b_i] 表示相乘后需要缩并的指标对，a_i 是 A 的指标，b_i 是 B 的指标。而且，对应的指标必须是不同类型的才可以缩并。

作为例子，我们求两个一阶张量，也就是矢量的内积：

```
> u := create( [1], array( [1, m, n] ) );
u := table([
  compts = [l, m, n]
  index_char = [1]
])
> v := create( [-1], array( [a, b, c] ) );
v := table([
  compts = [a, b, c]
  index_char = [-1]
])
```

得到的结果是一个 0 阶张量，也就是标量，它的指标域是空数组：

```
> prod( u, v, [1, 1] );
table([
  compts = l a + m b + n c
  index_char = [ ]
])
```

在张量的阶数范围内，缩并的指标对可以有任意多，也可以没有，没有时就表示两个张量的外积（并乘）。

```
> W := prod( u, v );
W := table([
  compts = 
$$\begin{bmatrix} l a & l b & l c \\ m a & m b & m c \\ n a & n b & n c \end{bmatrix}$$

  index_char = [1, -1]
])
```

对于单个张量，也可以进行缩并运算，所用的函数是 `contract`。它的调用格式和 `prod` 基本相同，区别在于只有一个张量对象。

例如，我们也可以通过对于上面两个矢量的外积再进行缩并的手段得到它们的内积：

```
> contract( W, [1, 2] );
table([
  compts = l a + m b + n c
  index_char = [ ]
])
```

13.2.5 张量的转置

如果保持基矢量的顺序不变，调换张量的指标顺序，得到的新张量，称为原张量的转置张量。在 Maple 中，通过 `tensor` 工具包中的函数 `permute_indices` 可以求得张量的转置张量。该函数的调用格式如下：

permute_indices (T, permutation)

这里，`T` 是原张量；`permutation` 是一个有序表，有序表的长度等于 `T` 的阶数，它表示转置的方式，第 i 个元素的值表示转置后张量的第 i 个指标在原来张量中的位置。例如对于一个三阶张量，如果调用该函数是第二个参数为 `[3, 1, 2]`，则表示原张量的第 3 个指标转置后成为新张量的第一个指标，第 1 个指标转置后称为第 2 个，原来的第 2 个则成为第 3 个。

我们通过一个简单的例子来说明它的用法。为了使转置过程更清楚，我们将分量的位置在分量中予以体现：

```
> A_compts := matrix(3, 3):
> for i to 3 do
  for j to 3 do
    A_compts[i,j] := a.i.j
  od
od:
> A := create( [1,-1], eval(A_compts) );
A := table([
  compts =  $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ 
  index_char = [1, -1]
])
> permute_indices( A, [2, 1] );
table([
  compts =  $\begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$ 
  index_char = [-1, 1]
])
```

需要注意的是，张量的转置并不仅仅是分量矩阵的转置，同时，它的指标类型也相应的

改变了。

13.2.6 张量的对称化和反对称化

利用 `tensor` 工具包中的函数 `symmetrize` 和 `antisymmetrize` 可以分别对张量在指定的指标上进行对称化和反对称化。它们有以下的调用格式：

`symmetrize (T, [i1, i2, ...])`

`antisymmetrize (T, [i1, i2, ...])`

其中， T 是原张量， $[i1, i2, \dots]$ 是进行对称化/反对称化的指标。需要注意的是，对称化和反对称化只能对具有相同类型的指标进行，不同类型指标的分量对称化，是没有意义的，在 Maple 中也是不允许的。

举例来说，对于三阶张量 T ，如果给定的是混变指标形式 $[-1, 1, -1]$ ，那么调用 `symmetrize (T, [1, 3])` 就是对其分量进行如下的运算：

$$T_{i \bullet k}^{\bullet j} = \frac{1}{2} (T_{i \bullet k}^{\bullet j} + T_{k \bullet i}^{\bullet j})$$

同样，反对称化 `antisymmetrize (T, [1, 3])` 表示这样的运算：

$$T_{i \bullet k}^{\bullet j} = \frac{1}{2} (T_{i \bullet k}^{\bullet j} - T_{k \bullet i}^{\bullet j})$$

我们用一个对称二阶张量的例子来说明它们的用法：

```
> g_compts := array( symmetric, 1..3, 1..3):
  for i to 3 do
    for j from i to 3 do
      g_compts[i,j] := g.i.j:
    od:
  od:
g:=create([-1,-1],eval(g_compts));
g:=table([
  compts=
    [g11  g12  g13]
    [g12  g22  g23]
    [g13  g23  g33]
  index_char = [-1, -1]
])
```

由于它本身已经是对称张量，所以反对称化的结果将是 **0** 张量：

```
> antisymmetrize(g, [1,2]);
table([
  compts=
    [ 0  0  0]
    [ 0  0  0]
    [ 0  0  0]
  index_char = [-1, -1]
])
```

13.3 张量场函数的导数

13.3.1 张量分量对坐标的偏导数

张量分量对坐标的偏导数，也就是一般导数，可以用 `tensor` 工具包中的 `partial_diff` 函数求得。函数的调用格式如下：

partial_diff (U, coord)

其中，U 是有待求导的张量，coord 是坐标符号组成的有序表。它返回的是一个比 U 高一阶的张量数据类型，有这样的指标类型：[U[index_char], -1]。

但是，我们知道，在一般的曲线坐标系中，张量分量的偏导数并不是张量分量。由于 Maple 中的张量数据类型是以分量形式存储的，所以在表示上也可以表示非张量的分量，又如在下一节将要介绍的 Christoffel 符号。不过，在使用时需要将它们加以区别，不可以对它们在进行一些必须是张量分量才可以进行的运算。

这里通过一个简单的例子来说明它的用法，我们用它来求一个矢量分量对坐标的偏导数。首先生成一个 1 阶张量：

```
[> with(tensor):
> A := array(1..3, [f(r), g(theta), h(phi)]):
  U := create( [1], op(A) );
U:=table([
  compts=[f(r),g(θ),h(φ)]
  index_char=[1]
])
```

然后，定义坐标符号，再对 U 求偏导数：

```
[> coord := [ r, theta, phi ]:
> part_U := partial_diff( U, coord );
part_U:=table([
  compts=
$$\begin{bmatrix} \frac{\partial}{\partial r}f(r) & 0 & 0 \\ 0 & \frac{\partial}{\partial \theta}g(\theta) & 0 \\ 0 & 0 & \frac{\partial}{\partial \phi}h(\phi) \end{bmatrix}$$

  index_char=[1, -1]
])
```

对于度量张量的偏导数，可以调用函数 `d1metric` 和 `d2metric` 直接求得它对坐标的一阶和二阶偏导数。这两个函数的调用格式是这样的：

d1metric (g, coord)

d2metric (D1g, coord)

其中，g 是协变分量表示的度量张量，也就是具有指标类型[-1, -1]的度量张量；D1g 是

度量张量的一阶偏导，具有 $[-1, -1, -1]$ 的指标类型，可以用 `d1metric` 所得的结果作为这个参数；`coord` 是坐标符号组成的有序表。还需要注意的是，`g` 的分量矩阵必须具有对称的指标函数（生成时用 `symmetric` 参数，或者用 `entermetric` 生成）。

作为例子，我们用 `d1metric` 来求球坐标中度量张量的一阶偏导。首先定义球坐标中的度量张量：

```
[> g_compts := array(symmetric, sparse, 1..3, 1..3):
[> g_compts[1, 1] := 1:
[> g_compts[2, 2] := r^2:
[> g_compts[3, 3] := r^2*sin(theta)^2:
[> g := create( [-1, -1], eval(g_compts) );
g:=table([
  compts=
    [1  0  0
     0  r^2  0
     0  0  r^2 sin(theta)^2]
  index_char=[-1, -1]
])
```

然后调用 `d1metric` 求它的一阶偏导，并给定坐标符号：

```
[> D1g := d1metric( g, coord );
D1g:=table([
  compts= array(cfl, 1..3, 1..3, 1..3, [
    (1, 1, 1)=0
    .....
    (2, 1, 3)=0
    (2, 2, 1)=2 r
    (2, 2, 2)=0
    .....
    (3, 2, 3)=0
    (3, 3, 1)=2 r sin(theta)^2
    (3, 3, 2)=2 r^2 sin(theta) cos(theta)
    (3, 3, 3)=0
  ])
  index_char=[-1, -1, -1]
])
```

13.3.2 Christoffel 符号

在张量分析中，Christoffel 符号是如下定义的。把协变及时量对坐标的偏导数分别对逆变和协变基矢量分解：

$$\frac{\partial \mathbf{g}_j}{\partial x_i} = \Gamma_{ij,k} \mathbf{g}^k, \quad \frac{\partial \mathbf{g}_j}{\partial x_i} = \Gamma_{ij}^k \mathbf{g}_k$$

其中的系数 $\Gamma_{ij,k}$ 成为第一类 Christoffel 符号, Γ_{ij}^k 称为第二类 Christoffel 符号。我们知道 Christoffel 符号可以通过度量张量的导数求得。在 Maple 的 tensor 工具包中的函数 Christoffel1 和 Christoffel2 可以分别求得这两类 Christoffel 符号:

Christoffel1 (D1g)

Christoffel2 (ginv, Cf1)

其中, D1g 是度量张量对坐标的一阶偏导, 可以用函数 d1metric 求得 (参见 12.3.1); ginv 是逆变分量表示的度量张量; Cf1 是第一类 Christoffel 符号。

两个函数返回的都是三阶张量数据类型, 但实际上它们并不是张量分量。它们的返回值, 我们不妨用 Cf1 和 Cf2 表示, 分别与指标记号下的 Christoffel 符号有以下的对应关系:

$$\text{Cf1}[\text{compts}][i, j, k] = \Gamma_{ij,k}; \quad \text{Cf2}[\text{compts}][i, j, k] = \Gamma_{jk}^i$$

接着前一小节中的例子, 用它们来求球坐标系的两类 Christoffel 符号。首先利用 12.3.1 中的度量张量一阶偏导求第一类 Christoffel 符号:

```
[> Cf1 := Christoffel1( D1g );
```

然后用求逆函数 invert 求出逆变分量表示的度量张量, 并利用它求第二类 Christoffel 符号:

```
[> ginv := invert( g, detG );
[> Cf2 := Christoffel2( ginv, Cf1 );
```

由于 Christoffel 符号往往大部分分量为 0, 直接显示过于繁琐, 不利于察看。可以利用 tensor 中的 displayGR 工具显示它们的非 0 分量:

```
[> displayGR( Christoffel2, Cf2 );
```

The Christoffel Symbols of the Second Kind
non-zero components :
 $\{1,22\} = -r$
 $\{1,33\} = -r \sin(\theta)^2$
 $\{2,12\} = \frac{1}{r}$
 $\{2,33\} = -\sin(\theta) \cos(\theta)$
 $\{3,13\} = \frac{1}{r}$
 $\{3,23\} = \frac{\cos(\theta)}{\sin(\theta)}$

函数 displayGR 是用来显示广义相对论中常用张量信息的工具, 它有这样的调用格式:

displayGR (GR_name, object)

其中 GR_name 是和广义相对论有关的对象名称, 参见表 12.1; object 是需要显示的对象。

表 13.1 displayGR 所能显示的对象

名称	对象
coordinates	坐标符号
cov_metric	协变分量形式的度量张量

contra_metric	逆变分量形式的度量张量
detmetric	度量张量的行列式
Christoffel1	第一类 Christoffel 符号
Christoffel2	第二类 Christoffel 符号
Riemann	Riemann 张量 (曲率张量)
Ricci	Ricci 张量 (缩并的曲率张量)
Ricciscalar	Ricci 标量
Einstein	Einstein 张量
Weyl	Weyl 张量

13.3.3 张量分量对坐标的协变导数

张量场函数对坐标的导数在各基矢量上的分量称为张量分量的协变导数。例如, 三阶张量 \mathbf{T} 对坐标的导数可以对一组混变基矢量分解为:

$$\frac{\partial \mathbf{T}}{\partial x^l} = T_{i..l}^{..jk} \mathbf{g}_i^j \mathbf{g}_j^k \mathbf{g}_k$$

其中的系数 $T_{i..l}^{..jk}$ 就是对应分量的协变导数

$$T_{i..l}^{..jk} = T_{i..l}^{..jk} - T_{i..l}^{..jk} \Gamma_{il}^m + T_{i..l}^{..mk} \Gamma_{ml}^j + T_{i..l}^{..jm} \Gamma_{ml}^k$$

在 Maple 中, 可以用 tensor 工具包中的函数 cov_diff 完成这一复杂的运算。它具有以下的调用格式:

cov_diff (U, coord, Cf2)

其中, U 是一个任意阶的张量, coord 是坐标符号, Cf2 是第二类 Christoffel 符号。它的结果是一个比 U 高一阶的张量, 对应的指标类型是 [U[index_char], -1]。

作为例子, 我们用上一小节中得到的球坐标中的第二类 Christoffel 符号, 求一个矢量 \mathbf{F} 的协变分量对坐标的协变导数。

```
[> F := create([-1], array([f[1], f[2], f[3]]));
> cov_diff( F, coord, Cf2 );
table([
  comps = 
    [ 0      -f2/r      -f3/r
      -f2/r   rf1      -cos(theta)f3/sin(theta)
      -f3/r  -cos(theta)f3/sin(theta)  r sin(theta)^2 f1 + sin(theta) cos(theta) f2 ]
  index_char = [-1, -1]
])
```

13.3.4 标量场的方向导数

所谓方向导数，就是标量场的梯度在给定方向上的投影。利用 `tensor` 工具包中的 `directional_diff` 函数，可以求得标量场的方向导数。它具有以下的调用格式：

directional_diff (f, V, coord)

其中，f 是一个代数表达式，表示有待求导的标量场；V 是逆变分量形式的一阶张量，用来表示方向矢量场；coord 是坐标符号。它的返回值是一个代数表达式。

我们通过一个例子来简单地说明它的使用方法：

```
[> coord := [x, y, z]:
> f:= x^2 / (y + z);
                                     
$$f = \frac{x^2}{y+z}$$

> v:= create( [1], array( [x*y, y*z, z*x] ) );
V:=table([
  compts=[x y, y z, z x]
  index_char=[1]
])
> directional_diff(f, v, coord);
                                     
$$-\frac{x^2(-2y^2-yz+zx)}{(y+z)^2}$$

```

13.3.5 Riemann-Christoffel 张量

从张量分析中，我们知道，欧氏空间和 Riemann 空间的区别在于空间的 Riemann-Christoffel 张量（曲率张量）是否为 0。用 `tensor` 工具包中的函数 `Riemann`，可以求得空间的 Riemann-Christoffel 张量。它的调用格式是这样的：

Riemann (ginv, D2g, Cf1)

其中，ginv 是用逆变分量表示的度量张量；D2g 是度量张量对坐标的二阶偏导数，Cf1 是第一类 Christoffel 符号。

作为例子，我们用 Maple 求球面二维空间中的 Riemann-Christoffel 张量。首先建立球面坐标的度量张量：

```
[> g_compts := array( symmetric, sparse, 1..2, 1..2 ):
> g_compts[1, 1] := R^2:
> g_compts[2, 2] := R^2*sin(theta)^2:
> g := create( [-1, -1], eval(g_compts) );
g:=table([
  compts=
$$\begin{bmatrix} R^2 & 0 \\ 0 & R^2 \sin(\theta)^2 \end{bmatrix}$$

  index_char=[-1, -1]
])
```

然后，求出逆变分量表示的度量张量，度量张量的一阶、二阶导数，第一类 Christoffel

符号, 进而求出 Riemann-Christoffel 张量:

```
[> ginv := invert( g, 'detG' );
[> D1g := d1metric( g, [theta, phi] );
[> D2g := d2metric( D1g, [theta, phi] );
[> Cf1 := Christoffel1( D1g );
[> RMNN := Riemann( ginv, D2g, Cf1 );
```

Riemann-Christoffel 张量也可以用 displayGR 显示其非零分量:

```
[> displayGR( Riemann, RMNN );

The Riemann Tensor
non-zero components:

 $R_{1212} = R^2 \sin(\theta)^2$ 
character: [-1, -1, -1, -1]
```

球面的 Riemann-Christoffel 张量有非 0 分量, 由此可知, 球面空间不是欧氏空间。我们还可以由此求得球面的高斯曲率 K:

```
[> K := RMNN[compts][1,2,1,2]/detG;

 $K = \frac{1}{R^2}$ 
```

高斯曲率非 0, 所以球面不是一个可展曲面——不能由平面弯曲得到。

13.4 坐标变换

采用张量方法研究物理问题的一大优势是, 物理现象的描述和坐标系的选取没有关系。对于复杂的物理问题, 可以先在简单地坐标系中描述, 进而利用张量的坐标变换关系, 推广到任意的坐标系中去。

这一节中将介绍 Maple 的张量工具包中的两个坐标变换工具——Jacobian 和 transform。

13.4.1 坐标变换的 Jacobi 矩阵

利用 tensor 工具包中的函数 Jacobian, 可以求得给定变换的 Jacobi 转换矩阵以及 Jacobi 矩阵的逆。它的调用形式是这样的:

Jacobian (Y, Finv, yJx, xJy)

其中, Y 是新坐标系的坐标符号组成的有序表; Finv 是一组变换关系组成的有序表, 元素个数须与 Y 相等, 表中每个元素是一个等式, 写成“老坐标 = 新坐标的表达式”的形式; yJx, xJy 用于输出 (须为未赋值的变量), 分别是新坐标到老坐标和从老坐标到新坐标的 Jacobi 变换矩阵, 为了便于计算, 它们也被定义为张量数据类型, 分别具有指标类型 [1, -1] 和 [-1, 1] (同样, 它们也不是张量分量)。

作为例子, 我们求一个从笛卡尔坐标系向极坐标系转换的 Jacobi 矩阵。

首先, 指定坐标符号, 和坐标变换关系的表达式:

```
[> Y := [r, theta];
```

```
[> Finv := [ x=r*cos(theta), y=r*sin(theta) ]:
```

然后, 就可以求它们的 Jacobi 变换矩阵了:

```
[> Jacobian( Y, Finv, yJx, xJy );
> eval(yJx);
table([
  compts = 
$$\begin{bmatrix} \frac{\cos(\theta)}{\cos(\theta)^2 + \sin(\theta)^2} & \frac{\sin(\theta)}{\cos(\theta)^2 + \sin(\theta)^2} \\ -\frac{\sin(\theta)}{r(\cos(\theta)^2 + \sin(\theta)^2)} & \frac{\cos(\theta)}{r(\cos(\theta)^2 + \sin(\theta)^2)} \end{bmatrix}$$

  index_char = [1, -1]
])
> eval(xJy);
table([
  compts = 
$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -r \sin(\theta) & r \cos(\theta) \end{bmatrix}$$

  index_char = [-1, 1]
])
```

13.4.2 张量的坐标变换

有了坐标变换的 Jacobi 矩阵, 就可以对张量分量进行坐标变换了。只需调用 `tensor` 包中的函数 `transform` 就可以了, 它的调用格式如下:

transform (Tx, Finv, yJx, xJy)

其中, Tx 是老坐标中的张量; 其余的参数意义和 `Jacobian` 中的相同。不同的是, 这里 `yJx` 和 `xJy` 都是作为输入参数的。

通过一个例子来说明它的使用。这里, 就利用上一小节中得到的笛卡尔系到极坐标系的 Jacobi 矩阵来求度量张量在极坐标下的分量形式。笛卡尔坐标系中, 度量张量的矩阵是一个单位阵:

```
[> g_compts := array(symmetric, sparse, 1..2, 1..2 );
> g_compts[1,1] := 1: g_compts[2,2] := 1:
> g := create( [-1, -1], eval(g_compts) );
g:=table([
  compts = 
$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

  index_char = [-1, -1]
])
> simplify(transform( g, Finv, yJx, xJy ));
table([
  compts = 
$$\begin{bmatrix} 1 & 0 \\ 0 & r^2 \end{bmatrix}$$

  index_char = [-1, -1]
])
```

通过坐标变换,就得到了度量张量在极坐标系中的分量形式(虽然这也许并不是个求度量张量的好方法)。

13.5 张量对象的信息

13.5.1 张量信息的获取

Maple 的 `tensor` 工具包中有一组函数可以获取具有张量数据类型的对象的信息。它们并不是必要的,因为 `tensor_type` 的各个域都可以直接用[]获得。在编写程序时,它们可能会显得很方便。由于它们都比较简单,这里只将它们列在下面,而不加以具体举例说明。

- ✧ `get_compts (A)` 获取张量对象 A 的分量域
- ✧ `get_char (A)` 获取张量对象 A 的指标类型域
- ✧ `get_rank (A)` 获取张量对象的阶数

13.5.2 张量的指标函数

在介绍数组类型和线性代数时,我们已经接触过指标函数了,利用它们,可以生成一些特殊的矩阵。在 `tensor` 工具包中,也含有一些特殊的指标函数,利用它们,可以生成有一定规律的矩阵和数组,进而生成特殊的张量。

这里将它们所对应的特征列成下表:

表 13.2 张量的指标函数

函数名	对应的特征	用途
<code>cf1</code>	三个指标中的前两个对称	用于第一类 Christoffel 符号
<code>cf2</code>	三个指标中的后两个对称	用于第二类 Christoffel 符号
<code>cov_riemann</code>	满足对称关系: $R_{ijkl} = R_{klij}$, $R_{ijkl} = -R_{jikl}$, $R_{ijkl} = -R_{ijlk}$	用于 Riemann-Christoffel 张量和 Weyl 张量的协变分量形式
<code>d2met</code>	四个指标中的前两个、后两个分别对称	用于度量张量协变分量对坐标的二阶偏导数
<code>skew23</code>	三个指标中的后两个偏斜对称	用于 <code>connexF</code> 中的结构系数

下面通过一个简单的例子来说明使用这些指标函数所能达到的效果。首先定义一个以 `cov_riemann` 作为指标函数的四维数组,并对它的一个元素赋值:

```
> R:= array (cov_riemann, 1..4, 1..4, 1..4, 1..4);
      R:= array(cov_riemann, 1..4, 1..4, 1..4, 1..4, [ ])
> R[1,2,3,4]:= cos(theta)/r;
```

$$R_{1,2,3,4} = \frac{\cos(\theta)}{r}$$

可以看到,其他具有相应的对称性的元素也跟着变了:

$\left[\begin{array}{l} > \mathbf{R}[2,1,3,4]; \\ > \mathbf{R}[3,4,1,2]; \\ > \mathbf{R}[3,4,2,1]; \end{array} \right.$	$-\frac{\cos(\theta)}{r}$ $\frac{\cos(\theta)}{r}$ $-\frac{\cos(\theta)}{r}$
--	--

灵活地利用这些指标函数，对于这些具有特殊对称性的张量来说，可以减少很多输入工作。