Compute a convex hull via randomized incremental construction in CGAL

Thomas Low*

June 16, 2008

Abstract This paper contains an implementation and visualization of how to compute a convex hull via randomized incremental construction described in a paper by Kettner and Welzl [KW98]. It also illustrates the conservative property of this implementation.

Contents

1 Introduction 2

2 Randomized incremental construction 2
   2.1 The history graph 2
   2.2 Traversing the convex points 7
   2.3 The whole algorithm 10
   2.4 The program 12

3 Visualization 13
   3.1 Generating points 13
   3.2 Computing the convex hull 13
   3.3 Drawing a convex hull 16
   3.4 Creating a QT window 17
   3.5 The program 20

4 References 21

A The makefile 22

B Generating points 24

*Department of Simulation and Graphics, Faculty of Computer Science, Otto von Guericke University Magdeburg, Germany. thomas.low@st.ovgu.de
1 Introduction

There are many different ways on how to find a convex hull from a set of points. Computing them via randomized incremental construction with a conservative predicate, as described by Kettner and Welzl [KW98], preserves a conservative property. That means a predicate returns true only if the exact predicate is true. So the set of convex points, computed by the algorithm, contains at least every exact convex point and maybe other inner points. Now this hopefully considerably smaller set can be easily proved for correctness or be repaired with an exact algorithm like Andrew’s variant of the Graham scan algorithm. [And79]

We provide such a conservative implementation using CGAL in chapter 2 and want to illustrate their property by visualizing a convex hull in a QT window in chapter 3.

2 Randomized incremental construction

2.1 The history graph

The randomized incremental construction algorithm uses a data structure, so-called history graph, to compute separately both lower and upper convex hull of a set of points. It does almost all work. After inserting each point, we can easily extract the convex ones by iterating through him.

Basically the history graph describes a convex polygon around $n$ points, starting with just one edge between the lexicographically smallest and largest point. By inserting the $n + 1$-th point we expand the polygon, if it is not an inner point. Note that an old edge will not be removed. Now the history graph describes a convex polygon around $n + 1$ points. So the key point is a fast decision and expansion.

That is why we have to take a deeper look at the graph. Its basic component is an edge between two points. This edge contains pointers to the two points, its previous edge, its next edge and a boolean, which decides whether it is an inner one or not.

\[
\text{\textit{edge description}}
\]

\begin{verbatim}
template<class Point>
struct Edge {
    const Point *prv_p;
    const Point *nxt_p;
    Edge *prv_e;
    Edge *nxt_e;
    bool inner;
    Edge(const Point *prv, const Point *nxt) :
        prv_p(prv), nxt_p(nxt),
        prv_e(0), nxt_e(0), inner(false) {}
    ~Edge() {}
};
\end{verbatim}
Now we want to look at the most important part of the algorithm, the insertion of a point. At first we have to decide, whether the point is an inner point or not. Therefore we sequentially look at an edge, starting with the edge between the lexicographically smallest and largest point. If the point is left of this edge, we discard it. If not, we walk along the inner edges of the history graph towards the border. In each step we decide which next edge we should choose, according to the relative position between the point and an upper edge. If the considered edge lies on the border of the history graph and the point is left of this edge, we know that the point is outside of the current graph, so we insert him.

If our predicate is conservative, we only state that a point is left of the current edge, if we know that for sure. So we could make mistakes when the point is not left of the current edge, when he is inside of the convex hull.

```
3 insert point into history graph 3)
   Edge<Point> e = edges[0];
   while(!leftturn(*(e->prv_p),*(e->nxt_p), *p)) {
     if(e->inner) {
       if(e->prv_e != 0 && lessxy(*p, *(e->prv_e->nxt_p)) ) {
         e = e->prv_e;
       } else {
         e = e->nxt_e;
       }
     } else {
       if(!equal(*p,*(e->prv_p)) && !equal(*p,*(e->nxt_p))){
         update history graph 4a
       }
       break;
     }
   }
```
Now the considered edge becomes an inner edge and two new edges arise an extended border of the history graph. We only have to compute where we should attach these edges to ensure that the border remains convex. Therefore we start at our considered edge and go through the border of the history graph - once for the left side and once for the right side. When we find the first point so that the inserted edge will lead to a convex graph, we stop and insert the edge at this position. After that we have finished the updating. Now our new history graph contains an additional point and is still convex.

4a  \((update\ text\ graph\ 4a)\equiv\)

\[
\begin{align*}
\text{Edge<Point> } & \*left = e->prv\_e; \\
\text{Edge<Point> } & \*right = e->nxt\_e; \\
\text{edges.push}\_\text{back} & (\text{new } \text{Edge<Point>}(0,p)); \\
\text{edges.push}\_\text{back} & (\text{new } \text{Edge<Point>}(p,0)); \\
\text{Edge<Point> } & \*e1 = \text{edges[edges.size()-2]}; \\
\text{Edge<Point> } & \*e2 = \text{edges[edges.size()-1]}; \\
\text{e->inner } & = \text{true}; \\
\text{e->prv}\_e & = \*e1; \\
\text{e->nxt}\_e & = \*e2;
\end{align*}
\]

\((update\ text\ graph\ on\ the\ left\ side\ 4b)\)

\((update\ text\ graph\ on\ the\ right\ side\ 5)\)

Here you can see an algorithm to compute the insert position of the new edges. In each step it checks whether an edge between the current position and the point we want to insert would harm the convex property. If we continue with the next edge, we just have to mark the one before as an inner edge.

4b  \((update\ text\ graph\ on\ the\ left\ side\ 4b)\equiv\)

\[
\begin{align*}
\text{Edge<Point> } & \*tmp = e; \\
\text{e1->prv\_e } & = e->prv\_e; \\
\text{while} & (left != 0 \&\& \text{leftturn}\(*p, *(left->nxt\_p), *(left->prv\_p))) \{ \\
& left->inner = \text{true}; \\
& tmp = left; \\
& left = left->prv\_e; \\
& tmp->prv\_e = \*e1; \\
& tmp->nxt\_e = 0;
\}
\] if(left != 0) left->nxt\_e = e1;

\text{e1->prv}\_p & = tmp->prv\_p; \\
\text{e1->nxt}\_e & = \*e2; \\
\text{e1->prv}\_e & = left;
\]
The following code is the same algorithm for the right side.

(update history graph on the right side)

```c
5
 tmp = e;
e2->nxt_e = e->nxt_e;
while(right != 0 && leftturn(*p, *(right->nxt_p), *(right->prv_p))) {
    right->inner = true;
    tmp = right;
    right = right->nxt_e;
    tmp->prv_e = e2;
    tmp->nxt_e = e2;
}
if(right != 0) right->prv_e = e2;

e2->nxt_p = tmp->nxt_p;
e2->prv_e = e1;
e2->nxt_e = right;
```
Now we know how to insert a point into the history graph. But we also have to provide a few other operations. Therefore we look at the whole history graph class. It provides methods to create, destroy and return the size of our datastructure. It also defines a vector of pointers of edges to store the graph. Additionally we have to make an iterator available, which goes through the convex points, but we will discuss that in the next section.

```cpp
6 ⟨history graph 6⟩≡

template<class Kernel, class Point, class Less_xy_2>
class HistoryGraph {
    public:
        typedef HistoryGraphIterator<Point> vertex_const_iterator;

        HistoryGraph(const Point *p1, const Point *p2, const Less_xy_2& plessxy)
        : lessxy(plessxy) {
            typename Kernel::Left_turn_2 leftturn = kernel.left_turn_2_object();
            typename Kernel::Equal_2 equal = kernel.equal_2_object();
            edges.push_back(new Edge<Point>(p1, p2));
        }

        ~HistoryGraph() {
            typename std::vector<Edge<Point>* >::iterator edge_iterator =
                edges.begin();
            while(edge_iterator != edges.end()) {
                delete *edge_iterator;
                ++edge_iterator;
            }
        }

        void insert(const Point *p) {
            ⟨insert point into history graph 3⟩
        }

        vertex_const_iterator begin() {
            ⟨initialize iterator begin 7a⟩
        }

        vertex_const_iterator end() {
            ⟨initialize iterator end 7b⟩
        }

        int size() {
            return edges.size();
        }

    private:
        std::vector<Edge<Point>* > edges;
        Kernel kernel;
        typename Kernel::Left_turn_2 leftturn;
        Less_xy_2 lessxy;
        typename Kernel::Equal_2 equal;
};
```
2.2 Traversing the convex points

To find the convex points in our history graph we just have to follow the border of it. We know that both the lexicographically smallest and largest point are convex points. That is why they are both ends of this border. So we start with the smallest point and go through all incident edges until we find the last one. This is the edge to the second convex point. We call it \textbf{first}. Every next edge in this sequence leads to another convex point until we found the lexicographically largest one.

To do this comfortable, we have to provide a bidirectional iterator. The iterator which points to the beginning of our sequence will be initialized with the \textbf{first} edge. The iterator which points to the end will be initialized with the successor of the last element - in our case a zero pointer. Its position inside the history graph is well-defined by two pointers to the convex hull edges before and behind the current point.

\begin{itemize}
\item\textbf{initialize iterator begin 7a} ≡ Edge<Point> *first = edges[0];
\textbf{while(}first->prv_e != 0\textbf{) first = first->prv_e; return HistoryGraphIterator<Point>(first,0,first);} \\
\item\textbf{initialize iterator end 7b} ≡ Edge<Point> *first = edges[0];
\textbf{while(}first->prv_e != 0\textbf{) first = first->prv_e; return HistoryGraphIterator<Point>(first,0,0);} \\
\item\textbf{constructor of the history graph 7c} ≡ HistoryGraphIterator(Edge<Point>* first_e, Edge<Point>* before_e,
\textbf{Edge<Point>* current_e}) : first(first_e), before(before_e),
\textbf{current(current_e)} {
\textbf{last = first_e};
\textbf{while(}last->nxt_e != 0\textbf{) last = last->nxt_e;} 
}
\item\textbf{increment iterator 7d} ≡ \textbf{if(}current != 0\textbf{) }{
\textbf{before = current;} 
\textbf{current = current->nxt_e;} 
\textbf{}} \textbf{else if(}before != 0\textbf{) }{
\textbf{before = 0;}} \textbf{else }{
\textbf{current = first;}}
\end{itemize}
We do the same with decrementing the iterator. If we are one position behind the last element, we set the current position to last.

8a  \[\text{decrement iterator 8a)}\equiv \]
\[
\text{if}(\text{before} \neq 0) \{ \\
\quad \text{current} = \text{before}; \\
\quad \text{before} = \text{before} \rightarrow \text{prv}_e; \\
\} \text{ else if}(\text{current} \neq 0) \{ \\
\quad \text{current} = 0; \\
\} \text{ else } \\
\quad \text{before} = \text{last}; \\
\}
\]

According to our position in the sequence of edges we return a convex point. Because every edge has two points, we have to ensure, that we also return the last point.

8b  \[\text{dereference iterator 8b)}\equiv \]
\[
\text{if}(\text{current} \neq 0) \{ \\
\quad \text{return } *(\text{current} \rightarrow \text{prv}_p); \\
\} \text{ else } \\
\quad \text{return } *(\text{before} \rightarrow \text{nxt}_p); \\
\}
\]

The following code compares two iterators on the basis of their current position of the sequence consisting of the two edge pointers current and before.

8c  \[\text{iterator equality test 8c)}\equiv \]
\[
\text{return } (\text{this} \rightarrow \text{current} == \text{other} \rightarrow \text{current} \\
\quad \&\& \text{this} \rightarrow \text{before} == \text{other} \rightarrow \text{before});
\]
Now we just have to turn this basic operations into a bidirectional iterator. Therefore we use the class `iterator_facade` of the Boost C++ Libraries. [ASW06] We only have to derive our iterator from the class `iterator_facade` and it becomes a standard conforming bidirectional iterator.

```
9 (history graph iterator 9)≡
template<class Point>
class HistoryGraphIterator :
  public boost::iterator_facade<
    HistoryGraphIterator<Point>
  , const Point
  , boost::bidirectional_traversal_tag
> {
  public:
    HistoryGraphIterator() {}
    explicit HistoryGraphIterator(Point p) {}

    (constructor of the history graph 7c)

  private:
    friend class boost::iterator_core_access;

    void increment() {  
      (increment iterator 7d)
    }

    void decrement() {  
      (decrement iterator 8a)
    }

    bool equal(HistoryGraphIterator const& other) const  
    {  
      (iterator equality test 8c)
    }

    const Point& dereference() const {  
      (dereference iterator 8b)
    }

    Edge<Point>* first;
    Edge<Point>* before;
    Edge<Point>* current;
    Edge<Point>* last;
};
```
2.3 The whole algorithm

Now we are looking at the whole algorithm. A forward iterator provides our set of points. To initialize the history graph, we have to compute the lexicographically smallest and largest point in the sequence first. If they are the same or there is only one point, we are finished.

10a (compute smallest and largest point 10a)≡
\[
\begin{align*}
\text{std::pair<ForwardIterator, ForwardIterator>& min_max_pair = min_max_element(begin, end, lessxy, lessxy);} \\
\text{if(begin == end) return result;}
\end{align*}
\]
\[
\text{if(equal(*min_max_pair.first, *min_max_pair.second)) {}
\]
\[
\begin{align*}
*\text{result} &= *\text{min_max_pair.first}; \\
\text{return ++result;}
\end{align*}
\]

Once we have these two points we now can initialize the history graph with them. As mentioned before the history graph only computes the lower or upper convex hull. That is why we need to create two of them. The second one should be initialize mirrored. That means the lexicographically smallest point is now the largest and the other way round. We also have to mirror our ordering predicate, which we call lessxy_swap.

10b (create history graph 10b)≡
\[
\begin{align*}
\text{HistoryGraph<Kernel, Point, Less_xy_2> lower_hg = HistoryGraph<Kernel, Point, Less_xy_2>}(\&(*\text{min_max_pair.first}), \\
\&(*\text{min_max_pair.second}), \text{lessxy}); \\
\text{HistoryGraph<Kernel, Point, Less_xy_2_swap> upper_hg = HistoryGraph<Kernel, Point, Less_xy_2_swap>}(\&(*\text{min_max_pair.second}), \\
\&(*\text{min_max_pair.first}), \text{lessxy_swap});
\end{align*}
\]

Now it is time to insert all of our points one after another into both history graphs. While inserting points the algorithm does some restructuring as discussed before.

10c (insert points into history graph 10c)≡
\[
\begin{align*}
\text{while(begin != end) {}
\end{align*}
\]
\[
\begin{align*}
\text{lower_hg.insert}(\&(*\text{begin}); \\
\text{upper_hg.insert}(\&(*\text{begin}); \\
\text{begin++;}
\end{align*}
\]

After that we only have to extract the computed points from both history graphs. Therefore we are using an iterator, which goes through the graph and copies the convex points to the output iterator. To repair the computed points before returning them, we use Andrew’s variant of the Graham scan algorithm from CGAL. [And79]

10d (repair and copy the convex hull 10d)≡
\[
\begin{align*}
\text{if(Repair) {}
\end{align*}
\]
\[
\begin{align*}
\text{CGAL::Filtered_kernel_adaptor<Kernel> exact_kernel;}
\text{ch_graham_andrew_scan(upper_hg.begin(), upper_hg.end(), result, exact_kernel);}
\text{ch_graham_andrew_scan(lower_hg.begin(), lower_hg.end(), result, exact_kernel);}
\end{align*}
\]
\[
\text{else {}
\text{std::copy(lower_hg.begin(), --(lower_hg.end())), result);}
\text{std::copy(upper_hg.begin(), --(upper_hg.end())), result);}
\]
Before putting all together in one method, we define the predicates used above. Now we have a method to compute the convex hull via randomized incremental construction.

11a
\(\text{compute convex hull}\) 11a

\[
\begin{align*}
\text{template}< & \text{class Kernel, class Point, bool Repair, class ForwardIterator,} \\
& \text{class OutputIterator}> \\
\text{OutputIterator} & \\
\text{compute_convex_hull}(\text{ForwardIterator} \text{ begin}, \text{ForwardIterator} \text{ end},} \\
& \text{OutputIterator} \text{ result}) \\
& \{
\begin{align*}
\text{typedef typename Kernel::Less_xy_2} & \quad \text{Less_xy_2;} \\
\text{typedef typename Swap<Less_xy_2,1>::Type} & \quad \text{Less_xy_2_swap;}
\end{align*}
\end{align*}
\]

Kernel kernel;
\text{typename Kernel::Equal_2} \quad \text{equal} = \text{kernel.equal_2\_object}();
\text{Less_xy_2} \quad \text{lessxy} = \text{kernel.less_xy_2\_object}();
\text{Less_xy_2\_swap} \quad \text{lessxy\_swap} = \text{swap\_1(lessxy)};

10a
\text{(compute smallest and largest point)}

10b
\text{(create history graph)}

10c
\text{(insert points into history graph)}

10d
\text{(repair and copy the convex hull)}

return result;
\}

By implementing some wrapper functions, the user has the ability to choose a specific kernel and whether the repairing will be performed.

11b
\(\text{compute convex hull} 11a\) + 11b

\[
\begin{align*}
\text{template}< & \text{class Kernel, class ForwardIterator, class OutputIterator}> \\
\text{OutputIterator} & \\
\text{ch_randomized_incremental_construction}(\text{ForwardIterator} \text{ begin},} \\
& \text{ForwardIterator} \text{ end, OutputIterator} \text{ result}) \\
& \{
\begin{align*}
\text{typedef typename std::iterator\_traits<ForwardIterator>::value\_type} & \quad \text{Point;}
\text{return compute\_convex\_hull<Kernel, Point, false>(begin, end, result);} \\
\end{align*}
\end{align*}
\]

11c
\(\text{compute convex hull} 11a\) + 11c

\[
\begin{align*}
\text{template}< & \text{class Kernel, class ForwardIterator, class OutputIterator}> \\
\text{OutputIterator} & \\
\text{ch_randomized_incremental_construction\_repaired}(\text{ForwardIterator} \text{ begin},} \\
& \text{ForwardIterator} \text{ end, OutputIterator} \text{ result}) \\
& \{
\begin{align*}
\text{typedef typename std::iterator\_traits<ForwardIterator>::value\_type} & \quad \text{Point;}
\text{return compute\_convex\_hull<Kernel, Point, true>(begin, end, result);} \\
\end{align*}
\end{align*}
\]
2.4 The program

Finally we have to include some header files of CGAL and put all together.

```cpp
#ifndef CH_RANDOMIZED_INCREMENTAL_CONSTRUCTION_H
#define CH_RANDOMIZED_INCREMENTAL_CONSTRUCTION_H

#include <CGAL/basic.h>
#include <CGAL/Simple_cartesian.h>
#include <CGAL/algorithm.h>
#include <CGAL/functional_base.h>
#include <CGAL/ch_graham_andrew.h>
#include <CGAL/convexity_check_2.h>
#include <boost/iterator/iterator_facade.hpp>

using namespace CGAL;
```

```cpp
#endif //CH_RANDOMIZED_INCREMENTAL_CONSTRUCTION_H
```
3 Visualization

3.1 Generating points

Now we want to visualize a convex hull. So we have to generate some example points, compute their convex hull and draw them into a QT window. In order to generate points we use a slightly different version of the point generator from Tusch and Schirra [ST07]. In our case we do not need to ensure that every point is unique. You can find this variant in appendix B.

The generator provides four parameters. The first one states the radius of the circle, where the points are created. The next three arguments describe the number of extrem points, the number of points lying on the border and the number of points lying inside the convex hull.

\[\text{generate new points}\]
\[
\begin{verbatim}
input_points.clear();
random_degenerate_extreme_points_2<Point>(320,n_e,n_d,n_i,
   std::back_inserter(input_points));
\end{verbatim}
\]

3.2 Computing the convex hull

To illustrate the conservative property of this implementation, we compute three different versions of convex hulls. Therefore we use an exact, a probabilistic and a conservative kernel.

Obviously the exact kernel computes the correct convex hull. Our probabilistic kernel makes a mistake in 50 percent of all true answers. That means sometimes he shifts true to false. So he is conservative too.

\[\text{probabilistic kernel}\]
\[
\begin{verbatim}
template <class Kernel, int probability>
class ProbabilisticKernel : public Kernel {
public:
   struct Left_turn_2 {
      typedef typename Kernel::Point_2 Point_2;
      typedef Arity_tag< 3 > Arity;

      bool operator()(const Point_2 &p, const Point_2 &q, const Point_2 &r)const{
         Kernel kernel;
         typename Kernel::Left_turn_2 leftturn = kernel.left_turn_2_object();

         const bool lt = leftturn(p,q,r);
         if(lt){
            const int rand = default_random.get_int(0,100);
            if(rand < probability) return false;
         }
         return lt;
      }
   }

   Left_turn_2 left_turn_2_object() const{
      return Left_turn_2();
   }
};
\end{verbatim}
\]
The conservative kernel states always \textbf{false}, if the value is inside the error bound computed by Shewchuk.\cite{Shewchuk97} If we are sure, that our calculations are right, we return the computed answer. That means we can only make a mistake by shifting a \textbf{true} to a \textbf{false}.

\begin{verbatim}
conservative predicate kernel 14a)≡

```
template <class Kernel>
class Conservative_predicate_kernel : public Kernel {
public:
    struct Left_turn_2 {
        typedef typename Kernel::Point_2 Point_2;
        typedef Arity_tag< 3 > Arity;

        bool operator()(const Point_2 &p, const Point_2 &q, const Point_2 &r)const{
            const double ccwerrboundA = 3.33066907387547159342e-16;
            double detleft, detright, det;
            double detsum, errbound;
            detleft = (p.x() - r.x()) * (q.y() - r.y());
            detright = (p.y() - r.y()) * (q.x() - r.x());
            det = detleft - detright;
            detsum = detleft + detright;
            errbound = ccwerrboundA * detsum;
            if((det >= errbound) || (-det >= errbound)) {
                return (det > 0);
            } else {
                return false;
            }
        }

        Left_turn_2 left_turn_2_object() const{ return Left_turn_2(); }
    }
};
```

Now we define the kernels described above and some abbreviations.

\begin{verbatim}
(type definitions 14b)≡

```
typedef Simple_cartesian<double> Base_Kernel;
typedef Conservative_predicate_kernel<Base_Kernel> Conservative_Kernel;
typedef ProbabilisticKernel<Base_Kernel, 50 > Prob_Kernel;
typedef CGAL::Filtered_kernel_adaptor<Base_Kernel> Exact_Kernel;
typedef Base_Kernel::Point_2 Point;
typedef Polygon_2<Base_Kernel> Polygon;
typedef std::vector<Point> Point_Vector;
```
The following code calls our convex hull algorithm for every kernel and stores the result in three polygons.

(\textit{call convex hull algorithm }15)\equiv

\begin{verbatim}
prob_polygon.clear();
ch_randomized_incremental_construction<Prob_Kernel>(input_points.begin(),
    input_points.end(), std::back_inserter(prob_polygon));

cons_polygon.clear();
ch_randomized_incremental_construction<Conservative_Kernel>(
    input_points.begin(), input_points.end(),
    std::back_inserter(cons_polygon));

exact_polygon.clear();
ch_randomized_incremental_construction_repaired<Exact_Kernel>(
    input_points.begin(), input_points.end(),
    std::back_inserter(exact_polygon));

assert(CGAL::is_ccw_strongly_convex_2(exact_polygon.vertices_begin(),
    exact_polygon.vertices_end(),Exact_Kernel()));
\end{verbatim}
3.3 Drawing a convex hull

Now we can draw these polygons and their extreme points into a QT widget. Each polygon will be drawn in an unique color to illustrate their differences. We also draw the input points in a light gray.

```cpp
(widget->clear();
(widget->lock();

*widget << CGAL::PLUS << Color(220,220,220);
std::copy(input_points.begin(),input_points.end(),
          Ostream_iterator<Point, Qt_widget>(*widget));

if(cb_choice->currentItem() == 2) {
  *widget << Color(255,50,50);
} else {
  *widget << Color(255,220,220);
}
*widget << exact_polygon;

if(cb_choice->currentItem() == 0) {
  *widget << Color(50,50,255);
  *widget << prob_polygon;
  std::copy(prob_polygon.vertices_begin(), prob_polygon.vertices_end(),
            Ostream_iterator<Point, Qt_widget>(*widget));
}

if(cb_choice->currentItem() == 1) {
  *widget << Color(30,180,30);
  *widget << cons_polygon;
  std::copy(cons_polygon.vertices_begin(), cons_polygon.vertices_end(),
            Ostream_iterator<Point, Qt_widget>(*widget));
}
*widget << Color(255,50,50);
std::copy(exact_polygon.vertices_begin(), exact_polygon.vertices_end(),
          Ostream_iterator<Point, Qt_widget>(*widget));
(widget->unlock();
```
3.4 Creating a QT window

Finally we have to create a QT window with input boxes for the generator arguments, a choice box to change between the three kernels and an update button. Furthermore we have to connect events, like changed text in one of the input boxes or a resized window, to their corresponding update methods.

```cpp
17 (my window 17)≡
class Ric_window : public QMainWindow {
    Q_OBJECT
    (type definitions 14b)
public:
    Ric_window(int x, int y) {
        setCaption("Convex hull via recursive incremental construction");
        resize(x, y);

        widget = new CGAL::Qt_widget(this);
        widget->set_window(-x/2, x/2, -y/2, y/2);
        widget->show();
        setCentralWidget(widget);

        n_e = 10;
        n_d = 20;
        n_i = 300;

        (add toolbars 19a)

        connect(widget, SIGNAL(redraw_on_back()),
                this, SLOT(redraw_win()));
        connect(button_generate, SIGNAL(clicked()),
                this, SLOT(generate_new_points()));
        connect(edit_n_e, SIGNAL(textChanged(const QString &)),
                this, SLOT(updateParameters(const QString &)));
        connect(edit_n_d, SIGNAL(textChanged(const QString &)),
                this, SLOT(updateParameters(const QString &)));
        connect(edit_n_i, SIGNAL(textChanged(const QString &)),
                this, SLOT(updateParameters(const QString &)));
        connect(cb_choice, SIGNAL(activated(int)),
                this, SLOT(redraw_win()));
    }

    void show() {
        QMainWindow::show();
        generate_new_points();
    }

private slots:
    void redraw_win() {
        (redraw window 16)
    }

    void generate_new_points() {
        (generate new points 13a)
        (call convex hull algorithm 15)
    }
```
redraw_win();
}
void updateParameters(const QString &)
{
  ⟨update drawing parameters 19b⟩
};

private:
  Point_Vector input_points;
  Polygon prob_polygon, cons_polygon, exact_polygon;
  int n_e, n_d, n_i;
  Qt_widget *widget;
  QToolBar *generate_toolbar, *choice_toolbar;
  QLineEdit *edit_n_e, *edit_n_d, *edit_n_i;
  QLabel *label_n_e, *label_n_d, *label_n_i, *label_choice;
  QPushButton *button_generate;
  QComboBox *cb_choice;
  Qt_widget_standard_toolbar *std_toolbar;
};
Here you can see the creation of three toolbars. The first one is a standard toolbar which enables the user to zoom and drag. The second one provides the ability to change the generator arguments. And the third toolbar adds a combobox to change between the three kernels.

```cpp
19a  (add toolbars 19a)≡
std_toolbar = new CGAL::Qt_widget_standard_toolbar(widget, this,
 "Standard Toolbar”);
generate_toolbar = new QToolBar("Generation Toolbar", this, this, true);
choice_toolbar = new QToolBar("Kernel Choice Toolbar", this);

QString str;
label_n_e = new QLabel(" n_e: ", generate_toolbar);
drag_n_e = new QLineEdit(generate_toolbar);
drag_n_e->setFixedWidth(50);
drag_n_e->setText(str.setNum(n_e));

label_n_d = new QLabel(" n_d: ", generate_toolbar);
drag_n_d = new QLineEdit(generate_toolbar);
drag_n_d->setFixedWidth(50);
drag_n_d->setText(str.setNum(n_d));

label_n_i = new QLabel(" n_i: ", generate_toolbar);
drag_n_i = new QLineEdit(generate_toolbar);
drag_n_i->setFixedWidth(50);
drag_n_i->setText(str.setNum(n_i));
generate_toolbar->addSeparator();
button_generate = new QPushButton("generate", generate_toolbar);

label_choice = new QLabel(" Kernel : ", choice_toolbar);
cb_choice = new QComboBox(false, choice_toolbar);
cb_choice->insertItem("probabilistic");
cb_choice->insertItem("conservative");
cb_choice->insertItem("exact only");

19b  (update drawing parameters 19b)≡
str = edit_n_e->text();
n_e = std::max(3,str.toInt());
str = edit_n_d->text();
n_d = std::max(0,str.toInt());
str = edit_n_i->text();
n_i = std::max(0,str.toInt());

Finally we have to update the generator arguments from their corresponding input box.

19c  (create window 19c)≡
QApplication app(argc, argv);
Ric_window *window = new Ric_window(700,700);
app.setMainWidget(window);
window->show();
return app.exec();
```
3.5 The program

In order to make the CGAL and QT features used above available, we have to include some header files.

```cpp
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <stdlib.h>
#include "random_degenerate_extreme_points_2.h"
#include "ch_recursive_incremental_construction.h"

#include <CGAL/Random.h>
#include <CGAL/Polygon_2.h>

#include <CGAL/IO/Qt_widget_Polygon_2.h>
#include <CGAL/IO/Qt_widget_standard_toolbar.h>
#include <CGAL/IO/Color.h>
#include <CGAL/IO/Ostream_iterator.h>

#include <qapplication.h>
#include <qmainwindow.h>
#include <qtoolbar.h>
#include <qlineedit.h>
#include <qpushbutton.h>
#include <qlabel.h>
#include <qstring.h>
#include <qcombobox.h>

using namespace CGAL;
```

Now we can combine the code chunks in the order required by the compiler.

```cpp
#include "ric.moc"

int main(int argc, char **argv)
{
    #ifndef CGAL_USE_QT
        std::cout << "Sorry, this program needs QT \n";
    #else
        #create window 19c
    #endif
    return 0;
}
```
4 References


A The makefile

For the sake of completeness, here comes a makefile. Just typing make will create the executable and the documentation in .dvi format. For the first time, you have to extract the makefile “manually”.

notangle -t8 -R’makefile’ ric.nw > makefile

From there on, you can use the makefile, for example to extract the makefile by make makefile.

22a (makefile 22a)≡

all: ric.dvi ric
more: ric.ps ric.pdf ric

This document is transformed into a \TeX input file using noweave and then \TeX is used to create the documentation in .dvi or alternatively .ps or .pdf format.

22b (makefile 22a)+≡

PDFIMAGES =
EPSIMAGES =

%.tex: %.nw
   noweave -index -delay $< | cpif $@

%.dvi: %.tex %.bib $(EPSIMAGES)
   latex $(patsubst %.tex,%,<$)
   bibtex $(patsubst %.tex,%,<$)
   latex $(patsubst %.tex,%,<$)
   latex $(patsubst %.tex,%,<$)
   latex $(patsubst %.tex,%,<$)

%.ps: %.dvi
dvips $<

%.pdf: %.tex %.bib $(PDFIMAGES)
   pdflatex $(patsubst %.tex,%,<$)
   bibtex $(patsubst %.tex,%,<$)
   pdflatex $(patsubst %.tex,%,<$)
   pdflatex $(patsubst %.tex,%,<$)
   pdflatex $(patsubst %.tex,%,<$)
   pdflatex $(patsubst %.tex,%,<$)
   epstopdf $<

22
The command `notangle` is used to extract code chunks into files. This is done for the files `ric.cpp` and `makefile`. For the former, the `-L` options adds line references to the extracted code such that e.g. a debugger can reference to a line in the `.nw` file! For the latter, the option `-t8` ensures that tabs survive the extraction process.

```makefile
SOURCES = ric.cpp ch_recursive_incremental_construction.h\ 
  random_degenerate_extreme_points_2.h
NWFILE = ric.nw
NWLINE = -L

makefile: $(NWFILE)
  notangle -t8 -R'makefile' $< | cpif $@

$(SOURCES): $(NWFILE)
  notangle $(NWLINE) -R'$(subst _,\_,$@)' $< | cpif $@

code:
  touch $(NWFILE)
  $(MAKE) $(SOURCES)

nicecode:
  touch $(NWFILE)
  $(MAKE) $(SOURCES) -e NWLINE=""

The extracted files are compiled and linked using a CGAL makefile.

```makefile```
B  Generating points

```cpp
#ifndef RANDOM_DEGENERATE_EXTREME_POINTS_2_H
#define RANDOM_DEGENERATE_EXTREME_POINTS_2_H

#include <CGAL/basic.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/ch_akl_toussaint.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <vector>
#include <algorithm>
using namespace CGAL;

template <class Point_2,class OutputIterator>
OutputIterator random_degenerate_extreme_points_2(double r, int n_e, int n_d, int n_i,
                                                OutputIterator out) {
    assert(r > 0.0);
    assert(n_e >= 3);
    assert(n_d >= 0);
    assert(n_i >= 0);

    typedef typename Kernel_traits<Point_2>::Kernel Kernel;
    typedef Filtered_kernel_adaptor<Kernel> Exact_predicate_kernel;
    typedef std::vector<Point_2> Point_set;

    Exact_predicate_kernel exact_traits;
    std::vector<Point_2> S(n_d+n_e+n_i);
    Point_set uniques;
    Random_points_on_circle_2<Point_2> on_circle(r);
    typedef std::vector<Point_2>::iterator hull_end,last;
    const double circle_area = r * r * CGAL_PI;
    const double MIN_TRIANGLE_AREA = (n_e == 3) ? circle_area * 0.1 : 0.0;
    double area=0.0;
    typedef Triangle_2<Kernel> Triangle;
    while( !(area > MIN_TRIANGLE_AREA) ){
        last = S.begin();
        hull_end = last + n_e;
        uniques.clear();
        while(last != hull_end){
            *last = *on_circle++;
            uniques.push_back(*last);
            ++last;
        }
        area = std::abs(Triangle(S[0],S[1],S[2]).area());
    }
```

hull_end = ch_akl_toussaint(uniques.begin(), uniques.end(), S.begin(),
    exact_traits);
last = hull_end;
int e = std::distance(S.begin(), hull_end);

double overall_length = 0.0;
for(int s = 0; s < e; ++s)
    overall_length += std::sqrt(double(squared_distance(S[s], S[(s+1)%e])));

Random random_number_generator(rand());
double increment = overall_length / double(n_d);
double tick = random_number_generator.get_double() * increment;
double slice = 0.0;

for(int s = 0, d = 0; d < n_d; ++s, s %= e ) {
    Random_points_on_segment_2<Point_2> on_segment(S[s], S[(s+1)%e]);
    slice += std::sqrt(double(squared_distance(S[s], S[(s+1)%e])));
    for(; d < n_d && tick < slice; tick += increment){
        *last = *on_segment++;
        uniques.push_back(*last);
        ++last;
        ++d;
    }
}

unsigned int saved_seed = rand();
typedef Delaunay_triangulation_2<Exact_predicate_kernel> PointLocation;
typedef typename PointLocation::Locate_type Locate_type;
Locate_type l;
PointLocation triangulation(exact_traits);
srand(saved_seed);
triangulation.insert(S.begin(), hull_end);

int dummy=0;
Random_points_in_disc_2<Point_2> in_disc(r);
while(last != S.end()){
    *last = *in_disc++;
    triangulation.locate(*last, l, dummy);
    uniques.push_back(*last);
    if(l!=PointLocation::OUTSIDE_CONVEX_HULL)
        ++last;
}
std::random_shuffle(S.begin(), S.end(), random_number_generator);
std::copy(S.begin(), S.end(), out);

return out;
}
#endif