Iterators and Ranges for numerical problems

Karsten Ahnert

Ambrosys GmbH, Potsdam

November 17, 2014

Outline

Introduction

Iterators

Unique way to traverse containers
Unique way to apply iterative IO
Unique way of expressing algorithms

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; } );
```

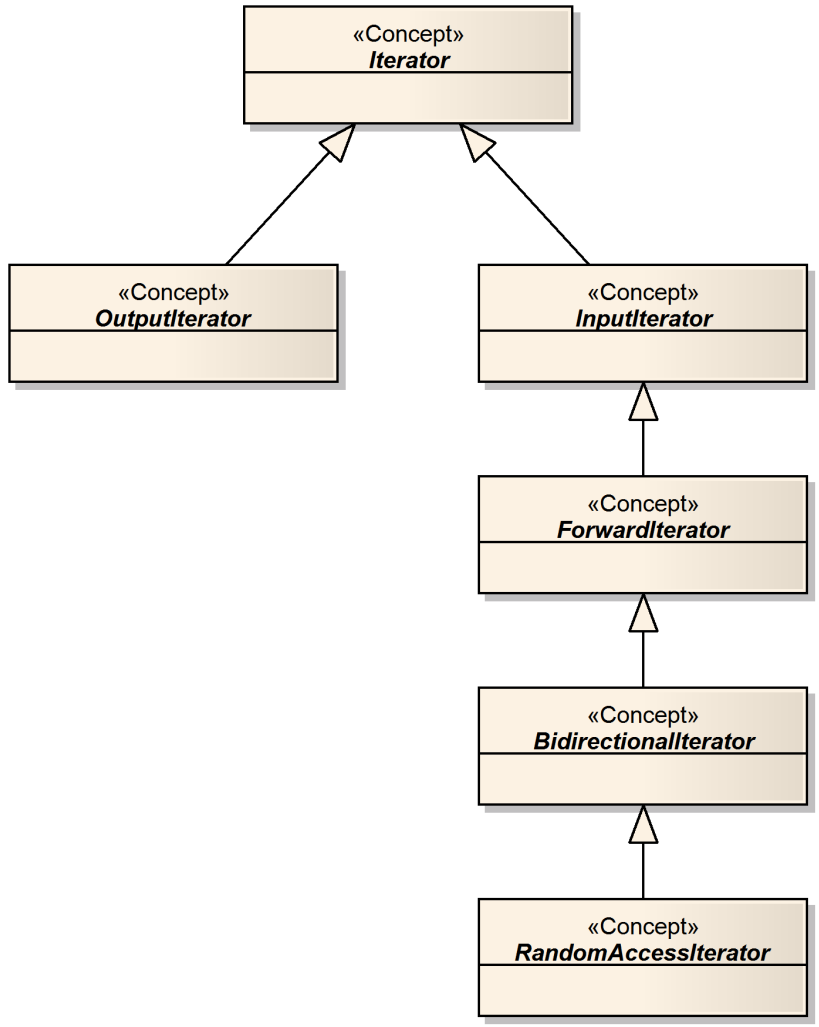
Output

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

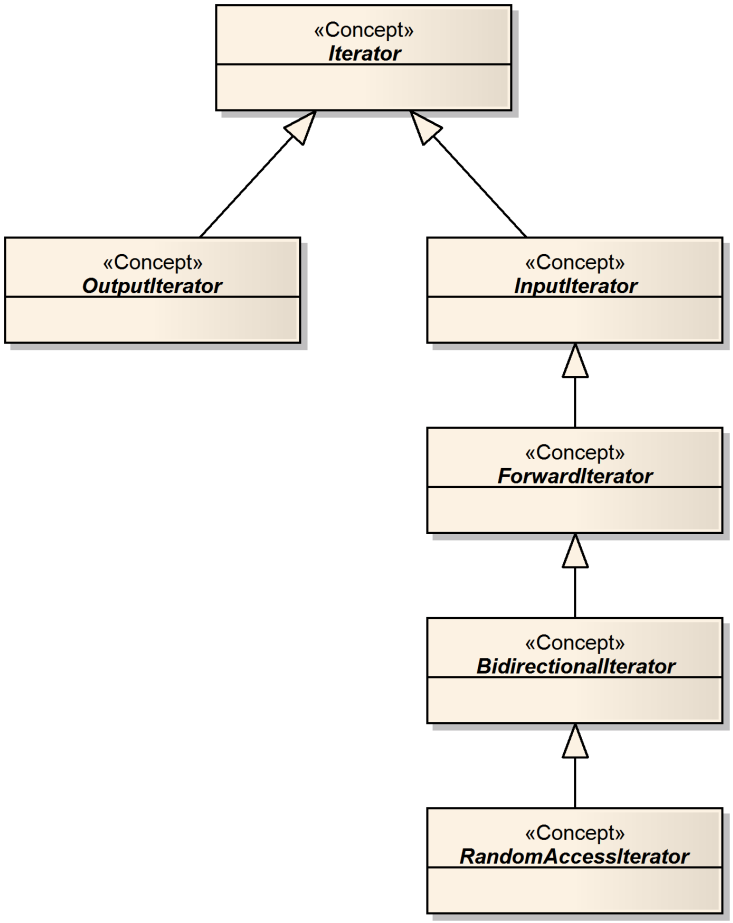
Examples – Combine algorithms

Find a nice real life example.

Iterator types – Concepts



Iterator types – Concepts

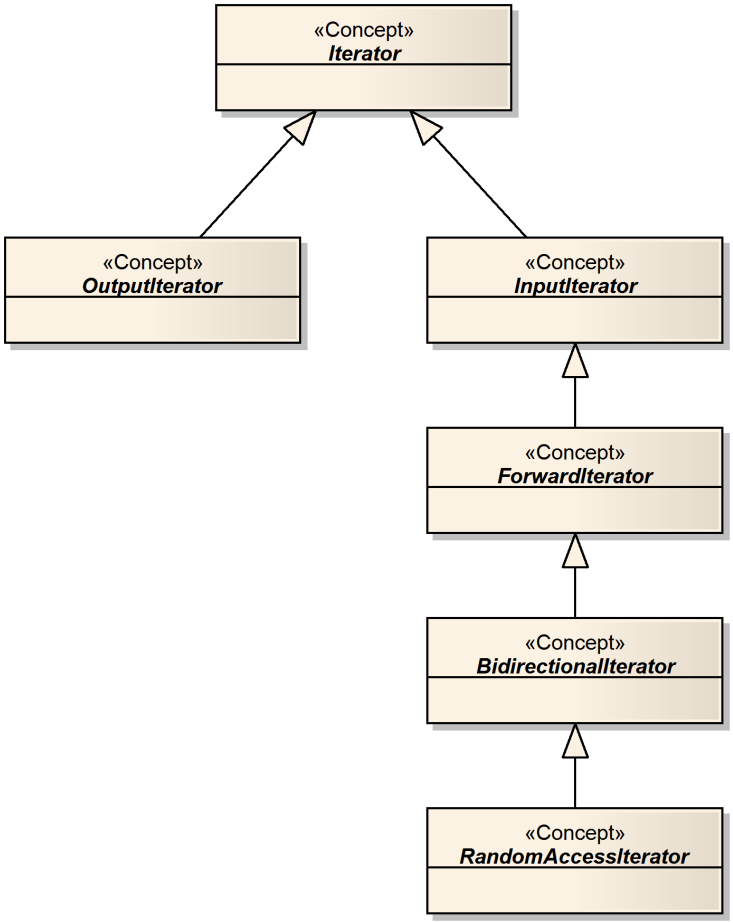


OutputIterator

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

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

Iterator types – Concepts



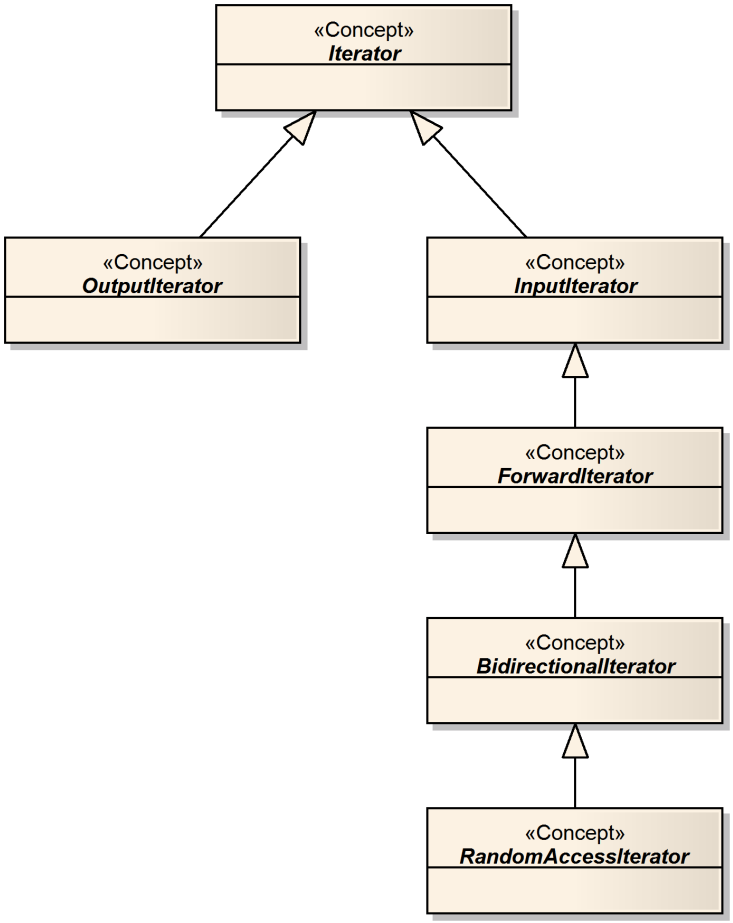
InputIterator a.k.a. Single-Pass Iterator

```
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

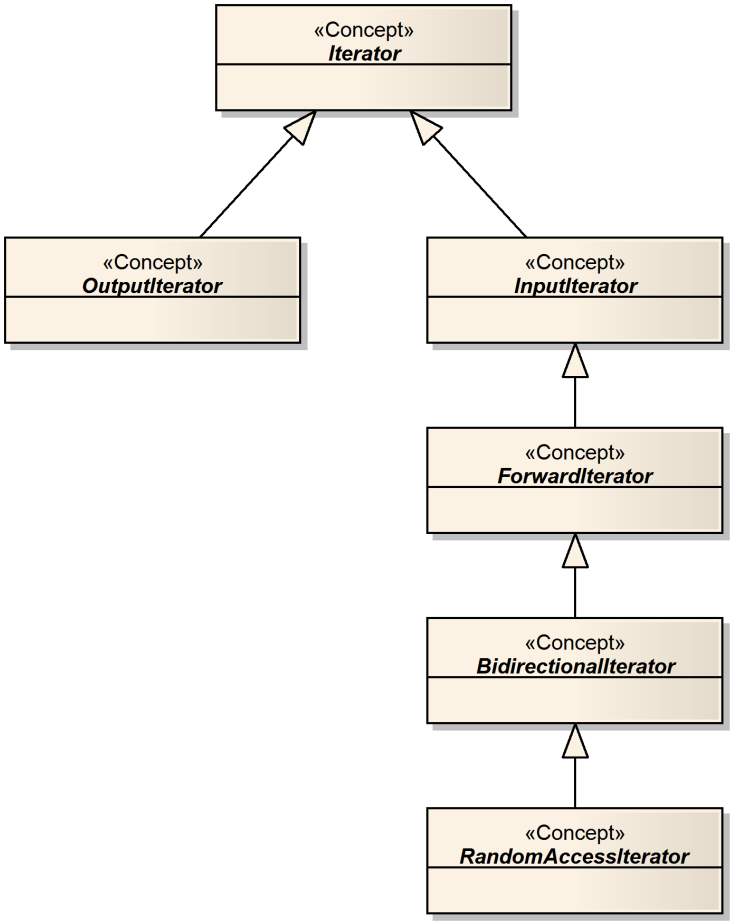


ForwardIterator

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

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

Iterator types – Concepts

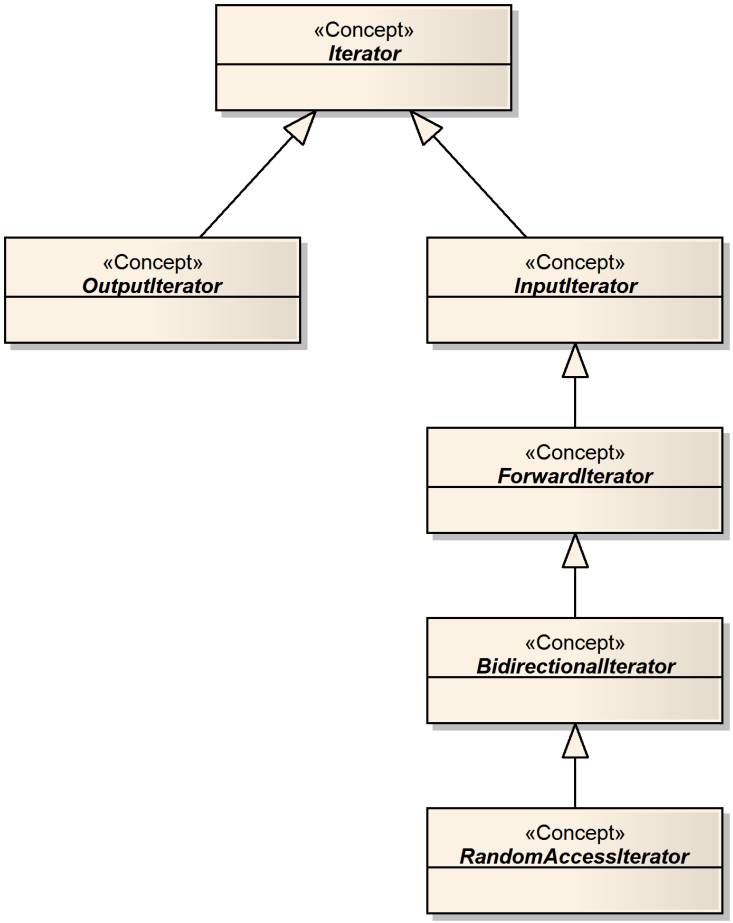


BidirectionalIterator

```
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`

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

is_partitioned
partition
partition_copy
stable_partition
partition_point
is_sorted
is_sorted_until
sort
partial_sort
partial_sort_copy
stable_sort
nth_element
lower_bound
upper_bound
binary_search
equal_range
merge
inplace_merge
includes
set_difference
set_intersection
set_symmetric_difference
set_union

is_heap
is_heap_until
make_heap
push_heap
pop_heap
sort_heap
max
max_element
min
min_element
minmax
minmax_element
lexicographical_compare
is_permutation
next_permutation
prev_permutation
iota
accumulate
inner_product
adjacent_difference
partial_sum

Ranges

Simplifying iterators
Generalization of iterators
First defined in Boost
Soon in the standard library?

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

Ranges – more examples from boost

Filters
complicated algorithms

Ranges in Boost

Ranges are pairs of iterators.
Memory overhead
Filters grow exponential in size

Ranges for the native C++

Introduces new concepts:
Iterable, Container, Sentinel
The range is the main abstraction not the iterator
It holds all informations
Concepts, asymmetric algorithms, sentinels have their own type.

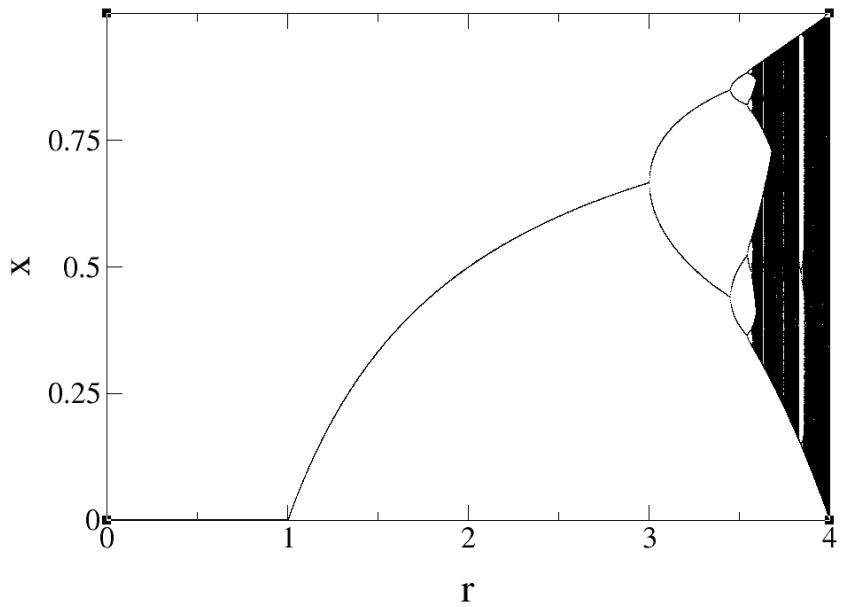
Iterators and ranges for dynamical systems

Dynamical systems – Maps

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

Example: Logistic map

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

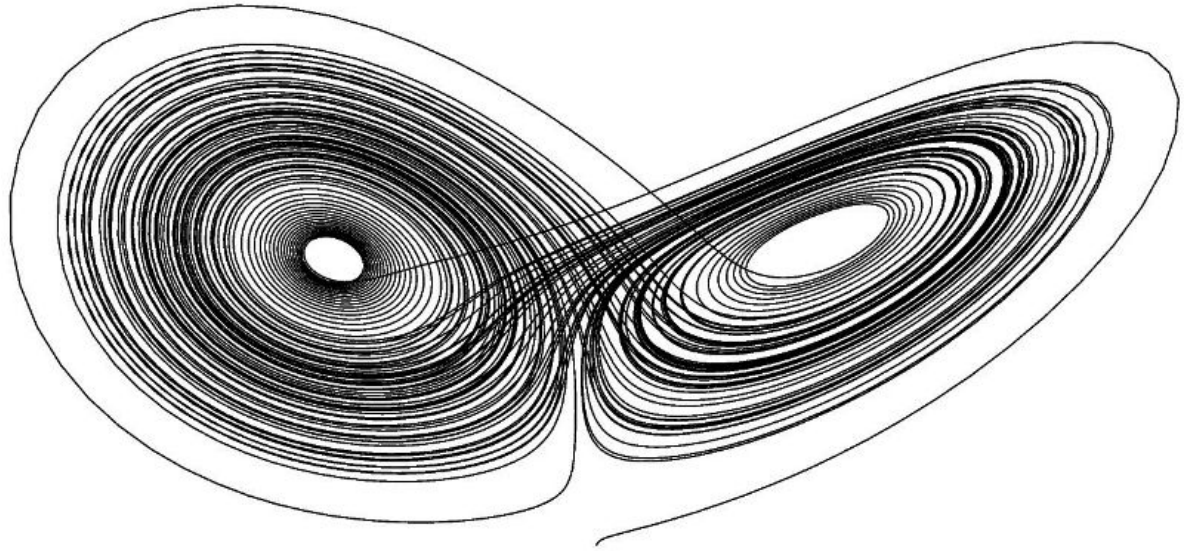


Dynamical systems – ODEs

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

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:

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

Newton method

Find the root

$$0 = f(x)$$

Newtons method

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

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 models the SinglePassRange concept

Map range - applications

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

Map range – implementation

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

Range algorithm:

```
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;
};
```

First examples

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; }
```

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; }
```

Problems: Projection

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

Iterators for GPUs algorithms

High-level libraries for GPUs

1. Thrust
2. VexCL
3. Boost.Compute
4. ViennaCL
5. Cuda-MTL

Thrust

STL-like library for Cuda
Design is based on iterators

Iterators in Thrust

`device_vector::iterator`

`host_vector::iterator`

special iterators

Algorithms

Implementation details of Thrust iterators

Special iterators for Thrust

zip iterator
transform iterator

Special problems - and solutions

Norm

Special problems - and solutions

Bucket sort

Solving an ensemble of low-dimensional ODEs

Lorenz example and ODEs

Conclusion

Outlook

References