Lazy generating equispaced non-integral values in range-based for loop in C++14

René Richter*

2015-07-09

Abstract

Due to rounding effects, iterations over floating-point values should be handled with care. Range-based for loops in modern C++ offer the opportunity to hide the tricky parts in lazy range generators without runtime overhead.

Motivation

Task: Generate a loop for n+1 equispaced values x_i in $[a, b] \in R$ with $i = 0...n$ (Fig. 1).

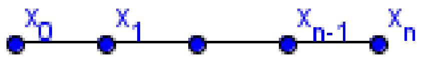

Figure 1: Line of equispaced points `x` in R, C or 3D.

Tricky floating-point numbers

Not obvious to absolute beginners, the naïve attempt

```
for (double x = a; x <= b; x += (b-a)/n) { /* ... fails */ }
```

will fail for most values of a, b, and n. It does not even guarantee to produce n+1 values. Floating-point numbers are a subset of R with limited accuracy[1]. Floating-point computations result in rounded values, if the result is not representable in floating-point machine format.[2] Abstain also from the equivalent

```
double dx = (b-a)/n;
for (double x = a; x <= b; x += dx) { /* ... fails */ }
```

or from

*http://namespace-cpp.de, C++ User Group Dresden: http://cpp-ug-dresden.blogspot.de/.

[1] 6 to 7 decimals for IEEE binary32 (`float`), 15 to 16 decimals for IEEE binary64 (`double`).

[2] See http://www.floating-point-gui.de.

```
double dx = (b-a)/n, stop = b + dx/2;
for (double x = a; x <= stop; x += dx) { /* ... fails */ }
```

loops. They accumulate rounding errors and are not even guaranteed to stop. If you are “lucky”, dx will evaluate to zero, or adding a small dx to a large x, it may give the same x ever and ever again, never reaching b — resulting in an infinite loop.

A better approach would guarantee n+1 values by using an integral loop counter and a scale transform:

```
for (int i = 0; i <= n; ++i)
{
    double x = ((n-i)*a + i*b)/n;
    /* ... */
}
```

However, for large values of n and a or b the computation may overflow, resulting in infinite values for x. Therefore, the division by n should be executed before multiplication, preferably before entering the loop body for better runtime performance. Refactoring the loop would give something like

```
double dx = (b-a)/n;
for (int i = 0; i <= n; ++i)
{
    double x = a + i*dx;
    /* ... */
}
```

Due to rounding errors, the last value `a+n*dx` is not guaranteed to be b, but a close neighbour (if dx not evaluates to zero). A second glance reveals some advantages:

- The loop gives exactly n+1 values, regardless if a < b, b < a, or a == b. It does not even insist, that the type of x is aware of comparison operators (e.g. complex numbers).
- Starting the for loop with `i = 1` or/and ending with `i <= n-1`, respectively, you can sample through (half-)open intervals, excluding the domain boundary a or/and b.

C++11 range-based for loops

The hand-written loop control is verbose. C++11 introduces range-based for loops, but creating a sequence of values before entering the loop would be a slow, memory-consuming task, with infinite recursion to the starting problem:

```
for (auto x : { 0.0, 0.25, 0.5, 0.75, 1.0 }) { /* ... */ }
```

The approach proposed here is not restricted to floating-point numbers. It also works for complex numbers and other types, providing operations + - * and /. Resembling a NumPy or Matlab expression,

```
for (auto x : linspace(a, b, n)) { /* ... */ }
for (auto x : linspace(a, b, n, bounds::open)) { /* ... */ }
for (auto x : linspace(a, b, n, bounds::leftopen)) { /* ... */ }
for (auto x : linspace(a, b, n, bounds::rightopen)) { /* ... */ }
```

will lazily generate equispaced values of x with n spaces between a and b.[3]

[3]Caveat: In Matlab and NumPy n denotes the number of generated values. The solution provided here does not shift sampling points, when choosing closed or (half-)open bounds.

Range iterators for lazy evaluation

A range-based for loop

```
for (declaration : expression) statement
```

is C++11's syntactic sugar, roughly equivalent[4] to something like

```
{
    auto&& __range = expression;
    for (auto __b = __range.begin(), __e = __range.end(); __b != __e; ++__b)
    {
        declaration = *__b;
        statement
    }
}
```

The `__range` is not necessarily a sequence container. It can be a placeholder which in turn gives forward iterators[5] `__b` and `__e` to the compiler:

```
struct SomeRangeGenerator
{
    class iterator;
    // ...
    iterator begin() const;
    iterator end()   const;
};
```

Lazy generating iterators provide one value at a time ("on demand") by calling operator `*`. The next value is computed by calling operator `++`. As long as operator `!=` returns `true`, the loop is not finished.

This technique is well-established[6] and used in libraries like Boost.Range[7] or cppitertools[8], and may be the foundation for the next Standard C++ Template Library version (Eric Niebler's "Ranges v3" proposal[9]). Also, more sophisticated applications[10] are possible.

A recent article[11] stated the problem of correctly iterating through a range of floating-point values. However, it fell short of providing a solution for them, and concentrated on integral domain types instead. In the following, the techniques shown there are applied to non-integral domains.

Modern C++ under the hood

A class template for the iteration domain $[a, b]$ is defined inside a namespace called `detail`:

[4]Simplified from http://en.cppreference.com/w/cpp/language/range-for.

[5]A forward iterator provides operators `*` for dereferencing a value, `++` for increment, and checks for equality `==` and `!=`.

[6]Matthew Wilson and John Torjo: Ranges, part 1: Concepts and Implementation, C/C++ User's Journal, Volume 22 Number 10, October (2004).

[7]http://www.boost.org/doc/libs/1_58_0/libs/range/doc/html/index.html.

[8]A C++ port of Python's itertools library. https://github.com/ryanhaining/cppitertools.

[9]See https://github.com/ericniebler/range-v3.

[10]Karsten Ahnert: Ranges and iterators for numerical problems, Meeting C++ (2014).

[11]Mikhail Semenov: Convenient Constructs For Stepping Through a Range. http://www.codeproject.com/Articles/876156/Convenient-Constructs-For-Stepping-Through-a-Range.

```
template <typename Domain, typename N>
class LinearGenerator
{
    Domain a_, dx_;
    N first_, last_;
public:
    iterator begin() const { return { a_, dx_, first_ }; }
    iterator end()   const { return { a_, dx_, last_ + 1 }; }
    // ...
};
```

It passes all needed information (start value `a`, step distance `dx` and step number) to iterators. These values come from the constructor:

```
    LinearGenerator(Domain a, Domain b, N n, N first, N last)
    : a_(a), dx_((b-a)*(1/scalar(n))), first_(first), last_(last)
    {}
```

Computing of the step distance as `dx = (b-a)/n` works for `float` or `double` domain type, since the integral value `n` is implicitly converted in floating-point, but not for complex numbers or vectors in linear algebra. These more "complex" types have an underlying scalar type, a metric or norm `abs(x)` and should define scalar multiplication: `dx = (b-a)*(1/scalar(n))`. With a conversion helper function

```
    static auto scalar(N n)
    {
        using std::abs;
        using ScalarType = decltype(abs(Domain{}));
        return static_cast<ScalarType>(n);
    }
```

we can go the extra mile, the compiler does not do.[12]

The iterator implementation is straight forward and mostly trivial:

```
    class iterator : public std::iterator<std::forward_iterator_tag, Domain>
    {
        Domain a_, dx_;
        N i_;
    public:
        iterator() : i_(0) {}
        iterator(Domain a, Domain dx, N i)
        : a_(a), dx_(dx), i_(i)
        {}

        bool operator==(const iterator& rhs) const { return i_ == rhs.i_; }
        bool operator!=(const iterator& rhs) const { return !(*this == rhs); }

        iterator& operator++()      { ++i_; return *this; }
        iterator  operator++(int)   { auto tmp = *this; ++*this; return tmp; }
        auto      operator*() const { return a_ + scalar(i_) * dx_; }
    };
```

[12] C++ allows only one implicit conversion.

The operator * accomplishes the computation in the same way as in the handwritten loop.

The boundary type of the interval $[a, b]$

```
enum class boundary { closed, rightopen, leftopen, open };
```

can be given as an optional parameter to a function, that encapsulates the range generator construction for convenience:

```
template <typename Start, typename End, typename N>
auto linspace(Start a, End b, N n, boundary type = boundary::closed)
{
    using Domain = decltype(a + (b - a));
    static_assert(!std::is_integral<Domain>::value, "use non-integral [a,b]");
    static_assert(std::is_integral<N>::value,       "use integral n");

    if (n < 1)
    {
        n = 1;
        type = boundary::open;
    }
    bool start_at_one = type == boundary::open || type == boundary::leftopen;
    bool end_before_n = type == boundary::open || type == boundary::rightopen;
    N first = start_at_one;
    N last  = n - end_before_n;

    return detail::LinearGenerator<Domain, N>(a, b, n, first, last);
}
```

The domain type of the interval $[a, b]$ can't be of integral type, whereas n shall be integral. By defining the domain type as `decltype(a + (b - a))` the function call allows not only floating-point types, but also user-defined types that provide operators for addition and subtraction, and mixing types for `a` and `b`, e.g. `int` and `double`, as long as the compiler can implicitly convert one type into another for these operations.[13]

Here, C++14's automatic function return type deduction reveals it's beauty. As alternatives, trailing return type syntax

```
template <typename Start, typename End, typename N>
auto linspace(Start a, End b, N n, boundary type = boundary::closed)
    -> detail::LinearGenerator<decltype(a + (b - a)),N>
{ /* ... */ }
```

or an extra default template parameter allow implementation in C++11:

```
template <typename Start, typename End, typename N,
    typename Domain = decltype(Start{} + (End{} - Start{}))>
detail::LinearGenerator<Domain,N>
linspace(Start a, End b, N n, boundary type = boundary::closed)
{ /* ... */ }
```

[13]According to C++ rules, `decltype(1.0+std::complex<float>(0,1))` is `std::complex<float>`, not `std::complex<double>` as one might think at first.

Comparison to other libraries

```
for (auto x : loop::linspace(a, b, n)) { /* ... */ }
```

(defined in `namespace loop`) achieves the same effect as the Boost.Range library in a more verbose manner:

```
for (auto x :
    boost::irange(0, n + 1) |
    boost::adaptors::transformed(
        [a, dx = (b-a)/n](int i) { return a + i*dx; } ))
{ /* ... */ }
```

A range from cppItertools requires all parameters to have exactly the same type:

```
auto dx = (b-a)/n;
for (auto x : iter::range(a, b + dx/2, dx))
{ /* ... */ }
```

Due to rounding errors, it may not guarantee n+1 values $x_0 \ldots x_n$. None of these alternatives is clearer and easier to use than the handwritten loop.

Performance

A benchmark using a C++14 library[14] on `double` type values shows no runtime overhead of lazy generated ranges over best handwritten for loops. Both timings are equal within clock resolution (Fig. 2). Noteworthy, a handwritten and inaccurate `x += dx` loop resulting in wrong loop count is slower than `x = a + i*dx` and the equivalent lazy generated range for `double` values. Interpolated values without and with division inside loop body have the worst runtime performance.

Conclusion

Range-based lazy generation provides a means for correctly working, easy to write loop control for floating-point and user-defined algebraic types without perfomance overhead. The implementation is done in a header-only library for easy reuse.[15]

[14] Nick Athanasiou: Benchmarking in C++. https://ngathanasiou.wordpress.com/2015/04/01/benchmarking-in-c/.
[15] See https://bitbucket.org/dozric/looprange.

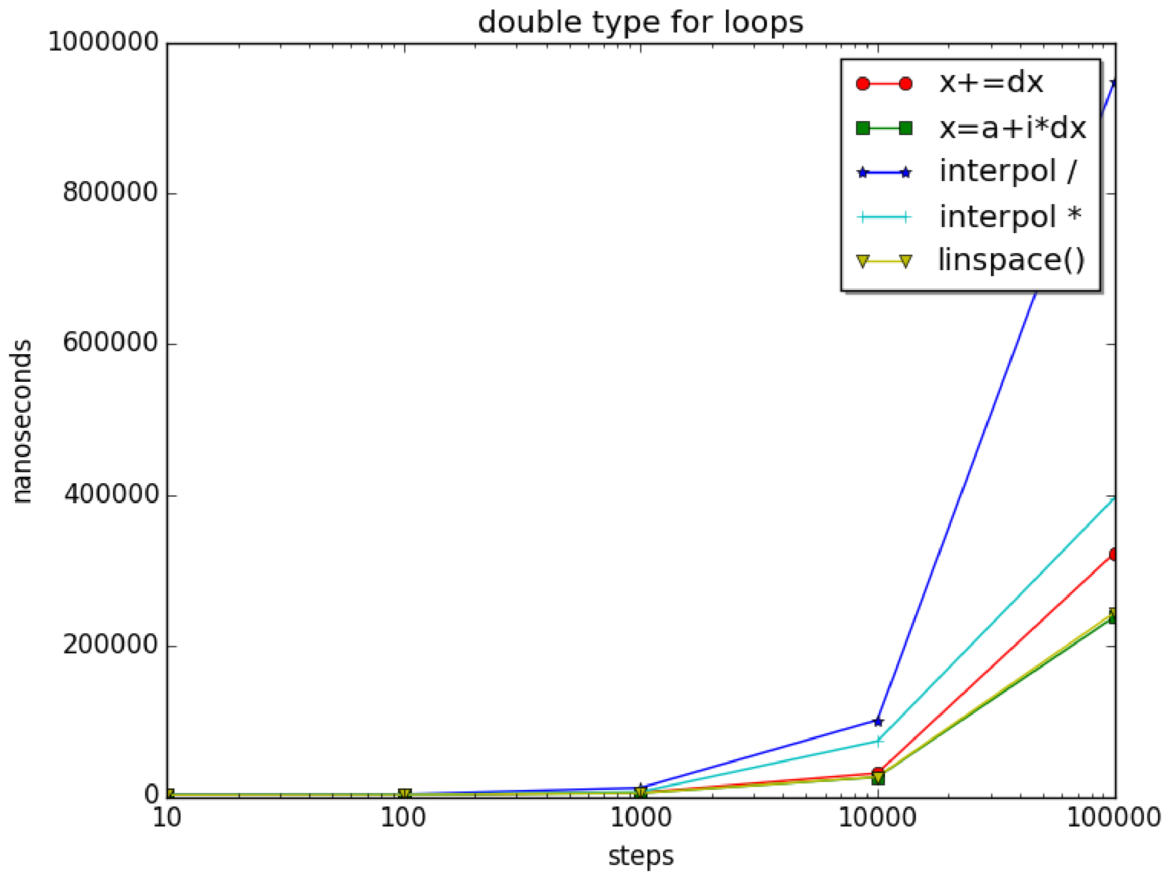

Figure 2: Runtime of handwritten loops and lazy generated ranges.