谨防浮点运算

请参阅实验室目录中的更多文档,网址为http://www.isima.fr/~hill/并测试下面的代码。

介绍

即使非常有经验的程序员在编写依赖浮点算法的代码时也会陷入几个陷阱。本文解释了使用浮点数（即float和double数据类型）时需要注意的事项。

在计算机科学培训的开始阶段,很快就会看到对用有限的位集合对实数进行编码的 IEEE 754 标准的研究,但它常常被计算机科学家遗忘。

当他们开始用他们最喜欢的计算机和语言产生错误的结果时,如果他们记了笔记,他们会在过去的讲座中再次挖掘……为了快速记忆“回忆”,请参阅以下 URL 如何将浮点数编码为二进制:http: //www.binaryconvert.com/ . 就像 1/3 无法用十进制精确表示一样,0.1 也无法用二进制精确表示（二进制数以以下无限重复的主题结束:1100 1100 1100,最终必须四舍五入才能以 32 位或 64 位存储数字。第一张图片给出了 32 位浮点数的表示,下一张给出了 64 位双精度数的表示。

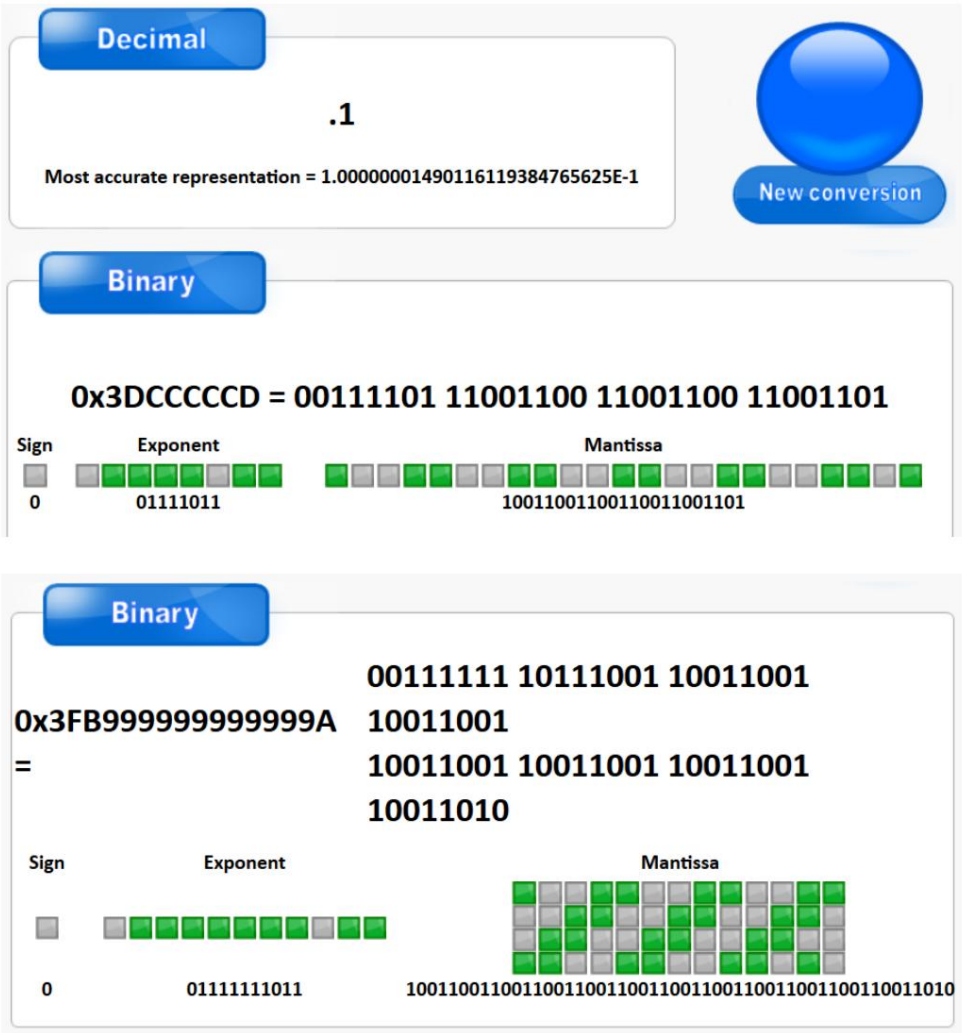

图 1:二进制中 0.1 的表示形式（简单和双精度）

它对我的计算有何影响?

让我们看一下上节课中你可能和我一起学习过的代码。即使你现在知道它返回“只有 1 是正确的”,你也应该尝试深入了解它的工作原理。

```
int主要()
{
        浮点数a = 0.7;
        浮点数b = 0.5;

        如果(a < 0.7)
        {
            if (b < 0.5) printf(  2个条件正确  );
            else printf(  只有1是正确的  );
        }
        else printf(  0个条件正确  );
}
```

代码用 C 语言编写,保持简单,但大多数语言(通常依赖于 C)都会表现出相同的行为。这
假设您使用 IEEE 754 浮点标准,则行为是 100% 可预测的。对于每个数值计算来说都应该如此,但您会发现这条细则已被许多人忽视(包括在高性能计算领域)。要查看存储在内存中的精确浮点值,需要使用诸如printf(“%0.18f”, variable)之类的格式。这表明下一个代码“%0.18f”精度不够。

在比较过程中,浮点数会提升为双精度数,并且由于浮点数不如双精度数精确,因此浮点数 0.7 与双精度数 0.7 不同。

在这种情况下,浮点型 0.7 在提升后就不如双精度型 0.7 了。

由于 0.5 是 2 的幂,它总是被精确表示,所以测试按预期进行:0.5 < 0.5 为假。

如果您想要简单的预期行为,请将 float 更改为 double 或将 .7 和 .5 更改为 .7f 和 .5f,您将获得预期的行为。

测试以下代码,以了解 float 和 double 的表示形式有何不同。

```
int 主要()
{
        float a = 0.7, b = 0.5; // 这些是浮点数
        双精度c=(双精度)a;
        双d = 0.7;

        printf(“%0.10f”,a);
        printf(“%0.15f”,c);

        printf(“%0.15f”, d);                    // 显示近似值
        printf(“%0.18f”, c);
        printf(“%0.18f”, d);                    // 显示所有详细信息

        如果(a < .7) {                                                  // 这个测试是 DOUBLE 类型的

            if (b < .5) printf(  2 是正确的  ); else printf(  1 是正确的  );          // 此处的测试也是 DOUBLE

        }
        else printf(  0 是正确的  );
}
```

不要测试平等

经验丰富的开发人员（几乎）从来不想写如下的代码：

```
双x;
双 y;
 ...
如果 (x == y) {...}
```

大多数浮点运算至少会造成微小的精度损失，因此即使两个数字在所有实际用途上相等，它们也可能不完全相等，直到最后一位，因此相等性测试很可能会失败。例如，以下代码片段打印-1.778636e-015。虽然从理论上讲，平方应该会撤消平方根，但往返运算略有不准确。

```
双精度 x = 10; 双精度 y =
sqrt(x);

和 *=  和;

如果 (x == y)
        cout <<  平方根是精确的\n  ;
别的
        cout<<xy<< “\n”;
```

大多数情况下，上述相等性测试应该写成如下所示的形式：

```
双重公差 =                                ...
如果 (fabs(x - y) < 容差) {...}
```

这里的公差（通常记为eps或epsilon）是某个阈值，它定义了什么才是“足够接近”的相等性。这就引出了一个问题：接近到什么程度才算足够接近。即使您可以快速想到，对于单精度，可以将epsilon设置为1e-6 ，对于双精度，可以将 epsilon 设置为1e-12 ，但在这个简短的演示中无法准确回答这个问题。您可以使用float.h中的现成常量，例如 FLT_EPSILON 和 DBL_EPSILON。但是，您必须对您的特定问题有所了解，才能知道在您的上下文中，接近到什么程度才算足够接近。

相对误差

对于 1 左右的数字，0.00001 的误差是合适的；对于 0.00001 左右的数字，误差太大；对于 10,000 左右的数字，误差太小。比较两个数字的一种更通用的方法（无论它们的范围如何）是检查相对误差。相对误差是通过将误差与预期结果进行比较来测量的。

其中一种计算方法如下：

```
相对误差 = fabs((结果 - 预期结果) / 预期结果);
```

如果结果为 99.5，且预期结果为 100，则相对误差为 0.005。

比起乘法和除法，更担心加法和减法

乘法和除法的相对误差总是很小。而加法和减法则可能导致完全的精度损失。

这一事实导致了许多 HPC 应用中遇到的一个主要问题。简化阶段非常微妙,在经过密集计算（您可能已经以最高的数值质量实现了密集计算)之后,最终的简化阶段可能会导致结果不准确、错误或不可重现。您已将一些计算映射到多个核心,每个核心都会给出结果,但随后您必须计算一个指标的平均值,该指标会将所有值“简化”为一个指标,而这通常会使用加法。如果您有大量加法（例如在 HPC 应用中）,这有时会极大地影响您的最终结果。

加法本身可能是一个问题（请参见下面代码中的差异）,但更重要的问题是减法;只有当两个相加的数字符号相反时,加法才会成为问题,因此您可以将其视为减法。尽管如此,代码可能用“+”编写,但您也会有减法,人们常说,在并行代码的缩减阶段,我们在“加法”方面遇到了问题。

```
浮点数                          = 0.1f;
浮点数和 = 0;

对于 (int i = 0; i < 10; ++i) 总和 += f;

浮点乘积 = f * 10;

printf(    sum = %1.15f, mul = %1.15f, mul2 = %1.15f\n    , sum, product, f * 10);
```

当两个被减的数字几乎相等时,减法就会出现问题。数字越接近相等,精度损失的可能性就越大。具体来说,如果两个数字的位数一致,则减法可能会损失 n 位精度。这种情况在极端情况下可能最容易看出:如果两个数字在理论上不相等,但在机器表示中相等,则它们的差将计算为零,精度损失 100%。

下面是一个经常出现这种精度损失的例子。函数f在点x的导数定义为h趋向于零时(f(x+h) - f(x))/h的极限。因此,计算函数导数的自然方法是对某个较小的h 求(f(x+h) - f(x))/h 。理论上， h越小,这个分数对导数的近似值就越好。在实践中,准确度在一段时间内会提高,但过了某个点， h值越小,对导数的近似值就越差。随着h变小,近似误差变小,但数值误差会增加。这是因为减法 f(x+h) -

f(x)就会出现问题。如果你取h足够小（毕竟,理论上越小越好）,那么f(x+h)就会等于f(x) 的机器精度。这意味着,无论函数是什么,只要你取h足够小,所有导数都会被计算为零。下面是计算sin(x)导数的示例

x = 1。

```
cout << std::setprecision(15);                                          // C++ 设置标准输出精度的方法

对于（int i = 1; i < 20; ++i）
{
          双h = pow（10.0，-i）；
          cout << (sin(1.0 + h) - sin(1.0))/h <<    \n    ;
}

cout<<“真实结果：”                                  << cos(1.0) <<    \n    ;
```

这是上述 C++ 代码的输出。为了使输出更容易理解,第一个错误数字之后的数字已被替换为句点。

```
0.4.........
0.53........
0.53........
0.5402.......
0.5402.......
0.540301.....
```

```
0.5403022.....
0.540302302...
0.54030235....
0.5403022.....
0.540301......
0.54034.......
0.53..........
0.544.........
0.55..........
0
0
0
0

真实结果:0.54030230586814
```

随着h 的减小,准确度会提高,直到h = 10-8。超过该点,准确度会由于减法中的精度损失而下降。当h = 10-16或更小时,输出恰好为零,因为sin(1.0+h)等于sin(1.0) （以机器精度计算）。（事实上， 1+h等于1 （以机器精度计算）。下面将详细介绍。）

上述结果是用 Visual C++ 计算的。在 Linux 上使用 4.2.3 以上的 gcc 进行编译时,结果相同,除了最后四个数字。VC++ 生成零,而 gcc 生成负数：-0.017...、-0.17...、-1.7... 和 17....)

当您的问题需要减法并且会导致精度损失时,您会怎么做?有时精度损失不是问题;双精度开始时有很多剩余的精度。当精度很重要时,通常可以使用一些技巧来改变问题,以便它不需要减法,或者不需要与您开始时相同的减法。在这种情况下,必须考虑像 Kahan （IEEE 754 标准的主要设计者)提出的补偿求和。

请参阅 CodeProject 文章避免溢出、下溢和精度损失了解使用代数技巧将二次公式改为更适合保持精度的形式的示例。另请参阅比较三种计算标准差的方法作为代数等效方法的表现如何有很大差异的一个例子。

浮点数有有限范围

大家都知道浮点数的范围是有限的,但这种限制可能会以意想不到的方式表现出来。例如,您可能会对以下代码行的输出感到惊讶。

```
浮点数 f = 16777216; cout << f <<
                                - -
                                    << f+1 <<   \n   ;
```

此代码打印两次值16777216。发生了什么?根据 IEEE 浮点运算规范,浮点类型为 32 位宽。其中 24 位用于有效数字（以前称为尾数） ,其余用于指数。数字16777216为224 ,因此浮点数

变量f没有剩余精度来表示f+1。如果f是double类型,则253也会出现类似现象,因为 64 位double将 53 位用作有效数字。以下代码打印0而不是1。

```
x = 9007199254740992; cout << ((x+1) -                                  // 2^53
x) <<   \n   ;
```

在将小数字添加到中等大小的数字时,我们也会用尽精度。例如,以下代码会打印“抱歉！” ,因为DBL_EPSILON （在float.h 中定义)是最小正数 e,因此在使用double类型时 1 + e != 1。

```
x = 1.0;
y = x + 0.5 *                    DBL_EPSILON;

if (x == y) cout <<    抱歉!\n   ;
```

类似地,常数FLT_EPSILON是最小正数 e,使得在使用时 1 + e 不为 1

浮点类型。这应该有助于您理解此类常量的定义。

使用对数避免溢出和下溢

上一节中描述的浮点数的局限性源于有效数字的位数有限。溢出和下溢也源于指数的位数有限。有些数字太大或太小而无法存储在浮点数中。

许多问题似乎需要计算一个中等大小的数字作为两个巨大数字的比率。
即使中间结果不能表示为浮点数,但最终结果也可以表示为浮点数。
在这种情况下,对数提供了一种解决方法。如果要计算大数M和N 的M/N ,请计算log(M) - log(N)并将exp()应用于结果。例如,概率通常涉及阶乘的比率,阶乘很快就会变得非常大。对于N > 170， N!大于DBL_MAX,这是双精度（无扩展精度)可以表示的最大数字。但可以按如下方式计算诸如200!/(190! 10!)之类的表达式而不会溢出：

```
x = exp(对数阶乘(200) - 对数阶乘(190) - 对数阶乘
                (10) );
```

一个简单但低效的logFactorial函数可以写如下：

```
双对数阶乘（int n）
{
        双精度总和 = 0.0;

        对于（int i = 2; i <= n; ++i）
        {
                总和+ = log（（double)i）;
        }

        返回总数;
}
```

更好的方法是使用对数伽马函数（如果有）。请参阅如何计算二项式概率了解更多信息。

数字运算并不总是返回“数字”

由于浮点数有其局限性,有时浮点运算会返回“无穷大”,表示“结果大于我能处理的范围”。例如,以下代码在 Windows 上打印1.#INF ,在 Linux 上打印inf 。

```
x = DBL_MAX;
cout<<2*x<<“\n”;
```

有时,返回有意义结果的障碍与逻辑有关,而不是有限精度。浮点数据类型表示实数（而不是复数）,并且没有平方为-1 的实数。这意味着,如果代码请求sqrt(-2),即使在无限

精度。在这种情况下,浮点运算返回NaN （代表“非数字”）。这些是代表错误代码而不是数字的浮点值。NaN值在 Windows 上显示为1.#IND ,在 Linux 上显示为NaN 。

一旦操作链遇到NaN,从此以后所有内容都将为NaN 。例如,假设您有一些代码量导致以下内容：

```
if (x - x == 0) // 执行某些操作
```

什么可能导致if语句后面的代码无法执行?如果 x 是NaN,那么x - x也是NaN
并且NaN不等于任何东西。事实上， NaN甚至不等于它们自己。这意味着表达式x == x可用于测试x是否为（可能是无限的)数字。有关无穷大和NaN 的更多信息,请参阅C++ 中的 IEEE 浮点异常。

详细信息

本文以“Stack overflow”中给出的示例以及我硕士和 ISIMA 模拟和高性能计算课程中给出的参考文章和附加评论为基础。请参阅： http: // www.codeproject.com/Articles/29637/Five-Tips-for-Floating-Point-Programming

---

如果你需要更深入地了解浮点运算,可以阅读《每个计算机科学家都应该知道的浮点运算》一文解释得非常详细。这也许是每个计算机科学家最希望知道的,但不幸的是,对于科学计算来说,很少有人能吸收其中介绍的所有内容。

无序执行（或动态执行）

顶尖科学家最近注意到了最近处理器中引入的一项新技术的影响。
微处理器内部使用非顺序指令（不遵守程序员在源代码中指定的顺序)进行优化可能会损害数值再现性以及最终结果。如果此类优化对许多应用（文字处理、图像处理和识别)无害,则可能会影响数值模拟。

确实:浮点运算不在 R 中运行,而是在 Q（分数集)中运行,因此加法和乘法不具有结合性。因此， a + (b + c) 不等于 (a + b) + c !!!

示例可以使用0.18f格式进行显示：

$$(10^{-3}+1)-1 \sim 0$$
$$10^{-3}+(1-1)=10^{-3}$$

```
>>> (pow(10,-3)+1)-1
0.0009999999999998899
>>> pow(10,-3)+(1-1)
0.001
>>>
```

图 2:Python 和 Matlab 示例

另一个例子 :表达式(260 - 2 60) + 1 等于 0 + 1,最终等于 1。在这种情况下 ,结果是正确的,因为减法按预期完成,但如果运算顺序改变为(260 + 1) - 2 60,则结果将为假并给出 0。使用双精度表示,只有 53 位

可用于尾数,表达式260 + 1 将四舍五入为260 。因此,加一后,我们仍然得到260 ,减去 2
60,将得到 0,这是一个错误的结果。除了处理
有溢出。

对于对高性能计算感兴趣的学生 - 发现补偿金额

对于并行计算,在减少阶段,当操作以不同的顺序进行时,我们可能会得到不同的数值结果。事实上,如上所述,加法（和减法)比乘法和除法更容易出现问题。为了弥补这一点,我们需要 Kahan 求和算法或其他类似的算法……

1) 看看 William Kahan 教授（伯克利)是谁,以及他提出了什么来弥补他所提出的标准可能出现的问题。

2) 生成具有不同范围的随机浮点数的巨大向量的示例,并实现 Kahan 求和的示例并检查与常规求和的差异（结果和计算时间）。

3)查看补偿金额的其他变体。

4)保留实验室和观察记录（无需发送报告）