JOHN D. COOK
CONSULTING

Navigation

Random number generation using C++ TR1

Overview

This article explains how to use the random number generation facilities in C++ using the TR1 (C++ Standards Committee Technical Report 1) extensions. We will cover basic uniform random number generation as well as generating samples from common probability distributions: Bernoulli, binomial, exponential, gamma, geometric, normal, and Poisson. We will point out a few things to watch out for with specific distributions such as parameterization conventions. Finally we will indicate how to generate from probability distributions not directly supported in the TR1 such as Cauchy, chi-squared, and Student t.

Support for TR1 extensions in Visual Studio 2008 is added as a feature pack. Other implementations include the Boost and Dinkumware. GCC added experimental support for C++ TR1 in version 4.3.

The code samples in this article use fully qualified namespaces for clarity. You could make your code easier to read by adding a few `using` statements to eliminate namespace qualifiers. See testing C++ TR1 random number generation for more sample code and a crude test of (your understand of) the distribution classes.

Outline

Getting started

Header and namespace

The C++ random number generation classes and functions are defined in the `<random>` header and contained in the namespace `std::tr1`. Note that `tr` is lower-case in C++. In English text “TR” is capitalized.

Core engines

At the core of any pseudorandom number generation software is a routine for generating uniformly distributed random integers. These are then used in a sort of bootstrap process to generate uniformly distributed real numbers. The uniform real numbers are used to generate from other distributions through transformations, acceptance-rejection algorithms, etc. It all starts with that original source of random integers.

In C++ TR1 you have your choice of several core generators that it calls "engines." The following four engine classes are supported in the Visual Studio 2008 feature pack.

- `linear_congruential` uses a recurrence of the form $x(i) = (A * x(i\text{-}1) + C) \bmod M$
- `mersenne_twister` implements the famous Mersenne Twister algorithm
- `subtract_with_carry` uses a recurrence of the form $x(i) = (x(i - R) - x(i - S) - cy(i - 1)) \bmod M$ in integer arithmetic
- `subract_with_carry_01` uses a recurrence of the form $x(i) = (x(i - R) - x(i - S) - cy(i - 1)) \bmod 1$ in floating point arithmetic

Setting seeds

Each engine has a `seed()` method that accepts an `unsigned long` argument to specify the random number generation seed. It is also possible to set the seed in more detail using template parameters unique to each engine.

Generating from a uniform distribution

Discrete integers

The class `uniform_int` picks integers with equal probability in a specified range. The constructor takes two arguments, the minimum and maximum numbers to pick from. Note that both extremes are included as possible values.

Example:

The following code will print 5 numbers picked randomly from the set 1, 2, 3, …, 52. This could model picking five cards from a deck of 52 cards if the cards are selected one at a time and placed back into the deck before the deck is shuffled and another card selected.

```
std::tr1::uniform_int<int> unif(1, 52);
for (int i = 0; i < 5; ++i)
    std::cout << unif(eng) << std::endl;
```

Real interval

Use the class `uniform_real` to generate floating point numbers from an interval. The constructor takes two arguments, the end points of the interval. These values default to 0 and 1. NB: the class generates values from a half-open interval. Given arguments *a* and *b*, the class generates values in the half-open interval [*a*, *b*). In other words, all values *x* satisfy *a* <= *x* < *b*.

The following code generates a random value from the interval [3, 7).

```
std::tr1::uniform_real<double> unif(3, 7);
double u = unif(eng);
```

Generating from non-uniform distributions

The C++ TR1 library supports non-uniform random number generation through distribution classes. These classes return random samples via the `operator()` method. This method takes an engine class as an argument. For example, the following code snippet prints 10 samples from a standard normal distribution.

```
std::tr1::mt19937 eng;  // a core engine class
std::tr1::normal_distribution<double> dist;
for (int i = 0; i < 10; ++i)
    std::cout << dist(eng) << std::endl;
```

Although the TR1 specification and the Visual Studio documentation both indicate that the distribution classes take default template parameter arguments, they appear to be required. Hence the `normal_distribution` class above is declared to have return type `double` even though `double` should be the default type.

All examples below will assume an engine class `eng` is in use.

Distributions directly supported

Bernoulli

A Bernoulli random variable returns 1 with probability *p* and 0 with probability 1-*p*. The class for generating samples from Bernoulli distribution is `bernoulli_distribution`. The constructor for this class takes one parameter, *p*. This parameter defaults to 0.5.

Interestingly the `bernoulli_distribution` class returns a `bool` rather than an `int` and so it actually returns true with probability *p* and false with probability 1-*p*. This is the only distribution class that returns a Boolean rather than a numeric type. For that reason **this is also the only distribution class that does not take a return type template parameter**.

Example:

The following code assumes an engine `eng` has been defined. It will print “six of one” a quarter of the time and will print “half dozen of another” the rest of the time.

```
std::tr1::bernoulli_distribution bern(0.25);
```

```
        std::cout << (bern(eng) ? "six of one" : "half dozen of another") <
```

Binomial

A binomial random variable returns the number of successes out of n independent Bernoulli trials, each with probability of success *p*. The class for generating samples from a binomial distribution, `binomial_distribution`, takes two template parameters: the type of *n* and the type of *p*. These default to `int` and `double` respectively. The constructor uses the default arguments *n* = 1 and *p* = 0.5.

Example:

The following code will simulate rolling five dice and counting the number of dice that come up with 1 showing. (All outcomes are equally likely, so you could think of this as counting the number of times any other specified number comes up.

```
        std::tr1::binomial_distribution<int, double> roll(5, 1.0/6.0);
        std::cout << roll(eng) << std::endl;
```

Exponential

The exponential distribution has two common parameterizations: rate and mean. Note that **the C++ TR1 standard specifies the rate parameterization**. Under this parameterization, the density function is $f(x) = \lambda e^{-\lambda x}$ and the mean is 1/λ.

The class for generating samples from an exponential distribution is `exponential_distribution`. If no rate parameter is specified, the constructor uses the default value λ = 1.

Example:

The following code finds the sum of 10 draws from an exponential distribution with mean 13.

```
const double mean = 13.0;
double sum = 0.0;
std::tr1::exponential_distribution<double< exponential(1.0/mean);
for (int i = 0; i < 10; ++i)
    sum += exponential(eng);
```

Gamma

The gamma distribution as specified in TR1 is parameterized so that the density function is $f(x) = x^{\alpha-1}e^{-x}/\Gamma(\alpha)$. The α parameter is known as the shape parameter. A more general parameterization includes a scale parameter β. With this parameterization the density function of the gamma distribution is $f(x) = (x/\beta)^{\alpha-1}e^{-x/\beta}/(\beta\Gamma(\alpha))$. The TR1 parameterization corresponds to β = 1.

There is another less common parameterization that uses a parameter corresponding to 1/β. I suspect that the standard committee adopted to use the one-parameter version of the gamma distribution to avoid confusion regarding the second parameter. Regardless of their motivation, there is no loss of generality in only implementing the one-parameter version. To generate samples from a gamma distribution with scale β, simply generate samples from a gamma distribution with scale 1 and multiply the samples by β.

The class for generating samples from a gamma distribution is `gamma_distribution`. The constructor takes one argument, the shape parameter α. This parameter defaults to 1.

Example:

The following code prints 10 samples from a gamma distribution with shape 3 and scale 1.

```
std::tr1::gamma_distribution<double> gamma(3.0);
for (int i = 0; i < 10; ++i)
    std::cout << gamma(eng) << std::endl;
```

Example:

The following code prints 10 samples from a gamma distribution with shape 3 and scale 5.

```
std::tr1::gamma_distribution<double> gamma(3.0);
for (int i = 0; i < 10; ++i)
    std::cout << 5.0*gamma(eng) << std::endl;
```

Geometric

A geometric distribution with parameter p counts the number of Bernoulli needed in order to see the first success where each trial has probability of success *p*.

The class for generating samples from a geometric distribution is `geometric_distribution`. The constructor takes one argument, the probability of success *p*.

Example:

The following code shows how to draw a sample from a geometric distribution with one chance in ten of success on each trial.

```
std::tr1::geometric_distribution<int, double> geometric(0.1);
```

```
        std::cout << geometric(eng) << std::endl;
```

Normal (Gaussian)

The normal (Gaussian) distribution takes two parameters, the mean μ and standard deviation σ.

The class for generating samples from a normal distribution is `normal_distribution`. Its constructor takes two parameters, μ and σ. These values default to 0 and 1 respectively if not explicitly supplied. Note that the class parameterizes the normal distribution in terms of its standard deviation, not its variance.

Example:

The following code prints a sample from a normal distribution with mean 3 and standard deviation 4 (i.e. variance 16).

```
        std::tr1::normal_distribution<double> normal(3.0, 4.0);
        std::cout << normal(eng) << std::endl;
```

Poisson

The Poisson distribution models the probability of various numbers of rare, independent events in a given time interval.

The class for generating samples from a Poisson distribution is `poisson_distribution`. The constructor takes one argument, the mean λ of the distribution.

Example:

The following code show how to draw a sample from a Poisson distribution with mean 7.

```
std::tr1::poisson_distribution<double> poisson(7.0);
std::cout << poisson(eng) << std::endl;
```

Other distributions

The C++ TR1 specification only includes some of the most common probability distributions. Other distributions can be derived as combinations of the distributions supported directly. The distributions below are only examples. There are many more possibilities described in the statistical literature.

Cauchy

To generate random samples from a Cauchy distribution, first generate a uniform sample from (-π/2, π/2) and take the tangent.

```
const double pi = 3.14159265358979;
std::tr1::uniform_real<double> uniform(-0.5*pi, 0.5*pi);
double u = uniform(eng);
std::cout << tan(u) << std::endl;
```

There’s something a little subtle in the code above. Since `uniform_real` generates from the half-open interval, it can sometimes return the left endpoint of the interval. The tangent function would overflow if evaluated exactly at -π/2. However, our numerical value `pi` above is slightly less than the true value of π due to truncating and so the `tan` function will not have a problem if `uniform` returns its smallest possible value.

Chi-squared

The chi-squared (χ^2) distribution is a special case of the gamma distribution. A chi squared distribution with ν degrees of freedom is the same as a gamma distribution with shape parameter ν/2 and scale parameter 2. Since the C++ TR1 implementation only directly supports gamma distributions with scale parameter 1, generate a sample from a gamma distribution with shape ν/2 and unit scale then multiply the sample by 2.

The following code generate a sample from a chi-squared distribution with 5 degrees of freedom.

```
double nu = 5.0;
std::tr1::gamma_distribution<double> gamma(0.5*nu);
std::cout << 2.0*gamma(eng) << std::endl;
```

Student *t*

To generate a sample from a Student *t* distribution with ν degrees of freedom, first generate a sample *X* from a normal(0,1) distribution and a sample *Y* from a chi-squared distribution with ν degrees of freedom. Then return *X*/sqrt(*Y*/ν).

The following code generates a sample from a Student t distribution with 7 degrees of freedom

```
double nu = 7.0;
std::tr1::normal_distribution<double> normal;
std::tr1::gamma_distribution<double> gamma(0.5*nu);
double x = normal(eng);
double y = 2.0*gamma(eng);
std::cout << x/sqrt(y/nu) << std::endl;
```

Snedecor *F*

To generate a sample from a Snedecor *F* distribution with ν and ω degrees of freedom, generate a sample *X* from a chi-squared distribution with ν degrees of freedom and a sample *Y* from a chi-squared distribution with ω degrees of freedom. Then return (*X*/ν)/(*Y*/ω).

Weibull

To generate samples from a Weibull distribution with shape parameter γ and scale parameter β, first generate a uniform sample u from (0, 1) then return β $(-\log(u))^{1/\gamma}$. (Note: There are at least two common parameterizations of the Weibull. These notes assume the parameterization given here.)

Rayleigh

To generate samples from a Rayleigh distribution with scale *b*, generate a uniform sample *u* from (0, 1) and return sqrt(-2 b^2 log(*u*)).

Troubleshooting

If you have trouble linking with the random number generation library in Visual Studio 2008, this post may help.

Other C++ resources

- regular expressions
- strings
- floating point exceptions
- math.h

RANDOMIZATION HELP