Iterators and ranges for numerical problems

Karsten Ahnert

Ambrosys GmbH, Potsdam

December 6, 2014

Outline

Motivation

Problem: Find the root of a function $\quad 0 = f(x)$

Motivation

Problem: Find the root of a function $\quad 0 = f(x)$

Newtons algorithm: $\quad x_{n+1} = x_n - f(x)/f'(x)$

Motivation

Problem: Find the root of a function $\quad 0 = f(x)$

Newtons algorithm: $\quad x_{n+1} = x_n - f(x)/f'(x)$

```
template< typename T , typename F , typename DF >
auto newton( T x , F f , DF df )
{
    while( std::abs( f(x) ) > 1.0e-12 )
        x = x - f(x) / df( x );
    return x;
};
// ...
double root = newton( x , f , df );
```

Motviation

Idea: Express the algorithm as a range

```
template< typename T , typename F , typename DF >
auto newton_range( T x , F f , DF df ) { ... };

// range is a lazy Newton algorithm
auto range = newton( x , f , df );

// perform the iteration
for( auto x : range )
{
    cout << x << endl;
}
cout << "Final value: " << range.value() << endl;
```

Motivation

Use ranges for three problems:

- Numerical Sequences
- Dynamical systems
- Convergence algorithms

The range is here a proxy for the algorithm!

The user can look inside the algorithm!
$\longrightarrow$ Debugging, logging, combine with other algorithms, $\dots$

Iterators – Overview

Operations

- Increment, dereference, assign, copy, decrement, …

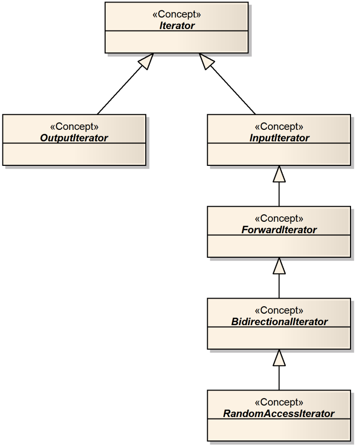
Use cases

- Traverse containers, IO, expressing algorithms

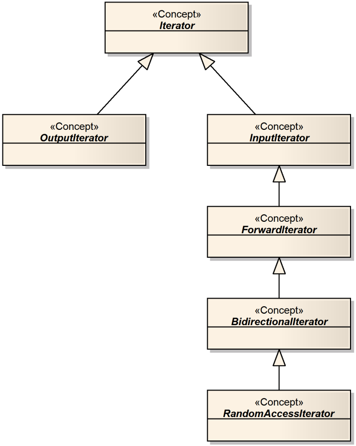
Special iterators

- `zip_iterator`, `transform_iterator`, ...

Iterator types – Concepts



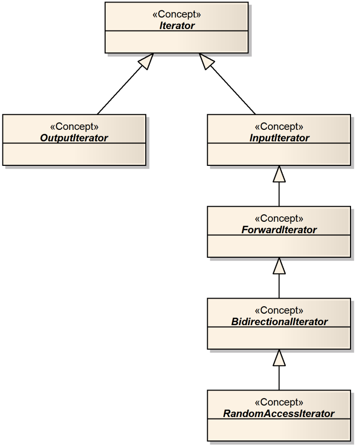
Iterator types – Concepts



Iterator

```
*i;
++i;
```

Iterator types – Concepts

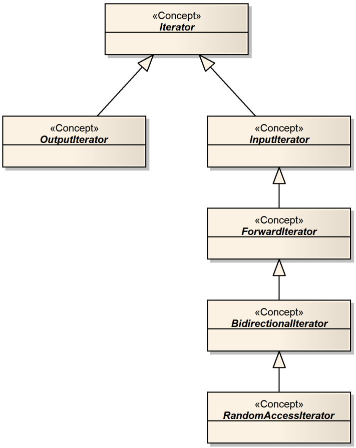


OutputIterator

```
*i = o;
*i++ = o;
i++;
++i;
```

`back_inserter`, `ostream_iterator`, …

Iterator types – Concepts



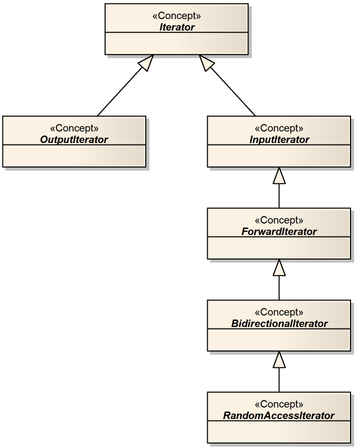
InputIterator

```
bool r = i != j;
val x = *i;
iterator j = ++i;
i++;
val x = *i++;
```

`istream_iterator`, `istreambuf_iterator`

But, if `i == j` then `++i != ++j`

Iterator types – Concepts

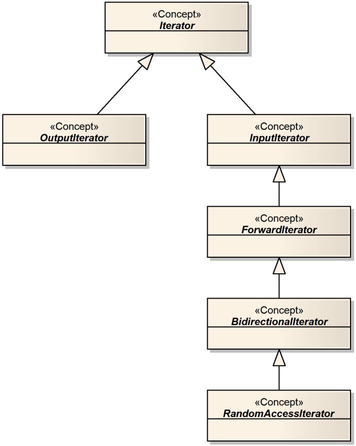


ForwardIterator

```
iterator j = i++;
```

But, if `i == j` then `++i == ++j`

Iterator types – Concepts

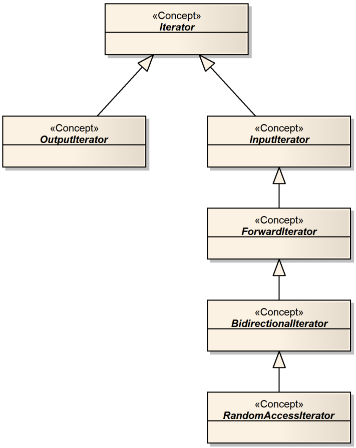


BidirectionalIterator

```
iterator j = --i;
iterator j = i--;
val x = *i--;
```

```
map< K , V >::iterator,
list< T >::iterator
```

Iterator types – Concepts



RandomAccessIterator

```
i += n;
i -= n;
val x = i[n];
long dist = i - j;
bool b = i < j;
```

`vector< T >::iterator`

Ranges

- Simplification and generalization of iterators
- Ranges in Boost:
  - Ranges are pairs of iterators.
  - Ranges can be adapted/decorated.

```
vector< double > values;

std::for_each( values.begin() , values.end() ,
    []( auto x ) { cout << x << endl; } );

boost::for_each( values ,
    []( auto x ) { cout << x << endl; } );
```

Ranges

Operations on ranges:

```
range_t r;
using iterator = boost::range_iterator< range_t >::type;
auto begin = boost::begin( r ); // begin iterator
auto end = boost::end( r );     // end iterator
```

Ranges

Operations on ranges:

```
range_t r;
using iterator = boost::range_iterator< range_t >::type;
auto begin = boost::begin( r ); // begin iterator
auto end = boost::end( r );      // end iterator
```

Implementing algorithms

```
template< typename Rng >
boost::for_each( Rng & r , auto op )
{
    std::for_each( boost::begin(r) , boost::end(r) , op );
}
```

Ranges for the C++ standard library

Eric Niebler: N4128: Ranges for the Standard Library.

- Ranges are based on iterators

Ranges for the C++ standard library

Eric Niebler: N4128: Ranges for the Standard Library.

- Ranges are based on iterators
- New concepts:
  - Iterable – Container, holding the elements
  - Range – Lightweight adapters (decorators)

Ranges for the C++ standard library

Eric Niebler: N4128: Ranges for the Standard Library.

- Ranges are based on iterators
- New concepts:
  - Iterable – Container, holding the elements
  - Range – Lightweight adapters (decorators)
- Variants of each algorithm
  - sort(rng.begin() , rng.end());
  - sort(rng);
  - rng | view::transform(op);
  - rng | cont::sort(op);

Three kind of ranges

- Pair of iterators
- Iterator and a count
- Iterator and a predicate

Three kind of ranges

- Pair of iterators
- Iterator and a count
- Iterator and a predicate

Algorithms are non-symmetric

```
template< typename Rng , typename Op >
void for_each( Rng & r , Op o )
{
    for_each( std::begin(r) , std::end(r) , o );
}

template< typename Iter , typename Sentinel , typename Op >
void for_each( Iter iter , Sentinel s , Op op )
{
    // ...
}
```

Numerical sequences

Numerical sequences

In numerical algorithms one needs often sequences of numerical values

- As part of an algorithm
- Reference data
- Test data

Example: Sampled function $f(x) = \sin(2x) + 0.1$

$$x_n = 0.1n \qquad \text{and} \qquad f_n = f(x_n) = \sin(2x_n) + 0.1$$

Numerical sequences

Range of integers

```
int n = 1024;
auto seq = boost::counting_range( 0 , n );
for( auto x : seq )
    cout << x << " ";
```

Output

```
0 1 2 3 ...
```

Numerical sequences

Sampling

```
using namespace boost::adaptors;

auto seq = boost::counting_range( 0 , n );
auto seq2 = seq | transformed(
    []( auto x ) { return 0.1 * x; } );

for( auto x : seq2 )
    cout << x << " ";
```

Output

```
0 0.1 0.2 0.3 ...
```

Numerical sequences

Sampled function

```
auto seq = boost::counting_range( 0 , n );
auto seq2 = seq | transformed(
    []( auto x ) { return 0.1 * x; } );
auto seq3 = seq2 | transformed(
    []( auto x ) { return sin( 2.0 * x ) + 0.1; } );

for( auto x : seq3 )
    cout << x << " ";
```

Output

```
0.1 0.298669 0.489418 0.664642 ...
```

Numerical sequences

Use a function

```
template< typename F >
auto sequence( int n , double sampling , F f )
{
    auto seq = boost::counting_range( 0 , n );
    return seq | transformed(
        [sampling,f]( auto x ) { return f(sampling*x); } );
}

auto seq = sequence(1024 ,0.1 ,
    []( auto x ) { return sin(2.0*x)+0.1;} );
for( auto x : seq )
    cout << x << " ";
```

Output

```
0.1 0.298669 0.489418 0.664642 ...
```

A teaser for the standard C++ ranges

Infinite sequences

```
using std::view;
template< typename F >
auto sequence( double sampling , F f )
{
    return iota(0) | transform(
        [sampling,f]( auto x ) { return f(sampling*x); } );
}

for( auto x : take( sequence( 0.1 , f ) , 10 ) )
    cout << x << endl;
```

Custom break conditions

```
auto seq = take_while( sequence( 0.1 , f ) ,
    []( auto x ) { return x < 1.0; } );

for( auto x : seq )
    cout << x << endl;
```

Comibining sequences – Zipped sequences

```
using std::view;
auto seq1 = sequence( 1024 , 0.01 , f1 );
auto seq2 = sequence( 1024 , 0.01 , f2 );
auto seq3 = zip( seq1 , seq2 );

for( auto x : seq3 )
    cout << std::get<0>(x) << " "<< std::get<1>(x) << "\n";
```

Combining sequences – Joined sequences

Append a sequence

```
auto seq3 = join( seq1 , seq2 );
```

Combining sequences – Joined sequences

Append a sequence

```
auto seq3 = join( seq1 , seq2 );
```

Cartesian product sequences – missing yet

Dynamical systems

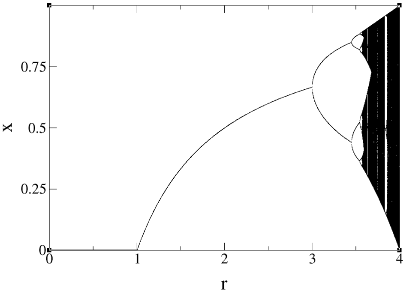
Dynamical systems – Maps

Map – time discrete dynamical system:

$$x_{n+1} = f(x_n)$$

Example: Logistic map

$$x_{n+1} = r\, x_n(1 - x_n)$$



Map range

Abstraction for $x_{n+1} = f(x_n)$

Two versions:

1. `map_range` - stop predicate
2. `counted_map_range` - iterates *n*-times

Map range

```
template< typename T , typename F , typename C >
class map_range
{
    struct iterator { ... };

  public:
    map_range( T value , F func , C condition )
    : m_value { std::move( value ) }
    , m_func { std::move( func ) }
    , m_condition( condition )
    {}

    iterator begin() const { return iterator( this ); }
    iterator end() const { return iterator( nullptr ); }
    T value() const { return m_value; }

private:
    mutable T m_value;
    mutable F m_func;
    C m_condition;
};
```

Map range

```
struct iterator {
    iterator( map_range const* _r ) : r( _r ) {}

    iterator& operator++() {
        r->m_value = r->m_func( r->m_value );
        if( r->m_condition( r->m_value ) ) {
            r = nullptr;
        }
        return *this;
    }

    T& operator*() const {
        return r->m_value; }
    bool operator==( iterator const& o ) const {
        return ( r == o.r ); }
    bool operator!=( iterator const& o ) const {
        return ! ( *this == o );
    }

    map_range const* r;
};
```

Factory functions

```
template< typename T , typename F , typename C >
auto make_map_range( T t , F f , C condition )
{
    return map_range< T , F , C >(
        std::move( t ) ,
        std::move( f ) ,
        std::move( condition ) );
}
```

```
template< typename T , typename F >
auto make_counted_map_range( T t , F f ,
    size_t max_iterations )
{
    return counted_map_range< T , F >(
        std::move( t ) ,
        std::move( f ) ,
        max_iterations );
}
```

Example - logistic map

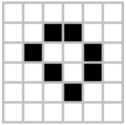
```
double r = 3.2;
auto f = [r]( auto x ) {return r * x * ( 1.0 - x ); };

auto range = make_counted_map_range( 0.5 , f , 1000 );
for( auto x : range )
{
    cout << x << endl;
}
```

Dynamical system – cellular automaton

Cellular automaton: Time-discrete and value-discrete

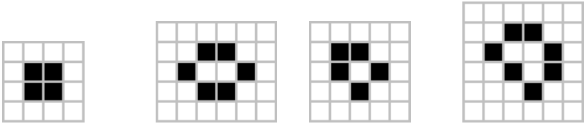
Conway's Game of Life

- Each cell has two states: *alive* or *dead*
- Transition rules
  - Less than two neighbors -> dead (under-population)
  - Two or three neighbors -> alive
  - More than three neighbors -> dead (over-population)
  - Dead cell with three neighbors -> alive (reproduction)

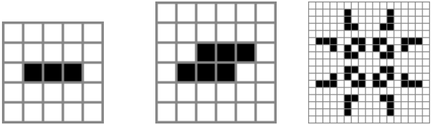


Game of life – patterns

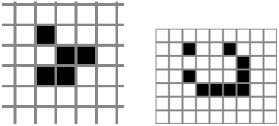
Still lifes:



Oscillators:



Spaceships:



Game of life can implement a Touring machine :)

Conway's game of life

```
using board = vector< vector< bool > >;

void show_board( board const& b ) { ... }
board next_board( board const& b ) { ... }

board b;
// initialize b

auto r = make_counted_map_range( b , next_board , 1000 );
for( auto b : r )
{
    show_board( b );
}
```

Map range for time discrete dynamical systems

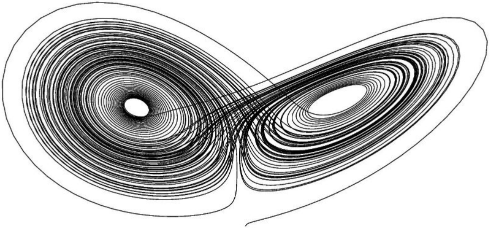
- The range is a proxy for the dynamical system
  - Customizable iteration body
  - Custom stopping criterions
  - N-times iteration
- Not optimized - a custom range could be optimized

Dynamical systems – ODEs

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x, t)$$

Example: Lorenz attractor

$$\dot{x} = \sigma(y - x) \quad , \quad \dot{y} = x(\rho - z) - y \quad , \quad \dot{z} = xy - \beta z$$



Numerical solution: $x(t + \Delta t) = F(x(t), f)$

Example – ODE solver

Example Boost.Odeint:

```
namespace odeint = boost::numeric::odeint;

auto lorenz = []( auto const& x , auto& dxdt , auto t )
{
    dxdt[0] = 10.0 * ( x[1] - x[0] );
    dxdt[1] = 28.0 * x[0] - x[1] - x[0] * x[2];
    dxdt[2] = -8.0 / 3.0 * x[2] + x[0] * x[1];
};

using state_type = std::array< double , 3 >;
odeint::runge_kutta4< state_type > stepper;

state_type x {{ 10.0 , 10.0 , 10.0 }};
double t = 0.0 , dt = 0.01;
for( ; t < 10.0 ; t += dt )
{
    stepper.do_step( lorenz , x , t , dt );
}
```

Example – ODE solver

```
auto make_ode_range = []( auto sys , auto stepper , auto x ,
    auto t0 , auto dt , auto t1 )
{
    auto solve = [sys,stepper,dt]( auto x ) mutable {
        stepper.do_step( sys , x.first , x.second , dt );
        x.second += dt;
        return x; };
    auto cond = [t1]( auto const& x ) { return x.second > t1; };
    auto range = make_map_range( std::make_pair(x,t0) , solver , cond );
    return range;
}
```

Example – ODE solver

```
auto make_ode_range = []( auto sys , auto stepper , auto x ,
    auto t0 , auto dt , auto t1 )
{
    auto solve = [sys,stepper,dt]( auto x ) mutable {
        stepper.do_step( sys , x.first , x.second , dt );
        x.second += dt;
        return x; };
    auto cond = [t1]( auto const& x ) { return x.second > t1; };
    auto range = make_map_range( std::make_pair(x,t0) , solver , cond );
    return range;
}
```

Can be used as

```
state_type x {{ 10.0 , 10.0 , 10.0 }};
stepper_type stepper;
auto range = make_ode_range( lorenz , stepper , x , 0.0 , 0.1 , 100.0 );

for( auto r : range )
    std::cout << r.second << " " << r.first[0] << " " << r.first[1] << "\n";
```

ODE ranges

Superior to integrate functions:

- Break conditions are easy
- ODE-Ranges can be used in a natural C++ way, `find`, `transform` , etc.

Map implementation has drawbacks -> custom implementation

- Better performance.
- Step size control – complicated iteration.
- Break condition can use the last two values.

Convergence methods

Newton method

Find the root: $0 = f(x)$

Newtons method

- Choose x_0
- Iterate $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

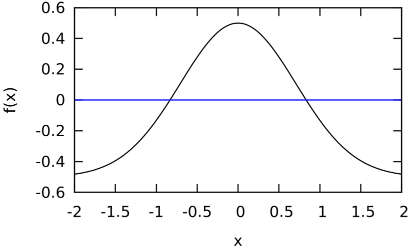
Newton method – Implementation

```
template< typename T, typename F, typename DF, typename C >
auto newton_range( T x , F f , DF df , C break_condtion )
{
    return make_map_range(
        x ,
        [f,df]( auto x ) { return x - f( x ) / df( x ); } ,
        break_condition );
}
```

Newton method – Example

Solve $\exp(-x^2) - 0.5 = 0$

```
auto f = []( auto x ) { return exp(-x*x) - 0.5; };
auto df = []( auto x ) { return -2.0*x * exp(-x*x); };
auto cond  = [f]( auto x ) {
    return std::abs(f(x)) < 1.0e-12; };

auto range = newton_range( 1.0 , f , df , cond );

for( auto r : range )
    std::cout << r << " : " << f( r ) << std::endl;
```



Similar problems

- Optimizations methods
  - Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS)
  - Simulated annealing
  - Genetic algorithms
  - Genetic programming
  - …
- Approximation of functions
- Iterative linear solvers
- Clustering methods (k-Means)
- …

Conclusion

- Iterators and ranges can be used for numerical problems
  - Numerical sequences, dynamical systems, optimization methods, ...
- Range is a proxy for the algorithm
- Custom range implementation needed for optimal performance

Outlook – open points

- Break condition on last two values, e.g. $|v_k - v_{k-1}| < \epsilon$
- Nested loops