嵌入式系统软件设计基础丛书

周航慈 著

嵌入式系统软件设计中的常用算法

北京航空航天大学出版社

嵌入式系统软件设计中的常用算法

周航慈　著

北京航空航天大学出版社

内 容 简 介

本书根据嵌入式系统软件设计需要的常用算法知识编写而成。基本内容有:线性方程组求解、代数插值和曲线拟合、数值积分、能谱处理、数字滤波、数理统计、自动控制、数据排序、数据压缩和检错纠错等常用算法。从嵌入式系统的实际应用出发,用通俗易懂的语言代替枯燥难懂的数学推导,使读者能在比较轻松的条件下学到最基本的常用算法,并为继续学习其他算法打下基础。

本书可作为电子技术人员自学常用算法的教材,也可作为高等院校电子技术类专业本科生、研究生的教学参考书。

图书在版编目(CIP)数据

嵌入式系统软件设计中的常用算法/周航慈著. —北京:
北京航空航天大学出版社,2010.1
ISBN 978-7-81124-943-9

Ⅰ.嵌… Ⅱ.周… Ⅲ.微型计算机—软件设计—算法
Ⅳ.TP311.5

中国版本图书馆 CIP 数据核字(2009)第 189989 号

嵌入式系统软件设计中的常用算法
周航慈 著
责任编辑 董云凤 张金伟
*
北京航空航天大学出版社出版发行
北京市海淀区学院路 37 号(100191) 发行部电话:010-82317024 传真:010-82328026
http://www.buaapress.com.cn E-mail:bhpress@263.net
印刷有限公司印装 各地书店经销
*
开本:787 mm×960 mm 1/16 印张:12.5 字数:280 千字
2010 年 1 月第 1 版 2010 年 1 月第 1 次印刷 印数:5 000 册
ISBN 978-7-81124-943-9 定价:24.00 元

前　言

嵌入式系统在各行各业的应用越来越广，我国从事嵌入式系统开发的人员也越来越多，从国内主要的几种电子杂志上可以看出，有关嵌入式系统应用的文章也越来越多。

在开发一种嵌入式系统产品时，主要是做两方面的设计：硬件设计和软件设计。在硬件设计方面，各个半导体公司竞相推出各种高性能、低功耗、低成本的 CPU 和外围芯片，这使我们在进行硬件设计时可以很快地得到最先进的芯片。在这种情况下，硬件设计的外部条件越来越好，集成度越来越高，在实现相同功能的情况下线路越来越简化。在软件设计方面，虽然开发工具和程序设计语言也在不断提高，但技术人员本身的软件素质无疑起决定作用。因此，软件设计水平在嵌入式系统产品开发的过程中占有重要的地位，直接影响到产品的功能和竞争能力。

我国目前绝大多数从事嵌入式系统开发的技术人员基本上不是计算机专业毕业的，有的可能还没有上过大学，他们未接受过系统的软件基础理论教育，软件设计水平仍不太高。在软件开发过程中，他们只是不自觉地采用了一些规律性的设计方法，或者模仿别人的程序设计方法，而有更多成熟的基本方法没有掌握，开发出来的软件水平不高，致使产品的功能和可靠性受到一定的制约。

软件设计是一门科学，有其自身的规律，也有很多成熟的理论和算法。要学习就要选教材，而目前所能选到的都是专为计算机专业编写的教材。这些教材起点较高，偏重理论叙述，不考虑嵌入式系统的硬件特点，对于广大嵌入式系统开发人员来说不是十分适合，学起来会感到比较抽象和吃力。

出于提高我国广大嵌入式系统开发人员软件素质的愿望，我们决定编写一本适合自学的关于常用算法的书。该书起点要求不高，只要掌握了 C 语言、学习了“数据结构”有关知识并从事过嵌入式系统开发工作的人员就可以看懂。学完本书后，对软件设计中常用的算法就能初步掌握。在进行软件设计时，可以减少很多盲目性，并为更系统、更深入地学习其他计算机软件设计理论打下良好基础。

本书主要内容如下：

第 1 章介绍常用线性方程组求解算法；

第 2 章介绍常用代数插值和曲线拟合算法；

第 3 章介绍常用数值积分算法；

第 4 章介绍常用能谱处理算法；

第 5 章介绍常用数字滤波算法；

第 6 章介绍常用数理统计算法；

第 7 章介绍常用自动控制算法；

第 8 章介绍常用数据排序算法；

第 9 章介绍常用数据压缩算法；

第 10 章介绍常用检错与纠错算法。

本书编写的原则是：尽量结合嵌入式系统的应用实例，采用通俗易懂的叙述方式，介绍最基本的核心内容，以便读者能够顺利入门，为进一步学习更多的算法打下基础。

在本书的编写过程中，得到北京航空航天大学出版社的大力支持，何立民教授给予了无私帮助，在此表示衷心感谢！周立功先生在本书的策划过程中起了很大促进作用，在此也表示衷心感谢！王冬霞参与了部分算法程序的调试工作，在此一并感谢！

由于作者水平有限，书中一定会有错误及不足之处，敬请广大读者予以指正，不胜感谢！

作　者

于东华理工大学

2009 年 8 月

目　录

第 1 章

常用线性方程组求解算法

在一个嵌入式系统中，往往需要从外部环境中获取若干个信息（多输入），然后通过数据处理，从中求解出若干个结果（多输出）。如果多输出与多输入之间为线性关系，则将其称为“多变量线性系统”。在多变量线性系统中，经常要碰到求解多元线性方程组的问题，线性方程组的一般形式为：

$$\begin{cases} A_{11}X_1 + A_{12}X_2 + \cdots + A_{1n}X_n = B_1 \\ A_{21}X_1 + A_{22}X_2 + \cdots + A_{2n}X_n = B_2 \\ \quad\quad\vdots \\ A_{n1}X_1 + A_{n2}X_2 + \cdots + A_{m}X_n = B_n \end{cases}$$

写成矩阵形式便是：

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{m} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix}$$

线性方程组的求解算法有直接法和迭代法等，它们各自又有分支。从嵌入式系统的特点出发，在这里只讨论最基本的主元消去法和行列式法，它们是直接法的代表。在本章的应用实例中，将介绍一个直接法和迭代法相结合的算法。

1.1 主元消去法

主元消去法又分总体选主元消去法和按列选主元消去法，由于前者每次选主元的范围大，速度相对慢，所以采用按列选主元消去法。主元消去法又分为无回代过程的主元消去法和有回代过程的主元消去法。

1.1.1 无回代过程的主元消去法

这里，通过求解一个具体的线性方程组来说明这一方法的计算步骤。

$$2x_1+3x_2+x_3=1 \tag{1-1}$$
$$x_1+2x_2-x_3=-3 \tag{1-2}$$
$$4x_1+2x_2-x_3=0 \tag{1-3}$$

写成矩阵形式便是：

$$\begin{bmatrix}2&3&1\\1&2&-1\\4&2&-1\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}=\begin{bmatrix}1\\-3\\0\end{bmatrix}$$

解：

第 1 步：为了提高计算的精度，在第 1 列找出绝对值最大的系数作为第 1 列的主元，这里是第 3 行的 4。为了使主元位于主对角线上，把方程组中的式(1－1)和式(1－3)交换位置，得到等效的方程组：

$$4x_1+2x_2-x_3=0 \tag{1-4}$$
$$x_1+2x_2-x_3=-3 \tag{1-5}$$
$$2x_1+3x_2+x_3=1 \tag{1-6}$$

写成矩阵形式便是：

$$\begin{bmatrix}4&2&-1\\1&2&-1\\2&3&1\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}=\begin{bmatrix}0\\-3\\1\end{bmatrix}$$

第 2 步：用主元除式(1－4)中各个元素，则方程组变为：

$$x_1+0.5x_2-0.25x_3=0 \tag{1-7}$$
$$x_1+2x_2-x_3=-3 \tag{1-8}$$
$$2x_1+3x_2+x_3=1 \tag{1-9}$$

写成矩阵形式便是：

$$\begin{bmatrix}1&0.5&-0.25\\1&2&-1\\2&3&1\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}=\begin{bmatrix}0\\-3\\0\end{bmatrix}$$

第 3 步：用高斯消去法将式(1－8)和式(1－9)中 x_1 的系数变为 0。

高斯消去法的算法原理如下：

对于一个 n 阶线性方程组，设当前主元位于第 i 行第 i 列(主元一定在主对角线上)，通过前面若干步骤的处理，已经将主元调整为 1，主元左边各个系数均已为零，则主元所在的第 i 行成为：

$$X_i+A_{i,i+1}X_{i+1}+\cdots+A_{i,n-1}X_{n-1}+A_{i,n}X_n=B_i \tag{1-10}$$

即：

$$X_i = B_i - (A_{i,i+1}X_{i+1} + \cdots + A_{i,n-1}X_{n-1} + A_{i,n}X_n) \tag{1-11}$$

这时待消元对象所在的第 k 行是：

$$A_{k,i}X_i + A_{k,i+1}X_{i+1} + \cdots + A_{k,n-1}X_{n-1} + A_{k,n}X_n = B_k \tag{1-12}$$

即：

$$X_i = [B_k - (A_{k,i+1}X_{i+1} + \cdots + A_{k,n-1}X_{n-1} + A_{k,n}X_n)]/A_{k,i} \tag{1-13}$$

由式(1-11)和式(1-13)可得：

$$(A_{k,i+1}X_{i+1} + \cdots + A_{k,n-1}X_{n-1} + A_{k,n}X_n) - A_{k,i} \times (A_{i,i+1}X_{i+1} + \cdots + A_{i,n-1}X_{n-1} + A_{i,n}X_n) = B_k - A_{k,i} \times B_i$$

即：

$$(A_{k,i+1} - A_{k,i} \times A_{i,i+1})X_{i+1} + \cdots + (A_{k,n} - A_{k,i} \times A_{i,n})X_n = B_k - A_{k,i} \times B_i \tag{1-14}$$

这时就得到一个 X_i 的系数为零的式(1-14)，用式(1-14)取代式(1-12)式便完成了第 k 行的消元过程。仔细观察式(1-14)，就可以得到消元的具体过程为：

$$\text{新式}(k) = \text{原式}(k) - A_{k,i} \times \text{式}(i)$$

对于这个线性方程组，目前主元是 A_{11}，并且 A_{11} 已经为 1，现在需要将 A_{21} 和 A_{31} 进行消元处理，具体过程如下：

$$\text{式}(1-8) - A_{21} \times \text{式}(1-7) = \text{式}(1-8) - \text{式}(1-7)$$

$$\text{式}(1-9) - A_{31} \times \text{式}(1-7) = \text{式}(1-9) - 2 \times \text{式}(1-7)$$

消元后得到线性方程组为：

$$x_1 + 0.5x_2 - 0.25x_3 = 0 \tag{1-15}$$

$$1.5x_2 - 0.75x_3 = -3 \tag{1-16}$$

$$2x_2 + 1.5x_3 = 1 \tag{1-17}$$

写成矩阵形式便是：

$$\begin{bmatrix} 1 & 0.5 & -0.25 \\ 0 & 1.5 & -0.75 \\ 0 & 2 & 1.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ -3 \\ 1 \end{bmatrix}$$

第 4 步：与第 1 步相同，在第 2 列找出绝对值最大的系数作为第 2 列的主元，不过查找范围是从第 2 行开始的以下各行。查找结果是第 3 行的 2。为了使主元位于主对角线上，把方程组中的式(1-16)和式(1-17)交换位置，得到等效的方程组：

$$x_1 + 0.5x_2 - 0.25x_3 = 0 \tag{1-18}$$

$$2x_2 + 1.5x_3 = 1 \tag{1-19}$$

$$1.5x_2 - 0.75x_3 = -3 \tag{1-20}$$

写成矩阵形式便是：

$$\begin{bmatrix}1 & 0.5 & -0.25\\ 0 & 2 & 1.5\\ 0 & 1.5 & -0.75\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=\begin{bmatrix}0\\ 1\\ -3\end{bmatrix}$$

第 5 步：与第 2 步相同，用主元除式(1－19)各个元素，则方程组变为：

$$x_1+0.5x_2-0.25x_3=0 \tag{1-21}$$

$$x_2+0.75x_3=0.5 \tag{1-22}$$

$$1.5x_2-0.75x_3=-3 \tag{1-23}$$

写成矩阵形式便是：

$$\begin{bmatrix}1 & 0.5 & -0.25\\ 0 & 1 & 0.75\\ 0 & 1.5 & -0.75\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=\begin{bmatrix}0\\ 0.5\\ -3\end{bmatrix}$$

第 6 步：与第 3 步相同，用高斯消去法将式(1－21)和式(1－23)中 x_2 的系数变为 0。具体过程如下：

$$式(1-21)-A_{12}\times式(1-22)=式(1-21)-0.5\times式(1-22)$$

$$式(1-23)-A_{32}\times式(1-22)=式(1-23)-1.5\times式(1-22)$$

消元后得到线性方程组为：

$$x_1-0.625x_3=-0.25 \tag{1-24}$$

$$x_2+0.75x_3=0.5 \tag{1-25}$$

$$-1.875x_3=-3.75 \tag{1-26}$$

写成矩阵形式便是：

$$\begin{bmatrix}1 & 0 & -0.625\\ 0 & 1 & 0.75\\ 0 & 0 & -1.875\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=\begin{bmatrix}-0.25\\ 0.5\\ -3.75\end{bmatrix}$$

第 7 步：与第 1 步相同，在第 3 列找出绝对值最大的系数作为第 3 列的主元，不过查找范围是从第 3 行开始的以下各行。因为第 3 行是最后一行，主元就是第 3 行的－1.875。

第 8 步：与第 2 步相同，用主元除式(1－26)中各个元素，则方程组变为：

$$x_1-0.625x_3=-0.25 \tag{1-27}$$

$$x_2+0.75x_3=0.5 \tag{1-28}$$

$$x_3=2 \tag{1-29}$$

写成矩阵形式便是：

$$\begin{bmatrix}1 & 0 & -0.625\\ 0 & 1 & 0.75\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=\begin{bmatrix}-0.25\\ 0.5\\ 2\end{bmatrix}$$

第 9 步：与第 3 步相同，用高斯消去法将式(1－27)和式(1－28)中 x_3 的系数变为 0。具体过程如下：

$$式(1-27)-A_{13}\times式(1-29)=式(1-27)-(-0.625)\times式(1-29)$$

$$式(1-28)-A_{23}\times式(1-29)=式(1-28)-0.75\times式(1-29)$$

消元后得到线性方程组为：

$$x_1 = 1 \tag{1-30}$$

$$x_2 = -1 \tag{1-31}$$

$$x_3 = 2 \tag{1-32}$$

写成矩阵形式便是：

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=\begin{bmatrix}1\\ -1\\ 2\end{bmatrix}$$

至此，方程组的 3 个根全部解出。分析总结以上求解过程，可以看出：每个未知数的系数对应一列系数，为了提高计算精度，从中找出绝对值最大的系数作为主元，并将主元调整到主对角线上。为了减少消元过程中的运算量，先将主元除该行各个系数和常数项，使主元归一化。其他各行通过和主元所在行进行线性运算，就可以将其他各行中主元对应列的系数变为 0(消元过程)。最后除主对角线各个系数均为 1 外，其他系数全部为零，成为单位系数矩阵，这时各个常数项的值就是各个根的值。无回代过程的主元消去法解线性方程组的算法流程图如图 1－1 所示。

无回代过程的主元消去法解线性方程组的程序如下：

程序清单 1－1　无回代过程的主元消去法程序

```
#include <stdio.h>
#include <math.h>          //需要使用其中的绝对值函数 fabs()
#define N   3              //线性方程组的阶数
float A[N][N+1];           //线性方程组的增广系数矩阵(将常数项 B 移到系数矩阵内,作为最后一列)
float X[N];                //线性方程组的解

void findmain(int i)  //寻找第 i 列的主元,并将其所在行交换到当前处理行位置上
{
    int j,k;
    float c;
    c=fabs(A[i][i]);k=i;                          //初始化主元在第 i 行
    for(j=i+1;j<N;j++)                            //寻找主元(绝对值最大)所在行
        if(fabs(A[j][i])>c){
            c=fabs(A[j][i]);
```

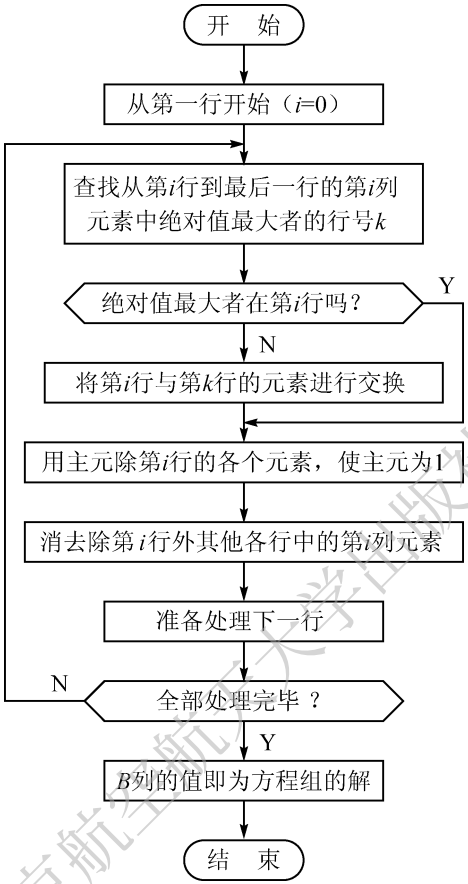

图 1－1　无回代过程主元消去法流程图

```
            k = j;
            }
    if (k! = i) for (j = 0;j< = N;j ++ ) {          //将主元所在行交换到当前处理行位置上
        c = A[k][j];
        A[k][j] = A[i][j];
        A[i][j] = c;
        }
}

void divmain (int i)                                //将主元所在行的各个系数除以主元,使主元为 1
{
    int j;
```

```
    float c;
    c = A[i][i];                                    //取出主元的原数值
    A[i][i] = 1.0;                                  //使主元为 1
    for (j = i + 1;j<= N;j ++ ) A[i][j]/ = c;       //将主元所在行的各个系数除以主元的
                                                    //原数值
}

void del (int i)                                    //进行第 i 列的消元处理
{
    int j,k;
    float c;
    for (j = 0;j<N;j ++ ) if (j! = i && A[j][i]) {
                                        //只处理除 i 行之外且 i 列系数非零的行(行下标为 j)
        c = A[j][i];A[j][i] = 0;        //记录被消系数的原数值,然后将其消零
        for (k = i + 1;k<= N;k ++ ) A[j][k] - = c * A[i][k];   //调整同行的其他系数
        }
}

int main ( )
{
    int i,j;
    printf("\n 请输入线性方程组的增广系数矩阵:\n");
    for(i = 0;i<N;i ++ ) {
        for(j = 0;j<N;j ++ ) {
            printf("A%d%d = ",i + 1,j + 1);
            scanf("%f",&A[i][j]);
            }
        printf("B%d = ",i + 1);
        scanf("%f",&A[i][N]);
        }
    getchar();
    for (i = 0;i<N;i ++ ) {                 //按行处理
        if (i<N - 1) findmain (i);          //寻找主元,并将其交换到当前处理行位置上
        divmain (i);                        //将当前处理行的各个系数除以主元,使主元为 1
        del (i);                            //进行消元处理
        }
    for (i = 0;i<N;i ++ ) X[i] = A[i][N];   //保存 n 阶线性方程组的解
    printf("\n 线性方程组的解:\n");
    for (i = 0;i<N;i ++ ) printf("X%d = %f     ",i + 1,X[i]);
```

```
    getchar();
    return 0;
}
```

1.1.2 有回代过程的主元消去法

在前面介绍的无回代过程的主元消去法中，主对角线右上方的系数在被消元前需要多次进行运算调整。以最右上方的常数项 B_1 为例，在式(1-4)中开始是 0，当将 A_{12} 进行消元时 B_1 被调整为式(1-24)中的 -0.25，当将 A_{13} 进行消元时 B_1 又被调整为式(1-30)中的 1。当线性方程组的未知数较多时，这种调整运算量就会明显增加。为了减少运算量，人们对前面的算法进行了改进，得到求解高阶线性方程组的有回代过程的主元消去算法。我们仍然以前面的方程组为例来说明算法过程：

第 1 步～第 5 步：算法与无回代过程算法相同。

第 6 步：与第 3 步相同，用高斯消去法将式(1-23)中 x_2 的系数变为 0。具体过程如下：

$$式(1-23)-A_{32}\times式(1-22)=式(1-23)-1.5\times式(1-22)$$

式(1-21)和式(1-22)维持不变，则方程组变为：

$$x_1+0.5x_2-0.25x_3=0 \tag{1-33}$$

$$x_2+0.75x_3=0.5 \tag{1-34}$$

$$-1.875x_3=-3.75 \tag{1-35}$$

写成矩阵形式便是：

$$\begin{bmatrix}1 & 0.5 & -0.25\\ 0 & 1 & 0.75\\ 0 & 0 & -1.875\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=\begin{bmatrix}0\\ 0.5\\ -3.75\end{bmatrix}$$

这里开始显示出和无回代过程算法的区别，它不是将主元所在列的其他系数全部消元，只是将主元下方的系数进行消元处理，即只将主对角线左下方的系数进行消元处理。

第 7 步：与第 1 步相同，在第 3 列找出绝对值最大的系数(即第 3 列的主元)，不过查找范围是从第 3 行开始的以下的各行。因为第 3 行是最后一行，主元就是第 3 行的 -1.875。

第 8 步：与第 2 步相同，用主元除式(1-35)中各个元素，则方程组变为：

$$x_1+0.5x_2-0.25x_3=0 \tag{1-36}$$

$$x_2+0.75x_3=0.5 \tag{1-37}$$

$$x_3=2 \tag{1-38}$$

写成矩阵形式便是：

$$\begin{bmatrix}1 & 0.5 & -0.25\\ 0 & 1 & 0.75\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=\begin{bmatrix}0\\ 0.5\\ 2\end{bmatrix}$$

至此，主对角线左下方的全部系数均为零，主对角线系数全部为 1，这时最后一行的常数项就是一个根，即 $X_n=B_n$，消元过程结束。

第 9 步：从倒数第 2 行开始回代过程，将式(1－38)的结果代入式(1－37)可得：

$$x_2=0.5-0.75x_3=0.5-0.75\times 2=-1$$

则方程组变为：

$$x_1+0.5\ x_2-0.25x_3=0 \tag{1-39}$$

$$x_2=-1 \tag{1-40}$$

$$x_3=2 \tag{1-41}$$

写成矩阵形式便是：

$$\begin{bmatrix}1 & 0.5 & -0.25\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=\begin{bmatrix}0\\ -1\\ 2\end{bmatrix}$$

第 10 步：继续回代过程，将式(1－40)和式(1－41)的结果代入式(1－39)可得：

$$x_1=0-0.5\ x_2+0.25x_3=-0.5\times(-1)+0.25\times 2=1$$

则方程组变为：

$$x_1=1 \tag{1-42}$$

$$x_2=-1 \tag{1-43}$$

$$x_3=2 \tag{1-44}$$

写成矩阵形式便是：

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}=\begin{bmatrix}1\\ -1\\ 2\end{bmatrix}$$

至此方程求解完毕。我们仔细分析回代过程可以看出，由于回代过程是从下向上进行的，当进行第 i 行的回代运算(求解 X_i)时，从第 $i+1$ 行到最后一行均已成为 $X_k=B_k$ 的形式，而：

$$X_i+A_{i,i+1}X_{i+1}+A_{i,i+2}X_{i+2}+\cdots+A_{i,n}X_n=B_i$$

故：

$$\begin{aligned}X_i&=B_i-(A_{i,i+1}X_{i+1}+A_{i,i+2}X_{i+2}+\cdots+A_{i,n}X_n)\\&=B_i-(A_{i,i+1}B_{i+1}+A_{i,i+2}B_{i+2}+\cdots+A_{i,n}B_n)\end{aligned}$$

有回代过程的算法显然比无回代过程的算法要麻烦一些，故这种算法只有在求解高阶线性方程组时才使用，以减少总的运算次数。

有回代过程的主元消去法程序流程图如图 1－2 所示。

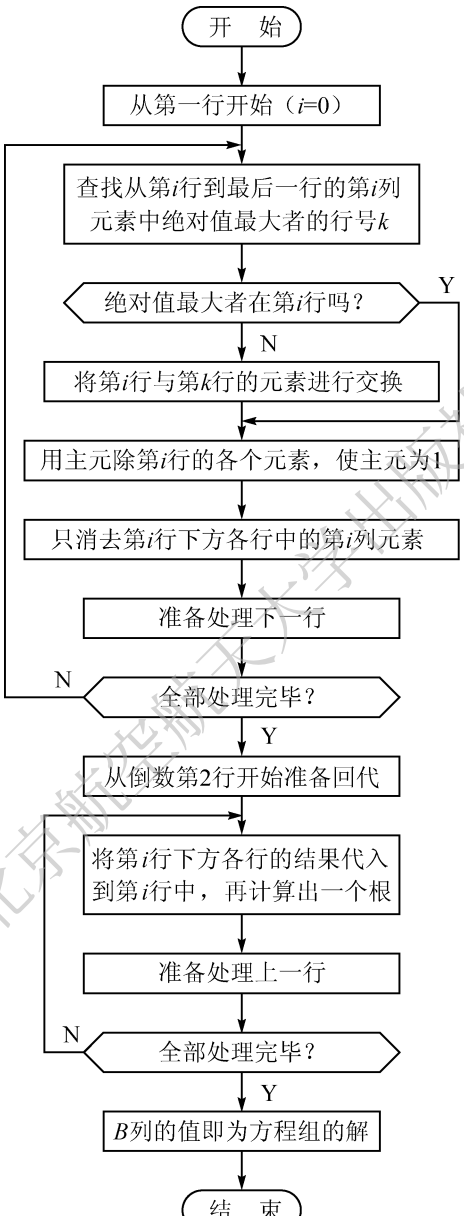

图 1－2　有回代过程主元消去法流程图

有回代过程的主元消去法程序如下：

程序清单 1－2　有回代过程的主元消去法程序

```
#include <stdio.h>
#include <math.h>                                   //需要使用其中的绝对值函数 fabs()
#define N   3                                       //线性方程组的阶数
float A[N][N+1];                                    //线性方程组的增广系数矩阵
float X[N];                                         //线性方程组的解

void findmain(int i)                    //寻找第 i 列的主元,并将其所在行交换到当前处理行位置上
{
    int j,k;
    float c;
    c=fabs(A[i][i]);k=i;                            //初始化主元在第 i 行
    for(j=i+1;j<N;j++)                              //在第 i 行以下寻找主元所在行
        if(fabs(A[j][i])>c){
            c= fabs(A[j][i]);
            k=j;
            }
    if(k!=i) for(j=0;j<=N;j++){                     //将主元所在行交换到当前处理行位置上
        c=A[k][j];
        A[k][j]=A[i][j];
        A[i][j]=c;
        }
}

void divmain(int i)                                 //将主元所在行的各个系数除以主元,使主元为 1
{
    int j;
    float c;
    c=A[i][i];                                      //取主元的原数值
    A[i][i]=1.0;                                    //使主元为 1
    for(j=i+1;j<=N;j++) A[i][j]/=c;                 //将主元所在行的各个系数除以主元的原数值
}

void del(int i)                                     //进行第 i 列的消元处理
{
    int j,k;
    float c;
```

```
    for (j = i + 1;j<N;j ++ ) if (A[j][i]) {    //从 i + 1 行开始进行消元处理
        c = A[j][i];A[j][i] = 0;                //记录被消系数的原数值,然后将其消零
        for (k = i + 1;k< = N;k ++ ) A[j][k] - = c * A[i][k];      //调整同行的其他系数
        }
}

int main ( )
{
    int i,j;
    printf("\n 请输入线性方程组的增广系数矩阵: \n");
    for(i = 0;i<N;i ++ ) {
        for(j = 0;j<N;j ++ ) {
            printf("A % d % d = ",i + 1,j + 1);
            scanf(" % f",&A[i][j]);
            }
        printf("B % d = ",i + 1);
        scanf(" % f",&A[i][N]);
        }
    getchar();
    for (i = 0;i<N;i ++ ) {                 //按行处理
        if (i<N - 1) findmain (i);          //寻找主元,并将其所在行交换到当前处理行位置上
        divmain (i);                        //将当前处理行的各个系数除以主元,使主元为 1
        if (i<N - 1) del (i);               //进行消元处理
        }
    for (i = N - 2;i> = 0;i -- )            //回代过程
        for (j = N - 1;j> i;j -- ) A[i][N] - = A[i][j] * A[j][N];
    for (i = 0;i<N;i ++ ) X[i] = A[i][N];       //保存 n 阶线性方程组的解
    printf("\n 线性方程组的解: \n");
    for (i = 0;i<N;i ++ ) printf("X % d = % f     ",i + 1,X[i]);
    getchar();
    return 0;
}
```


1.2 行列式法

1.2.1 行列式法概述

直接解线性方程组的另一种方法就是行列式法,又称克莱姆法则。其基本步骤如下:

对于线性方程组：

$$\begin{cases} A_{11}X_1 + A_{12}X_2 + \cdots + A_{1n}X_n = B_1 \\ A_{21}X_1 + A_{22}X_2 + \cdots + A_{2n}X_n = B_2 \\ \quad\quad\vdots \\ A_{n1}X_1 + A_{n2}X_2 + \cdots + A_{mn}X_n = B_n \end{cases}$$

如果该方程组的系数行列式不等于零，即：

$$D = \begin{vmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{mn} \end{vmatrix} \neq 0$$

那么，该方程组有唯一的一组解：

$$X_1 = D_1/D, \quad X_2 = D_2/D, \quad \cdots, \quad X_j = D_j/D, \quad \cdots, \quad X_n = D_n/D$$

这里 $D_j(j=1,2,\cdots,n)$是把系数行列式 D 中第 j 列的元素用方程组右端的常数项代替后所得的 n 阶行列式，即：

$$D_j = \begin{vmatrix} A_{11} & A_{12} & \cdots & A_{1,j-1} & B_1 & A_{1,j+1} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2,j-1} & B_2 & A_{2,j+1} & \cdots & A_{2n} \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{n,j-1} & B_n & A_{n,j+1} & \cdots & A_{mn} \end{vmatrix}$$

对于未知数的个数多于 3 个($n>3$)的线性方程组，要计算高阶行列式的值 D 就比较复杂，而且阶数愈高愈复杂。当 $n\leqslant 3$ 时，各行列式值的计算过程简单易行。因此，我们只讨论三元线性方程组的行列式解法。

1.2.2 三元线性方程组的行列式法

我们来研究三元线性方程组：

$$\begin{cases} A_{11}X + A_{12}Y + A_{13}Z = B_1 \\ A_{21}X + A_{22}Y + A_{23}Z = B_2 \\ A_{31}X + A_{32}Y + A_{33}Z = B_3 \end{cases}$$

未知数 X、Y、Z 的系数组成的三阶行列式：

$$D = \begin{vmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{vmatrix}$$

人们发现展开行列式的规律：在行列式 D 下重写第 1 行和第 2 行，如图 1-3 所示。

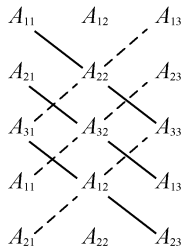

图 1-3　行列式 D 的展开方法

行列式 D 的展开式为：

$$D = A_{11}A_{22}A_{33} + A_{21}A_{32}A_{13} + A_{31}A_{12}A_{23} - A_{31}A_{22}A_{13} - A_{11}A_{32}A_{23} - A_{21}A_{12}A_{33}$$

配合图 1-3 可以看出，行列式 D 的值等于各实线上 3 元素乘积之和减去各虚线上 3 元素乘积之和。此法便于记忆，照此可计算其他三阶行列式的值，解三元线性方程组时可利用它。例如解线性方程组：

$$\begin{cases} 2X + 3Y + Z = 1 \\ X + 2Y - Z = -3 \\ 4X + 2Y - Z = 0 \end{cases}$$

解： 用行列式法得：

$$D = \begin{vmatrix} 2 & 3 & 1 \\ 1 & 2 & -1 \\ 4 & 2 & -1 \end{vmatrix} = -4 + 2 - 12 - 8 + 4 + 3 = -15$$

$$D_1 = \begin{vmatrix} 1 & 3 & 1 \\ -3 & 2 & -1 \\ 0 & 2 & -1 \end{vmatrix} = -15, \quad D_2 = \begin{vmatrix} 2 & 1 & 1 \\ 1 & -3 & -1 \\ 4 & 0 & -1 \end{vmatrix} = 15, \quad D_3 = \begin{vmatrix} 2 & 3 & 1 \\ 1 & 2 & -3 \\ 4 & 2 & 0 \end{vmatrix} = -30$$

因此，方程组的解为：

$$\begin{cases} X = D_1/D = 1 \\ Y = D_2/D = -1 \\ Z = D_3/D = 2 \end{cases}$$

解三元一次方程组的算法程序如下：

程序清单 1-3　行列式法解三元一次方程组的程序

```
#include <stdio.h>
float A[3][3];                    //三元一次方程组的系数矩阵
float B[3];                       //三元一次方程组的常数项
float X[3];                       //三元一次方程组的解
```

```
float DETA ( )                          //行列式计算
{
    return (A[0][0] * A[1][1] * A[2][2] + A[1][0] * A[2][1] * A[0][2] + A[2][0] * A[0][1] * A[1][2]
      - A[2][0] * A[1][1] * A[0][2] - A[0][0] * A[2][1] * A[1][2] - A[1][0] * A[0][1] * A[2][2]);
}

void swap (int k)                       //常数项与 k 列系数交换
{
    int i;
    float t;
    for (i = 0;i<3;i ++ ) {
        t = B[i];
        B[i] = A[i][k];
        A[i][k] = t;
        }
}

void XYZ ( )                            //求解三元一次方程组
{
    float d;
    d = DETA( );                        //求系数行列式的值
    swap (0);                           //将 X 的系数与常数项交换
    X[0] = DETA( )/d;                   //解出 X
    swap (0);                           //恢复原来的系数与常数项
    swap (1);                           //将 Y 的系数与常数项交换
    X[1] = DETA( )/d;                   //解出 Y
    swap (1);                           //恢复原来的系数与常数项
    swap (2);                           //将 Z 的系数与常数项交换
    X[2] = DETA( )/d;                   //解出 Z
    swap (2);                           //恢复原来的系数与常数项
}

int main ( )
{
    int i,j;
    printf("\n 请输入三元一次方程组的增广系数矩阵：\n");
    for(i = 0;i<3;i ++ ) {
        for(j = 0;j<3;j ++ ) {
            printf("A%d%d = ",i + 1,j + 1);
```

```
            scanf("%f",&A[i][j]);
            }
        printf("B%d=",i+1);
        scanf("%f",&B[i]);
        }
    getchar();
    XYZ();
    printf("\n线性方程组的解:\n");
    for(i=0;i<3;i++) printf("X%d=%f    ",i+1,X[i]);
    getchar();
    return 0;
}
```

1.3 应用实例

智能仪器都有较强的数据处理能力,能够将传感器检测到的大量原始数据进行处理,从各种干扰背景中提取有用信息,并通过进一步的计算,按规定的格式直接输出人们需要的结果。在进行数据处理时,输出结果和输入的原始数据之间是一种函数关系,其中有若干个称为仪器系数的常量。由于仪器硬件性能的老化(主要是传感器性能的老化)和环境参数的变化,使得输出结果的精度也会发生变化,为了保持原有的精度,必须对仪器进行校正。校正的方法是对一个或若干个"标准样品"进行检测,看看输出结果的精度是否符合要求,如果误差超标就调整仪器系数,使之达到规定要求,这种通过对标样进行检测来校正仪器的过程称为"仪器的标定"。在过去的模拟电子仪器时代,仪器系数的校正一般是通过调节电位器(改变放大器的增益)来实现。在带单片机的智能仪器中,仪器系数已经数字化,可以通过键盘来修改。在这里,介绍一台便携式 256 道微机能谱仪的仪器系数自动标定算法,作为一个求解线性方程组的应用实例。这种算法在检测完已知的标样后能自动将仪器系数校正好,不需要人工介入。

1.3.1 数学模型分析

便携式能谱仪通过检测放射线的强度和能量分布,计算出检测点地质环境中放射性元素的种类和含量,主要用于放射性地质普查。本仪器检测的对象是 3 种放射性元素,分别为铀、钍和钾(指放射性同位素 K^{40})。这 3 种放射性元素都能产生能量分布很广的放射线,形成互相重叠的能谱曲线。仪器通过探头检测到的能谱曲线是各种放射性元素产生的能谱曲线的合成。要想从这种合成能谱曲线中区分出各种不同的放射性元素来,必须利用各种不同元素的放射线能量分布的差异。在每种放射性元素的能谱曲线上,都有

一个或几个放射线能量比较集中的“峰”，而在这个能量位置上其他放射性元素产生的放射线强度相对较弱，这就是这种放射性元素的“特征峰”。以特征峰为中心，向左右各扩展一定范围，在能谱曲线上开出一个窗口，则窗口内检测到的放射线强度与该种放射性元素的含量强烈相关。在这台便携式 256 道能谱仪中，铀的特征峰为 138 道、钍的特征峰为 196 道、钾的特征峰为 115 道。铀的窗口范围为 132 道～146 道、钍的窗口范围为 190 道～210 道、钾的窗口范围为 110 道～122 道。习惯上将这 3 个窗口分别称为铀道、钍道和钾道，并对应地将这 3 个窗口中检测到的放射线计数率(CPS)分别称为铀道计数率、钍道计数率和钾道计数率。由于各种放射性元素的能谱互相重叠，所以各道计数率不仅与对应的放射性元素含量有关，也与其他种类放射性元素的含量有关。即使探头周围没有任何放射性物质，仪器也能检测到一定量的放射线，并使各道计数率有一个不为零的值，这个值称为“本底”，它是由宇宙射线等原因造成的。由此，可以得到：

$$N_u = r_1 Q_u + r_2 Q_t + r_3 Q_k + N_{u0} \tag{1-45}$$

$$N_t = t_1 Q_u + t_2 Q_t + t_3 Q_k + N_{t0} \tag{1-46}$$

$$N_k = k_1 Q_u + k_2 Q_t + k_3 Q_k + N_{k0} \tag{1-47}$$

式中，N_u、N_t、N_k——仪器实测的铀道计数率、钍道计数率和钾道计数率。

N_{u0}、N_{t0}、N_{k0}——铀道本底、钍道本底和钾道本底。

Q_u、Q_t、Q_k——地质环境中的铀含量、钍含量和钾含量。

r_1、r_2、r_3——铀、钍和钾元素含量对铀道计数率的影响系数。

t_1、t_2、t_3 ——铀、钍和钾元素含量对钍道计数率的影响系数。

k_1、k_2、k_3——铀、钍和钾元素含量对钾道计数率的影响系数。

如果 9 个系数和 3 个本底值已知，在检测到地质环境中的铀道计数率、钍道计数率和钾道计数率后，便可以通过求解三元一次方程组得到地质环境中的铀含量、钍含量和钾含量。三元一次方程组如下：

$$r_1 Q_u + r_2 Q_t + r_3 Q_k = N_U \tag{1-48}$$

$$t_1 Q_u + t_2 Q_t + t_3 Q_k = N_T \tag{1-49}$$

$$k_1 Q_u + k_2 Q_t + k_3 Q_k = N_K \tag{1-50}$$

式中，$N_U = N_u - N_{u0}$——扣除本底后，铀道净计数率。

$N_T = N_t - N_{t0}$ ——扣除本底后，钍道净计数率。

$N_K = N_k - N_{k0}$——扣除本底后，钾道净计数率。

用行列式法求解后，得到：

$$Q_u = R_1 N_U + R_2 N_T + R_3 N_K \tag{1-51}$$

$$Q_t = T_1 N_U + T_2 N_T + T_3 N_K \tag{1-52}$$

$$Q_k = K_1 N_U + K_2 N_T + K_3 N_K \tag{1-53}$$

式中，R_1、R_2、R_3——铀、钍和钾各道计数率对铀元素含量的影响系数。

T_1、T_2、T_3 ——铀、钍和钾各道计数率对钍元素含量的影响系数。

K_1、K_2、K_3——铀、钍和钾各道计数率对钾元素含量的影响系数。

这 9 个大写字母系数可以由前面的 9 个小写字母系数变换得到。求得这 9 个大写字母系数后，就可以通过检测得到的各道计数率减去各道本底，得到各道的净计数率，然后代入以上 3 式中直接得到 3 种放射性元素的含量。故仪器标定的目标就是求得这 9 个大写字母系数和 3 个本底计数率。

仪器在工作时是已知仪器系数和本底，测得各道计数率，然后计算出各种放射性元素的含量。仪器在标定时，各个标准模型中各种放射性元素的含量已知，在测得各道计数率后，再计算出仪器系数和各道本底。在标定时，一般有 5 个地质模型可供使用，它们的各种放射性元素含量均已知，只是含量各不相同：

- 铀模型的铀含量较高，钍和钾的含量较低。
- 钍模型的钍含量较高，铀和钾的含量较低。
- 钾模型的钾含量较高，铀和钍的含量较低。
- 混合模型的铀、钍和钾的含量均较高。
- 零值模型的铀、钍和钾的含量均很低，接近本底。

1.3.2 算法设计

由于 9 个大写字母系数可以由 9 个小写字母系数变换得到，故标定过程分为两步，第 1 步通过在已知模型上进行检测，计算出 9 个小写字母系数和 3 个本底计数率，第 2 步再由 9 个小写字母系数推导出 9 个大写字母系数。

9 个小写字母系数和 3 个本底计数率一共是 12 个未知数，需要 12 个互相独立的线性方程来求解。在一个模型上进行检测后可以得到 3 个计数率(铀道计数率、钍道计数率和钾道计数率)，也就可以列出 3 个线性方程，因此，对其中 4 个模型进行检测后就可以得到 12 个线性方程，求解出这 12 个未知数，再用一个模型(一般是混合模型)来进行标定结果的评价。

求解 12 元线性方程组是比较麻烦的，我们采用分而治之的方法将 3 个本底计数率与 9 个系数分开处理。用迭代算法求得 3 个本底计数率，用解 3 个三元线性方程组的方法求解 9 个系数。具体算法如下：

① 3 个本底计数率初始化为 0。

② 将铀道本底计数率 N_{u0} 当作常数看待，用在铀模型、钍模型和钾模型上分别检测得到的铀道计数率代入式(1－45)，得到三元线性方程组：

$$N_{Uu} = r_1 Q_{Uu} + r_2 Q_{Ut} + r_3 Q_{Uk} + N_{u0} \tag{1-54}$$

$$N_{Tu} = r_1 Q_{Tu} + r_2 Q_{Tt} + r_3 Q_{Tk} + N_{u0} \tag{1-55}$$

$$N_{Ku} = r_1 Q_{Ku} + r_2 Q_{Kt} + r_3 Q_{Kk} + N_{u0} \tag{1-56}$$

式中，N_{Uu}、N_{Tu}、N_{Ku}——铀模型、钍模型和钾模型上分别检测得到的铀道计数率。

Q_{Uu}、Q_{Ut}、Q_{Uk}——铀模型中的铀含量、钍含量和钾含量(已知)。

Q_{Tu}、Q_{Tt}、Q_{Tk}——钍模型中的铀含量、钍含量和钾含量(已知)。

Q_{Ku}、Q_{Kt}、Q_{Kk}——钾模型中的铀含量、钍含量和钾含量(已知)。

求解以上3式，得到3个系数r_1、r_2和r_3。

③ 将钍道本底计数率N_{t0}当作常数看待，用在铀模型、钍模型和钾模型上分别检测得到的钍道计数率代入式(1-46)，得到三元线性方程组：

$$N_{Ut} = t_1 Q_{Uu} + t_2 Q_{Ut} + t_3 Q_{Uk} + N_{t0} \tag{1-57}$$

$$N_{Tt} = t_1 Q_{Tu} + t_2 Q_{Tt} + t_3 Q_{Tk} + N_{t0} \tag{1-58}$$

$$N_{Kt} = t_1 Q_{Ku} + t_2 Q_{Kt} + t_3 Q_{Kk} + N_{t0} \tag{1-59}$$

式中，N_{Ut}、N_{Tt}、N_{Kt}——铀模型、钍模型和钾模型上分别检测得到的钍道计数率。

求解以上3式，得到3个系数t_1、t_2和t_3。

④ 将钾道本底计数率N_{k0}当作常数看待，用在铀模型、钍模型和钾模型上分别检测得到的钾道计数率代入式(1-47)，得到三元线性方程组：

$$N_{Uk} = k_1 Q_{Uu} + k_2 Q_{Ut} + k_3 Q_{Uk} + N_{k0} \tag{1-60}$$

$$N_{Tk} = k_1 Q_{Tu} + k_2 Q_{Tt} + k_3 Q_{Tk} + N_{k0} \tag{1-61}$$

$$N_{Kk} = k_1 Q_{Ku} + k_2 Q_{Kt} + k_3 Q_{Kk} + N_{k0} \tag{1-62}$$

式中，N_{Uk}、N_{Tk}、N_{Kk}——铀模型、钍模型和钾模型上分别检测得到的钾道计数率。

求解以上3式，得到3个系数k_1、k_2和k_3。

⑤ 将求出的9个系数和零值模型的放射性元素含量代入式(1-48)、式(1-49)和式(1-50)，得：

$$r_1 Q_{Zu} + r_2 Q_{Zt} + r_3 Q_{Zk} = N_U \tag{1-63}$$

$$t_1 Q_{Zu} + t_2 Q_{Zt} + t_3 Q_{Zk} = N_T \tag{1-64}$$

$$k_1 Q_{Zu} + k_2 Q_{Zt} + k_3 Q_{Zk} = N_K \tag{1-65}$$

式中，Q_{Zu}、Q_{Zt}、Q_{Zk}——零值模型中的铀含量、钍含量和钾含量(已知)。

由以上3式计算出零值模型理论上的铀道净计数率、钍道净计数率和钾道净计数率。

⑥ 将零值模型的铀道实测计数率、钍道实测计数率和钾道实测计数率分别减去铀道净计数率、钍道净计数率和钾道净计数率，得到铀道本底计数率、钍道本底计数率和钾道本底计数率。

⑦ 将新的各道本底计数率看作常数，返回第2步，重新计数9个系数，如此反复迭代5次后，3个本底计数率和9个系数的数值基本上稳定下来，再增加迭代次数已经没有意义。

⑧ 通过式(1-48)、式(1-49)、式(1-50)，可以得到式(1-51)、式(1-52)、式(1-53)，从而由9个小写字母系数转换成9个大写字母系数：

$$
\begin{aligned}
R_1 &= (t_2k_3 - t_3k_2)/\Delta \\
R_2 &= (r_3k_2 - r_2k_3)/\Delta \\
R_3 &= (r_2t_3 - r_3t_2)/\Delta \\
T_1 &= (t_3k_1 - t_1k_3)/\Delta \\
T_2 &= (r_1k_3 - r_3k_1)/\Delta \\
T_3 &= (r_3t_1 - r_1t_3)/\Delta \\
K_1 &= (t_1k_2 - t_2k_1)/\Delta \\
K_2 &= (r_2k_1 - r_1k_2)/\Delta \\
K_3 &= (r_1t_2 - r_2t_1)/\Delta
\end{aligned}
$$

式中，$\Delta = \begin{vmatrix} r_1 & r_2 & r_3 \\ t_1 & t_2 & t_3 \\ k_1 & k_2 & k_2 \end{vmatrix} = r_1t_2k_3 + r_2t_3k_1 + r_3t_1k_2 - r_1t_3k_2 - r_2t_1k_3 - r_3t_2k_1$

⑨ 将混合模型检测得到的各道计数率减去刚刚得到的各道本底计数率，得到各道的净计数率，再利用刚刚计算出来的 9 个大写字母系数，通过式(1－51)、式(1－52)、式(1－53)计算出混合模型中 3 种放射性元素的含量，然后和混合模型的已知含量数据进行比较，如果误差在允许范围之内，则标定的仪器系数有效，结束标定工作。如果误差超过允许范围，则标定失败，必须重新进行标定工作(重新检测 5 个模型数据，重新计算仪器系数)。

1.3.3 程序设计

按前面分析的算法可以得到如图 1－4 所示的程序流程图。

仪器系数自动标定程序如下：

程序清单 1－4　仪器系数自动标定程序

```
#include <stdio.h>
float former[5][3] = {{137.4,9.61,0.59},     //5 个模型的含量数据(已知)
                      {5.05,284.3,0.37},
                      {2.43,8.42,5.71},
                      {118.9,270.9,3.05},
                      {2.89,7.3,0.9}};
float cps[5][3];                             //5 个模型各道计数率的测量结果
float coef[3][3];                            //9 个小写字母系数(中间数据)
float COEFF[3][3];                           //9 个大写字母系数(待标定)
float cps0[3];                               //3 个本底计数率(待标定)
float A[3][3];                               //存放三元一次方程组的系数矩阵
float B[3];                                  //存放三元一次方程组的常数项
float X[3];                                  //存放三元一次方程组的解
```

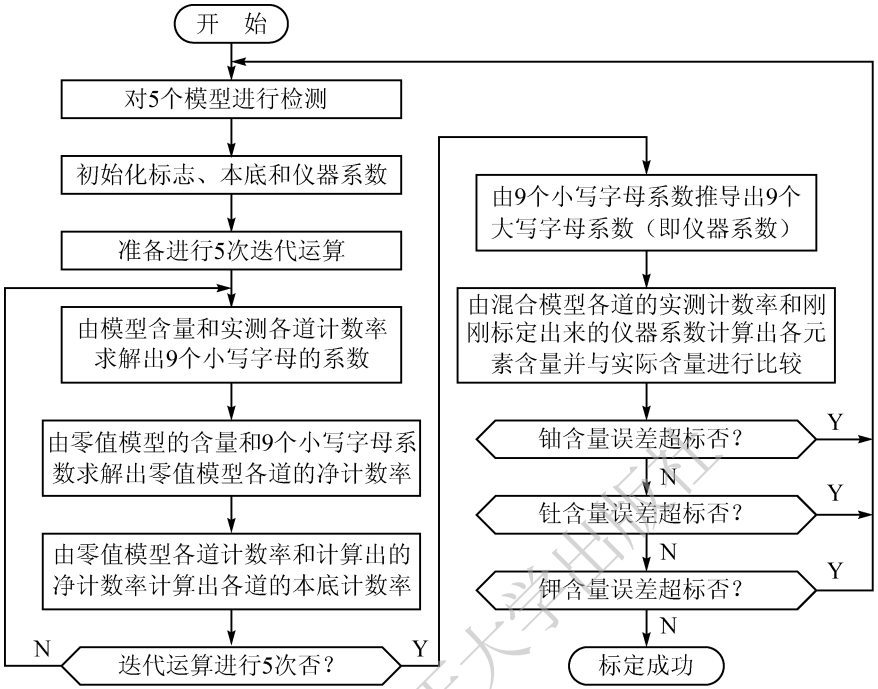

图 1-4　仪器系数自动标定算法

```
void test ( );                    //对 5 个模型进行检测,结果存入 cps 数组中,程序省略

float DETA ( )                    //行列式计算
{
    return (A[0][0] * A[1][1] * A[2][2] + A[1][0] * A[2][1] * A[0][2] + A[2][0] * A[0][1] * A[1][2]
      - A[2][0] * A[1][1] * A[0][2] - A[0][0] * A[2][1] * A[1][2] - A[1][0] * A[0][1] * A[2][2]);
}

void swap (int k)                 //常数项与 k 列系数交换
{
    int i;
    float t;
    for (i = 0;i<3;i ++ ) {
        t = B[i];
        B[i] = A[i][k];
        A[i][k] = t;
        }
```

```
}

void XYZ ( )                                          //求解三元一次方程组
{
    float d;
    d = DETA( );                                      //求系数行列式的值
    swap (0);                                         //将 X 的系数与常数项交换
    X[0] = DETA( )/d;                                 //解出 X
    swap (0);                                         //恢复原来的系数与常数项
    swap (1);                                         //将 Y 的系数与常数项交换
    X[1] = DETA( )/d;                                 //解出 Y
    swap (1);                                         //恢复原来的系数与常数项
    swap (2);                                         //将 Z 的系数与常数项交换
    X[2] = DETA( )/d;                                 //解出 Z
    swap (2);                                         //恢复原来的系数与常数项
}

void RTK (int k)                                      //求解第 k 行的 3 个小写字母系数
{
    int i,j;
    for (i = 0;i<3;i ++ )                             //准备方程组的系数
        for (j = 0;j<3;j ++ ) A[i][j] = former[i][j];
    for (i = 0;i<3;i ++ ) B[i] = cps[i][k] - cps0[k];  //准备方程组的常数项
    XYZ ();                                           //解出 3 个小写字母系数
    for (i = 0;i<3;i ++ ) coef[k][i] = X[i];          //保存 3 个小写字母系数
}

void conv ( )                                 //将 9 个小写字母系数转换成 9 个大写字母系数
{
    float d;
    int i,j;
    for (i = 0;i<3;i ++ )                                    //取 9 个小写字母系数
        for (j = 0;j<3;j ++ ) A[i][j] = coef[i][j];
    d = DETA( );                                             //求系数行列式的值
    COEFF[0][0] = (A[1][1] * A[2][2] - A[2][1] * A[1][2])/d;  //计算 9 个大写字母系数
    COEFF[0][1] = (A[2][1] * A[0][2] - A[0][1] * A[2][2])/d;
    COEFF[0][2] = (A[0][1] * A[1][2] - A[0][2] * A[1][1])/d;
    COEFF[1][0] = (A[1][2] * A[2][0] - A[1][0] * A[2][2])/d;
    COEFF[1][1] = (A[0][0] * A[2][2] - A[2][0] * A[0][2])/d;
```

```
    COEFF[1][2] = (A[1][0] * A[0][2] - A[0][0] * A[1][2])/d;
    COEFF[2][0] = (A[1][0] * A[2][1] - A[1][1] * A[2][0])/d;
    COEFF[2][1] = (A[0][1] * A[2][0] - A[0][0] * A[2][1])/d;
    COEFF[2][2] = (A[0][0] * A[1][1] - A[0][1] * A[1][0])/d;
}

 main ( )
{
    int  i,j;
    float c;
    while (1) {                          //只要标定尚未成功,就需要进行标定操作
        test ( );                        //首先对 5 个模型进行检测,将结果存入 cps[][]数组中
        for (i = 0;i<3;i ++ ) cps0[i] = 0;          //3 个本底计数率初始化为 0
        for (i = 0;i<5;i ++ ) {                     //进行 5 次迭代运算
            for (j = 0;j<3;j ++ ) RTK (j);          //计算 9 个小写字母系数
            for (j = 0;j<3;j ++ )                   //计算零值模型各道的净计数率
                B[j] = coef[j][0] * former[4][0] +
                coef[j][1] * former[4][1] + coef[j][2] * former[4][2];
            for (j = 0;j<3;j ++ ) cps0[j] = cps[4][j] - B[j];     //计算各道的本底计数率
            }
        conv ( );                            //将 9 个小写字母系数转换成 9 个大写字母系数
        for (i = 0;i<3;i ++ )                //计算混合模型各元素的含量
            B[i] = COEFF[i][0] * (cps[3][0] - cps0[0]) +
            COEFF[i][1] * (cps[3][1] - cps0[1]) + COEFF[i][2] * (cps[3][2] - cps0[2]);
        c = former[3][0]/B[0];
        if ( c<0.96 || c>1.04 ) continue;          //铀含量误差超过 4 %,标定失败,重测
        c = former[3][1]/B[1];
        if ( c<0.94 || c>1.06 ) continue;          //钍含量误差超过 6 %,标定失败,重测
        c = former[3][2]/B[2];
        if ( c<0.88 || c>1.12 ) continue;          //钾含量误差超过 12 %,标定失败,重测
        return (1);                                //标定成功,结束
        }
}
```

第 2 章

常用代数插值和曲线拟合算法

很多嵌入式系统都有信号检测功能，通过传感器对各类物理量进行测量。信号检测部分通常由传感器、信号调理电路和模/数转换电路组成。由于很多种类的传感器均存在一定程度的非线性(有的传感器甚至具有类似对数的转换特性)，即被检测的物理量与最终得到的数字量之间不是严格的线性关系，为了得到物理量的真实值，必须建立被检测物理量与转换后的数字量之间的对应关系(即“标度变换”)，建立这种对应关系的过程就是“仪器标定”。

进行仪器标定时，输入一个已知的物理量，得到一个 A/D 转换结果，就建立了一个对应关系，再输入另外一个已知的物理量，得到另外一个 A/D 转换结果，就又建立了一个对应关系，不断改变输入物理量的大小，就可以得到一系列对应的 A/D 转换结果。如果建立一个坐标系，用 y 轴表示被检测的物理量，用 x 轴表示 A/D 转换结果，则每一个对应关系就是这个坐标系中的一个点，n 个对应关系就由坐标图上的 n 个点表示，如图 2-1 所示。

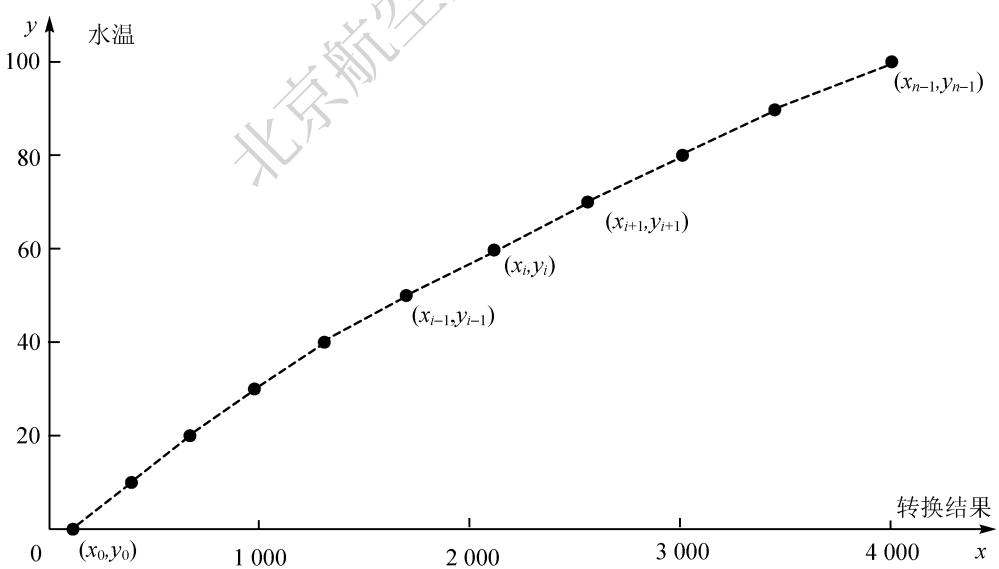

图 2-1　用坐标图表示对应关系

如果要完全掌握被检测物理量与 A/D 转换结果之间的对应关系，仪器标定的工作量将非常大。例如在采用 12 位 A/D 芯片的系统中，就有 4096 种可能的 A/D 转换结果，即存在 4096 个对应关系。标定全部对应关系不但工作量无法承受，记录全部对应关系也需要消耗大量硬件资源。实际可行的标定方法是：在被检测物理量的检测范围之内，只选择若干个检测点进行标定，然后利用这若干点的标定结果推算出全部对应关系。例如某热水器控制系统需要检测水温，检测范围为 0～100℃，采用 12 位 A/D，进行了 11 点的标定操作，得到如表 2－1 所列的 A/D 转换结果，其坐标图如图 2－1 所示。

表 2－1　A/D 转换结果与温度的对应关系

水温/℃	0	10	20	30	40	50	60	70	80	90	100
A/D 转换结果	86	376	687	1021	1379	1760	2164	2592	3043	3515	4008

当 A/D 转换结果是 1787 时，此时对应的水温是多少？根据表 2－1，水温应该在 40～50℃之间，能否再精确一些？完全可以。为此，想办法寻找一个函数形式来替代表 2－1，通过这个函数，就可以由 A/D 转换结果直接计算出对应的物理量。

若由 A/D 转换结果 x_i 可以得到对应的物理量 $y_i(i=0,1,2,\cdots,n)$，则它们之间一定存在着某种函数关系，记作 $y=f(x)$；但是 $f(x)$ 的数学表达式并不知道，甚至其关系无法用数学表达式来精确描述。如果能够找到某一函数 $g(x)$，使得 $g(x)$ 在 x_i 处与 $f(x)$ 相等，而在其他非标定点可以用 $g(x)$ 近似地代替 $f(x)$，则称 $g(x)$ 为 $f(x)$ 的插值函数，x_i 称为插值节点。也就是说，需要为一个未知的函数 $f(x)$ 寻找一个替身 $g(x)$，这个 $g(x)$ 必须满足以下条件：

- $g(x)$ 必须有一个明确的数学表达式，以便用它来计算任意 x 对应的 y 值，这是寻找它的最终目标。
- 在所有标定点（插值节点）上，$g(x_i)$ 与 $f(x_i)$ 等效，完全可以替代。
- 在标定点（插值节点）之间的区域，$g(x)$ 与 $f(x)$ 的误差很小，满足系统检测精度要求，用 $g(x)$ 来替代 $f(x)$ 才是可行的。

选择插值函数 $g(x)$ 的方法很多，函数类型不同，逼近 $f(x)$ 的效果就不同。如果选择的函数类型是代数多项式，就称为代数插值。从计算的观点看，插值问题就是在允许误差范围内用一简单函数表达式近似地代替某复杂关系，由已知点推算未知点，因而在数据处理中，插值有广泛的应用。

插值代数多项式的一般形式为：

$$g(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n$$

插值条件是：

$$g(x_i) = f(x_i) \qquad (i = 0,1,2,\cdots,n)$$

由插值条件就可以得到一个 $n+1$ 元的线性方程组，解此方程组，便可确定该多项式的各项系数(a_0，a_1，a_2，…，a_n)，从而确定 $g(x)$的具体形式。显然，一般说来，参与插值的节点愈多，$g(x)$的幂次就愈高，逼近的精度也就愈高，但是计算量也愈大。实际上人们还是根据允许的误差范围来确定 $g(x)$的阶次。在很多情况下，通过控制插值区间的范围，插值函数 $g(x)$为一次或二次多项式就可满足要求。因此，我们只讨论这两种插值算法，即线性插值(一次插值)算法和抛物线插值(二次插值)算法。

2.1 线性插值

2.1.1 算法原理

线性插值是代数插值的最简单的形式。假设变量 y 和自变量 x 的关系 $y=f(x)$如图 2-2中实曲线所示，插值函数 $y=g(x)$如图 2-2 中虚直线所示。

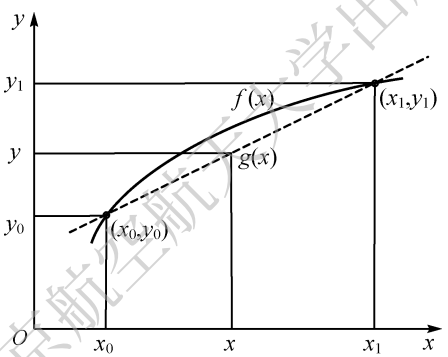

图 2-2　线性插值示意图

已知 y 在点 x_0和 x_1的对应值分别为 y_0和 y_1，现在以(x_0，y_0)和 (x_1，y_1)为两个插值节点，要求用一个线性插值函数 $g(x)=ax+b$，近似代替 $f(x)$。根据插值条件，应满足：

$$\begin{cases} ax_0+b=y_0 \\ ax_1+b=y_1 \end{cases}$$

解该方程组，便可确定线性插值函数 $g(x)$的参数 a 和 b。由图 2-2 可知，线性插值的几何意义是用通过点(x_0，y_0)和点(x_1，y_1)的直线近似地代替曲线 $y=f(x)$。很容易求得该直线表达式：

$$g(x)=y_0+\frac{y_1-y_0}{x_1-x_0}(x-x_0) \qquad \text{(点斜式)}$$

或：

$$g(x)=y_0\left(\frac{x_1-x}{x_1-x_0}\right)+y_1\left(\frac{x_0-x}{x_0-x_1}\right) \qquad \text{(两点式)}$$

若插值点 x 在 x_0 和 x_1 之间，称为内插，否则称为外插或外推。在这里，只考虑内插。由图 2-2 也可以看出，插值节点 x_0 和 x_1 之间的间距越小，那么在这一区间 $g(x)$ 与 $f(x)$ 之间的误差就越小，也就是说，当 x_0 和 x_1 之间的间距小到一定程度时，$f(x)$ 在 x_0 和 x_1 之间可以看做一条线段，与 $g(x)$ 几乎重合，完全可以用 $g(x)$ 来代替。但在实际情况下，x 的变化范围是比较大的，如果用一条直线 $g(x)$ 来代替 $f(x)$，误差太大，线性插值算法虽然简单，但效果太差。解决问题的办法是设置足够多的标定测试点（插值节点），将 $f(x)$ 分成足够多的小段，在每个小段采用线性插值算法。这时，$g(x)$ 不是一条直线，而是由一系列线段组成的折线，如图 2-1 中的虚线所示。因此，线性插值算法真正的应用方式是分段线性插值算法，或者说折线插值算法。

为了实现折线插值算法，必须记录所有标定结果，它们就是插值节点，即折线的一系列端点。为了记录这些端点，就需要两个表格，第 1 个表格为“标定”测试点 A/D 转换结果 $x_i(i=0,1,2,\cdots,n)$，第 2 个表格为对应的物理量精确值 $y_i(i=0,1,2,\cdots,n)$。实际计算时先将 A/D 转换结果和第 1 个表格中的各项数据进行比较，找到当前 A/D 转换结果所在的区间（即两个最邻近的节点）；然后在第 2 个表格中读出对应节点的精确值，再采用线性插值算法求出当前物理量的精确值。

如果传感器的非线性比较明显，而且各段的非线性程度不均匀，就要在非线性较明显的一段进行密集标定，而在非线性不明显的地方减少标定节点。使得在保证一定精度要求的前提下减少标定工作量和缩小表格规模。如果传感器的非线性不太明显，而且各段的非线性程度无明显差别，就可以按被检测物理量的范围均匀分布标定节点。在这种情况下，系统只需要一个表格记录各个标定节点的 A/D 转换结果就可以了。大多数传感器的非线性在测程范围之内都是不太明显的，通常采用均匀分布标定节点的办法只固化一个表格即可。

2.1.2 应用实例

用前面的温度检测作为应用实例，标定的结果见表 2-1。为了设计相关算法，仔细分析表 2-1，从中提取它所包含的信息，并用变量来保持，供程序设计使用：

- 标定节点均匀分布，间隔为 10.0℃，用变量 w 保存。
- 温度范围的起点为 0.0℃，用变量 w_0 保存，终点为 100.0℃，用变量 w_n 保存。
- 表格内容为 11 个标定节点的 A/D 转换结果，用数组 adc[11]保存。

由这 11 个标定节点连接成 10 段线段，依次为它们进行编号，编号顺序为 0～9，线段的编号用变量 i 保存。

在任何一个线段内，采用点斜式线性插值公式计算：

$$y = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) = y_0 + (y_1 - y_0)(x - x_0)/(x_1 - x_0)$$

式中，x——插值点的转换值。

y——插值点的温度值。

x_0——线段起点的转换值，对于编号为 i 的线段，有：

$$x_0 = \text{adc}[i]$$

x_1——线段终点的转换值，对于编号为 i 的线段，有：

$$x_1 = \text{adc}[i+1]$$

y_0——线段起点温度值，等于温度间隔与线段编号的乘积，即：

$$y_0 = w \times i$$

$y_1 - y_0$——线段的高度(线段两端的温度差)，即：

$$y_1 - y_0 = w$$

$x - x_0$——插值点转换值与线段起点转换值的差，即：

$$x - x_0 = x - \text{adc}[i]$$

$x_1 - x_0$——线段的宽度(线段两端的转换值之差)，即：

$$x_1 - x_0 = \text{adc}[i+1] - \text{adc}[i]$$

将以上变量代入点斜式插值公式，可以得到实际插值计算公式如下：

$$\begin{aligned} y &= y_0 + (y_1 - y_0)(x - x_0)/(x_1 - x_0) \\ &= w \times i + w \times (x - \text{adc}[i])/(\text{adc}[i+1] - \text{adc}[i]) \end{aligned}$$

现在剩下的问题是，如何确定某个转换值 x 所在线段的编号 i。方法也很简单：

- 转换值 x 必须在合理范围之内：低于第一个节点的转换值(adc[0])时一律以0.0℃为计算结果，达到或超过最后一个节点的转换值(adc[10])时一律以 100.0℃为计算结果。转换值 x 只有在 adc[0]～adc[10]之间才进行插值计算。
- 从第一条线段(编号为 0)开始，将转换值 x 和线段终点的转换值(adc[i+1])进行比较，如果超过它，说明不在当前线段范围内，继续比较下一条线段。
- 当转换值 x 小于某个线段终点的转换值时，就可以确定它的线段编号了。

由此，可以得到折线插值算法程序如下：

程序清单 2-1　折线插值算法验证程序

```
#include <stdio.h>
#define N   10                                  //折线由 10 段线段组成(即有 11 个插值节点)
float   w = 10.0;                               //插值节点间隔为 10.0℃(即 w = y1 - y0 = 10.0)
float   w0 = 0.0;                               //起点温度为 0.0℃
float   wn = 100.0;                             //终点温度为 100.0℃
int   adc[N + 1] = {86,376,687,1021,1379,       //记录 11 个标定节点的 A/D 转换结果
            1760,2164,2592,3043,3515,4008};
```

```
float line (int x)                          //折线插值算法,x 为 A/D 转换结果,返回对应温度值
{
    int i;
    if (x<adc[0]) return (w0);                          //A/D 转换结果过低,返回起点温度
    if (x> = adc[N]) return (wn);                       //A/D 转换结果过高,返回终点温度
    for (i = 0;i<N;i ++ )  if (x<adc[i + 1]) break;     //判断 x 所在区间
    return (w * i + w * (x - adc[i])/(adc[i + 1] - adc[i])); //用点斜式线性插值算法计算
                                                        //温度值
}

main ( )                                                //折线插值算法验证程序
{
    int x;
    float y;
    while (1) {
        printf ("\n 输入 A/D 转换结果:");
        scanf (" % d",&x);
        if (x = = 0) break;                             //输入 0 时结束验证
        y = line (x);                                   //执行折线插值算法
        printf ("\n 对应温度为 : % 2.2f℃ ",y);          //显示对应温度(显示两位小数)
        }
}
```

2.2 抛物线插值

2.2.1 算法原理

抛物线插值又称二次插值,它是以一元二次多项式去拟合某一段曲线。精度自然要比线性插值高。如图 2-3 所示,已知一条曲线 $y=f(x)$上的 3 点 $A(x_0,y_0)$、$B(x_1,y_1)$和 $C(x_2,y_2)$;过此 3 点可以作一抛物线(如图 2-3 中虚线所示),即一条二次曲线 $g(x)$,且是唯一的。

设 $g(x)=ax^2+bx+c$,已知 $g(x_i)=f(x_i)$,其中,$i=0,1,2$。

由此可得下列方程组:

$$\begin{cases} ax_0^2+bx_0+c=y_0 \\ ax_1^2+bx_1+c=y_1 \\ ax_2^2+bx_2+c=y_2 \end{cases}$$

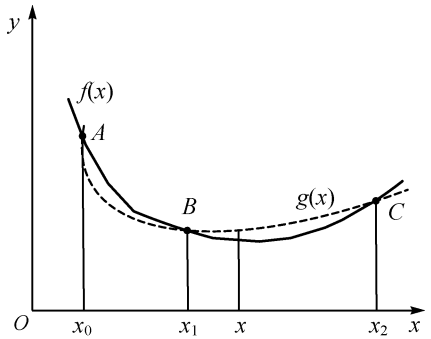

图 2-3　抛物线插值示意图

解此方程组可得 $g(x)$ 的系数 a、b 和 c，但是这样运算很麻烦。实际计算中，可以把 $g(x)$ 写成各种形式的二次多项式，然后运用待定系数法把 $g(x)$ 确定下来。这些方法包括抛物插值的拉格朗日(Lagrange)形式，抛物插值的牛顿(Newton)形式，逐次线性插值法等。在这里，只讨论逐次线性插值法，因为它能充分利用前面线性插值程序，也容易在计算机上实现。

已知 3 点 (x_0,y_0)、(x_1,y_1)、(x_2,y_2) 及插值点的 x 值，求对应的 y 值。

用直线点斜式公式：

第 1 步：过点 (x_0,y_0) 和 (x_1,y_1) 作直线 L_{01}，即：

$$L_{01} = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) \tag{2-1}$$

第 2 步：过点 (x_0,y_0) 和 (x_2,y_2) 作直线 L_{02}，即：

$$L_{02} = y_0 + \frac{y_2 - y_0}{x_2 - x_0}(x - x_0) \tag{2-2}$$

第 3 步：将 (x_1,L_{01}) 和 (x_2,L_{02}) 也理解为“点”，过这两点作“直线”，记之为 L_{012}，即：

$$L_{012} = L_{01} + \frac{L_{02} - L_{01}}{x_2 - x_1}(x - x_1) \tag{2-3}$$

第 4 步：把式(2-1)和式(2-2)代入式(2-3)，得：

$$L_{012} = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + \left[\frac{\dfrac{y_2 - y_0}{x_2 - x_0} - \dfrac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_1}\right](x - x_0)(x - x_1) \tag{2-4}$$

不难看出，L_{012} 是关于 x 的二次函数，并且满足原已知点关系：

$$L_{012}(x_i) = y_i \qquad (i = 0,1,2)$$

因此，L_{012} 就是所要求的二次插值多项式 $g(x)$。

仔细观察式(2-4)后可以看出，$(y_1-y_0)/(x_1-x_0)$ 是一阶差商，而方括号[ ]内的分式是二阶差商。式(2-3)是通过两个线性插值的结果获得 $g(x)$，所以称为逐次线性插

值。显然，式(2－1)、式(2－2)和式(2－3)都是线性插值中的点斜式表达式，因此，可以直接调用线性插值程序。

因为只考虑内插，所以插值点 x 应位于$[x_0, x_2]$的区间内，二次插值算法程序如下：

程序清单 2－2　抛物线插值算法验证程序

```
#include <stdio.h>
float x0,y0;                    //第 1 个插值节点的坐标
float x1,y1;                    //第 2 个插值节点的坐标
float x2,y2;                    //第 3 个插值节点的坐标
float x0t,y0t;                  //第 1 个插值节点坐标的备份
float x1t,y1t;                  //第 2 个插值节点坐标的备份
float L01,L02,L012;             //中间变量

float line (float x)            //点斜式线性插值算法
{
    return (y0 + (y1 - y0) * (x - x0)/(x1 - x0));
}

float qins (float x)            //以线性插值算法为基础的抛物线插值算法
{
    L01 =  line (x);            //过点(x0,y0)和(x1,y1)作直线 L01
    x1t = x1;y1t = y1;          //备份(x1,y1)的坐标数据
    x1 = x2;y1 = y2;            //将(x2,y2)复制到(x1,y1)中,以便重复使用线性插值函数
    L02 =  line (x);            //过点(x0,y0)和(x2,y2)作直线 L02
    x0t = x0;y0t = y0;          //备份(x0,y0)的坐标数据
    x0 = x1t;y0 = L01;          //将(x1,L01)复制到(x0,y0)中
    x1 = x2;y1 = L02;           //将(x2,L02)复制到(x1,y1)中
    L012 =  line (x);           //过"点"(x1,L01)和(x2,L02)作"直线"L012,完成抛物线插值
    x0 = x0t;y0 = y0t;          //恢复(x0,y0)的坐标数据
    x1 = x1t;y1 = y1t;          //恢复(x1,y1)的坐标数据
    return (L012);              //返回抛物线插值计算结果
}

main ( )                        //抛物线插值算法验证程序
{
    float x,y;
    printf ("\n 输入三个插值节点的坐标:");
    printf ("\nx0 = ");
    scanf (" %f",&x0);
```

```
    printf ("\ny0 = ");
    scanf (" % f",&y0);
    printf ("\nx1 = ");
    scanf (" % f",&x1);
    printf ("\ny1 = ");
    scanf (" % f",&y1);
    printf ("\nx2 = ");
    scanf (" % f",&x2);
    printf ("\ny2 = ");
    scanf (" % f",&y2);
    while (1) {
        printf ("\n 输入插值点的 x 坐标值:");
        scanf (" % f",&x);
        if (x = = 0) break;                          //输入 0 时结束验证
        y = qins (x);                                //执行抛物线插值算法
        printf ("\n 对应的 y 坐标值为: % f",y);      //显示对应的 y 值
        }
}
```

2.2.2 应用实例

在 2.1.2 小节用折线插值的方法完成了基于表 2-1 标定结果的温度求解算法,同时也可以用抛物线插值算法来解决这个问题,下面是相关的程序:

程序清单 2-3 用抛物线插值算法求解温度值的验证程序

```
#include <stdio.h>
#define N   10                                   //即共有 11 个标定节点
float   w = 10.0;                                //标定节点间隔为 10.0℃(即 w = y1 - y0 = 10.0)
float   w0 = 0.0;                                //起点温度为 0.0℃
float   wn = 100.0;                              //终点温度为 100.0℃
float x0,y0;                                     //第 1 个插值节点的坐标
float x1,y1;                                     //第 2 个插值节点的坐标
float L01,L02,L012;                              //中间变量
int   adc[N + 1] = {86,376,687,1021,1379,        //记录 11 个标定节点的 A/D 转换结果
     1760,2164,2592,3043,3515,4008};

float line (float x)                             //点斜式线性插值算法
{
    return (y0 + (y1 - y0) * (x - x0)/(x1 - x0));
```