数值计算第1次作业

211240021 田铭扬

§1 第1周上机(230215)实验报告

·1.1 问题

编程找出自己的计算机float类型与double类型的机器精度、下溢值和上溢值。

·1.2 原理与算法思路

·1.2.1“位”形式与浮点数形式的转换

由于计算机存储float类型和double类型的方式比较复杂，为了更加直观，我需要一个程序，按位显示一个float或double类型浮点数在内存中的“位形式”。同时我们需要一个反向程序，使得我能够输入一串二进制位，让它转化为一个浮点数。

实现方法很简单。以float为例，强制让unsigned int*类型的指针pui与float*类型指针的pf指向同一地址。那么正向地，只需用pf读入一个浮点数，然后对pui用二进制输出即可；反之亦然，读入二进制数，储存到pui中，再用pf输出。代码如下（此处为float的代码，double只需相应修改变量类型）：

```
#include <cstdio> //float to bit
using namespace std;
int main() {
  float *pf = 0;
  unsigned int *pui = 0;
  pf=new(float);
  while(1){
    scanf("%f",pf);
    pui = (unsigned int *)pf;
    for(int i=31;i>-1;i--)
      printf("%d",((*pui)>>i)%2);
    printf("\n\n");
  }
  return 0;
}
```

```
#include <cstdio> //bit to float
using namespace std;
int main() {
  float *pf = 0;
  unsigned int *pui = 0;
  pui=new(unsigned int);
  char input_c;
  while(1){
    *pui=0;
    for(int i=31;i>-1;i--){
      scanf("%c",&input_c);
      while(input_c<'0'||input_c>'9')
        scanf("%c",&input_c);
      *pui+=(input_c-'0')<<i;
    }
    pf = (float *)pui;
    printf("%.50f\n\n",*pf);
  }
  return 0;
}
```

·1.2.2 机器精度

由定义知，设相邻的两个机器数分别为a和b，则区间[a,b]中相对误差最大（精度最低）的点为 $\frac{a+b}{2}$ ，相对误差为 $\frac{a-b}{a+b}$ 。而由计算机存储float类型的方式知，[a,b]与[a*2^k,b*2^k](k为整数)相应的精度是一致的①，故只需遍历[1,2]间的所有相邻机器数，计算 $\frac{a-b}{a+b}$ 的最大值即可。（对于double类型，见1.3.1的讨论）代码如下：

```
#include <cstdio> //float
using namespace std;
float a,b;
double max_ep=-1.0;
double epsilon(double a, double b){ return (a-b)/(a+b); }
int main() {
   unsigned int *pui = new(unsigned int);
   float *pf = (float *)pui;
   *pf = 1.0;
```

```
for(int i=0;i<4194304;i++){ //2^22
      a=*pf; *pui+=1; b=*pf;
      max_ep=max(max_ep,epsilon(b,a));
   }
   printf("%.30f",max_ep);
   return 0;
}
```

·1.2.3 上溢值与下溢值

上溢值和下溢值可以使用 2.1 中的程序“半自动”地得出。

对于 float 类型的上溢值：通过查询资料知，其“位形式”为 01111111 01111111 11111111 11111111（指数位第 9 位为’0’，因为若为’1’，则值为 nan）, 对应的值为 340,282,346,638,528,859,811,704,183,484,516,925,440（十进制）或 ffff ff00 0000 0000 0000 0000 0000 0000（十六进制），记为 X。若要验证这一结论的正确性，只需取 X 的上一个机器数 Y（事实上，Y 为 01111111 01111111 11111111 11111110），然后写一个 float 类型的“A+B”程序，计算 X+(X-Y)或者 X+(X-Y)*2。若前者为 inf，或前者等于 X 而后者为 inf，则 X 就是上溢值。

对于下溢值：查资料得知 其“位形式”为 00000000 00000000 00000000 00000001（即指数部分为 000000000，查资料知此时存储的为非规格化的浮点数），即二进制的 2^{-149}——事实上，149=126+23——约等于十进制的 $1.40*10^{-45}$。验证方法与上溢值类似，记 $X=2^{-149}$，Y 为其前一个机器数（事实上，Y 为 00000000 00000000 00000000 00000010），计算 X-(Y-X).若结果为 0，则 X 就是下溢值。

对于 double 类型，是同理的。

·1.3 实验过程与结果

·1.3.1 机器精度

运行 2.2 中的代码即可得到 float 类型的机器精度，得到二进制的 2^{-24}，约等于十进制的 $5.96*10^{-8}$。

由于运算的速度限制，double 类型无法使用 float 类型的代码进行改动。但是注意到，沿用 2.2 中的记号[a,b], 在 a=1 时相对误差最大，为 2^{-24}；相对误差随 a 变大而变小，至 b=2 时误差为 2^{-25} 最小；而 a=2 时，误差又变回 2^{-24}（这恰好验证了 2.2 中的①）。因此可以断言，double 类型的机器精度——即最大相对误差——也在 a=1 时取到，为二进制下的 2^{-53}，约等于的 $1.11*10^{-16}$。

·1.3.2 上溢值与下溢值

实验过程如 2.3 所述，结果表明查询到的资料正确，详见“结论”部分。

·1.4 结论

对于我所使用的计算机，float 类型：

机器精度为 2^{-24}，约等于 $5.96*10^{-8}$；

上溢值为 ffff ff00 0000 0000 0000 0000 0000 0000，约等于 $3.40*10^{38}$；

下溢值为 2^{-149}，约等于 $1.40*10^{-45}$。

double 类型：

机器精度为 2^{-53}，约等于 $1.11*10^{-16}$；

上溢值为 ffff ffff ffff $f800*16^{-240}$，约等于 $1.80*10^{308}$；

下溢值为 2^{-1074}，约等于 $4.94*10^{-324}$。

§2 第3周上机(230301)实验报告

·2.1 问题1

·2.1.1 问题

编写程序计算y=x-sin(x)，使得有效位的丢失最多1位。

·2.1.2 原理分析与算法实现

熟知x≈0时sin(x)≈x，相减相消会导致有效位丢失。根据提示，当 $\frac{(x-\sin(x))}{x} \geq \frac{1}{2}$ 时，有效位不会丢失超过1位，故这时可以直接调用cmath库中的函数计算x-sin(x)并输出。而当 $\frac{(x-\sin(x))}{x} < \frac{1}{2}$ 时，可以考虑这样的思路：

将sin(x)进行Taylor展开，得 $x-\sin(x)=\frac{1}{3!}x^3-\frac{1}{5!}x^5+\ldots+\frac{(-1)^{n-1}}{(2n+1)!}x^{2n+1}+\ldots$ ，如果输入值x（假设为十进制小数）的最后一位有效位为a*10^{-b}，那么上式只需要计算到使得 $\left|\frac{x^{2n+1}}{(2n+1)!}\right| < 5\times 10^{-b-1}$ 的项，即可使有效位的丢失最多1位。子程序如下：

```
double _calc(double _x,int _b){//_b为上面假设的b
   double _result=_x-sin(_x);
   if(_result/_x>0.5) return _result;//此时直接计算不会导致丢失精度
   double _tmp=(_x*_x*_x/6), _epsilon=5*pow(10,-_b-1);//_tmp用于计算x的幂
   int _cnt=3;//用于计算分母上的阶乘
   _result=_tmp;
   while(abs(_tmp)>_epsilon){
     _tmp=(_tmp*_x*_x*(-1)/(_cnt+1)/(_cnt+2));
     _cnt+=2; _result+=_tmp;
   }
   return _result;
}
```

·2.2 问题2

·2.2.1 问题

计算 $y_n=\int_0^1 x^n e^x dx$ ，由分布积分得 $y_{n+1}=e-(n+1)y_n$ ，验证数值不稳定。

·2.2.2 原理分析与算法实现

易知 $y_0=e-1$ ，递推的实现是容易的，而积分可以通过黎曼积分的定义进行近似计算（代码如下）。然后分别输出两种方法得到的前n项（n充分大）结果，进行比较即可。

```
#include <cstdio>
#include <cmath>
using namespace std;
int n; double len,start,end;//后三项分别是步长、积分下限和上限
double _f(double _x){//计算函数值
   double _r=exp(_x);
   for(int i=0;i<n;i++) _r*=_x;
   return _r;
}
int main(){
   scanf("%d%lf%lf%lf",&len,&start,&end);//本题积分下限和上限分别为0和1
   double x=start+len/2,S=0;
   while(x<end){ S+=(_f(x)*len); x+=len; }
   printf("%.12f",S);//讲过尝试，输出12位小数对本题的比较就足够了
   return 0;
}
```

·2.2.3 结果与分析

使用递推和直接计算积分的方式分别计算了 y_0 到 y_{20} 的数据（计算积分时，取步长为 0.00001）。第 0~15 项两者数值相差不大，而将 y_{16} 到 y_{20} 的结果如下：

	y_{16}	y_{17}	y_{18}	y_{19}	y_{20}
积分	0.151460885341	0.143446774096	0.136239890758	0.129723899654	0.123803830520
递推	0.150348161837	0.162363077223	-0.204253561558	6.599099498066	-129.263708132859

其实即使只看递推的结果，不与直接积分的比较，也能知道对于此问题递推算法并不稳定——因为显然所求积分>0，而在第 18 项中，递推算法得到了负值！

究其原因，一方面，递推公式中 y_n 的系数 n+1 会使得误差 ε 被放大至多 n+1 倍，而当项数 n 较大时，相对误差界会开始快速地变大。另一方面，递推算法得到的结果从 y_{17} 开始不稳定，是因为 $17*y_{16}\approx 2.55$，与 e 的数值接近，开始出现了较明显的相减相消现象。

·2.3 问题 3

·2.2.1 问题

考虑递推 $x_{n+1}=\frac{13}{3}x_n-\frac{4}{3}x_{n-1}$ ，考虑初值(x_0,x_1)=(1,1/3)或(1,4)，算法是否稳定？

·2.3.2 原理分析与算法实现

所求为二阶齐次线性递推，可以通过特征方程式的方式求出其通项为 $x_n=A4^n+B\left(\frac{1}{3}\right)^n$ ，并根据初值计算出系数 $A=\frac{3x_1-x_0}{11},B=\frac{10x_0-3x_1}{11}$ 。分别用递推和通项两种方式计算出前 n 项（n 充分大）结果，进行比较即可。由于两种算法都很简单，在此不列出代码。

·2.3.3 结果与分析

对于第一组初值，在第 2 项时，两种算法得到的值就已经有约 10%的差距；而在 n>=15 时，使用递推竟然会得到递增的值（如下表）！这说明在此初值下，递推算法并不稳定。

	x_2	x_3	x_4	x_5	x_6
通项	0.090909090909	0.030303030303	0.010101010101	0.003367003367	0.000374111485
递推	0.111111111111	0.037037037037	0.012345679012	0.004115226337	0.001371742112

	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}
通项	0.000000057020	0.000000019007	0.000000006336	0.000000002112	0.000000000704
递推	0.000000073411	0.000000038108	0.000000067254	0.000000953022	0.000003808932

究其原因，在第一组初值下，通项公式的系数 A=0，这意味着恰好有 $x_{n+1}=x_n/3$。而注意到，递推公式中 x_n 系数 13/3 大约是 x_{n-1} 的系数 4/3 的三倍，这导致计算递推的时候出现了明显的相减相消现象，所以两种算法的结果在一开始就出现差距，递推算法不稳定。

而在第二组初值下并不存在这样的问题，两种算法的值在前 30 项没有出现较大差距（数据略），此时递推算法稳定。