Parametric Search Framework: reference manual

René van Oostrum and Remco C. Veltkamp

June 19, 2003

This document is the reference manual of the CGAL extension package for parametric search. It is not a full description of all classes of the framework and of all their functions. Instead, it only describes classes and functions that are meant to be used in or called from user-code.

This document is neither the right place to look for an overview of the framework, nor the place to look for information on how to use the framework; see the tutorial instead.

Class Bitonic_sorter< Traits, BidirectionalIter >

Definition

Class that can be used for sorting-based parametric search. After creating an object of this class, the parametric search is started by calling the method *run* of the Scheduler.

Is Model for the Concept

Process

Creation

The constructor takes three arguments: a pointer to the parent process, if any, or 0 otherwise (in typical use-cases, the first argument will be 0), and two iterators indicating a sequence of polynomials to be sorted:

Bitonic_sorter(process_base_t parent, BidirectionalIter first, BidirectionalIter last);*

A *Bitonic_sorter* object must be created on the heap, i.e., with *new*.

Concept Comparison

Definition

A Comparison is an object of a class derived from *comparison_base_t*, which is a typedef defined in a traits class, and defaults to *Comparison_base*. It evaluates the sign of a (problem-specific) polynomial at the (unknown) value λ^*, which is the solution to the parametric search problem. The sign of the polynomial at λ^* depends on the location of λ^* among the roots of the polynomial. This location is determined by the scheduler; once it is known, the sign of the polynomial can be computed efficiently by the Comparison object.

Creation

Comparisons must be created on the free store (i.e., with *new*). They may not be created on the stack (i.e., as local, global or static variables). Comparisons may not be deleted by user code (the *Scheduler* manages the life-time of all Comparisons).

The constructor of a Comparison has at least one argument *parent*, which is a pointer to the parent Process that created the Comparison. The type of this pointer is *process_base_t** (which is a typedef defined in a traits class, and defaults to *Process_base*. The Comparison must initialize its own base class with parent as argument:

comparison_base_t(parent)

The constructor must compute the roots of the polynomial associated with the comparison. Only for those roots that lie in the interval of known bounds on the solution to the parametric search (which are maintained by the *Scheduler*), an object of type *root_t* must be created, and passed to the Scheduler with the member function *spawn* (derived from *comparison_base_t*):

spawn(new root_t(this, value));

Here, *root_t* is a typdef defined in a traits class, and it defaults to *Root*. The parameter *value* is the numerical value of the root. Its type is *number_t*, which is a typedef defined in a traits class.

If no roots are spawned (i.e., if all of them lie outside the interval of bounds on the parametric search solution), the constructor must notify this to the scheduler by calling the following a member function (derived from *comparison_base_t*):

nothing_spawned()

Operations

The following virtual function of *comparison_base_t* must be overridden:

virtual void deduce(
const number_t& lower_bound,
const number_t& upper_bound)

This function is called by the *Scheduler*. It is guaranteed that when the *Scheduler* calls this function, the location of λ^* (the unknown solution to the parametric search problem) among the roots of the polynomial, associated with the Comparison, is known. This means that *deduce* can and must resolve the comparison

by deducing it efficiently, by evaluating the sign of the polynomial inbetween two consecutive roots r_i and r_{i+1} with $r_i < \lambda^* < r_{i+1}$.

Comparison classes that are to be used by the sorting classes *Bitonic_sorter* or *Quick_sorter*, provided with the parametric search framework for sorting-based parametric search, must provide an additional (static) member function *lazy_create_comparison*:

static comparison_base_t lazy_create_comparison*
(process_base_t parent, bool& comp_result,*
const polynomial_t& first, const polynomial_t& second)

The type *polynomial_t* is a typedef defined in a traits class. It denotes the type of the polynomials that can be compared by a Comparison. The actual type is problem-specific.

Before creating a comparison (which is expensive, since it involves dynamic allocation of memory) this member function must check whether the outcome of the comparison can already be deduced efficiently. This is the case when none of the roots of the difference polynomial of *first* and *second* lie within the interval of known bounds on the solution to the parametric search (these bounds are maintained by the *Scheduler*). If the outcome of the comparison can be deduced, the outcome must be stored in *comp_result*, and *lazy_create_comparison* must return a null pointer; no Comparison object may be created in this case. Otherwise, if the outcome of the comparison cannot be deduced efficiently, *lazy_create_comparison* must create a Comparison object (i.e., an object of its own class), and return a pointer to the newly created Comparison object. This Comparison object will then later, when the *Scheduler* has determined the location of λ^* among the roots of its polynomial, be told by the *Scheduler* to execute its member function *deduce*. To make sure that no unnecessary Comparison objects are created, the constructor of a Comparison class that is to be used in sorting-based parametric search must be made *private*. In this way, only the static member function *lazy_create_comparison* can create comparisons.

Has Models

Comparison_base

Class Comparison_base< Traits >

Definition

Abstract base class that provides the common part of the functionality that all Comparisons used in the parametric search framework share. Derived classes must be created on the free store.

Is Model for the Concept

Comparison

Creation

The constructor has one argument, *parent*, of type *Traits::process_base_t**; this argument points to the Process that created the Comparison.

Comparison_base(Traits::process_base_t parent)*

Note that objects of derived classes of *Comparison_base* may only be created on the heap (i.e., by using *new*). In user code, there must be no calls to *delete* on pointers to objects of derived classes of *Comparison_base*, as this is taken care of automatically by the *Scheduler*.

Operations

The destructor notifies the calling Process that the Comparison has completed its execution; derived classes can and must rely on this to happen automatically.

virtual ~Comparison_base() = 0;

The following function must be overridden by derived classes:

virtual void deduce
(const Traits::number_t& lower_bound,
const Traits::number_t& upper_bound) = 0

The semantics of this function are described in concept Comparison.

If a derived class of *Comparison_base* creates Roots (which may only be created in its constructor), these must be created on the heap (i.e., with *new*), and the pointer returned by new must be passed to the member function *spawn* of *Comparison_base*:

void spawn(Spawnable< Traits > child)*

Note: this function does not appear in the class definition of *Comparison_base*; they are inherited from *Spawner*, which is a base class of *Process_base* and *Comparison_base*. The class *Spawnable< Traits >* is a common base class of *Process_base*, *Comparison_base*, and *Root*.

If a derived class of *Comparison_base* doesn't create roots in its constructor, it must call the following function:

void nothing_spawned()

Concept Decision Problem Solver

Definition

A *Decision Problem Solver* is an object of a class that can, for a given numerical value, determine whether that value is less than, equal to, or greater than λ^*, the solution to the parametric search problem.

Types

the following type must be provided: *enum decision_t { less, equal, greater };*

Operations

The following member function must be provided: *decision_t decide(const number_t& arg);*

Has Models

Solver_base

See Also

Scheduler, Traits

Function frechet_distance

Definition

The function *frechet_distance* computes the Fréchet distance between two polygonal curves.

```
#include <frechet_distance.h>

template< typename Traits, typename InputIter >
void frechet_distance( const InputIter& first_polyline_first,
                       const InputIter& first_polyline_last,
                       const InputIter& second_polyline_first,
                       const InputIter& second_polyline_last )
```

This funtion is used as follows:

```
// create two polylines with 80 vertices each, and with all vertices
// in the square [0,1] x [0,1]

Polyline p( random_polyline( 80, 0, 1, 0, 1 ) );
Polyline q( random_polyline( 80, 0, 1, 0, 1 ) );

// compute the frechet distance between the polylines
frechet_distance< Frechet_traits >( p.begin(), p.end(),
                                    q.begin(), q.end() );
```

Requirements

1. *InputIter::value_type* is equivalent to *Traits::Point*.
2. *Traits* is a model for the concept Parametric Search Traits, as specified for sorting-based parametric search. Additionally, it must define the type *Point*.

Precondition: Both polygonal curves must be defined by at least two points, and neither of them may contain zero-length segments.

Postcondition: After the call to *frechet_distance*, the *Scheduler* maintains the lowerbound and upperbound on the Fréchet distance between the polygonal curves.

See Also

Scheduler

Concept Parametric Search Traits

Definition

A Parametric Search Traits class is a struct or class that contains type definitions, and that is used as a template argument for other classes (*Process_base*, *Comparison_base*, *Root*, and *Scheduler*, amongst others) . These other classes can than, through the Parametric Search Traits class, refer to the type definitions without actually knowing to what real type these type definitions resolve.

Types

The following types must be provided as typedefs for real types:

number_t
scheduler_t
process_base_t
comparison_base_t
root_t
decision_solver_t

Here, *number_t* refers to the number type that is used to represent and compute λ^*; *scheduler_t* denotes the type of the scheduler (at the moment, the only valid definition of *scheduler_t* is *Scheduler*< *Traits* >, provided with the framework); *process_base* specifies the base class for Process object (normally *Process_base*< *Traits* >, although users may develop their own base class according to the Process concept); *comparison_base_t* and *root_t* specify the classes that are to be used as base class for Comparison objects and Root objects, respectively (these can also be either classes provided with the framework, or user-defined classes); *decision_solver_t* indicates the concrete type of the Decision Problem Solver class(since it is problem-specific, no default implementation for this class is provided with the framework).

Apart from these required type definitions, users may also specify their own problem-specific types in the traits class.

If the framework is to be uses for sorting-based parametric search, using either *Bitonic_sorter* or *Quick_sorter*, then additionally the following types must be provided as typedefs for real types:

polynomial_t
comparison_t

Since both sorting algorithms solve the parametric sort problem by sorting polynomials (which requires the comparison of polynomials), users must implement a class that represents these Polynomials (*polynomial_t*), and a Comparison class that can compare two polynomials at the (unknown) value λ^* (*comparison_t*).

Concept Polynomial

Definition

Parametric search entails the comparison of polynomials. The precise form and representation of these is problem-specific. For instance, a polynomial of degree one, $f(x) = ax + b$, may be represented by the tuple (a,b), but it can also be seen as a line in two dimensions, and be represented by two distinct points on the line. We therefore place no requirements on Polynomials in general. However, Polynomials that are used in sorting-based parametric search (with either *Quick_sorter* or *Bitonic_sorter*) must comply with the following requirements:

- Polynomials are stored in containers of the C++ standard library, and must therefore be a model of *Assignable*, as defined in this standard. Informally, this means that it must be possible to make copies of objects of type Polynomial, and to assign values to variables of that type. A copy must be equivalent to the original.
- Objects of the (user-defined) Comparison class must be able to compare two Polynomial objects at the (unknown) value λ^*.

Concept Process

Definition

A Process is an object of a class derived from *process_base_t*, which is a typedef defined in a traits class, and defaults to *Proces_base< Traits >*. An object of such a derived class is a specialized function object that can be suspended and resumed, to allow for synchronization with other Processes and with Comparisons (objects of classes derived from *comparison_base_t*, which is a typedef defined in a traits class, and defaults to *Comparison_base< Traits >*

The computation performed by a Process is partitioned in a number of so-called *schedulable member functions*; these are member functions that have no parameters, and have return type *void*. Schedulable member functions are scheduled by calling *schedule* (inherited from *process_base_t*) in the member functions of a Process. Schedulable member functions may call other member functions of any type. Both schedulable and ordinary member functions may create other (child) Processes and Comparisons by using the *spawn* member function (inherited from *process_base_t*).

Upon request of the *Scheduler*, schedulable member functions are executed in the order in which they are scheduled. Each executed scheduled member function runs to completion. If during the execution of the scheduled member function any new child Processes or Comparisons are created (whether directly by the scheduled member function, or by any other member function called by the scheduled member function), the Process is suspended at the end of the scheduled member function. Only when all created child Processes and Comparisons have terminated, the Process itself will be resumed again by the *Scheduler* (i.e., the next scheduled member function will be called). If the child Processes and Comparisons created by the last scheduled member function have terminated, the Process itself is terminated as well.

At least one schedulable member function must be scheduled in the constructor of a Process. Scheduled member functions may schedule themselves or other member functions (this allows for iteration).

Creation

Processes must be created on the free store (i.e., with *new*). They may not be created on the stack (i.e., as local, global or static variables). Processes may not be deleted by user code (the *Scheduler* manages the life-time of all Processes).

The constructor of a Process has at least one argument *parent*, which is a pointer to the parent Process that created the Process; its type is *process_base_t**. This pointer may be a null-pointer (i.e., the Process is not created by another Process, but by a free function). The Process must initialize its own base class, *process_base_t*, with *parent* as argument:

process_base_t(parent)

The body of the constructor of a Process must schedule at least one of its own schedulable member functions (i.e., member functions without arguments and with return type *void*).

Has Models

Process_base, Bitonic_sorter, Quick_sorter

Class Process_base< Traits >

Definition

Abstract base class that provides the common part of the functionality that all Processes used in the parametric search framework share. Derived classes must be created on the free store. Other than the destructor, no member functions are virtual.

Is Model for the Concept

Process

Creation

The constructor has one argument, *parent*, of type *Process_base**; this arguments points to the (parent) Process that creates this (child) Process; if the child Process is created by a free function instead of by another process, *parent* must be a null pointer.

Process_base(Process_base parent)*

Note that objects of derived classes of *Process_base* may only be created on the heap (i.e., by using *new*). In user code, there must be no calls to *delete* on pointers to objects of derived classes of *Process_base*, as this is taken care of automatically by the *Scheduler*.

Operations

The destructor notifies the calling Process that the child Process has completed its execution; derived classes can and must rely on this to happen automatically.

virtual ~Process_base() = 0;

The functionality of the derived classes of *Process_base*(other than the inherited functionality of *Process_base* itself) must be implemented in one or more schedulable member functions of type *void (Derived::*)()* (i.e., member functions without arguments, returning *void*). These member functions must be registered by using the following function, provided by the base class *Process_base*:

template< typename Derived > void schedule(void (Derived:: mfp)());*

If a derived class of *Process_base* creates new Processes or Comparisons (objects of classes derived from *Traits::comparison_base_t*), these must be created on the heap (i.e., with *new*), and the pointer returned by *new* must be passed to the member function *spawn* of *Process_base*:

void spawn(Spawnable< Traits > child);*

Note: this function does not appear in the class definition of *Process_base*; it is inherited from *Spawner*, which is a common base class of *Process_base* and *Traits::comparison_base_t*. The class *Spawnable< Traits >* is a common base class of *Process_base*, *Traits::comparison_base_t*, and *Traits::root_t*.

Class Quick_sorter< Traits, BidirectionalIter >

Definition

Class that can be used for sorting-based parametric search. After creating an object of this class, the parametric search is started by calling the method *run* of the Scheduler.

Is Model for the Concept

Process

Creation

The constructor takes three arguments: a pointer to the parent process, if any, or 0 otherwise (in typical use-cases, the first argument will be 0), and two iterators indicating a sequence of polynomials to be sorted:

Quick_sorter(process_base_t parent, BidirectionalIter first, BidirectionalIter last);*

A *Quick_sorter* object must be created on the heap, i.e., with *new*.

Concept Root

Definition

A Root is an object of class *root_t*, which is a typedef defined in a traits class, and defaults to *Root*.

A Comparison may create one or more Roots, each of which contain the numerical value of a root of the polynomial associated with the Comparison, and a pointer to the creating Comparison. For some of the Roots, the *Scheduler* runs a *Decision Problem Solver* on the numerical value stored in the Root. For all other Roots, the *Scheduler* deduces the outcome of the *Decision Problem Solver* without actually running it. Either way, when the outcome is known for a Root, it is deleted by the scheduler. Upon being deleted, a Root must notify the Comparison that created it.

Creation

Roots must be created on the free store (i.e., with *new*). They may not be created on the stack (i.e., as local, global or static variables). Roots may not be deleted by user code (the *Scheduler* manages the life-time of all Processes).

Has Models

Root

Class Root< Traits >

Definition

Class that can be used by the parametric search framework to store roots of Comparisons.

Is Model for the Concept

Root

Creation

The constructor has two arguments. The first one, *parent*, of type *Traits::comparison_base_t** points to the Comparison that creates this Root. The second one, *value*, contains the numerical value of one of the roots associated with the creating Comparison.

Root(Traits::comparison_base_t parent, const Traits::number_t& value)*

Note that objects of class *Root* may only be created on the heap (i.e., by using *new*). In user code, there must be no calls to *delete* on pointers to objects of class *Root*, as this is taken care of automatically by the *Scheduler*.

See Also

Comparison, Scheduler, Traits

Class Scheduler< Traits >

Definition

Class that runs the parametric search by suspending and resuming Processes, evaluating Comparisons and running a *Decision Problem Solver* on selected Roots.

Creation

Since there must be only one instance of the *Scheduler* class, it is implemented as a singleton; its constructor is private. The instance can be accessed through the following public static member function:

static Scheduler instance();*

Operations

To start the parametric search, one must create one Process (that in turn will probably create other Processes and Comparisons, and so on, when it is activated by the *Scheduler*). This first Process has no parent process; it is created in a free function or in a member function of a non-Process class. Also, one must create a *Decision Problem Solver*.

After creating the first process and the *Decision Problem Solver*, the scheduler must be started by calling the following member function:

void run()

After *run()* has been called and executed, one can query the scheduler for the bounds on λ^*, the solution of the parametric search problem:

const Traits::number_t lower_bound() const
const Traits::number_t upper_bound() const

These functions either return both the same value (which is then equal to λ^*), or they return an interval from which λ^* can usually be easily computed; this is problem dependent. It may also be the case that the values returned by the two functions are different from each-other due to rounding errors; this may happen, for instance, if one defines *number_t* in the Traits class to be *double*.

The lower bound and upper bound on λ^* are initialized to $-\infty$ and ∞, respectively, and they are refined during the run of the scheduler. Updates of the bounds are usually expensive, and if one has prior information on the bounds, it will speed-up the parametric search if one sets the bounds to these known values. This can be done with the following member functions:

increase_lower_bound_to(const Traits::number_t& bound)
void decrease_upper_bound_to(const Traits::number_t& bound)

If the *Scheduler* is used to solve consecutive/independent parametric search problems, it must be reset before the creation of the first process of each run:

void reset()

This will, among other things, reset the bounds on λ^* to $-\infty$ and ∞, respectively.

The following functions may be called to gather statistical information of a run of the parametric search framework:

std::size_t no_processes()
std::size_t no_comparisons()
std::size_t no_comparisons_total()
std::size_t no_roots()
std::size_t no_computations()

The first two of these return the number of Process and Comparison objects, that have been created, respectively. The third function returns the number of comparisons that have been made in total (including the calls to *lazy_create_comparison* that didn't result in the actual creation of a Comparison object). The function *no_roots* returns the number of created Root objects; the last function returns the number of times that the (expensive) decision algorithm has been called on a root.

See Also

Process, Comparison

Class Solver_base< Traits >

Definition

Abstract base class that provides the common part of the functionality that all Decision Problem Solvers used in the parametric search framework share.

Is Model for the Concept

Decision Problem Solver

Types

The following type is provided:

enum decision_t less, equal, greater

Creation

The constructor of *Solver_base* takes no arguments:

Solver_base()

Note that objects of derived classes of *Solver_base* may only be created on the heap (i.e., by using *new*). In user code, there must be no calls to *delete* on pointers to objects of derived classes of *Solver_base*, as this is taken care of automatically by the *Scheduler*.

There can be at most one object of a class derived of *Solver_base< Traits >* at a time. Upon creation of such an object, the Scheduler will delete the previously created Decision Problem Object, if any.

Operations

The following function must be overridden by derived classes:

virtual void decision_t (const Traits::number_t& arg) = 0

This function must decide if its argument *arg* is less than, equal to, or greater than λ^*, and return the appropriate return value.

See Also

Decision Problem Solver

Class Sorting_traits<Traits, Number, Polynomial, Comparison, Solver>

Definition

This is a base class for a Parametric Search Traits class to be used with sorting-based parametric search (using either of the two classed *Quick_sorter* or *Bitonic_sorter*. The first template argument is the traits class to be defined (i.e., the derived class); the second template argument is the number type used to represent and compute λ^*; the remaining three argument are template type arguments, each taking one template parameter (a traits class), and representing the Polynomial, Comparison, and Decision Problem Solver classes:

struct MyTraits: public Sorting_traits< MyTraits, double, MyPolynomial, MyComparison, MySolver {}*;*

Is Model for the Concept

Parametric Search Traits

Types

The types *scheduler_t*, *process_base_t*, *comparison_base_t*, *root_t*, and *solver_base_t* resolve to the classes provided with the framework. The types *number_t*, *polynomial_t*, *comparison_t*, and *decision_solver_t* resolve to the template parameters:

typedef Scheduler< Traits > scheduler_t;
typedef Process_base< Traits > process_base_t
typedef Comparison_base< Traits > comparison_base_t;
typedef Root< Traits > root_t;
typedef Solver_base< Traits > solver_base_t;

typedef Number number_t;
typedef Polynomial< Traits > polynomial_t;
typedef Comparison< Traits > comparison_t;
typedef Solver< Traits > decision_solver_t;

Creation

Since this is a base class for traits classes to be used with the parametric search framework, no objects of *Solver_traits< Traits >* or classes derived thereof will normally be created.