## 1.随机数由生成器和分布器结合产生

生成器generator：能够产生离散的等可能分布数值

分布器distributions: 能够把generator产生的均匀分布值映射到其他常见分布，如均匀分布uniform，正态分布normal，二项分布binomial，泊松分布poisson

## 2.分布器利用运算符()产生随机数，要传入一个generator对象作为参数

**[cpp]** [view plaincopy](http://blog.csdn.net/akonlookie/article/details/8223525)

1. std::default\_random\_engine generator;
2. std::uniform\_int\_distribution<int> dis(0,100);
3. for(int i=0;i<5;i++)
4. {
5. std::cout<<dis(generator)<<std::endl;
6. }

        如果嫌每次调用都要传入generator对象麻烦，可以使用std::bind，要包含头文件functional

auto dice = std::bind(distribution,generator)以后就可以直接调用dice()产生复合均匀分布的随机数。但是多次运行上例会发现 每次产生的随机数序列都一样，因为没有设定种子（同cstdlib库中的rand和srand关系)

**[cpp]** [view plaincopy](http://blog.csdn.net/akonlookie/article/details/8223525)

1. std::default\_random\_engine generator;
2. std::uniform\_int\_distribution<int> dis(0,100);
3. auto dice= std::bind(dis,generator);
4. for(int i=0;i<5;i++)
5. {
6. std::cout<<dice()<<std::endl;
7. }

## 3.种子

        除了random\_device生成器（真随机数生成器或叫f非确定性随机数生成器）以外（linux中有效，windows下其实也是伪随机），所有在 库中定义的随机数引擎都是伪随机数生成器，他们都利用了特定的算法实现，这些生成器都需要一个种子。种子可以是一个数值，或者是一个带有generate 成员函数的对象。简单的应用中，用time作种子即可。

说明：如果不设定种子，那么产生的随机数序列每次都一样，如上代码，产生5个1到6之间的随机数，但是每次都是82 13 91 84 12

改为如下代码，可以使每次产生的随机数序列不同：

**[cpp]** [view plaincopy](http://blog.csdn.net/akonlookie/article/details/8223525)

1. std::default\_random\_engine generator(time(NULL));
2. std::uniform\_int\_distribution<int> dis(0,100);
3. auto dice= std::bind(dis,generator);
4. for(int i=0;i<5;i++)
5. {
6. std::cout<<dice()<<std::endl;
7. }

## 4.关于生成器

        C++11标准提供了三个生成器模版类可以实例化为生成器，但需要有一定的数学功底才懂得每个模版参数的意义，可参照算法出处的论文。

这三个生成器类模版为：

linear\_congruential\_engine 线性同余法

mersenne\_twister\_engine 梅森旋转法

substract\_with\_carry\_engine滞后Fibonacci

线性同余法举例

template <class UIntType, UIntType a, UIntType c, UIntType m>

class linear\_congruential\_engine;

第一个参数：生成器类型unsigned int,unsigned long等

第二到第四个参数：是线性同余法公递推公式Nj+i =(AxNj+C) (mod M)里的三个常数值A，C，M

要求：如果m不为0，a,c的值要小于m

如一会介绍的常用生成器：

**[cpp]** [view plaincopy](http://blog.csdn.net/akonlookie/article/details/8223525)

1. typedef linear\_congruential<unsigned long, 16807, 0, 2147483647> minstd\_rand0;

**[cpp]** [view plaincopy](http://blog.csdn.net/akonlookie/article/details/8223525)

1. typedef linear\_congruential<unsigned long, 48271, 0, 2147483647> minstd\_rand;

可见如果自己实例化模版类很麻烦，需要很强的数序知识，所以有几个常用的几个模版实例化生成器，他们都是需要一个种子参数就可以：

### 4.1线性同余法：

minstd\_rand()

minstd\_rand0

利用适配器变种后的线性同余法

knuth\_b     minstd\_rand0 with shuffle\_order\_engine

### 4.2梅森旋转法：

default\_random\_engine()

mt19937

mt19937\_64

### 4.3滞后Fibonacci法

ranlux24\_base

ranlux48\_base

利用适配器变种后的滞后Fibonacci法：

ranlux24              ranlux24\_base with discard\_block\_engine

ranlux48              ranlux48\_base with discard\_block\_engine

三个适配器：discard\_block\_engine     shuffle\_order\_engine   independent\_bits\_engine

## 5.关于分布器

易知，如果只用generator配上seed只能产生离散的等可能分布，产生的数值在generator的min和max之间，并且结果都是UIntType的值。无法很好的控制产生数值的分布区间和分布概率。如果要实现这种功能就要用到分布器。

作用1：改变生成类型，利用模版参数

作用2：改变值区间，利用实例构造函数参数。或其响应的成员函数设置参数。

作用3：改变概率分布，选用不同的分布器类型

### 5.1均匀分布：

uniform\_int\_distribution           整数均匀分布

uniform\_real\_distribution         浮点数均匀分布

### 5.2伯努利类型分布：（仅有yes/no两种结果，概率一个p，一个1-p）

bernoulli\_distribution     伯努利分布

binomial\_distribution      二项分布

geometry\_distribution     几何分布

negative\_biomial\_distribution   负二项分布

### 5.3 Rate-based distributions:

poisson\_distribution  泊松分布

exponential\_distribution 指数分布

gamma\_distribution 伽马分布

 weibull\_distribution 威布尔分布

extreme\_value\_distribution 极值分布

### 5.4正态分布相关：

normal\_distribution         正态分布

chi\_squared\_distribution 卡方分布

cauchy\_distribution        柯西分布

fisher\_f\_distribution       费歇尔F分布

student\_t\_distribution  t分布

### 5.5分段分布相关：

discrete\_distribution 离散分布

piecewise\_constant\_distribution 分段常数分布

piecewise\_linear\_distribution 分段线性分布



string &append(const char \*s); //把c类型字符串s连接到当前字符串结尾  
string &append(const char \*s,int n);//把c类型字符串s的前n个字符连接到当前字符串结尾  
string &append(const string &s); //同operator+=()  
string &append(const string &s,int pos,int n);//把字符串s中从pos开始的n个字符连接到当前字符串的结尾  
string &append(int n,char c); //在当前字符串结尾添加n个字符c  
string &append(const\_iterator first,const\_iterator last);//把迭代器first和last之间的部分连接到当前字符串的结尾

std::pair主要的作用是将两个数据组合成一个数据，两个数据可以是同一类型或者不同类型。例如 std::pair<int,float> 或者 std：：pair<double,double>等。pair实质上是一个结构体，其主要的两个成员变量是first和second，这两 个变量可以直接使用。初始化一个pair可以使用构造函数，也可以使用std::make\_pair函数，make\_pair函数的定义如下：

template pair make\_pair(t1 a, t2 b) { return pair(a, b); }

## eigen与matlab对应函数列表

// A simple quickref for Eigen. Add anything that's missing.

// Main author: Keir Mierle

#include <Eigen/Dense>

Matrix<double, 3, 3> A; // Fixed rows and cols. Same as Matrix3d.

Matrix<double, 3, Dynamic> B; // Fixed rows, dynamic cols.

Matrix<double, Dynamic, Dynamic> C; // Full dynamic. Same as MatrixXd.

Matrix<double, 3, 3, RowMajor> E; // Row major; default is column-major.

Matrix3f P, Q, R; // 3x3 float matrix.

Vector3f x, y, z; // 3x1 float matrix.

RowVector3f a, b, c; // 1x3 float matrix.

VectorXd v; // Dynamic column vector of doubles

double s;

// Basic usage

// Eigen // Matlab // comments

x.size() // length(x) // vector size

C.rows() // size(C,1) // number of rows

C.cols() // size(C,2) // number of columns

x(i) // x(i+1) // Matlab is 1-based

C(i,j) // C(i+1,j+1) //

A.resize(4, 4); // Runtime error if assertions are on.

B.resize(4, 9); // Runtime error if assertions are on.

A.resize(3, 3); // Ok; size didn't change.

B.resize(3, 9); // Ok; only dynamic cols changed.

A << 1, 2, 3, // Initialize A. The elements can also be

4, 5, 6, // matrices, which are stacked along cols

7, 8, 9; // and then the rows are stacked.

B << A, A, A; // B is three horizontally stacked A's.

A.fill(10); // Fill A with all 10's.

// Eigen // Matlab

MatrixXd::Identity(rows,cols) // eye(rows,cols)

C.setIdentity(rows,cols) // C = eye(rows,cols)

MatrixXd::Zero(rows,cols) // zeros(rows,cols)

C.setZero(rows,cols) // C = ones(rows,cols)

MatrixXd::Ones(rows,cols) // ones(rows,cols)

C.setOnes(rows,cols) // C = ones(rows,cols)

MatrixXd::Random(rows,cols) // rand(rows,cols)\*2-1 // MatrixXd::Random returns uniform random numbers in (-1, 1).

C.setRandom(rows,cols) // C = rand(rows,cols)\*2-1

VectorXd::LinSpaced(size,low,high) // linspace(low,high,size)'

v.setLinSpaced(size,low,high) // v = linspace(low,high,size)'

// Matrix slicing and blocks. All expressions listed here are read/write.

// Templated size versions are faster. Note that Matlab is 1-based (a size N

// vector is x(1)...x(N)).

// Eigen // Matlab

x.head(n) // x(1:n)

x.head<n>() // x(1:n)

x.tail(n) // x(end - n + 1: end)

x.tail<n>() // x(end - n + 1: end)

x.segment(i, n) // x(i+1 : i+n)

x.segment<n>(i) // x(i+1 : i+n)

P.block(i, j, rows, cols) // P(i+1 : i+rows, j+1 : j+cols)

P.block<rows, cols>(i, j) // P(i+1 : i+rows, j+1 : j+cols)

P.row(i) // P(i+1, :)

P.col(j) // P(:, j+1)

P.leftCols<cols>() // P(:, 1:cols)

P.leftCols(cols) // P(:, 1:cols)

P.middleCols<cols>(j) // P(:, j+1:j+cols)

P.middleCols(j, cols) // P(:, j+1:j+cols)

P.rightCols<cols>() // P(:, end-cols+1:end)

P.rightCols(cols) // P(:, end-cols+1:end)

P.topRows<rows>() // P(1:rows, :)

P.topRows(rows) // P(1:rows, :)

P.middleRows<rows>(i) // P(:, i+1:i+rows)

P.middleRows(i, rows) // P(:, i+1:i+rows)

P.bottomRows<rows>() // P(:, end-rows+1:end)

P.bottomRows(rows) // P(:, end-rows+1:end)

P.topLeftCorner(rows, cols) // P(1:rows, 1:cols)

P.topRightCorner(rows, cols) // P(1:rows, end-cols+1:end)

P.bottomLeftCorner(rows, cols) // P(end-rows+1:end, 1:cols)

P.bottomRightCorner(rows, cols) // P(end-rows+1:end, end-cols+1:end)

P.topLeftCorner<rows,cols>() // P(1:rows, 1:cols)

P.topRightCorner<rows,cols>() // P(1:rows, end-cols+1:end)

P.bottomLeftCorner<rows,cols>() // P(end-rows+1:end, 1:cols)

P.bottomRightCorner<rows,cols>() // P(end-rows+1:end, end-cols+1:end)

// Of particular note is Eigen's swap function which is highly optimized.

// Eigen // Matlab

R.row(i) = P.col(j); // R(i, :) = P(:, i)

R.col(j1).swap(mat1.col(j2)); // R(:, [j1 j2]) = R(:, [j2, j1])

// Views, transpose, etc; all read-write except for .adjoint().

// Eigen // Matlab

R.adjoint() // R'

R.transpose() // R.' or conj(R')

R.diagonal() // diag(R)

x.asDiagonal() // diag(x)

R.transpose().colwise().reverse(); // rot90(R)

R.conjugate() // conj(R)

// All the same as Matlab, but matlab doesn't have \*= style operators.

// Matrix-vector. Matrix-matrix. Matrix-scalar.

y = M\*x; R = P\*Q; R = P\*s;

a = b\*M; R = P - Q; R = s\*P;

a \*= M; R = P + Q; R = P/s;

R \*= Q; R = s\*P;

R += Q; R \*= s;

R -= Q; R /= s;

// Vectorized operations on each element independently

// Eigen // Matlab

R = P.cwiseProduct(Q); // R = P .\* Q

R = P.array() \* s.array();// R = P .\* s

R = P.cwiseQuotient(Q); // R = P ./ Q

R = P.array() / Q.array();// R = P ./ Q

R = P.array() + s.array();// R = P + s

R = P.array() - s.array();// R = P - s

R.array() += s; // R = R + s

R.array() -= s; // R = R - s

R.array() < Q.array(); // R < Q

R.array() <= Q.array(); // R <= Q

R.cwiseInverse(); // 1 ./ P

R.array().inverse(); // 1 ./ P

R.array().sin() // sin(P)

R.array().cos() // cos(P)

R.array().pow(s) // P .^ s

R.array().square() // P .^ 2

R.array().cube() // P .^ 3

R.cwiseSqrt() // sqrt(P)

R.array().sqrt() // sqrt(P)

R.array().exp() // exp(P)

R.array().log() // log(P)

R.cwiseMax(P) // max(R, P)

R.array().max(P.array()) // max(R, P)

R.cwiseMin(P) // min(R, P)

R.array().min(P.array()) // min(R, P)

R.cwiseAbs() // abs(P)

R.array().abs() // abs(P)

R.cwiseAbs2() // abs(P.^2)

R.array().abs2() // abs(P.^2)

(R.array() < s).select(P,Q); // (R < s ? P : Q)

// Reductions.

int r, c;

// Eigen // Matlab

R.minCoeff() // min(R(:))

R.maxCoeff() // max(R(:))

s = R.minCoeff(&r, &c) // [s, i] = min(R(:)); [r, c] = ind2sub(size(R), i);

s = R.maxCoeff(&r, &c) // [s, i] = max(R(:)); [r, c] = ind2sub(size(R), i);

R.sum() // sum(R(:))

R.colwise().sum() // sum(R)

R.rowwise().sum() // sum(R, 2) or sum(R')'

R.prod() // prod(R(:))

R.colwise().prod() // prod(R)

R.rowwise().prod() // prod(R, 2) or prod(R')'

R.trace() // trace(R)

R.all() // all(R(:))

R.colwise().all() // all(R)

R.rowwise().all() // all(R, 2)

R.any() // any(R(:))

R.colwise().any() // any(R)

R.rowwise().any() // any(R, 2)

// Dot products, norms, etc.

// Eigen // Matlab

x.norm() // norm(x). Note that norm(R) doesn't work in Eigen.

x.squaredNorm() // dot(x, x) Note the equivalence is not true for complex

x.dot(y) // dot(x, y)

x.cross(y) // cross(x, y) Requires #include <Eigen/Geometry>

//// Type conversion

// Eigen // Matlab

A.cast<double>(); // double(A)

A.cast<float>(); // single(A)

A.cast<int>(); // int32(A)

A.real(); // real(A)

A.imag(); // imag(A)

// if the original type equals destination type, no work is done

// Note that for most operations Eigen requires all operands to have the same type:

MatrixXf F = MatrixXf::Zero(3,3);

A += F; // illegal in Eigen. In Matlab A = A+F is allowed

A += F.cast<double>(); // F converted to double and then added (generally, conversion happens on-the-fly)

// Eigen can map existing memory into Eigen matrices.

float array[3];

Vector3f::Map(array).fill(10); // create a temporary Map over array and sets entries to 10

int data[4] = {1, 2, 3, 4};

Matrix2i mat2x2(data); // copies data into mat2x2

Matrix2i::Map(data) = 2\*mat2x2; // overwrite elements of data with 2\*mat2x2

MatrixXi::Map(data, 2, 2) += mat2x2; // adds mat2x2 to elements of data (alternative syntax if size is not know at compile time)

// Solve Ax = b. Result stored in x. Matlab: x = A \ b.

x = A.ldlt().solve(b)); // A sym. p.s.d. #include <Eigen/Cholesky>

x = A.llt() .solve(b)); // A sym. p.d. #include <Eigen/Cholesky>

x = A.lu() .solve(b)); // Stable and fast. #include <Eigen/LU>

x = A.qr() .solve(b)); // No pivoting. #include <Eigen/QR>

x = A.svd() .solve(b)); // Stable, slowest. #include <Eigen/SVD>

// .ldlt() -> .matrixL() and .matrixD()

// .llt() -> .matrixL()

// .lu() -> .matrixL() and .matrixU()

// .qr() -> .matrixQ() and .matrixR()

// .svd() -> .matrixU(), .singularValues(), and .matrixV()

// Eigenvalue problems

// Eigen // Matlab

A.eigenvalues(); // eig(A);

EigenSolver<Matrix3d> eig(A); // [vec val] = eig(A)

eig.eigenvalues(); // diag(val)

eig.eigenvectors(); // vec

std:map<int, CString> enumMap;

这样就定义了一个用int作为关键字检索CString条目的map对象，std表示命名空间，map对象在std名字空间中，为了方便，在这 里我仍然使用了CString类，其实应该使用标准C++的std::string类，我们对模板类进行一下类型定义，这样用的方便，当然，不定义也可 以，代码如下：

typedef std:map<int, CString> UDT\_MAP\_INT\_CSTRING;  
UDT\_MAP\_INT\_CSTRING enumMap;

如此map对象就定义好了，增加，改变map中的条目非常简单，因为map类已经对[]操作符进行了重载，代码如下:

enumMap[1] = "One";  
enumMap[2] = "Two";  
.....  
enumMap[1] = "One Edit";

或者insert方法

enumMap.insert(make\_pair(1,"One"));

返回map中目前存储条目的总数用size()方法：

int nSize = enumMap.size();

查找map中是否包含某个关键字条目用find方法，传入的参数是要查找的key，在我们的例子里，是一个int数据，map中的条目数据是顺 序存储的，被称作为一个sequence，在这里需要提到的是begin()和end()两个成员，分别代表map对象中第一个条目和最后一个条目，这两 个数据的类型是iterator，iterator被定义为map中条目的类型，查找是否包含某个条目的代码如下：

int nFindKey = 2;            //要查找的Key  
UDT\_MAP\_INT\_CSTRING::iterator it;    //定义一个条目变量(实际是指针)  
it = enumMap.find(nFindKey);  
if(it == enumMap.end()) {  
    //没找到  
}  
else {  
    //找到  
}

//find的时候注意key的数据类型，最好用CString之类的能消除数据类型差异的key，否则可能会出现强制转换后仍找不到的情况。

需要说明的是iterator, begin(), end()是STL模板类的一个通用概念，操作方法也大同小异

通过map对象的方法获取的iterator数据类型是一个std::pair对象，包括两个数据 iterator.first 和 iterator.second 分别代表关键字和存储的数据

移除某个条目用erase() 该成员方法的定义如下

iterator erase(iterator it);  
iterator erase(iterator first, iterator last);  
size\_type erase(const Key& key);

分析一下这三个重载方法定义，大家不用说也能看明白一点点了吧，第一个通过一个条目对象删除，这个对象可以从find之类的方法获得，第二个定义删除一个范围，需要一个起始条目和一个终止条目，第三个通过关键字删除，这个与我们的想法和习惯最接近，代码例子如下：

enumMap.erase(1);            //删掉关键字“1”对应的条目  
enumMap.erase(enumMap.begin());        //删掉第一个条目  
enumMap.erase(enumMap.begin(), enumMap.begin() + 1);    //删掉起始的两个条目

呵呵，增删改查都说完了，相信读过本文，map对象也应该会使用了，这些是我1个多星期来钻研的结果，拿出来与大家分享。

最后，还有一个clear()，不用问，全删的时候就不要一个一个erase了,clear()就相当于enumMap.erase(enumMap.begin(), enumMap.end());