Abstract

- This lecture is an overview of the fundamental tools and techniques for numeric computation. For some, numerics are everything. For many, numerics are occasionally essential. Here, we present the basic problems of size, precision, truncation, and error handling in standard mathematical functions. We present multidimensional matrices and the standard library complex numbers.

Overview

- Precision, overflow, sizes, errors
- Matrices
- Random numbers
- Complex numbers

- Please note:
  - Again, we are exploring how to express the fundamental notions of an application domain in code
  - Numerical computation, especially linear algebra
  - We are showing the basic tools only. The computational techniques for using these tools well, say in engineering calculations, you’ll have to learn elsewhere
  - That’s “domain knowledge” beyond the scope of this presentation

Precision, etc.

- When we use the built-in types and usual computational techniques, numbers are stored in fixed amounts of memory
  - Floating-point numbers are (only) approximations of real numbers
    - float x = 1.0/333;
    - float sum = 0; for (int i=0; i<333; ++i) sum+=x;
    - cout << sum << "\n";
    - 0.999999
  - Integer types represent (relatively) small integers only
    - short y = 40000;
    - int i = 1000000;
    - cout << y << "   " << i*i << "\n";
    - -25536   -727379968
  - There just aren’t enough bits to exactly represent every number we need in a way that’s amenable to efficient computation

Sizes, overflow, truncation

// 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;  // chars really are very small integers
  s = i;
  l = f+1;  // what if i was the largest int?
  lg = f+1;  // beware: a long may not be any larger than an int
  // and anyway, 14 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 << i%ch << "\n"; ch++; }  // try this
}
If in doubt, you can check

- The simplest way to test
  - Assign, then compare

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

Math function errors

- If a standard mathematical function can’t do its job, it sets `errno` from `<cerrno>`, for example

```c
void f(double negative, double very_large)
{
    errno = 0; // no current errors
    sqrt(negative); // not a good idea
    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";
}
```

Matrices

- The standard `vector` and the built-in array are one dimensional
- What if we need 2 dimensions? (e.g. a matrix)
- N dimensions?

A vector (e.g. `Matrix<int> v(4)`), also called a 1 dimensional Matrix, or even a 1*N matrix

A 3*4 matrix (e.g. `Matrix<int> m(3,4)`), also called a 2 dimensional Matrix

C-style multidimensional Arrays

- A built-in facility

```c
int a[4]; // 1 dimensional array
double ad[3][4]; // 2 dimensional array
char ac[3][4][5]; // 3 dimensional array
```

- Basically, Arrays of Arrays

C-style multidimensional Arrays

- Problems
  - C-style multidimensional Arrays are Arrays of Arrays
  - Fixed sizes (i.e. fixed at compile time)
    - If you want to determine a size at run-time, you’ll have to use free store
    - Can’t be passed cleanly
    - An Array turns into a pointer at the slightest provocation
  - No range checking
    - As usual, an Array doesn’t know its own size
    - No Array operations
    - Not even assignment (copy)
  - A major source of bugs
    - And, for most people, they are a serious pain to write
  - Look them up only if you are forced to use them
    - E.g. TC++PL, Appendix C, pp 836-840
C-style multidimensional Arrays

- You can’t pass multidimensional Arrays cleanly
  ```c
  void f1(int a[3][5];)
  ```
  - 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
  ```c
  void f2(int [ ];[5], int dim1);
  ```
  - 1st dimension can be a variable
  ```c
  void f3(int[ ];[ ], int dim1, int dim2);
  ```
  - error (and wouldn't work anyway)
  ```c
  void f4(int* m, int dim1, int dim2)
  ```
  - odd, but works

  ```c
  for (int i=0; i<dim1; ++i)
  for (int j = 0; j<dim2; ++j) m[i*dim2+j] = 0;
  ```

A Matrix library

- “like your math/engineering textbook talks about Matrixs”
  - Or about vectors, matrices, tensors
- Matrices of any dimension
  - 1, 2, and 3 dimensions actually work (you can add more if/as needed)
  - Matrices are proper variables/objects
  - You can pass them around
- Usual Matrix operations
  - Subscripting: ( )
  - Slicing: [ ]
  - Assignment: =
  - Scaling operations (+, -, *, /, %, etc.)
  - Fused vector operations (e.g., res[i] = a[i] + b[i];)
  - Dot product (res = sum of a[i]*b[i])
- Performs equivalently to hand-written low-level code
- You can extend it yourself as needed (“no magic”)

```c
Matrix<double> a(3,4);
Matrix<int> b(3,4);
Matrix<double,2> c(3,4);
Matrix<double,3> d(3,4,5);
```
2D and 3D Matrices

// 2D space (e.g. a game board):
enum Piece { none, pawn, knight, queen, king, bishop, rook };
Matrix<Piece,2> board(R8,R8); // a chessboard
Piece init_pos[8] = { 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<int> a2 = a; // copy initialization
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) element (Fortran style), but range checked
a[i]; // ith row (C-style), but range checked

// 3D Matrix (very like 1D and 2D)
Matrix<int,3> a(10,20,30);
a.size(); // number of elements in dimension 1
a.dim1(); // number of elements in dimension 2
a.dim2(); // number of elements in dimension 3

// 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]; // ith element (Fortran style), but range checked

// Random numbers
- A “random number” is a number from a sequence that matches a distribution, but where it is hard to predict the next number in the sequence.
  - Uniformly distributed integers <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
  - Use
    - int rand_lnt(int max) { return rand()%max; }
    - int rand_int(int min, int max) { return rand()+(max-min); }
  - If that’s not good enough (not “random enough”) or you need a nonuniform distribution, use a professional library
    - E.g. boost::random (also C++0x)

// Using Matrix
- See book
  - Matrix I/O
    - §24.5.4; it’s what you think it is
  - Solving linear equations example
    - §24.6; it’s just like your algebra textbook says it is
Complex

Standard library complex types from `<complex>`

```cpp
template<class T>
class complex {
T re, im; // a complex is a pair of scalar values, a coordinate pair

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

```cpp
// 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);
}
```

Numeric limits

- Each C++ implementation specifies properties of the
  built-in types
  - used to check against limits, set sentinels, etc.
- From `<limits>`
  - for each type
    - min() // smallest value
    - max() // largest value
  - For floating-point types
    - Lots (look it up if you ever need it)
    - E.g. max_exponent10()
- From `<limits.h>` and `<float.h>`
  - INT_MAX // largest int value
  - DBL_MIN // smallest double value

A great link

- http://www-gap.dcs.st-and.ac.uk/~history/
  - A great link for anyone who likes math
  - Or simply needs to use math
  - Famous mathematicians
    - Biography, accomplishments, plus curio,
      for example, who is the only major mathematician to win an Olympic medal?
  - Famous curves
  - Famous problems
  - Mathematical topics
    - Algebra
    - Analysis
    - Numbers and number theory
    - Geometry and topology
    - Mathematical physics
    - Mathematical astronomy
  - The history of mathematics
  - …