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

René Richter

namespace-cpp.de

2015-07-09

C++ usergroup DRESDEN

Outline

C++ usergroup DRESDEN

Motivation

> *I do not mind lying, but I hate inaccuracy.*
>
> — *Samuel Butler*

Recently on `http://isocpp.org` :

- Eric Niebler: Ranges v3.
- Ryan Haining: Python's builtins and itertools library in C++.
- Mikhail Semenov: Convenient Constructs For Stepping Through a Range of Values. (2015-02-17)

This talk is about

- floating-point numbers
- implementation techniques for ranges
- modern C++ features

This talk is about

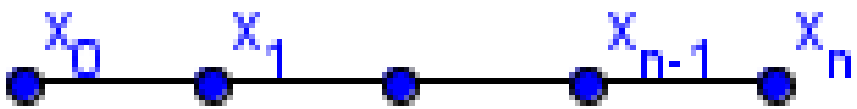
- floating-point numbers
- implementation techniques for ranges
- modern C++ features

Let's play a sort of "Bullshit bingo":

- Shout "14" when you recognize C++14 feature!

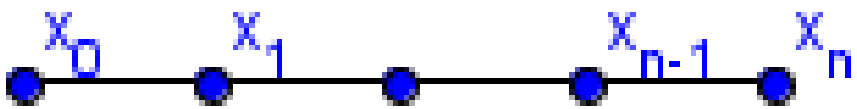
A simple task

Generate a loop for $n + 1$ equispaced values x_i in given $[a, b] \in \mathbb{R}$ with $i = 0...n$.



A simple task

Generate a loop for $n + 1$ equispaced values x_i in given $[a, b] \in \mathbb{R}$ with $i = 0...n$.



Suggestions?

```
... your code here ...
```

Naïve attempts

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

Naïve attempts

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

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

Naïve attempts

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

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

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

Naïve attempts

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

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

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

Rounding errors:

- `std::numeric_limits<double>::exact == false`
- `x + dx` a machine number?
- `x += dx` may not change `x`
- $\therefore$ no guarantee for $n + 1$ values

C++ usergroup DRESDEN

Scale transformation

mapping $i \in [0, n] \subset \mathbb{N}$ to equispaced $x \in [a, b] \subset \mathbb{R}$

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

C++ usergroup DRESDEN

Scale transformation

mapping $i \in [0, n] \subset \mathbb{N}$ to equispaced $x \in [a, b] \subset \mathbb{R}$

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

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

C++ usergroup DRESDEN

Introducing auto

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

Advantages:

- exactly $n+1$ values, works for $a \lessgtr b$
- no `operator<` required (`std::complex<T>`, ...)
- with/without a, b

Modern C++

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

How to compute range?

Existing libraries

Use Boost!

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

Existing libraries

Use Boost!

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

cppitertools (not guarantee for $n + 1$ values)

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

What, if . . . ?

closed and (half-)open intervals $x_{0|1} \ldots x_{n-1|n}$

What, if . . . ?

closed and (half-)open intervals $x_{0|1} \ldots x_{n-1|n}$

```
using namespace loop;

for (auto x : linspace(a, b, n))                    // [a,b]
for (auto x : linspace(a, b, n, bounds::leftopen))  // (a,b]
for (auto x : linspace(a, b, n, bounds::rightopen)) // [a,b)
for (auto x : linspace(a, b, n, bounds::open))      // (a,b)
```

create $n+1|n|n-1$ values *without changing other data points*

What, if . . . ?

closed and (half-)open intervals $x_{0|1} \ldots x_{n-1|n}$

```
using namespace loop;

for (auto x : linspace(a, b, n))                    // [a,b]
for (auto x : linspace(a, b, n, bounds::leftopen))  // (a,b]
for (auto x : linspace(a, b, n, bounds::rightopen)) // [a,b)
for (auto x : linspace(a, b, n, bounds::open))      // (a,b)
```

create $n+1|n|n-1$ values *without changing other data points*

Déja vu? NumPy/Matlab: always n values!

C++ usergroup DRESDEN

C++11 syntactic sugar

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

similar to

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

__range not necessarily a sequence container

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

Forward iterator has operators * ++ == !=

Generator template

```
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 }; }
```

Generator template

```
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 }; }

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

Iterator: mostly harmless

```
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_; }
    // operator++() : ++i_ ...
};
```

One interesting line in `iterator`:

```
Domain operator*() const { return a_ + scalar(i_) * dx_; }
```

Tricks with types in `LinearGenerator<Domain, N>`:

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

C++ usergroup DRESDEN

Hiding details

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

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

C++14

Automatic function return type deduction

```
template <typename Start, typename End, typename N>
auto linspace(Start a, End b, N n,
              boundary type = boundary::closed)
{
    // ...
    return detail::LinearGenerator<Domain, N>
                                    (a, b, n, first, last);
}
```

C++11

Extra default template parameter

```
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)
{ ... }
```

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>
{ ... }
```

C++ usergroup DRESDEN

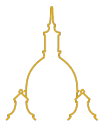
Almost always auto [Herb Sutter: GotW #94]

. . . doesn't mean always auto!

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

What's wrong with

```
auto operator*() const { /* ... */ }
auto operator++()      { /* ... */ }
auto operator++(int)   { /* ... */ }
```



auto strips reference and cv-qualifier!

```
// wrong
auto operator++()    { /* ... */ } // iterator copy!
auto operator++(int) { /* ... */ } // iterator
```

++++iter doesn't increment twice!

New syntax in C++14:

```
// ok
decltype(auto) operator++()    { /* ... */ } // iterator&
auto           operator++(int) { /* ... */ } // iterator
```

Flexible parameter types

Mixing types in $[a, b]$ possible:

```
using std::literals;

for (auto x : linspace(0 , 1.,      4)) // x is double
for (auto x : linspace(0., 4.+2.i,  4)) // x is complex<double>
for (auto x : linspace(0., 4.+2.if, 4)) // x is complex<float>!
```

C++ usergroup DRESDEN

Flexible parameter types

Mixing types in $[a, b]$ possible:

```
using std::literals;

for (auto x : linspace(0 , 1.,      4)) // x is double
for (auto x : linspace(0., 4.+2.i,  4)) // x is complex<double>
for (auto x : linspace(0., 4.+2.if, 4)) // x is complex<float>!
```

Sometimes not obvious to newbies:

- `4+2i`, `4+2.i` syntax error
- `double{} + std::complex<float>{}` $\mapsto$ `std::complex<float>` (narrowing `double` to `float`)

Implementation for integral ranges

```
for (auto i : range(5))                  // 0 1 2 3 4
for (auto i : range(5, 10))              // 5 6 7 8 9
for (auto i : range(0, 10, 2))           // 0 2 4 6 8
for (auto i : range(0, 10, 2, true))     // 0 2 4 6 8 10
for (auto i : range(10, 0, -2))          // 10 8 6 4 2
for (auto i : range(10, 0, -2, true))    // 10 8 6 4 2 0
for (auto i : countdown(5))              // 4 3 2 1 0
```

Mixing `signed`/`unsigned` integral types allowed

Beware expert-friendly C++:

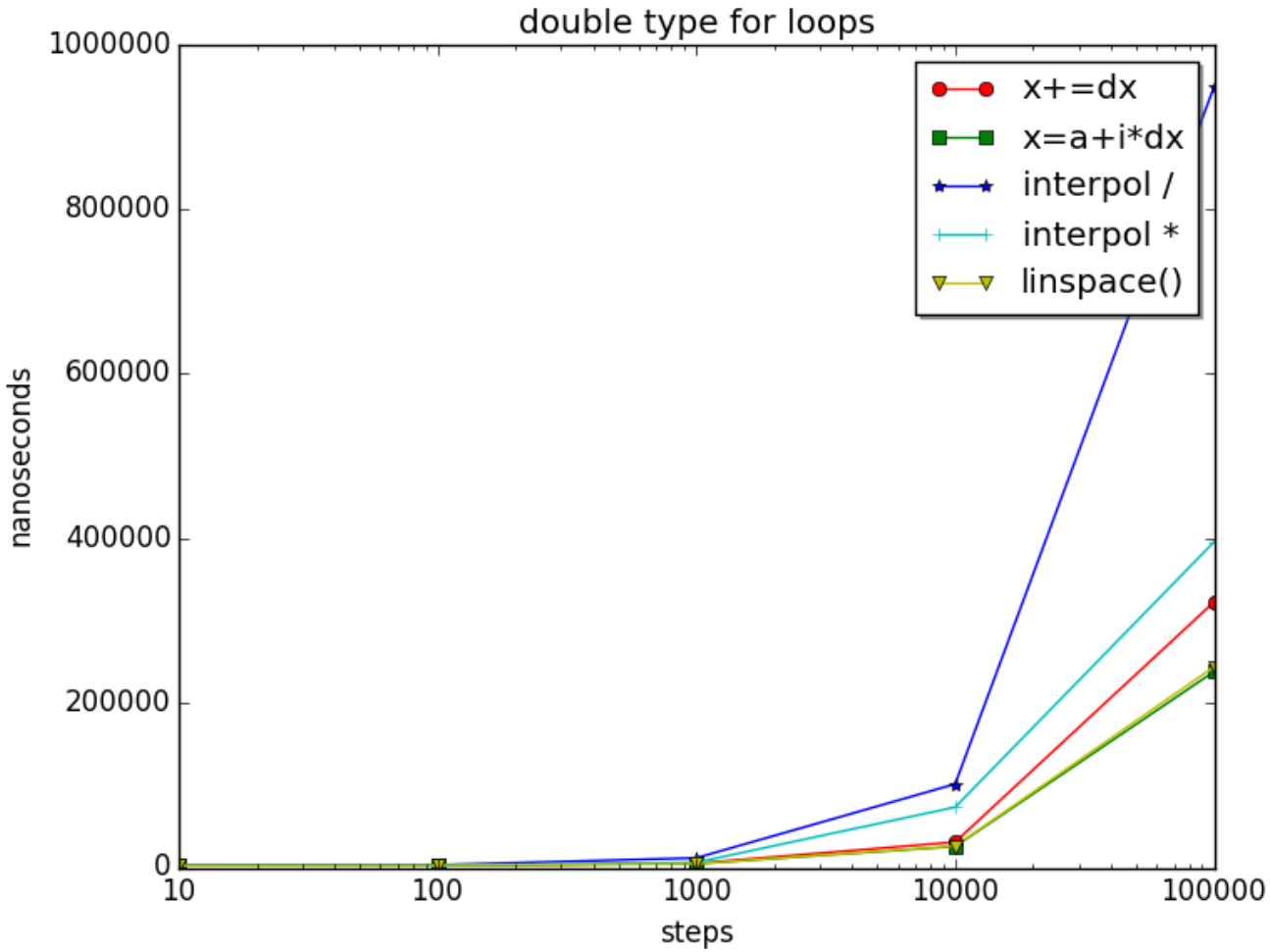
- `std::common_type_t<char, char>` $\mapsto$ `char`
- `decltype(char{} + char{})` $\mapsto$ `int`

C++ usergroup DRESDEN

... and other types: `start += incr`

```
for (auto s : generate("Ba"s, 5, "na")) // ...
```

Performance: Fast as handwritten



Links

- Numpy linspace(): `http://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html`
- MATLAB linspace(): `http://de.mathworks.com/help/matlab/ref/linspace.html`
- Boost irange(): `http://www.boost.org/doc/libs/1_57_0/libs/range/doc/html/range/reference/ranges/irange.html`
- Ryan Haining: cppitertools. `https://github.com/ryanhaining/cppitertools`
- Mikhail Semenov: Convenient Constructs For Stepping Through a Range. `http://www.codeproject.com/Articles/876156/Convenient-Constructs-For-Stepping-Through-a-Range`

Tools (C++14):

- Nick Athanasiou: Benchmarking in C++.
  `https://ngathanasiou.wordpress.com/2015/04/01/benchmarking-in-c/`

Implementation, source code:

- `https://bitbucket.org/dozric/looprange`