Iterators and Ranges for numerical problems

Karsten Ahnert

Ambrosys GmbH, Potsdam

November 24, 2014

Outline

Introduction

Motivation

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

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

```
double x = 1.0;
auto f = []( auto x ) { return exp( -x*x ) - 0.5 ; };
auto df = []( auto x ) { return -2.0*x * exp( -x*x ); };
while( std::abs( f(x) ) > 1.0e-12 ) {
    cout << x << " " << f(x) << endl;
    x = x - f(x) / df( x );
}
```

Problems with this code?

Motivation

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

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

```
double x = 1.0;
auto f = []( auto x ) { return exp( -x*x ) - 0.5 ; };
auto df = []( auto x ) { return -2.0*x * exp( -x*x ); };
while( std::abs( f(x) ) > 1.0e-12 ) {
    cout << x << " " << f(x) << endl;
    x = x - f(x) / df( x );
}
```

Problems with this code?

- Not optimized.

Motivation

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

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

```
double x = 1.0;
auto f = []( auto x ) { return exp( -x*x ) - 0.5 ; };
auto df = []( auto x ) { return -2.0*x * exp( -x*x ); };
while( std::abs( f(x) ) > 1.0e-12 ) {
    cout << x << " " << f(x) << endl;
    x = x - f(x) / df( x );
}
```

Problems with this code?

- Not optimized. – Not today!
- Raw loops. Sean Parent - “No raw loops”

  Use algorithms and iterators!

Motivation

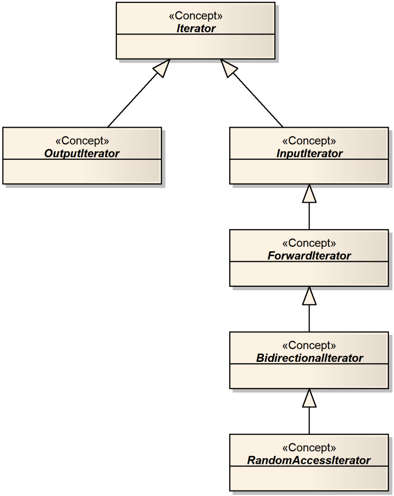
Try to use iterators in three problems:

- Numerical Sequences
- Dynamical systems
- Convergence algorithms

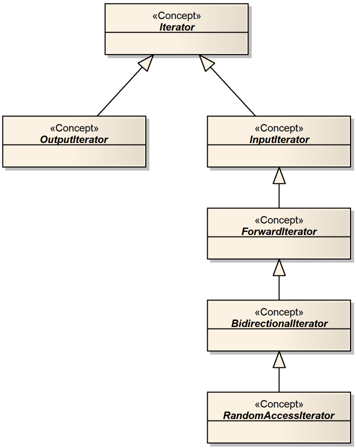
Iterators

- Traverse containers
- IO
- Expressing algorithms

Iterator types – Concepts



Iterator types – Concepts

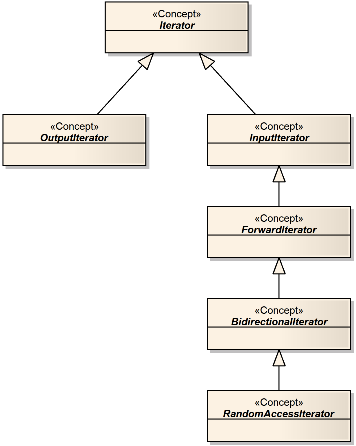


OutputIterator

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

Are special, `back_inserter`, `ostream_iterator`, ...

Iterator types – Concepts



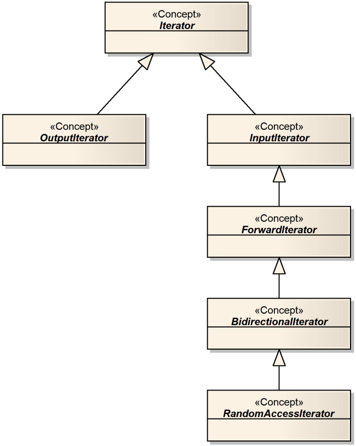
InputIterator a.k.a. Single-Pass Iterator

```
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

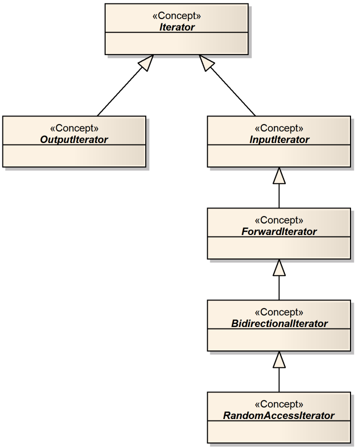


ForwardIterator

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

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

Iterator types – Concepts

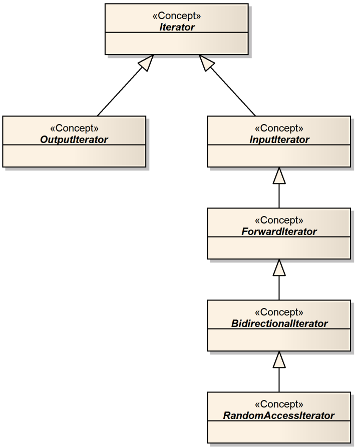


BidirectionalIterator

```
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

- Simplifying iterators
- Generalization of iterators
- Ranges in boost
  - Ranges are pairs of iterators.
  - Ranges can be decorated.
  - Memory overhead

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

Ranges for the native C++

- Introduces new concepts:
  - Iterable, Container, Sentinel
- Range is the main abstraction not the iterator
- Asymmetric algorithms - begin and end may have different types

Numerical sequences

Numerical sequences

In numerical algorithms one needs often sequences of numerical values

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

Numerical sequences

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

Output

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

Numerical sequences

```
using namespace boost::adaptors;

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

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

Output

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

Numerical sequences

```
auto seq = boost::counting_range( 0 , n );
auto seq2 = seq | transformed( []( auto x ) {
    return 0.01 * static_cast< double >( 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:

```
auto sequence( int n , double sampling , auto f ) {
    auto seq = boost::counting_range( 0 , n );
    auto seq2 = seq | boost::adaptors::transformed( [sampling,f]( auto x ) {
        return f( sampling * static_cast< double >( 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 ...
```

Numerical sequences

More complicated sequence

```
template< typename T >
auto sequence2( int n , double sampling , T f )
{
    auto seq = boost::counting_range( 0 , n );
    return seq | boost::adaptors::transformed( [sampling,f]( auto i ) {
            double x = sampling * static_cast< double >( i );
            return std::make_tuple( x , f( x ) ); } );

}

auto seq = sequence2( n , 0.1 , []( auto x ) { return sin( 2.0 * x ) + 0.1;
} );
for( auto x : zseq )
    cout << std::get< 0 >( x ) << " " << std::get< 1 >( x ) << "\n";
```

Output

0 0.1

0.1 0.298669

0.2 0.489418

...

Numerical ranges and Standard C++ Ranges

Advantages:

- The ranges are natively decorated, not the iterators.
- Support for infinite ranges.

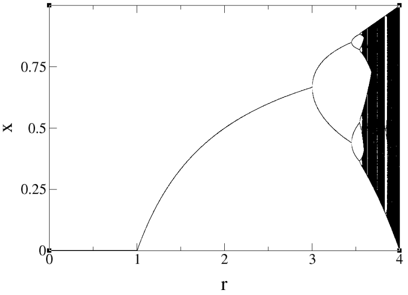
Dynamical systems

Dynamical systems – Maps

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

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

Models the SinglePassRange concept

Map range – implementation

```
template< typename T1 , ... > class map_range
{
    struct iterator { ... };
 public:
    // ...
    iterator begin() { return iterator( this ); }
    iterator end() { return iterator( nullptr ); }
    // ...
};
```

Range algorithms redirect to iterator algorithms:

```
template< typename R , typename F >
void for_each( R const& r , F f ) {
    std::for_each( r.begin() , e.end() , f );
}
```

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 ); }

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;
};
```

Counted map range

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

public:
    counted_map_range( T value , F func , size_t
        max_iterations )
    : m_current_iteration { 0 }
    , m_max_iterations { max_iterations }
    , m_value { std::move( value ) }
    , m_func { std::move( func ) }
    {}

    iterator begin() const { return iterator( this ); }
    iterator end( void ) { return iterator( nullptr ); }

private:
    mutable size_t m_current_iteration = 0;
    const size_t m_max_iterations;
    mutable T m_value;
    mutable F m_func;
};
```

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 l = [r]( auto x ) {return r * x * ( 1.0 - x ); };
auto range = make_counted_map_range( 0.5 , l , 1000 );
for( auto x : range ) { cout << x << endl; }
```

Dynamical system – cellular automaton

A cellular automaton is a time discrete and value discrete dynamical system

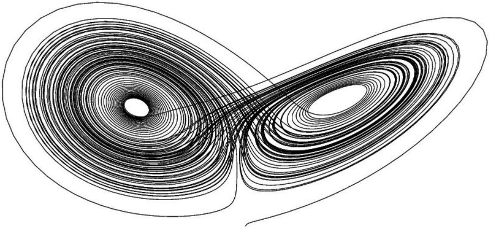
a

Dynamical systems – ODEs

$$\frac{dx}{dt} = f(x, t)$$

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: $x(t + \Delta t) = F(x(t))$

Example – ODE solver

Solve $\dot{x} = f(x, t)$, Solver $F : x(t + \delta t) = F(x(t))$
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;
stepper.do_step( lorenz , x , t , dt );
t += dt;
```

Integrate functions:

```
auto obs = []( auto x , auto t ) { std::cout << t << " " << x[0] << "\n"; };
odeint::integrate_const( stepper , lorenz , x , 0.0 , 10.0 , dt , obs );
```

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

Example – Newton method

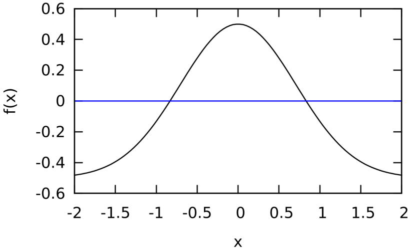
```
auto newton_range(
    auto x , auto f , auto df ,
    auto break_condition )
{
    return make_map_range(
        x ,
        [f,df]( auto x ) { return x - f( x ) / df( x ); } ,
        break_condition );
}
```

Example – Newton method

```
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
  - Genetic algorithms, genetic programming, simulated annealing, ...
- Approximation of functions

Conclusion

- Iterators and ranges can be used for numerical problems
- For some problems they provide the correct abstraction
- If it can be applied to all problems is not clear
- The new range library for Standard C++ might ease a lot of things

References

- odeint
- github talk
- github map range

Backup

Example – basic use

```
for( auto iter = values.begin() ;
     iter != values.end() ;
     ++iter )
{
    cout << *iter << endl;
}
```

Example – basic use

```
for( auto iter = values.begin() ;
     iter != values.end() ;
     ++iter )
{
    cout << *iter << endl;
}
```

C++11 - use range based for

```
for( auto v : values )
{
    cout << v << endl;
}
```

Example – Container traversal

```
list< double > values;
list< double > values2( values.size() );
```

Can be used in

```
transform( values.begin() , values.end() ,
           values2.begin() ,
           []( double x ) {
               return x * 2.0; } );
```

Example – Container traversal

```
vector< double > values;
vector< double > values2( values.size() );
```

Can be used in

```
transform( values.begin() , values.end() ,
           values2.begin() ,
           []( double x ) {
               return x * 2.0; } );
```

Examples – IO

Input

```
vector< double > values;
copy_if( istream_iterator< double >( cout ) ,
         istream_iterator< double >() ,
         back_inserter( values ) ,
         []( double x ) { return x > 0.0; } );
```

Output

```
vector< double > values;
// fill values
copy_if( values.begin() , values.end() ,
         ostream_iterator< double >( std::cout , "\n" ) ,
         []( double x ) { return x > 0.0; } );
```

Algorithms

I all_of
I any_of
I none_of
I for_each
I count
I count_if
I mismatch
I equal
I find
I find_if
I find_if_not
F find_end
I,F find_first_if
F adjacent_find
F search
F search_n
I,O copy
I,O copy_if
I,O copy_n
B,O copy_backward
I,O move
B,O move_backward
F fill
F fill_n
I,O transform
F generate
I generate_n

F remove
F remove_if
I,O remove_copy
I,O
remove_copy_if
F replace
F replace_if
I,O replace_copy
I,O
replace_copy_if
F swap_ranges
F iter_swap
B reverse
B,O reverse_copy
F rotate
F,O rotate_copy
R random_shuffle
R shuffle
F unique
I,O unique_copy

I is_partitioned
F,B partition
I,O partition_copy
B stable_partition
F partition_point
F is_sorted
F is_sorted_until
R sort
R partial_sort
I,R partial_sort_copy
R stable_sort
R nth_element

F lower_bound
F upper_bound
F binary_search
F equal_range
I,O merge
B inplace_merge
I includes
I,O set_difference
I,O set_intersection
I,O
set_symmetric_difference
I,O set_unition

R is_heap
R is_heap_until
R make_heap
R push_heap
R pop_heap
R sort_heap

F max_element
F min_element
F minmax_element
I lexicographical_compare
F is_permutation
B next_permutation
B prev_permutation

F iota
I accumulate
I inner_product
I,O adjacent_difference
I,O partial_sum

Examples – generalized iota

Generalized Iota:

```
size_t n = 10;
auto iota = make_counted_map_range( 1 , []( auto x ) {
    return x * 2; } , 10 );

std::vector< int > values;
boost::copy( iota_range , std::back_inserter( values ) );
for( auto i : values ) { cout << i << endl; }
```

Examples – generalized iota

Problem: We can not easily generate a square iota:
$1, 4, 9, 16, 25, 36, ...$

Examples – generalized iota

Problem: We can not easily generate a square iota:
$1, 4, 9, 16, 25, 36, \ldots$
Introduce a projected map range.

```
auto iota_range = make_projected_counted_map_range(
    1
    , []( auto x ) { return x+1 ; }
    , 11
    , []( auto x ) { return x*x; }
    );
for( auto i : iota_range ) { std::cout << i << std::endl; }
```

Map range - applications

- Generalized iota
- Ordinary differential equations
- Maps (dynamical maps)
- Functional random number generators
- Opmization methods
  - Genetic algorithms, Simulated annealing, . . .

Map range - applications

- Generalized iota
- Ordinary differential equations
- Maps (dynamical maps)
- Converging algorithms
- Functional random number generators

Iterators for GPUs algorithms

High-level libraries for GPUs

- Thrust
- VexCL
- Boost.Compute
- ViennaCL
- Cuda-MTL

Thrust

Thrust is STL-like library for Cuda – Based on iterators.

```
thrust::host_vector<int> h_vec( 1024 );
std::generate(h_vec.begin(), h_vec.end(), rand);

thrust::device_vector<int> d_vec = h_vec;
thrust::sort(d_vec.begin(), d_vec.end());

thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

Iterators in Thrust

- `device_vector< T >::iterator`
- `host_vector< T >::iterator`
- Special (fancy) iterator
  - `zip_iterator`
  - `transform_iterator`
  - `permutation_iterator`
  - `constant_iterator`, `counting_iterator`, `discard_iterator`, `reverse_iterator`

Custom algorithms

Special iterators for Thrust

Calculate the norm of a vector

$$||x|| = \sum_{i=1}^{N} x_i^2$$

```
thrust::device_vector< double > x;
// fill x

double n = thrust::reduce(x.begin(), x.end(), 0.0); // ?
```

Special iterators for Thrust

Calculate the norm of a vector

$$||x|| = \sum_{i=1}^{N} x_i^2$$

```
thrust::device_vector< double > x;
// fill x

auto op = []( auto x , auto y ) { return x + y*y; };
double n = thrust::reduce(x.begin() ,x.end() ,0.0 ,op); //
    ?
```

Special iterators for Thrust

Calculate the norm of a vector

$$||x|| = \sum_{i=1}^{N} x_i^2$$

```
thrust::device_vector< double > x;
// fill x

op = []( auto x ) { return x*x; };
double n = thrust::reduce(
    thrust::make_transform_iterator(x.begin(), op) ,
    thrust::make_transform_iterator(x.end(), op), 0.0);
    // correct
```

SAXPY

$s = ax + y \qquad s, x, y \in \mathbb{R}^N,\ a \in \mathbb{R}$

Special problems - and solutions

Bucket sort

Solving an ensemble of low-dimensional ODEs

Lorenz example and ODEs