你也能用电脑计算 π=3.14159… 到千万位

杨自强 杨庆 著

清华大学出版社
http://www.tup.tsinghua.edu.cn

你也能用电脑计算

π=3.14159 … 到千万位

ISBN 7-302-01372-1

9 787302 013723 >

定价：10.00元

你也能用电脑计算

$\pi = 3.14159\cdots$到千万位

杨自强　杨庆　著

清华大学出版社

(京)新登字 158 号

内 容 提 要

圆周率 π 含有无穷多位，本书深入浅出地向具有高中以上文化程度的电脑爱好者介绍古今中外有关 π 的算法，特别是近几年国际上采用的最新、最快方法。本书既介绍算法的数学内容，也介绍其计算机程序实现，并使用 BASIC 语言给出几个完整程序。本书还简要介绍创世界纪录的超亿位计算中所使用的新技术，包括同时使用多台电脑并行计算的方法，这些内容对于专业人士也有参考价值 。书中列出 π 的前 20000 多位数值可满足不同使用需要。

书　名：你也能用电脑计算 π＝3.14159…到千万位

作　者：杨自强 杨庆　著

出版者：清华大学出版社(北京清华大学学研大厦，邮编 100084)

http://www.tup.tsinghua.edu.cn

印刷者：清华大学印刷厂

发行者：新华书店总店北京发行所

开　本：787×1092 1/32　印张：6.75　字数：146 千字

版　次：2001 年 4 月第 1 版　2001 年 4 月第 1 次印刷

书　号：ISBN 7-302-01372-1/TP・526

印　数：0001～4000

定　价：10.00 元

前　言

本书是介绍电脑上计算圆周率 $\pi = 3.14159\cdots$ 到千千万万位的专门著作的通俗版本。只要具有中学以上数学水平的电脑爱好者都可以读懂。

(1) 如果你只想知道 π 的前两万位或更多位数字，请看附录 B 。

(2) 如果你想知道古今中外的算 π 公式，特别是电脑时代的 π 值计算方法，请看本书的前两章。其中第 2 章的最后一节介绍了包括至 1997 年的世界上最新、最快的算法。

(3) 本书既介绍算 π 的数学公式，也讨论和具体给出用 BASIC 语言编制的算 π 程序。程序既包括每个只有十余行，但都可以算出十多位 π 值的简单程序；也包括仅百余行却可以算出几万位的分段算法程序；甚至还包括含有千余行，并可以算出上百万位的相当专业的计算程序。后两者都采用了当今世界上最先进的算法。令人意外的是，基于 1997 年才正式发表的世界最新算法的分段程序，不仅程序短，而且计算公式也简单到初中学生都能看懂，值得细读（见第 4 章）。

(4) 本书为支持千千万万位 π 值计算而建立的任意精度（亦称多精度）运算系统也完全可以用于其他需要超高精度运算的场合。

我们撰写本书的动机是，考虑到目前计算机已经十分普及，在科学与工程计算、企事业管理、办公自动化、网络通信、以及环境模拟、动画制作、文字处理、电子游戏、娱乐活

动等方面都起到了重要的作用，并且在我国，计算机正在逐渐进入家庭，但是目前一般家庭使用计算机的水平不算高，能充分使用并能联网漫游内容丰富的网络世界的人不多。不少为上中学的孩子购买了计算机的家长发现，计算机用得好并不容易，因此它的用处并不像期待的那么高。计算机在许多家庭除了用作辅助教学外，更主要的是作为游戏娱乐的工具。当然，刚买计算机的时候，孩子还学学打字，但后来打字的机会也极少。

在书店里，家长们和老师们很难找到一些适合中学生阅读的计算机应用方面的科普读物，特别是真正能吸引中学生们开发计算机的“计算”功能的读物，为了弥补这方面的不足，我们撰写了这本可供课外（兴趣小组）阅读的小册子。我们希望，对计算机有兴趣的读者，除了熟悉那些令人如痴如醉的电子游戏和神奇而又满天飞舞的 Windows 之外，也通过这本小册子看看计算机应用的另一个方面 —— “计算”能力方面。计算功能是人类发明计算机的初衷，也是计算机早年最重要，甚至几乎是惟一的应用。但是数值计算又与令某些人乏味的数学紧密相联，为了能吸引读者，我们精心地挑选了一个大家都熟悉而又确实是十分有趣的题目：圆周率 π 的计算方法与程序。虽然本书涉及不少十分深入内容，但表述上力求做到深入浅出，对于小数超出中学范围的数学内容给予必要的补充说明。如果读者真能对计算发生兴趣，这就是对我们的最大安慰。

本书首先介绍人类认识和计算圆周率 π 的有趣历史，从古希腊阿基米德的圆内接与外切正多边形的周长近似，中国古代刘徽割圆术，祖冲之的疏率、密率，欧洲文艺复兴、科

学飞跃时期的级数方法……直到当今世界上可算出上亿位 π 值的最先进方法，包括 1997 年才面世的可从任意一位算起的最新方法，书中也回答许多人可能会提出的“算这么多位 π 值有什么用？”的问题。本书既重视算法的数学基础，也重视在计算机上的实际计算方法，特别是介绍在计算机上作超长位数（成千上万位）运算的方法，并建立一个能当此任的“多精度运算系统”。书中在介绍程序设计要点之后，还给出具体的 BASIC 语言程序，它们既可以在 DOS 自带的 QBasic 环境下解释执行，也可以在 Quick BASIC 甚至 Windows 下的 Visual(可视)BASIC 下编译执行。通过阅读本书，有上机操作条件的读者完全可以自己动手算 π 到千千万万位。要想亲自计算，但又不想做过多录入工作的读者，可直接与作者联系获得本书的全部程序。

本书的两作者一老一青，后生毕业于计算机科学系，并专门从事软件研究，老者一生在计算机上从事数值计算研究，还实际参加过中国科学院课题中的《超高精度 π 值的并行计算》研究工作。令他意外的是，虽然这是一项高科技的课题，但关心和打听的人远远多于他一生从事的其他计算研究项目。关心的人从询问课题意义、计算方法、国际水平直至索取 π 的前几千位结果。因此，我们把本书看作是该课题研究成果的科普宣传，从而书中还涉及计算数学中的某些基本知识，介绍能提高超长位数运算速度千万倍的当代最新计算方法，以及多台计算机（或处理器）同时对一个大问题并行计算的初步知识，正是这些综合技术使得目前国际上最大、最快的超级并行计算机可以算出上亿位的 π 值，这些内容对专业人士也不无裨益。

最后，作为国内外第一本专门详细介绍 π 值计算的书，我们还在附录 B 中给出 π 值前两万位的完整结果及其后直至百万位的部分位，以满足各方面的要求。值得一提的是，这些结果是使用本书程序算出的，并与我们过去的计算结果以及国际上发表的结果核对过，最后又直接由计算机输出照相排版，因此是高度可靠的。

作者十分感谢马志强先生，他协助我们在 Windows 98 的 Visual BASIC 环境下调试了本书的算 π 程序，并提了一些建议。魏公毅研究员在审阅本书时提出了很好的改进意见，作者深表谢忱。

作者 于北京 中关村
2000 年 10 月

本书给出的所有程序和前 50 万位 π 值都已记录在一片软盘上，需要的读者可直接与作者联系。

地　址：北京 中关村 中国科学院 软件园区
邮　编：100080
E-mail: yangzq@sun.ihep.ac.cn

目 录

第 1 章　圆周率 π 史话

圆是自然界中普遍存在而且逗人喜爱的图形，它充满着神秘的色彩。大家知道，在具有相同周长的几何图形中，圆的面积最大。圆中最神秘的是圆周率 π 。当我们还是小学生的时候，我们已经学会了计算圆的周长和面积：

$$\text{圆周长：}\quad L = 2\pi R$$

$$\text{圆面积：}\quad S = \pi R^2$$

式中 R 为圆半径。当 $R = 1$ 时，圆面积 $S = \pi$ ，这时，计算圆面积和计算 π 是等价的。在教科书中，圆周率 $\pi = 3.14$ 或 3.1416 ，其实 π 在数学上是个无理数，其位数是无穷无尽的，随着人类认识的进步，准确的位数不断增加。最先是 $\pi = 3$, 稍后是 $3.1, 3.14, \cdots$ 。使用当今国际上最大、最快的超级并行计算机，人们已经可以算出上亿位的 π 值。下面是其中的前 200 位：

3.1415926535 8979323846 2643383279 5028841971 6939937510
5820974944 5923078164 0628620899 8628034825 3421170679
8214808651 3282306647 0938446095 5058223172 5359408128
4811174502 8410270193 8521105559 6446229489 5493038196

人们可能会问：算这么多位 π 值有什么现实意义？问题的回答将留在本书的最后一章，本章的任务主要是介绍人类认识圆和圆周率 π 的某些重要的史料。

1.1 古典方法

人们认为，三千年前人类就有了圆的抽象概念。我国春秋时代的《墨经》已经给出了圆的定义，圆周率 π 是圆的概念的核心。$\pi = 3.1416$ 的纪录最早出现在古希腊阿利亚布哈塔的著作中。在人们还不知道圆周率的时候，古代的数学家是通过圆的内接或外切正多边形的周长和面积来近似计算圆的周长和面积的，古希腊著名科学家阿基米德就是利用这些几何学知识给出 π 值的估计，他通过计算圆的内接与外切正 96 边形的周长得到

$$\frac{6336}{2017\frac{1}{4}} < \pi < \frac{14688}{4675\frac{1}{2}}$$

如果用小数表示，上式就是

$$3.1409\cdots < \pi < 3.1428\cdots$$

1.1.1 祖冲之在圆周率计算中的贡献

我国也是世界上对圆周率计算有重要贡献的国家之一。史料《隋书 · 律历志》上有如下一段话：

> “古之九数，圆周率三，圆径率一，其述疏舛。自刘歆、张衡、刘徽、王蕃、皮延宗之徒各设新率，未臻折衷。宋末，南徐州从事史祖冲之更开密法，以圆径一亿为一丈，圆周盈数三丈一尺四寸一分五厘九毫二秒七忽。朒数三丈一尺四寸一分五厘九毫二秒六忽，正数在盈朒二数之间。密率：圆径一百一十三，圆周三百五十五。约率：圆径七，圆周二十二。”

这段话的大体意思是说，先前的圆周率取值为 3，很是粗糙，于是刘歆、张衡、刘徽、王蕃、皮延宗等人各自给出了新的圆周率，但都不够理想，其后宋末 (指公元五世纪南北朝时的宋) 的祖冲之使用更精密的方法，用九位数运算 (一亿为一丈)，得到的圆周率 π 在如下两数之间

$$3.1415926 < \pi < 3.1415927 \tag{1.1}$$

并且还得到

$$\text{密率：}\frac{355}{113}, \qquad \text{疏率：}\frac{22}{7} \tag{1.2}$$

上式如果用小数表示，就是

$$\text{密率：}\pi = 3.1415929\cdots, \qquad \text{疏率：}\pi = 3.1428\cdots$$

祖冲之给出的圆周率准确到小数点后第 7 位，这个纪录在世界上保持了一千多年，直到 16 世纪，德国人奥托和荷兰人安托尼兹才重新发现密率，所以国内、外多数人称 $\frac{355}{113}$ 为“祖率”，以纪念祖冲之的伟大贡献。

1.1.2 刘徽割圆术

令人婉惜的是，祖冲之的上述重要工作被记载得太简单，没有包括他所使用的方法。后人只能根据当时的情况分析，多数人认为祖冲之的计算方法可能是基于魏晋时期刘徽的割圆术. 刘徽是我国古代著名的数学家，他的最大功绩是为《九章算术》作注 (公元 263 年，祖冲之出生前 166 年)。《九章算术》中指出圆面积的计算方法是“半周半径相乘得积步”(意思是圆的半周长与半径长相乘得圆的面积)。刘徽对此作

注，用割圆术证明此圆面积公式的正确性，并指出“周三径一”的粗糙。祖冲之擅长数学，又是掌管天文历法的官吏，他肯定熟知一百多年前刘徽的工作。基于这样的认识，我们有必要介绍刘徽的割圆术。所谓割圆，就是等分圆周，并依次连接等分点组成圆的内接正多边形。刘徽认为：

> “割之弥细，所失弥少，割之又割，以至于不可割，则与圆合体而无失矣。”

也就是说，圆的内接正多边形的边数越多，其面积与圆面积之差越小。当边数增加到无法增加时 (按现代数学，此应理解为无限多或极限情形。) 则该正多边形的面积便与圆面积没有差别了。

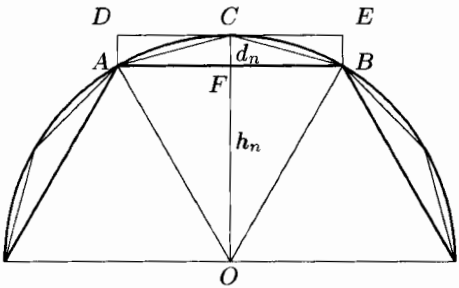

图 1-1 刘徽割圆术示意图

刘徽割圆术的具体方法如图 1-1 所示，这个方法与阿基米德把圆介于圆的内接与外切正多边形的思路不完全相同。记圆的面积为 S, 其内接正 n 边形的面积为 S_n ，图中 AB 是它的一边。当分点加倍之后，面积记为 S_{2n} 。若把图中两个四边形 $\diamond AOBC$ 和 $\square ABED$ 的面积分别记为 $S_{\diamond AOBC}$ 和

$S_{\square ABED}$，则有

$$S > n \times S_{\diamond AOBC} = S_{2n},$$

$$S < n \times \left(S_{\diamond AOBC} + \frac{1}{2} S_{\square ABED}\right) = S_{2n} + (S_{2n} - S_n)$$

从而

$$S_{2n} < S < S_{2n} + (S_{2n} - S_n) \tag{1.3}$$

上式就是刘徽用于算 π 的圆面积不等式。根据刘徽的计算，当半径为 10 寸时

$$96 \text{ 边形面积：} \quad S_{96} = 313\frac{584}{625} \text{寸}^2$$

$$192 \text{ 边形面积：} \quad S_{192} = 314\frac{64}{625} \text{寸}^2$$

由 (1.3) 式得

$$314\frac{64}{625} < 100\pi < 314\frac{169}{625}$$

刘徽作舍入处理后，取

$$\pi = 3.14 \qquad \text{或} \qquad \frac{157}{50}$$

后来他又得到更精确的圆周率

$$\pi = \frac{3927}{1250}, \qquad \text{化为小数就是 } 3.1416$$

所以后人亦称 3.14 与 3.1416 为“徽率”。

现在让我们来体会一下用刘徽割圆术算 π 的实际过程。首先我们需要导出由圆内接正 n 边形递推得到正 $2n$ 边形的公式。在图 1-1 中，令半径为 1，这时圆面积 $S=\pi$，所以算 π 等价于算圆面积 S。记圆内接正 n 边形的面积为 S_n，其一边长为 $a_n=|AB|$。由商高定理得

$$h_n=|OF|=\sqrt{1-\left(\frac{a_n}{2}\right)^2} \tag{1.4}$$

$$d_n=|FC|=1-h_n \tag{1.5}$$

多边形由 n 边变为 $2n$ 边之后，其面积在原来的基础上增加了 $\frac{n}{2}$ 个矩形 $\square ABED$ 的面积，故有

$$S_{2n}=S_n+n\frac{a_n}{2}d_n \tag{1.6}$$

新的边长由商高定理得

$$a_{2n}=\sqrt{\left(\frac{a_n}{2}\right)^2+d_n^2}=\sqrt{2\left(1-\left(\frac{a_n}{2}\right)^2\right)}=\sqrt{2d_n} \tag{1.7}$$

有了递推式，我们就可以从某个圆内接正多边形开始计算。众所周知，从正 6 边形起算是方便的，因为这时的边长和面积为

$$a_6=1, \qquad S_6=\frac{3}{2}\sqrt{3}$$

依次使用式 (1.4)~(1.7) 便可得到正 12 边形的面积 S_{12} 和边长 a_{12}

$$h_6=\sqrt{1-\left(\frac{a_6}{2}\right)^2} \qquad d_6=1-h_6$$

$$S_{12} = S_6 + 6\frac{a_6}{2}d_6$$

$$a_{12} = \sqrt{2d_6}$$

得到 a_{12} 后，再次使用式 (1.4)~(1.7) 便可得到正 24 边形的面积和边长。不断地重复这样的计算过程，又可以得到边数为 48, 96, 192,$\cdots$ 的内接正多边形的面积。

迭代的概念：在计算数学中，类似上述的反复递推过程称为**迭代过程**。从上一步到下一步的式子 (此处是 (1.4)~(1.7)) 称为**迭代式**，计算开始时指定的值 $\left(\text{此处是}\, a_6 = 1,\ S_6 = \frac{3}{2}\sqrt{3}\right)$ 称为**迭代初始值**，或简称**初值**。值得注意的是，迭代方法是本书的重要计算方法之一，今后将频繁地被使用。在上述刘徽割圆术的几个迭代式中，除了 2 个开平方运算之外，尚有 7 个四则运算 (式中多次出现的相同运算，如 $\frac{a_n}{2}$ 等，均只统计一次。) 如果每步都列出上限 $S_{2n} + n\frac{a_n}{2}d_n$，也只增加一个四则运算。姑且把开方也看作是一个运算，那么每次迭代 (割圆) 不超过 10 个运算。

如果祖冲之当年确实是使用刘徽的割圆术来计算 π，那么按《隋书 · 律历志》记载，他使用 9 位数运算。若从圆的内接正六边形开始，为达到他所需的精度，需要割圆 12 次，并正确地得到直至 $6 \times 2^{12} = 24576$ 边形的面积。特别有

$$S_{24576} = 3.14159261\text{丈}^2$$

$$S_{12288} = 3.14159251\text{丈}^2$$

再由刘徽不等式 (1.3) 得 S (即 π) 的估计

$$S_{24576} < S < S_{24576} + (S_{24576} - S_{12288})$$

$$3.14159261 < S < 3.14159271$$

即使在今天，用纸和笔手工地完成上述演算（对 9 位数至少进行 12×9 次四则运算，其中还包括 24 次开方）也是非常艰辛的。何况在祖冲之那个年代，中国尚未发明算盘，而且当今国人在小学生时代就已学会的那些适合纸上作业的竖式手算方法还没有传入中国。后者直到 18 世纪 (祖冲之死后一千二百多年) 才在我国流行开来。祖冲之只靠一把小棒状的“算筹”，要得到那样精确的结果是多么不容易啊！时代在进步，特别是在计算机已经相当普及的今天，使用 BASIC 语言编制程序，在计算机上实现刘徽割圆算法又是十分容易的事。在本书第 4 章 4.1 节中，含有几个能计算十余位 π 值的程序。它们的计算公式虽然各式各样，但程序都很简单，每个的长度都不超过十余行。

1.1.3 古典方法类的特征与几个方法的比较

在结束算 π 的早期历史介绍之前，我们鼓励读者思考如下的问题：阿基米德与刘徽都是通过几何途径来估计 π 的上、下限的。显然你也可以基于“圆面积介于同边数的圆内接与外切正多边形之间”的几何思路来估计 π 的上、下限。当你据此导出从正 n 边形递推到正 $2n$ 边形公式之后，你会发现：你的公式与阿基米德的或刘徽的公式，在迭代的每一步都有相近的运算量。表面上，因刘徽的公式无需计算外切正多边形，似乎应有较少的运算量，但细致地分析一下便知，同边数的圆外切正多边形与圆内接正多边形是相似形，其周长与面积均有一定的比例。算出其中的一个，另一个只需乘上此比例（面积是乘比例的平方）便得。也就是说，这和刘徽计算矩形面

积 $S_{\square ABED}$ 的运算量是差不多的。不过从计算精度来看，却是刘徽的公式最好。这里我们想告诉读者，如果你因自己一时导不出有关的式子而难于理解上面的分析和比较的话，请不要烦恼，因为本书第 4 章 4.1 节讨论 BASIC 程序时，将会出现这些式子。虽然用古典的几何方法来计算 π 十分直观、易懂，但它们的效率都很低。表 1-1 是作者使用 BASIC 程序（采用约 16 位十进制数的“双倍精度”运算）实现刘徽割圆

表 1-1　刘徽割圆术估计 π 的迭代过程

次数	边数	下界	上界
01	12	3.000000000000000	3.401923788646684
02	24	3.105828541230249	3.211657082460499
03	48	3.132628613281238	3.159428685332227
04	96	3.139350203046867	3.146071792812496
05	192	3.141031950890510	3.142713698734152
06	384	3.141452472285462	3.141872993680415
07	768	3.141557607911858	3.141662743538253
08	1536	3.141583892148319	3.141610176384779
09	3072	3.141590463228050	3.141597034307782
10	6144	3.141592105999272	3.141593748770493
11	12288	3.141592516692158	3.141592927385044
12	24576	3.141592619365384	3.141592722038610
13	49152	3.141592645033691	3.141592670701998
14	98304	3.141592651450768	3.141592657867844
15	196608	3.141592653055037	3.141592654659306
⋮	⋮	⋮	⋮
25	6×2^{25}	3.141592653589792	3.141592653589794
26	6×2^{26}	3.141592653589793	3.141592653589793

术估计 π 的迭代过程记录。容易看出随着迭代次数的增加，π 的有效位数也增加。但是后者增加的速度很慢，大约每三步才得两位，正是这样的原因，没有人能用几何方法手算出 40 位以上的 π 值。荷兰人鲁道尔夫为计算 35 位 π 值，几乎把精力拖垮。据史料记载，几何途径的最高纪录是 Von Ceulen 在 1610 年创造的，他运用阿基米德方法算出了 35 位 π 值。

1.2 用级数计算 π 值

如前节所述，用几何途径计算 π 值太慢。为寻求更快的计算公式，世界各国的数学家经历了漫长而艰苦的探索。但直至四百多年前，欧洲文艺复兴，科学技术飞跃，特别是新的数学工具出现之后，π 值的算法研究才有可能获得突破，并使其有效位数得到明显的延伸。

第一个放弃几何途径获得结果的似乎是法国的韦特，他给出不断开方的公式：

$$\frac{2}{\pi}=\sqrt{\frac{1}{2}}\times\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}\times\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}}\cdots$$

但是，近三百年来算 π 的主流是级数方法。主要是根据 π 与三角函数的关系，并把三角函数展开成级数来计算 π。为了更容易理解其后的内容，我们首先回顾三角学中 **角的弧度表示方法**。在初学三角函数时，用度“°”来度量一个角的大小。如直角是 90°，平角是 180°，转一圈是 360° …。但在高等数学中，常使用弧度来表示。1弧度 $=180°/\pi\simeq 57.296°$。这时平角是 π 弧度，转一圈是 2π 弧度。几个特殊角 30°，45°，60°，90°

依次是 $\pi/6$, $\pi/4$, $\pi/3$, $\pi/2$ 弧度。

17 世纪的 60 、 70 年代，英国的牛顿和葛里高利先后得出 $\sin^{-1}x$, $\mathrm{tg}^{-1}x$, $\sin x, \cos x$, $\mathrm{tg}x$, $\sec x$ 和 e^x 等一些超越函数的级数表达式。1671 年，葛里高利又发现了其后人们以他的名字命名的级数

$$\theta = \mathrm{tg}\theta - \frac{1}{3}\mathrm{tg}^3\theta + \frac{1}{5}\mathrm{tg}^5\theta \cdots \tag{1.8}$$

三年后，德国的莱布尼兹 (1674) 令上式的 $\theta = \frac{\pi}{4}$，得到了 π 的表达式

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} \cdots \tag{1.9}$$

开创了级数算 π 的先河。紧接着是牛顿 (1676) 在级数

$$\sin^{-1}x = x + \frac{1}{2} \times \frac{x^3}{3} + \frac{1 \times 3}{2 \times 4} \times \frac{x^5}{5} + \frac{1 \times 3 \times 5}{2 \times 4 \times 6} \times \frac{x^7}{7} + \cdots \tag{1.10}$$

中令 $x = \frac{1}{2}$ 得到

$$\frac{\pi}{6} = \frac{1}{2} + \frac{1}{2 \times 3} \times \frac{1}{2^3} + \frac{1 \times 3}{2 \times 4 \times 5} \times \frac{1}{2^5} + \frac{1 \times 3 \times 5}{2 \times 4 \times 6 \times 7} \times \frac{1}{2^7} + \cdots \tag{1.11}$$

其后，英国麦欣 (Machin) 在 1706 年发现了恒等式

$$\frac{\pi}{4} = 4\mathrm{tg}^{-1}\frac{1}{5} - \mathrm{tg}^{-1}\frac{1}{239} \tag{1.12}$$

并用此式算 π 到小数点后 100 位。这是人类算 π 历史的重大里程碑。值得指出的是，麦欣公式实际上是把 π 的计算转化

为 $\mathrm{tg}^{-1}x$ 的计算，而后者的级数展开式，如前所述，在当时已经获得：

$$\mathrm{tg}^{-1}x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots \tag{1.13}$$

其后，威加 (1794) 算出 137 位。1844 年，有计算神童之称的达斯使用公式

$$\frac{\pi}{4} = \mathrm{tg}^{-1}\frac{1}{2} + \mathrm{tg}^{-1}\frac{1}{5} + \mathrm{tg}^{-1}\frac{1}{8} \tag{1.14}$$

并整整花了两个月，又把 π 算到 200 位。值得一提的是，英国的谢克斯在 1873 年同样是使用麦欣恒等式，却把 π 算到 707 位。虽然，后人发现他的结果只有前面的 527 位才是正确的，但这 527 位的纪录依然是电子计算机出现之前的最高纪录了。

第 2 章　电子计算机时代的 π 值计算

电子计算机的出现，为 π 值计算提供了极为有效的工具，算得的 π 值位数与日俱增。表 2-1 中收集了在电子计算机上计算 π 值的一些重要纪录。

表 2-1　电子计算机时代 π 值计算的若干重要纪录

作者	年份	位数	机器型号	耗时
Reitwiesner	1949	2037	ENIAC	70 小时
Nicholson 等	1954	3089	NORC	13 分
Felton	1958	10000	Pegasus	33 小时
Genuys	1959	16167	IBM 704	4.3 小时
Shanks 等	1962	10 万	IBM 7090	8.7 小时
Gilloud 等	1966	25 万		
Gilloud 等	1967	50 万		
Kanada 等	1982	200 万	NEC	
Kanada 等	1983	1600 万	HITAC S-810	
Gosper 等	1985	1700 万	Symbolics	
Bailey	1988	2936 万	CRAY	
Kanada 等	1988	2.01 亿	HITAC S-820	
Gregory 等	1989	4.80 亿	IBM 3090VF	
Chudnovsky	1989	4.80 亿	CRAY-2	
Kanada 等	1989	5.36 亿	HITAC S-820	
Gregory 等	1989	10.11 亿	IBM 3090VF	
Gregory 等	1991	22.60 亿		
Kanada 等	1995	32.21 亿	HITAC S-3800	
Kanada 等	1995	64 亿	HITAC SR2201	

注：1997 年曾有人分别算出过第 100 亿和 10000 亿位附近不多的几位 π 值，但因为他们没有计算其前头的位（见 2.2.2 节 3.），所以不列入本表.

从上表可以看出，在不到五十年的时间内，算出的 π 值长度，从刚开始时的二千多位到最近的几十亿位，长度增加超过了三百万倍。早期计算机使用的公式，主要是继承历史优秀遗产。随着位数越算越多，人们不得不研究出一些速度更快的新公式，本章着重介绍的正是当今国际上这些最先进的算法。

2.1 继承历史优秀遗产

在电子计算机上进行 π 值计算，原则上可以沿用二、三百年前的级数方法。事实上在电子计算机的早期，人们确实是这样做的。例如，表 2-1 中的前四项工作 (除 Felton 外) 都是使用著名的麦欣公式

$$\pi = 16\mathrm{tg}^{-1}\frac{1}{5} - 4\mathrm{tg}^{-1}\frac{1}{239} \tag{2.1}$$

顺便指出，今天人们计算 $\mathrm{tg}^{-1}x$ 时，至少可以使用如下两个式子中的任一个：

$$\mathrm{tg}^{-1}x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots \tag{2.2}$$

$$\mathrm{tg}^{-1}x = \frac{y}{x}\left(1 + \frac{2}{3}y + \frac{2\times 4}{3\times 5}y^2 + \frac{2\times 4\times 6}{3\times 5\times 7}y^3 + \cdots\right) \tag{2.3}$$

其中

$$y = \frac{x^2}{1+x^2} \tag{2.4}$$

另一个例子是 1962 年算出 10 万位 π 值的辛克斯 (Shanks)

等人所使用的公式

$$\pi = 24\mathrm{tg}^{-1}\frac{1}{8} + 8\mathrm{tg}^{-1}\frac{1}{57} + 4\mathrm{tg}^{-1}\frac{1}{239} \tag{2.5}$$

此式虽然不同于麦欣公式，但也是上个世纪就已经得到的(史多墨， 1896)。这个公式是为了更快地算出 π 值而提出的，类似的公式还有

$$\pi = 48\mathrm{tg}^{-1}\frac{1}{18} + 32\mathrm{tg}^{-1}\frac{1}{57} - 20\mathrm{tg}^{-1}\frac{1}{239} \tag{2.6}$$

$$\pi = 20\mathrm{tg}^{-1}\frac{1}{7} - 8\mathrm{tg}^{-1}\frac{3}{79} \tag{2.7}$$

前者来自大数学家高斯 (1863) 的著作，后者 (包括式 (2.3) 和式 (2.4)) 也曾被 18 世纪著名数学家欧拉用来算 π，他仅花了一个小时便手算出 20 位 π 值。

2.2 研究更快的 π 值计算公式

如果要算的 π 值位数不很多，那么沿用一百多年前就给出的级数公式一般不会有什么问题。但当要求计算的位数很多时 (例如几十万位以上)，只靠计算机硬件的进步，也难在合理的时间内完成作业，因此，我们需要更快的计算公式。本节首先介绍用于衡量公式快慢的“ m 阶收敛”概念 (通常 $m = 1, 2, 4$)。通俗地说，若某公式在下一步计算中，所得结果的有效位数大体是上一步的 m 倍，就称为 m 阶收敛式。前面介绍过的所有算 π 公式，本质上都是 1 阶的。只是到了 20 世纪 70、80 年代，世界上才出现了 2 阶和 4 阶公式，从

而使算 π 工作跃上了一个新台阶，并出现了 π 值计算超亿位的新纪录。

2.2.1 高阶收敛的概念

假如某公式在计算的各步 (指迭代中的步或级数求和中的一项和) 依次得到如下结果

$$u_1, u_2, \cdots, u_k, u_{k+1}, \cdots$$

而且后一步的结果比前一步更加靠近真解 u，那么此计算过程便是“收敛”的，进而若存在正常数 c 和正整数 m，使得下式成立

$$|u_{k+1} - u| \leqslant c|u_k - u|^m \tag{2.8}$$

则称此计算式具有 m 阶收敛速度。1 阶收敛亦称线性收敛 (这时应有 $c < 1$)，2 阶收敛亦称平方收敛。显然，阶数越高，收敛速度越快。例如，对于 2 (或 4) 阶收敛公式，下一步的有效位数是上一步的 2 (或 4) 倍 (当 $c = 1$ 时)。在阶数相同时，c 值越小，收敛越快。

表 2-2 不同阶公式有效位数在迭代过程中的不同变化

公式类型	各步的有效位数 (d 为初值位数)					
阶数和 c 值	1 步	2 步	3 步	4 步	5 步	6 步
1 阶 $c = \frac{1}{10}$	d+1	d+2	d+3	d+4	d+5	d+6
1 阶 $c = \frac{1}{1000}$	d+3	d+6	d+9	d+12	d+15	d+18
2 阶 $c = 1$	$2d$	$4d$	$8d$	$16d$	$32d$	$64d$
4 阶 $c = 1$	$4d$	$16d$	$64d$	$256d$	$1024d$	$4096d$

通过表 2-2 和表 2-3 可以形象地看出，不同阶公式对应

的有效位数在计算过程中的不同变化，以及为达到相同有效位数而需要的不同计算次数。为便于比较，表中的初值全都假设有 d 位有效数字。需要指出的是，从计算的总体代价来看，每一步的计算量是同样值得重视的。我们不应当为片面追求高阶而过分增加每一步的计算量。例如，当一个 4 阶公式的每步运算时间超过 2 阶的两倍时，那么从总的计算时间来看是得不偿失的。

表 2-3　不同阶公式达到 16000 位结果所需的计算次数

阶数和 c 值	所需迭代次数 k（d 为初值位数）	
1 阶 $c=\frac{1}{10}$	$d+k=16000$,	即 $k=16000-d$
1 阶 $c=\frac{1}{1000}$	$d+3k=16000$,	即 $k=(16000-d)/3$
2 阶 $c=1$	$d\times 2^k=16000$,	即 $k=\log_2 16000-\log_2 d<14$
4 阶 $c=1$	$d\times 4^k=16000$,	即 $k=\log_4 16000-\log_4 d<7$

高阶而又计算量小的公式是数学专家们梦寐以求的，但在 π 值计算中要找到一个能用的高阶公式实在不容易。事实是，在历史的长河中，人们用于算 π 的几乎全是 1 阶收敛的公式。快慢主要表现在系数 c 的数值和每步差别有限的计算量上。

首先，让我们回顾第 1 章介绍过的以刘徽割圆术为代表的古典方法。仍用 S 、 S_n 和 S_{2n} 分别记圆、圆内接正 n 边形和正 $2n$ 边形的面积，易知

$$S-S_{2n}=(S-S_n)-(S_{2n}-S_n)$$

由图 1-1，三角形 ACF 面积大于以斜边 AC 为弦的弓形的 p

倍 (按保守的估计， $p=1,2,3$)，即

$$S_{2n}-S_n>p\times(S-S_{2n})$$

最后得

$$S-S_{2n}<\frac{1}{p+1}(S-S_n) \tag{2.9}$$

根据关于收敛阶的定义公式 (2.8)，刘徽割圆术是 1 阶收敛的，系数 $c=1/(p+1)\simeq 1/4$。由于 $(1/4)^4<1/100<(1/4)^3$，所以这里的分析结论与前章根据实际计算结果 (表 1-1) 的观察 (大约每三步得两位) 相吻合。

现在我们来分析 π 值计算的另一类公式。如前所述，它们都是本世纪 60, 70 年代常用的公式，包括式 (2.1), 式 (2.5)～式 (2.7)，特点是基于 $\mathrm{tg}^{-1}x$ 的级数展开式。为使问题简单，我们取

$$\mathrm{tg}^{-1}x=x-\frac{x^3}{3}+\frac{x^5}{5}-\frac{x^7}{7}+\cdots$$

这时 π 值计算收敛速度的分析，本质上就是上述 $\mathrm{tg}^{-1}x$ 级数展开式收敛速度的分析。今记此级数的通项为

$$t_i=(-1)^{i-1}\frac{x^{2i-1}}{2i-1}$$

记其前 k 项和为 $u_k=t_1+t_2+\cdots+t_k$，又记 $\mathrm{tg}^{-1}x$ 的真值为 u, 则有

$$
\begin{aligned}
|u_{k+1}-u| &= \left|(-1)^{k+1}\frac{x^{2k+3}}{2k+3}+(-1)^{k+2}\frac{x^{2k+5}}{2k+5}+\cdots\right| \\
&= \left|x^2\left((-1)^{k}\frac{x^{2k+1}}{2k+3}+(-1)^{k+1}\frac{x^{2k+3}}{2k+5}+\cdots\right)\right| \\
&< x^2\left|(-1)^{k}\frac{x^{2k+1}}{2k+1}+(-1)^{k+1}\frac{x^{2k+3}}{2k+3}+\cdots\right|
\end{aligned}
$$

亦即

$$
|u_{k+1}-u| < x^2|u_k-u| \tag{2.10}
$$

这是系数为 $c=x^2$ ($|x|<1$) 的 1 阶收敛式。显然，$|x|$ 的值越小，收敛速度越快。由于式 (2.1) 中有最大的 x 值 ($x=1/5$)，它的计算速度自然比不上其他的几个式子。又式 (2.5) 和式 (2.6) 两式的最大 x 值分别是 1/8 和 1/18，差距较大，所以式 (2.6) 明显地优于式 (2.5)。

为了说明高阶收敛公式对超高精度 π 值计算的必要性，让我们实际地体会 1 阶公式算 π 的运算量。就以其中最快的式 (2.6) 为例，并且只考虑计算 $\mathrm{tg}^{-1}\frac{1}{18}$。由式 (2.10) 和 $(1/18^2)^2 \simeq 10^{-5}$，易知大约每迭代 2 步得 5 位。如果要算 100 万位，就要迭代 40 万步。虽然，每步迭代只有几个运算，但它们却是 100 万位数字之间的运算。如果按手算的方法理解计算机上的运算，那么，两个 100 万位数字之间的乘法就需要 $(100\text{万})^2 = 1$ 万亿个数位之间的乘法和加法运算。仅此一项，40 万步下来，就是 40亿^2 个数位之间的乘法和加法计算。用每秒一亿次运算的机器作业，也得等上 40 亿秒 (约合 111 万小时)，这是不能接受的。正是这个原因，在高阶收敛公式诞

生之前，计算机上算 π 的纪录一直在几十万位上徘徊。

2.2.2 当今国际上 π 值计算的几个新公式

1. 基于算术几何均值的公式

最早出现的二阶收敛公式是 1976 年萨拉明 (Salamin) 基于“算术几何均值” (英文缩写为 AGM) 概念的迭代式。过去我们知道的 n 个数的“算术均值”是指这 n 个数相加之和除以 n；而 n 个数的“几何均值”是指这 n 个数相乘之积开 n 次方。萨拉明的算术几何均值含义要比单纯的算术均值或几何均值更广一些。这时设 a_0, b_0, c_0 是三个正数，且满足 $a_0^2 = b_0^2 + c_0^2$，今定义算术均值序列 $\{a_k\}$，和几何均值序列 $\{b_k\}$ 如下：

$$a_{k+1} = \frac{1}{2}(a_k + b_k) \tag{2.11}$$

$$b_{k+1} = \sqrt{a_k b_k} \tag{2.12}$$

这两个序列的值在计算过程中越来越靠近，并最终稳定在一个值上，这个值就称为 (a_0, b_0) 的算术几何均值，用符号简记为 $AGM(a_0, b_0)$。现在再由序列 $\{a_k\}$, $\{b_k\}$ 定义序列 $\{c_k\}$

$$c_k^2 = a_k^2 - b_k^2 \tag{2.13}$$

并给出初值

$$a_0 = 1, \quad b_0 = 1/\sqrt{2} \tag{2.14}$$

最终得 π 的计算式

$$\pi = \frac{4(AGM(a_0, b_0))^2}{1 - 2^2 c_1^2 - 2^3 c_2^2 - 2^4 c_3^2 - \cdots} \tag{2.15}$$

我们不想再用理论证明上式确实是二阶收敛的，因为这类证明实在乏味。下面只是通过实际的数值计算结果 (见表 2-4) 来说明此式确实收敛神速。计算中使用 BASIC 中的双倍精度 (约 16 位十进制数) 运算，并记上式分母的前 k 项之和为 S_k。此外，为看出 π 的估计值随迭代步数 k 的变化过程，我们把 a_k 和 b_k 分别看作是 $AGM(a_0, b_0)$ 的估计值就可以算得对应

表 2-4　算术几何均值法的计算过程

公式		结果
$S_0 = 1,\ \ a_0 = 1,\ \ b_0 = 1/\sqrt{2}$	=	0.707106781186547
$a_1 = \frac{1}{2}(a_0 + b_0)$	=	0.853553390593274
$b_1 = \sqrt{a_0 b_0}$	=	0.840896415253715
$c_1^2 = a_1^2 - b_1^2$	=	0.021446609406726
$S_1 = S_0 - 2^2 c_1^2$	=	0.914213562373095
$\pi_{a_1} = 4a_1^2/S_1$	=	3.187672642712109
$\pi_{b_1} = 4b_1^2/S_1$	=	3.093836321356054
$a_2 = \frac{1}{2}(a_1 + b_1)$	=	0.847224902923494
$b_2 = \sqrt{a_1 b_1}$	=	0.847201266746891
$c_2^2 = a_2^2 - b_2^2$	=	0.000040049756187
$S_2 = S_1 - 2^3 c_2^2$	=	0.913893164323602
$\pi_{a_2} = 4a_2^2/S_2$	=	3.141680293297657
$\pi_{b_2} = 4b_2^2/S_2$	=	3.141505000352047
...	...	...
$\pi_{a_3} = 4a_3^3/S_3$	=	3.141592653895458
$\pi_{b_3} = 4b_3^2/S_3$	=	3.141592653284151
...	...	...

的 π_{a_k} 和 π_{b_k}，显然 $\pi_{a_k} > \pi_{b_k}$。从表 2-4 的计算结果看，第 1 步精确到小数点后 1 位，第 2 步是 3 位，第 3 步 9 位（第 7 章的表 7-1 还列出其后各步的精确位数分别是 19, 41, 85, 172,$\cdots$）。所以可以相信式 (2.15) 是 2 阶收敛的公式。1983 年，Kanada 等人曾经使用这个公式算出了 1600 万位 π 值，创造了当时的世界纪录。

2. 速度最快的波尔温 (Borwein) 高阶公式

在 π 值的高阶算法研究中，最好的结果来自两个都叫波尔温的数学家。他们首先在 1984 年发表了 **另一个 2 阶收敛公式**：

$$a_0 = \sqrt{2}, \quad b_0 = 0, \quad p_0 = 2 + \sqrt{2}, \tag{2.16}$$

$$\begin{cases} a_{k+1} = \dfrac{(\sqrt{a_k} + 1/\sqrt{a_k})}{2} \\ b_{k+1} = \dfrac{\sqrt{a_k}(1 + b_k)}{a_k + b_k} \\ p_{k+1} = \dfrac{p_k b_{k+1}(1 + a_{k+1})}{1 + b_{k+1}} \end{cases} \tag{2.17}$$

式中 $p_k \to \pi$。

1986 年，他们又发现了对于某些基本常数，可以构造更高阶收敛公式的一般技术，特别是 **$1/\pi$的 4 阶收敛公式**：

$$a_0 = 6 - 4\sqrt{2}, \quad y_0 = \sqrt{2} - 1, \tag{2.18}$$

$$\begin{cases} y_k = \dfrac{1 - (1 - y_{k-1}^4)^{1/4}}{1 + (1 - y_{k-1}^4)^{1/4}} \\ a_k = a_{k-1}(1 + y_k)^4 - 2^{2k+1} y_k (1 + y_k + y_k^2) \end{cases} \tag{2.19}$$

式中 $a_k \to 1/\pi$。

表 2-5　波尔温公式在前几次迭代中的有效位数

公式	各步的有效位数							
阶数	1 步	2 步	3 步	4 步	5 步	6 步	7 步	8 步
2 阶	3	7	18	39	83	170	344	693
4 阶	7	40	170	693	2787	11170	44700	178823

表 2-5 记录了我们实际使用上述两个高阶公式在前若干次迭代中所得到的有效位数，从中可以看出，这些公式确实是达到了他们所说的收敛阶数。

Bailey 于 1988 年计算 2936 万位 π 值时就曾同时使用波尔温的 2 阶和 4 阶式相互验证。其后，所有其他人超过亿位的计算工作 (见表 2-1) 都与波尔温的 4 阶公式有关。本书作者也曾使用 4 阶公式算出 2016 万位 π 值。总之，波尔温的 4 阶公式是迄今计算 π 值的最快公式。在本书其后提供的 BASIC 程序中，亦将包含此算法。

3. 可从任一位起算的 BBP 公式

90 年代后期，π 值算法研究中出现了新的方向，其特点是计算 π 的某一位的值时，不需要像传统那样首先算出它之前的所有位的值。有关公式是由白莱、波尔温、布罗菲三人于 1997 年提出的，这是形式并不复杂的一个无穷和：

$$\pi = t_0 + t_1 + t_2 + \cdots + t_k + \cdots \tag{2.20}$$

和式中的通项是

$$t_k = \frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right) \tag{2.21}$$

人们用三个作者姓氏的第一个字母 BBP 来简称这个算法。BBP 的作者们曾列出根据此式算出的从第 100 亿位开始的十多位 π 值。本书作者也曾使用 BBP 公式算出 120 亿位附近的 π 值。

最值得注意的是，BBP 算法中所说的“位”不是通常人们习惯的十进制位，而是 16 进制的位。注意到式中相邻两项的差异中正好含有 16 的因子，从某 (16 进制) 位开始计算少数几位就不会太难。例如计算 π 的小数点后第 $n+1$ 位开始的 m 位数等价于计算 $16^n\pi$ 的前 m 位小数，于是可用 16^n 乘式 (2.20) 得 $16^n\pi$ 的表达式，其通项记为

$$16^n t_k = 16^{n-k}\Big(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\Big) \quad (2.22)$$

为计算 $16^n\pi$ 的前 m 位小数，可把这些项划分为两部分：其中第一部分是 $k \leqslant n$ 的各项，在求和的过程中略去整数只留小数；第二部分是 $k > n$ 的各项，它们都是小数。当所需的 m 很小时，上述计算方法有较好的效率。特别是不留整数可以避免含有很多位的多精度数的运算与存储的麻烦（这时只需用双倍精度操作）。容易明白，这里的运算量主要集中在第一部分 (若 n 很大时)，因为第二部分所需的计算只是很少的几项 (数目与 m 相仿)。

通过上面的论述可知，BBP 公式的计算量大体与 n 成正比，但 m 要取小值 (例如 $m \leqslant 10$)，否则需要分段重复计算。例如需要计算第 50000 位之后的 24 位，但一次计算只允许 8 位，即 $m = 8$，这时就需要重复 3 次计算：即第 1 次令 $n = 50000$ 得到第 50001 ~ 50008 位，第 2 次令 $n = 50008$ 得

到第 50009 ~ 50016 位，第 3 次令 $n = 50016$ 得到第 50017 ~ 50024 位 (详见第 4 章 4.2 节的计算程序)。

另外，如果像传统那样计算 π 的前面许多许多位时，BBP 公式的速度远不如波尔温的高阶公式，因为 BBP 公式本质上只有 1 阶的收敛速度。事实上，每项求和只能增加 16 进制的 1 位 (相当于 10 进制的 $\lg 16 \simeq 1.20412$ 位)。毕竟 BBP 公式能较快地算出 π 的某一指定位的值，而且不需要像传统那样首先算出它之前的所有位的值，这确实是 π 值计算中的一个重大事件。并引起同行们的极大兴趣。受此启发，其他人也提出过类似的式子，例如可用于 4 进制位形式的如下式子：

$$t_k = \frac{(-1)^k}{4^k}\left(\frac{2}{4k+1} + \frac{2}{4k+2} - \frac{1}{4k+3}\right) \tag{2.23}$$

尽管人们都已认识到：如果发现一个能够直接用于我们熟悉的十进制 (而不是二进制) 位的公式将有更大的应用价值，但遗憾的是至今尚未能如愿。

第 3 章　本书使用的程序语言

通过前面的讨论，超高精度 π 值计算的数学公式已经解决，剩下来的问题是具体编制相应的计算机程序。本书使用新版本的 BASIC 程序设计语言，它具有结构化和模块化等当代高级语言特色。本书给出的 BASIC 程序既可以在 DOS 自带的 QBasic 环境下解释执行，也可以在 DOS 环境下的 Quick BASIC，甚至 Windows 下的 Visual BASIC 内编译执行。本章的任务是简要介绍我们将要使用的新版 BASIC 的某些特点和上机操作方法。如果读者已经熟悉这些 BASIC，那么除了 3.1 节中关于选用 BASIC 的原因之外的本章大多数内容都可以跳过。

3.1　QBasic 与 Quick BASIC

由于 π 值计算是典型的科学工程计算，它涉及庞大的数据存储和运算，专业人士首选的程序语言通常应是 FORTRAN。但考虑到非专业人士或多或少地接触过 BASIC 语言，而且每部 PC 机上都自带了 BASIC 系统，这样一来，选用 BASIC 就有了合理的理由。首先是读者对这种语言不陌生，自然容易理解本书的内容；其次，想亲自动手算 π 的人们也可以省去四处寻觅或购买其他程序语言系统的麻烦；最后是近些年来 BASIC 语言的巨大进步，一些新的 BASIC 系统在保持原来

BASIC 易学易用特点的同时，还吸收了其他高级语言中的先进成分。例如，在新的 BASIC 系统中，每个语句前面的行号已非必须，赋值语句中的符号 LET 亦可有可无；条件、转移与循环语句更加丰富和完善；子程序的调用也可以有其他高级语言那样的形式。这些新特点，尤其是结构化和模块化的特点，使得一个 BASIC 程序与其他高级语言程序十分相似，以至于仅凭某个程序的少数几行内容很难断言它是来自 BASIC 还是 FORTRAN 或别的什么语言。

我们使用的 BASIC 语言具有上述新特色，它的蓝本是 QBasic 和 Quick BASIC 。 Quick BASIC 是微软公司 1987 年推出的新一代 BASIC 语言，它具有结构化和模块化等特点。Quick BASIC 又是一个集成的程序开发环境，可通过菜单操作实现编程、调试、修改、运行 (解释或编译执行均可) 。其后微软公司又在 Quick BASIC 基础上推出 QBasic 。后者作为 DOS 的一部分最早出现在 DOS 5.0 上， DOS 中的文本编辑器 EDIT 就需要有它的支持才能运行。由于它被看作是 DOS 中的一个“命令”，所以规模受到限制。不过在语法结构上仍包括了上面介绍过的种种新特点，而且仍然是一个集成的程序开发环境，有关的上机操作方法也与 Quick BASIC 十分相似。对用户而言，它们之间的最大差别是， QBasic 没有编译执行功能，用户的程序只能解释执行。值得强调的是，一个可在 QBasic 上解释执行的程序，往往可以不经任何修改就能在 Quick BASIC 中解释或编译执行。

本书给出的算 π 程序，不仅可以在 DOS 下的 QBasic 和 Quick BASIC 上运行，而且也可以在 Windows 下的 Visual BASIC 上运行，是具有上述新特色的高度兼容的 BASIC 程

序。众所周知，一个 BASIC 程序在编译执行时，其运行速度要比解释执行时更快，对于我们的算 π 程序来说，这一点显得尤为重要。在 Quick BASIC 上实测表明，在几千位的计算中，其时间差别可以达到三五十倍。但在 Visual BASIC 中，编译与解释执行两者之间的时间差别没有这样明显，同样是在几千位的计算中，运行时间比值大体上只有 2 : 5 。不过在编译执行时，Visual BASIC 所需的时间要比 Quick BASIC (4.0) 多 30% 至 40% 。

本书不是 BASIC 的入门书，介绍 BASIC 的目的，是为了使非专业读者能看懂和使用我们的任意精度运算程序，特别是为熟悉老 BASIC 的读者铺砌一条通向当代 BASIC 的捷径。本章将通过一些简单的程序例子，由浅入深地说明这些新 BASIC 语言的特点和最简单的上机操作方法。这里介绍的实际上是 QBasic 或 Quick BASIC ，至于 Visual BASIC 中的某些特别之处，将留在下章结合算 π 的具体程序给出必要的说明。

3.2 用例子展示新 BASIC 特色

数的单精度和双精度表示

众所周知，在计算机上进行数值运算时，其精度总是有限的。例如有如下一段 BASIC 程序：

```
x = 54321
y = .678901
z = x + y
PRINT x, y, z
END
```

在现代的 BASIC 中，语句行号并不是必须的，所以我们的程序都不用行号。上述程序运行后印出结果

```
54321      .678901      54321.68
```

此处 z 的结果只有 7 位有效数字 (有舍入)，而不是精确的 11 位。这是因为 BASIC 中的浮点运算，如不特别声明，都默认只有“单精度”。这时每个数的内部存储占 4 个字节 (共 32 位二进制)，其中一个字节用来表示数的指数部分，其余 3 个字节用来存放数字，约合 7 位十进制数字，所以印出来的结果只有 7 位有效数字。为了得到更精确的结果，可对程序中的常数和变量作“双精度”说明，办法之一是在常数和名字之后都加 # 号，这时程序为

```
x# = 54321#
y# = .678901#
z# = x# + y#
PRINT x#, y#, z#
END
```

印出的结果中，z 得到了希望的 11 位有效数字：

```
54321      .678901      54321.678901
```

BASIC 中双精度数的内部存储为 8 个字节，含有不少于 15 位十进制的精度。在新 BASIC 中，双精度型变量也可以通过 DIM 语句来说明，而不必再尾加 # 号，这时上述程序变为

```
DIM x AS DOUBLE, y AS DOUBLE, z AS DOUBLE
x = 54321#
y = .678901#
z = x + y
PRINT x, y, z
END
```

值得注意的是，这里的 DIM 语句只是分别说明 x, y, z 均为双精度变量(注意，不是数组)。如果要说明数组，则数组名后应有维数定义，例如

```
DIM d(20) AS DOUBLE
```

说明 $d(20)$ 是个数组，其元素是 $d(0), d(1), d(2), \cdots, d(20)$。

QBasic 的数值型变量或数组可有四种类型，现归纳在表 3-1 中。

表 3-1　QBasic 中数值型变量或数组的四种类型

类型	类型名	符号	字节	举例
整　型	INTEGER	%	2	n% 或 DIM n AS INTEGER a%(9) 或 DIM a(9) AS INTEGER
长整型	LONG	&	4	L& 或 DIM L AS LONG m&(12) 或 DIM m(12) AS LONG
单精度实　型	SINGLE	!	4	s! 或 DIM s AS SINGLE r!(62) 或 DIM r(62) AS SINGLE
双精度实　型	DOUBLE	#	8	d# 或 DIM d AS DOUBLE v#(31) 或 DIM v(31) AS DOUBLE

注：符号！可有可无。

BASIC 本身不支持比双精度更高的精度，但用户可以在 BASIC 上自己建立用于更多位数运算的“多精度系统”(详见第 5, 6 两章)。

[例 3.2.1]　计算 n 的阶乘 $n!$。

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1$$

下面的程序用于计算 15!

```
DIM p AS DOUBLE
n = 15
p = 1#
FOR k=1 TO n
  p = p * k
NEXT k
PRINT n; p
END
```

如上所述，常数后的 # 号用于强调它是双精度型常数 (在这个具体的程序中，为省事，1# 中的 # 号也可以不写，因为它们给左边的双精度型变量赋值时会自动转换为所需的类型，而且 1 本身是整数，在转换过程中也不会招来误差)。这里的循环 "FOR ··· NEXT" 方式与老 BASIC 无异。在循环结束后，程序输出 n 及其阶乘值。我们再次强调，这里的 DIM 语句只是说明 p 为双精度变量而不是数组。

[**例 3.2.2**] 计算两矢量的内积和夹角。

设有两个 n 维矢量

$$a = (a_1, a_2, \cdots, a_n), \quad b = (b_1, b_2, \cdots, b_n)$$

它们的内积定义为

$$a \cdot b = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \tag{3.1}$$

两矢量的夹角余弦是

$$\cos\alpha = \frac{a \cdot b}{\sqrt{a \cdot a}\sqrt{b \cdot b}} \tag{3.2}$$

因为要多次计算内积，所以先编出一个计算内积的函数过程如下：

```
FUNCTION dotp (a(), b(), n)
  ab = 0
  FOR i = 1 TO n
    ab = ab + a(i) * b(i)
  NEXT i
  dotp = ab
END FUNCTION
```

在 BASIC 中，函数过程由保留字 FUNCTION 开始，以 END FUNCTION 结尾。在开头的保留字后是函数名及有关的参数，本例是 $dotp(a(\ ),b(\ ),n)$，其中 n 是单个变量，而带 ( ) 的 a 与 b 则是数组。接其后的是实现该函数功能的语句块，本例中是先把一个变量 ab 置零值，然后通过循环语句把相应分量的乘积 $(a(i)*b(i),\ i$ 从1到 $n)$ 逐步累加到 ab 中，并在循环结束后把所得的结果 (内积) 给函数名赋值。在函数过程中，给函数名赋值是必不可少的。

有了计算内积的函数 *dotp* 之后，两矢量的夹角余弦就不难计算了。下面便是这样的一个程序例子。

```
DIM a(5), b(5)
READ a(1), a(2), a(3), a(4), a(5)
READ b(1), b(2), b(3), b(4), b(5)
DATA 1.5, 2.0, 3.3, 4.2, 5.0
DATA -5.0, 2.5, -3.0, 4.6, 5.5
n = 5
aa = dotp(a(), a(), n)
bb = dotp(b(), b(), n)
cosa = dotp(a(), b(), n) / SQR(aa * bb)
PRINT aa; bb; cosa
END
```

例中的 DIM 语句说明 a 与 b 都是 5 维数组，它们的值分别由两个 DATA 语句给出，并由两个 READ 语句依次读入。通过对 *dotp* 函数的 3 次引用 (每次都有不同的参数)，分别得到内积 $a\cdot a$, $b\cdot b$ 和 $a\cdot b$，进而根据定义 (3.2) 算出两矢量的夹角余弦 $\cos\alpha$。最后印出如下结果：

```
59.78,    91.66,    .4649893
```

[例 3.2.3] 计算组合 C_m^n。

排列与组合是初等数学中的重要内容，组合的定义是

$$C_m^n = \frac{m!}{n!(m-n)!} \tag{3.3}$$

式中 $n!$ 是阶乘，即 $n! = n(n-1)(n-2)\cdots 3\cdot 2\cdot 1$。为了减少运算量，可视 n 的大小分别选用如下的组合等价式

$$C_m^n = \frac{m(m-1)(m-2)\cdots(n+1)}{(m-n)!}$$

$$C_m^n = \frac{m(m-1)(m-2)\cdots(m-n+1)}{n!}$$

这里给出一个完整的程序，它根据键盘输入的 m,n 值计算并输出组合数 C_m^n。主程序按反复执行设计 (用 DO$\cdots$LOOP 循环形式)，直至输入的 $m\leqslant 0$ 才结束。主程序通过调用一个子程序 *cmn* 来完成组合的计算。在子程序 *cmn* 中，将根据 n 是否大于 $m/2$，自动选取等价式 (3.3) 中运算量最小的一个公式来计算组合，其结果放在双精度变量 c 内。在计算组合过程中所需的连乘积 (包括阶乘) 又是通过引用函数 *dprod#* 来完

成的，函数名的最后一个符号为 # 意味着该函数的结果是双精度型的。函数 *dprod#* 有两个参数，用以表示开始和结束连乘的两个数，且规定大数在前，小数在后，于是 $dprod\#(n,1)$ 的结果就是阶乘 $n!$ 。

```
DECLARE SUB cmn (m!, n!, c AS DOUBLE)
DECLARE FUNCTION dprod# (nbig!, nsmall!)
DO
  INPUT "m="; m
  IF m <= 0 THEN EXIT DO
  INPUT "n="; n
  CALL cmn(m, n, cc#)
  PRINT "m, n, Cmn ="; m; n; cc#
LOOP
END

SUB cmn (m, n, c AS DOUBLE)
IF n <= m / 2 THEN
  k = n
ELSE
  k = m - n
END IF
IF k = 0 THEN
  c = 1
  EXIT SUB
END IF
c = dprod#(m, m - k + 1) / dprod#(k, 1)
END SUB

FUNCTION dprod# (nbig, nsmall)
d# = nbig
```

```
FOR i = nbig - 1 TO nsmall STEP -1
  d# = d# * i
NEXT i
dprod# = d#
END FUNCTION
```

本程序中出现一些老 BASIC 没有的语法现象，例如在主程序的开头的两个 DECLARE 语句，它是由 QBasic 或 Quick BASIC 系统在编辑或存盘时自动加上的，用以说明本程序含有用户自己编写的子程序 *cmn* 和函数 *dprod#*，并列出这些过程的参数和类型。但你总可以删去它们，尤其是你的子程序已经有了变化之后。再如出现保留字 EXIT，它的含义与 GO TO 相近，都是用来跳过某些程序语句。但 EXIT 不需要语句号码或语句标号，从而使程序更加易读，是结构化程序中被推荐用来代替颇受非议的 GO TO 语句的语法成分。本程序中先后出现 EXIT DO (见主程序，用于跳出循环) 和 EXIT SUB (见子程序，用于跳出子程序)，今后读者还将会看到 EXIT FOR (跳出 FOR ··· NEXT 循环) 和 EXIT FUNCTION (跳出函数) 等语句。

3.3 上机操作初步

如 3.1 节所述，QBasic 和 Quick BASIC 都是微软公司推出的一个集成型程序开发环境，可通过菜单操作实现编程、调试、修改、运行，而且它们的上机操作方法也都十分相似，只是后者更为完备，程序不仅可以被解释执行，还可以被编译执行，从而大大缩短了程序的作业运行时间。所以这小节的重点是介绍 QBasic 的基本操作，至于 Quick BASIC，只需

介绍在 QBasic 内没有的编译执行方法便妥。虽然本书的程序可以在 Windows 的 Visual BASIC 环境内运行，但我们不打算介绍这方面的操作方法，因为要说清楚有关的操作，特别是 Windows 98 下的操作实在需要很多篇幅，这对于只有十万字左右的本书来说，恐有喧宾夺主之嫌。

1. 准备工作

首先在硬盘上建立一个子目录 (假设为 C:\PI) , 并把 DOS 子目录 (假设为 C:\DOS) 的如下两个文件

```
QBASIC.EXE    QBASIC.HLP
```

拷贝到此子目录内。相应的操作命令是

```
C:>MD  C:\PI
C:>COPY  C:\DOS\QBASIC.*  C:\PI
```

其后转入此子目录内

```
C:>CD  C:\PI
```

注 1：“C:>”是 DOS 自动显示在屏幕上的提示符，不是用户键入的符号，为方便起见，今后本书将不再出现这些符号。

注 2：这里假设用户的 DOS 版本是 5.0 或更高，否则请升级之后再作上述操作。

注 3：如果用户能够找到 Quick BASIC 系统，最好先把它安装好之后，再把 QBasic 的上述两文件拷贝到 Quick BASIC 所在的子目录内 (为方便，仍假设此子目录名为 C:\PI) 。

注 4：如果用户使用汉字系统，则上述操作可放在启动汉字系统之前或之后执行。但是我们建议最好不要使用汉字，免得它们与 BASIC 争夺宝贵的常规内存。

2. 进入 QBasic

进入 QBasic 的方法是使用命令 QBASIC 或 QBASIC *fn*，因 QBasic 是集成开发环境，使用命令 QBASIC 进入之后，用户可以利用它提供的环境编辑 (录入) 自己的程序，其后还可以把编辑好的程序记录在磁盘上的一个文件内 (假设其名字为 *fn*.BAS，扩展名必须是 BAS，详细操作见后), 以备日后再次使用。但是 QBasic 允许用户使用自己习惯的其他 (文本) 编辑环境 (如 DOS 的 EDIT 或 Windows 的 NOTEPAD) 先把程序录入一个文件内 (为方便仍假设其名字为 *fn*.BAS)，其后使用命令 QBASIC *fn* 进入 QBasic (注意：此命令中的扩展名 BAS 可以省略)。

3. QBasic 的工作窗口

使用命令 QBASIC *fn* 进入 QBasic 时，用户可以立刻在屏幕上看到程序 *fn*.BAS，否则只能看到一个欢迎画面 (见图 3-1)，并且告诉用户，此时若按回车键 Enter 可以查看 QBasic 的指南，或按 ESC 键清除含有此欢迎词的对话框。清除了对话框后，屏幕上就出现编程用的工作窗口 (见图 3-2)。

4. 编辑和运行 QBasic 程序

图 3-2 是 QBasic 中最常见的一种窗口，可用于编辑和运行程序。图中的所有汉字是我们为解说而加上的，去掉这些汉字之后，整个窗口就只剩下顶上的菜单栏和底部的提示栏。这个窗口被分为上、下两个子窗，上面的大窗口称为 **观察窗** 或程序窗，用来编辑程序；下面的小窗口称为 **立即窗**，在其上键入的命令将会被立即执行。

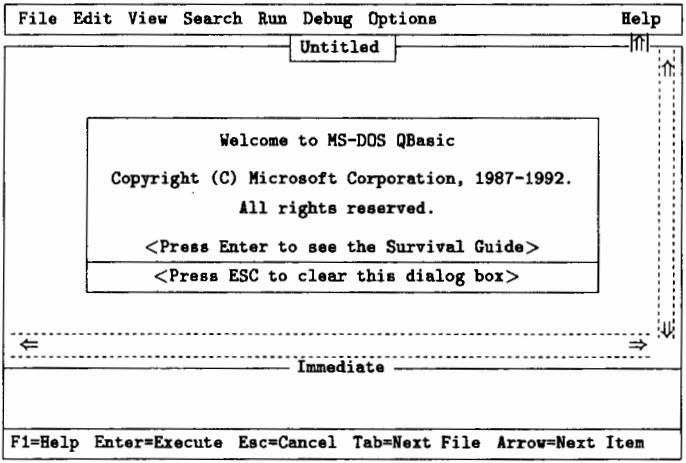

图 3-1 QBasic 工作窗口中的欢迎画面

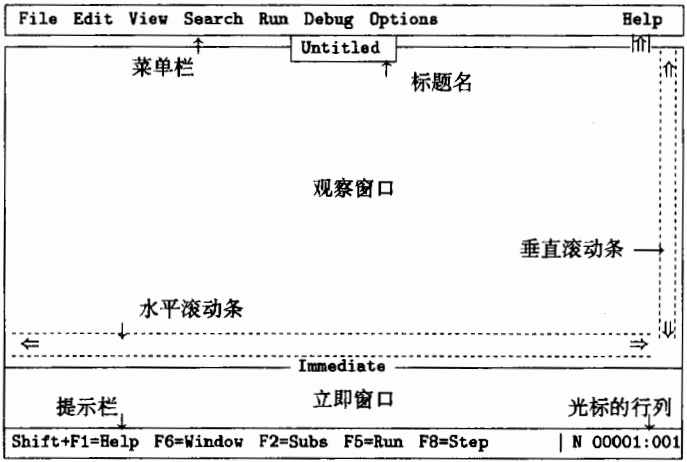

图 3-2 QBasic 中编程用的工作窗口

编辑程序时，可以使用键盘中的字母、数字和运算符键，以及光标移动键 $\leftarrow,\rightarrow,\uparrow,\downarrow$ 等。值得注意的是，当光标离开刚输入的一行时， QBasic 立刻对键入的该行内容作语法检查，凡是遇到关键字都自动变成大写字母 (如果用户是用小写字母键入的话)，并在运算对象之间统一插入一个空格 (如果用户没留空格或留的空格不规范的话)。另外，如果用户是从磁盘读入程序文件 (亦请参看 5.)，那么它在观察窗上被显示之前， QBasic 也对该文件的各行内容作同样的处理，其中包括自动除去换行符 (如果有的话)，把原来用换行符拆成多行的内容合并成一个长行。图 3-3 中录入了一个计算 $\sqrt{101},\sqrt{102},\sqrt{103},\sqrt{104}$ 的简单程序例子，可供参考。

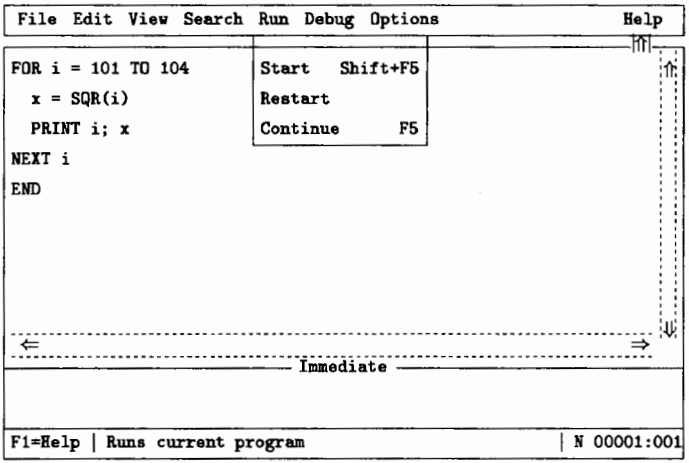

图 3-3 QBasic 工作窗口中编程和运行

如果想执行已在观察窗内的那个程序，请用鼠标或键盘

选取窗口顶部 Run 菜单中的 Start 命令。这时的键盘操作是依次按 Alt 键，R 键和 S 键，其中按 Alt 键后，窗口顶部菜单栏中的每个菜单的首字母变亮 (此时也称菜单被激活)，继而按 R 键时，弹出 Run (运行) 菜单 (见图 3-3)，再按 S 键表示要开始 (解释) 执行该程序，此时窗口消失，程序执行完毕后 (执行时间随问题而异，当算很多位 π 值时，等待时间可能很长)，屏幕上出现结果 (见图 3-4)，并且屏幕的底部出现一串提示：

```
Press any key to continue
```

其含义是按任何一个键以继续进程，亦即返回原来窗口。

```
C:\PI>QBASIC
 101   10.04988
 102   10.0995
 103   10.14889
 104   10.19804




Press any key to continue
```

图 3-4 QBasic 的输出结果屏幕

注 1：如果用户的程序中没有一个 PRINT 语句被执行，或者使用了 OPEN 和 PRINT # 语句把结果输出到一个文件

中 (见下章 [例 4.1.2]~ [例 4.1.6]) , 那么屏幕上不会出现结果, 但在计算结束时, 上述英文提示仍会出现在屏幕底部。

注 2：如果返回窗口后, 又要再查看输出屏幕上的结果, 请在窗顶 View 菜单内选 Output Screen 。

观察窗口与立即窗口之间可以通过按 F6 键交替切换。例如, 欲立即计算 $\sqrt{2}$, 只需按 F6 , 直至窗口中字符 Immediate (立即) 的位置明亮起来 (即立即窗被激活) 之后, 键入命令

```
PRINT  SQR(2)
```

回车后窗口消失, 屏幕上出现结果 1.414214 , 此屏幕的形式与上述运行观察窗程序后的输出相同 (见图 3-4 , 但原来该图的 4 行结果被替换为现在的一行结果) 。其后也可以按照屏幕底的英文提示, 按任何一个键返回到原来的窗口。

5. QBasic 程序的磁盘存取

在 QBasic 观察窗口中经编辑、修改所得的程序可以存入磁盘。另一方面, 也可以把磁盘上的程序 (或计算结果等其他文本文件) 读入观察窗口。其方法是用鼠标或键盘选取窗口顶部的 File 菜单, 这时的键盘操作是依次按 Alt 键和 F 键, 其中按 Alt 键后, 窗口顶部菜单栏中的每个菜单的首字母变亮, 继而按 F 键时, 弹出 File (文件) 菜单 (见图 3-5) 。 File 菜单内含有 New, Open, Save, Save As, Print, Exit 等六种选择, 现分类介绍如下。

如欲 **存盘** , 可选 Save (按 S 键, 其含义是存盘) , 也可选 Save As (A 键, 意即改名存盘) , 对于 Save , 若观察窗口的程序已经命名, 则在磁盘上产生一个以此为名字的程序文件。值得注意的是, 存盘速度极快, 时间一闪而过, 且存盘

后屏幕没有改变，以至有些用户误认为此操作尚未响应，为确保真正存盘，往往又多余地再发一次命令。若观察窗口的程序尚未命名 (上述几个窗口中，标题名的位置上均出现字符 Untitled，其含义是观察窗的程序尚未命名)，则用户选 Save 等价于选 Save As。对于后者，QBasic 立刻在观察窗上弹出一个 Save As 对话框，这时用户把自己需要的名字输入到标有 File Name: 的框内并回车，当改名存盘完成后，对话框消失，重新回到编辑状态。

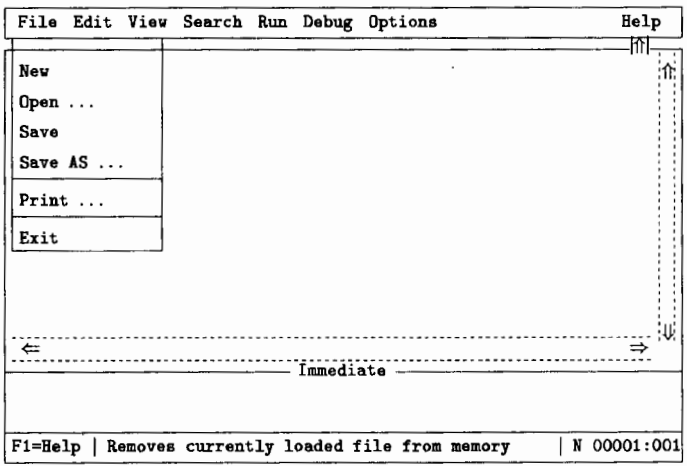

图 3-5 QBasic 弹出的 File 菜单

从磁盘上 **读取** 文件请选 Open (或按 O 键，意即打开文件)，这时观察窗上弹出一个 Open 对话框 (见图 3-6)。此框划分为几个小区，其中最常用的两个是 File Name (欲打开的文件名) 和 Files (当前子目录中以 BAS 为扩展名的全部文件

名)。通常用户可用鼠标在 Files 小区中选出他想打开的程序名 (在该名字上单击或双击鼠标左键。双击，则该程序被直接打开；单击，该名字被传送到 File Name 小区上。) 如果要打开的程序或其他文件名不在 Files 小区上，就只好直接把要打开的文件名键入 File Name 区上，再按回车键或用鼠标选对话框中的 <OK>，文件就被打开。

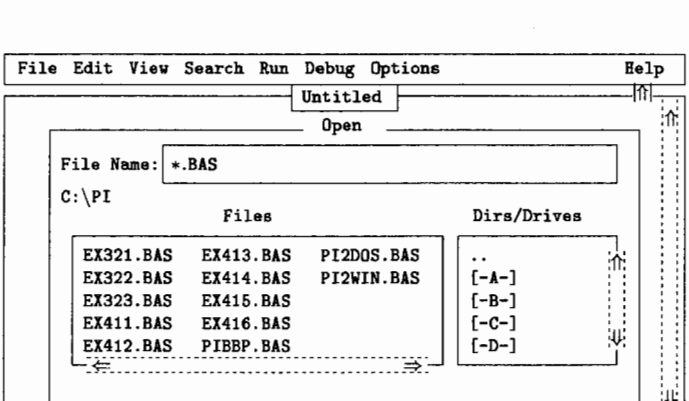

图 3-6 QBasic 中的 Open 对话框

注 1：如 2. 所述，文件被读入观察窗时，如果用户的 BASIC 程序是由其他文本编辑器录入的，那么在保留字的大、小写，空格数目和换行处理等方面将有差别。

注 2：如果用户的程序含有过程模块 (指子程序或函数)，则观察窗上只能看到主程序。若要看过程模块时需在窗顶 View 菜单内选 SUBs。例如 3.2 节的 [例 3.2.3] (其程序

名为 EX323.BAS) 如被读入，则观察窗上只能看到其中的主程序 (前 12 行)。当在 View 菜单内选 SUBs 后，则又弹出另一个菜单，上面有

```
EX323.BAS
cmn
dprod
```

这时若用鼠标左键双击 *cmn* 或 *dprod*，则观察窗上出现相应的 *cmn* 或 *dprod* 模块 (若原观察窗上的程序被修改过，则新模块出现前会弹出要求确认存盘的对话框，详见 6. 和图 3-7)。

6. File 菜单中的其他选择和退出 QBasic

这里介绍 File 菜单中的其他选择： Print, New 和 Exit。

Print (或按 P 键) 的含义是 **打印**，作此选择后，窗口上弹出含有如下内容的打印对话框：

```
    ( ) Selected Text only
    ( ) Current Window
    (·) Entire Program
--------------------------------
 < OK >   < Cancle >   < Help >
```

前三行的含义依次是“只打印所选的内容”，“打印当前窗口的那个模块”和“打印整个程序”。对于 [例 3.2.3] 而言，整个程序包括主程序，子程序 *cmn* 和函数 *dprod* 三者的全部内容；当前窗口的那个模块则可以是主程序或子程序 *cmn* 或函数 *dprod* 三者中的某一个，这要看当时谁在观察窗而定；至于“所选的内容”是指当时观察窗上特别明亮的那些行的内容，它是通过鼠标左键单击要选的首行并且 (不松手) 拖着光标到要选的最后一行 (然后松手) 这样的方式产生的。打印对

话框弹出时，第 3 行的（ ）内有一个黑点“·”，表示要求打印整个程序，如果用户只想打印当前窗口的那个模块或所选的内容，请用鼠标左键单击对话框中对应的（ ），使之出现黑点。选好打印对象之后，再用鼠标左键单击 <OK> 便执行打印。

New (或按 N 键) 的含义是 **编辑一个新程序**，它是为编辑完一个程序之后再编辑另一个新程序而设。当原观察窗的内容存盘后再没有改动时，作此选择后可立刻得到一个空白的观察窗，就如同用命令 QBASIC 进入 QBasic 之后用 Esc 键清除欢迎画面后得到的空白窗口那样，否则会弹出要求确认是否存盘的对话框 (见图 3-7)。

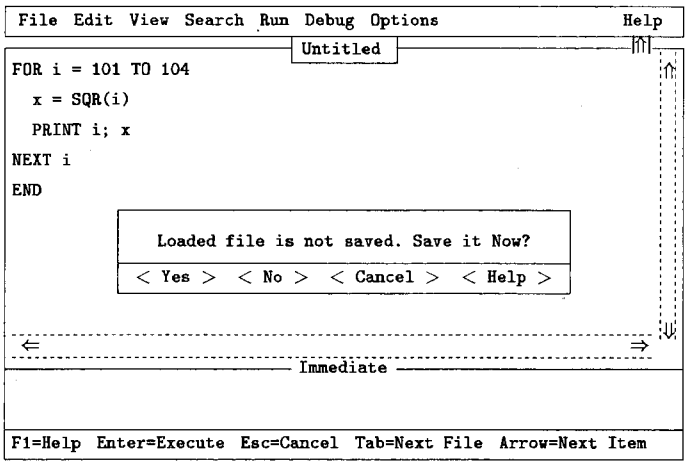

图 3-7 QBasic 中确认存盘的对话框

其中英文字串

```
Loaded file is not saved.  Save it now?
```

是说观察窗口的文件还没有存盘，并问用户是否存盘。当用户选 Yes，则存盘后再清除观察窗；如果用户认为窗口上的内容没有保留价值，可选 No 直接清除观察窗；如果又觉得编辑或修改还没有完成，可选 Cancel 返回观察窗。Exit (或按 X 键) 的含义是**退出**QBasic，当原观察窗的内容存盘后再没有改动时，作此选择将立刻退出 QBasic，并返回 DOS，否则会弹出要求确认是否存盘的对话框 (见图 3-7)。当用户选 Yes，则存盘后再退出；如果用户认为窗口上的内容没有保留价值，可选 No 直接退出；如果此时又觉得尚不该退出 QBasic，可选 Cancel 返回观察窗。

7. Quick BASIC 的上机操作与编译执行简介

Quick BASIC 的上机操作方法与前面介绍的 QBasic 操作方法极为相似。主要差别只有两点：

(1) 进入命令，对于老一点的版本 (如 4.0 版) 是

```
QB      或      QB fn
```

其中 fn 是用户程序名。对于新一点的版本 (如 7.10) 是

```
QBX     或      QBX fn
```

即用 QB 或 QBX 代替 QBASIC。对于较新的 BASIC 系统，一个数组可以超过 64KB，但进入时应附加参数 /AH 指出，这时命令变为

```
QBX /AH      或      QBX /AH fn
```

但一般的 Quick BASIC 也只能在 640KB 的常规内存作业 (其中 DOS 和 BASIC 系统已经占去了相当一部分)，而 π 值计

算却需要多个大数组，所以单个数组超过 64KB 对 π 计算而言意义不大。

(2) Quick BASIC 具有 QBasic 所没有的编译执行功能。所谓编译执行是指用户要求 BASIC 系统先把用户自己的程序编译成一个可执行文件，其后再运行这个可执行文件获得计算结果。对于运算量很大的问题，这种方法可以节省大量的计算机时，对于我们的 π 值计算程序而言更是如此。下面就来介绍这方面的操作方法。类似本节 2. 编辑和运行 QBasic 程序的方法，在观察窗内准备好要运行的程序之后，请用鼠标或键盘选取窗口顶部的 Run 菜单 (若用键盘操作则是依次按 Alt 键和 R 键)，这时弹出 Run (运行) 菜单，它比 QBasic 的菜单有更多的选择，特别是有如下一行英文字符：

```
Make EXE File ...
```

其含义是产生一个可执行文件，用鼠标左键单击该行 (或按 X 键)，又弹出一个更大的对话框，随着版本的升级，该框内容越来越丰富。但用户只需注意以下几项：

```
EXE File Name:
( ) Stand-Alone EXE
(·) EXE Requring BRT  Module
------------------------------------------------------
 < Make EXE >   < Make EXE and Exit >   < Cancel >
```

其中第一行用以指定可执行文件的名字，但一般不用理会，因为它已自动设置了常规的名字 (仍用观察窗口的标题名，但扩展名必为 EXE)。第二三两行用以指定所产生的可执行文件的运行环境。现在第三行的 ( ) 内有一个黑点 "·"，表示产生的文件将来运行时仍需借助 Quick BASIC 环境内的某些

模块的支持；如果用户想产生一个独立的可执行文件，请用鼠标左键单击第二行的（ ），使之出现黑点。最后是确定编译后返回观察窗 (选 <Make EXE>) 还是返回 DOS (选 <Make EXE and Exit>)。如果临时改变了主意，不想编译了，也可以选 <Cancel> 返回观察窗口。

值得指出的是，对于用户自己新设计的程序，通常先在解释执行环境下调试和排除错误，其后才编译执行。

第 4 章　无需多精度系统的算 π 程序

本章给出一些算 π 的 BASIC 程序，其中 4.1 节的程序每个只有十余行，但都可以算出十多位 π 值。给出这些程序的目的是继续通过例子说明 QBasic 程序中的某些语法现象。同样重要的考虑是复习 π 值的计算公式，并且也为第 7 章即将讨论的多精度系统支持下的 π 值计算程序提供一个蓝本。该节既包括第 1 章介绍过的基于几何直观的经典方法程序 (见 [例 4.1.1] ~ [例 4.1.3])，也包括第 2 章介绍过的当今国际上最新、最快方法的有关程序 (见 [例 4.1.4] ~ [例 4.1.6])。一旦读者掌握了下一章将要介绍的用于多精度运算的程序系统的使用方法，便可把这些程序中涉及的双精度算术运算和开方运算逐一替换成调用多精度系统中相应的子程序，并最终获得千万位的 π 值。

4.2 节给出一个可计算几万位 π 值的分段算法程序，虽然它可以轻松地算出小数点后上万位的 π 值，但却不要多精度系统的支持。其公式简单到只有初中程度的读者都可看懂，程序也仅有百余行。这是国际上 1997 年正式发表的最新方法。

4.1　几个计算十余位 π 值的 BASIC 程序

本节程序具有复习算法和程序方面的承上启下性质，而且每个程序都很短，读者不妨逐一上机试算以获得一些感性

的认识。

[例 4.1.1] 根据刘徽割圆术所得的 π 值上、下限。

算法已在 1.1.2 节给出。本程序从边长为 1 的 6 边形起算，它能准确算出小数点后 15 位，即 3.141592653589793 。本书表 1-1 所列的内容是由这个程序算出的。在新的 BASIC 中，除保留字 REM 表示该行内容属于注解，从而不影响实际的计算结果外，还可用单引号 “ ’ ” 代替 REM 。特别是单引号可以出现在某行的中间，其含义是单引号及其后的内容是注解，它们不影响单引号前的该行原有内容。因此我们常用这个办法来指出某程序语句所对应的算法中的公式编号，例如程序中的 ’ see (1.5) 表示请参考 (1.5) 式。

```
REM ex411.bas
DIM n AS LONG, a AS DOUBLE, d AS DOUBLE
DIM s AS DOUBLE, s2 AS DOUBLE
n = 6
a = 1
s = 1.5# * SQR(3#)
FOR k = 1 TO 26
  d = 1 - SQR(1 - a * a / 4)     ' see (1.5)
  s2 = s + n * a * d / 2         ' see (1.6)
  n = n + n
  PRINT k; n; s2; s2 + (s2 - s) ' see (1.3)
  a = SQR(2 * d)                 ' see (1.7)
  s = s2
NEXT k
END
```

通过上面的 BASIC 程序不难看出，如果只是满足于普通的精度 (单或双精度)，用 BASIC 语言来编写一个算 π 程序

是十分容易的。另外，如果已经有人提供了实现任意精度基本运算的子程序系统 (即把加、减、乘、除和开方等运算逐一写成子程序)，那么你要实现任意精度的 π 值计算也是很简单的。这时你只需把上面程序中涉及的算术运算和开方运算逐一改为调用相应的子程序便可 (下同)。

[例 4.1.2] 根据阿基米德方法所得的 π 值上、下限。

古希腊的阿基米德通过计算同边数的圆外切与内接正多边形的周长得到 π 值上、下限。这个方法的实现与刘徽割圆术 (见第 1 章 1.1.2 节) 十分相似。记圆内接正 n 边形边长之半为 b_n，由图 4-1 得

$$h_n = |OF| = \sqrt{1 - b_n^2}, \quad d_n = 1 - h_n \tag{4.1}$$

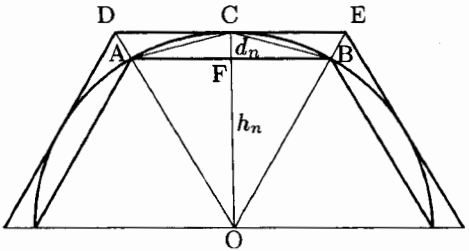

图 4-1 利用圆外切与内接正多边形估计 π 值

因圆周长 $L = 2\pi R$，所以当半径 $R = 1$ 时，其半周长为 π。据此可用圆外切与内接正多边形周长之半来估计 π 值的上、下限。内接正多边形周长之半是

$$L_n = n * b_n \quad (\pi\text{的下限}) \tag{4.2}$$

又因两三角形 OAB 与 ODE 相似，其相应边长成比例，即 $h_n:1$，从而外切正多边形周长之半是

$$L_n/h_n \quad (\pi\text{的上限}) \tag{4.3}$$

此外，由圆内接正 n 边形变为正 $2n$ 边形的半边长递推公式可仿照式 (1.7) 导出

$$b_{2n}=\frac{1}{2}\sqrt{b_n^2+d_n^2}=\sqrt{d_n/2} \tag{4.4}$$

```
REM ex412.bas
DIM n AS LONG, b AS DOUBLE
DIM L AS DOUBLE, d AS DOUBLE
OPEN "ex412.res" FOR OUTPUT AS #2
n = 6
b = .5#                  ' b=a/2, half length
FOR k = 0 TO 19
  L = n * b                        ' see (4.2)
  d = 1 - SQR(1 - b * b)           ' see (4.1)
  PRINT #2, k; n; L; L / (1-d)     ' see (4.3)
  n = n + n
  b = SQR(d / 2)                   ' see (4.4)
NEXT k
END
```

本程序使用 OPEN 语句定义了一个用于存放输出结果的文件 *ex412.res*，并且它被称之为 2 号文件。当程序运行时，带相应文件号的语句 PRINT #2 的输出将按顺序放到该文件中。这里的文件名可由用户任意指定，但在 QBasic 和 Quick BASIC 中，名字 *con* 或 *prn* 有特别含义，这时程序把结果输

出到显视屏幕或打印机上，而不是在硬盘上建立文件 *con* 或 *prn* 。使用文件方式输出结果是方便的，例如在调试程序时，可取文件名 *con* ，这时在屏幕上能看到结果，当需要保存结果时，可把 *con* 改为所需的文件名。

本程序从边长为 1 的 6 边形起算，它的输出如表 4-1 所示。由刘徽割圆术程序 (见 [例 4.1.1] 或表 1-1) 或其他程序可知，π 的准确的前 15 位值应是 3.141592653589793 。但根据此处表中的计算结果，虽然在第 15 步已准确到小数点后 9 位 3.141592653 ，可是继续迭代时，不仅不能提高精度，反而得到更差的结果，这对于初入门的计算者来说是难以理解的。其实这个程序并没有什么错误，问题出在 **两个非常相近的数**

表 4-1　阿基米德方法估计 π 的迭代过程

次数	边数	下界	上界
0	6	3.000000000000000	3.464101615137754
1	12	3.105828541230249	3.215390309173472
2	24	3.132628613281238	3.159659942097500
3	48	3.139350203046867	3.146086215131434
4	96	3.141031950890510	3.142714599645368
⋮	⋮	⋮	⋮
12	24576	3.141592645034610	3.141592670702917
13	49152	3.141592651449375	3.141592657866452
14	98304	3.141592653075215	3.141592654679484
15	196608	3.141592653325344	3.141592653726411
16	393216	3.141592653325344	3.141592653425611
17	786432	3.141592653325344	3.141592653350410
18	1572864	3.141592655993386	3.141592655999653
19	3145728	3.141592645321216	3.141592645322782

相减 上。因为程序中有一个给 d 赋值的语句

```
d = 1 - SQR(1 - b * b)
```

当边数 n 增加时，半边长 b 缩小，从而 $SQR(1-b*b)$ 接近 1，用 1 减这样的数，结果的有效数位大量丢失，例如第 15 步时，半边长 $b \simeq 2^{-16} = 0.15 \times 10^{-4}$，这时 $SQR(1-b*b) \simeq 0.9999999999$，所以 1 减此数所得的 d 至少损失 10 位十进制有效数字。又因为双精度数存取时尾数不超过 16 位，所以 d 的有效数字不会多于 6 位，进而 b 及最终的 L 的有效数字也不会多于 6 位。至于表 4-1 中该步的实际结果多于 6 位有效数字的原因，可能是某些计算机设计中，运算器比存储器有更多的数位 (譬如双精度在运算器内是 80 位二进制，而存储器才 64 位)，为证实这一点，在程序中把 d 的赋值语句拆成如下两句：

```
d = SQR(1 - b * b)
d = 1 - d
```

其结果的有效数位将进一步减少。总之，在估计 π 的实际计算中，阿基米德方法不如刘徽割圆术精确和稳定。

通过这个例子读者已经看到，两个相近的数相减可能会导致计算结果的有效数位丢失，因此在选取算法时应尽可能避免这种现象。但是我们不希望因为这个例子给初入门的读者造成草木皆兵的惶恐，毕竟在一般的计算中，这样严重的有效数位丢失现象并不常见。本书的其他算 π 程序例子几乎都能算出准确到小数点后 15 位的结果。

[例 4.1.3] 用圆外切与内接多边形面积估计 π 值。

在上面例子的基础上，注意到两相似三角形的面积之比

是相应边长之比的平方，则本例的公式与程序都不难给出。下面给出的程序能准确算出小数点后 15 位的值，但为了节省篇幅，结果不再列出。

```
REM ex413.bas
DIM n AS LONG, b AS DOUBLE
DIM s AS DOUBLE, d AS DOUBLE
OPEN "ex413.res" FOR OUTPUT AS #2
n = 6
b = .5#
s = 1.5# * SQR(3#)
FOR k = 0 TO 26
  d = 1 - SQR(1 - b * b)
  PRINT k; n; s; s / (1 - d) ^ 2
  s = s + n * b * d
  b = SQR(d / 2)
  n = n + n
NEXT k
END
```

[例 4.1.4] 用 BBP 法估计 π 值。

BBP 法是由白莱、波尔温、布罗菲三人于 1997 年提出来的，并且本书的第 2 章 2.2.2 节已作了介绍。它是一个无穷和：

$$\pi = t_0 + t_1 + t_2 + \cdots + t_k + \cdots \tag{4.5}$$

和式中的通项是

$$t_k = \frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right) \tag{4.6}$$

这里给出的程序计算 10 步之后，获得准确到小数点后 15 位的结果 (见表 4-2)。

```
REM ex414.bas
DIM pi AS DOUBLE
DIM k8 AS DOUBLE, d16 AS DOUBLE
OPEN "ex414.res" FOR OUTPUT AS #2
pi = 0: d16 = 1: k8 = 0
FOR k = 0 TO 10
  pi = pi + (4 / (k8 + 1) _
    - 2 / (k8 + 4) - 1 / (k8 + 5) _
    - 1 / (k8 + 6)) / d16
  PRINT #2, k; pi
  d16 = d16 * 16
  k8 = k8 + 8
NEXT k
END
```

表 4-2　BBP 方法估计 π 的计算过程

k	π	k	π
0	3.133333333333333	6	3.141592653572881
1	3.141422466422466	7	3.141592653588973
2	3.141587390346582	8	3.141592653589752
3	3.141592457567436	9	3.141592653589791
4	3.141592645460336	10	3.141592653589793
5	3.141592653228088		

值得注意的是，本程序把 3 个短语句放在同一行 (第 5 行) 内。一般的 BASIC 约定，这时要在每个语句之间用一个冒号“:”来分隔。另外，当一个语句太长时，可以拆成若干行。QBasic 和 Quick BASIC 约定用一个下划线“_”放在行尾作为**换行符**。由于 Visual BASIC 在构造名字时把下划线视同字母，所以其换行符改为两个符号“ _”，即一个空

白符再加一个下划线。考虑到我们的程序要在 Visual BASIC 中运行，并且在前两种 BASIC 语句中，空白符在许多场合中是可以随意地添加的，因此我们的换行符都采用后一个约定(见程序的 7 ~ 9 行)。需要注意的是，如果这样的程序被前两种 BASIC 读入时，换行符将自动消失，能看见的仍然是一个未拆分的长行。换句话说，这两种 BASIC 设置换行符的主要目的是满足书面印刷的需要。当然，在 Windows 的 Visual BASIC 中，由于窗口数目很多，每个窗口的面积较小，设置换行符可以减少使用窗口横向滚动条来查看编码的机会，所以它不会改变用户的换行符。

[例 4.1.5] 用算术几何平均法 AGM 估计 π 值。

算术几何平均法 (AGM) 是萨拉明于 1976 年提出的一个具有二阶收敛速度的快速算法，其计算公式已在本书第 2 章的式 (2.11)~ 式 (2.15) 式中给出。本例给出的程序经 4 步计算就获得几乎准确到小数点后 14 位的结果 (见表 4-3，又 π 的 15 位准确值是 3.141592653589793)。其后迭代未能达到 15 位的原因，是算法中含有非常相近的两数相减 (见程序倒数第 7 行的赋值语句 $c2 = a2 - b2$ 及 [例 4.1.2] 的说明)。

```
REM ex415.bas
DIM a AS DOUBLE, b AS DOUBLE, s AS DOUBLE
DIM a2 AS DOUBLE, b2 AS DOUBLE, c2 AS DOUBLE
DIM pa AS DOUBLE, pb AS DOUBLE
OPEN "ex415.res" FOR OUTPUT AS #2
a = 1
b = 1 / SQR(2#)                    ' see (2.14)
s = 1
FOR k = 1 TO 5
```

```
    b2 = a * b
    a = (a + b) / 2                ' see (2.11)
    b = SQR(b2)                    ' see (2.12)
    a2 = a * a
    c2 = a2 - b2                   ' see (2.13)
    s = s - c2 * 2 ^ (k + 1)       ' see (2.15)
    pa = 4 * a2 / s                ' see (2.15)
    pb = 4 * b2 / s                ' see (2.15)
    PRINT #2, k; pb; pa
NEXT k
END
```

表 4-3　AGM 方法估计 π 的迭代过程

次数	下　界	上　界
1	3.093836321356053	3.187672642712108
2	3.141505000352043	3.141680293297652
3	3.141592653284143	3.141592653895450
4	3.141592653589808	3.141592653589808
5	3.141592653589832	3.141592653589833

[**例 4.1.6**]　用波尔温的 4 阶收敛公式估计 π 值。

根据本书第 2 章 2.2.2 节介绍，波尔温 (1986) 的 4 阶收敛公式是

$$a_0 = 6 - 4\sqrt{2}, \qquad y_0 = \sqrt{2} - 1 \tag{4.7}$$

$$\begin{cases} y_k = \dfrac{1 - (1 - y_{k-1}^4)^{1/4}}{1 + (1 - y_{k-1}^4)^{1/4}}, \\ a_k = a_{k-1}(1 + y_k)^4 - 2^{2k+1} y_k (1 + y_k + y_k^2) \end{cases} \tag{4.8}$$

式中 $a_k \to 1/\pi$ 。下面程序的前 2 步计算结果为

```
3.141592646213541    3.141592653589788
```

其中第 2 步结果已准确到小数点后 14 位。

```
REM ex416.bas
DIM a AS DOUBLE, y AS DOUBLE
DIM y4 AS DOUBLE, pi AS DOUBLE
OPEN "ex416.res" FOR OUTPUT AS #2
a = 6 - 4 * SQR(2#)
y = SQR(2#) - 1
FOR k = 1 TO 3
  y4 = SQR(SQR(1 - y ^ 4))
  y = (1 - y4) / (1 + y4)
  a = a*(1+y)^4 - 2^(2*k+1) *y*(1+y+y^2)
  pi = 1 / a
  PRINT #2, k; pi
NEXT k
END
```

4.2 可算万位 π 值的 BBP 分段算法程序

上节的 [例 4.1.4] 给出一个使用 BBP 公式计算 π 值的程序，但它只是一个非常初级的示意性程序，完全没有发挥 BBP 公式特有的优点。本节再次使用 BBP 公式，并充分利用它可以从任意一位算起的特点，给出分段算 π 程序。这是一个十分精巧的程序，它不需要多精度系统的支持，并且使用很少的存储单元，却可以轻松地算出小数点后几万位的 π 值，体现了国际上 π 值计算的最新成就。它的计算公式简单，核心程序长度不超过 100 行，值得认真研读。

4.2.1 算法技巧

1. 公式回顾

让我们首先回顾 BBP 公式，根据第 2 章 2.2.2 节 3.，BBP 公式是个无穷和式：

$$\pi = t_0 + t_1 + t_2 + \cdots + t_k + \cdots \tag{4.9}$$

和式中的通项是

$$t_k = \frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right) \tag{4.10}$$

现在让我们来计算 π 的小数点后第 $n+1$ 位开始的 m 位数 (这里所说的“位”，如不特别声明，都应理解为 16 进制的位)，这等价于计算 $16^n\pi$ 中的前 m 位小数。由式 (4.10)，$16^n\pi$ 的通项是

$$16^n t_k = 16^{n-k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right) \tag{4.11}$$

把这些项按下标分为两部分，第一部分对应于 $k \leqslant n$，其他为第二部分，其中第一部分各项的值可能含有整数，但是我们只需要小数部分，所以在求和的过程中略去整数只留小数；第二部分各项，它们都只含小数。如果所需的小数位数很少 (例如 $m = 8$)，那么完全可以用双精度 (它有不少于 13 位 16 进制的精度) 进行运算，而无需多精度数系统的支持。当 n 很大时，运算量主要集中在第一部分，而第二部分所需的计算总是很少的几项 (数目与 m 大体相当)。

2. 分段计算

如果在第 1 回计算中, 令 $n=0$, 则可得到 π 的小数点后的第 1 至 m 位 (16 进制); 接着在第 2 回计算中, 令 $n=m$, 又可得到 π 的小数点后的第 $m+1$ 至 $2m$ 位; 再接着在第 3 回计算中, 令 $n=2m$, 又可得到 π 的小数点后的第 $2m+1$ 至 $3m$ 位; 一般地在第 i 回计算中, 令 $n=(i-1)m$, 可得到 π 的小数点后的第 $(i-1)m+1$ 至 im 位; ……如此重复计算 1000 回, 若 $m=8$, 就可以得到 8000 位 16 进制数, 这相当于十进制的 9632 位 (16 进制的 1 位相当于十进制的 $\lg 16 \simeq 1.20412$ 位)。

分段算法无需多精度系统支持, 从而大大减少了程序设计的工作量 (见 4.2.2 节)。甚至在计算几千位 π 值时也有很不错的效率。但是分段重复计算的特点决定了它越往后计算越吃力。假如算头一个 m 位需要 1 个单位时间, 那么算前 $2m$ 位就需要 $1+2$ 个单位时间。一般地算前 lm 位就需要 $1+2+\cdots+l=l(l+1)/2$ 个单位的时间, 也就是说, 所需时间平方地增长。我们在 CPU 为 AMD-K6/233 的 PC 机上使用 QBasic(解释执行) 和 Quick BASIC(4.0 版, 以下简称 QB) 编译执行其后给出的程序时, 实测得到的作业时间如表 4-4 所示。从该表可以看出, 不论哪个 BASIC 系统, 作业时间确实是随位数迅速增加的, 因此想用 BBP 方法算几十万位是不现实的。在第 7 章我们将会看到一些更适合于计算很多位的程序, 例如在多精度系统支持下的波尔温 4 阶程序, 即使算出 3.27 万位 π 值也不超过 4 分钟。此外, 通过这个表我们也清楚地看到, 解释执行的 QBasic 的运算速度比编译执行的 QB

慢得多。以 5000 位为例，两者竟相差 43 倍。因此，万位以上的计算中，我们只用编译执行的 QB 。

表 4-4　BBP 分段算 π 程序的作业时间

十进位数	1000	2000	3000	4000	5000	
QBasic	2'11"	9'48"	23'26"	43'18"	69'57"	
QB 编译	3"	14"	32"	1'00"	1'37"	
十进位数	10000	20000	30000	40000	50000	78510
QB 编译	7'05"	31'32"	76'	142'	231'	631'

3. 基于“平方与乘法”的算法技巧

为了能对较大的 n 值进行计算，现把通项公式改写为

$$16^n t_k = \frac{4\times 16^{n-k}}{8k+1} - \frac{2\times 16^{n-k}}{8k+4} - \frac{16^{n-k}}{8k+5} - \frac{16^{n-k}}{8k+6} \tag{4.12}$$

在计算每个分式的小数部分时，如果 $n-k>0$ 且值很大 (例如 $n-k>1000$)，那么在不使用多精度系统的前提下，要获得准确的 m 位小数还需要一些计算技巧。下面就是具体的算法：首先把各分式的核心计算部分概括为如下形式：

$$16^{n-k}/p \tag{4.13}$$

如果 r 是 16^{n-k} 除以 p 所得的余数，那么所需的分数就是

$$r/p \tag{4.14}$$

这样一来，问题转化为求余数 r 。BASIC 语言本身含有求余数的运算符 MOD，人们可能想到用如下的语句来实现：

```
r = 16^(n-k) MOD p
```

但在实际计算中，因为 16^{n-k} 可能很大，并超出 BASIC 约定范围，所以不能直接使用上面的语句来计算 r 。下面是一个可行的计算步骤：

(1) 把 $n-k$ 写成二进制，并假设此二进制有 $i+1$ 位。例如 $n-k=25$(十进制) 的二进制形式是 11001，它有 5 位。

(2) 基于依次“平方”递推算出 16^{2^0}, 16^{2^1}, 16^{2^2}, $\cdots$, 16^{2^i} 分别除以 p 所得的余数与上述 $i+1$ 位二进制对应。为了清晰地描述这个计算过程，这里用 c MOD p 表示真正使用 BASIC 中的运算符 MOD 求余，而用我们的方法完成这个运算时则记为“ c 取模 p ”。另外注意到

$$16^{2^j} = 16^{2\times 2^{j-1}} = (16^{2^{j-1}})^2$$

则不难依次算出 16^{2^j}, $j=0,1,2,\cdots,i$:

$$16^{2^0} \text{ MOD } p = 16 \text{ MOD } p$$

$$16^{2^j} \text{ 取模 } p = (16^{2^{j-1}} \text{ 取模 } p)^2 \text{ MOD } p,\ j=1,2,\cdots,i \quad (4.15)$$

即上一个余数平方之后再求余得到下一个余数。

(3) 基于逐次“乘法”计算 $r=$ “ 16^{n-k} 取模 p ”。在写出 $n-k$ 的二进制，并算出相应的 $i+1$ 个“ 16^{2^j} 取模 p ”之后，就不难准确地计算 r 了。这时首先令 $r=1$ ，如果 $n-k$ 的二进制中与 2^j 对应的那位为 1 ，就把“ 16^{2^j} 取模 p ”与 r 相乘，并把乘积除以 p 所得的余数重新放回 r, 即

$$r = (r \times \text{“}16^{2^j} \text{ 取模 } p\text{”}) \text{ MOD } p \quad (4.16)$$

例如 $n-k$ 的二进制形式是 11001，这时与 2^0, 2^3, 2^4 对应的三位为 1，于是有

$$r = 1$$
$$r = (r \times \text{“}16^{2^0} \text{ 取模 } p\text{”}) \text{ MOD } p$$
$$r = (r \times \text{“}16^{2^3} \text{ 取模 } p\text{”}) \text{ MOD } p$$
$$r = (r \times \text{“}16^{2^4} \text{ 取模 } p\text{”}) \text{ MOD } p$$

由上面的计算公式 (4.15) 和式 (4.16) 易知，只要 p^2 不超过计算机的有效范围，r 的计算结果就不会有误差。以双精度运算为例，双精度的有效范围不小于十进制的 15 位精度 (严格说是 $2^{53}-1$ 以内)，从而 p 值在三五千万以内不会带来误差。值得注意的是，BASIC 语言实现 “d MOD p” 的方法不同于其他语言，它要先将 d 变为整型数，从而不允许 d 超过长整数的表示范围 $(2^{31}-1)$，即使这里的 d 与 p 都定义为双精度型也不例外。这样一来，我们甚至无法计算 7000 位 π 值。为使 d 的值确能达到双精度范围，最简单的方法是自己写一个语句代替 BASIC 中的 MOD，这时 d MOD p 可改为

```
d = d - p * INT(d / p)
```

4.2.2 一个完整的程序

下面是实现上述分段 BBP 算法的一个完整程序，除去主程序外，还包括三个子程序或函数。其中函数过程

```
FUNCTION dmod# (p AS DOUBLE)
```

具有上一小节介绍的 “平方与乘法” 技巧计算余数 r 的功能，它返回的结果是 r/p。本函数还使用了存放 $n-k$ 的二进制形式的公用数组 $nkj(32)$。

计算 $16^n\pi$ 中的前几位 16 进制小数的任务由子程序 *bbp* 完成：

```
SUB bbp (n AS LONG, s AS DOUBLE)
```

其中参数 s 用于存放结果。注意：程序中用变量 *lng* 代替上节算法公式中的 m，并在主程序中有语句 $lng = 8$，其含义是每回要算出 8 位 16 进制数。虽然子程序 *bbp* 中没有明显出现变量 *lng*，但在求和的循环中最多算至第 $n + 12$ 项，即

```
FOR k = 0 TO n + 12
```

其中的 12 是用来保证结果 s 能有 8 位以上的精度。这个子程序还给公用数组 *nkj*(32) 赋值并多次调用函数 *dmod#* 。

分段重复计算的任务由主程序完成。它将调用最后那个子程序程 *prtpi* 把得到的十六进制结果转换成十进制形式并输出到文件 *pibbp.res* 中去，而十六进制结果则由主程序直接保留在文件 *bbpblast* 内。主程序运行时屏幕上首先显示如下三行信息：

```
Iterate from NIT1 to NIT2,  output per NJMP
  NIT1=0 means to continue the last computation
Now, input NIT1, NIT2, NJMP
```

本来在显示中直接使用汉字对读者更为方便，但是汉字会占用宝贵的常规存储空间，从而使 QBasic 的解题规模缩小，所以本书的程序都用英文注释。但这不会影响不熟悉英文的读者对程序的理解，因为英文的含义都会在适当的地方被解释。上述三行信息的含义是让用户从键盘上输入三个参数：*nit*1, *nit*2, *njmp*, 并指出它们分别表示程序将从第 *nit*1 回开

始分段重复计算，直至第 $nit2$ 回结束，且每隔 $njmp$ 回印出此前算出的中间结果 (屏幕只显示其最后的 50 至 99 位十进制数)。例如：若键入的三个参数依次是 1, 104, 40，则屏幕上有如下输出：

```
pi=..7245870066 0631558817 4881520920 9628292540 9171536436
    7892590360 0113305305 4882046652 13841
       385 digits for repeating  40    time=22:08:59--22:09:00
pi=..4201995611 2129021960 8640344181 5981362977 4771309960
    5187072113 4999999837
       770 digits for repeating  80    time=22:08:59--22:09:01
pi=3.
      1005 digits for repeating 104    time=22:08:59--22:09:03
End of Pi job
```

其中“..”表示前头数字有删节。另外，当最终结果超过 500 位时，屏幕上只出现“pi=3.”，完整的结果可在文件 *pibbp.res* 中找到。输出中间结果对于及时了解计算进程是必要的，特别是需要长时间运算的问题。但是不宜频率过高以免耗费过多的作业时间。通常可令 $njmp = 104$ 或 52(即大约每 1000 位或 500 位十进制。

在绝大多数场合下，令 $nit1 = 1$，表示计算是从头开始的；但也可以令 $nit1 = 0$，表示用户以前曾经运行过此程序，现在要接着上一次结果往下计算 (这时用户不应破坏上次运行时产生的文件 *bbpblast*)。

1. 更简单的处理结果子程序

最后的那个子程序程 *prtpi* 用于把 16 进制结果转换成十进制并输出。由于考虑输出结果要有漂亮的形式 (每行 50 位，每 10 位留一空格，每 1000 位加序号)，而且允许对前头的一

些位进行删节；此外结果输出到文件的同时，还考虑是否送往屏幕等因素，致使该子程序较长 (超过整个程序的 1/3) 。其实，把 16 进制转换成十进制的核心语句不多，例如，当计算了 $nit2$ 回，16 进制结果已依次放入双精度数组 $e(nit2)$ (每个元素含有 8 位 16 进制小数)，则可改用如下只有 16 行的简单子程序输出十进制结果。此程序在每次 FOR j 循环中，di 都得到 5 位十进制结果。

```
SUB prtt (e() AS DOUBLE, nit2)
d16lng# = 16# ^ 8
n5 = (INT(1.20412 * (8 * nit2)) + 4) \ 5
FOR j = 1 TO n5
  e(nit2) = 100000# * e(nit2)
  FOR i = nit2 TO 2 STEP -1
    di = INT(e(i))
    e(i) = e(i) - di
    e(i - 1) = 100000# * e(i - 1) + di / d16lng#
  NEXT i
  di = INT(e(1))
  e(1) = e(1) - di
  PRINT #2, RIGHT$(STR$(di + 100000), 5);
  IF j MOD 10 = 0 THEN PRINT #2, " "
NEXT j
END SUB
```

2. 完整的 BBP 分段算法程序列表

下列程序可在 QBasic 或 Quick BASIC 上运行。在后者编译执行时，曾计算了 8150 步，并得到正确的 78510 位 π 值。作业时间详见前面的表 4-4 。

```
REM *** *** Program PIBBP ****************************
DIM nit1 AS LONG, nit2 AS LONG, njmp AS LONG
DIM lng AS INTEGER, mlng AS LONG, m AS LONG
DIM s AS DOUBLE, d16lng AS DOUBLE, wk AS DOUBLE
DIM SHARED nkj(32) AS INTEGER

OPEN "pibbp.res" FOR OUTPUT AS #2
PRINT "Iterate from NIT1 to NIT2,  output per NJMP"
PRINT "   NIT1=0 means to continue the last computation"
INPUT "Now, input NIT1, NIT2, NJMP"; nit1, nit2, njmp
t0$ = TIME$
lng = 8
d16lng = 16# ^ lng

REDIM ss(nit2) AS DOUBLE
IF nit1 = 0 THEN
  OPEN "bbpblast" FOR INPUT AS #1
  nit1 = 1
  DO
    INPUT #1, s
    ss(nit1) = s / d16lng
    nit1 = nit1 + 1
  LOOP UNTIL EOF(1)
  CLOSE #1
  OPEN "bbpblast" FOR APPEND AS #1
ELSE
  OPEN "bbpblast" FOR OUTPUT AS #1
END IF

IF nit1 > nit2 THEN nit2 = nit1
FOR m = nit1 TO nit2
  mlng = lng * (m - 1)
```

```
  CALL bbp(mlng, s)
  wk = INT(s * d16lng)
  ss(m) = wk / d16lng
  PRINT #1, wk
  IF (m MOD njmp) = 0 THEN CALL _
     prtpi(ss(), -m, lng, s, t0$)
NEXT m
CALL prtpi(ss(), nit2, lng, s, t0$)
PRINT "End of Pi job"
END

REM *** *** BBP algorithm ****************************
SUB bbp (n AS LONG, s AS DOUBLE)
DIM p AS DOUBLE, p2 AS DOUBLE, p3 AS DOUBLE
DIM p4 AS DOUBLE, p16 AS DOUBLE, tk AS DOUBLE
DIM k AS LONG, nk AS LONG, j AS INTEGER
p = 1#
p16 = 1#
s = 0#
FOR k = 0 TO n + 12
  p2 = p + 3#
  p3 = p + 4#
  p4 = p + 5#
  IF k <= n THEN
    IF k = 0 THEN
      s = 2# / 15#
    ELSE
      nk = n - k
      FOR j = 1 TO 32
        nkj(j) = nk MOD 2
        nk = nk \ 2
        IF nk = 0 THEN EXIT FOR
```

```
      NEXT j
      nkj(0) = j
      s = s + (4# * dmod#(p) - 2# * dmod#(p2) _
          - dmod#(p3) - dmod#(p4))
      DO WHILE s > 1#
        s = s - 1#
      LOOP
      DO WHILE s < 0#
        s = s + 1#
      LOOP
    END IF
  ELSE
    p16 = p16 / 16#
    tk = p16 * (4# / p - 2# / p2 - 1# / p3 - 1# / p4)
    s = s + tk
    IF s > 1# THEN s = s - 1#
    IF tk < 1D-17 THEN EXIT FOR
  END IF
  p = p + 8#
NEXT k
END SUB

REM *** *** 16^nk mod p ******************************
FUNCTION dmod# (p AS DOUBLE)
DIM b16 AS DOUBLE, r AS DOUBLE, j AS INTEGER
r = 1#
b16 = 16#
FOR j = 1 TO nkj(0)
  IF nkj(j) = 1 THEN
    r = b16 * r
    r = r - p * INT(r / p)
  END IF
  IF j = nkj(0) THEN EXIT FOR
```

```
  b16 = b16 * b16
  b16 = b16 - p * INT(b16 / p)
NEXT j
dmod# = r / p
END FUNCTION

REM *** *** Convert from 16 to 10 and print ***********
SUB prtpi (d() AS DOUBLE, mm AS LONG, lng AS INTEGER, _
    s AS DOUBLE, t0$)
DIM m AS LONG, n AS LONG, n1 AS LONG, n5 AS LONG
DIM i AS LONG, i1 AS LONG, j AS LONG
DIM d16lng AS DOUBLE, di AS DOUBLE
DIM c0 AS STRING * 5, ch(3) AS STRING * 5
m = ABS(mm)
REDIM e(m) AS DOUBLE

n = INT(1.204119983# * (lng * m))
n5 = (n + 4) \ 5
n = 5 * n5
ch(1) = "pi=3.": ch(2) = "pi=..": ch(3) = "      "
IF mm < 0 THEN
  n1 = n - 50
  IF n1 < 0 THEN n1 = 0
  n1 = (n1 \ 50) * 10 + 1
ELSE
  n1 = 1
END IF
i = n1
IF i > 2 THEN i = 2
PRINT #2, ch(i); : PRINT ch(i);

c0 = "00000"
FOR i = 1 TO m - 1
```

```
  e(i) = d(i)
NEXT i
e(m) = s
d16lng = 16# ^ lng
FOR j = 1 TO n5
  e(m) = 100000# * e(m)
  IF m > 1 THEN
    i = m
    DO WHILE i > 1
      di = INT(e(i))
      e(i) = e(i) - di
      e(i - 1) = 100000# * e(i - 1) + di / d16lng
      i = i - 1
    LOOP
  END IF
  i1 = INT(e(1))
  e(1) = e(1) - i1
  IF j >= n1 THEN
    c6$ = STR$(i1)
    ka = LEN(c6$)
    c$ = LEFT$(c0, 6 - ka) + RIGHT$(c6$, ka - 1)
    PRINT #2, c$; : IF mm < 53 THEN PRINT c$;
    IF (j MOD 2) = 0 THEN
      IF (j MOD 10) = 0 THEN
        PRINT #2, " ": IF mm < 53 THEN PRINT " "
        IF (j MOD 200) = 0 THEN
          PRINT #2, SPACE$(16); "^^"; 5 * j; "digits"
          IF mm < 53 THEN PRINT SPACE$(16); _
                                "^^"; 5 * j; "digits"
        END IF
        PRINT #2, ch(3); : IF mm < 53 THEN PRINT ch(3);
      ELSE
```

```
        PRINT #2, " "; : IF mm < 53 THEN PRINT " ";
      END IF
    END IF
  END IF
NEXT j
PRINT #2, " ": PRINT " "
PRINT #2, SPACE$(8); n; "digits for repeating "; m;
PRINT #2, "   time="; t0$; "--"; TIME$
PRINT SPACE$(8); n; "digits for repeating "; m;
PRINT "   time="; t0$; "--"; TIME$
END SUB
```

第 5 章　计算机上多精度数的运算方法

超高精度的 π 值计算是建立在两个基础之上的，其一是优异的数学公式，其二是计算机上实现超长 (多精度) 数运算的方法。对于前者，第 2 章已有详细的介绍，后者的讨论正是本章的任务。

5.1　单精度、双精度与多精度

众所周知，计算机上进行数值运算时，其精度总是有限的。对于包括 BASIC 在内的大多数语言来说，通常使用 2 至 8 个字节存放一个数。例如

类　型	字节	有效范围或位数 (精度)
整　型	2	−32768~32767
长整型	4	−2147483648~2147483647
单精度实型	4	7 位十进制
双精度实型	8	15 位十进制

虽然某些高级语言 (如某些专业使用的 FORTRAN) 还允许“四精度”变量，这时的有效位数又比双精度增加一倍以上。但是计算机上运算的一般变量的有效位数总是有限的，为实现超长位数的 π 值运算，我们需要有“多精度” (也称“任意精度”) 数的表示与运算方法。

5.2 计算机上多精度数的表示方法

5.2.1 科学记数法

科学记数法是用指数形式表示一个实数的方法。例如，-45.309 可表示为

$$-45.309\times 10^0,\quad -4.5309\times 10^1,\quad -0.45309\times 10^2$$

其标准型为

$$a=\pm 0.a_1a_2\cdots a_t\times 10^m \tag{5.1}$$

此处 $\pm$ 是数的符号，例中是“$-$”；$a_1,a_2,\cdots,a_t$ 是 t 位尾数，每个 a_i 都是 0 到 9 的整数 (通常 $a_1\neq 0$)，例中 $t=5$，a_i 依次是 4, 5, 3, 0, 9；m 称为阶 (或指数)，可正可负，例中是 $m=2$。与标准型等价的算术表达式为

$$a=\pm 10^m\Big(\frac{a_1}{10^1}+\frac{a_2}{10^2}+\cdots+\frac{a_t}{10^t}\Big) \tag{5.2}$$

其中 $0\leqslant a_i\leqslant(10-1)$, $(a_1\neq 0)$。

5.2.2 多精度数的表示方法

上面的表达式 (5.2) 可以一般化为“p 进制”，这时只需把 10 改为 p 便得

$$a=\pm p^m\Big(\frac{a_1}{p^1}+\frac{a_2}{p^2}+\cdots+\frac{a_t}{p^t}\Big) \tag{5.3}$$

其中 $0\leqslant a_i\leqslant(p-1)$, $(a_1\neq 0)$。当某位的值超过 p 时，便是逢 p 进 1 地向上进位。当 $p=10$ 时，就是我们习惯的十进

制，当 $p=2$ 时，便是二进制。特别令 $p=10000$，就可以得到“万进制”。万进制数中的一位相当于十进制中的四位。例如有如下的一个十进制数：

$$a=-123456789.012345$$

因为

$$a=-10000^3\Big(\frac{1}{10000}+\frac{2345}{10000^2}+\frac{6789}{10000^3}+\frac{0123}{10000^4}+\frac{4500}{10000^5}\Big)$$

所以它可以被看作是一个含有 5 位尾数的万进制数，该数符号为“ $-$ ”，阶数为 $m=3$。在计算机的程序设计中，一个 t 位的万进制数，可用一个含有 $t+2$ 个元素的数组 a 来对应(见表 5-1)。我们约定 $a(1)$ 中的数符只可能取如下三个值中的一个：$+1$, -1, 0，它们分别与数 $a>0$, $a<0$, $a=0$ 相对应。显而易见，**数的精度随数组元素的增加而提高**，用户可以根据自己的意愿任意指定所需的精度 (即指定数组元素的数目)。这便是任意精度数的含义。设计任意精度数的目的，是要获得比计算机上约定的常规变量有更多的精度，所以我们有时也称之为多精度数或长尾数。若数的精度实在太高，我们有时也称这类数具有超高精度或超长尾数。

表 5-1　p 进制数各位与数组元素间的对应关系

数组元素	$a(1)$	$a(2)$	$a(3)$	$a(4)$	$a(5)$	$\cdots$	$a(t+2)$
存放内容	数符 $\pm$	阶 m	a_1	a_2	a_3	$\cdots$	a_t
例中内容	-1	3	1	2345	6789	$\cdots$	4500

在其后的程序设计中，我们主要使用上述万进制的数组表示方法。为了让读者有更深刻的印象，在表 5-2 中再给出这

种表示方法的几个具体例子。值得注意的是，第 1 位尾数 a_1 (对应着数组元素 $a(3)$) 的值常常小于 1000 (即左边有 0)，在计算其对应的十进制阶时要特别留意。例如表中的第一个数，万进制中的 1 阶，本来对应着十进制的 $1\times4=4$ 阶，但因 a_1 左边出现 3 个 0，所以最终的十进制的阶是 $1\times4-3=1$。同理，第二个数最终的十进制阶是 $-4\times4-2=-18$。

表 5-2　数组表示的万进制数及其对应的十进制数实例

整型数组						对应的十进制数
数组元素	$a(1)$	$a(2)$	$a(3)$	$a(4)$	$a(5)$	
存放内容	1	1	0003	1415	9265	0.314159265×10^{1}
存放内容	1	−4	0056	0123	4560	$0.560123456\times10^{-18}$
存放内容	−1	0	1111	2222	8500	-0.111222285×10^{0}
存放内容	−1	−2	0789	0123	4500	$-0.789012345\times10^{-9}$
存放内容	0	0	0000	0000	0000	0

5.3　多精度数的四则运算方法

本节讨论如何在计算机上实现多精度数的四则运算。其原则适用于上节约定的任一种 p 进制数的数组表示方法。读者将会发现，这里介绍的加法、减法和乘法运算实质上只是小学生的竖式手算方法的翻版，而除法则完全不同。

5.3.1　标准型加法运算

我们从一个简单的数值例子说起。算法是模仿十进制手算过程得到的，包括对准小数点，对应的位相加，处理进位和

四舍五入等步骤。设有 a, b 两个万进制数

$$a = 10000^3\left(\frac{123}{10000^1} + \frac{4567}{10000^2} + \frac{8901}{10000^3} + \frac{2345}{10000^4}\right)$$

$$b = 10000^1\left(\frac{7890}{10000^1} + \frac{1234}{10000^2} + \frac{5678}{10000^3} + \frac{9012}{10000^4}\right)$$

今求和 $c = a + b$，要求结果亦有 4 位万进制精度。计算时，对准小数点可理解为对准值大的阶（简称为“高阶对位”）。这里就是让 b 的阶等于 a 的阶，这时 b 的尾数右移，即

$$b = 10000^3\left(\frac{0}{10000^1} + \frac{0}{10000^2} + \frac{7890}{10000^3} + \frac{1234}{10000^4} + \frac{5678}{10000^5}\right)$$

这里为保证结果精度而多保留了一个附加位。其后是尾数的对应位相加得

$$a = 10000^3\left(\frac{123}{10000^1} + \frac{4567}{10000^2} + \frac{16791}{10000^3} + \frac{3579}{10000^4} + \frac{5678}{10000^5}\right)$$

处理进位与舍入后得到的结果是

$$a = 10000^3\left(\frac{123}{10000^1} + \frac{4568}{10000^2} + \frac{6791}{10000^3} + \frac{3580}{10000^4}\right)$$

下面我们用数学公式把计算步骤作归纳和补充。设有两个 p 进制数

$$a = \pm p^m\left(\frac{a_1}{p^1} + \frac{a_2}{p^2} + \cdots + \frac{a_t}{p^t}\right)$$

$$b = \pm p^n\left(\frac{b_1}{p^1} + \frac{b_2}{p^2} + \cdots + \frac{b_t}{p^t}\right)$$

它们的尾数都有 t 位。要作的计算是求和，即 $c=a+b$，并要求 c 亦保留 t 位精度。为方便起见，不妨假设 a, b 都是正数，且 $a \geqslant b$。我们把这种类型的加法称为标准型加法，至于更一般情形的加法运算，将留在 5.3.3 节中讨论。下面是计算步骤：

1. *高阶对位* 这时以大的数的阶为准，把小的数重新表示，使其阶也与大数的阶相同。由 $a \geqslant b$ 的假设，实际上就是把 b 的尾数右移 $m-n$ 位。此外在对阶过程中增添一位附加位以保证结果也有 t 位有效数字。对阶后的结果记为

$$b' = \pm p^m \left(\frac{b'_1}{p^1} + \frac{b'_2}{p^2} + \cdots + \frac{b'_t}{p^t} + \frac{b'_{t+1}}{p^{t+1}} \right)$$

式中

$$b'_i = \begin{cases} 0, & \text{当 } 1 \leqslant i \leqslant m-n \\ b_{i-m+n}, & \text{当 } m-n < i \leqslant t+1 \end{cases}$$

2. *尾数相加* 就是对应的位相加，记为

$$c'_i = a_i + b'_i, \quad i = 1, 2, \cdots, t+1$$

3. *进位* 从 c' 的第 $t+1$ 位开始，自右至左，逢 p 进 1。进位后的结果依次记为

$$c''_0 c''_1 c''_2 \cdots c''_{t+1}$$

此处的 c''_0 是 c'_1 向前进位所得。如果 c'_1 没有向前进位，那么 $c''_0 = 0$。

4. *右规* 如果 $c''_0 = 0$，此时无需右规，否则把

$$c''_0 c''_1 c''_2 \cdots c''_{t+1}$$

依次向右移一位，并且阶 m 加 1。把经历本步之后 (不论是否真正需要右规) 所得的阶记为 m'，各位记为 c_i^*，则

$$\begin{cases} m'=m, & c_i^*=c_i'', \quad (i=1,2,\cdots,t+1) & \text{当} c_0''=0 \\ m'=m+1, & c_i^*=c_{i-1}'', \ (i=1,2,\cdots,t+1) & \text{当} c_0''\neq 0 \end{cases}$$

5. 舍入　若 $c_{t+1}^*<p/2$，结果无需舍入，最终有

$$c=\pm p^{m'}\Big(\frac{c_1}{p^1}+\frac{c_2}{p^2}+\cdots+\frac{c_t}{p^t}\Big)=\pm p^{m'}\Big(\frac{c_1^*}{p^1}+\frac{c_2^*}{p^2}+\cdots+\frac{c_t^*}{p^t}\Big)$$

若 $c_{t+1}^*\geqslant p/2$，则在 c_t^* 上加 1，并去掉 c_{t+1}^*。如果这时的 c_t^* 又产生进位，那么还需要“规格化”处理，即再次的进位处理，甚至还可能包括再次的右规。以十进制为例，如果舍入前的结果是 0.99996×10^2，现在舍入为只含 4 位有效数字，因第 5 位产生进位，需作规格化处理为 0.1000×10^3。

注：根据上节的多精度数的数组表示法，数 a 尾数的第 i 位对应着数组元素 $a(i+2)$，数的阶对应着 $a(2)$，而数符与 $a(1)$ 对应。

5.3.2　标准型减法运算

计算差 $c=a-b$，仍假设 a, b 都是正数，且 $a\geqslant b$。我们把这种类型的减法称为标准型减法，至于更一般情形的减法运算，将留在 5.3.3 节中讨论。下面是计算步骤：

1. 高阶对位　与加法运算类似，但在对阶过程中增添两位附加位以保证结果也有 t 位有效数字。对阶后的结果记为

$$b'=\pm p^m\Big(\frac{b_1'}{p^1}+\frac{b_2'}{p^2}+\cdots+\frac{b_{t+1}'}{p^{t+1}}+\frac{b_{t+2}'}{p^{t+2}}\Big)$$

其前 $t+1$ 位与加法同，而 b'_{t+2} 就是原来 b 中移到第 $t+2$ 位的值再加上可能有的舍入补偿值 Δ。通常 $\Delta=1$，仅当 b 中该位之后的所有数位的值都等于零，才令 $\Delta=0$ (与通常的四舍五入有差别)。

2. *尾数相减* 就是对应的位相减，记为

$$c'_i=a_i-b'_i,\quad i=1,2,\cdots,t+2$$

3. *借位* 从 c' 的第 $t+2$ 位开始，自右至左，若某位的值为负，则向其左近邻位借 1，之后本位加 p。借位后的结果依次记为

$$c''_1c''_2\cdots c''_{t+2}$$

由于假设 $a\geqslant b$，此处的 c''_1 不可能为负。

4. *左规* 如果 $c''_1\neq 0$，此时无需左规，否则检查

$$c''_1c''_2\cdots c''_{t+2}$$

若其最左边有 l 个连续为零的位，则把它们依次向左移 l 位，右方空出的 l 位用 0 来填补，并且阶 m 减 l。把经历本步之后 (不论是否真正需要左规) 的阶记为 m'，各位记为 c^*_i。

5. *舍入* 同加法运算，最终保留 t 位尾数。

5.3.3 一般情形的加、减法运算

前两小节我们讨论了多精度数的加、减法运算

$$c=a+b,\quad c=a-b$$

但都限定它们具有标准型，即被运算的两数同为正，且第一个数大于第二个数

$$a>0,\ b>0,\quad 且\quad a\geqslant b$$

本节讨论更一般的加、减法运算。实际上我们只是说明：一般情形的加、减法运算都可以简化为标准型的加、减法运算。

1. 首先讨论 **一般加法**。当 a, b 两数中有一个为 0 的时候，c 就等于另一个，这时加法运算简化为数的传送 (拷贝)。所以只需讨论 $a \neq 0$, $b \neq 0$ 的情形。

(1) 先考虑 $|a| > |b|$ 的情形。这时若 a, b 同号，则作

$$|a| + |b|, \quad \text{(标准型加法)}$$

所得结果冠以 a 的符号便得 c。若 a, b 异号，则作

$$|a| - |b|, \quad \text{(标准型减法)}$$

所得结果冠以 a 的符号便得 c。

(2) 现在考虑 $|a| \leqslant |b|$ 的情形。这时若 a, b 同号，则作

$$|b| + |a|, \quad \text{(标准型加法)}$$

所得结果冠以 b 的符号便得 c。若 a, b 异号，则作

$$|b| - |a|, \quad \text{(标准型减法)}$$

若所得结果的尾数为 0(因标准型减法的结果已经“左规”，所以只需检查第 1 位是否为 0)，则 $c = 0$，否则所得结果冠以 b 的符号便得 c。

2. 对于 **一般的减法** $a - b$，可简单地改变 b 的符号 (把 b 变为 $-b$)，然后化为一般情形的加法运算 $a + (-b)$。

综上所述，**一般情形的加、减法运算都可以简化为标准型的加、减法运算。**

5.3.4 乘法运算

两个多精度数相乘 $c=a\times b$ 的算法也是模仿十进制手算过程得到的。包括依次用 b 的每一位乘 a，然后相加、进位、四舍五入和阶的处理等步骤。

首先看一个简单的数值例子。短的位数容易看清楚算法的实质，所以本例把万进制数暂时改为百进制数。令 a, b 为如下两个百进制数 (各含 3 位)：

$$a=-100^2\Big(\frac{1}{100^1}+\frac{23}{100^2}+\frac{40}{100^3}\Big)$$

$$b=100^1\Big(\frac{30}{100^1}+\frac{10}{100^2}+\frac{05}{100^3}\Big)$$

为得到乘积的尾数，现仿照十进制竖式手算格式进行

$$\begin{array}{rrrrrr}
 & & & 1 & 23 & 40 \\
 & & \times & 30 & 10 & 05 \\
\hline
 & & & 5 & 115 & 200 \\
 & & 10 & 230 & 400 & \\
+) & 30 & 690 & 1200 & & \\
\hline
 & 37 & 14 & 40 & 17 & 00
\end{array}$$

左规、舍入 (仍取 3 位百进制) 和阶处理后得如下结果

$$c=-100^2\Big(\frac{37}{100^1}+\frac{14}{100^2}+\frac{40}{100^3}\Big)$$

阶的处理原则是：当乘积的位数是相乘两数位数之和时，积的阶也取相乘两数阶之和；但当乘积的位数小于此和数 (或说积的左面第 1 位为 0)，这时积的阶应减 1 。

下面把计算步骤作归纳和补充。设有两个 p 进制数

$$a = \pm p^m \left(\frac{a_1}{p^1} + \frac{a_2}{p^2} + \cdots + \frac{a_t}{p^t} \right)$$

$$b = \pm p^n \left(\frac{b_1}{p^1} + \frac{b_2}{p^2} + \cdots + \frac{b_t}{p^t} \right)$$

它们的尾数都有 t 位。现在作乘法运算 $c = a \times b$，并要求 c 保留 t 位精度。

1. *尾数相乘* 乘积尾数的 $2t$ 位记为

$$c'_1 c'_2 \cdots c'_{2t}$$

计算开始时，它们中的每一位都取 0 值。其后从 b 的各位 $b_1, b_2, \cdots, b_t$ 中依次 (逆序或正序均可) 取出一位 (记为 b_j)，把它与 a 中各位 $a_1, a_2, \cdots, a_t$ 相乘得到的 t 个积按顺序加到 $c'_{j+1} c'_{j+2} \cdots c'_{j+t}$ 的各位中。例如 b_j 首先取 b_t，然后是 b_{t-1}，直至做完 b_1 后，尾数相乘便完成。显然，在进位之前，$c'_1 = 0$。

2. *进位* 从 c' 的第 $2t$ 位开始，自右至左，逢 p 进 1。进位后的尾数依次记为

$$c''_1 c''_2 \cdots c''_{2t}$$

3. *左规* 若 $c''_1 \neq 0$，则无需左规，结果的阶为 $m+n$，否则把

$$c''_1 c''_2 \cdots c''_{2t}$$

依次向左移 1 位，且阶为 $m+n-1$。若把经历本步之后 (不论是否真正需要左规) 的阶记为 m'，各位记为 c^*_i，则

$$\begin{cases} m' = m+n, & c^*_i = c''_i, \quad (i = 1, 2, \cdots, t+1) & \text{当} c''_1 \neq 0 \\ m' = m+n-1, & c^*_i = c''_{i+1}, \ (i = 1, 2, \cdots, t+1) & \text{当} c''_1 = 0 \end{cases}$$

4. 舍入　同加法运算，最终保留 t 位尾数。

5.3.5 除法运算

这里简单介绍两个多精度数的除法运算 $c = a/b$。易知

$$c = \frac{a}{b} = a \times \left(\frac{1}{b}\right)$$

也就是说，如能得到 b 的倒数 $1/b$，其后只需再作一个乘法便得商 c。至于一个多精度数的倒数运算方法，将留在 5.4 节中讨论。

5.3.6 关于乘、除法运算的计算工作量与快速算法

容易明白，当尾数位数 t 很长时，两个多精度数相乘的计算工作量主要集中在尾数相乘上。因为这时起码需要位数之间的 t^2 个乘法和 t^2 个加法。虽然，进位等运算也需要几 t 个位数间的四则运算，但当 t 很大时，几个 t 与 t^2 相比几乎可以被忽略。今假设要计算十进制 100 万位的 π 值。如果使用万进制，即每个数组元素存放 4 个十进制位，那么万进制尾数有 $t = 25$ 万位，而 $2t^2 = 1250$ 亿。这是相当庞大的计算量，即使是用每秒 1 亿次的机器来计算，也需要 20 多分钟才能完成两个超高精度数的乘法运算。根据十进制手算的经验，**除法要比乘法更复杂**，从而需要更多的计算时间。幸好在 60 至 70 年代，世界上出现了能快速完成这种长尾数相乘的计算方法。虽然此快速算法涉及 FFT 等很复杂的数学理论，本章不打算进一步讨论 (有兴趣的读者可在本书的最后一章看到梗概的介绍)，但是本书提供了一个实现这种快速乘法的易用程

序。通过这个程序，乘法中 t^2 量级的计算量下降到 $t \times \log_2(t)$ 量级。当 $t = 250000$ 时，对数 $\log_2(t) < 18$，从而工作量只是原来的 $1/13888$。正是这个原因，我们的除法运算方案放弃了继续模仿十进制手算的思路，改用倒数与乘法来实现。

5.4 倒数运算与开方

本节介绍的算法均与传统手算方法不同。它们都基于计算数学中的牛顿迭代法。虽然此法涉及较多的高等数学概念，但我们介绍时将尽可能使用通俗的方法，务使只有中学数学水平的读者可以接受。为避免过多的数学概念模糊了我们的目标，我们将尽快地开列计算倒数 $1/a$ 与开方 $\sqrt{a}$ 的公式。此前尚未接触过牛顿迭代法及与之有关的导数概念的读者，如果对这些内容有兴趣，请看本章末尾 (5.4.5 节) 我们专门为此准备的补充介绍。

5.4.1 牛顿迭代法

1. 解方程求数值解

牛顿迭代法是计算数学中给出方程的数值解的方法。譬如有如下两个方程

$$\frac{1}{x} - a = 0 \tag{5.4}$$

$$x^2 - a = 0 \tag{5.5}$$

式中 a 是已知常数，x 是未知数。求解方程就是要求出 x，使方程式成立。上述两方程分别称为倒数方程与平方根方程，

这是因为它们的解分别是

$$x = \frac{1}{a} \tag{5.6}$$

$$x = \sqrt{a} \tag{5.7}$$

所谓数值解，是指方程中的常数已经给出了具体的数值，要求把解的数值也真正算出来，而不是只给出解所满足的符号表达式。例如，式 (5.7) 只是方程 (5.5) 的用符号表达的解，而不是数值解。当方程 (5.5) 中的常数给出了具体的数值之后，譬如 $a = 3$，这时根据方程算出的具体值 $x = 1.73205\cdots$ 才称为方程的数值解。如上所述，牛顿迭代法是用来具体算出方程的数值解的方法，而不是推演公式的方法。

2. 函数与导数

通常把要解的方程形式地记为

$$f(x) = 0 \tag{5.8}$$

这里的 $f(x)$ 是一个泛指的记号，称为关于变量 x 的函数。它既可以是式 (5.4) 中的 $1/x - a$ 或式 (5.5) 中的 $x^2 - a$，更可以有其他的形式，如 $\sin x$, $\log x$ 等。当函数是 $x^3 - a$ 时就得到立方根方程。另外，大多数常用的函数 $f(x)$ 都有另一个称之为导数的函数 $f'(x)$ 与之对应。例如，倒数方程与平方根方程中的函数和对应的导数分别是

$$f(x) = \frac{1}{x} - a \quad \text{对应导数是} \quad f'(x) = -\frac{1}{x^2} \tag{5.9}$$

$$f(x) = x^2 - a \quad \text{对应导数是} \quad f'(x) = 2x \tag{5.10}$$

$$f(x)=\frac{1}{x^2}-a \quad \text{对应导数是} \quad f'(x)=-\frac{1}{x^3} \tag{5.11}$$

函数 $f(x)$ 在点 $x=x_0$ 处的导数，记为 $f'(x_0)$。实际上它就是函数在该点处的变化率。把 $y=f(x)$ 看作是曲线时，$f'(x_0)$ 就是曲线在该点处的切线斜率。在物理学中，若用 $s=f(t)$ 描述运动物体的距离 s 与时间 t 关系，那么 $f'(t_0)$ 就是物体在 $t=t_0$ 时刻的瞬时速度。要想进一步了解导数概念的读者，请看本章末尾 5.4.5 节的补充介绍。

3. 牛顿迭代公式

用于寻找方程 $f(x)=0$ 数值解的牛顿迭代公式是

$$x_{k+1}=x_k-\frac{f(x_k)}{f'(x_k)} \qquad k=0,1,2,3,\cdots \tag{5.12}$$

迭代的含义是，先指定一个迭代初位 x_0(后面还将讨论获得初值的方法)，根据式 (5.12)，有了 x_0 就可以算出 x_1，这时

$$x_1=x_0-\frac{f(x_0)}{f'(x_0)}$$

继而从 x_1 算出 x_2

$$x_2=x_1-\frac{f(x_1)}{f'(x_1)}$$

再从 x_2 算出 x_3

$$x_3=x_2-\frac{f(x_2)}{f'(x_2)}$$

如法泡制就可得到一个序列 $x_0,x_1,x_2,x_3,\cdots$，通常这个序列

在迭代过程中逐渐趋向一个稳定值 (记为 x^*)，它就是方程 $f(x)=0$ 的 (数值) 解。

牛顿迭代法中的一个关键问题是迭代初值的选定。我们可以使用任何一种方法给出初值。当初值较接近真解时，迭代过程就容易收敛到所需的解。由于牛顿迭代是二阶收敛的，迭代过程中，下一步的有效位数大体上是前一步的两倍，也就是说，其收敛速度相当快。但是，当初值距离真解太远时，有可能得不到稳定的解 (这时称迭代过程发散，详见后面的数值例子)。值得庆幸的是，我们只是把牛顿迭代法用于计算超高精度的倒数与开方值，而它们的初值完全可以通过计算机内常规精度运算获得。例如 BASIC 中的平方根函数 SQR(a)，在双精度场合已超过 15 位十进制的精度，是个好得不能再好的初值了。

要想进一步了解牛顿迭代式来源的读者，也请看本章末尾的补充介绍。

5.4.2 利用牛顿迭代法作倒数运算

求 a 的倒数 $1/a$ 等价于解方程

$$\frac{1}{x}-a=0$$

由式 (5.9)，函数和导数分别是

$$f(x)=\frac{1}{x}-a, \qquad f'(x)=-\frac{1}{x^2}$$

由式 (5.12)，这时的牛顿迭代公式是

$$x_{k+1} = x_k - \Big(\frac{1}{x_k} - a\Big)\Big/\Big(-\frac{1}{x_k^2}\Big) = x_k + x_k - ax_k^2$$

经整理后得

$$x_{k+1} = x_k(2 - ax_k) \quad k = 0, 1, 2, \cdots \tag{5.13}$$

[例 5.4.1] 求 7 的倒数 1/7 。

众所周知， 1/7 是个无限循环小数，它的循环节是六位数字 142857 。现在让我们来看看牛顿迭代公式是怎样算出结果的。

给出初值 $x_0 = 0.1$

其后各步 $x_1 = 0.1 \times (2 - 7 \times 0.1) = 0.13$

$x_2 = 0.13 \times (2 - 7 \times 0.13) = 0.1417$

$x_3 = 0.1417 \times (2 - 7 \times 0.1417) = 0.14284777$

$x_4 = \cdots = 0.1428571422421897$

$x_5 = \cdots = 0.1428571428571428$

上述结果是用一个简单的 BASIC 程序算出的。它使用双精度运算 (约有 15 位左右的十进制精度) ，长尾数可能有舍入误差，所以上列 x_4 尾数中的最后几位可能与手算的精确结果有差异。根据上面的计算结果，我们确实可以看到，每步所得的有效数位翻番地增长。

在计算中，若把初值改为 $x_0 = 0.2$ ，经 6 步运算也能得到小数点后 16 位的精度。但当取太差的初值时 (例如 $x_0 = 0.5$) ，迭代失败，下面就是计算中得到的发散序列：

$$0.5,\ -.75,\ -5.4375,\ -217.83984375,\ -332615.06236, \cdots$$

上面的话似乎都是针对常规精度 (单或双精度) 运算说的，但其结论对超长位数运算情形也是完全适用的。这时你只需把迭代式中的每个运算 (在目前的求倒数运算中，式 (5.13) 只含减法与乘法) 都理解成多精度运算便妥。如果已经有人提供了多精度四则运算子程序系统 (即把加、减、乘等运算逐一写成子程序，本书的下一章就提供这样的子程序)，那么你要实现多精度的倒数运算或别的什么算法也是很简单的，这时你只需把算式中涉及的四则运算逐一改为调用相应的子程序来实现就可以了。

5.4.3 利用牛顿迭代法作开方运算

求 a 的平方根 $\sqrt{a}$ 等价于解方程

$$x^2 - a = 0$$

由式 (5.10)，函数和导数分别是

$$f(x) = x^2 - a, \quad f'(x) = 2x$$

由式 (5.12)，这时的牛顿迭代公式是

$$x_{k+1} = x_k - \frac{x_k^2 - a}{2x_k} = x_k - \frac{x_k}{2} + \frac{a}{2x_k}$$

经整理后得

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right), \quad k = 0, 1, 2, \cdots \tag{5.14}$$

[例 5.4.2] 求 4 的平方根 $\sqrt{4}$。

结果是显然的：$\sqrt{4} = 2$，我们只是想通过又一个例子再次体会牛顿迭代公式的收敛过程。

给出初值 $x_0 = 1$

其后各步 $x_1 = \frac{1}{2} \times \left(1 + \frac{4}{1}\right) = 2.5$

$$x_2 = \frac{1}{2} \times \left(2.5 + \frac{4}{2.5}\right) = 2.05$$

$$x_3 = \frac{1}{2} \times \left(2.05 + \frac{4}{2.05}\right) = 2.000609756097561$$

$$x_4 = \cdots = 2.000000092922295$$

$$x_5 = \cdots = 2.000000000000002$$

通过这个例子，我们再次看到牛顿迭代公式的二次收敛特性，小数点后 0 的数目，在每次迭代后都增加了一倍。上述计算过程写成 BASIC 程序也是十分简单的。

```
DIM a AS DOUBLE, x AS DOUBLE
a = 4
x = 1
FOR k=1 TO 5
  x = (x + a / x) / 2
  PRINT k; x
NEXT k
END
```

5.4.4 同时算出 $\sqrt{a}$ 与 $\frac{1}{\sqrt{a}}$ 的方案

在 π 值计算中，有些公式既需要算出 $\sqrt{a}$，也需要算出 $1/\sqrt{a}$。原则上，我们可以先按 5.4.3 节方法算出 $\sqrt{a}$，然后再用倒数运算得到 $1/\sqrt{a}$。本节的目的是给出一个更快的算法，它不仅可以同时算出 $\sqrt{a}$ 和 $1/\sqrt{a}$，而且总的计算量甚至比只算一个 $\sqrt{a}$ 还省。其基本思路是，首先用牛顿迭代法直接算出 $1/\sqrt{a}$，然后用乘法得到 $\sqrt{a}$(如果 $\sqrt{a}$ 也是需要的话)，

即

$$\sqrt{a} = a \times \left(\frac{1}{\sqrt{a}}\right) \tag{5.15}$$

直接求 $1/\sqrt{a}$ 等价于解方程

$$\frac{1}{x^2} - a = 0$$

由式 (5.11)，其函数和导数分别是

$$f(x) = \frac{1}{x^2} - a, \qquad f'(x) = -\frac{2}{x^3}$$

牛顿迭代式是

$$x_{k+1} = x_k - \left(\frac{1}{x_k^2} - a\right) \Big/ \left(-\frac{2}{x_k^3}\right) = x_k + \frac{x_k - a x_k^3}{2}$$

或

$$x_{k+1} = \frac{x_k}{2}\left(3 - a x_k^2\right) \qquad k = 0, 1, 2, \cdots \tag{5.16}$$

对比直接求 $\sqrt{a}$ 与 $1/\sqrt{a}$ 的两个迭代式 (5.14) 和式 (5.16)，容易看出，在迭代的每一步，前者含有一个多精度数之间的除法运算 a/x_k，如前所述，它要使用多精度数的倒数运算实现，这其间又是一轮完整的牛顿迭代，所需的总运算量肯定多于直接求 $1/\sqrt{a}$ 的迭代式 (5.16) 中的 3 个多精度乘法运算。顺便指出，式 (5.16) 中还含有一个除法运算 $x_k/2$，虽然 x_k 是个 t 位的多精度数，但因除数 2 只是个短整数，其运算量只有几个 t 的量级，与两个多精度数相乘的工作量 t^2 相比几乎可以忽略。

5.4.5 关于导数与牛顿迭代式的补充介绍

本小节内容是为没有接触过导数概念，但又对求解方程的牛顿迭代法的来源有兴趣的读者而写。表述上力求通俗易懂而又篇幅不多，希望读后会确实感到学到了有用的新方法。

1. 导数的概念

设 x, y 分别是自变量和因变量，它们有函数关系

$$y = f(x) \tag{5.17}$$

现在我们研究函数 $f(x)$ 在点 $x = x_0$ 处的变化率。给 x_0 以增量 Δx，则函数 $f(x)$ 也取得相应的增量 Δy。它们的比值为

$$\frac{\Delta y}{\Delta x} = \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} \tag{5.18}$$

它显示了函数从 x_0 到 $x_0 + \Delta x$ 之间的平均变化。当 Δx 值接近于 0 时，此比值 (严格的数学用语是 Δx 趋向于 0 时比值的极限值) 称为函数 $f(x)$ 在 $x = x_0$ 处的导数，记为 $f'(x_0)$，实际上它就是函数 $f(x)$ 在 $x = x_0$ 处的变化率 (见图 5-1)。

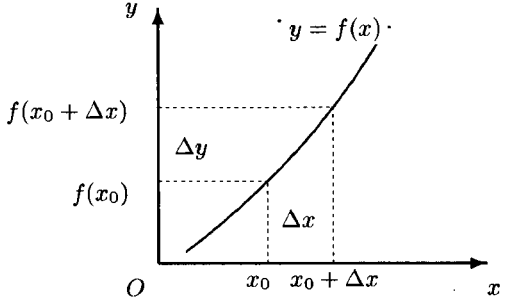

图 5-1 导数 (函数变化率) 示意图

2. 几个基本的导数

(1) $f(x)=c, c$ 为常数，即 $f(x)$ 是平行于 x 轴的一条直线，则对于任意一点 x_0，都有 $f(x_0)=f(x_0+\Delta x)=c$，于是 $\Delta y=0$，$\Delta y/\Delta x=0$，从而 $f'(x)=0$。

(2) $f(x)=ax, a$ 为常数，这时 $f(x)$ 是通过原点的一条直线，对于任意的 x_0，有

$$\frac{\Delta y}{\Delta x}=\frac{f(x_0+\Delta x)-f(x_0)}{\Delta x}=\frac{a(x_0+\Delta x)-ax_0}{\Delta x}=a$$

从而 $f'(x)=a$，即导数等于该直线的斜率。

(3) $f(x)=ax^2, a$ 为常数，这时 $f(x)$ 是通过原点的一条抛物线，对于点 x_0，有

$$\frac{\Delta y}{\Delta x}=\frac{a(x_0+\Delta x)^2-ax_0^2}{\Delta x}=2ax_0+a\Delta x$$

因 Δx 要趋向 0，上式第二项消失，最终有 $f'(x)=2ax$。

(4) $f(x)=1/x$, 对于点 x_0，有

$$\Delta y=\frac{1}{x_0+\Delta x}-\frac{1}{x_0}=-\frac{\Delta x}{(x_0-\Delta x)x_0}$$

从而 $\Delta y/\Delta x=-1/(x_0^2-x_0\Delta x)$，当 Δx 趋向 0，得 $f'(x)=-\frac{1}{x^2}$。

(5) 对于更一般的幂函数，可证

$$f(x)=ax^n \quad \text{的导数是} \quad f'(x)=anx^{n-1} \tag{5.19}$$

其证明可仿照 (3) 并使用牛顿二项式

$$(a+b)^n = a^n + na^{n-1}b + \frac{n(n-1)}{2}a^{n-2}b^2 + \cdots + b^n$$

式 (5.19) 是个重要的导数式子，且 n 可以不限于正整数。例如 $n = \pm\frac{1}{2}, -1, -2, \cdots$ 公式依然成立，从而包含了远比 (1) 至 (4) 更丰富的结论。以 (4) 为例，这时 $f(x) = 1/x = x^{-1}$，从而 $f'(x) = -x^{-2}$。又如 $f(x) = a/\sqrt{x} = ax^{-\frac{1}{2}}$，有 $f'(x) = -\frac{a}{2}x^{-\frac{3}{2}}$。

(6) 可以证明：和的导数等于导数的和，即 $f(x) = f_1(x) + f_2(x)$ 的导数是

$$f'(x) = f_1'(x) + f_2'(x) \tag{5.20}$$

事实上，由 (5.18) 式，易知

$$\frac{\Delta y}{\Delta x} = \frac{(\Delta y_1 + \Delta y_2)}{\Delta x} = \frac{\Delta y_1}{\Delta x} + \frac{\Delta y_2}{\Delta x}$$

下面是一个例子：

$$\text{若 } f(x) = ax^2 + bx + c, \text{ 则 } f'(x) = 2ax + b \tag{5.21}$$

通过这里的简短介绍，我们不难验证 5.4.1 节中与倒数、开方有关的几个导数 (见式 (5.9) ~ 式 (5.11))。

3. 与导数有关的几个概念

(1) 物理学中的速度概念　众所周知，物理学中关于匀速和加速运动的公式分别是

$$s = vt, \qquad s = \frac{1}{2}at^2 \tag{5.22}$$

式中 t 是时间，s 是距离，v, a 分别是速度与加速度。若把 s 看作是关于变量 t 的函数，依次记为 $s_1(t)$, $s_2(t)$，根据上述式 (5.19) 的结论，其导数分别是

$$s_1'(t) = v, \qquad s_2'(t) = at \tag{5.23}$$

根据导数是函数变化率的概念，距离关于时间的导数就是距离关于时间的变化率，即物理学中“速度”的概念。由式 (5.22)，导数在匀速运动中就是普通的速度，在加速运动中就是加速度与时间的乘积，后者表明这时的速度是随时间变化的，在 t_0 那一时刻的速度是 at_0 。

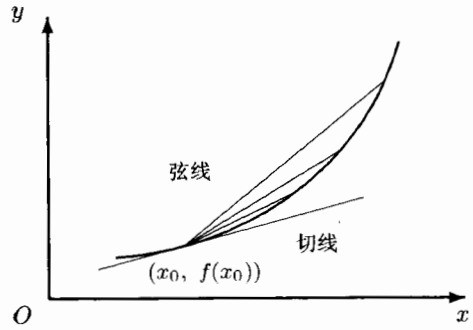

图 5-2 弦线逐渐靠近切线

(2) 切线的斜率　在图 5-2 中，连接点 $(x_0,\ f(x_0))$ 与点 $(x_0 + \Delta x,\ f(x_0 + \Delta x))$ 的弦，当 Δx 越来越小时，过点 $(x_0,\ f(x_0))$ 的弦就越来越靠近过点 $(x_0,\ f(x_0))$ 的切线。若 Δx 趋向 0，弦线与切线重合，于是函数 $f(x)$ 在点 $x = x_0$ 的导数就是曲线 $y = f(x)$ 在点 $(x_0,\ f(x_0))$ 的切线的斜率。

(3) 函数的最大和最小值　在图 5-3 中容易看出，在函数达到最大或最小的地方，其切线与 x 轴平行 (斜率为 0)。如上所述，切线斜率可通过导数 $f'(x)$ 求得，所以求函数在何处达到最大或最小等价于解方程 $f'(x)=0$。例如由式 (5.21)，$f(x)=3x^2-4x+1$ 的导数是 $f'(x)=6x-4$，所以在 $x=\frac{2}{3}$ 处有最小值

$$f\left(\frac{2}{3}\right)=3\times\left(\frac{2}{3}\right)^2-4\times\frac{2}{3}+1=-\frac{1}{3}$$

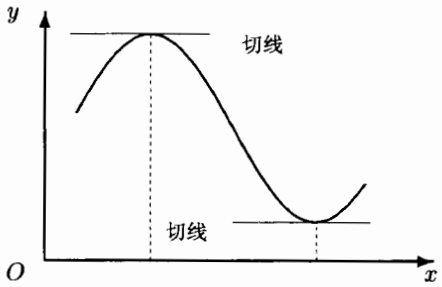

图 5-3　函数的最大与最小值

4. 牛顿迭代式的导出

根据 1.关于导数的概念，不难导出 5.4.1 节的牛顿迭代式 (5.12)。事实上，由 (5.18) 式并令 $x_1=x_0+\Delta x$ 便得

$$\frac{f(x_1)-f(x_0)}{x_1-x_0}=f'(x_0) \tag{5.24}$$

由导数定义，当 Δx 趋向 0，即 x_1 趋向 x_0 时，上式等号才成立，否则只是一个近似式。如果 x_1 是方程 $f(x)=0$ 的解 (即

$f(x_1)=0)$，而 x_0 是 x_1 附近的一个点，那么 (5.24) 式可变为

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{5.25}$$

如上所述仅当 x_1 趋向 x_0 时，上式等号才成立。在其他情形，由它算出的 x_1 不一定是真解 x_1 。但我们有理由相信，只要 x_0 与 x_1 不是相距很远，算出的 x_1 将比原来的 x_0 更靠近真解。于是把刚算出的 x_1 看作是 (5.25) 式中的 x_0 ，并使用该式再算一次，而把得到的更新结果记为 x_2 。这样的反复计算过程可简写成

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \qquad k=0,1,2,3,\cdots \tag{5.26}$$

这便是出现在 5.4.1 节中的牛顿迭代式 (5.12) 。

第 6 章　多精度运算程序系统 BMP

6.1　与多精度数对应的数组类型选择

我们在第 5 章已经约定了多精度数的表示形式，并讨论了多精度数之间的四则运算和开方运算方法。其要点是把一个 t 位 p 进制数

$$a = \pm p^m \left(\frac{a_1}{p^1} + \frac{a_2}{p^2} + \cdots + \frac{a_t}{p^t} \right)$$

与一个数组 (元素不小于 $t+2$ 个) 对应，具体的对应关系是

数组元素:	$a(1)$	$a(2)$	$a(3)$	$a(4)$	$a(5)$	$\cdots$	$a(t+2)$
存放内容:	数符 $\pm$	阶 m	a_1	a_2	a_3	$\cdots$	a_t

在第 5 章中，我们甚至还进一步列举了好些例子说明“万进制” (即 $p = 10000$) 数与十进制数的对应关系。尽管如此，在程序设计中仍留下很大的灵活性。因为在 BASIC 中，数值型数组有四种类型 (见第 5 章 5.1 节)。而第 5 章讨论的多精度数的四则运算实现方法中，用整型或实型数组都是可以的。例如，可以是万进制 (4 位十进制) 对应整型数组，也可以是亿进制 (8 位十进制) 对应长整型数组，还可以用十万或百万进制 (5 或 6 位十进制) 对应单精度实型数组等等。在作出选择时，除应考虑节省存储空间外，还要考虑在这种表示方法的基础上，其四则运算的程序设计是否方便，以及程序运行

速度是否令人满意等。显然，亿进制在乘法运算实现时比较麻烦 (因两个 8 位数相乘可以得到 16 位的结果，即使是采用双精度实型数表示，有时也不能把它的全部有效数字绝对精确地表出)。在 PC 机上，通常整型 (定点) 运算要比实型 (浮点) 运算快，而且万进制与两字节的整型数组对应时，存储单元也比较节省。所以我们认为 **万进制与整型数组对应是个较好的组合**，并且在其后的程序设计中使用这种组合。这样一来，t 位尾数的多精度数 a 所对应的数组可用 BASIC 的维数语句定义如下：

```
DIM a%(t4)  或  DIM a(t4) AS INTEGER
```

此处要求数组的下标上限 $t4$ 不小于 $t+4$，这是因为在多精度数的表示中，数符和阶各占一个单元。除此之外，在运算过程中，还需留出两个单元作为保证运算精度的附加位 (见第 5 章 5.3.1 和 5.3.2 节关于加、减法的高阶对位)。

最后应该指出，QBasic 限定每个整型数组的元素个数不超过 32768，所以用它表示的多精度数的最高精度，实际上只有十进制的十三万位左右。若想更多的位数，就要使用 Quick BASIC、Visual BASIC 或其他种类的计算机语言。

6.2 BMP 系统中的主要子程序

我们在 **万进制数对应整型数组** 的约定下，开发了一个 BASIC 语言的多精度运算系统 BMP，它既可以在 DOS 环境下的 QBasic 和 Quick BASIC 上运行，也可以在 Windows 下的 Visual BASIC 上运行。BMP 中的主要子程序如表 6-1 所示，其源程序将在附录 A 中给出。如上所述，各子程序参数

中的数组，如无特别声明，均应理解为整型数组。当多精度数的尾数是 t 位时，数组的下标上限不小于 $t+4$ 。为使子程序调用简单，数组下标的上限没有明显地出现在参数中。作为替代手段，在主程序中应把这个上限申明为全局 (共享) 变量。关于这类申明的语句，可参看 6.3 节或下章的计算程序。

表 6-1　多精度运算系统 BMP 中主要子程序列表

程序名及参数	功　能	算法根据及注
$bcommon(m,cfn\$)$	置全局量	m 位十进制运算，见注 1
$badd(a(),b(),c())$	加法 $c=a+b$	见 5.3.3 及 5.3.1, 5.3.2 节
$bsub(a(),b(),c())$	减法 $c=a-b$	见 5.3.3 及 5.3.1, 5.3.2 节
$bmul(a(),b(),c())$	乘法 $c=a*b$	调用子程序 $bmul1/bmul2$
$bdiv(a(),b(),c())$	除法 $c=a/b$	$a*(1/b)$ 见 5.3.5 节
$binv(a(),x())$	倒数 $x=1/a$	牛顿迭代见 5.4.2 节
$bsqrt(a(),x(),y(),s)$	开方 $x=\sqrt{a}$,	牛顿迭代见 5.4.4, 5.4.3 节
	同时 $y=1/\sqrt{a}$	当 $s=-1$, 只求 $y=1/\sqrt{a}$
$bmuli(a(),b1,c())$	短乘 $c=a*b1$	$b1$ 为一个整型数
$bdivi(a(),b1,c())$	短除 $c=a/b1$	$b1$ 为一个整型数
$bdtom(d1,b())$	双送多 $b=d1$	双精度数 $d1$ 变为多精度 b
$bmtod(a(),d1)$	多送双 $d1=a$	多精度数 a 变为双精度 $d1$
$bcomp(a(),b(),r,m)$	比较 $\|a\|\sim\|b\|$	绝对值比较，见注 2
$bcopy(a(),b())$	拷贝 $b=a$	
$bprint(cc\$,a(),m,k)$	输出 $a(1)\cdots a(m)$	每行 $\|k\|$ 个，见注 3
$bprt50(cc\$,a(),m,sc)$	输出 $a(1)\cdots a(m)$	印 π 用，见注 4
$binput(a(),m)$	输入 $a(1)\cdots a(m)$	
$bmul1(a(),b(),c())$	直接乘 $c=a*b$	见 5.3.4 节
$bmul2(a(),b(),c())$	FFT 乘 $c=a*b$	见注 5
$tadd(a(),b(),c())$	标准加 $c=a+b$	见 5.3.1 节
$tsub(a(),b(),c())$	标准减 $c=a-b$	见 5.3.2 节

限于版面，表 6-1 中各子程序的说明比较简短，下面将用注解方式对几个子程序作必要的补充。

注 1：这是 BMP 系统中作准备工作用的子程序，包括给全局 (共享) 变量赋初值，并打开用作输出的文件 cfn(文件号为 #2) 等。长整数 m 是用户指定的运算中使用的尾数位数 (十进制)。

注 2：r 是比较后产生的结果，有值 0, 1, 2 分别对应于 $|a|=|b|$, $|a|>|b|$, $|a|<|b|$。本子程序实际上是逐一比较 $a(i)$ 与 $b(i)$, $i=2,3,\cdots,n$, m 是第一次出现 $a(i)\neq b(i)$ 时的 i 值。若所比较的值全部相等，则 $m=n+1$。

注 3：使用 PRINT #2 输出 cc\$ 和 $a(1),a(2),\cdots,a(m)$，此处 cc\$ 是字符串，整型数组 a 对应着一个万进制数。输出时每行含 $|k|$ 个数组元素，各元素之间有一空格分隔 (即每个元素 $a(i)$ 占 5 个字符)。但首行例外，首行只含符号串 cc\$(不论原来有多少个字符，输出时只取其前 5 个)。当参数 $k<0$ 时，输出除指向 #2 所对应的文件外，还以相同格式显示在屏幕上。例：若 $a(\,)$ 中以万进制存放了 π 值，则

```
CALL bprint("pi =",a( ),28,10)
```

对应的输出是

```
pi =      1    1
 0003 1415 9265 3589 7932 3846 2643 3832 7950 2884
 1971 6939 9375 1058 2097 4944 5923 0781 6406 2862
 0899 8628 0348 2534 2117 0679
```

注 4：使用 PRINT #2 输出 cc\$ 和 $a(1),a(2),\cdots,a(|m|)$，此处 cc\$ 是字符串，整型数组 a 对应着一个万进制数。输出尾数时每行必定是含有 50 位十进制数字，每十位之间有一个

空格分隔。具体格式与 cc\$ 的最后一个字符内容是否为小数点 "." 有关。例：若 $a(\,)$ 中以万进制存放了 π 值，则

```
CALL bprt50("pi =",a( ),28,1)
```

对应的输出是

```
pi =    1    1
  0003141592 6535897932 3846264338 3279502884 1971693993
  7510582097 4944592307 8164062862 0899862803 4825342117
  0679
```

上面的输出格式对应于 cc\$ 的最后字符非 "."，这时首行是符号串、数符和阶，即 cc\$, $a(1)$, $a(2)$ 。其余各行的左边都一律留两个空格。

cc\$ 最后字符为 "." 所对应的格式，是 BMP 为输出 π 值而专门设计的。此格式也适用于万进制数中恰有 1 位整数 (即阶 $a(2)=1, a(3)\neq 0$) 的情形。打印时，每行的左边留出 k 个空格 (k 为 cc\$ 的长度，$k\leqslant 5$)，但首行的 k 个空格先被置上 cc\$，然后 $a(3)$ 的整数值被放入第 1 至 4 个位置上 (所以当字符 cc\$ 的长度较短，而整数的位数又较多时，某些字符可能被覆盖)。例：若 $a(\,)$ 中以万进制存放了 π 值，则

```
CALL bprt50(" .",a( ),28,1)
```

对应的输出是

```
3.1415926535 8979323846 2643383279 5028841971 6939937510
  5820974944 5923078164 0628620899 8628034825 3421170679
```

本子程序的第 3 个参数 m 的值可正可负，$a(|m|)$ 是要输出的最后一个元素。当 $m<0$ 时，程序对于 6 行以上的输出作删节，这时除保留头两行和最后的 3 或 4 行 (含有 104 至

200 位) 之外，其他内容被一行删节号替代 (请看下一章的 π 值计算程序输出例子)。

此外，本子程序的最后一个参数 sc 可取值 0 或 1。若 $sc \neq 0$，上述输出内容还将显示在屏幕上，为用户及时了解 π 值的计算进程提供方便。

注 5：本子程序使用 FFT(快速傅立叶变换) 方法实现两个多精度数相乘，运算速度大大提高，但数学与程序设计都十分复杂。详见前章 5.3.6 节和最后一章。

6.3 BMP 系统子程序调用方法与实例

6.3.1 全局变量的申明与赋值

如上所述，为使调用 BMP 子程序时所需填写的参数个数尽可能少些，存放尾数为 t 位的多精度数的数组下标上限 $(t+4)$ 没有明显地出现在参数中。作为替代手段，这时需要在主程序中把与此上限有关的变量 $n = t+2$，$nmax = t+4$ 申明为全局 (共享) 变量。另外，在我们的程序设计方案中，已经考虑到变更进制 p 等的灵活性，也要求在主程序中把 p 等参数申明为全局变量。实现这类申明和赋值的方法是使用如下 **一段用于全局变量申明与赋值的标准语句**:

```
COMMON SHARED nmax, n
COMMON SHARED p AS INTEGER, pd2 AS INTEGER
COMMON SHARED l2p31 AS LONG, d2p53 AS DOUBLE
      ' here l2p31 = L2P31, not 12p31
COMMON SHARED powi AS INTEGER, powbd AS INTEGER
COMMON SHARED powl AS LONG
OPTION BASE 1
```

```
DIM SHARED pow(167) AS LONG

CALL bcommon (m, cfn$)
```

注 1：OPTION BASE 1 意味着数组元素改为从 1 开始。最后被调用的子程序 *bcommon* 是由 BMP 系统提供的，用于全局变量赋值等准备工作，详见附录 A 。调用时，用户需具体给出参数 m 与 $cfn\$$ 的值，其中 m 为用户要求算 π 的 (十进制) 位数；$cfn\$$ 是存放输出结果的文件名。例如要计算 8192 位 π 值，结果放入文件 *pi.res* 内，则有

```
CALL bcommon (8192, "pi.res")
```

子程序 *bcommon* 内含有如下一些语句：

```
n = (m + 3) \ 4 + 2
nmax = n + 2
p = 10000
pd2 = p / 2
l2p31 = 2147483647                ' = 2^31 - 1
d2p53 = 9007199254740991#         ' = 2^53 - 1
powi = 1
powbd = 32600
OPEN cfn$ FOR OUTPUT AS #2
```

其中 $n-2$ 是多精度数的尾数长度， $nmax$ 为多精度数组的长度 (包括数符、阶和两个附加位)， $p=10000$ 表示使用万进制。 $l2p31 = 2^{31}-1$(首字符是字母 L 的小写，不是数字 1) 和 d2p53$= 2^{53}-1$ 分别是长整数和双精度实型整数的最大值。名字中含有字母 *pow* 的几个变量和数组都用于扩大多精度的阶. BMP 系统虽是万进制与短整型数组对应的产物，但

阶的取值范围却由短整数的 $-32768 \sim 32767$ 扩大为长整数的 $-2147483648 \sim 2147483647$。这时设“阶” j 是短整数，BMP 约定若 $|j| \leqslant powbd = 32600$ 则阶就是 j 自身，否则阶由长整数组的元素 $pow(|j| - powbd)$ 来表示，这时 j 实际上用于指示真实阶的位置。OPEN 语句打开文件 $cfn\$$ 作为存放输出结果之用，其中文件号 #2 与 BMP 系统的输出子程序 *bprint* 和 *bprt*50 的文件号一致。

注 2：在 Windows 的 Visual BASIC 中，上述申明稍有差异，这时只需把 COMMON SHARED 修改为 GLOBAL。详见下章 π 值计算程序。

6.3.2 调用 BMP 系统子程序的实例

我们通过一些简单的例子来说明调用 BMP 系统子程序的方法。

[例 6.3.1] 用 BMP 系统计算 $\frac{1}{\sqrt{a}}$。

在第 5 章 5.4.4 节中，我们介绍了直接算出 $1/\sqrt{a}$ 的牛顿迭代公式：

$$x_{k+1} = \frac{x_k}{2}(3 - ax_k^2) \quad k = 0, 1, 2 \cdots \tag{6.1}$$

并指出此式不含两个多精度数之间的除法，在多精度运算中将比先算 $\sqrt{a}$，然后再用除法得 $1/\sqrt{a}$ 的方案有更高的效率。为了更清晰地说明一般双精度计算程序与多精度计算程序的差别，我们首先写出前者，然后机械地把前者转换为后者。计算中令 $a = 49$，迭代初值 $x_0 = 0.1$，结果是循环小数：

$$1/\sqrt{49} = 0.142857142857142857\cdots$$

一般双精度计算程序 (迭代 6 次)

```
OPEN "pi631.res" FOR OUTPUT AS #2
a# = 49#
x# = .1#
FOR k=1 TO 6
  x# = x# * (3# - a# * x# * x#) / 2#
  PRINT #2, "k, x ="; k; x#
NEXT k
END
```

多精度计算程序 (迭代 9 次)

```
REM  此程序应先抄录 6.3.1 节中一段用于全局变量
REM  申明与赋值的标准语句, 但最后的调用语句改为

CALL bcommon(200, "pi631.res")

REM ex631.bas
DIM a(nmax) AS INTEGER, x(nmax) AS INTEGER
DIM u(nmax) AS INTEGER, w(nmax) AS INTEGER
CALL bdtom(49#, a())          ' a = 49#
CALL bdtom(.1#, x())          ' x = .1#
CALL bdtom(3#, u())           ' u = 3#
FOR k = 1 TO 9
  CALL bmul(x(), x(), w())    ' w = x*x
  CALL bmul(a(), w(), w())    ' w = a*x*x
  CALL bsub(u(), w(), w())    ' w = 3 - a*x*x
  CALL bmul(x(), w(), w())    ' w=x*(3-a*x*x)
```

```
    CALL bdivi(w(), 2, x()) ' x=x*(3-a*x*x)/2
    PRINT #2, "k ="; k
    CALL bprt50("x=", x(), n, 1)
NEXT k
END
```

双精度程序已十分清晰，不再解释，其第 5, 6 两步的结果是

.1428571428570444,　.1428571428571428

至于多精度计算程序，其前头几行是使用 BMP 时必须有的一段用于全局变量申明与赋值的标准语句。根据 6.3.1 节，全局变量的名字有如下 10 个：

$nmax$, n, p, $pd2$, $l2p31$, $d2p53$, $powi$, $powl$, pow

它们有专门的含义，用户不要另作它用。紧跟的子程序调用

```
CALL bcommon(200, "pi631.res")
```

是为给 BMP 系统中的全局变量赋值，这也是 BMP 系统规定的。此处由用户指定的两个参数表示运算中取 200 位十进制数，且运算结果放入文件 $pi631.res$ 内。再其后的各行是牛顿迭代过程，除 a, x 两个数组外，又定义了另外两个数组 u, w 用于存放中间结果。其他内容确实是前面的双精度程序中的赋值语句的机械翻译，在表 6-1 中均能查到被调用的各子程序的功能。

最后需要指出的是，用户程序中调用的 BMP 子程序应从附录 A 中一一检出附在这里的程序之后。检出时应注意不要遗漏了被间接调用的子程序。因为整个 BMP 系统不大，所以最保险的办法是把附录 A 中的所有子程序全都放在这里的程序之后。

本程序经过 9 步迭代后获得 192 位有效数字，由子程序 *bprt*50 输出的结果如下：

```
x=     1     0
  1428571428 5714285714 2857142857 1428571428 5714285714
  2857142857 1428571428 5714285714 2857142857 1428571428
  5714285714 2857142857 1428571428 5714285714 2857142857
  1428571428 5714285714 2857142857 1428571428 5698037954
```

注：由于不同的 BASIC 系统对于双精度数中不能用二进制精确表示的数 (如 0.1) 的处理稍有差异，因此 BMP 中与此有关的某些子程序 (如把双精度数转换为多精度数的子程序 *bdtom*) 的结果也可能稍有差异。例如，本程序使用了下面的语句

```
CALL bdtom(.1#, x())          ' x = .1#
```

给迭代赋初值：$x_0 = 0.1$，而这时的初值可能有微小差异并导致其后迭代结果在非有效数字区域上也有差异。例如第 1 步的结果对于 DOS 6.22 上的 QBasic 而言是

```
x=     1     0
  1255000000 0000000424 5749999999 9997736016 2499999999
  5811630062 5000000000 0000000000 0000000000 0000000000
  0000000000 0000000000 0000000000 0000000000 0000000000
  0000000000 0000000000 0000000000 0000000000 0000000000
```

而 Quick BASIC (4.0) 却得到精确的尾数，即除 1255 之外全为 0 。好在牛顿迭代法对微小的初值变化不那么敏感，计算依然正常地进行，并最终得到正确的结果。本程序前 5 步迭代所得的有效数字与简单的双精度程序完全相同，其第 5 至第 9 各步结果所得的有效数位依次是

```
12,  24,  48,  96,  192
```

符合牛顿迭代具有二阶收敛的一般规律. 为了去掉多精度计算中的无效运算以节省运算时间, 在牛顿迭代过程中, 参加计算的位数开始时可以少些, 其后随迭代进程倍增. 以本程序为例, 可令前 6 步的 $n=10$, 其后各步 $n=(n-4)*2+4$, 以代替一成不变的 $n=52$. BMP 系统中的求倒数和开方子程序 *binv* 和 *bsqrt* 都采用了类似的处理方法 (详见附录 A).

[例 6.3.2] 用 BMP 系统计算组合 C_m^n.

在第 3 章 [例 3.2.3]中, 为了熟悉 QBasic 语言特点, 我们介绍了计算组合

$$C_m^n=\frac{m!}{n!(m-n)!} \tag{6.2}$$

的一个简单程序. 这里为了熟悉多精度系统 BMP 中子程序的调用, 再次使用这个例子. 如上例那样, 为了更清晰地说明一般双精度计算程序与多精度计算程序的差别, 我们只是机械地把第 3 章的双精度程序转换为多精度程序.

```
REM  此程序应先抄录 6.3.1 节中一段用于全局变量
REM  申明与赋值的标准语句, 但最后的调用语句改为

CALL bcommon(120, "pi632.res")

REM ex632.bas
DIM cc(nmax) AS INTEGER
DO
  INPUT "m="; m
  IF m <= 0 THEN EXIT DO
  INPUT "n="; nn
  CALL cmn(m, nn, cc())
```

```
    PRINT "m, n ="; m; nn
    CALL bprint("Cmn =", cc(), n, -10)
    PRINT " "
  LOOP
  END

  SUB cmn (m, nn, c() AS INTEGER)
  DIM p(nmax) AS INTEGER, q(nmax) AS INTEGER
  IF nn <= m / 2 THEN
    k = nn
  ELSE
    k = m - nn
  END IF
  IF k = 0 THEN
    CALL bdtom(1#, c())          ' c = 1#
    EXIT SUB
  END IF
    '  c = dprod#(m, m-k+1) / dprod#(k, 1)
  CALL dprod(m, m - k + 1, p())
  CALL dprod(k, 1, q())
  CALL bdiv(p(), q(), c())
  END SUB

  SUB dprod (nbig, nsmall, p() AS INTEGER)
  dbig# = nbig
  CALL bdtom(dbig#, p())
  FOR i% = nbig - 1 TO nsmall STEP -1
    CALL bmuli(p(), i%, p())
  NEXT i%
  END SUB
```

如同上例那样，程序中的前几行包括对 BMP 的全局变量申明和赋值。并约定运算中取 120 位十进制数，运算结果存入文件 *pi*632.*res* 。要注意的是，原来的双精度程序中的变量名 n 与 BMP 的全局变量名字冲突，现已改为 nn 。原来子程序 *cmn* 中存放结果的最后一个双精度参数已按惯例改为存放多精度数的整型数组；最后是用于连乘积的函数 *dprod* 被改为子程序，因为现在的结果不再是单个变量，而是存放多精度数的一个数组。与此相应，子程序中增加了一个数组参数 $p(\,)$ 来存放结果。除此以外，在这个子程序中的乘法没有使用乘数和被乘数都是多精度数的那个子程序 *bmul* ，而是使用一个多精度数与一个整型数的乘法子程序 *bmuli* ，后者的运行时间显然大为节省。令人意外的是这段程序并不比原来的双精度程序长多少，个别子程序的长度甚至没有变化。

第 7 章　多精度的 π 值计算程序

本章在多精度系统 BMP 的基础上给出两个算 π 子程序和调用它们的主程序。它们所用的算法包括算术几何平均法和波尔温 4 阶收敛公式，这是专业人士都在使用着的当今世界上最快的算法。有关的算法已在第 2 章 2.2.2 节给出，并且在讲解如何编写 BASIC 程序时，由双精度型变量编写的十余行示范性程序也曾出现在第 4 章内 (见 [例 4.1.5] 和 [例 4.1.6])。为了更好地理解这里的多精度程序，阅读时最好对比着那里的简单程序，并记住第 6 章表 6-1 中关于多精度运算系统 BMP 的主要子程序功能 (或把它们写在卡片上供随时查阅)。

7.1　π 值计算的两个子程序

由于两个子程序有类似的程序设计风格，所以我们首先给出它们的完整程序列表，然后再作一些补充说明。

7.1.1　算术几何平均 (AGM) 方法子程序列表

这一小节只给出完整的子程序列表，有关说明将留在另一子程序列表之后的 7.1.3 节统一给出。

```
REM *** *** compute pi (m digits, AGM)
SUB a1piagm (m)
REDIM a(nmax) AS INTEGER, b(nmax) AS INTEGER
```

```
REDIM a2(nmax) AS INTEGER, b2(nmax) AS INTEGER
REDIM c2(nmax) AS INTEGER, s(nmax) AS INTEGER
DIM dg AS LONG, j2 AS INTEGER

t0$ = TIME$
PRINT #2, "Algorithm AGM"
logp = INT(LOG(p) / 2.30257)
          ' estimate kk -- the number of iterations
dg = 41
FOR kk = 5 TO 20
  IF m <= dg THEN EXIT FOR
  dd = 2 + kk MOD 2
  IF kk > 10 THEN dd = 3 - kk MOD 2
  IF kk > 12 THEN dd = 2
  IF kk = 16 THEN dd = 1
  dg = 2 * dg + dd
NEXT kk
IF kk > 7 AND m - 10 < dg / 2 THEN kk = kk - 1
               ' set initial values
CALL bdtom(1#, a())              ' a0=1
CALL bdtom(2#, c2())
CALL bsqrt(c2(), s(), b(), -1)   ' b0=1/sqr(2)
CALL bcopy(a(), s())             ' s0=1
j2 = 2
PRINT #2, "step = 0"; " /"; kk; "  ";
PRINT #2, t0$; "--"; TIME$
' QB 01
PRINT "step = 0"; " /"; kk; "  "; t0$; "--"; TIME$
' VB 04
Form1.Text1.Text = Form1.Text1.Text & "step = 0" _
```

```
  & " / " & kk & "  " & t0$ & "--" & Time$ & _
  Chr$(13) & Chr$(10)
Form1.Text1.Refresh
                ' iteration
FOR k = 1 TO kk
  CALL bmul(a(), b(), b2())      ' (b1)^2 = a0*b0
  CALL badd(a(), b(), c2())
  CALL bdivi(c2(), 2, a())       ' a1 = (a0+b0)/2
  CALL bsqrt(b2(), b(), b(), 1) ' b1 = sqr(a0*b0)
  CALL bmul(a(), a(), a2())
  CALL bsub(a2(), b2(), c2())    ' (c1)^2=a1^2-b1^2
  IF k >= 14 THEN
    CALL bmuli(c2(), 16384, c2())
    IF k = 14 THEN j2 = 1
  END IF
  j2 = 2 * j2
  CALL bmuli(c2(), j2, c2())     ' 2^(k+1)*c1
  CALL bsub(s(), c2(), s())      ' s1=s0-2^(k+1)*c1
  CALL binv(s(), c2())
  CALL bmuli(c2(), 4, c2())
  CALL bmul(a2(), c2(), a2())    ' pia = 4(a1)^2/s1
  CALL bmul(b2(), c2(), b2())    ' pib = 4(b1)^2/s1
  CALL bcomp(a2(), b2(), r, mk) ' a2(mk) <> b2(mk)
  digits = logp * (mk - 4) + 5 _
    - LEN(STR$(a2(mk) - b2(mk)))
  IF r = 0 THEN digits = digits + 1
  PRINT #2, "step ="; k; "/"; kk; "  ";
  PRINT #2, t0$; "--"; TIME$; "  digits ="; digits
' QB 02
```

```
  PRINT "step ="; k; "/"; kk; "  ";
  PRINT t0$; "--"; TIME$; "  digits ="; digits
' VB 05
  If k > 4 And k Mod 3 = 2 Then Form1.Text1.Text _
    = " "
  Form1.Text1.Text = Form1.Text1.Text & "step = " _
    & k & " / " & kk & "  " & t0$ & "--" & Time$ _
    & "  digits = " & digits & Chr$(13) & Chr$(10)
  CALL bprt50(" .", b2(), -mk, 1)
' VB 01
  Form1.Text1.Refresh
NEXT k
PRINT #2, " "
CALL bprt50(" .", b2(), mk, 0)
END SUB
```

7.1.2 波尔温 4 阶收敛算法子程序列表

下面是完整的子程序列表:

```
REM *** *** compute pi (m digits, Borwein 4-order)
SUB a2pib4 (m)
REDIM a(nmax) AS INTEGER, b(nmax) AS INTEGER
REDIM c(nmax) AS INTEGER, d(nmax) AS INTEGER
REDIM x(nmax) AS INTEGER, y(nmax) AS INTEGER
REDIM one(nmax) AS INTEGER, dg(11) AS LONG
DIM j2 AS INTEGER

t0$ = TIME$
PRINT #2, "Algorithm B4"
dg(1) = 7: dg(2) = 40: dg(3) = 170: dg(4) = 693
```

```
dg(5) = 2787: dg(6) = 11170: dg(7) = 44700
dg(8) = 178823: dg(9) = 715318: dg(10) = 2861296
dg(11) = 11445209
          ' estimate kk -- the number of iterations
FOR kk = 1 TO 11
  IF m <= dg(kk) THEN EXIT FOR
NEXT kk
dg(kk) = m
IF kk > 3 AND m - 10 < dg(kk - 1) THEN kk = kk - 1
                    ' set initial values
CALL bdtom(2#, one())
CALL bsqrt(one(), x(), x(), 1)    ' x=sqr(2)
CALL bmuli(x(), 4, a())
one(3) = 6
CALL bsub(one(), a(), a())     ' a0 = 6-4*sqr(2)
one(3) = 1
CALL bsub(x(), one(), y())     ' y0 = sqr(2)-1
CALL bmuli(x(), 12, b())
one(3) = 17
CALL bsub(one(), b(), x())  ' x0=y0^4=17-12*sqr(2)
one(3) = 1
j2 = 2
PRINT #2, "step = 0"; " /"; kk; "  ";
PRINT #2, t0$; "--"; TIME$
' QB 01
PRINT "step = 0"; " /"; kk; "  "; t0$; "--"; TIME$
' VB 04
Form1.Text1.Text = Form1.Text1.Text & "step = 0" _
  & " / " & kk & "  " & t0$ & "--" & Time$ & _
  Chr$(13) & Chr$(10)
```

```
Form1.Text1.Refresh
                        ' iteration
FOR k = 1 TO kk
  CALL bsub(one(), x(), x())
  CALL bsqrt(x(), y(), y(), 1)
  CALL bsqrt(y(), x(), x(), 1)   ' x=[1-y0^4]^(1/4)
  CALL badd(one(), x(), y())
  CALL bsub(one(), x(), x())
  CALL bdiv(x(), y(), y())       ' y1 = (1-x)/(1+x)
  CALL bmul(y(), y(), b())
  CALL bmul(b(), b(), x())       ' x1 = y1^4
  CALL badd(one(), b(), c())
  CALL bmul(c(), y(), c())
  CALL badd(c(), b(), d())
  IF k >= 7 THEN
    CALL bmuli(d(), 8192, d())
    IF k = 7 THEN j2 = 1
  END IF
  j2 = 4 * j2                    ' j2 = 2^(2k+1)
  CALL bmuli(d(), j2, d())       ' d=j2*(y1+y1^2+y1^3)
  CALL bmuli(c(), 4, c())
  CALL badd(one(), c(), c())
  CALL bmuli(b(), 6, b())
  CALL badd(c(), b(), c())
  CALL badd(c(), x(), c())
  CALL bmul(c(), a(), a())
  CALL bsub(a(), d(), a())       ' a1=a0*(1+y1)^4-d
  mk = (dg(k) + 3) \ 4 + 3
  IF k < kk THEN
    nn = n
```

```
    n = mk + 2 * kk
    IF n > nn THEN n = nn
  END IF
  CALL binv(a(), b())
  IF k < kk THEN n = nn
  PRINT #2, "step ="; k; "/"; kk; "  ";
  PRINT #2, t0$; "--"; TIME$; "  digits ="; dg(k)
' QB 02
  PRINT "step ="; k; "/"; kk; "  ";
  PRINT t0$; "--"; TIME$; "  digits ="; dg(k)
' VB 05
  If k > 4 And k Mod 3 = 2 Then Form1.Text1.Text _
    = " "
  Form1.Text1.Text = Form1.Text1.Text & "step = " _
    & k & " / " & kk & "  " & t0$ & "--" & Time$ _
    & "  digits = " & dg(k) & Chr$(13) & Chr$(10)
  CALL bprt50(" .", b(), -mk, 1)
' VB 01
  Form1.Text1.Refresh
NEXT k
PRINT #2, " "
CALL bprt50(" .", b(), mk, 0)
END SUB
```

7.1.3 关于两子程序的补充说明

1. 这两个子程序都需要在多精度系统 BMP 的支持下才能运行，所以调用前需作必要的准备工作 (见 7.2 节有关的主程序)。子程序中的参数 m 是用户要求计算 π 的位数，数组上限 $nmax$ 是个公用变量，其值由 BMP 系统的赋值子程序 *bcommon* 根据 m 自动给出 (见 6.3.1 或 7.2 节)。函数 TIME$

用于计时。程序中有如下三个独立的英文注解行：

```
' estimate kk -- the number of iterations
' set initial values
' set iteration
```

它们的含义分别是“估计迭代次数 kk”，“置初值”和“迭代”。事先估计并输出计算所需的迭代次数 kk，为用户带来方便，但它不是算法本身所必需。其中第一个子程序所使用的估计算法和第二个子程序所用的数表都是根据我们过去的计算结果构造的 (请参考 7.3 节 2.)。至于其后的置初值和迭代等语句段的含义，只要对比着第 4 章 [例 4.1.5] 和 [例 4.1.6] 内的简单程序就容易明白。

2. 我们要求提供的程序可以分别在 DOS 下的 QBasic 或 Quick BASIC (统称 QB) 以及 Windows 下的 Visual BASIC(简称 VB) 中运行，所以特别留意所使用的语句的通用性。但 QB 与 VB 毕竟有很大的差别，尤其是向屏幕输出方面，前者只对应惟一的屏幕，后者却可以对应着不同的 Window。对于算 π 程序而言，作业时间可能长达数小时，而适时的屏幕输出对于了解计算进程又显得至关重要，为此，我们采用同时并列 QB 与 VB 两类屏幕输出语句的方法，并分别在它们的前面加上风格统一的注解行。例如有

```
' QB 01      或      ' VB 04
```

前面的注解行表示其后的 1 行语句用于 QB，而后者则是其后的 4 行语句用于 VB。用户可根据需要只取其一。值得注意的是，PRINT #2 是向硬盘写文件语句，而不是屏幕输出语句。仅当 PRINT 后不带 # 号的才是 QB 中的屏幕输出语

句。此外，附录 A 中也有少数几个子程序的屏幕输出语句需作同样的取舍处理。

3. AGM 算法子程序中有如下几行

```
IF k >= 14 THEN
  CALL bmuli(c2(), 16384, c2())
  IF k = 14 THEN j2 = 1
END IF
j2 = 2 * j2
CALL bmuli(c2(), j2, c2())     ' 2^(k+1)*c1
```

用于实现公式中的“ $2^{k+1}*c$ ”运算。未经认真思考的人常会只写后面的两行，而且在计算万位以内的 π 值时也不会有问题。但当要算的位较多时， 2^{k+1} 将因超过短整数的表值范围 $2^{15}-1$ 而出错。目前调用一或二次短乘子程序 bmuli 的处理方法，使程序能算更多的位数。另一个算法子程序也有类似的情形。

7.2 π 值计算的主程序

1. 用于 DOS 下 QB 的主程序

下面的主程序可用于 QBasic 或 Quick BASIC 。

```
COMMON SHARED nmax, n
COMMON SHARED p AS INTEGER, pd2 AS INTEGER
COMMON SHARED l2p31 AS LONG, d2p53 AS DOUBLE
    .         ' here l2p31 = L2P31, not l2p31
COMMON SHARED powi AS INTEGER, powbd AS INTEGER
COMMON SHARED powl AS LONG
OPTION BASE 1
```

```
DIM SHARED pow(167) AS LONG
     ' the above lines are used for declaration
CLS
INPUT "Algorithm(1 or 2) = "; alg
INPUT "Number of digits = "; m
IF alg = 1 THEN
  CALL bcommon(m, "piagm.res")
  CALL a1piagm(m)
ELSE
  CALL bcommon(m, "pib4.res")
  CALL a2pib4(m)
END IF
PRINT "End of Pi job"
END
```

其中的第一段 (8 行) 是用于全局变量申明的标准语句，它们是多精度系统 BMP 规定的，这段内容一字不变地来自第 6 章 6.3.1 节。其后有两个键盘输入语句，用户应在提示下输入算法代号 (*Algorithm* = 1 表示使用 AGM 算法，2 表示用波尔温 4 阶收敛式) 以及要求计算的 π 值位数 (Number of digits)。当使用 AGM 算法时，程序调用 BMP 子程序 *bcommon* 给全局变量赋值的同时，还打开文件 *piagm.res* 为其后接受计算结果作准备。对于波尔温法，其文件名为 *pib4.res* 。

用户运行程序时，请首先把上节两个子程序和附录 A 中的全部 BMP 子程序附在这个主程序之后组成一个完整程序 (命名为 *qpi2.bas* , 注意别忘了 7.1.3 的 2. 中所说的 QB 与 VB 屏幕输出语句的取舍处理)，然后再使用如下命令进入 BASIC 系统：

```
QBASIC  qpi2    或    QB  qpi2
```

再往后的操作请参考第 3 章的最后一节。

2. 用于 Windows 下 Visual BASIC 的程序

在 Visual BASIC 中，上面的 QB 主程序变成如下的一个“工程”。此工程由两个 Private 子程序构成，一个用于“开始” (Begin)，另一个用于“结束” (End)，前者几乎就是 QB 的主程序，但键盘读入与屏幕输出的语句有差别。

```
Private Sub Command1_Click()
alg = InputBox("Algorithm(1 or 2) = ")
m = InputBox("Number of digits = ")
If alg = 1 Then
  Call bcommon(m, "c:\winpi\piagm.res")
  Call a1piagm(m)
Else
  Call bcommon(m, "c:\winpi\pib4.res")
  Call a2pib4(m)
End If
Text1.Text = Text1.Text & "End of Pi job"
End Sub

Private Sub Command2_Click()
End
End Sub
```

附属于这个工程的“模块”则由以下四部分组成：

- 语句 Attribute VB_Name = "Module1"
- 用于全局变量申明的语句；

- 上节的两个子程序;
- 附录 A 中的全部 BMP 子程序。

后三者实际上就是 QB 完整程序 *qpi2.bas* 中删去主程序自 CLS 开始的 12 行语句 (当然 QB 与 VB 屏幕输出语句的取舍处理中应改为选 VB 的)。此外, 8 行全局变量申明语句中的形式也要按如下原则修改:

COMMON SHARED 或 DIM SHARED 都改为 Global。

7.3 运算结果与运算时间

1. 程序输出例子

下面是当 $m = 1024$ 时波尔温算法子程序的输出结果:

```
digits = 1024   n = 258   nmax = 260
Algorithm B4
step = 0 / 5   11:39:22--11:39:36
step = 1 / 5   11:39:22--11:40:21  digits = 7
3.14159264

step = 2 / 5   11:39:22--11:41:07  digits = 40
3.1415926535 8979323846 2643383279 5028841971

step = 3 / 5   11:39:22--11:41:53  digits = 170
3.1415926535 8979323846 2643383279 5028841971 6939937510
  5820974944 5923078164 0628620899 8628034825 3421170679
  8214808651 3282306647 0938446095 5058223172 5359408128
  4811174502 8410270193 62
```

为节省篇幅, 此处有删节

```
step = 5 / 5   11:39:22--11:43:40  digits = 1024
```

```
3.1415926535 8979323846 2643383279 5028841971 6939937510
 5820974944 5923078164 0628620899 8628034825 3421170679
    ...                  ...                  ...
 5982534904 2875546873 1159562863 8823537875 9375195778
 1857780532 1712268066 1300192787 6611195909 2164201989
 3809525720 1065493285 0000
```

结果印有迭代步数 *step*、时间和该步所得的位数 *digits*。步数中分子是当步序号，分母是整个计算所需步数。印出的位数总是 4 的倍数 (与万进制对应)，其中的有效位数在 *digits* 后给出，但最后结果正确的位数比要求的 m 少。在百万位的计算中，损失位数不超过 11 位 (见表 7-2)。结果被同时送往屏幕与存入文件，两者的其主要差别，是当最终结果位数较多时，屏幕上有删节 (见第 5 步)，而文件上总是完整的 (见附录 B)。此外屏幕上不输出有关算法的第二行内容“ *Algorithm B4* ”。

另一个子程序亦有类似的结果输出，只是有关算法的第二行内容变为“ *Algorithm AGM* ”，而且各步的有效位数也不同。

2. 迭代步数与有效位数的关系

根据我们过去的计算经验， AGM 算法迭代各步实际获得的有效位数如表 7-1 所示。顺便指出，在 AGM 子程序中，我们自己构造了一个简单的算法来估计各步的有效位数 (见 FOR *kk* 循环) 以便在迭代之前就能估出 m 位计算所需的迭代次数。该估计从第 5 步开始，在百万位以内的各步估计值与实际值之差不超过 1 位 (仅在第 16, 18, 19 三步分别有 $+1$, -1, -1 的误差)。

波尔温 4 阶收敛式中迭代各步所得的有效位数已放入该

子程序的数组 $dg(11)$ 内. 它们也是根据我们过去的实际计算获得的。

表 7-1　AGM 方法迭代各步所得的有效位数

步数	1	2	3	4	5	6	7
位数	1	3	9	19	41	85	172
步数	8	9	10	11	12	13	
位数	347	696	1395	2792	5586	11175	
步数	14	15	16	17	18	19	
位数	22352	44706	89413	178829	357661	715323	

3. 作业时间

这里列出算术几何平均法 (AGM) 和波尔温 4 阶收敛公式 (B4) 的作业时间，它们来自 CPU 为 AMD-K6/233 的 PC 机。软件环境方面，我们使用了三种 BASIC 系统，它们分别是 VB, QB40 和 QBasic，其中 VB 是微软公司用于 Windows 95 的 Visual BASIC(5.0 中文企业版) 的简称；QB40 是 Microsoft 公司用于 DOS 的 Quick BASIC (4.0) 的简称。QBasic 来自 DOS 6.22 版。

我们这里给的运算位数都有 2^k 的形式，这是考虑到在快速乘法运算中具有最好的效率 (详见第 8 章)。在我们的计算中，当取 10000 位时，QBasic 出现存储器出界；取 40000 位时，QB40 也出界。因为这两种 BASIC 只用常规存储器，更大容量的存储器对它们没有意义。但 VB 依赖于存储器容量，在目前机器的标准配置中，算出上百万是不会有问题的。

在计算速度方面，QB40 编译执行最快，VB 编译执行

其次，QBasic 只能解释执行，所以最慢。由表 7-2 和表 7-3，在 32768 位计算中，QB40 与 VB 的速度比大约是 3 : 2；由表 7-3 和表 7-4，在 8192 位计算中，QB40 与 QBasic 的速度比超过 71 : 1 。

表 7-2　两个算法在 VB 编译执行中的作业时间

运算位数	32768	65536	131072	262144	524288	1048576
正确位数	32759	65527	131062	262134	524277	1048565
AGM 方法	7'38"	17'22"	38'03"	88'10"	197'02"	—
B4 方法	5'52"	14'16"	29'00"	70'39"	150'36"	360'35"

表 7-3　两个算法在 QB40 编译执行中的作业时间

运算位数	512	1024	2048	4096	8192	16384	32768
AGM 方法	2"	4"	10"	23"	56"	2'11"	4'57"
B4 方法	2"	3"	8"	20"	43"	1'49"	3'48"

表 7-4　两个算法在 QBasic(解释执行) 中的作业时间

运算位数	512	1024	2048	4096	8192
AGM 方法	1'57"	5'05"	12'21"	29'00"	66'50"
B4 方法	1'28"	4'18"	9'23"	24'22"	57'01"

顺便说一句，虽然某些 Quick BASIC 的新版本有更多的功能，但速度上可能比我们使用的 4.0 老版本还低。其实使用较为专业的 FORTRAN 语言会有更高的运算速度。例如使用 Microsoft 公司的 FORTRAN PowerStation (4.0) 实现 B4 算法，在几十万位以内的计算中，其作业时间仅为 VB 编译执行的 1/10 左右。但是正如第 3 章所述，本书选用 BASIC 语言是从科普角度来考虑的。

第 8 章　上亿位 π 值计算的新技术

本章内容是为有兴趣的读者简要介绍创世界纪录的上亿位 π 值计算中所使用的一些新技术。超长位数 π 值计算依赖着数学的进步和计算机的进步。其中高速收敛的数学公式方面，波尔温的 4 阶收敛式至今仍然是最快的一个，我们已在第 2 章 (见式 (2.19)) 作过介绍，并在上一章给出了相应的计算程序。使用该式作 16 步迭代就可以得到一百亿位以上的 π 值。本章将介绍另一个需要数学解决的问题，即超长尾数乘法运算的加速算法，它使用快速傅立叶变换 (FFT) 技术。此外，还介绍多个计算机 (或处理器) 并行计算以缩短作业时间的一些概念。最后讨论超高精度 π 值计算的现实意义。

8.1　提高超长位数乘法速度的方法 —— FFT

本节内容主要为学过高等数学的读者而写。我们首先介绍离散傅立叶变换、快速傅立叶变换和卷积计算，然后把它们用于超长尾数乘法运算，以提高其运算速度。

1. 离散傅立叶变换

高等数学中有一个称为离散傅立叶变换的数学工具，简称为 DFT 。它把长度为 n 的 (复数) 序列 $x=(x_0,x_1,\cdots,x_{n-1})$ 变换成另一个同长度的序列 $X=(X_0,X_1,\cdots,X_{n-1})$ 。变换公式是

$$X_k = \sum_{j=0}^{n-1} x_j w^{jk}, \quad k = 0, 1, \cdots, n-1 \tag{8.1}$$

式中 $w = e^{2\pi i/n}$, $i = \sqrt{-1}$。由复数表示方法，式 (8.1) 中的 X_k 也可以改写为

$$X_k = \sum_{j=0}^{n-1} x_j \cos\left(\frac{2\pi jk}{n}\right) - i \sum_{j=0}^{n-1} x_j \sin\left(\frac{2\pi jk}{n}\right) \tag{8.2}$$

傅立叶变换存在着逆变换 (简称逆 DFT)，它把序列 $X = (X_0, X_1, \cdots, X_{n-1})$ 变回 $x = (x_0, x_1, \cdots, x_{n-1})$。变换公式是

$$x_k = \frac{1}{n} \sum_{j=0}^{n-1} X_j w^{-jk}, \quad k = 0, 1, \cdots, n-1 \tag{8.3}$$

或把 x_k 改写为

$$x_k = \frac{1}{n} \sum_{j=0}^{n-1} X_j \cos\left(\frac{2\pi jk}{n}\right) + \frac{i}{n} \sum_{j=0}^{n-1} X_j \sin\left(\frac{2\pi jk}{n}\right) \tag{8.4}$$

傅立叶变换及其逆变换常常分别简记为

$$X = F(x), \quad x = F^{-1}(X) \tag{8.5}$$

2. 快速傅立叶变换

对于离散傅立叶变换式 (8.1)，人们常说它含有的运算量是 n^2 级的，用符号记为 $O(n^2)$。显然，这样说的时候是把一个复数乘法 $x_j w^{jk}$ 和一个复数加法合起来看作是一个运算。

自从 60 年代起，国际上出现了简称为 FFT 的快速傅立叶变换算法，如果长度 n 有 2 的幂的形式 (即 $n=2^m$)，此算法通过对式 (8.1) 的一系列分解和 $w^{n/2+p}=-w^p$ 等关系把计算变成为 m 次迭代，从而使运算量级从 $O(n^2)$ 下降为 $O(nm)=O(n\log_2 n)$。例如当 $n=2^{10}=1024$，则 $n\log_2 n=10\times 1024$，它大约是 n^2 的百分之一；又如当 $n=2^{20}=1048576$，则 $n\log_2 n$ 大约是 n^2 的五万分之一。而且当 n 越大，加速比越大。由于逆变换与正变换有着相同的算法核心，所以快速算法对逆变换同样适用。正是因为速度的跃升，与傅立叶变换有关的方法在近三十年来得到了广泛的应用。特别是因计算量太大，过去被认为只有理论价值的某些算法变为实际可行。随着对 FFT 研究的深入，被变换的序列长度可以不限于 2 的幂的形式，特别是对于长度为若干素数幂的乘积形式 (例如 $2^{p_1}3^{p_2}5^{p_3}$，其中 $p_j=0,1,2,\cdots$) 也有很好的效率。此外，还有一些专门对实数序列变换的有效算法。考虑到介绍 FFT 算法细节需占较多篇幅，而且 FFT 已有三十多年的历史，它在当今的科学工程计算、信号与图形处理等诸多方面起着极为关键的作用，详细介绍它的书籍已有很多，这里就不再展开了。

3. 卷积与快速傅立叶变换

傅立叶变换的重要应用之一是计算如下定义的卷积：

$$c(x,y)=(c_0(x,y),c_1(x,y),\cdots,c_{n-1}(x,y)) \tag{8.6}$$

式中

$$c_k(x,y)=\sum_{j=0}^{n-1}x_j y_{k-j},\quad k=0,1,\cdots,n-1 \tag{8.7}$$

这里假设序列是循环的，即 $y_{k-j}=y_{n+k-j}$。换句话说，当上式下标 $k-j<0$ 时，可理解为 $n+k-j$。所以这里定义的卷积也称为循环卷积。由卷积理论，有如下两个关系式

$$F[c(x,y)]=F(x)F(y) \tag{8.8}$$

$$c(x,y)=F^{-1}\{F[c(x,y)]\}=F^{-1}\{F(x)F(y)\} \tag{8.9}$$

也就是说，序列 x 与 y 的循环卷积 $c(x,y)$ 等于 x 与 y 分别作傅立叶变换之后相乘再取逆傅立叶变换。这个理论结果很早就被证明了，但它的运算量是 3 个傅立叶变换，按当时的看法是 $O(3n^2)$ 级，而直接使用式 (8.6) 的运算量只有 $O(n^2)$，所以当时谁都不会真正使用式 (8.9) 来计算卷积。但当 FFT 问世以后，情况改变了，每个傅立叶变换的运算量降为 $O(n\log_2 n)$，这时式 (8.9) 的运算量只有 $O(3n\log_2 n)$，在 n 较大时，它完全可能小于 $O(n^2)$。

4. 用 FFT 加速超长尾数的乘法运算

这里揭示，乘法运算实际上可以看作是一个卷积计算。下面只讨论乘法中最费时间的尾数相乘部分。设两个多精度数各有 t 位尾数：

$$a=(a_0,a_1,\cdots,a_{t-1}),\quad x=(b_0,b_1,\cdots,b_{t-1})$$

现仿照十进制竖式手算格式进行尾数乘法 (暂不考虑进位)。为方便起见，竖式中令 $t=4$(见下页)，式中的 c'_k 是它所在的那列之和，在该列的每一项中，a 和 b 两个下标之和总是

等于 k。于是可记为

$$c'_k = \sum_{j=0}^{k} a_j b_{k-j}, \quad k = 0, 1, \cdots, 2t-2 \tag{8.10}$$

$$\begin{array}{cccccccc}
 & & & & a_0 & a_1 & a_2 & a_3 \\
 & & & \times & b_0 & b_1 & b_2 & b_3 \\
\hline
 & & & & a_0b_3 & a_1b_3 & a_2b_3 & a_3b_3 \\
 & & & a_0b_2 & a_1b_2 & a_2b_2 & a_3b_2 & \\
 & & a_0b_1 & a_1b_1 & a_2b_1 & a_3b_1 & & \\
+) & a_0b_0 & a_1b_0 & a_2b_0 & a_3b_0 & & & \\
\hline
 & c'_0 & c'_1 & c'_2 & c'_3 & c'_4 & c'_5 & c'_6
\end{array}$$

式 (8.10) 与卷积定义式 (8.7) 相近，但有两点差异：其一是求和上限，这里是 k 而不是 $t-1$；其二是当此处 a_j 的下标大于 $t-1$ 时，a_j 的值应理解为 0，对于 b_{k-j} 也应作同样的理解。这样一来，如果想把乘法运算化为卷积计算，就需要用 0 来把尾数加长一倍。于是令 $n = 2t$，且

$$x = (x_0, x_1, \cdots, x_{n-1}) = (a_0, a_1, \cdots, a_{t-1}, 0, 0, \cdots, 0)$$

$$y = (y_0, y_1, \cdots, y_{n-1}) = (b_0, b_1, \cdots, b_{t-1}, 0, 0, \cdots, 0)$$

这时，容易验证

$$c'_k = c_k(x, y), \quad k = 0, 1, \cdots, 2t-2$$

即乘积中的各位 c'_k 可从卷积中的前 $n-1$ 项获得。换句话说，3. 中关于用 FFT 加快卷积计算的设想就变成为用 FFT 加速超长尾数乘法的技术。如 2. 所述，当尾数很长时，其运算速度可以提高千、万倍。本书多精度系统 BMP 的乘法运算子程

序 *bmul*，将根据尾数长短自动选用传统算法或 FFT 算法。并且对于后者，为了节省所需的存储空间和运算时间，我们采用了更为精巧和高效的算法以避免真正大量地补 0。

8.2 使用多台机器并行计算

对于许多需要长时间计算的问题，如果能使用多台机器(或处理器)并行计算，往往会加快整个作业的进程，对于超长位数 π 值的计算而言，这一点显得尤为重要。本节将以此为背景，简要介绍使用网络上的多台机器作并行计算问题。

8.2.1 Linux 操作系统与并行虚拟平台 PVM

1. PC 机上的 Linux 操作系统

这里所说的网络，既可以是由大型机或高档工作站群组成的计算机网络，也可以是由 PC 机群组成的微机网络，甚至是由大、小和型号不同机器构成的网络。至于每个机器上的操作系统，在大型机和工作站上自然是 UNIX，在 PC 机上我们使用 Linux，而不是 DOS 或 Windows。Linux 是个 32 位的系统，可以看作是 UNIX 的 PC 版，它的内核由芬兰人 Linus Torvalds 开发，而且它的源码是公开的，其后通过世界各地的电脑爱好者在 Internet 上参与共同开发和维护，系统日渐成熟。几乎所有重要免费软件已被移植到 Linux 上，并且也出现了在 Linux 上开发的商业软件。Linux 是个可以自由拷贝和免费使用的系统，可在 Internet 网上通过匿名 FTP 获得，例如使用如下网址：

```
ftp.cdrom.com:/pub/linux/slackware
```

2. 并行虚拟平台 PVM

要在网络上作并行计算，就得有一个相应的操作环境。PVM 是当今国际上十分流行的并行虚拟平台。PVM 实际上只是一个软件，它给用户提供一个可移植的异构编程环境。PVM 不仅得到大型机或高档工作站生产厂家的支持，而且在 PC 机上，它也可以通过 Linux 操作系统来运行。其实 BASIC，FORTRAN 等语言也是可移植的异构编程环境，因为用这些语言编制的计算程序可以在不同结构 (异构) 的机器上运行而几乎不需要作甚么修改 (可移植)。但是生产不同结构机器的厂家，需要自己或委托其他软件公司开发出支持他们机器的这些语言系统。PVM 与 FORTRAN 等语言的差别在于后者用于计算工作的编程，而前者用于机器之间信息交换与调度的编程。

PVM 可根据用户要求把网上的几台乃至上百台机器 (或处理器) 组成一台并行虚拟机，在这样的虚拟机上作计算，用户的感觉就象在一台“真正”的并行机上那样。值得注意的是，真正并行机的价格非常昂贵，一般学校与研究单位在经济上难以承受。但是有十多台高档微机的单位却是到处可见，如果他们的微机已经联网，那么要变成一台并行虚拟机就不再需要额外的投资了，因为作为软件的 PVM 同样是可以在网上自由下载和免费使用的，譬如使用匿名 FTP 在如下地址获得：

```
netlib2.cs.utk.edu
```

PVM 系统目前提供 C 语言与 FORTRAN 语言接口，对用户来说，实际上是提供了一个函数或子程序库。用户在使

用这些语言编制程序时，通过对 PVM 库函数的调用实现对虚拟机的控制，并实现进程调度和各机器之间的通信。有关例子请参考 8.2.3 节。

8.2.2 并行加速比与并行算法设计

建立了并行虚拟平台之后，下一步就是并行算法设计，其目标是把全部计算工作尽可能均衡地分配给组成此虚拟机的各台机器 (或处理器)，以便尽快地完成整个作业。并行算法的优劣可以用 **并行加速比** 来度量。设 t_p 是 p 个处理器并行计算完成作业所需的时间，则 p 个处理器的加速比定义为

$$S_p = \frac{t_1}{t_p} \tag{8.11}$$

式中 t_1 是并行计算中只用 1 个处理器完成整个作业的时间。有时也用老传统的单机串行计算所需的时间代替上式中的 t_1。由于单机串行计算的系统开销小于并行系统，所以两个定义是有差别的。另一方面，并不是所有计算工作都能同时进行的，为反映这一现实，设不能并行计算那部分所占的时间为 s(百分比)，则 p 个处理器所能达到的最大加速比是

$$\max S_p = s + (1 - s)p \tag{8.12}$$

为了简单，上式没有考虑并行计算中的通信和同步等时间开销。我们希望通过并行算法设计能使加速比尽可能大。这项工作有时并不费力气，有时却很困难，往往需要对已习惯了的单机串行计算概念的更新，甚至是在大量的数学推导工作之后才能做到。

8.2.3 基于 BBP 公式的 π 值并行计算程序

本书第 4 章 (4.2 节) 介绍了一个不需多精度系统支持的算 π 方法与程序。这个基于 BBP 公式的分段计算程序具有如下特点：

- 每次循环可得一段结果 (十六进制 8 位)；
- 计算第 l 段所需的时间与 l 成正比，从而计算前 l 段的时间与 $l(l+1)/2$ 成正比；
- 各次循环之间没有依赖关系，可并行计算。

如果要求使用 np 台机器来并行计算得到 L 段结果，并把第 $1 \sim l_1$ 段的计算任务分配给第 1 台机器，第 $l_1+1 \sim l_2$ 段分配给第 2 台机器 $\cdots\cdots$ 第 $l_{np-1}+1 \sim L$ 段分配给第 np 台机器，现在的问题是，这些 l_k, $k=1,2,\cdots,l_{np-1}$ 应取什么样的值才能获得最大的加速比，下面是在各台机器有相同速度的假设下给出问题的答案。因 L 段的全部计算时间与 $L(L+1)/2$ 成正比，为简单起见，在 L 较大时我们把时间近似为 L^2，这样一来，我们这里要做的实际上是把时间等分为 np 份：

$$l_1^2 = \frac{1}{np}L^2, \quad \text{即} \quad l_1 = \sqrt{\frac{1}{np}}L$$

$$l_2^2 - l_1^2 = \frac{1}{np}L^2, \quad \text{得} \quad l_2 = \sqrt{\frac{2}{np}}L$$

一般地，

$$l_k^2 - l_{k-1}^2 = \frac{1}{np}L^2, \quad \text{得} \quad l_k = \sqrt{\frac{k}{np}}L \tag{8.13}$$

1. BBP 并行计算程序

由于并行虚拟平台 PVM 不支持 BASIC，这里的程序只好改用 FORTRAN 语言。为突出并行处理内容，我们精简了程序，并且对并行计算中不需修改的子程序和函数只保留带有名字的首行而删去其他内容。我们相信，作此处理之后，并加上程序末尾的若干注释，即使没有接触过 FORTRAN 语言的读者也能大体理解并行计算是怎么一回事。

我们选用“主从”方式作并行计算，这时程序由“主”、“从”两个程序组成。“主”程序在一台机器上运行 (为方便称为“主”机)，其余的 $np-1$ 台“从”机都运行相同的“从”程序。“主”程序组织整个并行计算，包括给从机分配任务、传递参数，并接收从机送回来的结果，最后还输出全部结果。而“从”程序只管接受和完成分配给它的那几段 π 值计算任务，并把算出的结果送回主机。程序中凡是以字母 pvmf 开头的子程序都来自 PVM 子程序库，用以实现并行操作。

2. “主”程序列表

```
      PROGRAM     bbpm
      INCLUDE     '/usr/local/pvm3/include/fpvm3.h'
      INTEGER*4   tids(0:31),Lk(0:33)
      PARAMETER   (maxL=8150, lng=8)
      REAL*8      ss(maxL),wk(maxL),d16lng,s
c
      CALL pvmfmytid(mytid)
c1
      WRITE(*,'(a)')
     +  ' input Number of Processes and Iterations'
```

```
      READ(*,*) np, L
c                np <= 32,  L <= 8150
      OPEN(8,file='bbpf.res')
c2
      DO 10 k=1,np
        Lk(k)=sqrt(k/real(np))*L
   10 CONTINUE
      Lk(0)=0
      Lk(np+1)=0
c3
      CALL pvmfspawn('bbps',pvmdefault,'*',np-1,
     +     tids(1),ierr)
      DO 20 k=1,np-1
        CALL pvmfinitsend(pvmdefault,ibuf)
        CALL pvmfpack(integer4,Lk(k-1),2,1,ierr)
        CALL pvmfsend(tids(k),0,ierr)
   20 CONTINUE
c4
      d16lng=16.0d0**lng
      DO 30 m=Lk(np-1)+1,Lk(np)
        mlng=lng*(m-1)
        CALL bbp(mlng,s)
        ss(m)=dint(s*d16lng)/d16lng
   30 CONTINUE
c5
      DO 40 k=1,np-1
        CALL pvmfrecv(tids(k),1,ibuf)
        CALL pvmfunpack(real8,ss(Lk(k-1)+1),
     +       Lk(k)-Lk(k-1),1,ierr)
```

```
   40 CONTINUE

      CALL prtpi(ss,L,lng,s,wk)
c6          stop all slaves
      DO 50 k=1,np-1
        CALL pvmfinitsend(pvmdefault,ibuf)
        CALL pvmfpack(integer4,Lk(np),2,1,ierr)
        CALL pvmfsend(tids(k),0,ierr)
   50 CONTINUE
      CALL pvmfexit(ierr)
      END
ccccccccccccccccccccccccccccccccccccccccccccccccccc
      SUBROUTINE  bbp(n,s)
c       ... ... ...
      REAL*8 FUNCTION dmodulo(p)
c       ... ... ...
      SUBROUTINE  prtpi(d,m0,lng,s,e)
c       ... ... ...
```

3. “从”程序列表

```
      PROGRAM     bbps
      INCLUDE     '/usr/local/pvm3/include/fpvm3.h'
      INTEGER*4   tids(0:31),Lk(2)
      PARAMETER   (maxLk=2445, lng=8)
      REAL*8      sk(maxLk),d16lng,s
c11                    maxLk=0.3*maxL
      CALL pvmfmytid(mytid)
      CALL pvmfparent(tids(0))
c12
  10  CALL pvmfrecv(tids(0),0,ibuf)
```

```
      CALL pvmfunpack(integer4,Lk,2,1,ierr)
      IF(Lk(2).LE.0) THEN
        CALL pvmfexit(ierr)
        STOP
      ENDIF
c13
      d16lng=16.0d0**lng
      DO 30 m=Lk(1)+1,Lk(2)
        mlng=lng*(m-1)
        CALL bbp(mlng,s)
        sk(m-Lk(1))=dint(s*d16lng)/d16lng
   30 CONTINUE
c14
      CALL pvmfinitsend(pvmdefault,ibuf)
      CALL pvmfpack(real8,sk,Lk(2)-Lk(1),1,ierr)
      CALL pvmfsend(tids(0),1,ierr)
      GO TO 10
      END
ccccccccccccccccccccccccccccccccccccccccccccccccc
      SUBROUTINE  bbp(n,s)
c       ... ... ...
      REAL*8 function dmodulo(p)
c       ... ... ...
```

注 1：这里先对某些 FORTRAN 语言的成分作一些说明。程序中首行 PROGRAM 给程序命名；INCLUDE 是把其后指定的文件内容插入本程序，此处是 PVM 约定的内容；在变量与数组类型说明中，INTEGER*4 相当于长整数，REAL*8 相当于双精度实数；PARAMETER 指出后面的量是常数；循环 DO 10 k = … 10 CONTINUE 相当于 BASIC 中的 FOR

k = ··· NEXT k 。

注 2：程序定义的数组 *tids*(0 : 31) 是 PVM 约定的，数组大小与任务数目对应。这里最多不超过 32 个任务。在 PVM 中，一台机器可以完成多个任务，例如，当网上只有 9 台机器可用，而实际上又需要有 32 个任务时，PVM 将会自动地把这 32 个任务分配给 9 台机器，每台机器完成一个任务后再被分配另一个任务，直至它们把 32 个任务全都做完。PVM 还允许只有一台机器的情形，这样一来，用户就可以在一台甚至是不联网的机器上调试和模拟运行自己的并行程序。这里为叙述与解释简单起见，暂时约定一个机器只能作一个任务，于是机器与任务两个词等同，这时 *tids*(0) 对应着“主”机，其他元素与“从”机对应。两个程序的头一个执行语句都调用 *pvmfmytid*，以便向 PVM 报到和获得自己的任务标识符 (简记为 tid)。“从”程序中紧跟的调用 *pvmfparent* 用于获得“主”机的 *tid*，以确认主从关系。两个程序离开 PVM 时都应调用 *pvmfexit*。

注 3：“主”程序注解行 *c*1 之后的 FORTRAN 语句从键盘获得参数 *np*, *L*，前者指定用于并行计算的机器台数 (实为任务数，真正的机器配置通常是在程序运行前在 PVM 平台上给出的。) 后者是算 π 的段数。本程序约定 $np \leqslant 32$, $L \leqslant 8150$，后者约合十进制的 $8150 \times 8 \times 1.20412 = 78508$ 位。程序还打开用于存放结果的文件 *bbpf.res*。注解行 *c*2 之后的 FORTRAN 语句根据式 (8.13) 算出能使加速比最大的任务分配方案。

注 4：“主”程序注解行 *c*3 之后调用一系列 PVM 子程序，其中 pvmfspawn 用于生成新任务，亦即给从机布置任务。这里是所有的从机都执行同一个“从”程序 *bbps*。其实这里

的几个参数可以指出哪个程序仅在哪个机器上执行，因此在异构网络上进行复杂的并行计算时，指定不同的从机执行不同的程序是完全正常的。在紧跟其后的循环是给 $np-1$ 个从机依次发送数据，其中第一个子程序建立并初始化发送缓冲区；第二个把要发送的数据打包，这里是把 2 个 4 字节的整数 $Lk(k-1)$, $Lk(k)$ 打包 (数组中被取出的元素的下标间隔由第四个参数指定)，如果还有其他类型的数据，可多次调用这个子程序打包；子程序 *pvmfsend* 把缓冲区中已打包的数据发送到由第一个参数指出的那台从机，发送时还贴上由第二个参数给出的标签 (在不同场合发送含义不同的数据应有不同的标签，现在的标签是 0 。)

注 5：现在让我们把目光转到“从”程序。注解行 $c12$ 之前的语句已在注 1 和注 2 中作了说明，其后是调用 *pvmfrecv* 接收来自主机 $tids(0)$ 且标签为 0 的数据，且使用 *pvmfunpack* 把接收到的数据解包 (它是打包的逆过程) 之后送到第二个参数 Lk 中。如果收到的第二个数据 $Lk(2)$ 为 0，则退出 PVM；否则进行 $c13$ 至 $c14$ 之间的计算工作，即计算 π 的第 $Lk(1)+1$ 至 $Lk(2)$ 段结果 (根据“主”程序发送方式，第 k 个从机的 $Lk(1)=\sqrt{\frac{k-1}{np}}L+1$，$Lk(2)=\sqrt{\frac{k}{np}}L$。计算得到的结果 (即放在双精度数组 sk 内的 $Lk(2)-Lk(1)$ 个元素) 由 c14 后的程序打包并贴上标签 1 送回“主”机，此后从程序又转到前头等待接收新的任务。

注 6：现在回到“主”程序。它把数据打包送出之后 (见注解行 $c4$ 之前) 没有立即坐等“从”机送回结果，而是充分利用这段时间去计算本应由第 np 个从机承担的任务。注解行

c5 后面的循环是依次接收从机发回的结果 (标签为 1)，并在解包后按顺序放入数组 *ss* 内，以便其后的 FORTRAN 子程序 *prtpi* 统一译成十进制后输出。

注 7："主" 程序注解行 *c*6 开始是发命令让 $np-1$ 台从机依次退出 PVM，实际上，这里是重复 *c*4 之前的打包发送数据，只不过这时令第二个数据为 0。最后 "主" 机也退出 PVM，整个程序工作结束。

题外话：上面的例子虽然简单，但却让读者实实在在地看到在网络上作并行计算的一个完整例子，从而为电脑爱好者揭开了并行计算的神秘面纱。

一方面，这个例子确实太简单，以致可以在没有网络、没有并行平台的场合，由人工在 BASIC 环境下穿针引线地完成。譬如几个电脑爱好者可以合作"并行地"进行这项计算。首先把 4.2 节 BBP 分段算 π 的 BASIC 程序稍加修改 (只需删去最后那个用于输出的子程序和主程序中对它的两处调用，还应删去主程序中对于数组 *ss* 的说明和两处赋值。) 并且根据式 (8.13) 的原则合理分配各人该算的任务；然后各人用各人的机器同时计算各自承担的那几段 π 值；算出结果之后，各人把结果文件 (即程序自动产生的文件 *bbpblast*) 都拷贝到同一台机器上，拷贝时文件要改名 (例如依次改为 $r1$, $r2$, $r3$, $\cdots$) 以免名字冲突；最后是请合作者中的一位另编几行程序把这些结果文件依次读入到一个双精度数组中 (读入的同时，都应除以双精度形式的 16^8)，并调用前面曾删去的那个子程序 *prtpi* 把它们统一译成十进制后输出 (请参考原来的主程序)。

另一方面，国际上确实有人基于 BBP 公式在网络上并行算 π 的百亿，甚至万亿位 (当然是十六进制位)。但他们不是

“算前 x 位”，而是“只算第 x 位”（实际上是只算第 x 位及其后紧跟的少数几位，类似我们程序中只算由 8 位组成的某一段。）如前所述，算前 x 位的工作量与 x^2 成正比，而只算第 x 位的工作量只与 x 成正比。如 4.2 节的表 4-5 所示，我们曾使用该节的 BASIC 程序算出过 π 的 78510 十进制位，如果用这段时间只算第 x 位，则 x 将是数以亿计。

8.2.4 基于多精度系统的 π 值并行计算

本书介绍的所有算 π 公式都可以在多精度系统上实现，其中波尔温 4 阶公式具有最高的速度。第 7 章 7.1.2 节给出的程序是实现该算法的 BASIC 版本。这些快速公式都采用迭代法，且下一步要在上一步结束后才能开始。这意味着并行处理要通过多精度系统内的并行改造才能实现。但是要全面改造一个具有几十个过程，超过千行的系统要投入很多人力，是人们不情愿的事。作为可行的方案是，把精力集中在最能提高并行加速比的计算环节上。根据我们过去的单机串行计算经验，用波尔温 4 阶公式算 π，即使已经采用 FFT 技术加速，但全部乘法运算（包括把除法化为乘法，以及倒数、开方迭代中的乘法运算）在整个作业时间中所占的百分比仍然很高。例如计算 1 万位占 96%，在 6.5 万位时是 98.8%，且此百分比随计算长度增加。这表明，在超长位数的 π 值计算中，只对乘法作并行处理的效果与全面并行处理没有本质差别。因此多精度系统的并行改造顺理成章地简化为“并行只在乘法上安排”。

乘法的并行方案是“分段并行”，并使用“主从”模式运行。设段数为 m，每段长是 n 位十进制。以 $m=2$ 为例，这

时把乘数与被乘数分别记为

$$A = A_1 \times 10^n + A_2, \qquad B = B_1 \times 10^n + B_2$$

式中 A_i 或 B_j 都表示段，各含 n 位。结果 AB 的 $4n$ 位可分别由 4 个段间乘积 $(A_iB_j,\ i=1,2,\ j=1,2)$ 适当累加得到：

$$AB = A_1B_1 \times 10^{2n} + (A_1B_2 + A_2B_1) \times 10^n + A_2B_2$$

乘法分段并行处理中，“主”机上的程序把乘数与被乘数各分为 m 个等长的段 (若不能等分，可用零补尾)，并把 m^2 个段与段之间的乘法分配到各“从”机上运算 (各段间乘法用 FFT 实现)；其后从机把运算结果传回主机作累加，进位和规舍等。由于一个段间乘法在一个从机上执行，因此各从机的任务单一且负荷均匀。主机虽然负责所有其它的串行运算并在乘法中担负分段、发送及接收、累加等工作，但如上所述，其总的工作量不大 (但存储空间要求较大)，在各个从机作分段乘时，它无事可做，因此也可以留下一段乘法在主机上作业，以使主机与从机的负荷更为均匀些，这时所用的机器总数是 m^2 个。

8.3 超高精度 π 值计算的现实意义

圆周率 π 有极其广泛的应用，现实生活中凡与圆或球形有关的设计和计算都离不开 π。但是实际应用中通常只取十位左右，多的时候也不超过 20 位。有人形象地说，40 位就足以用来计算银河系的圆周而误差小于一个质子大小。人们不禁要问：算出千、万位，甚至几十亿位 π 值的意义何在？

在我们看来，超长 π 值计算最重要的价值在于**检测和显示计算机的性能**。这也是本书作者近几年开展 π 值并行计算研究的主要原因。我们把这个问题的讨论放在全书末尾，特别是放在具体给出超长 π 值计算程序和网络计算介绍之后，其目的是想通过数字把问题说得具体些和清楚些，从而给读者留下更深刻的印象。根据前面的介绍，基于多精度系统的 π 值计算，当位数很多时，所需的存储空间也很多，运算工作量也极大。如果使用网络并行计算，其信息交换量也很频繁和惊人，譬如使用第 7 章 7.1.2 节的波尔温算法子程序计算前 1 百万位，就需要 13.5MB 存储空间以及 1329.6 亿个双精度运算。如果要算 1 千万位，则相应的数字变为 135MB 和 17510.4 亿[①]。此外，还应加上计算机的系统软件和计算语言等的开销。在机器的存储空间不足时，还可以通过人工或系统软件自动的虚拟存储安排，使用大量的硬盘资源作周转。至于网络并行计算场合的信息交换量，以计算 π 值 1 千万位为例，采用 8.2.4 节的乘法分段方案，并设段数为 3，使用 9 台机器，那么一个 1 千万位乘法所需发出和接收的信息量都是 $9 \times \frac{10}{3} = 30$MB，如前面脚注所述，整个计算有 304 次这

① 譬如要算十进制的 n 位，这时存放这样的一个多精度数的短整型数组需 $n/2$ 字节. 波尔温算法子程序及多精度系统内各子程序共需 11 个这样的整型数组和 4 个双精度数组 (后者用于 FFT 计算，每个 $2n$ 字节)，总计 $13.5n$ 字节. 至于计算量方面，如 8.2.4 节所述，若计算超过几万位，多精度乘法计算时间占全部计算时间的 98% 以上，为简单起见，这里只统计多精度乘法次数 (倒数、除法和开方分别折合为 4, 5 和 7 次乘法). 这时准备阶段需 7 次乘法，迭代每步是 27 次. 在多精度乘法的 FFT 加速实现中 (见 8.1 节)，每个乘法含 3 次 FFT，而一次 FFT 大体上是 $n \log_2 n$ 个双精度复数的乘法和加法 (相当于 $8n \log_2 n$ 个双精度运算)，所以一次 n 位多精度乘法运算大体需要 $24n \log_2 n$ 个双精度运算. 记 k 为获得 n 位 π 所需的迭代次数，则算出 n 位 π 共需 $(7+27k)*24n \log_2 n$ 个双精度运算.

样的乘法，信息的总交换量是 $304 \times 2 \times 30 = 18240$MB 。已经是很可怕的计算工作量有时还要翻番，譬如当你不知道你所算的结果是否正确时，你就需要使用两个具有不同算法的程序 (例如 7.1.1 和 7.1.2 节的算术几何平均法和波尔温四阶法程序) 分别算出结果并相互比较，只有这两份结果完全相同时才能认为结果是可信的。π 值计算程序还有一个很重要的特点，即在漫长计算过程中的任何一点错误都将导致最终不能获得正确的结果。总而言之，通过上面的介绍可以看出，能算出足够多位 π 值的程序可以用来测试机器的各个部件和网络，这种测试具有长时间、满负荷、高强度、对错误敏感和考核全面等优点。例如 1986 年，一个算 π 程序确实发现了某超级计算机中的一台有硬件上的问题。目前算 π 程序业已成为计算机的一个标准测试软件。

另一方面，一台计算机在合理的时间内正确地算出 π 值位数的多少也反映了该机硬件和软件系统的可靠性以及该机的综合计算能力。加之人们从学生时代就与 π 打交道，对它的计算有亲切感，所以容易为人们津津乐道。可能是这个原因，日本的日立公司多年来与东京大学的 Kanada 等人合作，在日立公司的每种新型号超级计算机面世时都拿算 π 程序来测试，并屡创算 π 长度的世界纪录 (见本书第 2 章的表 2-1)，收到明显的宣传效果。当今计算技术飞速发展，各种超级计算机相继问世，并且竞争日益激烈。算 π 程序的上述价值，有力地促进了 π 值计算方法的研究并取得令人震惊的成果。计算机技术的进步又使得算 π 如虎添翼，算得的位数大幅度攀升。如 20 世纪 70 年代二阶收敛的算术几何平均法使算出的 π 的位数突破千万位， 20 世纪 80 年代的波尔温四阶收敛

公式使纪录超过亿位大关。进入 20 世纪 90 年代，计算机技术的飞跃，上千个处理器的超级并行机的出现，又使 π 的纪录攀升到 64 亿位。1997 年正式发表的可从 π 的任一 (十六进制) 位开始计算的 BBP 公式，使人们可以算出 π 在第百亿位乃至第万亿位之后的少数几位 (十六进制，见第 4 章 4.2 节)。

谈到 **推动 π 值计算研究的数学方面原因** 时，有些数学家认为可能是人们想搞清楚 π 的各位的值是不是随机的。在 18 世纪 60 年代从数学上严格证明 π 是个无理数之前，一些人辛辛苦苦地计算几十位，甚至一百位 π 值的原因往往与他想知道 π 值是否会重复有关。如果重复了，π 就是个有理数。在 π 被理论证明是个无理数之后，大家知道它是不能被除尽的了。但是仍有一个问题未得到回答，这就是无穷多位的 π 中，各位的值是不是随机的？这又导致一些数学家去算 π，在电脑出现以前，已有人算出正确的 527 位。电脑发明以后，计算更为方便了。据文献记载，1988 年 Bailey 在算出 2936 万位的同时，使用统计方法对这些位进行了较全面的随机性检验，他的最后结论是这 2936 万位的值是“完全随机的”。

下面我们讲述一个发生在 1998 年的有趣故事作为 π 是不是一个无理数的脚注。据某报纸报道，1998 年 6 月，国外有个电脑神童用 25 台电脑计算，据称发现 π 在小数后第 12500 亿位可以除尽云云。记者为此采访了国内的数学家，但数学家们认为这是“绝不可能的”，因为 π 是无理数这一结论业已被严格的数学理论证明。要不是神童的程序有问题，就是机器的软件或硬件有问题。我们当然同意数学家们的看法。此外，从 π 值的计算角度看，神童的结论也是不能成立的。首先从目前已知的计算公式和计算机条件看，不具备计算 π 的前

12500 亿位的条件 (读者可根据前两段的方法算出此计算所需的存储空间与计算量便知) 。神童即使进行计算，也只能是用 BBP 公式只计算 π 的第 12500 亿位及其后少数几位 (例如十位左右，这时计算复杂性明显下降，几乎无需存储空间) 。但是 BBP 公式 (见第 2 或 4 章) 是个无穷和式：

$$\pi = t_0 + t_1 + t_2 + \cdots + t_k + \cdots$$

式中通项

$$t_k = \frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right)$$

虽然 t_k 的值越到后面越小，但它总不会是 0，从而不可能有“除尽”的时候。

现在让我们把话题重新回到 π 值计算的现实意义上来。如上所述，算 π 程序所具有的检测和显示计算机性能的实用价值，加上数学自身的动力促进了 π 值的算法研究。与此同时，**π 值的算法研究也带动了相关的计算方法研究**并取得重要的成果。例如波尔温在 80 年代得到 π 的四阶收敛公式的同时，还发现了对某些基本常数构造高阶收敛公式的一般技术。顺便说一句，在其他计算领域中，如此高阶的收敛公式也不多见。同样，BBP 作者们在 90 年代得到算 π 公式的同时，还得到诸如 π^2, $\log(2)$, $\log^2(2)$, $\log(9/10)$ 等数学常数的计算公式。这些公式的特点是可以只算某一位，而不必算出它前头各位，从而打破了千百年来的计算传统，得到了数学界的高度评价。

对于已经算出的 π 的千千万万位数字本身而言，恐怕也不是一点用处都没有的。过去已有人把这些数字**作为随机数**

来用。如前所述，在较全面的统计检验未发现这样做有何不妥之后，人们用起来就会更放心。随机数在求解某些数学问题(例如在计算机上随机搜索数学方程的解，粒子输运方程中的蒙特卡洛方法等)、保密通信、环境模拟，甚至电子游戏程序的编制等方面都很有用。

附录 A
多精度运算系统 BMP 源程序

本附录给出任意精度运算系统 BMP 的全部 25 个子程序和 3 个函数的 BASIC 程序编码。它们的总长为 1073 行，各子程序按字母顺序排列，依次是

1. badd	2. bcommon	3. bcomp	4. bcopy
5. bdiv	6. bdivi	7. bdtom	8. binput
9. binv	10. bmtod	11. bmul	12. bmul1
13. bmul2	14. bmuli	15. bprint	16. bprt50
17. bsqrt	18. bsub	19. fftc	20. fftr2
21. fftrl	22. tadd	23. tcarry	24. tfit2
25. tnorm	26. tpowa&	27. tpower%	28. tsub

各子程序的功能已在正文 (第 6 章及第 7 章) 中作过介绍，在此不再重复。此外，考虑到本书版面窄小的实际情况，程序已作了适当的调整，尽可能使用短名字和短语句以减少换行的麻烦，例如把一个长语句 (如 DIM 和 PRINT 等) 拆成两或三个，实在不便拆开的则按 BASIC 的换行规则处理 (即行尾加上由空格和下划线 " _" 构成的换行符)，但后者在 QBasic 或 Quick BASIC 读入时将自动恢复为一个长行。

我们要求本书给出的任意精度运算系统 BMP 的全部子程序可以分别在 DOS 下的 QBasic 或 Quick BASIC (统称 QB) 以及 Windows 下的 Visual BASIC (简称 VB) 中运行，所以特

别留意所使用的语句的通用性。对于无法统一的屏幕输出语句 PRINT 则使用同时并列，并加上注解的处理方法。例如内容为“QB 01”的注解表示其后的 1 行只用于 QB，而“VB 04”则是其后的 4 行用于 VB。因此在程序运行前，用户必须删去其中不需要的语句。为了避免人工查找和删除可能带来的错误，也可使用如下一个简单的自动处理程序：

```
OPEN "bmp.bas" FOR INPUT AS #3
INPUT "1 (QB) or 2 (VB) = "; qv
IF qv = 1 THEN
  e$ = "Q"
  OPEN "qbbmp.bas" FOR OUTPUT AS #4
ELSE
  e$ = "V"
  OPEN "vbbmp.bas" FOR OUTPUT AS #4
END IF

FOR k = 1 TO 2000
LINE INPUT #3, c$
IF LEN(c$) = 7 AND (LEFT$(c$, 5) = "' QB " OR _
  LEFT$(c$, 5) = "' VB ") THEN
  d$ = MID$(c$, 3, 1)
  m = VAL(MID$(c$, 6, 2))
  FOR i = 1 TO m
    LINE INPUT #3, c$
    IF d$ = e$ THEN PRINT #4, c$
  NEXT i
ELSE
  PRINT #4, c$
END IF
IF EOF(3) THEN EXIT FOR
NEXT k
END
```

被处理的程序放在文件 *bmp.bas* 内. 因为第 7 章内两个算 π 子程序也要作同样的处理, 所以最好也把它们与这里的 BMP 子程序放在一起处理. 在运行此程序时, 屏幕有如下提示:

```
1 (QB) or 2 (VB) =
```

若用 1 回答, 则在文件 *qbbmp.bas* 中得到用于 QB 的程序; 若用 2 回答, 则在文件 *vbbmp.bas* 中得到用于 VB 的程序.

BMP 全部子程序编码

```
REM *** *** c=a+b
SUB badd (a() AS INTEGER, b() AS INTEGER, _
    c() AS INTEGER)
IF a(1) = 0 THEN
  CALL bcopy(b(), c())
ELSEIF b(1) = 0 THEN
  CALL bcopy(a(), c())
ELSE
  CALL bcomp(a(), b(), r, m)
  IF r = 1 THEN
    IF a(1) * b(1) > 0 THEN
      CALL tadd(a(), b(), c())
      c(1) = a(1)
    ELSE
      CALL tsub(a(), b(), c())
      c(1) = a(1) * SGN(c(3))
    END IF
  ELSEIF a(1) * b(1) > 0 THEN
    CALL tadd(b(), a(), c())
    c(1) = b(1)
  ELSE
    CALL tsub(b(), a(), c())
```

```
    c(1) = b(1) * SGN(c(3))
  END IF
  CALL tnorm(c())
END IF
END SUB

REM *** *** set common values
SUB bcommon (m, cfn$)
n = (m + 3) \ 4 + 2
nmax = n + 2
p = 10000
pd2 = p / 2
l2p31 = 2147483647              ' = 2^31 - 1
d2p53 = 9007199254740991#       ' = 2^53 - 1
powi = 1
powbd = 32600
OPEN cfn$ FOR OUTPUT AS #2
PRINT #2, "digits ="; m; "  n ="; n; _
  "  nmax ="; nmax
' QB 01
PRINT "digits ="; m; "  n ="; n; "  nmax ="; nmax
' VB 04
Form1.Text1.Text = Form1.Text1.Text & "digits = " _
  & m & "  n = " & n & "  nmax = " & nmax & _
  Chr$(13) & Chr$(10)
Form1.Text1.Refresh
END SUB

REM *** *** r=0,1,2 if |a|=|b|, |a|>|b|, |a|<|b|
SUB bcomp (a() AS INTEGER, b() AS INTEGER, r, m)
r = 0
m = n + 1
```

```
c2% = tpower%(a(2), b(2), c2%, -2) ' powl=a(2)-b(2)
IF powl <> 0 THEN
  m = 2
  IF powl > 0 THEN
    r = 1
  ELSE
    r = 2
  END IF
ELSE
  FOR i = 3 TO n
    IF a(i) > b(i) THEN
      r = 1
      m = i
      EXIT FOR
    ELSE
      IF a(i) < b(i) THEN
        r = 2
        m = i
        EXIT FOR
      END IF
    END IF
  NEXT i
END IF
END SUB

REM *** *** b=a
SUB bcopy (a() AS INTEGER, b() AS INTEGER)
FOR i = 1 TO n
  b(i) = a(i)
NEXT i
END SUB

REM *** *** c=a/b
SUB bdiv (a() AS INTEGER, b() AS INTEGER, _
```

```
    c() AS INTEGER)
REDIM w(n + 2) AS INTEGER
CALL binv(b(), w())
CALL bmul(a(), w(), c())
END SUB

REM *** *** c=a/b, where b is a integer variable
SUB bdivi (a() AS INTEGER, b AS INTEGER, _
    c() AS INTEGER)
DIM lb AS LONG, lp AS LONG, lw AS LONG, lt AS LONG
IF a(1) = 0 THEN
  CALL bdtom(0#, c())
  EXIT SUB
END IF
lb = ABS(b)
IF lb = 0 THEN
' QB 01
  PRINT "b = 0 in a/b,  see SUB bdivi"
' VB 04
  Form1.Text1.Text = Form1.Text1.Text & _
    "b = 0 in a/b,  see SUB bdivi" & _
    Chr$(13) & Chr$(10)
  Form1.Text1.Refresh
  STOP
END IF
kk% = c(2)
lp = p
sb = SGN(b)
lt = 0
FOR i = 3 TO n
  lw = a(i) + lt
  c(i) = INT(lw / lb)
```

```
  lt = (lw - lb * c(i)) * lp
NEXT i
ib = 0
FOR i = 3 TO n
  IF c(i) > 0 THEN EXIT FOR
  ib = ib + 1
NEXT i
IF ib > 0 THEN
  ib3 = ib + 3
  FOR i = ib3 TO n
    c(i - ib) = c(i)
  NEXT i
END IF
FOR i = n - ib + 1 TO n + 1
  c(i) = INT(lt / lb)
  lt = (lt - lb * c(i)) * lp
NEXT i
c2% = powbd + 1
pow(1) = ib
c(2) = tpower%(a(2), c2%, kk%, 2) ' c(2) = a(2)-ib
c(1) = a(1) * sb
IF c(n + 1) > pd2 THEN c(n) = c(n) + 1
CALL tnorm(c())
END SUB

REM *** *** b=d, where d is a double variable
SUB bdtom (d AS DOUBLE, b() AS INTEGER)
DIM dp AS DOUBLE, dd AS DOUBLE, dw AS DOUBLE
FOR i = 1 TO n
  b(i) = 0
NEXT i
IF d = 0 THEN EXIT SUB
dp = p
```

```
b(1) = SGN(d)
dd = ABS(d)
IF INT(dd) = dd AND dd <= d2p53 THEN
  FOR i = n TO 3 STEP -1
    IF dd = 0 THEN EXIT FOR
    b(2) = b(2) + 1
    dw = INT(dd / dp)
    b(i) = dd - dw * dp
    dd = dw
  NEXT i
  k = b(2)
  FOR i = 1 TO k
    b(2 + i) = b(n - k + i)
    b(n - k + i) = 0
  NEXT i
ELSE
  DO WHILE dd >= 1
    dd = dd / dp
    b(2) = b(2) + 1
  LOOP
  DO WHILE dd < 1 / dp
    dd = dd * dp
    b(2) = b(2) - 1
  LOOP
  n1 = INT(LOG(d2p53) / LOG(dp)) + 4
  IF n1 > n THEN n1 = n
  FOR i = 3 TO n1
    dw = INT(dd * dp)
    b(i) = dw
    dd = dd * dp - dw
  NEXT i
END IF
END SUB
```

```
REM *** *** input a(1) to a(m) from #1
SUB binput (a() AS INTEGER, m)
FOR i = 1 TO m
  INPUT #1, a(i)
NEXT i
END SUB

REM *** *** x=1/a
SUB binv (a() AS INTEGER, x() AS INTEGER)
REDIM w(n + 2) AS INTEGER, two(n + 2) AS INTEGER
DIM dx AS DOUBLE, a2 AS INTEGER
IF a(1) = 0 THEN
' QB 01
  PRINT "a=0 in 1/a,  see SUB binv"
' VB 04
  Form1.Text1.Text = Form1.Text1.Text & _
    "a=0 in 1/a,  see SUB binv" & _
    Chr$(13) & Chr$(10)
  Form1.Text1.Refresh
  STOP
ELSE
  IF ABS(a(2)) < 76 THEN
    CALL bmtod(a(), dx)
    dx = 1 / dx
    CALL bdtom(dx, x())
  ELSE
    a2 = a(2)
    a(2) = 0
    CALL bmtod(a(), dx)
    a(2) = a2
    dx = 1 / dx
```

```
    CALL bdtom(dx, x())
    x(2) = tpower%(x(2), a2, c2%, 2)   ' x(2)=x(2)-a2
  END IF
  CALL bdtom(2#, two())
  nn = n
  n1 = 2
  DO WHILE n1 < nn - 2
    n1 = n1 + n1
    n = n1 + 4
    IF n > nn THEN n = nn
    CALL bmul(a(), x(), w())
    CALL bsub(two(), w(), w())
    CALL bmul(x(), w(), x())
  LOOP
END IF
END SUB

REM *** *** b=a, where a is a double variable,
SUB bmtod (a() AS INTEGER, d AS DOUBLE)
DIM dp AS DOUBLE
d = 0
c2% = tpower%(a(2), 1, c2%, -2)   ' powl = a(2) - 1
IF powl < -77 THEN EXIT SUB
IF a(1) <> 0 THEN
  dp = p
  FOR i = n TO 3 STEP -1
    d = d / dp + a(i)
  NEXT i
  d = a(1) * d * dp ^ powl
END IF
END SUB

REM *** *** c=a*b  two methods
```

```
SUB bmul (a() AS INTEGER, b() AS INTEGER, _
    c() AS INTEGER)
IF n < 50 THEN
  CALL bmul1(a(), b(), c())
ELSE
  CALL bmul2(a(), b(), c())
END IF
END SUB

REM *** *** c=a*b  (direct method)
SUB bmul1 (a() AS INTEGER, b() AS INTEGER, _
    c() AS INTEGER)
nn2 = n + n
REDIM d(nn2) AS LONG
DIM dp AS LONG, dw AS LONG
IF a(1) * b(1) = 0 THEN
  CALL bdtom(0#, c())
  EXIT SUB
END IF
dp = p
k2 = INT(l2p31 / (dp * dp)) - 1
n1 = nn2 - 2
FOR i = 1 TO n1
  d(i) = 0
NEXT i
k1 = n - 2
FOR i = 3 TO n
  dw = a(i)
  FOR j = 1 TO k1
    d(i + j) = d(i + j) + CLNG(b(j + 2)) * dw
  NEXT j
  IF i = n OR (i MOD k2) = 0 THEN
    n1 = n + i + 1
```

```
    n2 = n1 + 1
    n3 = n1 - 3
    FOR k = 4 TO n3
      dw = INT(d(n2 - k) / p)
      d(n1 - k) = d(n1 - k) + dw
      d(n2 - k) = d(n2 - k) - dw * dp
    NEXT k
  END IF
NEXT i
c(2) = tpower%(a(2), b(2), c2%, 1)   ' c(2)=a(2)+b(2)
IF d(3) = 0 THEN
  IF d(n + 2) >= pd2 THEN d(n + 1) = d(n + 1) + 1
  n1 = n + 1
  FOR i = 4 TO n1
    d(i - 1) = d(i)
  NEXT i
  c(1) = a(1) * b(1)
  c(2) = tpower%(c(2), 1, c(2), 2)    ' c(2)=c(2)-1
ELSE
  IF d(n + 1) >= pd2 THEN d(n) = d(n) + 1
  c(1) = a(1) * b(1)
END IF
FOR i = 3 TO n
  c(i) = d(i)
NEXT i
CALL tnorm(c())
END SUB

REM *** *** c=a*b    (FFT method)
SUB bmul2 (a() AS INTEGER, b() AS INTEGER, _
    c() AS INTEGER)
IF a(1) * b(1) = 0 THEN
  CALL bdtom(0#, c())
```

```
  EXIT SUB
END IF
n1 = n - 2
k = tfit2(n1)                    ' k = 2^m >= n1
REDIM ar(k) AS DOUBLE, br(k) AS DOUBLE
REDIM ai(k) AS DOUBLE, bi(k) AS DOUBLE
DIM dj AS DOUBLE, dp AS DOUBLE, dw AS DOUBLE
DIM pi AS DOUBLE, pa AS DOUBLE, pb AS DOUBLE
DIM ss AS DOUBLE, cc AS DOUBLE, s1 AS DOUBLE
DIM c1 AS DOUBLE, arn AS DOUBLE, brn AS DOUBLE
DIM ar1 AS DOUBLE, ar2 AS DOUBLE, br1 AS DOUBLE
DIM br2 AS DOUBLE, ain AS DOUBLE, bin AS DOUBLE
DIM ai1 AS DOUBLE, ai2 AS DOUBLE
DIM bi1 AS DOUBLE, bi2 AS DOUBLE
                                 ' copy double array
FOR i = 1 TO n1
  ar(i) = a(i + 2)
  br(i) = b(i + 2)
NEXT i
IF n1 < k THEN
  FOR i = n1 + 1 TO k
    ar(i) = 0#
    br(i) = 0#
  NEXT i
END IF
                                 ' FFT,  length 2^m
dp = p
pi = 3.141592653589793#
n2 = 2 * k
nd2 = k / 2
nd21 = nd2 + 1
pb = 2# * pi / n2
ss = 0#
```

```
cc = 1#
s1 = SIN(pb)
c1 = COS(pb)
isc = 0
FOR j = 1 TO k
  ai(j) = ar(j) * cc - br(j) * ss
  bi(j) = ar(j) * ss + br(j) * cc
  pa = ss
  ss = ss * c1 + cc * s1
  cc = cc * c1 - pa * s1
  isc = isc + 1
  IF isc = 8 THEN
    isc = 0
    pa = j * pb
    ss = SIN(pa)
    cc = COS(pa)
  END IF
NEXT j
CALL fftc(k, 1, ar(), br())
CALL fftc(k, 1, ai(), bi())
                                    ' inverse FFT
FOR j = 2 TO nd2
  arn = ar(k + 2 - j)
  brn = br(k + 2 - j)
  ain = ai(k + 1 - j)
  bin = bi(k + 1 - j)
  ar1 = ar(j) + arn
  bi1 = ar(j) - arn
  ar2 = ai(j) + ain
  bi2 = ai(j) - ain
  br1 = br(j) + brn
```

```
  ai1 = br(j) - brn
  br2 = bi(j) + bin
  ai2 = bi(j) - bin
  ar(j) = (ar1 * br1 + ai1 * bi1) * .25#
  br(j) = (ar2 * br2 + ai2 * bi2) * .25#
  ai(j) = (ai1 * br1 - ar1 * bi1) * .25#
  bi(j) = (ai2 * br2 - ar2 * bi2) * .25#
NEXT j
ar1 = 2# * ar(1)
ar2 = ai(1) + ai(k)
br1 = 2# * br(1)
br2 = bi(1) + bi(k)
ai2 = bi(1) - bi(k)
bi2 = ai(1) - ai(k)
ar(1) = ar1 * br1 * .25#
br(1) = (ar2 * br2 + ai2 * bi2) * .25#
ai(1) = ar(nd21) * br(nd21)
bi(1) = (ai2 * br2 - ar2 * bi2) * .25#

FOR j = nd2 TO 1 STEP -1
  ar(2 * j) = br(j)
  ar(2 * j - 1) = ar(j)
  ai(2 * j) = bi(j)
  ai(2 * j - 1) = ai(j)
NEXT j
ai(nd21) = -ai(nd21)
CALL fftrl(k, -1, ar(), ai())
                                    ' carry
ERASE br, bi
REDIM pr(n2) AS INTEGER
```

```
dj = 0
FOR j = k TO 1 STEP -1
  dw = INT(ar(j) / 2# + dj + .5#)
  IF dw < dp THEN
    pr(2 * j) = dw
    dj = 0
  ELSE
    dj = INT(dw / dp)
    pr(2 * j) = dw - dj * dp
  END IF
  IF j = 1 THEN EXIT FOR
  dw = INT(ai(j - 1) / 2# + dj + .5#)
  IF dw < dp THEN
    pr(2 * j - 1) = dw
    dj = 0
  ELSE
    dj = INT(dw / dp)
    pr(2 * j - 1) = dw - dj * dp
  END IF
NEXT j
pr(1) = dj
                ' normalization -- n elements only
c(2) = tpower%(a(2), b(2), c2%, 1)   ' c(2)=a(2)+b(2)
IF pr(1) = 0 THEN
  IF pr(n) >= pd2 THEN pr(n - 1) = pr(n - 1) + 1
  FOR i = 3 TO n
    c(i) = pr(i - 1)
  NEXT i
  c(2) = tpower%(c(2), 1, c(2), 2)    ' c(2)=c(2)-1
ELSE
  IF pr(n - 1) >= pd2 THEN pr(n - 2) = pr(n - 2) + 1
```

```
  FOR i = 3 TO n
    c(i) = pr(i - 2)
  NEXT i
END IF
c(1) = a(1) * b(1)
IF c(3) = p THEN
  c(3) = 1
  c(2) = tpower%(c(2), 1, c(2), 1)    ' c(2)=c(2)+1
END IF
CALL tnorm(c())
END SUB

REM *** *** c=a*b, where b is a integer variable
SUB bmuli (a() AS INTEGER, b AS INTEGER, _
    c() AS INTEGER)
DIM a2 AS INTEGER, c2 AS INTEGER, sb AS INTEGER
DIM lb AS LONG, lp AS LONG
DIM lw AS LONG, lt AS LONG
IF a(1) = 0 OR b = 0 THEN
  CALL bdtom(0#, c())
  EXIT SUB
END IF
cc% = c(2)
lb = ABS(b)
lp = p
a2 = a(2)
sb = SGN(b)
a(2) = 0
lt = 0
FOR i = n TO 2 STEP -1
  lw = a(i) * lb + lt
  IF lw > lp THEN
```

```
    lt = INT(lw / lp)
    c(i) = lw - lp * lt
  ELSE
    lt = 0
    c(i) = lw
  END IF
NEXT i
c(1) = a(1) * sb
c2 = c(2)
a(2) = a2
c(2) = c2
sb = 0
IF (c(2) <> 0) THEN
  IF c(n) >= pd2 THEN c(n - 1) = c(n - 1) + 1
  FOR i = n TO 3 STEP -1
    c(i) = c(i - 1)
  NEXT i
  sb = 1
END IF
CALL tcarry(c(), n)
c(2) = tpower%(a2, sb, cc%, 1)    ' c(2) = a2+sb
CALL tnorm(c())
END SUB

REM *** *** print a(1) to a(m) from #2,  k/line
SUB bprint (cc$, a() AS INTEGER, m, k)
DIM ct AS STRING * 5
ct = cc$
ka = ABS(k)
IF a(2) <= powbd THEN
  a2 = a(2)
ELSE
```

```
  a2 = pow(a(2) - powbd)
END IF
PRINT #2, ct; "   "; a(1); "   "; a2
' QB 01
IF k < 0 THEN PRINT ct; "   "; a(1); "   "; a2
' VB 03
If k < 0 Then Form1.Text1.Text = Form1.Text1.Text _
  & ct & "   " & a(1) & "   " & a2 & _
  Chr$(13) & Chr$(10)
j = INT((m + ka - 3) / ka)
FOR i = 1 TO j
  ik = i * ka + 2
  IF i = j THEN ik = m
  FOR ii = (i - 1) * ka + 3 TO ik
    cp$ = " " + RIGHT$(STR$(a(ii) + 10000), 4)
    PRINT #2, cp$;
' QB 01
    IF k < 0 THEN PRINT cp$;
' VB 02
    If k < 0 Then Form1.Text1.Text = _
      Form1.Text1.Text & cp$
  NEXT ii
  PRINT #2, " "
' QB 01
  IF k < 0 THEN PRINT " "
' VB 02
  If k < 0 Then Form1.Text1.Text = _
    Form1.Text1.Text & " " & Chr$(13) & Chr$(10)
NEXT i
END SUB

REM *** *** print a(1) to a(m) from #2,  50dgs/line
```

```
SUB bprt50 (cc$, a() AS INTEGER, m, sc)
ma = ABS(m)
ka = LEN(cc$)
IF MID$(cc$, ka, 1) = "." THEN
  m1 = 3
  ca3$ = STR$(a(1) * a(3))
  kb = LEN(ca3$)
  IF ka > 5 THEN ka = 5
  IF kb > ka - 1 THEN kb = ka - 1
  chead$ = RIGHT$(cc$, ka)
  MID$(chead$, ka - kb, ka - 1) = RIGHT$(ca3$, kb)
  MID$(chead$, ka, 1) = "."
  kh = ka
ELSE
  kh = 2
  m1 = 2
  chead$ = SPACE$(kh)
  IF a(2) <= powbd THEN
    a2 = a(2)
  ELSE
    a2 = pow(a(2) - powbd)
  END IF
  PRINT #2, cc$; "   "; a(1); "   "; a2
' QB 01
  IF sc <> 0 THEN PRINT cc$; "   "; a(1); "   "; a2
' VB 03
  If sc <> 0 Then Form1.Text1.Text = _
    Form1.Text1.Text & cc$ & "   " & a(1) & "   " _
    & a2 & Chr$(13) & Chr$(10)
END IF
ls = INT((ma - m1 + 24) / 25)
jp = 0
```

```
IF m < 0 THEN
  IF ls <= 3 THEN
    jp = 1
  ELSE
    jp = ls - 1
  END IF
END IF
m2 = 25 - m1
j25 = 25
FOR i = 1 TO ls
  IF i = 1 OR i >= jp THEN
    c100$ = SPACE$(100)
    IF i = ls THEN j25 = ma - m1 - 25 * (i - 1)
    FOR j = 1 TO j25
      ca$ = STR$(a(25 * i + j - m2) + 10000)
      MID$(c100$, j * 4 - 3, 4) = RIGHT$(ca$, 4)
    NEXT j
    FOR jj = 0 TO 1
      PRINT #2, chead$;
' QB 01
      IF sc <> 0 THEN PRINT chead$;
' VB 02
      If sc <> 0 Then Form1.Text1.Text = _
        Form1.Text1.Text & chead$
      FOR j = 0 TO 4
        jjj = 50 * jj + 10 * j + 1
        IF j < 4 THEN PRINT #2, _
          MID$(c100$, jjj, 10); " ";
        IF j = 4 THEN PRINT #2, MID$(c100$, jjj, 10)
        IF sc <> 0 THEN
' QB 03
          IF j < 4 THEN PRINT _
```

```
          MID$(c100$, jjj, 10); " ";
        IF j = 4 THEN PRINT MID$(c100$, jjj, 10)
' VB 06
        If j < 4 Then Form1.Text1.Text = _
          Form1.Text1.Text & _
          Mid$(c100$, jjj, 10) & " "
        If j = 4 Then Form1.Text1.Text = _
          Form1.Text1.Text & Mid$(c100$, jjj, _
          10) & Chr$(13) & Chr$(10)
      END IF
    NEXT j
    chead$ = SPACE$(kh)
  NEXT jj
  IF i = 1 AND jp > 1 THEN
    PRINT #2, "     ...                    ...";
    PRINT #2, SPACE$(19); "..."
    IF sc <> 0 THEN
' QB 02
      PRINT "     ...                    ...";
      PRINT SPACE$(19); "..."
' VB 03
      Form1.Text1.Text = Form1.Text1.Text & _
        "     ...                    ..." & _
        Space$(19) & "..." & Chr$(13) & Chr$(10)
    END IF
  END IF
  IF m > 0 AND i < ls AND (i MOD 10) = 0 THEN
    PRINT #2, SPACE$(13); "^^  "; 100 * i;
    PRINT #2, " digits"
    IF sc <> 0 THEN
' QB 01
      PRINT SPACE$(13); "^^  "; 100 * i; " digits"
```

```
' VB 03
        Form1.Text1.Text = Form1.Text1.Text & _
          Space$(13) & "^^  " & 100 * i & _
          " digits" & Chr$(13) & Chr$(10)
      END IF
    END IF
  END IF
NEXT i
END SUB

REM *** *** x=a^(1/2), y=a^(-1/2)
SUB bsqrt (a() AS INTEGER, x() AS INTEGER, _
    y() AS INTEGER, s)
REDIM w(n + 2) AS INTEGER, three(n + 2) AS INTEGER
DIM dy AS DOUBLE, a2 AS INTEGER, a22 AS INTEGER
IF a(1) = 0 THEN
  IF s <> -1 THEN CALL bdtom(0#, x())
  CALL bdtom(1D+308, y())
ELSE
  IF ABS(a(2)) < 76 THEN
    CALL bmtod(a(), dy)
    dy = 1# / SQR(dy)
    CALL bdtom(dy, y())
  ELSE
    a2 = a(2)
    la2& = tpowa&(a(2))
    la3& = la2& \ 2
    a(2) = la2& - 2 * la3&
    CALL bmtod(a(), dy)
    a(2) = a2
    dy = 1# / SQR(dy)
```

```
    CALL bdtom(dy, y())
    c2% = powbd + 1
    pow(1) = la3&
    y(2) = tpower%(y(2), c2%, c3%, 1) ' y(2)=y(2)+la3&
  END IF
  CALL bdtom(3#, three())
  nn = n
  n1 = 2
  DO WHILE n1 < nn - 2
    n1 = n1 + n1
    n = n1 + 4
    IF n > nn THEN n = nn
    CALL bmul(y(), y(), w())
    CALL bmul(a(), w(), w())
    CALL bsub(three(), w(), w())
    CALL bmul(y(), w(), w())
    CALL bdivi(w(), 2, y())
  LOOP
  IF s <> -1 THEN CALL bmul(a(), y(), x())
END IF
END SUB

REM *** *** c=a-b
SUB bsub (a() AS INTEGER, b() AS INTEGER, _
    c() AS INTEGER)
DIM sb AS INTEGER, sc AS INTEGER
sb = b(1)
b(1) = -sb
CALL badd(a(), b(), c())
sc = c(1)
b(1) = sb
c(1) = sc
```

```
END SUB

REM *** *** FFT, complex sequence
SUB fftc (n, isign, xr() AS DOUBLE, xi() AS DOUBLE)
DIM c AS DOUBLE, c1 AS DOUBLE, dn AS DOUBLE
DIM s AS DOUBLE, s1 AS DOUBLE, s2 AS DOUBLE
DIM tr AS DOUBLE, ti AS DOUBLE, d AS DOUBLE
dn = n
m = INT(LOG(dn) / LOG(2#) + .01#)
n1 = n - 1
n2 = n / 2
j = 1
FOR i = 1 TO n1
  IF i < j THEN
    d = xr(i)
    xr(i) = xr(j)
    xr(j) = d
    d = xi(i)
    xi(i) = xi(j)
    xi(j) = d
  END IF
  k = n2
  DO WHILE k < j
    j = j - k
    k = k / 2
  LOOP
  j = j + k
NEXT i
le = 1
FOR l = 1 TO m
  le1 = le
  le = le + le
```

```
    s = 0#
    c = 1#
    d = 3.141592653589793# / le1
    s1 = SIN(isign * d)
    c1 = COS(d)
    isc = 0
    FOR ip = 1 TO le1
      FOR i = ip TO n STEP le
        j = i + le1
        tr = c * xr(j) - s * xi(j)
        ti = c * xi(j) + s * xr(j)
        xr(j) = xr(i) - tr
        xi(j) = xi(i) - ti
        xr(i) = xr(i) + tr
        xi(i) = xi(i) + ti
      NEXT i
      s2 = s
      s = s * c1 + c * s1
      c = c * c1 - s2 * s1
      isc = isc + 1
      IF isc = 8 THEN
        isc = 0
        s2 = ip * d
        s = SIN(isign * s2)
        c = COS(s2)
      END IF
    NEXT ip
  NEXT l
  IF isign = -1 THEN
    FOR i = 1 TO n
      xr(i) = xr(i) / dn
      xi(i) = xi(i) / dn
```

```
  NEXT i
END IF
END SUB

REM *** *** FFT, 2 real sequences
SUB fftr2 (n, isign, rrcx() AS DOUBLE, _
    ricx() AS DOUBLE)
DIM ra AS DOUBLE, rb AS DOUBLE
DIM rc AS DOUBLE, rd AS DOUBLE
i1 = n / 2
ra = rrcx(n)
rb = ricx(n)
FOR i = 1 TO i1
  i2 = n - i + 1
  i3 = n - i
  rrcx(i2) = rrcx(i3)
  ricx(i2) = ricx(i3)
NEXT i
rc = ricx(1)
ricx(1) = ra
rrcx(i1 + 1) = rc
ricx(i1 + 1) = rb
FOR i = 2 TO i1
  i2 = n - i + 2
  ra = rrcx(i) - ricx(i2)
  rb = ricx(i) - rrcx(i2)
  rc = rrcx(i) + ricx(i2)
  rd = ricx(i) + rrcx(i2)
  rrcx(i) = ra
  ricx(i) = rd
  rrcx(i2) = rc
  ricx(i2) = -rb
NEXT i
```

```
CALL fftc(n, isign, rrcx(), ricx())
END SUB

REM *** *** FFT, real and double-length sequence
SUB fftrl (n, isign, rrcx() AS DOUBLE, _
    ricx() AS DOUBLE)
DIM ra AS DOUBLE, rb AS DOUBLE
DIM rrcx0 AS DOUBLE, ricx0 AS DOUBLE, rc AS DOUBLE
DIM rrcxi AS DOUBLE, ricxi AS DOUBLE, rd AS DOUBLE
DIM rrcw AS DOUBLE, ricw AS DOUBLE, arg AS DOUBLE
i1 = n / 2
i2 = i1 + 1
i3 = n - 1
ra = 2# * rrcx(i2)
rb = 2# * ricx(i2)
rc = rrcx(1)
rd = ricx(1)
FOR i = i2 TO i3
  rrcx(i) = rrcx(i + 1)
  ricx(i) = ricx(i + 1)
NEXT i
FOR i = 2 TO i1
  i4 = n - i + 1
  arg = (i - 1) * isign * 3.141592653589793# / n
  rrcxi = rrcx(i) + rrcx(i4)
  ricxi = ricx(i) - ricx(i4)
  rrcw = COS(arg)
  ricw = SIN(arg)
  rrcx0 = rrcx(i) - rrcx(i4)
  ricx0 = ricx(i) + ricx(i4)
  rrcx(i4) = rrcx0 * rrcw - ricx0 * ricw
  ricx(i4) = ricx0 * rrcw + rrcx0 * ricw
```

```
  rrcx(i) = rrcxi
  ricx(i) = ricxi
NEXT i
rrcx(1) = rc + rd
ricx(1) = ra
rrcx(n) = rc - rd
ricx(n) = -rb
CALL fftr2(n, isign, rrcx(), ricx())
END SUB

REM *** *** c=|a|+|b|  |a|>|b|
SUB tadd (a() AS INTEGER, b() AS INTEGER, _
    c() AS INTEGER)
REDIM s(n + 2) AS INTEGER
DIM a1 AS INTEGER, b1 AS INTEGER, c1 AS INTEGER
b1 = tpower%(a(2), b(2), a1, -2)  ' k = a(2)-b(2)
k = INT(pow1)
IF k >= n - 1 THEN
  FOR i = 2 TO n
    c(i) = a(i)
  NEXT i
ELSE
  n1 = n + 1
  n2 = n + 2
  a1 = a(n1)
  b1 = b(n1)
  a(n1) = 0
  b(n1) = 0
  s(n1) = 0
  k2 = k + 2
  FOR i = 1 TO k2
    s(i) = 0
  NEXT i
```

```
  k3 = k + 3
  FOR i = k3 TO n1
    s(i) = b(i - k)
  NEXT i
  FOR i = 3 TO n1
    s(i) = s(i) + a(i)
  NEXT i
  CALL tcarry(s(), n)
  IF s(2) = 0 THEN
    c1 = 0
  ELSE
    FOR i = 1 TO n
      s(n2 - i) = s(n1 - i)
    NEXT i
    c1 = 1
  END IF
  c(2) = tpower%(a(2), c1, c2%, 1)   ' c(2)=a(2)+c1
  IF s(n1) > pd2 THEN s(n) = s(n) + 1
  FOR i = 3 TO n
    c(i) = s(i)
  NEXT i
  a(n1) = a1
  b(n1) = b1
END IF
END SUB

REM *** *** carry
SUB tcarry (a() AS INTEGER, m)
DIM y AS INTEGER
FOR i = m TO 3 STEP -1
  x = a(i) / p
  y = FIX(x)
  IF x < 0 AND y <> x THEN y = y - 1
```

```
  a(i) = a(i) - p * y
  a(i - 1) = a(i - 1) + y
NEXT i
END SUB

REM *** *** fit power 2
FUNCTION tfit2 (m)
  m1 = 4
  DO WHILE m > m1
    m1 = m1 + m1
  LOOP
  tfit2 = m1
END FUNCTION

REM *** *** normalization
SUB tnorm (a() AS INTEGER)
IF n >= 4 THEN
  FOR i = n TO 4 STEP -1
    IF a(i) <> p THEN EXIT SUB
    a(i) = 0
    a(i - 1) = a(i - 1) + 1
  NEXT i
END IF
IF a(3) = p THEN
  a(3) = 1
  a(2) = tpower%(a(2), 1, c2%, 1)   ' a(2) = a(2)+1
END IF
END SUB

REM *** *** pick up power   ( abs(a(2)) > 32600 )
FUNCTION tpowa& (a AS INTEGER)
DIM ka AS INTEGER
ka = ABS(a) - powbd
```

```
IF ka < 1 THEN
  tpowa& = a
ELSE
  IF ka > powi THEN
' QB 01
    PRINT "power is incorrect,  see FUNCTION tpowa&"
' VB 04
    Form1.Text1.Text = Form1.Text1.Text & _
      "power is incorrect,  see FUNCTION tpowa&" _
      & Chr$(13) & Chr$(10)
    Form1.Text1.Refresh
    STOP
  END IF
  tpowa& = pow(ka)
END IF
END FUNCTION

REM *** *** power operating  ( abs(a(2)) > 32600 )
FUNCTION tpower% (a AS INTEGER, b AS INTEGER, _
         c AS INTEGER, job)
DIM ka AS INTEGER
IF ABS(job) = 1 THEN
  powl = tpowa&(a) + tpowa&(b)
ELSE
  powl = tpowa&(a) - tpowa&(b)
END IF
IF job < 0 THEN
  tpower% = c
ELSE
  IF ABS(powl) <= powbd THEN
    tpower% = powl
  ELSE
```

```
    IF ABS(c) <= powbd THEN
      powi = powi + 1
      IF powi + powbd > 32767 THEN
' QB 01
        PRINT "abs(power) > 32767,  see SUB tpower%"
' VB 04
        Form1.Text1.Text = Form1.Text1.Text & _
          "abs(power) > 32767,  see SUB tpower%" _
          & Chr$(13) & Chr$(10)
          Form1.Text1.Refresh
        STOP
      END IF
      tpower% = powbd + powi
      pow(powi) = powl
    ELSE
      ka = ABS(c) - powbd
      IF ka > powi THEN
' QB 01
        PRINT "power is incorrect, see SUB tpower%"
' VB 04
        Form1.Text1.Text = Form1.Text1.Text & _
          "power is incorrect, see SUB tpower%" _
          & Chr$(13) & Chr$(10)
          Form1.Text1.Refresh
        STOP
      END IF
      tpower% = c
      pow(ka) = powl
    END IF
  END IF
END IF
END FUNCTION
```

```
REM *** *** c=|a|-|b|   |a|>|b|
SUB tsub (a() AS INTEGER, b() AS INTEGER, _
    c() AS INTEGER)
REDIM s(n + 2) AS INTEGER
DIM a1 AS INTEGER, a2 AS INTEGER
DIM b1 AS INTEGER, b2 AS INTEGER
b1 = tpower%(a(2), b(2), a1, -2)   ' k = a(2)-b(2)
k = INT(pow1)
k3 = k + 3
n1 = n + 1
n2 = n + 2
a1 = a(n1)
a2 = a(n2)
b1 = b(n1)
b2 = b(n2)
a(n1) = 0
a(n2) = 0
b(n1) = 0
b(n2) = 0
FOR i = 1 TO n2
  s(i) = 0
NEXT i
IF k >= n THEN
  s(n2) = 1
ELSE
  FOR i = k3 TO n2
    s(i) = b(i - k)
  NEXT i
  IF k > 2 THEN
    n3 = n + 3 - k
    j = 0
    FOR i = n3 TO n2
```

```
      IF b(i) <> 0 THEN
        j = 1
        EXIT FOR
      END IF
    NEXT i
    IF j = 1 THEN s(n2) = s(n2) + 1
  END IF
END IF
FOR i = 3 TO n2
  s(i) = a(i) - s(i)
NEXT i
CALL tcarry(s(), n2)
j = 0
FOR i = 3 TO n2
  IF s(i) <> 0 THEN EXIT FOR
  j = j + 1
NEXT i
IF j < n THEN
  IF (n - j) >= (n - 1) THEN
    IF s(j + n1) >= pd2 THEN
      s(j + n) = s(j + n) + 1
      s(j + n1) = 0
    END IF
  END IF
  IF n < (n2 - j) THEN
    n3 = n
  ELSE
    n3 = n2 - j
  END IF
  FOR i = 3 TO n3
    c(i) = s(i + j)
  NEXT i
  n3 = n3 + 1
```

```
  IF n3 <= n THEN
    FOR i = n3 TO n
      c(i) = 0
    NEXT i
  END IF
  c2% = powbd + 1
  pow(1) = j
  c(2) = tpower%(a(2), c2%, c3%, 2)   ' c(2)=a(2)-j
ELSE
  FOR i = 2 TO n
    c(i) = 0
  NEXT i
END IF
a(n1) = a1
a(n2) = a2
b(n1) = b1
b(n2) = b2
END SUB
```

附录 B
π 的前两万位及其后的部分位

本附录给出 π 值前两万位的完整结果及其后直至百万位的部分结果。前者可由本书三个程序中的任一个算出，它们是无需多精度系统支持的 BBP 算法程序 (见第 4 章)、需要多精度系统支持的算术几何平均 (AGM) 算法程序和波尔温四阶算法程序 (见第 7 章)。两万位以后的结果是由最后一个程序在 Windows 的 Visual BASIC 上得到的。所列结果是计算机直接输出的，并且与我们过去的计算结果以及国际上发表的结果核对过，最后用照相排版，因此是高度可靠的。

3.1415926535 8979323846 2643383279 5028841971 6939937510
5820974944 5923078164 0628620899 8628034825 3421170679
8214808651 3282306647 0938446095 5058223172 5359408128
4811174502 8410270193 8521105559 6446229489 5493038196
4428810975 6659334461 2847564823 3786783165 2712019091
4564856692 3460348610 4543266482 1339360726 0249141273
7245870066 0631558817 4881520920 9628292540 9171536436
7892590360 0113305305 4882046652 1384146951 9415116094
3305727036 5759591953 0921861173 8193261179 3105118548
0744623799 6274956735 1885752724 8912279381 8301194912
9833673362 4406566430 8602139494 6395224737 1907021798
6094370277 0539217176 2931767523 8467481846 7669405132
0005681271 4526356082 7785771342 7577896091 7363717872

1468440901 2249534301 4654958537 1050792279 6892589235
4201995611 2129021960 8640344181 5981362977 4771309960
5187072113 4999999837 2978049951 0597317328 1609631859
5024459455 3469083026 4252230825 3344685035 2619311881
7101000313 7838752886 5875332083 8142061717 7669147303
5982534904 2875546873 1159562863 8823537875 9375195778
1857780532 1712268066 1300192787 6611195909 2164201989
^^ 1000 digits
3809525720 1065485863 2788659361 5338182796 8230301952
0353018529 6899577362 2599413891 2497217752 8347913151
5574857242 4541506959 5082953311 6861727855 8890750983
8175463746 4939319255 0604009277 0167113900 9848824012
8583616035 6370766010 4710181942 9555961989 4676783744
9448255379 7747268471 0404753464 6208046684 2590694912
9331367702 8989152104 7521620569 6602405803 8150193511
2533824300 3558764024 7496473263 9141992726 0426992279
6782354781 6360093417 2164121992 4586315030 2861829745
5570674983 8505494588 5869269956 9092721079 7509302955
3211653449 8720275596 0236480665 4991198818 3479775356
6369807426 5425278625 5181841757 4672890977 7727938000
8164706001 6145249192 1732172147 7235014144 1973568548
1613611573 5255213347 5741849468 4385233239 0739414333
4547762416 8625189835 6948556209 9219222184 2725502542
5688767179 0494601653 4668049886 2723279178 6085784383
8279679766 8145410095 3883786360 9506800642 2512520511
7392984896 0841284886 2694560424 1965285022 2106611863
0674427862 2039194945 0471237137 8696095636 4371917287
4677646575 7396241389 0865832645 9958133904 7802759009
^^ 2000 digits

9465764078 9512694683 9835259570 9825822620 5224894077
2671947826 8482601476 9909026401 3639443745 5305068203
4962524517 4939965143 1429809190 6592509372 2169646151
5709858387 4105978859 5977297549 8930161753 9284681382
6868386894 2774155991 8559252459 5395943104 9972524680
8459872736 4469584865 3836736222 6260991246 0805124388
4390451244 1365497627 8079771569 1435997700 1296160894
4169486855 5848406353 4220722258 2848864815 8456028506
0168427394 5226746767 8895252138 5225499546 6672782398
6456596116 3548862305 7745649803 5593634568 1743241125
1507606947 9451096596 0940252288 7971089314 5669136867
2287489405 6010150330 8617928680 9208747609 1782493858
9009714909 6759852613 6554978189 3129784821 6829989487
2265880485 7564014270 4775551323 7964145152 3746234364
5428584447 9526586782 1051141354 7357395231 1342716610
2135969536 2314429524 8493718711 0145765403 5902799344
0374200731 0578539062 1983874478 0847848968 3321445713
8687519435 0643021845 3191048481 0053706146 8067491927
8191197939 9520614196 6342875444 0643745123 7181921799
9839101591 9561814675 1426912397 4894090718 6494231961
^^ 3000 digits
5679452080 9514655022 5231603881 9301420937 6213785595
6638937787 0830390697 9207734672 2182562599 6615014215
0306803844 7734549202 6054146659 2520149744 2850732518
6660021324 3408819071 0486331734 6496514539 0579626856
1005508106 6587969981 6357473638 4052571459 1028970641
4011097120 6280439039 7595156771 5770042033 7869936007
2305587631 7635942187 3125147120 5329281918 2618612586
7321579198 4148488291 6447060957 5270695722 0917567116

7229109816 9091528017 3506712748 5832228718 3520935396
5725121083 5791513698 8209144421 0067510334 6711031412
6711136990 8658516398 3150197016 5151168517 1437657618
3515565088 4909989859 9823873455 2833163550 7647918535
8932261854 8963213293 3089857064 2046752590 7091548141
6549859461 6371802709 8199430992 4488957571 2828905923
2332609729 9712084433 5732654893 8239119325 9746366730
5836041428 1388303203 8249037589 8524374417 0291327656
1809377344 4030707469 2112019130 2033038019 7621101100
4492932151 6084244485 9637669838 9522868478 3123552658
2131449576 8572624334 4189303968 6426243410 7732269780
2807318915 4411010446 8232527162 0105265227 2111660396
^^ 4000 digits
6655730925 4711055785 3763466820 6531098965 2691862056
4769312570 5863566201 8558100729 3606598764 8611791045
3348850346 1136576867 5324944166 8039626579 7877185560
8455296541 2665408530 6143444318 5867697514 5661406800
7002378776 5913440171 2749470420 5622305389 9456131407
1127000407 8547332699 3908145466 4645880797 2708266830
6343285878 5698305235 8089330657 5740679545 7163775254
2021149557 6158140025 0126228594 1302164715 5097925923
0990796547 3761255176 5675135751 7829666454 7791745011
2996148903 0463994713 2962107340 4375189573 5961458901
9389713111 7904297828 5647503203 1986915140 2870808599
0480109412 1472213179 4764777262 2414254854 5403321571
8530614228 8137585043 0633217518 2979866223 7172159160
7716692547 4873898665 4949450114 6540628433 6639379003
9769265672 1463853067 3609657120 9180763832 7166416274
8888007869 2560290228 4721040317 2118608204 1900042296

6171196377 9213375751 1495950156 6049631862 9472654736
4252308177 0367515906 7350235072 8354056704 0386743513
6222247715 8915049530 9844489333 0963408780 7693259939
7805419341 4473774418 4263129860 8099888687 4132604721
^^ 5000 digits
5695162396 5864573021 6315981931 9516735381 2974167729
4786724229 2465436680 0980676928 2382806899 6400482435
4037014163 1496589794 0924323789 6907069779 4223625082
2168895738 3798623001 5937764716 5122893578 6015881617
5578297352 3344604281 5126272037 3431465319 7777416031
9906655418 7639792933 4419521541 3418994854 4473456738
3162499341 9131814809 2777710386 3877343177 2075456545
3220777092 1201905166 0962804909 2636019759 8828161332
3166636528 6193266863 3606273567 6303544776 2803504507
7723554710 5859548702 7908143562 4014517180 6246436267
9456127531 8134078330 3362542327 8394497538 2437205835
3114771199 2606381334 6776879695 9703098339 1307710987
0408591337 4641442822 7726346594 7047458784 7787201927
7152807317 6790770715 7213444730 6057007334 9243693113
8350493163 1284042512 1925651798 0694113528 0131470130
4781643788 5185290928 5452011658 3934196562 1349143415
9562586586 5570552690 4965209858 0338507224 2648293972
8584783163 0577775606 8887644624 8246857926 0395352773
4803048029 0058760758 2510474709 1643961362 6760449256
2742042083 2085661190 6254543372 1315359584 5068772460
^^ 6000 digits
2901618766 7952406163 4252257719 5429162991 9306455377
9914037340 4328752628 8896399587 9475729174 6426357455
2540790914 5135711136 9410911939 3251910760 2082520261

8798531887 7058429725 9167781314 9699009019 2116971737
2784768472 6860849003 3770242429 1651300500 5168323364
3503895170 2989392233 4517220138 1280696501 1784408745
1960121228 5993716231 3017114448 4640903890 6449544400
6198690754 8516026327 5052983491 8740786680 8818338510
2283345085 0486082503 9302133219 7155184306 3545500766
8282949304 1377655279 3975175461 3953984683 3936383047
4611996653 8581538420 5685338621 8672523340 2830871123
2827892125 0771262946 3229563989 8989358211 6745627010
2183564622 0134967151 8819097303 8119800497 3407239610
3685406643 1939509790 1906996395 5245300545 0580685501
9567302292 1913933918 5680344903 9820595510 0226353536
1920419947 4553859381 0234395544 9597783779 0237421617
2711172364 3435439478 2218185286 2408514006 6604433258
8856986705 4315470696 5747458550 3323233421 0730154594
0516553790 6866273337 9958511562 5784322988 2737231989
8757141595 7811196358 3300594087 3068121602 8764962867
^^ 7000 digits
4460477464 9159950549 7374256269 0104903778 1986835938
1465741268 0492564879 8556145372 3478673303 9046883834
3634655379 4986419270 5638729317 4872332083 7601123029
9113679386 2708943879 9362016295 1541337142 4892830722
0126901475 4668476535 7616477379 4675200490 7571555278
1965362132 3926406160 1363581559 0742202020 3187277605
2772190055 6148425551 8792530343 5139844253 2234157623
3610642506 3904975008 6562710953 5919465897 5141310348
2276930624 7435363256 9160781547 8181152843 6679570611
0861533150 4452127473 9245449454 2368288606 1340841486
3776700961 2071512491 4043027253 8607648236 3414334623

5189757664 5216413767 9690314950 1910857598 4423919862
9164219399 4907236234 6468441173 9403265918 4044378051
3338945257 4239950829 6591228508 5558215725 0310712570
1266830240 2929525220 1187267675 6220415420 5161841634
8475651699 9811614101 0029960783 8690929160 3028840026
9104140792 8862150784 2451670908 7000699282 1206604183
7180653556 7252532567 5328612910 4248776182 5829765157
9598470356 2226293486 0034158722 9805349896 5022629174
8788202734 2092222453 3985626476 6914905562 8425039127
^^ 8000 digits
5771028402 7998066365 8254889264 8802545661 0172967026
6407655904 2909945681 5065265305 3718294127 0336931378
5178609040 7086671149 6558343434 7693385781 7113864558
7367812301 4587687126 6034891390 9562009939 3610310291
6161528813 8437909904 2317473363 9480457593 1493140529
7634757481 1935670911 0137751721 0080315590 2485309066
9203767192 2033229094 3346768514 2214477379 3937517034
4366199104 0337511173 5471918550 4644902636 5512816228
8244625759 1633303910 7225383742 1821408835 0865739177
1509682887 4782656995 9957449066 1758344137 5223970968
3408005355 9849175417 3818839994 4697486762 6551658276
5848358845 3142775687 9002909517 0283529716 3445621296
4043523117 6006651012 4120065975 5851276178 5838292041
9748442360 8007193045 7618932349 2292796501 9875187212
7267507981 2554709589 0455635792 1221033346 6974992356
3025494780 2490114195 2123828153 0911407907 3860251522
7429958180 7247162591 6685451333 1239480494 7079119153
2673430282 4418604142 6363954800 0448002670 4962482017
9289647669 7583183271 3142517029 6923488962 7668440323

2609275249 6035799646 9256504936 8183609003 2380929345

^^ 9000 digits

9588970695 3653494060 3402166544 3755890045 6328822505
4525564056 4482465151 8754711962 1844396582 5337543885
6909411303 1509526179 3780029741 2076651479 3942590298
9695946995 5657612186 5619673378 6236256125 2163208628
6922210327 4889218654 3648022967 8070576561 5144632046
9279068212 0738837781 4233562823 6089632080 6822246801
2248261177 1858963814 0918390367 3672220888 3215137556
0037279839 4004152970 0287830766 7094447456 0134556417
2543709069 7939612257 1429894671 5435784687 8861444581
2314593571 9849225284 7160504922 1242470141 2147805734
5510500801 9086996033 0276347870 8108175450 1193071412
2339086639 3833952942 5786905076 4310063835 1983438934
1596131854 3475464955 6978103829 3097164651 4384070070
7360411237 3599843452 2516105070 2705623526 6012764848
3084076118 3013052793 2054274628 6540360367 4532865105
7065874882 2569815793 6789766974 2205750596 8344086973
5020141020 6723585020 0724522563 2651341055 9240190274
2162484391 4035998953 5394590944 0704691209 1409387001
2645600162 3742880210 9276457931 0657922955 2498872758
4610126483 6999892256 9596881592 0560010165 5256375678

^^ 10000 digits

5667227966 1988578279 4848855834 3975187445 4551296563
4434803966 4205579829 3680435220 2770984294 2325330225
7634180703 9476994159 7915945300 6975214829 3366555661
5678736400 5366656416 5473217043 9035213295 4352916941
4599041608 7532018683 7937023488 8689479151 0716378529
0234529244 0773659495 6305100742 1087142613 4974595615

1384987137 5704710178 7957310422 9690666702 14498637**46**
4595280824 3694457897 7233004876 4765241339 0759204**340**
1963403911 4732023380 7150952220 1068256342 747164**6024**
3354400515 2126693249 3419673977 0415956837 535551**6673**
0273900749 7297363549 6453328886 9844061196 496162773**4**
4951827369 5588220757 3551766515 8985519098 666539**3549**
4810688732 0685990754 0792342402 3009259007 017319**6036**
2254756478 9406475483 4664776041 1463233**905** 651343**3068**
4495397907 0903023460 4614709616 9688688501 4083470**405**
4607429586 9913829668 2468185710 3188790652 870366**5083**
2431974404 7718556789 3482308**943** 1068287027 228097**3624**
8093996270 6074726455 3992539944 2808113736 943388729**4**
0630792615 9599546262 4629707062 5948455690 347119729**9**
6409089418 0595343932 5123623550 8134949004 3642785271
^^ 11000 digits
3831591256 8989295196 4272875739 4691427253 4366941532
3610045373 0488198551 7065941217 3524625895 4873016760
0298865925 7866285612 4966552353 3829428785 4253404830
8330701653 7228563559 1525347844 5981831341 1290019992
0598135220 5117336585 6407826484 9427644113 7639386692
4803118364 4536985891 7544264739 9882284621 8449008777
6977631279 5722672655 5625962825 4276531830 0134070922
3343657791 6012809317 9401718598 5999338492 3549564005
7099558561 1349802524 9906698423 3017350358 0440811685
5265311709 9570899427 3287092584 8789443646 0050410892
2669178352 5870785951 2983441729 5351953788 5534573742
6085902908 1765155780 3905946408 7350612322 6112009373
1080485485 2635722825 7682034160 5048466277 5045003126
2008007998 0492548534 6941469775 1649327095 0493463938

2432227188 5159740547 0214828971 1177792376 1225788734
7718819682 5462981268 6858170507 4027255026 3329044976
2778944236 2167411918 6269439650 6715157795 8675648239
9391760426 0176338704 5499017614 3641204692 1823707648
8783419689 6861181558 1587360629 3860381017 1215855272
6683008238 3404656475 8804051380 8016336388 7421637140

^^ 12000 digits

6435495561 8689641122 8214075330 2655100424 1048967835
2858829024 3670904887 1181909094 9453314421 8287661810
3100735477 0549815968 0772009474 6961343609 2861484941
7850171807 7930681085 4690009445 8995279424 3981392135
0558642219 6483491512 6390128038 3200109773 8680662877
9239718014 6134324457 2640097374 2570073592 1003154150
8936793008 1699805365 2027600727 7496745840 0283624053
4603726341 6554259027 6018348403 0681138185 5105979705
6640075094 2608788573 5796037324 5141467867 0368809880
6097164258 4975951380 6930944940 1515422221 9432913021
7391253835 5915031003 3303251117 4915696917 4502714943
3151558854 0392216409 7229101129 0355218157 6282328318
2342548326 1119128009 2825256190 2052630163 9114772473
3148573910 7775874425 3876117465 7867116941 4776421441
1112635835 5387136101 1023267987 7564102468 2403226483
4641766369 8066378576 8134920453 0224081972 7856471983
9630878154 3221166912 2464159117 7673225326 4335686146
1865452226 8126887268 4459684424 1610785401 6768142080
8850280054 1436131462 3082102594 1737562389 9420757136
2751674573 1891894562 8352570441 3354375857 5342698699

^^ 13000 digits

4725470316 5661399199 9682628247 2706413362 2217892390

3176085428 9437339356 1889165125 0424404008 9527198378
7386480584 7268954624 3882343751 7885201439 5600571048
1194988423 9060613695 7342315590 7967034614 9143447886
3604103182 3507365027 7859089757 8272731305 0488939890
0992391350 3373250855 9826558670 8924261242 9473670193
9077271307 0686917092 6462548423 2407485503 6608013604
6689511840 0936686095 4632500214 5852930950 0009071510
5823626729 3264537382 1049387249 9669933942 4685516483
2611341461 1068026744 6637334375 3407642940 2668297386
5220935701 6263846485 2851490362 9320199199 6882851718
3953669134 5222444708 0459239660 2817156551 5656661113
5982311225 0628905854 9145097157 5539002439 3153519090
2107119457 3002438801 7661503527 0862602537 8817975194
7806101371 5004489917 2100222013 3501310601 6391541589
5780371177 9277522597 8742891917 9155224171 8958536168
0594741234 1933984202 1874564925 6443462392 5319531351
0331147639 4911995072 8584306583 6193536932 9699289837
9149419394 0608572486 3968836903 2655643642 1664425760
7914710869 9843157337 4964883529 2769328220 7629472823

^^ 14000 digits

8153740996 1545598798 2598910937 1712621828 3025848112
3890119682 2142945766 7580718653 8065064870 2613389282
2994972574 5303328389 6381843944 7707794022 8435988341
0035838542 3897354243 9564755568 4095224844 5541392394
1000162076 9363684677 6413017819 6593799715 5746854194
6334893748 4391297423 9143365936 0410035234 3777065888
6778113949 8616478747 1407932638 5873862473 2889645643
5987746676 3847946650 4074111825 6583788784 5485814896
2961273998 4134427260 8606187245 5452360643 1537101127

4680977870 4464094758 2803487697 5894832824 1239292960
5829486191 9667091895 8089833201 2103184303 4012849511
6203534280 1441276172 8583024355 9830032042 0245120728
7253558119 5840149180 9692533950 7577840006 7465526031
4461670508 2768277222 3534191102 6341631571 4740612385
0425845988 4199076112 8725805911 3935689601 4316682831
7632356732 5417073420 8173322304 6298799280 4908514094
7903688786 8789493054 6955703072 6190095020 7643349335
9106024545 0864536289 3545686295 8531315337 1838682656
1786227363 7169757741 8302398600 6591481616 4049449650
1173213138 9574706208 8474802365 3710311508 9842799275
^^ 15000 digits
4426853277 9743113951 4357417221 9759799359 6852522857
4526379628 9612691572 3579866205 7340837576 6873884266
4059909935 0500081337 5432454635 9675048442 3528487470
1443545419 5762584735 6421619813 4073468541 1176688311
8654489377 6979566517 2796623267 1481033864 3913751865
9467300244 3450054499 5399742372 3287124948 3470604406
3471606325 8306498297 9551010954 1836235030 3094530973
3583446283 9476304775 6450150085 0757894954 8931393944
8992161255 2559770143 6858943585 8775263796 2559708167
7643800125 4365023714 1278346792 6101995585 2247172201
7772370041 7808419423 9487254068 0155603599 8390548985
7235467456 4239058585 0216719031 3952629445 5439131663
1345308939 0620467843 8778505423 9390524731 3620129476
9187497519 1011472315 2893267725 3391814660 7300089027
7689631148 1090220972 4520759167 2970078505 8071718638
1054967973 1001678708 5069420709 2232908070 3832634534
5203802786 0990556900 1341371823 6837099194 9516489600

7550493412 6787643674 6384902063 9640197666 8559233565
4639138363 1857456981 4719621084 1080961884 6054560390
3845534372 9141446513 4749407848 8442377217 5154334260
^^ 16000 digits
3066988317 6833100113 3108690421 9390310801 4378433415
1370924353 0136776310 8491351615 6422698475 0743032971
6746964066 6531527035 3254671126 6752246055 1199581831
9637637076 1799191920 3579582007 5956053023 4626775794
3936307463 0569010801 1494271410 0939136913 8107258137
8135789400 5599500183 5425118417 2136055727 5221035268
0373572652 7922417373 6057511278 8721819084 4900617801
3889710770 8229310027 9766593583 8758909395 6881485602
6322439372 6562472776 0378908144 5883785501 9702843779
3624078250 5270487581 6470324581 2908783952 3245323789
6029841669 2254896497 1560698119 2186584926 7704039564
8127810217 9913217416 3058105545 9880130048 4562997651
1212415363 7451500563 5070127815 9267142413 4210330156
6165356024 7338078430 2865525722 2753049998 8370153487
9300806260 1809623815 1613669033 4111138653 8510919367
3938352293 4588832255 0887064507 5394739520 4396807906
7086806445 0969865488 0168287434 3786126453 8158342807
5306184548 5903798217 9945996811 5441974253 6344399602
9025100158 8827216474 5006820704 1937615845 4712318346
0072629339 5505482395 5713725684 0232268213 0124767945
^^ 17000 digits
2264482091 0235647752 7230820810 6351889915 2692889108
4555711266 0396503439 7896278250 0161101532 3516051965
5904211844 9499077899 9200732947 6905868577 8787209829
0135295661 3978884860 5097860859 5701773129 8155314951

6814671769 5976099421 0036183559 1387778176 9845875810
4466283998 8060061622 9848616935 3373865787 7359833616
1338413385 3684211978 9389001852 9569196780 4554482858
4837011709 6721253533 8758621582 3101331038 7766827211
5726949518 1795897546 9399264219 7915523385 7662316762
7547570354 6994148929 0413018638 6119439196 2838870543
6777432242 7680913236 5449485366 7680000010 6526248547
3055861598 9991401707 6983854831 8875014293 8908995068
5453076511 6803337322 2651756622 0752695179 1442252808
1651716677 6672793035 4851542040 2381746089 2328391703
2754257508 6765511785 9395002793 3895920576 6827896776
4453184040 4185540104 3513483895 3120132637 8369283580
8271937831 2654961745 9970567450 7183320650 3455664403
4490453627 5600112501 8433560736 1222765949 2783937064
7842645676 3388188075 6561216896 0504161139 0390639601
6202215368 4941092605 3876887148 3798955999 9112099164
^^ 18000 digits
6464411918 5682770045 7424343402 1672276445 5893301277
8158686952 5069499364 6101756850 6016714535 4315814801
0545886056 4550133203 7586454858 4032402987 1709348091
0556211671 5468484778 0394475697 9804263180 9917564228
0987399876 6973237695 7370158080 6822904599 2123661689
0259627304 3067931653 1149401764 7376938735 1409336183
3216142802 1497633991 8983548487 5625298752 4238730775
5955595546 5196394401 8218409984 1248982623 6737714672
2606163364 3296406335 7281070788 7581640438 1485018841
1431885988 2769449011 9321296827 1588841338 6943468285
9006664080 6314077757 7257056307 2940049294 0302420498
4165654797 3670548558 0445865720 2276378404 6682337985

2827105784 3197535417 9501134727 3625774080 2134768260
4502285157 9795797647 4670228409 9956160156 9108903845
8245026792 6594205550 3958792298 1852648007 0683765041
8365620945 5543461351 3415257006 5974881916 3413595567
1964965403 2187271602 6485930490 3978748958 9066127250
7948282769 3895352175 3621850796 2977851461 8843271922
3223810158 7444505286 6523802253 2843891375 2738458923
8442253547 2653098171 5784478342 1582232702 0690287232

^^ 19000 digits

3300538621 6347988509 4695472004 7952311201 5043293226
6282727632 1779088400 8786148022 1475376578 1058197022
2630971749 5072127248 4794781695 7296142365 8595782090
8307332335 6034846531 8730293026 6596450137 1837542889
7557971449 9246540386 8179921389 3469244741 9850973346
2679332107 2686870768 0626399193 6196504409 9542167627
8409146698 5692571507 4315740793 8053239252 3947755744
1591845821 5625181921 5523370960 7483329234 9210345146
2643744980 5596103307 9941453477 8457469999 2128599999
3996122816 1521931488 8769388022 2810830019 8601654941
6542616968 5867883726 0958774567 6182507275 9929508931
8052187292 4610867639 9589161458 5505839727 4209809097
8172932393 0106766386 8240401113 0402470073 5085782872
4627134946 3685318154 6969046696 8693925472 5194139929
1465242385 7762550047 4852954768 1479546700 7050347999
5888676950 1612497228 2040303995 4632788306 9597624936
1510102436 5553522306 9061294938 8599015734 6610237122
3547891129 2547696176 0050479749 2806072126 8039226911
0277722610 2544149221 5765045081 2067717357 1202718024
2968106203 7765788371 6690910941 8074487814 0490755178

```
            ^^    20000 digits
```

其后结果是跳跃的，每隔一千或一万甚至数万位印出 50 位，直至一百万位止．给出这些数据的目的是方便读者校验自己算出的 π 值．

```
9759556808 5683629725 3867913275 0555425244 9194358912
            ^^    21000  digits
5612278584 3296846647 5133365736 9238720146 4723679427
            ^^    22000  digits
3390998224 9862406703 1076831844 6607291248 7475403161
            ^^    23000  digits
4155403016 1950568065 4180939409 9820206099 9414021689
            ^^    24000  digits
8353013617 8653673760 6421667781 3773995100 6589528877
            ^^    25000  digits
2271724863 2202889842 5125287217 8260305009 9451082478
            ^^    30000  digits
5091576463 9074693619 8815078146 8526213325 2473837651
            ^^    40000  digits
0652623405 3394391421 1127181069 1052290024 6574236041
            ^^    50000  digits
7057843289 5980288233 5059828208 1966662490 3585778994
            ^^    60000  digits
2193448667 3248275907 9468078798 1942501958 2622320395
            ^^    70000  digits
0726460556 8592577993 2207033733 3398916369 5043466906
            ^^    80000  digits
3024138051 6490717456 5964853748 3546691945 2358031530
            ^^    90000  digits
7015078933 7728658035 7127909137 6742080565 5493624646
```

^^ 100000 digits
9241840841 4755758253 9385240963 3020513497 0474069539
^^ 110000 digits
6136953089 9547584883 0242619627 5898594151 3780515805
^^ 120000 digits
8958428956 8075266320 3627696299 4601808349 4199491270
^^ 130000 digits
9952429068 6799545683 0838859301 3667218521 1353641724
^^ 140000 digits
8500384293 8167759796 1912729990 4591343960 4283622782
^^ 150000 digits
2663242818 1883297682 5708783089 0164354054 6417536779
^^ 160000 digits
1416395267 7732246933 3134089892 2821352367 1609562825
^^ 170000 digits
2750026839 8089195124 3857147278 8922881800 4727979105
^^ 180000 digits
7832987606 3046816009 5210972977 8751360186 3205458955
^^ 190000 digits
0400704911 1330970230 4687661585 7483135080 1444759928
^^ 200000 digits
9248800082 3409032034 8071465228 9022230806 1496574270
^^ 210000 digits
0092262588 3777140487 9655160155 4359374511 0789847180
^^ 220000 digits
4146778016 9732280054 5673449942 5372413845 8236775963
^^ 230000 digits
5503291068 0168291821 5054886572 9790830657 3320056671
^^ 240000 digits
5140252095 1792646811 5682982031 3618827396 4266233216

```
          ^^   250000  digits
6833548515 8606931459 7838528361 6945633618 6314956895
          ^^   300000  digits
0144835846 4003533691 2705562119 5890629893 5556277917
          ^^   350000  digits
4956017828 2456727368 5631218502 0980470362 4641761986
          ^^   400000  digits
4043992505 6649562424 6838020340 5089480290 0026376220
          ^^   450000  digits
4953622322 2219746596 1933252907 4042487602 5138195242
          ^^   500000  digits
0674425080 3288180503 3938921875 6524445169 9554137647
          ^^   600000  digits
2927311830 4531949237 5073949535 9713935327 3415735585
          ^^   700000  digits
5645023674 0863265104 7956555552 3905693241 3659061767
          ^^   800000  digits
3873825659 9464349788 0323272175 8292694516 9986312673
          ^^   900000  digits
5678796130 3311646283 9963464604 2209010610 5779458151
          ^^   1000000  digits
```