USTC

Chapter 24
Numerics

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

Overview

- 本章概要说明数值计算的基本工具和技术
  - 对某些人来说，数值计算就是一切；对更多人来说，它只在某些时候是必要的
- Contents
  - 精度、溢出、大小与错误
  - 矩阵
  - 随机数
  - 复数
- 注意：
  - 探索如何用代码表达应用领域的基本符号
    - 数值计算，主要是线性代数
  - 仅仅显示基本工具，而使用它们的计算技术(工程计算)需要额外学习
    - 属于“领域知识”，超出了我们的范围

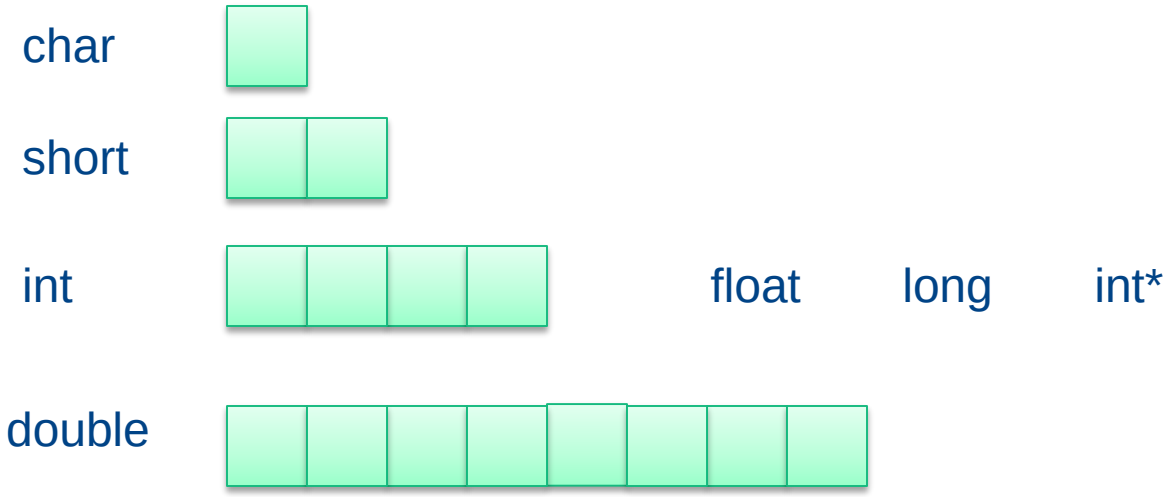
精度等

- 当我们使用内置类型和普通计算技术时，数值占用固定大小的内存
  - 浮点数类型是数学上实数的近似
    - float x = 1.0/333;
    - float sum = 0; for (int i=0; i<333; ++i) sum+=x;
    - cout << sum << "\n";
    - 0.999999
  - 整数类型只能表达相对小的整数
    - short y = 40000;
    - int i = 1000000;
    - cout << y << " " << i*i << "\n";
    - -25536 -727379968

随时检查结果是否合理

- 实际上，没有足够的位数能够精确地表示每个数字，我们需要一种方式以进行有效地计算

大小



- C++ 内建类型的准确大小依赖于硬件平台和编译器
  - 上述大小是在Windows平台 (Microsoft编译器)、Linux平台 (GCC)
  - **sizeof(x)** 提供了 **x** 的大小(字节)—— 对象或类型
  - 根据定义，**sizeof(char)==1**
- 除非你有很好的理由，否则最好只使用 **char**、**int** 和 **double**

大小、溢出、截断

```
// when you calculate, you must be aware of possible overflow and truncation
// Beware: C++ will not catch such problems for you

void f(char c, short s, int i, long lg, float fps, double fpd)
{
    c = i;          // yes: chars really are very small integers
    s = i;
    i = i+1;        // what if i was the largest int?
    lg = i*i;       // beware: a long may not be any larger than an int
                    // and anyway, i*i is an int – possibly it already overflowed
    fps = fpd;
    i = fpd;        // truncates: e.g. 5.7 -> 5
    fps = i;        // you can lose precision (for very large int values)

    char ch = 0;
    for (int i = 0; i<500; ++i) { cout << int(ch) << "\t"; ch++; } // try this
}
```

如果有疑问，就应该检查

- 最简单的方法是测试
  - 赋值然后比较

```
void f(int i)
{
    char c = i;          // may throw away information you don't want to loose
    if (c != i) {
            // oops! We lost information, figure out what to do
    }
    // …
}
```

数学函数错误

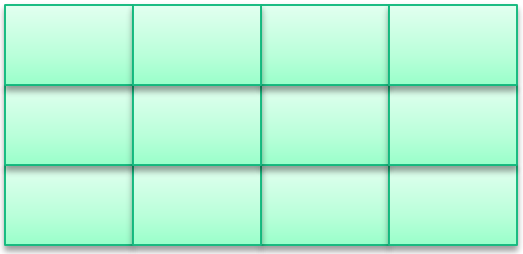
- 如果一个标准数学函数无法工作，它会设置 <cerrno> 中的 errno
  - 例如：

```
void f(double negative, double very_large)
    // primitive (1974 vintage, pre-C++) but ISO standard error handling
{
    errno = 0;              // no current errors
    sqrt(negative);                 // not a good idea
    if (errno)   { /* ... */ }      // errno!=0 means 'something went wrong"
    if (errno == EDOM)// domain error
        cerr << "sqrt() not defined for negative argument";
    pow(very_large,2);  // not a good idea
    if (errno==ERANGE)          // range error
        cerr << "pow(" << very_large << ",2) too large for a double";
}
```

矩阵

- 标准的 vector 和内建的数组都是一维的
- 如果我们需要一个二维的呢?
  - 如：矩阵
  - N 维的呢?

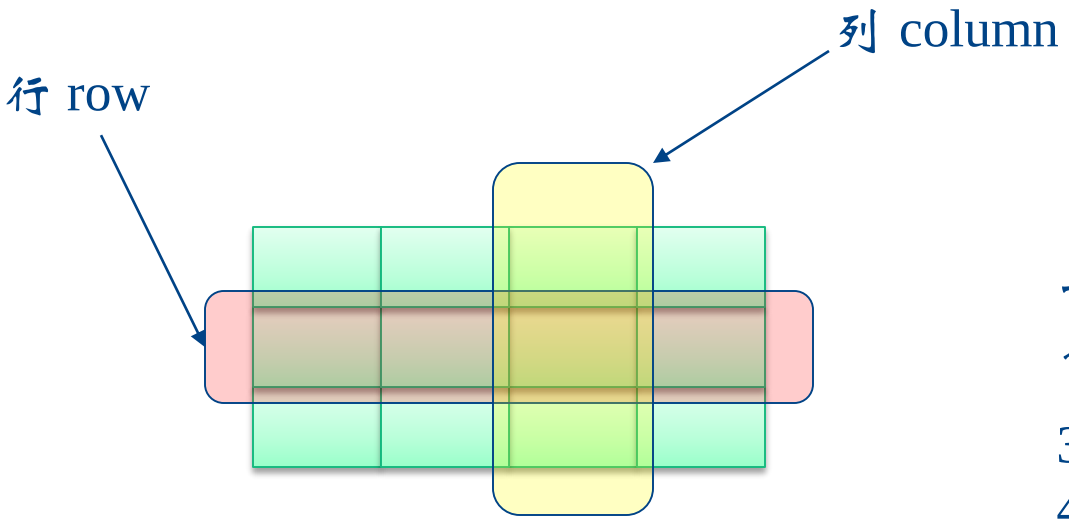
一个向量 (e.g. **Matrix<int> v(4)**)，
也称之为一维矩阵或 1*N 的矩阵



一个 3*4 的矩阵 (e.g. **Matrix<int> m(3,4)**)，
也称之为二维矩阵

矩阵

- **Matrix<int> m(3,4);**



一个 3*4 的
也称之为二维矩阵

3 行
4 列

C 风格的多维数组

- 一种内建工具

```
int ai[4];                  // 1 dimensional array
double ad[3][4];            // 2 dimensional array
char ac[3][4][5];           // 3 dimensional array
ai[1] =7;
ad[2][3] = 7.2;
ac[2][3][4] = 'c';
```

  - 基本上看做：数组的数组

C 风格的多维数组

- 问题
  - C 风格的多维数组是数组的数组
  - 固定大小 (i.e. 编译时就固定下来)
    - 如果想在运行时再确定大小，必须使用自由存储
  - 不能干净地传递参数
    - 只能转换为指向其首元素的指针
  - 没有越界检查
    - 通常，数组不知道自己的大小
  - 没有数组的整体运算
    - 甚至没有赋值 (拷贝)
- 错误的主要来源
  - 对大多数人，这些是很严重的头痛问题
- 只要你想使用它，就看看上面的内容吧!

C 风格的多维数组

❖ 你不能干净地传递多维数组参数

```
void f1(int a[3][5]);                    // useful for [3][5] matrices only
```

- Can't read vector with size from input and then call f1
  - (unless the size happens to be 3*5)
- Can't write a recursive/adaptive function

```
void f2(int [ ][5], int dim1);           // 1st dimension can be a variable

void f3(int[ ][ ], int dim1, int dim2);  // error (and wouldn't work anyway)

void f4(int* m, int dim1, int dim2)      // odd, but works
{
    for (int i=0; i<dim1; ++i)
        for (int j = 0; j<dim2; ++j) m[i*dim2+j] = 0;
}
```

Matrix 库

// on the course site Matrix.h

```
#include "Matrix.h"
void f(int n1, int n2, int n3)
{
    Matrix<double> ad1(n1);          // default: one dimension
    Matrix<int,1> ai1(n1);
    Matrix<double,2>  ad2(n1,n2);    // 2 dimensional
    Matrix<double,3> ad3(n1,n2,n3); // 3 dimensional
    ad1(7) = 0;                      // subscript using ( ) – Fortran style
    ad1[7] = 8;                      // [ ] also works – C style
    ad2(3,4) = 7.5;  // true multidimensional subscripting
    ad3(3,4,5) = 9.2;
}
```

多维访问必须是( )，[]只能用于一维访问

Matrix 库

- “向你的数学/工程教科书查询什么是矩阵”
  - 或者向量、矩阵、张量
- 编译时和运行时检查
- 任意维的矩阵
  - 1、2、3 是实际能工作的 (必要时你能够添加更多维)
- Matrices 是正常的变量/对象
  - 你可以传递它们
- 常用的矩阵操作
  - 下标：( )
  - 单下标：[ ]
  - 赋值：=
  - 标量操作 (+=, -=, *=, %=, etc.)
  - 复合的向量操作 (e.g., **res[i] = a[i]*c+b[2]**)
  - 点积操作 (res = sum of **a[i]*b[i]**)
- 性能等价于手工写的底层代码
- 如果需要，你自己可以扩展它 (库实现没有使用“魔法”)

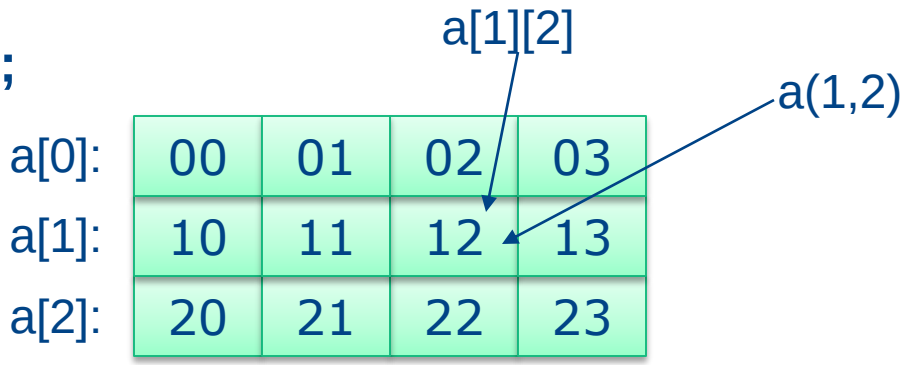
Matrix 库

```
// compile-time and run-time error checking
void f(int n1, int n2, int n3)
{
    Matrix<double> ad1(5); // default: one dimension
    Matrix<int> ai(5);
    Matrix<double> ad11(7);
    Matrix<double,2>  ad2(n1);        // error: length of 2nd dimension missing
    Matrix<double,3> ad3(n1,n2,n3);
    Matrix<double,3> ad33(n1,n2,n3);
    ad1(7) = 0;       // Matrix_error exception; 7 is out of range — run-time
    ad1 = ai;         // error: different element types
    ad1 = ad11;       // Matrix_error exception; different dimensions — run-time
    ad2(3) = 7.5;     // error: wrong number of subscripts
    ad3 = ad33;       // ok: same element type, same dimensions, same lengths
}
```

Matrix 库

- 我们将矩阵看做为 (row, column)：

Matrix<int> a(3,4);



- 该矩阵元素在内存中的布局是“行主次序”：

00	01	02	03	10	11	12	13	20	21	22	23

Matrix 库

```
void init(Matrix<int,2>& a)
    // initialize each element to a characteristic value
{
    for (int i=0; i<a.dim1(); ++i)
        for (int j = 0; j<a.dim2(); ++j)
            a(i,j) = 10*i+j;
}
void print(const Matrix<int,2>& a)
    // print the elements of a row by row
{
    for (int i=0; i<a.dim1(); ++i) {
        for (int j = 0; j<a.dim2(); ++j)
            cout << a(i,j) << '\t';
        cout << '\n';
    }
}
```

二维和三维矩阵

```
// 2D space (e.g. a game board):
enum Piece { none, pawn, knight, queen, king, bishop, rook };
Matrix<Piece,2> board(8,8);              // a chessboard
Piece init_pos[] = { rook, knight, bishop, queen, king, bishop, knight, rook };

// 3D space (e.g. a physics simulation using a Cartesian grid):
int grid_nx;          // grid resolution; set at startup
int grid_ny;
int grid_nz;
Matrix<double,3> cube(grid_nx, grid_ny, grid_nz);
```

1D Matrix

```
Matrix<int> a(10);          // means Matrix<int,1> a(10);

a.size();                   // number of elements
a.dim1();                   // number of elements
int* p = a.data();          // extract data as a pointer to a C-style array
a(i);                       // i^th element (Fortran style), but range checked
a[i];                       // i^th element (C-style), but range checked
Matrix<int> a2 = a;         // copy initialization
a = a2;                     // copy asssignment
a *= 7;                     // scaling a(i)*=7 for each i (also +=, -=, /=, etc.)
a.apply(f);                 // a(i)=f(a(i)) for each element a(i)
a.apply(f,7);               // a(i)=f(a(i),7) for each element a(i)
b =apply(f,a);              // make a new Matrix with b(i)==f(a(i))
b =apply(f,a,7);            // make a new Matrix with b(i)==f(a(i),7)

Matrix<int> a3  = scale_and_add(a,8,a2);      // fused multiply and add
int r = dot_product(a3,a);                    // dot product
```

2D Matrix (very like 1D)

```
Matrix<int,2> a(10,20);

a.size();                  // number of elements
a.dim1();                  // number of elements in a row
a.dim2();                  // number of elements in a column
int* p = a.data();         // extract data as a pointer to a C-style array
a(i,j);                    // (i,j)th element (Fortran style), but range checked
a[i];                      // ith row (C-style), but range checked
a[i][j];                   // (i,j)th element C-style
Matrix<int> a2 = a;        // copy initialization
a = a2;                    // copy asssignment
a *= 7;                    // scaling (and +=, -=, /=, etc.)
a.apply(f);                // a(i,j)=f(a(i,j)) for each element a(i,j)
a.apply(f,7);              // a(i,j)=f(a(i,j),7) for each element a(i,j)
b=apply(f,a);              // make a new Matrix with b(i,j)==f(a(i,j))
b=apply(f,a,7);            // make a new Matrix with b(i,j)==f(a(i,j),7)
a.swap_rows(7,9);          // swap rows a[7] ⇔ a[9]
```

3D Matrix (very like 1D and 2D)

```
Matrix<int,3> a(10,20,30);

a.size();                    // number of elements
a.dim1();          // number of elements in dimension 1
a.dim2();          // number of elements in dimension 2
a.dim3();          // number of elements in dimension 3
int* p = a.data();           // extract data as a pointer to a C-style Matrix
a(i,j,k);                    // (i,j,k)th element (Fortran style), but range checked
a[i];                        // ith row (C-style), but range checked
a[i][j][k];                  // (i,j,k)th element C-style
Matrix<int> a2 = a;          // copy initialization
a = a2;                      // copy asssignment
a *= 7;                      // scaling (and +=, -=, /=, etc.)
a.apply(f);                  // a(i,j,k)=f(a(i,j)) for each element a(i,j,k)
a.apply(f,7);                // a(i,j,k)=f(a(i,j),7) for each element a(i,j,k)
b=apply(f,a);                // make a new Matrix with b(i,j,k)==f(a(i,j,k))
b=apply(f,a,7);              // make a new Matrix with b(i,j,k)==f(a(i,j,k),7)
a.swap_rows(7,9); // swap rows a[7] ⇔ a[9]
```

使用 Matrix

- 具体参见教材
  - Matrix I/O
    - §24.5.4; 它如你想象般工作
  - 求解线性方程组
    - §24.6; 就像你学习的代数教材上一样

随机数

- “随机数”是服从某种分布的序列中的一个数字，特点是你无法根据序列前一部分内容预测出下一个数
  - 均匀分布<cstdlib>
    - **int rand()** *// a value in [0:RAND_MAX]*
    - **RAND_MAX** *// the largest possible value for rand()*
    - **void srand(unsigned);** *// seed the random number generator*
  - 使用
    - **int rand_int(int max) { return rand()%max; }**
    - **int rand_int(int min, int max) {return min+rand_int(max-min); }**
  - 如果上述函数工作不太好(没有真正的随机)或你需要一个非均匀分布的随机数，需要使用专业库
    - e.g. **boost::random** (also C++0x)

复数

- 复数及其标准函数定义 <complex> 中

```
template<class T> class complex {
    T re, im; // a complex is a pair of scalar values, a coordinate pair
public:
    complex(const T& r, const T& i) :re(r), im(i) { }
    complex(const T& r) :re(r),im(T()) { }
    complex() :re(T()), im(T()) { }

    T real() { return re; }
    T imag() { return im; }

    // operators:  =  +=  -=  *=  /=
};

// operators:  +  -  /  *  ==  !=

// whatever standard mathematical functions that apply to complex:
// pow(), abs(), sqrt(), cos(), log(), etc. and also norm()  (square of abs())
```

复数

// use complex<T> exactly like a built-in type, such as double

// just remember that not all operations are defined for a complex (e.g. <)

```
typedef complex<double> dcmplx; // sometimes complex<double> gets verbose

void f( dcmplx z, vector< complex<float> >& vc)
{;
    dcmplx z2 = pow(z,2);
    dcmplx z3 = z2*9+vc[3];
    dcmplx  sum = accumulate(vc.begin(), vc.end(), dcmplx);
}
```

数值限制

- 每个 C++ 实现都指明了内建类型的属性
  - 用来检查数值限制、设置哨兵机制等
- 在 **<limits>** 中
  - 对每个类型
    - **min()** // smallest value
    - **max()** // largest value
    - …
  - 对浮点类型数
    - 有较多(需要时请查看)
    - e.g. **max_exponent10()**
- 在 **<limits.h>** 和 **<float.h>** 中
  - **INT_MAX** // largest **int** value
  - **DBL_MIN** // smallest **double** value

数值限制

- 它们对开发底层工具是很重要的
- 如果你觉得你需要它们，表明你的工作很可能比较靠近硬件，但这些属性还有其他用途
  - 例如：

```
void f(const vector<int>& vc)
{
    // pedestrian (and has a bug):
    int smallest1 = v[0];
    for (int i = 1; i<vc.size(); ++i) if (v[i]<smallest1) smallest1 = v[i];

    // better:
    int smallest2 = numeric_limits<int>::max();
    for (int i = 0; i<vc.size(); ++i) if (v[i]<smallest2) smallest2 = v[i];

    // or use standard library:
    vector<int>::iterator p = min_element(vc.begin(),vc.end());
    // and check for p==vc.end()
}
```

计算最小值

更多链接

- http://www-gap.dcs.st-and.ac.uk/~history/
  - 对喜欢数学的人，它是一个极好的网站
    - 或者简单的使用数学
  - 著名的数学家
    - 传记、成就，及一些“古董”
  - 著名的函数曲线
  - 著名的数学问题
  - 数学主题
    - 代数 Algebra
    - 分析 Analysis
    - 数论 Numbers and number theory
    - 几何和拓扑学 Geometry and topology
    - 数学物理学 Mathematical physics
    - 数理天文学 Mathematical astronomy
  - 数学历史
  - …