Parametric Search Package: Tutorial

René van Oostrum and Remco C. Veltkamp

June 17, 2003

This document is the tutorial of the CGAL extension package for parametric search. It gives an overview of the parametric-search framework, and explains how to use it by describing implementations of several applications of parametric search. The full documentation of all classes and functions that are meant to be called from user-code can be found in the reference manual.

1 Introduction

Parametric search, the optimization technique by Megiddo [3], has always been regarded to be more of theoretical than of practical value. We provide a framework for parametric search that makes implementing applications of the technique considerably simpler, especially for sorting-based parametric search.

Theoretical background, some observations regarding the use of Quicksort for sorting-based parametric search, and a discussion about the need for Cole's optimization [2] can be found in our accompanying paper [6], see also [5].

This document is organized as follows. In Section 2 we discuss the basic ideas of parametric search. In Section 3 we give an overview of the important classes of the framework, and we explain how we use explicit template instantiation to be able to retain separate compilation. In Section 4 we show how to implement applications of sorting-based parametric search. As an illustration, we give the code that implements the example discussed in Section 2. Finally, in Section 5 we deal with general parametric search, illustrating the main ideas with the code of one of the classes of the framework.

2 Parametric search

In this section we will first explain parametric search in general terms, and then illustrate it with a frequently used example: the problem of finding the root of the median function of a set of monotone increasing linear functions. In Section 4.2 we will show how the parametric search framework can be used to solve this specific problem.

Parametric search is a technique for solving optimization problems for which there is a corresponding decision problem $P(\lambda)$ that is monotonous in λ, i.e., if $P(\lambda_0)$ is true, then $P(\lambda)$ is true for all $\lambda < \lambda_0$. The objective of the optimization problem is to find λ^*, the maximum value for which $P(\lambda)$ is true. If we want to apply parametric search to solve the optimization problem, then we need an algorithm A_s that solves the decision problem. More specifically, for any given value λ, A_s must be able to determine whether $\lambda < \lambda^*$, $\lambda = \lambda^*$, or $\lambda > \lambda^*$. Furthermore, the flow of control of A_s must depend on comparisons, each of which in turn depends on the sign of a polynomial in λ. The idea is then to run a "generic version" of A_s on the unknown value λ^*. Whenever a comparison is to be made, we have to determine the sign of a polynomial p at λ^*. But how do we dicide whether $p(\lambda^*) < 0$, $p(\lambda^*) = 0$, or $p(\lambda^*) > 0$ if the value of λ^* is unknown? The solution lies in the fact that the decision problem is monotonous. We compute the k roots $r_1, r_2, \ldots, r_k$ of p (these are concrete numbers, as opposed to the "symbolic" value λ^*), and run the concrete version of A_s on each of these roots. This tells us whether $r_i < \lambda^*$, $r_i < \lambda^*$, or $r_i < \lambda^*$, for each of the roots. After this, we know the location of λ^* among the roots of p: it lies either in between two consecutive roots, or it is smaller than the smallest root, or it is larger than the largest root. Assume that λ^* lies in between two consecutive roots r_i and r_{i+1} (the other cases can be treated similarly). Since the sign of $p(\lambda)$ is the same for any λ in the interval (r_i, r_{i+1}), and λ lies in this interval, we can determine the sign of $p(\lambda^*)$ by computing the sign of $p(\lambda)$ for an arbitrary λ in the interval. The sign of $p(\lambda^*)$ is all that is needed to resolve the comparison in the generic version of A_s, and hence we can resume its computation.

Running the conrete version of A_s on the roots of the polynomials establishes progressively tighter bounds on the value of λ^*. Its exact value is conveyed in either of two ways: either λ^* is a root of one of the polynomials itself (and this will be discovered by the concrete version of A_s), or we end up with a small interval of bounds on λ^* from which its value is relatively easy to compute.

We will illustrate this with an example (already given by Megiddo [3], and repeated by many others). Let $Y_i(\lambda) = a_i\lambda + b_i$ be a set of n linear functions such that all the a_i's are positive (in other words: Y_i is a set of

n lines in the plane with positive slope). Define $F(\lambda)$ to be the median of $Y_1(\lambda), Y_2(\lambda), \ldots, Y_n(\lambda)$ for all real values λ. F is a piecewise linear function with $O(n^2)$ breakpoints. Our task is to determine λ^*, the solution to the equation $F(\lambda) = 0$. Note that this problem can be solved trivially in linear time by calculating the median of the roots of the n linear equations. However, the purpose of this example is to illustrate the parametric search technique.

The decision problem corresponding to the above optimization problem is to determine, for a given value λ, whether $F(\lambda) < 0$, $F(\lambda) = 0$, or $F(\lambda) > 0$. It is straightforward to design an algorithm with a running time of $O(n)$ to solve the decision problem: compute all the values $Y_i(\lambda)$, determine the median of these values, and test the median against 0.

It may not be immediately clear how such an algorithm can be run generically on the unknown value λ^*: Computing the median of $Y_1(\lambda^*), Y_2(\lambda^*), \ldots, Y_n(\lambda^*)$ involves comparing function values at λ^*. How can we compare $Y_i(\lambda^*)$ and $Y_j(\lambda^*)$ without knowing their values? The key is that $Y_i(\lambda)$ and $Y_j(\lambda)$ are polynomials. If we define the difference polynomial Y_{ij} to be $Y_{ij}(\lambda) = Y_i(\lambda) - Y_j(\lambda)$, then $Y_i(\lambda) < Y_j(\lambda)$ iff $Y_{ij}(\lambda) < 0$; $Y_i(\lambda) = Y_j(\lambda)$ iff $Y_{ij}(\lambda) = 0$; and $Y_i(\lambda) > Y_j(\lambda)$ iff $Y_{ij}(\lambda) > 0$. In other words: the outcome of the comparison of $Y_i(\lambda)$ and $Y_j(\lambda)$ is determined by the sign of the polynomial $Y_{ij}(\lambda) < 0$. Since we want to compare Y_i and Y_j at λ^*, we have to compute the sign of $Y_{ij}(\lambda^*)$. We don't know the value of λ^*, but that poses no real problem. Y_{ij} is a polynomial of degree one, and we can compute its single real root r_{ij}. Next, we run the algorithm for the decision problem on r_{ij}. If $F(r_{ij}) < 0$, we know that $\lambda^* > r_{ij}$ (since F is monotone increasing). So in that case, $Y_{ij}(\lambda^*)$ must be greater than zero, since Y_{ij} is also monotone increasing. Similarly, if $F(r_{ij}) = 0$ then $Y_{ij}(\lambda^*) = 0$ (and $\lambda^* = r_{ij}$), and if $F(r_{ij}) > 0$ then $Y_{ij}(\lambda^*) < 0$. In this way we can compare $Y_i(\lambda^*)$ and $Y_j(\lambda^*)$ without knowing the actual value of λ^*.

After the generic algorithm has finished, we know which of the functions Y_i is the median at λ^*. The value of λ^* is then simply the x-coordinate of the intersection of the median function with the x-axis.

The running time of this algorithm is $O(n^2)$ if we employ a linear time median finding algorithm for both the generic and the concrete version of A_s. We see that performing one comparison is rather expensive, namely $O(n)$ time. Megiddo therefore suggests to replace the generic version of A_s by a parallel algorithm A_p. In our case, we use a parallel sorting algorithm that uses $O(n)$ processors to sort n items in $O(\log n)$ parallel time (after sorting, we can easily find the median). We simulate the parallel algorithm sequentially, but instead of running the decision problem on each root immediately as we encounter it, we collect all the $O(n)$ roots of the comparisons that are performed during one parallel step. We determine the median r_m of the roots, and run the concrete version of A_s on it to decide whether $r_m < \lambda^*$, $r_m = \lambda^*$, or $r_m > \lambda^*$. From the outcome for r_m we can immediately deduce the outcome for at least half of the roots. For instance, if $r_m < 0$, then $r_i < 0$ for all $r_i < r_m$. One such step takes linear time, and we must repeat it $O(\log n)$ time to resolve all comparisons of one parallel step of the generic algorithm. Since the algorithm takes $O(\log n)$ parallel steps, the overall running time is $O(n \log^2 n)$. Cole [2] shows how this can be further reduced to $O(n \log n)$.

There are two main problems to parametric search. The first one is that it may be difficult to design an efficient parallel algorithm for the decision problem at hand. Fortunately, in several cases (such as in the above example), the generic algorithm can be replaced by sorting. Still, efficient parallel sorting algorithms exist, but they are complicated, and/or have large hidden constants in their time complexity. However, the parallelism of the generic algorithm is not essential; it is important that independent comparisons can be batched and resolved in a binary-search fashion as explained above. In our companion paper [6] we show that Quicksort allows the batching of comparisons.

The second difficulty lies in the batching of comparisons. This is difficult to implement, whether the generic algorithm is parallel or not. When the algorithm is to perform a comparison, execution must be suspended, and another part of the algorithm must be executed to collect another comparison, and so on. When enough comparisons have been collected, they must be resolved. Next, the algorithm must be resumed in all the places where it has been suspended before. This completely disrupts the flow of control, and it has been

considered as difficult to implement.

Our framework makes it much easier to implement parametric search (especially when the generic algorithm can be replaced by sorting), as it takes care of all the suspending and resuming of computation with hardly any effort from the user. In Section 4.2 we show how to implement the example given in this section, and Section 4.3 we give a more complex example. But first, we give a brief overview of the framework and its important classes in the next section.

3 The parametric search framework

3.1 Important classes

In the parametric search framework, we distinguish five important types of classes (see Figure 1):

1. Processes: these are classes derived from `Process_base`, which is a base class provided by the framework. Through the functionality of the base class, Processes can create other processes and/or Comparisons, and suspend and resume their computation.
2. Comparisons: these are classes derived from `Comparison_base`. They can compare polynomials at the unknown value λ^* if they know the outcome of the decision problem for the roots of the difference polynomial.
3. Roots are created by Comparisons and batched by the Scheduler.
4. Decision Problem Solver: this class, derived from `Solver_base`, determines whether a Root is less than, equal to, or greater than λ^*.
5. Scheduler: this class tells Processes to perform a part of their computation. This results in the creation of other Processes and Comparisons, which in turn causes Roots to be created. The Scheduler then computes the median Root, and runs the Decision Problem Solver on this Root. As explained in Section 2, this immediately gives the outcome of the decision problem of half the roots. Next, the Scheduler tells the Comparisons for which the answer to the decision problem for all Roots is known to compare their polynomials at λ^*. When all Comparisons created by a Process have done their work, the Scheduler tells the Process to do the next step of its computations. This may again result in the creation of other Processes, Comparisons, and Roots, and this cycle continues until there are no more Processes, or until one of the Roots turns out to be equal to λ^* (in which case the answer is found, and the computation is aborted).

3.2 Compilation model

The usual way of implementing classes in C++ is the following: class definitions (containing *declarations* of member functions, amongst other things) are specified in a *header file* (usually `*.h`), and the *definitions* of the member functions are defined in an *implementation file* (usually `*.cpp`, `*.cc`, or `*.C`). This makes it possible to do *separate compilation*: a class *A* that needs to call member functions of class *B* only needs the class definition of class *B* (specified in `B.h`), and the implementation of class *A* can be compiled even if the member functions of class *B* have not been defined yet.

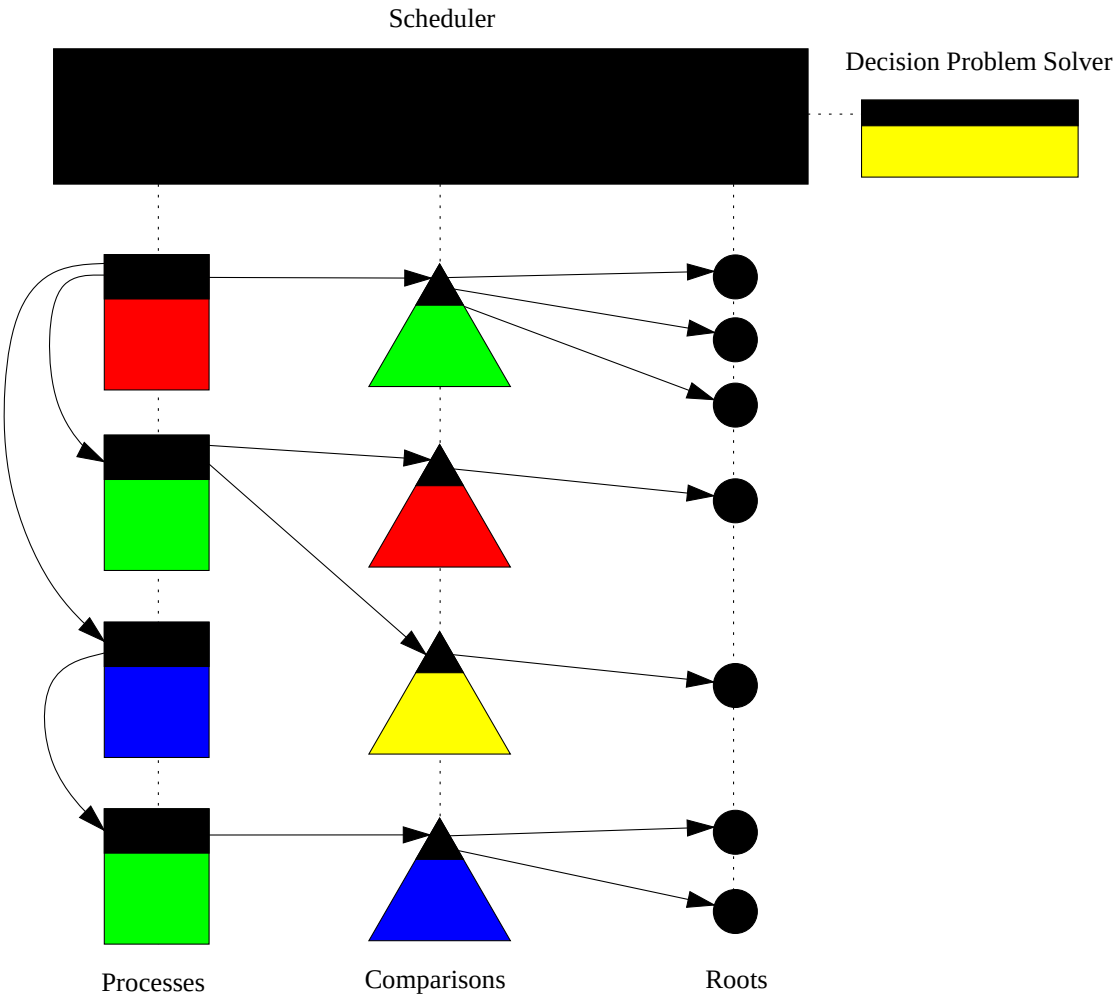

Figure 1: Class diagram: black parts are classes provided by the framework (Scheduler, Process_base, Solver_base, Comparison_base, Root), colored parts are derived classes to be implemented by the user.

If classes are template classes, this becomes a bit more complicated. The C++-standard defines a model for separate compilation of class templates using the `export` keyword, but at the moment of writing there is only one compiler (Comeau) that supports the `export` keyword.

The usual work-around is to put the definitions of the member functions of the template classes in the header files as well. In practice, this means that all code of all template classes is somehow included in one main `*.cpp` file, and only one object file will be created by the compiler. A modification of one template class member function makes it necessary to recompile the whole application. Although there are compilers that are intelligent enough to only compile the code that has been changed since the previous compilation, in general this way of working is seen as a drawback of template classes.

There is another solution, based on *explicit template instantiation*, and we use it in our framework. The idea is as follows: template class definitions and definitions of member functions are separated again into `*.h` and `*.cpp` files. Suppose that we have created a template class `MyClass<T>` in this way, and we want to use it with `T = int`. We cannot simply `#include myclass.h` and write `MyClass<int> myobject`: the linker will not find the definitions of the member functions of `MyClass<int>`. However, we can create a file that includes `myclass.cpp` (which contains the definitions of the member functions of `MyClass<T>`), and explicitly instantiates `MyClass<int>`:

```
#include "myclass.cpp"
template class MyClass<int>;
```

Compiling this file will result in an object file that contains the definitions of the member functions of `MyClass<int>`, and this object file can then be linked with the other object files. In this way, we can still do separate compilation.

In the parametric search framework, the template argument is usually a traits class. Rather than specifying this traits class hard-coded in all `*.cpp` files that contain explicit template instantiations, we add one level of indirection: the template arguments is a typedef `traits_t` that is defined in `instantiate.h`:

```
// file instantiate.h
typedef traits_t MyTraitsClass

// file myclass_inst.cpp
#include "instantiate.h"
#include "myclass.cpp"

template class MyClass<traits_t>;
```

In this way, we can define (and change) the specific template parameter of the explicit instantiations of many template classed by only defining (and changing) it in one place, namely, in `instantiate.h`.

In the next section it will become clear how this is used in practice.

4 Sorting-based parametric search

Since in several applications of parametric search the generic decision algorithm can be replaced by sorting (as long as at least the same comparisons are performed), we implemented two sorting algorithms, namely,

Quicksort and Bitonic Sort. These algorithms were implemented using our framework, and both implementations "know" how to batch comparisons. Problems that can be solved with sorting-based parametric sort can be implemented relatively easy using either of these two sorting algorithms. In practice, Quicksort runs faster than Bitonic sort.

4.1 Tasks for the user

The user who wants to use the parametric search framework to do sorting-based parametric search must implement a small number of classes:

- A class that represents the polynomials in the generic algorithm. In the example presented in Section 2 these are the lines Y_i.
- A class that solves the decision problem for any concrete value λ. It must meet the requirements of the concept **Solver**.
- A class that can compare two polynomials in λ at the value λ^*. This class must be derived from `comparison_base`, and it must meet the requirements of the concept **Comparison**, as specified in the reference manual.
- A `Traits` class to links all the relevant classes of the framework and the user-defined classes together. The three above-mentioned classes will be instantiated with the `Traits` class as a template argument.

Furthermore, the user must create a file `instantiate.h` that defines a typedef `traits_t`, which resolves to the user-defined `Traits` class. The presence of this file is needed for the explicit instantiation of the framework template classes, as explained in Section 3.2. Furthermore, if the user wants to exploit explicit template instantiation herself, she must create one or more `*.cpp` files that contain the explicit template instantiation. Each of these `*.cpp` files must include `instantiate.h`.

4.2 Example: median of monotone increasing linear functions

In this section we show how to use the parametric search framework to solve the median-of-lines problem explained in Section 2 by implementing the required classes (see Section 4.1.) The files with the C++ source code for this example can be found in the subdirectory `Tutorial/median_of_lines` of the parametric search package. The implementation of this example does not use any part of the CGAL library.

4.2.1 Class `Line`

The polynomials in the example are lines with positive slope. We implement it with a simple `struct` named `Line`; see `line.h` and `line.cpp`. The class represents the line $y = ax + b$, and stores a and b. The number type to represent a and b is not hard-coded; it is specified in a traits class, which is used as a template parameter to the line class.

4.2.2 Class `Line_solver`

The second class we implement is the class `Line_solver` that solves the decision problem for any concrete value λ. This class is a model of the concept **Solver**. It has a function named `decide` that takes a

real value as input (as before, the specific number type is specified in the traits class), and it returns an `enum decision_t`, with value `less`, `equal`, or `greater`. In our case, the decision problem is solved by evaluating the y-coordinates of the lines Y_i at the specified value `x_coord`, computing the median of these y-coordinates, and testing whether the median is less then, equal to, or greater than zero (see Section 2).

Since the decision solver needs the lines to solve the decision problem, its constructor takes the lines as input, and stores them in a member container.

To compare two lines, we define a class `Line_comparator` as a nested class of `Line_solver`. Its constructor takes the x-coordinates at which the lines must be compared. Its function call operator (`operator()`) takes two lines as arguments, and returns `true` if the first argument is less than the second argument at the specified x-coordinate. The type of the lines is specified in the traits class; its typedef-ed name is `polynomial_t`. Note: `Line_comparator` compares two lines at any *concrete* x-coordinate. This class should not be confused with the class for comparing two lines at the *unknown* value λ^*. The latter class is presented in the next section.

To determine the median of the lines at the given x-coordinate, we use the algorithm `nth_element` in the algorithms library, as defined by the C++ standard. It can be used as a median finding algorithm, and runs in linear time.

4.2.3 Class `Line_comp`

This class, which implements the concept **Comparison**, performs a (possibly deferred) comparison between two lines at the unknown value λ^*. This operation is possibly expensive: if the x-coordinate of the intersection of the two lines lies within the current interval of bounds on λ^*, we must compute the median line at this x-coordinate, and this takes $O(n)$ time. Otherwise, if the x-coordinate of the intersection does not lie inside the interval of bounds on λ^*, we can simply deduce the outcome of the comparison in $O(1)$ time; see Section 2. In general, the parametric search framework will choose to defer the expensive operation, collecting it in a batch of comparisons that is resolved efficiently later on.

In our example program, we use the `Quicksort` class to replace the generic version of the algorithm for the decision problem. Whenever the `Quicksort` class must compare two lines at λ^*, it creates an object of our class `Line_comp`. However, such a creation is expensive, and there may be many of them. Therefore, there is an additional requirement for comparison classes (derived from `Comparison_base`) that are used in cooperation with `Quicksort` (or `Bitonic_sort`). The comparison class (`Line_comp` in our example) must have a static member function `lazy_create_comparison` that first checks whether the outcome of the comparison can be deduced in $O(1)$ time. This is the case if al roots of the difference polynomial lie outside the interval of bounds on λ^*. If so, the result of the comparison is passed to the caller (through a reference argument `comp_result`), and no object of type `Line_comp` is created. Otherwise, if the outcome cannot be deduced efficiently, `lazy_create_comparison` creates a `Line_comp` object, and returns it to the caller.

The static member function `lazy_create_comparison` is the only function that may create an object of type `Line_comp` (it is a so-called *factory method*), and the constructor of `Line_comp` is therefore `private`. The implementation of `lazy_create_comparison` is straightforward; see the source code. It uses a utility function `do_deduce` to do the deduction.

The implementation of the constructor is also straightforward. One of its arguments is a pointer to the calling `Process` (which is a `Quicksort` object in our example). The constructor of `Line_comp` must pass this *parent process* to its direct base class, `comparison_base`. Furthermore, the data members of `Line_comp` are initialized with the appropriate arguments to the constructor. The body of the constructor creates a new `root_t` object. The constructor of `root_t` has two arguments: (a) a pointer to the calling process

(Line_comp itself), and (b) the x-coordinate of the intersection of the two lines that must be compared at λ^*. Since lazy_create_comparison already computed this intersection, and stored it in the static member static_root_value, we can copy it from there. (This is safe, since only lazy_create_comparison creates Line_comp objects, so static_root_value will hold a correct value when the constructor of Line_comp is called.)

Finally, objects that model the **Comparison** concept are required to have a member function deduce. This member function is only called on **Comparison** objects (Line_comp objects in our example) for which the root_values (in our example, the x-coordinate of the intersection of two lines that are compared at λ^*) lie *outside* the interval of bound on λ^*. This means that the outcome of the comparison can be deduced in $O(1)$ time. The function deduce simply forwards the work to the utility function do_deduce, that is also used by lazy_create_comparison.

4.2.4 Class Line_traits

This class implements the concept **Traits** as laid out in the reference manual of the parametric search package. The Line_traits class mainly contains type definitions for basic types (i.e., number_type for the number type used for calculations) and for the important classes in the framework. This way, the main classes of the framework (that all have a Traits class as template argument) can refer to each other without having to know the specific type of the cooperating classes. As a result, it is relatively easy to replace one or more of the classes in the framework, should the need arise. The other classes need not be changed; only the relevant typedef in the Traits class must be modified.

For sorting-based parametric search using the default components of the framework, there is a very convenient way to define an appropriate traits class. We derive Line_traits publicly from Sorting_traits (defined in traits_base.h, and provide as arguments Line_traits itself, the desired number type (double in our example), the type of the polynomials (Line), the type of the class for comparing two polynomials (Line_comp), and the type of the class that solves the decision problem (Line_solver). This is our resulting file line_traits.h:

```
// include standard components of the framework
#include "traits_base.h"

// include our own problem-specific classes
#include "line.h"
#include "line_comp.h"
#include "line_solver.h"

// define the traits class simply by deriving from Sorting_traits
struct Line_traits :
  public Sorting_traits< Line_traits, double,
                         Line, Line_comp, Line_solver > {};
```

4.2.5 Remaining tasks

The user is required to create a file instantiate.h, that contains a typedef traits_t, resolving to our traits class (Line_traits):

```
#include "line_traits.h"

typedef Line_traits traits_t;
```

Furthermore, if the user wants to exploit explicit template instantiation, she must create one or more `*.cpp` files with the instantiations. For instance, for our `Line_solver` class, the file `line_solver_inst.cpp` is as follows:

```
#include "instantiate.h"
#include "line_solver.cpp"

template class Line_solver< traits_t >;
```

In the example code, there is a second instantiation file `line_comp_inst` for our class `Line_comp`. For our very simple class `Line`, we defined the constructor inline in `line.h`, and there are no other member functions. This removes the need for an accompanying file `line.cpp` and any explicit template instantiation for this class.

Finally, we write a small driver program (see `median_of_lines.cpp`) to put the framework and the user-defined classes to work. In this program, we first create a number of lines with positive slope. Next, we create a `Line_solver` object that takes the lines as argument:

```
  typedef Line_solver< Line_traits > solver_t;
  new solver_t( lines.begin(), lines.end() );
```

As most of the framework classes and user-defined classes derived thereof, the `Line_solver` object must be created dynamically (i.e., with `new`). The framework takes care of the lifetime of the `Line_solver` object; we may not `delete` it ourselves. Hence, there is no need to store the pointer returned by `new`.

Next, we create an object of type `Quick_sorter`; it will sort our lines at the unknown value λ^*:

```
  typedef std::vector< line_t >::iterator iter_t;
  typedef pms::Quick_sorter< Line_traits, iter_t > sorter_t;
  new sorter_t( 0, lines.begin(), lines.end() );
```

The second template argument of `Quicksorter` tells us the type of the iterator of the container in which we store our lines. In our case, we used a `std::vector`, but we could for instance have used a `std::list` just as well.

Again, we create the `Quick_sorter` object dynamically, and leave it to the framework to `delete` it when it is no longer needed. The first argument to the `Quick_sorter` constructor indicates its *parent process object* (see Section 5). In our case, there is no such object, so we pass a zero.

Note that the namespace `pms` is a shorthand for `ns_parametric_search`. This is the namespace that contains the framework components.

Next, we put the framework to work:

```
  pms::Scheduler< Line_traits >::instance()->run();
```

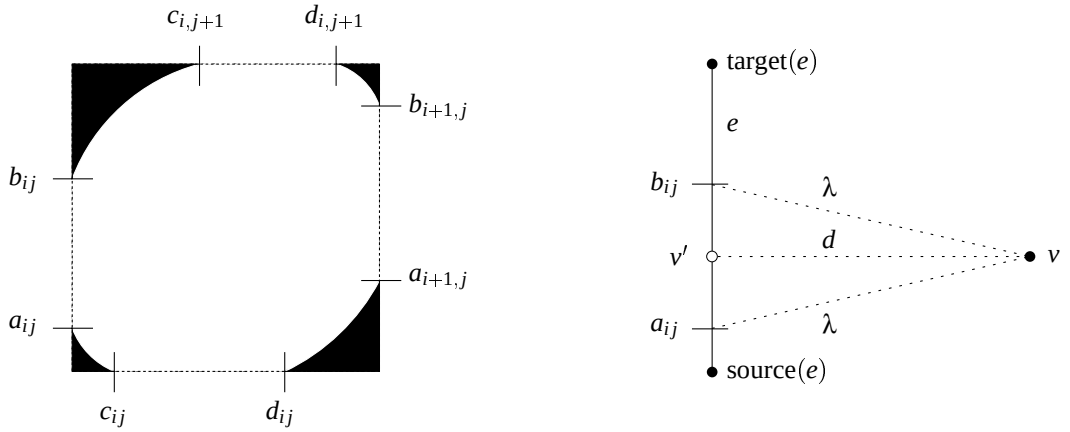

Figure 2: Critical points in a cell of the diagram (left) and the computation of the critical points from an edge and a vertex (right).

Note that we never created a `Scheduler` object. It is implemented as a *singleton class*. The static member function `instance` will create the one and only `Scheduler` object if it hasn't been created before, and it returns a pointer to the `Scheduler` object.

When the framework returns from the call to `run`, the lines will have been sorted at the value λ^*. The rest of the code in `median_of_lines.cpp` computes and prints the x-coordinate of the intersection of the median line with the x-axis.

4.3 Example: Fréchet distance between two polygonal lines

In this section, we show how to implement the algorithm by Alt and Godau [1] for computing the Fréchet distance between two polygonal curves; we assume that the reader is familiar with the cited paper.

Informally, let P be a polygonal curve, defined by the vertices $p_0, \ldots, p_m$. Its edges are $e_i^P = (p_{i-1}, p_i)$ for $1 \leq i \leq m$, and its length is m. Let polygonal Q with vertices q_j, edges e_j^Q, and length n be defined similarly. The algorithm by Alt and Godau [1] uses a so-called *free space diagram* to compute the Fr'echet distance between P and Q. This is a rectangular diagram of $m \times n$ squares C_{ij}; the free space in each square is the intersection of the square with a (possibly empty) ellips (see Figure 2, left). The size of the ellipses depends on a parameter λ: a point in cell C_{ij} is in the free space if the Euclidean distance between the corresponding points on e_i^P and e_j^Q at most λ. If, for a given λ, there is a path from $(0,0)$ to (m,n) in the diagram that is monotone in both x and y and that lies completely in the free space, then the Fréchet distance between P and Q is at most λ.

Important (critical) points in the diagram are the intersections of the ellipses with their cells (the points a_{ij}, b_{ij}, c_{ij}, and d_{ij} in Figure 2 (left). The precise location of these points can be computed from the value of λ and the distance from a vertex of one curve to an edge of the other curve, as illustrated in Figure 2 (right). In fact, these values are polynomials in λ. Each cell has four boundary edges, but each boundary edge (and the two critical points a_{ij} and b_{ij}, or c_{ij} and d_{ij} on it) is shared by two cells (except for some edges of cells in the first/last row or column in the diagram). Therefore, in our implementation we maintain for each cell only the critical points on the left and bottom boundary edge. To represent critical points on the right boundary edges of cells in the rightmost column and on the top edges of cells in the topmost row, we extend the diagram with an extra row and column. The non-relevant boundary edges of these cells are called `phantom_edge`, and we don't compute critical points on them.

Alt and Godau show that for $\lambda = \lambda^*$, one of the following t conditions must hold:

- λ is minimal, and both $(0,0)$ and (m,n) in the diagram are in the free space.
- $a_{ij} = b_{ij}$ or $c_{ij} = d_{ij}$ for some i, j.
- $a_{ij} = b_{kj}$ or $c_{ij} = d_{ik}$ for some i, j, k.

The first condition (which states that the Fréchet distance between the two curves is at least the distance between the begin- and endpoints of the curves) can easily be tested separately, and gives a lower bound on λ^*. The second and third conditions can be tested by sorting the polynomials a_{ij}, b_{ij}, c_{ij}, and d_{ij} at the (unknown) value λ^*. In other words: sorting-based parametric search can be applied here.

In the following sections we will explain how the various classes of the framework are implemented. The source code for this example can be found in the subdirectory `Tutorial/frechet_distance` of the parametric search package.

4.3.1 Class `Frechet_polynomial`

The polynomials in this example are the values a_{ij}, b_{ij}, c_{ij}, and d_{ij}, as explained above. By sorting these at the unknown value λ^*, we are guaranteed to compare the polynomials that are critical (corresponding to the second or third case in the list of conditions above).

In our implementation, we call a_{ij} and b_{ij} `left_edge_polynomials` (since they are located at the left edge of some cell in the diagram). Similarly, c_{ij} and d_{ij} are called `bottom_edge_polynomials`. In the same way, we call a_{ij} and c_{ij} `lower_bound_polynomials` (since they bound the intersection of the ellips and the diagram edge from below), and b_{ij} and d_{ij} are called `upper_bound_polynomials`.

We remark that it is not always necessary to compare the numerical values of two polynomials. For instance, we only need to compare the numerical values of a_{ij} and b_{kl} if $i = k$ (in other words, we only need to compare the values of two `left_edge_polynomials` if they are in the same row of the diagram). Something similar holds for c_{ij} and d_{kl}: we only need to compare the values of two `bottom_edge_polynomials` if they are in the same column. Also, we don't need to compare the values of a `left_edge_polynomial` and a `bottom_edge_polynomial`. We exploit these facts to avoid unnecessary (expensive) computations by storing, for each polynomial, whether it is a `left_edge_polynomial` or not, whether it is a `lower_bound_polynomial` or not. We also store the row and column of the cell for each polynomial. More details of these optimizations are discussed in Section 4.3.3.

Also, we note that taking square roots (often needed in distance computations) is usually expensive, especially when number types as `leda_real` are used. We therefore avoid taking square roots as much as possible, by comparing squared distances whenever possible. So for each polynomial, we pre-compute the (squared and normal) distance of the appropriate vertex v to the appropriate edge e (see Figure 2, right), and the (squared and normal) distance from the source of the edge to the projection of v on the supporting line of e (v' in Figure 2, right; note that v' lies halfway a_{ij} and b_j. Finally, we compute the (squared) minimum value of λ (inconveniently called `eps` in the source code) for which a_{ij} and b_{ij} (or c_{ij} and d_{ij}) are defined (note that the projection of v' on the supporting line of e doesn't necessarily lie on e itself, and therefore this minimum value for λ isn't necessarily equal to the (squared) distance from v to e.

This leads to the definition of `Frechet_polynomial` as it is in the file `frechet_polinomial.h`.

4.3.2 Class `Frechet_solver`

The class `Frechet_solver` is a model of the concept **Solver** (see the reference manual). It has a function `decide` that takes a real value λ as input, and it returns an enum `decision_t` with value `less`, `equal`, or `greater`.

The constructor of `Frechet_solver` takes two `Polylines` as input; basically, these are `std:vectors` of `Points`. The constructor then builds the diagram of cells, following the algorithm by Alt and Godau [1].

The function `decide` computes, for any input value λ, whether λ is less than, equal to, or greater then λ^*, the Fréchet distance between the two polylines. To do so, it first computes, for all cells in the diagram, the intersections of the ellipses with the boundaries of the cells. Next, it computes for all cells of the diagram in a top-to-bottom and left-to-right order, the points on the boundary edges of the cells that can be reached from the lower-left corner of the diagram with a path that is monotone in both x and y. If the top-right corner of the diagram can be reached with such a monotone path, then λ is greater than or equal to λ^*; otherwise, λ is less than λ^*. To distinguish between `equal` and `greater`, we maintain for each cell if the monotone path to that cell has *slack*, i.e., if the value λ is tight for that cell or not.

4.3.3 Class `Frechet_comp`

The class `Frechet_comp` models the concept **Solver**. It compares two `Frechet_polynomials` at the unknown value λ^*. As explained in Section 4.3.1, we avoid the comparison of the numerical values of two polynomials if we already know that the comparison will not reveal the value of λ^*. Instead, we define a simple order on the `frechet_polynomials` such that they can be compared and sorted relatively easy:

- Since there is no need to compare the numerical values of `left_edge_polynomials` (the values a_{ij} and b_{ij} with the numerical values of `bottom_edge_polynomials` (the values c_{ij} and d_{ij}), we define all `left_edge_polynomials` to be less than `bottom_edge_polynomials`.
- Since it is only useful to compare the numerical values of two `left_edge_polynomials` if they are in the same row of the diagram, we define `left_edge_polynomials` a_{ij} and b_{ij} to be less than `left_edge_polynomials` a_{kl} and b_{kl} if $l < j$ (i.e., if they are not in the same row, they are ordered by row number). Analogously, `bottom_edge_polynomials` are ordered by column number if they are not in the same column of the diagram).

If the numerical values of two `frechet_polynomials` have to be compared (i.e., if we compare two `left_edge_polynomials` in the same row of the diagram, or two `bottom_edge_polynomials` in the same column), then we proceed as follows:

Suppose that we need to compare the polynomials a_{ij} and a_{kj} at λ^*. Doing the comparison for any given value λ is straightforward (see Figure 3, left), but since we don't know the value if λ^*, we must take a different approach. First, we compute the value λ' for which the polynomials a_{ij} and a_{kj} are equal (see Figure 3, right). This value is easily found by determining the intersection of (the supporting line of) the edge e_j^Q and the bisector of p_i and p_k. It is easy to see that $a_{ij} > a_{kj}$ for all values $\lambda > \lambda'$, and similarly, $a_{ij} < a_{kj}$ if $\lambda < \lambda'$ (unless λ is so small that one or both of a_{ij} and a_{kj} vanish; we will discuss this below).

In other words: to resolve the comparison of a_{ij} and a_{kj} at λ^*, we need to compare λ^* and λ'. If λ' is less than the current lower bound on λ^*, or greater than the current upper bound on λ^*, the comparison of λ^* and λ' comes for free, and we can immediately determine the outcome of a_{ij} and a_{kj} at λ^*; this is done in the function `lazy_create_comparison` of the class `Frechet_comp`. Otherwise, if λ' lies inbetween the lower

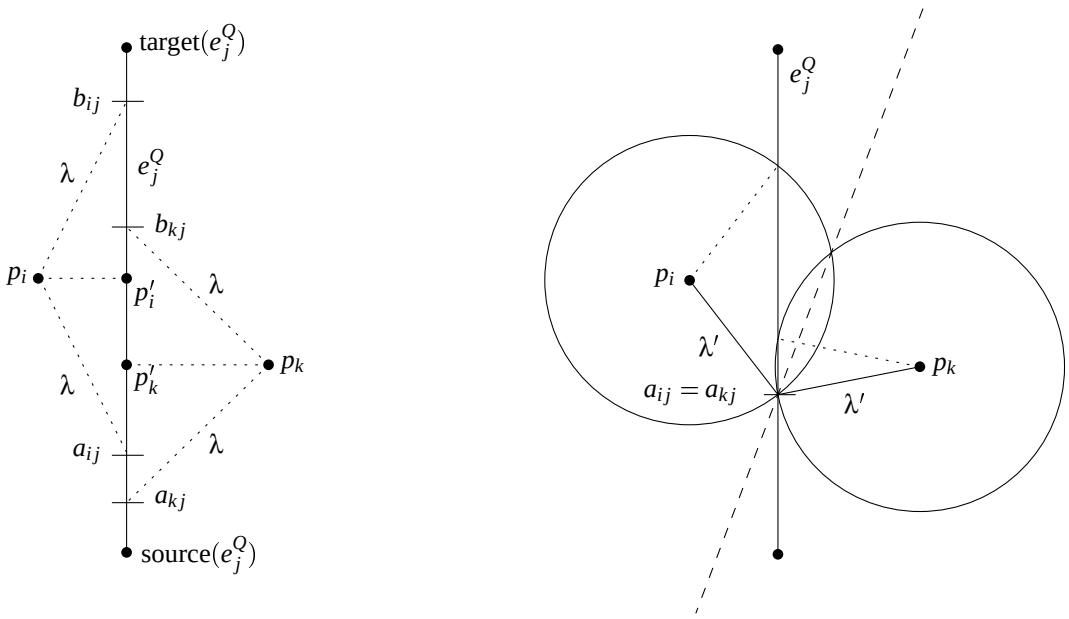

Figure 3: Comparing a_{ij} and a_{kj} for a given value λ (left), and computing the value λ' such that $a_{ij} = a_{kj}$ (right).

and upper bound on λ^*, we would have to run the decision process on λ'. We let the framework take care of this; if this situation arises, then `lazy_create_comparison` creates an object of class `Frechet_comp`. The constructor of this object then creates a `Root` object with value λ'.

At some point, after the framework has determined the outcome of the decision process for λ', it will call the function `deduce` of the `Frechet_comp` that spawned the `Root` object with value λ'. At this point, the relative location of λ^* with respect to λ' is known, so the comparison of a_{ij} and a_{kj} at λ^* can be performed, as described above.

There are some technicalities left to discuss. As we remarked above, `Frechet_polynomials` may vanish; for instance, for values of λ smaller than the distance from p_k to p_k' in Figure 3 (left), a_{kj} and b_{kj} do not exist. For efficiency reasons, we would like to add another simple ordering criterion for the polynomials at λ^*: if both polynomials to be compared vanish at λ^*, we consider them equal; otherwise, if only one of the polynomials vanishes at λ^*, we consider it to be the smaller of the two; otherwise, if both polynomials are defined at λ^*, their order is determined by their numerical values.

This causes the following problem with the approach of comparing polynomials that we described above: suppose that for the situation depicted in Figure 3, we want to compare a_{ij} and a_{kj} at λ^*. Assume that we have a lowerbound λ_l and an upperbound λ_u on λ^*; also assume that λ' (the value for which $a_{ij} = a_{kj}$) is greater than λ_u. Furthermore, let λ_{kj} be the value for which $a_{kj} = b_{kj}$ (so both a_{kj} and b_{kj} vanish for values less than λ_{kj}, and let λ_{ij} be defined similarly. Assume the following order: $\lambda_{ij} < \lambda_l < \lambda_{kj} < \lambda_u < \lambda'$. Now, to determine the outcome of the comparison of a_{ij} and a_{kj} at λ^*, we cannot simply compare the two polynomials for an arbitrary value in the interval $[\lambda_l, \lambda_u]$, since the outcome depends on whether this arbitrary value is less than λ_{kj} (in which case a_{kj} is not defined, and therefore less than a_{ij}) or greater than λ_{kj} (in which case both polynomials are defined, and $a_{ij} < a_{kj}$). Now, if the order of two polynomials is not properly defined, the polynomials will in general not be sorted properly, and the comparison of the two polynomials that determine λ^* may even not be carried out.

The problem would disappear if the interval of bounds on λ^* didn't contain any of the values λ_{ij} for which the polynomials a_{ij}, b_{ij}, c_{ij}, or d_{ij} vanishes. This suggests the following solution: collect all these values

λ_{ij}, consider them as `Roots`, and use the parametric search framework to locate λ^* among them by running the decision process on these values in a binary search fasion. This will result in an upperbound and a lowerbound on λ^*; these bounds are two consecutive values λ_{ij}. In fact, one of the values λ_{ij} may be λ^*; this will be detected by the framework, and the computation will be aborted. In our implementation, this is done prior to sorting as a preprocessing step; see the class `Frechet_preprocessor`.

4.3.4 Class `Frechet_traits`

The definition of the required traits class is almost as simple as in the implementation of the median-of-lines algorithm presented in Section 4.2. Again, we derive the traits class from `Sorting_traits`, and provide the appropriate template parameters. In the body of the traits class, we define an additional type `Point` that determines the input type.

```
#include "traits_base.h"

// include our own problem-specific classes
#include "frechet_polynomial.h"
#include "frechet_comp.h"
#include "frechet_solver.h"

#include "cgal_kernel.h"
#include <CGAL/Point_2.h>

// define the traits class simply by deriving from Sorting_traits
struct Frechet_traits :
  public Sorting_traits< Frechet_traits, long double,
                         Frechet_polynomial, Frechet_comp, Frechet_solver >
{
  typedef CGAL::Cartesian< double >        CGAL_kernel_t;
  typedef CGAL::Point_2< CGAL_kernel_t >   Point;
};
```

5 General parametric search

For parametric-search problems that cannot be solved with sorting-based parametric search, our framework provides lower-level facilities for batching comparisons and suspending and resuming computation. This is perhaps best illustrated with a (toy) example:

Suppose that we have the following function `f()`:

```
1.  void f(){
2.     int i, j;
3.
4.     g1(i); g2(j);  // call by reference.
5.     i += j;
6.  }
```

Suppose that we want to execute this code in the parametric-search setting, and that the outcome of the calls

to g1() and g2() in line 4 depends on mutually independent comparisons. We would like to batch those comparisons. If we restrict our attention to line 4 in the code above, then we see that first g1() should be partially executed (up to the comparison it makes); next, g2() must be partially executed; then both g1() and g2() have to wait until their comparisons are resolved. Next, g1() must resume its computation and set the value of i, and g2() is resumed to set the value of j.

Suppose that g1() and g2() know how to take care of their suspension, comparison batching, and resumption themselves, and that the only thing f() has to do in line 4 is the calling of the two functions. Then f() still has to wait until both g1() and g2() have completed their computation of i and j before it can compute the new value of i in line 5. How should f() suspend its computation after line 4, and how should it know that g1() and g2() have finished, so that it can resume its computation in line 5?

The solution that our framework takes is that functions are turned into objects of **Process** classes. Let's call these classes corresponding to the three functions above proc_f, proc_g1, and proc_g2, respectively. All three are derived from Process_base. The calling of g1() and g2() by f() translates to the dynamic creation (i.e., with new) of a proc_g1 and a proc_g2() object. The functionality of g1() and g2() is implemented in member functions of the proc_g1 and proc_g2 objects. These member functions are called by the framework (i.e., by the Scheduler at some point, but not necessarily immediately after the creation of the objects. The framework must now that proc_f must be resumed after the computation of proc_g1 and proc_g2 has completed. Therefore, some house-keeping has to be done, and this is taken care of by the function spawn() (inherited from Process_base). The complete translation of the calls to g1() and g2 looks as follows:

```
spawn( new proc_g1(i) );
spawn( new proc_g2(j) );
```

After these two calls to spawn(), proc_f itself should be suspended, and resumed again when both spawned Processes have finished. This is done by partitioning the computation of proc_f into *schedulable member functions*; these are member functions that take no parameters, and return void. These member functions must be *scheduled* (for instance, in the constructor of proc_f); the framework takes care of calling these member functions in the order in which they are scheduled. This is best illustrated by the complete code of proc_f:

```
template< class Traits >
class proc_f : public Traits::process_base_t {
  public:
    void memfun_1(){
      spawn( new proc_g1(i) );
      spawn( new proc_g2(j) );
    }

    void memfun_2(){
      i += j;
    }

    proc_f( process_base_t* parent ) // constructor
    : process_base_t( parent ), // initialize base class
    {
      schedule( &proc_f::memfun_1 );
      schedule( &proc_f::memfun_2 );
    }
```

```
  private: // data members
    int i, j;
};
```

When a `proc_f` object is created, its constructor is called, which schedules the two member functions; the type of the argument to `schedule` must be "pointer to member function of derived class of `Process_base`, taking no arguments, and returning void". The member functions of `proc_f` have the appropriate type, so we can indeed use pointers to these functions as arguments to `schedule`.

The two scheduled member functions are not executed immediately; they are called at a later time deemed right by the framework. So at some point, the framework calls the first scheduled member function, `memfun_1()`. This function creates two new processes, `proc_g1` and `proc_g2`. After the creation of these two Processes, the computation of `memfunc_1()` is finished, and `proc_f` is suspended. The constructors of the created Processes, in turn, schedule their own member functions. When all scheduled member functions of both `proc_g1` and `proc_g2` have been called (by the framework), these processes are no longer needed, and they are deleted by the framework. When all processes, created by `memfun_1()`, have been deleted, the scheduler resumes `proc_f` by calling the next scheduled member function, `memfun_2`. This member function created no further Processes, so after it has finished, the scheduler nows that `proc_f` is done (there are no further scheduled member functions), and the `proc_f` object (which must also have been created with `new`) is deleted by the scheduler.

Note that in the original function `f()`, the variables `i` and `j` were local. In the translated version, `proc_f`, they cannot be replaced by local variables of the member functions, since they must outlive these member functions. Therefore, they are turned into data members.

Note that schedulable member functions can also be scheduled from other member functions than the constructor. This allows for iteration, as will be illustrated in the next section.

5.1 Example: Quicksort

As noted before, in several applications of parametric search, the generic algorithm can be replaced by sorting. We therefore implemented Quicksort in the parametric search setting (i.e., with batching of comparisons, and suspension and resumption of computation) using the lower-level facilities of the framework. For users who implement their own generic algorithm or sorting algorithm, this illustrates the use of the lower-level facilities, so we will present the implementation in this section.

The general idea behind the use of Quicksort for sorting-based parametric search is that the comparisons in the partitioning step are independent, and hence, they can be batched. Recall that Quicksort sorts an array by choosing a pivot element, and then partitioning the array such that all elements less than the pivot come before the pivot, and all elements greater than the pivot come after it. Next, the sub-arrays (the part before the pivot and the part after it) are partitioned recursively.

Recursion is possible with our framework (Process objects can spawn Process objects of their own type), but we would advice against it, since the memory usage may be very large. Normally, the memory usage of recursion (i.e., the calling depth, or stack size) is linear in the *depth* of the recursion tree. However, using our framework, the memory consumption of recursion may be the *size* of the recursion tree, since the created Process objects stay alive as long as the framework deems necessary. Therefore, we suggest the users of our framework to replace recursion with iteration whenever possible; this is also what we do in our implementation of Quicksort.

The class `Quick_sorter` can be found in the files `quicksorter.{h,cpp}` in the main directory of the

framework. When we restrict our attention to the public member functions, the class looks as follows:

```
//-------------------------------------------------
template< class Traits, class BidirectionalIter >
class Quick_sorter : public Traits::process_base_t
//-------------------------------------------------
{
public: // member functions
  Quick_sorter( process_base_t* parent,
                BidirectionalIter first,
                BidirectionalIter last );
  ~Quick_sorter();
};
```

The constructor takes a pointer to the parent process (which may be null), and a pair of iterators indication the range of items to be sorted. These iterators may for instance be pointing into a `std::vector`, a `std::list`, or any other standard C++ container (or even a user-defined container, as long as it supports the requirements of a bidirectional iterator). The constructor copies the input items into a private `std::vector`; after sorting, the destructor (not shown in this document) copies the sorted items back into the input container.

The code for the constructor is as follows (initialization of data members is omited below):

```
//-------------------------------------------------
template< class Traits, class BidirectionalIter >
Quick_sorter< Traits, BidirectionalIter >::Quick_sorter
( process_base_t* parent,
  BidirectionalIter first, BidirectionalIter last )
//-------------------------------------------------
  : process_base_t( parent ),     // initialize base
    first_( first )               // save iterator
{
    // copy input in private std::vector, and initialize remaining
    // data members; not shown here

    schedule( &Quick_sorter< Traits, BidirectionalIter >::do_comparisons );
    schedule( &Quick_sorter< Traits, BidirectionalIter >::do_partition );
}
```

The two functions scheduled in the last two lines of the constructor are private member functions; we will discuss them below.

The data members of `Quick_sorter` are the following:

- `BidirectionalIter first_;`
  This is the iterator indicating the beginning of the sequence of items to be sorted. It is needed by the destructor, which copies back the sorted items into the original container.
- `std::vector< polynomial_t > polynomials_;`
  These are the items to be sorted. In the parametric-search setting, the items are always polynomials.

- `std::vector< wrapper< bool > > comparison_results_;`
  In the partitioning step, we don't swap items that are in the wrong order immediately. Instead, we do the partitioning in two passes: first, we compare all items with the pivot, and only after all comparisons have been resolved, we swap items based on the outcome of the comparisons. This vector is used to store the outcome of the comparisons. We would like to have used a simple `std::vector` of `bools`, but since `vector< bool >` is specialized in a somewhat strange way (see Meyers [4], item 18 "avoid using `vector< bool >`."), we have to wrap the `bools`. In any case, after all the comparisons have been resolved, the value of `comparison_results_[i]` is `true` if `polynomials_[i] < polynomials_[pivot]`, and `false` otherwise.

- `std::vector< std::pair< std::size_t, std::size_t > > segments_;`
  Since we replace the recursive partitioning of Quicksort by iteration, we need some way to indicate which sub-segments of the whole container `polynomials_` need to be partitioned. These sub-segments are stored in `segments_`; they are represented by pairs of indices, indicating for each sub-segment the first and the one-beyond-the-last item in `polynomials_` belonging to the sub-segment.

- `std::vector< std::pair< std::size_t, std::size_t > > new_segments_;`
  After one pass of partitioning, the set of sub-segments of `polynomials_` that are to be partitioned will change. Segments of size 1 are sorted and need no further consideration; other segments are split into two segments, that are dealt with in subsequent steps. The variable `new_segments_` is used to construct this updated set of sub-segments during an iterative step of the partitioning algorithm. After this step, `segments_` and `new_segments_` are swapped, so that `segments_` correctly represents the sub-segments to be partitioned in the next iterative step.

- `std::size_t input_size_;`
  This variable stores the size of the container of items that is to be sorted. It is used by the constructor for reserving sufficient space for the container data members (this is not strictly necessary, as STL containers may grow, but reserving the proper amount of space is generally much more efficient).

The partitioning of the segments of `polynomials_` is done by the private member functions `do_comparisons()` and `do_partitioning()` as follows. (See Figure 4; for simplicity, we replaced polynomials by integers, to make it easier for the reader to compare two items. One can think of these integers as the values of the polynomials at λ^*.):

- For each segment, we pick the polynomial element as pivot.

- All the polynomials of each segment are compared with the pivot of the segment. The results of these comparisons are stored in `comparison_results_`. If a polynomial is less than the pivot, the result of the comparison of that polynomial and the pivot is `true`; if the polynomial is greater than or equal to the pivot, the outcome of the comparison is `false`. The comparison step is done by the function `do_comparisons`. The actual comparisons of polynomials are performed by Comparison objects that are spawned; these are resolved at some later time, deemed right by the Scheduler. The `Quick_sorter` object is suspended after `do_comparisons()` has finished, i.e., after all Comparison objects have been spawned.

- When all spawned Comparison objects have performed their comparison, the scheduler tells the `Quick_sorter` object to execute its next scheduled member function, i.e., `do_partition()`. This member function swaps polynomials within segments, such that the segment is partitioned into two sub-segments. The first sub-segment contains only polynomials that are all less than the pivot; the second sub-segment contains only polynomials that are greater than or equal to the pivot. To decide whether it has to swap two polynomials, the function `do_partition()` uses `comparison_results_` to look-up in $O(1)$ time whether a polynomial is less than a pivot or not.

The code of `do_comparisons` is as follows:

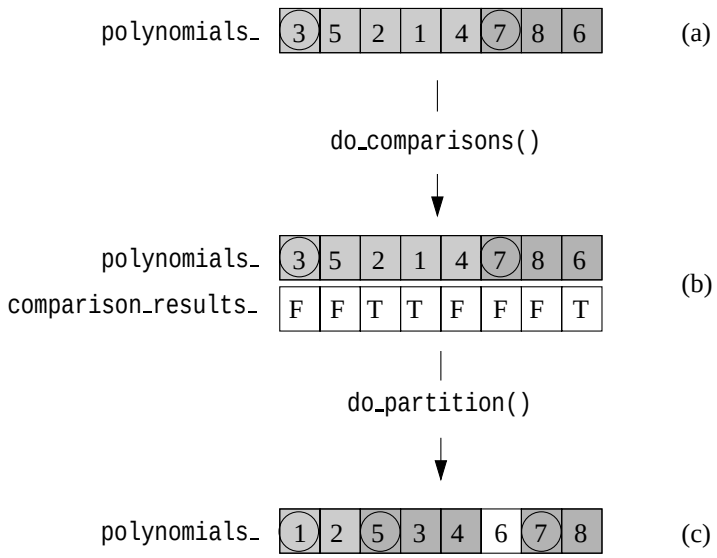

Figure 4: Two-phase partitioning; polynomials are denoted with integers. (a) two segments, each with a pivot (circled) (b) result of comparing all elements with their pivots (`true` if element is less than pivot, `false` otherwise) (c) swapping elements leads to new situation: each segment is split into two subsegments, each with its own pivot. Segments of size 1 are sorted and need no further consideration.

```
//-------------------------------------------------
template< class Traits, class BidirectionalIter >
void
Quick_sorter< Traits, BidirectionalIter >::do_comparisons()
//-------------------------------------------------
{
  typedef std::vector<
    std::pair< std::size_t, std::size_t > >::iterator iter_t;

  for( iter_t iter = segments_.begin(); iter != segments_.end(); ++iter )
    do_segment_comparisons( *iter );
}
```

In other words: for each segment, this function calls `do_segment_comparisons()`. The implementation of this function is straightforward: it compares each polynomial in the segment with the first polynomial in the segment (i.e., the pivot):

```
//-------------------------------------------------
template< class Traits, class BidirectionalIter >
void
Quick_sorter< Traits, BidirectionalIter >::do_segment_comparisons
( const std::pair< std::size_t, std::size_t >& segment )
//-------------------------------------------------
{
  for( std::size_t polynomial = segment.first;
       polynomial != segment.second; ++polynomial ){
    compare( polynomial, segment.first );
  }
}
```

The comparison of a polynomial and the pivot is delegated to the function compare:

```
//-------------------------------------------------
template< class Traits, class BidirectionalIter >
void
Quick_sorter< Traits, BidirectionalIter >::compare
( std::size_t polynomial, std::size_t pivot )
//-------------------------------------------------
{
  // easy case
  if( polynomial == pivot )
    comparison_results_[ polynomial ] = false;

  // too bad; we have to do real work
  else{
    spawn( comparison_t::lazy_create_comparison
           ( this,
             comparison_results_[ polynomial ],
             polynomials_[ polynomial ],
             polynomials_[ pivot ] ) );
  }
}
```

This function first checks an easy case: the comparison of the pivot with itself. Since the pivot is not less than the pivot, the result of this comparison is `false`.

The interesting case is taken care of by the static member function `lazy_create_comparison()` of the Comparison class. In general (this is problem-specific), this function first checks if the result of the comparison can be computed efficiently without actually creating a comparison object. If this is the case, then `lazy_create_comparison()` sets the second (by-reference) argument to the appropriate value. Otherwise, a Comparison object is created, which will evaluate the comparison of the last two arguments at a later time, at the command of the Scheduler.

When all spawned Comparison object have performed their comparison, the Scheduler calls the next scheduled member function, i.e., `do_partition()`. The implementation of this member function is rather technical and for the largest part not very interesting, thus we omit most of it it here. We do, however, want to discuss the following: after the partitioning of each segment, it is split into two sub-segments, containing polynomials that are smaller than, and greater than or equal to the pivot, respectively. Sub-segments of size one need no further processing, since they are done. Larger sub-segments have to be partitioned further, so if there are any such segments, the functions `do_comparisons()` and `do_partion()` need to be called again. This is taken care of by re-scheduling them at the end of `do_partition()`, if necessary:

```
if( !segments_.empty() ){
  schedule( &Quick_sorter< Traits, BidirectionalIter >::do_comparisons );
  schedule( &Quick_sorter< Traits, BidirectionalIter >::do_partition );
}
```

This completes the discussion of the implementation of Quicksort in the sorting-based parametric-search setting.

References

[1] H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *Internat. J. Comput. Geom. Appl.*, 5:75–91, 1995.

[2] R. Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. ACM*, 34(1):200–208, 1987.

[3] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, 1983.

[4] S. Meyers. *Effective STL.* Addison-Wesley, Boston, 2001.

[5] R. van Oostrum and R.C. Veltkamp. *Parametric search made practical.* Technical Report UU-CS-2002-050, Utrecht University, 2001.
URL: `http://www.cs.uu.nl/research/techreps/UU-CS-2002-050.html`.

[6] R. van Oostrum and R.C. Veltkamp. Parametric search made practical. To appear in a special issue of *Computational Geometry: Theory and Applications.*