Experiments on the Reliability of Practical Point-in-Polygon Strategies

Stefan Schirra
Otto von Guericke University Magdeburg
Faculty of Computer Science, Department of Simulation and Graphics
Magdeburg, Germany
stschirr at ovgu.de
September 11, 2008

Abstract. We experimentally study the reliability of geometric software for point location in simple polygons. The code we tested works very well for random query points, but it often fails for degenerate and also nearly degenerate queries.

1 Introduction

Assume you would like to test points for inclusion in a simple polygon. Most likely, you will end up using one of the so-called practical point-in-polygon strategies instead of implementing one of the more sophisticated theoretically optimal point location data structures developed in computational geometry. Code for such practical point-in-polygon strategies is available on the www and in software libraries. Most of the available floating-point based code is very efficient and works well for query points chosen uniformly at random inside the bounding box of the polygon. However, the code we tested often fails for degenerate and also nearly degenerate queries, see also Fig. 1 where queries answered correctly are marked by a grey box •, false positives by a red disk ●, and false negatives by a green disk ○.

Fig. 1. Results for query points near or on the edges and the diagonals of a real-world polygon with 30 edges for strategies crossings (left) and halfplane (right).
This document describes the experimental setup for the experiments reported in [5] and provides further results. It does not describe the actual point-in-polygon code that is tested. We use the original code\(^1\) that was available on the www or code made available to us\(^2\).

1.1 How To (Re)Run The Experiments

Finally, we have a program that takes a simple polygon, generates query points, checks the results of selected point-in-polygon strategies for this polygon and these query points, and then reports the fraction of false positives and false negatives for each of the strategies. In the sequel, this test program is called pip. Since the code is partially based on CGAL, a CGAL installation is required to compile the code. If LEDA is available as well, visualization of the results is possible, too. Most requested arguments are given on the command line, some others are defined in a file, for example, the strategies to be tested are listed in file algorithms.txt which is supposed to be located in the directory where the program is started. Use pip as follows:

\[
\text{pip [QueryPointString [VisualizationType] ] [QueryPointFile] Polygonfile}
\]

Here QueryPointString is a string over the alphabet \{B, C, D, E, F, L, N, S, T, U, V\}. Each letter triggers generation of a certain type of query points as described in Section 4. Duplicates are ignored. If no such string is given, query points are generated uniformly at random. Parameter VisualizationType \in \{S, V\} provides limited access on the visualization style, see Section 6. Polygonfile must be specified. Its first line must contain the number \(n\) of vertices, the following \(n\) lines must contain the Cartesian coordinates of these vertices readable as double precision floating-point numbers. QueryPointFile is relevant only if QueryPointString contains an F. In this case, QueryPointFile must contain Cartesian coordinates of query points readable as double precision floating-point numbers.

Example: \(\text{pip DS V data/randomPolygons/random2opt_64_7}\)

In order to generate the program, extract the code using

\[
\text{notangle -L -R'main' nameOfThisFile > pip.C}
\]

compile it with your CGAL installation, and link it to the related point-in-polygon code. Your makefile should contain something like

\[
\text{pip: pip.C pip.o ptinpoly.o}
\]
\[
\$(CGAL_CXX)$(EXE_OPT)pip pip.o ptinpoly.o $(LDFLAGS)
\]

2 Practical Point-in-Polygon Strategies

Our selection of existent code includes the fastest algorithms from the beautiful graphic gems collection of Haines [4], namely crossings, a “macmartinized” crossing number algorithm, see also [1], the

---

\(^1\) Conversion from K&R style to ANSI C was made in order to turn the code into valid C++ code.

\(^2\) Thanks to Robert Walker for making his CSG-based point-in-polygon code available to us.
triangle-fan algorithms *halfplane*, *barycentric*, and *spackman*, and finally *grid*, the name says it all. Barycentric and spackman compute barycentric coordinates in addition to point location. The tested version of *halfplane* uses sorting and *grid* uses a $20 \times 20$ grid. For the sake of completeness, we also tested Weiler’s code [8] which computes the winding number using quadrant movements.

(excerpts from the header of Haines’ ptinpoly.c)≡

\begin{itemize}
  \item crossings - count the crossing made by a ray from the test point
  \item weiler angle summation - sum the angles using quad movements
  \item half-plane testing - test triangle fan using half-space planes
  \item barycentric coordinates - test triangle fan w/barycentric coords
  \item spackman barycentric - preprocessed barycentric coordinates
  \item grid testing - grid imposed on polygon
\end{itemize}

Furthermore, we consider Franklin’s PNPOLY code [3], which is another crossing number based algorithm.

Walker and Snoeyink [7] use CSG representations of polygons for point location. We include their code, *csg*, in our case study because of the reported efficiency, although the code is not publicly available.

Finally, we also consider point location code for polygons from CGAL using the obvious inexact geometry kernel with double precision floating-point coordinates. Of course, with an exact kernel, CGAL’s point location code for polygons is fully reliable. The same holds for LEDA’s rat_polygon code (floating-point filtered rational kernel).

More recently we added what we call RTR, the crossing number code from the real-time rendering book (2nd ed.) [1], as listed on page 585. The authors of the book refer to Haines’ code, i.e, *crossings*, although RTR and *crossings* are different. This explains why RTR has been added to the test set much later. We “adapted” the code from the book.

Overall, we have
\begin{itemize}
  \item crossings
  \item weiler
  \item halfplane
  \item barycentric
  \item spackman
  \item grid
  \item PNPOLY
  \item csg
  \item RTR
  \item CGAL
\end{itemize}

We do not discuss the underlying algorithms here, but instead of this refer the interested reader to the original papers and web pages.

### 2.1 Polygon Concept and Models

For the sake of generic testing we use an object oriented framework where we wrap point-in-polygon testing in a member function of a polygon class. We define a simple polygon concept that allows for construction from a point sequence, where the corresponding point concept must allow for construction from a pair of double coordinates. The polygon concept requires a member function `bool contains(Point p)` that is supposed to return true if and only if point `p` lies
inside the closure of a simple polygon.

For each of the above point-in-polygon code we define a corresponding model of this polygon concept. Whenever it is feasible, we use a CGAL point type with double coordinates as the corresponding point type, since this point type is used for generating query points as well.

\[\text{typedef for cgal point type with double coordinates} \equiv \text{typedef CGAL::Point_2<CGAL::Simple_cartesian<double> > Point;}\]

We have the following

\[\text{polygon classes} \equiv \langle \text{crossings polygon class} \rangle \langle \text{weiler polygon class} \rangle \langle \text{halfplane polygon class} \rangle \langle \text{barycentric polygon class} \rangle \langle \text{spackman polygon class} \rangle \langle \text{grid polygon class} \rangle \langle \text{pupoly polygon class} \rangle \langle \text{csg polygon class} \rangle \langle \text{CGAL based polygon class} \rangle \langle \text{rat leda polygon class} \rangle \langle \text{CGAL based exact polygon class} \rangle \langle \text{RTR polygon class} \rangle\]

A corresponding traits class provides requested information on the related types.

\[\text{point in polygon traits} \equiv \text{template <class PolygonType> class Polygon_traits }\]
\[\{
\text{public:}
\text{typedef typename PolygonType::Point Point;}
\text{typedef PolygonType Polygon;}
\};\]

### 2.2 Some Shared Code Fragments

Before we present the different polygon models corresponding to the different point-in-polygon strategies, we present some common code chunks.

The first one creates an array of double coordinates and assigns the coordinates of points of type Point to this array. Such an array is used by Haines for representing the vertices of the polygon. numverts stores the number of vertices of the polygon.

\[\text{assign coordinates to array pgon and set numverts} \equiv \text{numverts = std::distance( first, beyond); pgon = new double[numverts][2]; int k = 0; for ( ; first != beyond; ++first, ++k )}\]
{ pgon[k][0] = first->x();
    pgon[k][1] = first->y();
}

The second piece of code transforms a query point of type Point into the “type” expected by Haines’ code, namely an array double point[2]. First, such an object is defined, then coordinates are assigned.

⟨assign q’s coordinates to array point⟩ ≡
  double point[2];
  point[0] = q.x();
  point[1] = q.y();

Note that here we use double precision coordinates everywhere, so there are no conversion errors involved.

2.3 crossings

We start with a model that uses crossings for point in polygon testing. For point location it uses the following function from Haines [4].

```c
int CrossingsTest( double pgon[][2], int numverts, double point[2])
```

uses a crossing test to check whether point is contained in the polygon defined by the coordinates of the numverts vertices in pgon. Shoots a test ray along x-axis. The strategy, from MacMartin, is to compare vertex Y values to the testing point’s Y and quickly discard edges which are entirely to one side of the test ray. Returns 1, if point is inside and 0 if outside.

⟨crossings polygon class⟩ ≡
```
class Polygon_crossings
{
    public:
        ⟨typedef for cgal point type with double coordinates⟩
    private:
        double (*pgon)[2];
        int numverts;
    public:
        template <class ForwardIterator>
        Polygon_crossings( ForwardIterator first, ForwardIterator beyond)
        { ⟨assign coordinates to array pgon and set numverts⟩ }`
        ~Polygon_crossings() { delete[] pgon; }
    bool
        contains( const Point& q)
        { ⟨assign q’s coordinates to array point⟩
            return CrossingsTest( pgon, numverts, point );
        }
};
```
2.4 weiler

The graphics gems collection contains Weiler's point-in-polygon method as well. The point location method we use is

\[
\text{int } \text{WeilerTest( double pgon[][2], int numverts, double point[2])}
\]

tracks quadrant movements using Weiler’s algorithm; optimized by Haines for testing purposes. Checks whether point is contained in the polygon defined by the coordinates of the numverts vertices in pgon. Returns 1, if point is inside, 0 if outside.

as described by Weiler [8]. The code we used is contained in [4].

\[
\langle \text{weiler polygon class} \rangle \equiv
\]

class Polygon_weiler
{
public:
  typedef for cgal point type with double coordinates
private:
  double (*pgon)[2];
  int numverts;
public:
  template <class ForwardIterator>
  Polygon_weiler( ForwardIterator first, ForwardIterator beyond)
  { (assign coordinates to array pgon and set numverts) }
  ~Polygon_weiler() { delete[] pgon; }
  bool
  contains( const Point& q)
  { (assign q's coordinates to array point)
    return WeilerTest( pgon, numverts, point );
  }
};

2.5 halfplane

The triangle-fan algorithms halfplane requires some preprocessing. This preprocessing step computes equations for the supporting lines of the polygon edges. Haines’ code uses a struct PlaneSet for this purpose, where he stores the coefficients of the computed line equation. Our polygon class for point-in-polygon testing via halfplane computes the line equations for the halfplanes at construction time using PlaneSetup(). Function PlaneCleanup() frees the memory allocated for the line equations that define the halfplanes. PlaneTest does the actual point location.

\[
\text{pPlaneSet } \text{PlaneSetup( double pgon[][2], int numverts)}
\]

\[
\text{void } \text{PlaneCleanup( pPlaneSet p\_plane\_set)}
\]

\[
\text{int } \text{PlaneTest( pPlaneSet p\_plane\_set, int numverts, double point[2])}
\]

The comment in Haines’ code says:

/* Split the polygon into a fan of triangles and for each triangle test if
 * the point is inside of the three half-planes formed by the triangle's edges. */
Call setup with 2D polygon \(_\text{pgon}_\) with \(_\text{numverts}_\) number of vertices, which returns a pointer to a plane set array.

Call testing procedure with a pointer to this array, \(_\text{numverts}_\), and test point \(_\text{point}_\), returns 1 if inside, 0 if outside.

Call cleanup with pointer to plane set array to free space.

SORT and CONVEX can be defined for this test.

*/

```cpp
⟨halfplane polygon class⟩≡

```cpp
class Polygon_halfplane
{
    public:
        ⟨typedef for cgal point type with double coordinates⟩
    private:
        double (*pgon)[2];
        int numverts;
        PlaneSet* p_plane_set;
    public:
        template <class ForwardIterator>
        Polygon_halfplane( ForwardIterator first, ForwardIterator beyond)
        {
            (assign coordinates to array pgon and set numverts)
            p_plane_set = PlaneSetup( pgon, numverts );
        }
        ~Polygon_halfplane() { PlaneCleanup( p_plane_set ); delete[] pgon; }
        bool contains( const Point& q)
        {
            (assign q's coordinates to array point)
            return PlaneTest( p_plane_set, numverts, point );
        }
};
```

For barycentric we use

```cpp
int BarycentricTest( double pgon[][2], int numverts, double point[2])
```
splits the polygon into a fan of triangles and tests for each triangle tests if the point has barycentric coordinates which are inside the triangle. Returns 1 if inside, 0 is outside.

```cpp
⟨barycentric polygon class⟩≡

class Polygon_barycentric
{
    public:
        ⟨typedef for cgal point type with double coordinates⟩
    private:
        double (*pgon)[2];
        int numverts;
    public:
        template <class ForwardIterator>
        Polygon_barycentric( ForwardIterator first, ForwardIterator beyond)
```
Just like halfplane, spackman involves a little preprocessing. We use the following function from [4].

```cpp
pSpackmanSet SpackmanSetup( double pgon[][2], int numverts, int *p_numrec)
void             SpackmanCleanup( pSpackmanSet p_spackman_set)
int              SpackmanTest( double anchor[2],
                                 pSpackmanSet p_spackman_set,
                                 int numrec,
                                 double point[2])
```

The comment in Haines’ code says:

```c
/* Split the polygon into a fan of triangles and for each triangle test if
 * the point has barycentric coordinates which are inside the triangle.
 * Use Spackman's normalization method to precompute various parameters.
 * Call setup with 2D polygon _pgon_ with _numverts_ number of vertices,
 * which returns a pointer to the array of the parameters records and the
 * number of parameter records created.
 * Call testing procedure with the first vertex in the polygon _pgon[0]_,
 * a pointer to this array, the number of parameter records, and test point
 * _point_, returns 1 if inside, 0 if outside.
 * Call cleanup with pointer to parameter record array to free space.
 * SORT can be defined for this test.
 * (CONVEX could be added: see PlaneSetup and PlaneTest for method)
 */
```

(spackman polygon class)≡

class Polygon_spackman
{
    public:
        (typedef for cgal point type with double coordinates)
    private:
        int numverts;
        int numrec;
        double (*pgon)[2];
        SpackmanSet* p_spackman_set;
    public:
        template <class ForwardIterator>
        Polygon_spackman( ForwardIterator first, ForwardIterator beyond)
        { (assign coordinates to array pgon and set numverts)
            p_spackman_set = SpackmanSetup( pgon, numverts, &numrec );
        }
}

2.7 spackman
2.8 grid

According to Haines (and confirmed by our experiments) Grid is often the fastest point-in-polygon test. However, it requires some preprocessing as well. We use resolution 20 as in [4].

```cpp
void GridSetup( double pgon[][2], int numverts, int resolution, pGridSet p_gs)
void GridCleanup( pGridSet p_gs)
int GridTest( pGridSet p_gs, double point[2])
```

The comment in Haines’ code says:

`/* Impose a grid upon the polygon and test only the local edges against the * point. *
* Call setup with 2D polygon _pgon_ with _numverts_ number of vertices, *
* grid resolution _resolution_, and a pointer to a grid structure _p_gs_. *
* Call testing procedure with a pointer to this array and test point _point_, *
* returns 1 if inside, 0 if outside. *
* Call cleanup with pointer to grid structure to free space. */

/* Strategy for setup: *
* Get bounds of polygon, allocate grid. *
* "Walk" each edge of the polygon and note which edges have been crossed *
* and what cells are entered (points on a grid edge are always considered *
* to be above that edge). Keep a record of the edges overlapping a cell. *
* For cells with edges, determine if any cell border has no edges passing *
* through it and so can be used for shooting a test ray. *
* Keep track of the parity of the x (horizontal) grid cell borders for *
* use in determining whether the grid corners are inside or outside. */
```

(grid polygon class)≡
```cpp
class Polygon_grid
{
    public:
        // typedef for cgal point type with double coordinates
    private:
        int numverts;
        int resolution;
        double (*pgon)[2];
        GridSet grid_set;
    public:
        template <class ForwardIterator>
        Polygon_grid( ForwardIterator first, ForwardIterator beyond)
        { resolution = 20;
            // assign coordinates to array pgon and set numverts
            GridSetup( pgon, numverts, resolution, &grid_set );
        }
```
2.9 PN POLY

Franklin’s PN POLY is only 8 lines of code, so we decided to copy it to the contains() member function. The requested copyright notice is much longer, however:

```cpp
Polygon_grid() { GridCleanup( &grid_set ); delete[] pgon; }
bool contains( const Point& q)
{return GridTest( &grid_set, point );}
};
```

PN POLY

As said above, the PN POLY code is inlined in the contains() member function.

```cpp
class Polygon_pnpoly
{
  public:
```

As said above, the PN POLY code is inlined in the contains() member function.
typedef for cgal point type with double coordinates

private:
  double* xp;
  double* yp;
  int npol;

public:
  template <class ForwardIterator>
  Polygon_pnpoly( ForwardIterator first, ForwardIterator beyond)
  { npol = std::distance( first, beyond);
    xp = new double[npol];
    yp = new double[npol];
    int k = 0;
    for ( ; first != beyond; ++first, ++k )
      { xp[k] = first->x();
        yp[k] = first->y();
      }
  }

bool contains( const Point& q)
  {
    double x = q.x();
    double y = q.y();
    int i, j, c = 0;
    for (i = 0, j = npol-1; i < npol; j = i++)
      { if (((yp[i]<=y) && (y<yp[j])) ||
          ((yp[j]<=y) && (y<yp[i])))
          
        { x = (xp[j] - xp[i]) * (y - yp[i]) / (yp[j] - yp[i]) + xp[i])
        {
          c = !c;
        }
      }
    return c;
  }

};

2.10 csg

The next polygon class uses code by Walker and Snoeyink [7] which first computes a csg representation of a polygon and uses this representation for point location. It uses the following functions by Walker (designed analogously to Haines’ code).

int CSGSetup( double pgon[][2],
              int numverts,
              int ordered,
              pCSGSet csg_set,
              double bbox[][2])

void CSGCleanup( pCSGSet csg_set)

int CSGTest( pCSGSet csg_set, double point[2])

(csg polygon class) ≡
class Polygon_csg
{
  public:
    (typedef for cgal point type with double coordinates)
  private:
double (*pgon)[2];
int numverts;
double bbox[4][2];
CSGSet csg_set;
public:
template <class ForwardIterator>
Polygon_csg( ForwardIterator first, ForwardIterator beyond)
{
    ⟨assign coordinates to array pgon and set numverts⟩
    bbox[0][0] = 0.0;
    bbox[0][1] = 0.0;
    bbox[1][0] = 1.0;
    bbox[1][1] = 0.0;
    bbox[2][0] = 1.0;
    bbox[2][1] = 1.0;
    bbox[3][0] = 0.0;
    bbox[3][1] = 1.0;
    CSGSetup( pgon, numverts, 1, &csg_set, bbox );
}
~Polygon_csg() { CSGCleanup( &csg_set ); delete[] pgon; }
bool contains( const Point& q)
{
    ⟨assign q’s coordinates to array point⟩
    return CSGTest( &csg_set, point );
}
};

2.11 no-sort-csg

NOT YET IMPLEMENTED!

2.12 RTR

The next strategie, which we call RTR, is a crossing number algorithm as well. It is the code as listed on page 585 of the real-time rendering book (2nd ed.) [1]. However, this code is more subtle than crossings and more reliable (for counterclockwise oriented simple polygons). Note that RTR and crossings are different, although Haines and Akenine-Möller refer the reader to the crossings code in [1].

⟨RTR polygon class⟩≡
class Polygon_RTR
{
    public:
    ⟨typedef for cgal point type with double coordinates⟩
    private:
    std::vector< Point> vertices;
    public:
    template <class ForwardIterator>
    Polygon_RTR( ForwardIterator first, ForwardIterator beyond)
    {
        std::copy( first, beyond, std::back_inserter(vertices));
    }
    bool contains( const Point& q)
```cpp
bool result = false;
Point e0 = vertices[vertices.size()-1];
Point e1 = vertices[0];
bool y0 = (e0.y()>=q.y());
for(unsigned int i=1; i<=vertices.size(); i++)
{ bool y1 = (e1.y() >= q.y());
  if ( y0 != y1)
  { if(( (e1.y()-q.y())*(e0.x()-e1.x())
      >=(e1.x() - q.x())*(e0.y()-e1.y()) ) == y1 )
    { result = !result; } }
  y0 = y1;
  e0 = e1;
  if ( i<vertices.size() )
  { e1 = this->vertices[i]; }
}
return result;
}
```

### 2.13 **CGAL with simple Cartesian kernel and double precision floats**

The CGAL code we use is instantiated with the simple Cartesian kernel. Note that this kernel is inexact due to rounding errors of floating-point arithmetic! We derive from CGAL’s polygon class to adjust the interface for conforming to our polygon concept.

```cpp
typedef CGAL::Polygon_2<CGAL::Simple_cartesian<double> > PolygonSC;
class Polygon_cgalSCdouble : public PolygonSC
{
  public:
  (typedef for cgal point type with double coordinates)
  template <class InputIterator>
  Polygon_cgalSCdouble( InputIterator first, InputIterator beyond)
    : PolygonSC( first, beyond)
  {}
  bool
  contains(const Point& q) const
  { return !has_on_unbounded_side(q); }
};
```

### 2.14 **LEDA’s rational kernel**

Finally, we need fully reliable code to check correctness of the computed results. The code we trust is LEDA’s code with LEDA’s rational kernel. This kernel uses homogeneous coordinates with arbitrary precision integers to achieve correctness for rational computations. It uses floating-point filters to reduce the exact computation overhead. The underlying point-in-polygon testing algorithm is a reliable implementation of the crossings number algorithm.
\textit{rat\_leda polygon class}≡

\begin{verbatim}
class Polygon_rat_leda : public leda::rat_polygon
{
    public:
        typedef leda::rat_point Point;

        template <class InputIterator>
        Polygon_rat_leda( InputIterator first, InputIterator beyond)
        { leda::list< leda::rat_point > L;
            for ( ; first != beyond; ++first)
                { L.append( *first ); }
            leda::rat_polygon::operator=(
                leda::rat_polygon( L, SIMPLE, DISREGARD_ORIENTATION ) );
        }

    bool contains(const Point& q) const
    { return leda::rat_polygon::contains( q); }
};
\end{verbatim}

\subsection{2.15 CGAL's exact kernel}

Alternatively, for people without LEDA, we use CGAL's exact predicates exact constructions kernel.
Most likely, there are no constructions involved with CGAL's point in polygon code, so exact predicates inexact construction would suffice, but we are not eager to check this. Again we derive from CGAL's polygon class to adjust the interface for conforming to our polygon concept.

\textit{CGAL based exact polygon class}≡

\begin{verbatim}
typedef CGAL::Exact_predicates_exact_constructions_kernel Exact_kernel;
typedef CGAL::Polygon_2< Exact_kernel> Exact_polygon;

class Polygon_exact : public Exact_polygon
{
    public:
        typedef Exact_kernel::Point_2 Point;

        template <class InputIterator>
        Polygon_exact( InputIterator first, InputIterator beyond)
        { Exact_polygon( first, beyond) }

    bool contains(const Point& q) const
    { return !has_on_unbounded_side(q); }
};
\end{verbatim}

\section{3 Simple Polygons}

We use four types of polygons:

\begin{itemize}
    \item [(a)] “random” polygons generated using the 2-opt heuristic using either CGAL's corresponding generator or the corresponding generator from RPG by T. Auer and Martin Held [2].
    \item [(b)] “random” orthogonal polygons generated using code by Ana Paula Tomás based on the Inflate-Cut heuristic by Tomás and Bajuelos [6].
    \item [(c)] real-world polygons from GIS data, more precisely, city and village boundaries.
\end{itemize}
Fig. 2. (a) a random polygon with 256 vertices, (b) a random orthogonal polygon with 200 vertices, (c) a real-world polygon with 313 vertices, (d) a rotated H-shaped polygon.

(d) a few special artificial polygons

All random polygons have vertices in the unit square $[0, 1] \times [0, 1]$. For the sake of comparability, the GIS data are scaled to this range as well. Fig. 2 shows examples.

## 4 Query Points

Usually, points generated uniformly at random are used for testing point-in-polygon code. For such randomly generated points, the code works very well almost all of the time, because degenerate or nearly degenerate query points are very rare, though not impossible.

In order to challenge the point-in-polygon code we test, we consider several query point generators, that generate query points that create points that are degenerate or nearly degenerate at least for some of the algorithms.
4.1 Random Query Points

The first one generates points uniformly at random inside the $[0, 1] \times [0, 1]$ box using a LEDA random source both for generating $x$- and $y$-coordinates.

```cpp
template <typename InputIterator, typename OutputIterator>
OutputIterator p_u_b( InputIterator first, InputIterator beyond, OutputIterator res, int N )
{
    typedef for cgal point type with double coordinates
    leda::random_source rsx;
    leda::random_source rsy;
    ...
}
```
double x, y;
for (int i=1; i<=N; ++i)
{ rsx >> x; rsy >> y;
  *res++ = Point(x,y);
} 
}

4.2 Query Points (almost) on the Polygon Edges

We use CGAL’s generator for creating points on a segment. Since we use floating-point coordinates, these points are not necessarily exactly on the segment, but rather almost on the segment. Note that the CGAL generator produces equidistant points including source and target of the segment. Thus the set of query points generated by the following generator includes every polygon vertex twice!

\[
\langle \text{points on segments} \rangle \equiv 
\]
\[
\text{template } \langle \text{typename InputIterator, typename OutputIterator} \rangle 
\text{OutputIterator}
\text{p_o_s}( \text{InputIterator first, InputIterator beyond, OutputIterator res, int per_edge } )
\{
\text{(typedef for cgal point type with double coordinates)}
\text{std::list< Point > L;}
\text{std::copy( first, beyond, std::back_inserter(L) );}
\text{std::list< Point >::iterator pit, qit;}
\text{pit = qit = L.begin();}
\text{for ( ++pit; pit != L.end(); ++pit) }
\text{res = CGAL::points_on_segment_2( *qit, *pit, per_edge, res);}
\text{qit = pit;}
\text{res = CGAL::points_on_segment_2( *L.rbegin(), *L.begin(), per_edge, res);}
\text{return res;}
\}
\]

Thus we provide yet another generator which excludes polygon vertices, i.e., it uses only the points in the interior of the edges.

\[
\langle \text{points only on interior of segments} \rangle \equiv 
\]
\[
\text{template } \langle \text{typename InputIterator, typename OutputIterator} \rangle 
\text{OutputIterator}
\text{p_o_is( InputIterator first, InputIterator beyond, OutputIterator res, int per_edge )}
\{
\text{(typedef for cgal point type with double coordinates)}
\text{std::list< Point > L;}
\text{std::list< Point > R;}
\text{std::copy( first, beyond, std::back_inserter(L) );}
\text{std::list< Point >::iterator sit, tit;}
\text{CGAL::points_on_segment_2( *L.rbegin(), *L.begin(), per_edge+2,}
\text{ std::back_inserter(R) );}
\text{sit = R.begin(); ++sit;}
\text{tit = R.end(); --tit;}
\text{res = std::copy( sit, tit, res);}
\}
The following procedure generates points “on” the edges analogously to \( p_{\text{no\_s}} \), but it reports only those points that are not exactly on the edges. In general almost all of the points are not on the edges, except for the vertices. In very special cases, for example, horizontal and vertical edges, no points are reported for an edge.

\[
\langle \text{points not on segments} \rangle \equiv

\#ifdef CGAL_USE_LEDA

template<typename InputIterator, typename OutputIterator>
OutputIterator
p_no_s( InputIterator first, InputIterator beyond, OutputIterator res, int per_edge )
{

typedef CGAL point type with double coordinates

std::list< Point > L;
std::copy( first, beyond, std::back_inserter(L) );
std::list< Point >::iterator pit, qit;
leda::rat_point rat_p, rat_q, rat_r;
std::vector< Point > local;
pit = qit = L.begin();
for ( ++pit; pit != L.end(); ++pit )
{
CGAL::points_on_segment_2( *qit, *pit, per_edge+2, std::back_inserter(local));
rat_p = leda::rat_point( pit->x(), pit->y() );
rat_q = leda::rat_point( qit->x(), qit->y() );
for ( std::vector< Point >::iterator rit = local.begin(); rit != local.end(); ++rit )
{ rat_r = leda::rat_point( rit->x(), rit->y() );
if ( ! collinear( rat_p, rat_q, rat_r ) )
{ *res++ = *rit; }
}
qit = pit;
}
local.erase( local.begin(), local.end() );
pit = L.begin();
CGAL::points_on_segment_2( *qit, *pit, per_edge, std::back_inserter(local));
rat_p = leda::rat_point( pit->x(), pit->y() );
rat_q = leda::rat_point( qit->x(), qit->y() );
for ( std::vector< Point >::iterator rit = local.begin(); rit != local.end(); ++rit )
{ rat_r = leda::rat_point( rit->x(), rit->y() );
if ( ! collinear( rat_p, rat_q, rat_r ) )
{ *res++ = *rit; }
}
\}
\]
4.3 Query Points on the Supporting Lines of Polygon Edges

(points on extended segments) \equiv

\begin{verbatim}
template<typename InputIterator, typename OutputIterator>
OutputIterator
p_o_l( InputIterator first, InputIterator beyond, OutputIterator res,
     int per_edge )
{
  // typedef for cgal point type with double coordinates
  std::list< Point > L;
  std::copy( first, beyond, std::back_inserter(L) );
  std::list< Point >::iterator pit, qit;
  pit = qit = L.begin();
  Point s, t;
  for ( ++pit; pit != L.end(); ++pit)
    { clip_line_at_unit_box( *qit, *pit, s, t);
      res = CGAL::points_on_segment_2( s, t, per_edge, res);
      qit = pit;
    }
  clip_line_at_unit_box( *L.rbegin(), *L.begin(), s, t);
  res = CGAL::points_on_segment_2( s, t, per_edge, res);
  return res;
}
\end{verbatim}

4.4 Query Points on Triangle-Fan Diagonals

We also generate query points on the triangle edges considered in the triangle-fan algorithms

\textit{spackman}, \textit{barycentric}, and \textit{halfplane} which do not coincide with polygon edges. These edges all start at the first polygon vertex \( q_{first} \). Earlier versions of this generator also created query points on the two polygon edges incident to \( q_{first} \). We refrain from generating such points in order to keep the generators more disjoint. As above for polygon boundary edges, we have two versions, one including polygon vertices, and one that doesn’t. The former generates \( q_{first} \) as a query vertex many times.

(points on diagonals) \equiv

\begin{verbatim}
template<typename InputIterator, typename OutputIterator>
OutputIterator
p_o_d( InputIterator first, InputIterator beyond, OutputIterator res,
     int per_diagonal )
{
  // typedef for cgal point type with double coordinates
  std::list< Point > L;
  std::copy( first, beyond, std::back_inserter(L) );
  if ( L.size() <= 3 ) return res;
  std::list< Point >::iterator pit, qit, fit;
  pit = qit = L.begin();
  fit = L.end();
  ++pit;
  for ( ; pit != fit; ++pit)
    { clip_line_at_unit_box( *qit, *pit, s, t);
      res = CGAL::points_on_segment_2( s, t, per_diagonal, res);
      qit = pit;
    }
  return res;
}
\end{verbatim}
{ res = CGAL::points_on_segment_2( *pit, *qit, per_diagonal, res); }  
    return res; 
}

⟨ points on interior of diagonals ⟩ ≡

template <typename InputIterator, typename OutputIterator>
OutputIterator
p_o_id( InputIterator first, InputIterator beyond, OutputIterator res,  
    int per_diagonal )
{
    typedef for cgal point type with double coordinates  
    std::list< Point > L;  
    std::copy( first, beyond, std::back_inserter(L) );  
    if ( L.size() <= 3 ) return res;  
    std::list< Point > R(per_diagonal+2);  
    std::list< Point >::iterator pit, qit, fit, sit, tit;  
    pit = qit = L.begin();  
    ++pit; ++pit;  
    fit = L.end();  
    --fit;  
    for ( ; pit != fit; ++pit)  
    { CGAL::points_on_segment_2( *pit, *qit, per_diagonal+2, R.begin() );  
        sit = R.begin(); ++sit;  
        tit = R.end(); --tit;  
        res = std::copy( sit, tit, res);  
    }  
    return res; 
}

4.5 Query Points at Cross Lines at Polygon Vertices

For crossing number algorithms, polygon vertices on the ray shoted from the query points are de-
egenerate cases. Usually, horizontal or vertical rays are used. Therefore it makes sense to have a  
generator that creates query points vertically above and below and horizontally to the left and right of  
the polygon vertices, i.e., on the horizontal and vertical lines through the polygon vertices. Thus the  
current version creates points on these horizontal and vertical lines clipped to the unit square. Since  
CGAL::points_on_segment_2 generates points equally spaced, we would get a grid like distribu-
tion of query points. Instead of CGAL::points_on_segment_2 we generate points on the horizontals  
and verticals uniformly ar random. Previous versions used a slightly different method to generate the  
query points.

⟨ points on crosses at vertices ⟩ ≡

template <typename InputIterator, typename OutputIterator>
OutputIterator
p_c_v( InputIterator first, InputIterator beyond, OutputIterator res,  
    int per_crossline )
{
    typedef for cgal point type with double coordinates  
    double xmin = 0.0; double ymin = 0.0;  
    double xmax = 1.0; double ymax = 1.0;  
    Point P, Q;  
    std::list< Point > L;  
    std::copy( first, beyond, std::back_inserter(L) );  
    std::list< Point >::iterator pit;
4.6 Query Points near Polygon Vertices

The last generator simply perturbs the polygon vertices inside a box of a specified length $\varepsilon$.

$$\langle \text{points near vertices} \rangle \equiv$$

```cpp
template <typename InputIterator, typename OutputIterator>
OutputIterator
p_n_v( InputIterator first, InputIterator beyond, OutputIterator res,
      int per_vertex, double eps = 0.00015 )
{
    typedef for cgal point type with double coordinates
    std::list< Point > L;
    std::copy( first, beyond, std::back_inserter(L) );
    std::list< Point >::iterator pit;
    for ( int i = 0; i < per_vertex; ++i )
    { std::list< Point > PL;
      std::copy( L.begin(), L.end(), std::back_inserter(PL));
      CGAL::perturb_points_2( PL.begin(), PL.end(), eps );
      res = std::copy( PL.begin(), PL.end(), res);
    }
    return res;
}
```

4.7 Little Helpers

$$\langle \text{little helpers} \rangle \equiv$$

```cpp
template <class Kernel>
void
clip_line_at_unit_box( const CGAL::Point_2<Kernel>& p,
                       const CGAL::Point_2<Kernel>& q,
                       CGAL::Point_2<Kernel>& s,
                       CGAL::Point_2<Kernel>& t )
{
    CGAL::Line_2<Kernel> l(p,q);
    CGAL::Segment_2<Kernel> seg;
    CGAL::Iso_rectangle_2<Kernel> unit_box( CGAL::Point_2<Kernel>(0,0),
                                           CGAL::Point_2<Kernel>(1,1) );
```
CGAL::Object obj = intersection( unit_box, l);
CGAL::assign( seg, obj);
s = seg.start();
t = seg.target();
}

5 More on Experimental Setup

The working horse of our experimentation work takes two sequences of points. The first one defines
the counterclockwise sequence of polygon vertices, the second one the set of query points. We return
a vector of boolean values that stores the results of the calls of the contains member function for
the sequence of query points:

⟨create polygon and answer queries⟩≡
template <class InputIterator1, class InputIterator2, class Traits>
std::vector<bool>
compute_pip_results( InputIterator1 first1, InputIterator1 beyond1,
        InputIterator2 first2, InputIterator2 beyond2,
        const Traits& pp)
{ typedef typename Traits::Point Point;
typedef typename Traits::Polygon Polygon;
std::vector<bool> B;
Polygon P(first1, beyond1);
for ( ; first2 != beyond2; ++first2)
{ B.push_back( P.contains( *first2) ); }
return B;
}

5.1 Reading Polygon Data

We read polygon data from file and store them in containers using different point types.

⟨containers⟩≡
std::vector<Point> VV;
std::vector<Exact_kernel::Point_2> VE;
#if CGAL_USE_LEDA
leda::list< leda::point> VL;
leda::list< leda::rat_point> VR;
#endif // CGAL_USE_LEDA

The file name is the last argument specified on the command line. Points are initially stored in con-
tainer VV and are converted subsequently, see Subsection 5.3.

⟨read polygon vertex coordinates from file⟩≡
std::ifstream F(argv[argc-1]);
int N; F >> N;
double x, y;
for (int i=1; i<=N; ++i)
{ F >> x; F >> y;
  VV.push_back( Point(x,y)); }

22
5.2 Generating Query Points

Query points are generated as CGAL points with double coordinates and then transformed to other point types as requested. Thus the query points are stored in different containers.

\[
\text{(containers)} + \equiv \\
\begin{align*}
\text{std::vector<Point> QV;} \\
\text{std::vector<Exact\_kernel::Point\_2> QE;} \\
\text{\#ifdef CGAL\_USE\_LEDA} \\
\text{leda::list<leda::point> QL;} \\
\text{leda::list<leda::rat\_point> QR;} \\
\text{\#endif // CGAL\_USE\_LEDA}
\end{align*}
\]

The selection of generators is specified by a command line argument. For each generator, you have to add a corresponding capital letter to the first argument (no blanks). If no generator is specified, query points are generated uniformly at random. Initially, query points are stored in a container QV.

\[
\text{(create query points)} \equiv \\
\begin{align*}
\text{std::string QPS;} \\
\text{if ( argc >= 3) } \\
\text{\{} QPS = std::string(argv[1]); } \\
\text{else } \\
\text{\{} QPS = std::string("U"); } \\
\text{\} } \\
\text{if ( QPS.find("U") != std::string::npos ) } \\
\text{\{} p_u_b( VV.begin(), VV.end(), std::back_inserter(QV), 3000 ); } \\
\text{\} } \\
\text{if ( QPS.find("S") != std::string::npos ) } \\
\text{\{} p_o_s( VV.begin(), VV.end(), std::back_inserter(QV), 20 ); } \\
\text{\} } \\
\text{if ( QPS.find("T") != std::string::npos ) } \\
\text{\{} p_o_t( VV.begin(), VV.end(), std::back_inserter(QV), 20 ); } \\
\text{\} } \\
\text{if ( QPS.find("B") != std::string::npos ) } \\
\text{\{} p_o_b( VV.begin(), VV.end(), std::back_inserter(QV), 20 ); } \\
\text{\} } \\
\text{if ( QPS.find("L") != std::string::npos ) } \\
\text{\{} p_o_l( VV.begin(), VV.end(), std::back_inserter(QV), 20 ); } \\
\text{\} } \\
\text{if ( QPS.find("D") != std::string::npos ) } \\
\text{\{} p_o_d( VV.begin(), VV.end(), std::back_inserter(QV), 20 ); } \\
\text{\} } \\
\text{if ( QPS.find("E") != std::string::npos ) } \\
\text{\{} p_o_e( VV.begin(), VV.end(), std::back_inserter(QV), 20 ); } \\
\text{\} } \\
\text{if ( QPS.find("C") != std::string::npos ) } \\
\text{\{} p_c_v( VV.begin(), VV.end(), std::back_inserter(QV), 10 ); } \\
\text{\} } \\
\text{if ( QPS.find("V") != std::string::npos ) } \\
\text{\{} std::copy( VV.begin(), VV.end(), std::back_inserter(QV) ); } \\
\text{\} } \\
\text{if ( QPS.find("N") != std::string::npos ) } \\
\text{\{} p_n_v( VV.begin(), VV.end(), std::back_inserter(QV), 20 ); } \\
\text{\} } \\
\text{if ( QPS.find("F") != std::string::npos ) } \\
\text{\{} std::ifstream QF(argv[argc-2]); } \\
\end{align*}
\]
int n; QF >> n;
Point p;
for (int i=1; i<=n; ++i)
{ QF >> p;
  QV.push_back(p);
}

5.3 Point Type Conversion

The template functor below performs point conversion in a straightforward way, assuming that the origin point type allows for accessing Cartesian double coordinates via \(x()\) and \(y()\) member functions, just like a CGAL point does. The target point type must be constructible from two doubles.

\[\text{little helpers} \equiv \]

\[
\text{template } \langle \text{class OriginPointType, class TargetPointType}\rangle \\
\text{class convert_point} \\
\{ \\
\text{public:} \\
\text{TargetPointType} \\
\text{operator() (const OriginPointType& p) } \\
\{ \text{return TargetPointType} (p.x(), p.y()); \} \\
\};
\]

\[\text{convert points} \equiv \]

\[
\text{std::transform( VV.begin(), VV.end(), std::back_inserter(VE),} \\
\text{convert_point< Point, Exact_kernel::Point_2>() );} \\
\text{std::transform( QV.begin(), QV.end(), std::back_inserter(QE),} \\
\text{convert_point< Point, Exact_kernel::Point_2>() );} \\
#\text{ifdef CGAL_USE_LEDA} \\
\text{std::transform( VV.begin(), VV.end(), std::back_inserter(VL),} \\
\text{convert_point< Point, leda::point>() );} \\
\text{std::transform( QV.begin(), QV.end(), std::back_inserter(QL),} \\
\text{convert_point< Point, leda::point>() );} \\
\text{std::transform( VV.begin(), VV.end(), std::back_inserter(VR),} \\
\text{convert_point< Point, leda::rat_point>() );} \\
\text{std::transform( QV.begin(), QV.end(), std::back_inserter(QR),} \\
\text{convert_point< Point, leda::rat_point>() );} \\
#\text{endif} // \text{CGAL_USE_LEDA}
\]

5.4 Selection of Point Location Strategies

We use a very elementary tool to select the strategies to be tested: The strategies to be tested are simply configured via a plain file algorithms.txt to be located in the directory where the program is started. Each line provides the name of strategy which is added to a dictionary of strategies to be tested. We use a dictionary of type \text{std::set\< std::string\>}.  

\[\text{select algorithms} \equiv \]

\[
\text{std::set\< std::string\> algos;} \\
\text{std::ifstream AF("algorithms.txt");} \\
\text{std::string str;}
\]
while ( !AF.eof() )
{
    AF >> str;
    algos.insert( str);
}

5.5 Computation of Point Location Results

\textit{compute correct results} ≡
\begin{verbatim}
#ifdef CGAL_USE_LEDA
    std::vector<bool> B_correct
    = compute_pip_results( VR.begin(), VR.end(), QR.begin(), QR.end(),
                           Polygon_traits<Polygon_rat_leda>() );
#else
    std::vector<bool> B_correct
    = compute_pip_results( VE.begin(), VE.end(), QE.begin(), QE.end(),
                           Polygon_traits<Polygon_exact>() );
#endif // CGAL_USE_LEDA
\end{verbatim}

The following piece of code actually performs all point-in-polygon tests with the selected strategies and reports the results using tools defined in Section 6. For each of the defined strategies we test whether it has been selected and if so we perform the point location queries for this strategy and the given polygon.

\textit{compute and report point location results} ≡
\begin{verbatim}
std::vector<bool> B;
if ( algos.find("crossings") != algos.end() )
{
    std::cout << std::setw(17) << "crossings";
    B = compute_pip_results( VV.begin(), VV.end(), QV.begin(), QV.end(),
                             Polygon_traits< Polygon_crossings>() );
    report_pip_results( VL, QL, B_correct, B, leda::string("crossings.ps"),
                        visualize, separately );
}
\end{verbatim}

\textit{compute and report point location results}\phantom{x}+ ≡
\begin{verbatim}
if ( algos.find("weiler") != algos.end() )
{
    std::cout << std::setw(17) << "weiler";
    B = compute_pip_results( VV.begin(), VV.end(), QV.begin(), QV.end(),
                             Polygon_traits< Polygon_weiler>() );
    report_pip_results( VL, QL, B_correct, B, leda::string("weiler.ps"),
                        visualize, separately );
}
\end{verbatim}

\textit{compute and report point location results}\phantom{x}+ ≡
\begin{verbatim}
if ( algos.find("halfplane") != algos.end() )
{
    std::cout << std::setw(17) << "halfplane";
    B = compute_pip_results( VV.begin(), VV.end(), QV.begin(), QV.end(),
                             Polygon_traits< Polygon_halfplane>() );
\end{verbatim}
report_pip_results( VL, QL, B_correct, B, leda::string("halfplane.ps"),
    visualize, separately );
}

⟨compute and report point location results⟩≡
if ( algos.find("barycentric") != algos.end() )
{
    std::cout << std::setw(17) << "barycentric";
    B = compute_pip_results( VV.begin(), VV.end(), QV.begin(), QV.end(),
        Polygon_traits< Polygon_barycentric>() );
    report_pip_results( VL, QL, B_correct, B, leda::string("barycentric.ps"),
        visualize, separately );
}

⟨compute and report point location results⟩≡
if ( algos.find("spackman") != algos.end() )
{
    std::cout << std::setw(17) << "spackman";
    B = compute_pip_results( VV.begin(), VV.end(), QV.begin(), QV.end(),
        Polygon_traits< Polygon_spackman>() );
    report_pip_results( VL, QL, B_correct, B, leda::string("spackman.ps"),
        visualize, separately );
}

⟨compute and report point location results⟩≡
if ( algos.find("grid") != algos.end() )
{
    std::cout << std::setw(17) << "grid";
    B = compute_pip_results( VV.begin(), VV.end(), QV.begin(), QV.end(),
        Polygon_traits< Polygon_grid>() );
    report_pip_results( VL, QL, B_correct, B, leda::string("grid.ps"),
        visualize, separately );
}

⟨compute and report point location results⟩≡
if ( algos.find("pnpoly") != algos.end() )
{
    std::cout << std::setw(17) << "pnpoly";
    B = compute_pip_results( VV.begin(), VV.end(), QV.begin(), QV.end(),
        Polygon_traits< Polygon_pnpoly>() );
    report_pip_results( VL, QL, B_correct, B, leda::string("pnpoly.ps"),
        visualize, separately );
}

⟨compute and report point location results⟩≡
if ( algos.find("csg") != algos.end() )
{
    std::cout << std::setw(17) << "csg";
    B = compute_pip_results( VV.begin(), VV.end(), QV.begin(), QV.end(),
        Polygon_traits< Polygon_csg>() );
    report_pip_results( VL, QL, B_correct, B, leda::string("csg.ps"),
        visualize, separately );
}
\begin{itemize}
\item compute and report point location results
\end{itemize} + \equiv
\begin{itemize}
\item if ( algos.find("cgalSCdouble") != algos.end() )
\begin{itemize}
\item std::cout << std::setw(17) << "cgalSCdouble";
\item B = compute_pip_results( VV.begin(), VV.end(), QV.begin(), QV.end(),
Polygon_traits<Polygon_cgalSCdouble>() );
\item report_pip_results( VL, QL, B_correct, B, leda::string("cgalSCdouble.ps"),
visualize, separately );
\end{itemize}
\end{itemize}
\begin{itemize}
\item if ( algos.find("RTR") != algos.end() )
\begin{itemize}
\item std::cout << std::setw(17) << "RTR";
\item B = compute_pip_results( VV.begin(), VV.end(), QV.begin(), QV.end(),
Polygon_traits<Polygon_RTR>() );
\item report_pip_results( VL, QL, B_correct, B, leda::string("RTR.ps"),
visualize, separately );
\end{itemize}
\end{itemize}
\begin{itemize}
\item if ( algos.find("cgalExact") != algos.end() )
\begin{itemize}
\item std::cout << std::setw(17) << "cgalExact";
\item B = compute_pip_results( VE.begin(), VE.end(), QE.begin(), QE.end(),
Polygon_traits<Polygon_exact>() );
\item report_pip_results( VL, QL, B_correct, B, leda::string("cgalExact.ps"),
visualize, separately );
\end{itemize}
\end{itemize}
\begin{itemize}
\item if ( algos.find("rat_points") != algos.end() )
\begin{itemize}
\item std::cout << std::setw(17) << "rat_points";
\item report_pip_results( VL, QL, B_correct, B_correct, leda::string("rat_points.ps"),
visualize, separately );
\end{itemize}
\end{itemize}
\end{itemize}

\begin{itemize}
\item compute and report point location results
\end{itemize} + \equiv
\begin{itemize}
\item std::cout << std::endl;
\end{itemize}

5.6 Data I/O

For the sake of reproducibility of our experiments and for making the use of our test data for related experiments more easy, we provide functions and code fragments that write a test-instance to files. A test instance consist of

\begin{itemize}
\item the polygon data, i.e., the floating-point coordinates of the vertices of a polygon (in counter-clockwise order)
\item the query point data, i.e., the floating-point coordinates of the sequence of query points
\item the correct answers for the above sequence of query points as a sequence of Boolean values
\end{itemize}

The corresponding file names are
6 Reporting Query Results

The program always produces some output for the command line. Fig. 4 shows an example.

\[\text{command line output header} \equiv \]
\[
\text{cout} \ll \text{endl} \ll \text{string(argv[argc-1])} \ll \text{endl} \ll \text{endl;} \\
\text{cout} \ll \text{setw(17)} \ll \text{" ";} \\
\text{cout} \ll \text{setw(8)} \ll \text{" false ";} \\
\text{cout} \ll \text{" \";} \\
\text{cout} \ll \text{setw(10)} \ll \text{" false ";} \\
\text{cout} \ll \text{endl;} \\
\text{cout} \ll \text{setw(17)} \ll \text{" \";} \\
\]
Fig. 4. Sample command line output.

\begin{verbatim}
std::cout << std::setw(10) << "positives";
std::cout << std::setw(12) << "negatives";
std::cout << std::endl;
\end{verbatim}

In addition, visualization can be enabled. We use components from LEDA for visualizing the defects of point-in-polygon testing. The user specifies the look and feel of visualization via command line arguments. The number of command line argument triggers visualization, an S as second argument triggers separate visualization of correctness and fraction of false results, see Fig. 5.

\begin{verbatim}
⟨visualization settings⟩≡
  bool visualize = false;
  bool separately = false;
  if ( argc >= 4 )
    { visualize = true;
      if ( leda::string(argv[2]) == leda::string("S") )
        { separately = true;
        }
    }
\end{verbatim}

The user may specify the colors used for correct, false positive, and false negative results via file colors.txt to be located in the directory where the program is started.

\begin{verbatim}
⟨visualization settings⟩+≡
  read_colors_from_file("colors.txt");
\end{verbatim}

\begin{verbatim}
⟨read colors from file⟩≡
  void
  read_colors_from_file(std::string S)
  {
    leda::color cc = correctcolor;
    leda::color fp = falsepositivecolor;
    leda::color fn = falsenegativecolor;
    try
      {
```
Fig. 5. Calling pip with different visualization arguments: S enables visualization in separate windows, while a V enables visualization in a single window.

```cpp
std::ifstream F(S.c_str());
F >> correctcolor;
F >> falsepositivecolor;
F >> falsenegativecolor;
}
catch(...)
{
    correctcolor = cc;
    falsepositivecolor = fp;
    falsenegativecolor = fn;
}
```

If no file colors.txt exists, default values are used.

```
⟨color defaults⟩ ≡
  leda::color correctcolor = leda::color(176,176,176);
  leda::color falsepositivecolor = leda::color(255,20,20);
  leda::color falsenegativecolor = leda::color(20,255,20);
```

The function below gets the polygon vertices and the query points as lists of leda::point, furthermore references to two vectors of boolean values, where the first contains the correct results of point-in-polygon testing whereas the second contains the results to be compared to the correct ones.
void report_pip_results( leda::list< leda::point> PL,
        leda::list< leda::point> QL,
        std::vector<bool>& correct,
        std::vector<bool>& checked,
        leda::string postscriptfilename,
        bool visualize = false,
        bool separately = false)
{
    Window_postscript_stream* Wp;
    Window_postscript_stream* W2p;
    if ( visualize )
    {
        if ( separately )
        {
            Wp = new Window_postscript_stream(500,500,1.0,postscriptfilename);
            W2p = new Window_postscript_stream(500,100,1.5,
                leda::string("B")+postscriptfilename);
            W2p->init(0.0,1.0,1.0);
            W2p->display();
        }
        else
        {
            Wp = new Window_postscript_stream(500,575,1.5,postscriptfilename);
            W2p = Wp;
        }
    }
    Wp->init(0.0,1.0,0.0);
    Wp->display();
    visualize_pip_results(PL, QL, correct, checked, Wp);
    double fp, fn;
    fraction_of_false_results( correct, checked, fp, fn);
    std::cout << std::setw(8) << std::fixed << std::setprecision(1) << fp*100 << " % ";
    std::cout << std::setw(8) << std::fixed << std::setprecision(1) << fn*100 << " % ";
    //std::cout << "false_positives: " << fp*100 << " % ";
    //std::cout << "false_negatives: " << fn*100 << " % ";
    std::cout << std::endl;
    if ( visualize )
    {
        visualize_false_fraction( fp, fn, W2p);
        Wp->read_mouse();
        if ( separately )
        {
            delete W2p;
        }
        delete Wp;
    }
}

void visualize_pip_results( leda::list< leda::point> PL,
        leda::list< leda::point> QL,
        std::vector<bool>& correct,
        std::vector<bool>& checked,
        Window_postscript_stream* Wptr )
{
    leda::point p;
    leda::point q = PL.front();
    forall(p,PL)
    { Wptr->draw_segment(p,q,leda::black); }
\begin{verbatim}
q = p;
}
Wptr->draw_segment(PL.front(), PL.back(), leda::black);
Wptr->set_point_style(leda::box_point);
assert( QL.size() == correct.size() );
assert( QL.size() == checked.size() );
int i = 0;
forall(p,QL)
{ if ( correct[i] == checked[i] )
{ Wptr->draw_point( p.xcoord(), p.ycoord(), correctcolor);
} ++i;
}
Wptr->set_point_style(leda::disc_point);
i = 0;
forall(p,QL)
{ if ( correct[i] != checked[i] )
{ if ( checked[i] )
{ Wptr->draw_point( p.xcoord(), p.ycoord(), falsepositivecolor);
} else
{ Wptr->draw_point( p.xcoord(), p.ycoord(), falsenegativecolor);
}
} ++i;
}
\end{verbatim}

The following function computes the percentages of errors.

\begin{verbatim}
⟨ count false results ⟩ ≡
void
fraction_of_false_results( std::vector<bool>& correct,
                          std::vector<bool>& checked,
                          double& false_positives_fraction,
                          double& false_negatives_fraction )
{
int false_positives = 0;
int false_negatives = 0;
int i;
for ( i = 0; i < correct.size(); ++i )
{ if ( correct[i] != checked[i] )
{ if ( checked[i] ) false_positives++;
else false_negatives++;
}
false_positives_fraction = (double)false_positives/i;
false_negatives_fraction = (double)false_negatives/i;
}
\end{verbatim}

\begin{verbatim}
⟨ visualization of results ⟩ +≡
void
visualize_false_fraction( double fp, double fn,
                          Window_postscript_stream* Wptr )
{
Wptr->draw_filled_rectangle( leda::point( 0.0, 1.05),

32
7 The Source File(s)

or putting it all together.

We have to include some files from CGAL for point type, polygon type, query point generation and inexact Cartesian kernel as well as an exact CGAL kernel.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
```

```
#include <vector>
#include <list>
#include <string>
```

```
#define CGAL_USE_LEDA
#include <LEDA/geo/point.h>
#include <LEDA/core/list.h>
// CGAL_USE_LEDA
```

```
#include <ptinpoly.h>
#include <csg.h>
#include <csg_tree.h>
```

```
#include <Window_postscript_stream.h>
```

```
typedef
  (typedef for cgal point type with double coordinates)
```
(global function definitions) ≡
 { little helpers }
 { query point generators }
 { create polygon and answer queries }
 { count false results }
 { visualization of results }
 { reporting results }
 { read colors from file }

(main) ≡
 { includes }
 { typedefs }
 { polygon classes }
 { point in polygon traits }
 { color defaults }
 { global function definitions }
 int
 main(int argc, char** argv)
 { if ( argc < 2 ) return EXIT_FAILURE;
  { containers }
  { read polygon vertex coordinates from file }
  { create query points }
  { write polygon to file }
  { write query points to file }
  { read query points from file }
  { convert points }
  { select algorithms }
  { compute correct results }
  { write correct location bits to file }
  { command line output header }
  { visualization settings }
  { compute and report point location results }
  return EXIT_SUCCESS;
 }

8 Partial Analysis of Failures

STILL TO BE WRITTEN!
“simulation of simplicity” \Rightarrow inconsistent answers for vertex queries
References

A More Stuff

A.1 Declaration of Used Functions

\[
\text{⟨declaration of functions from Haines⟩} \equiv \\
pPlaneSet \\
\text{PlaneSetup}( \text{double pgon}[][2], \text{int numverts} ); \\
\text{void} \\
\text{PlaneCleanup}( \text{pPlaneSet p]\_plane_set} ); \\
pSpackmanSet \\
\text{SpackmanSetup}( \text{double pgon}[][2], \text{int numverts, int *p_numrec} ); \\
\text{void} \\
\text{SpackmanCleanup}( \text{pSpackmanSet p_spackman_set} ); \\
\text{void} \\
\text{GridSetup}( \text{double pgon}[][2], \text{int numverts, int resolution, pGridSet p_gs} ); \\
\text{void} \\
\text{GridCleanup}( \text{pGridSet p_gs} ); \\
\text{int} \\
\text{CrossingsTest}( \text{double pgon}[][2], \text{int numverts, double point}[2] ); \\
\text{int} \\
\text{WeilerTest}( \text{double pgon}[][2], \text{int numverts, double point}[2] ); \\
\text{int} \\
\text{PlaneTest}( \text{pPlaneSet p]\_plane_set}, \text{int numverts, double point}[2] ); \\
\text{int} \\
\text{BarycentricTest}( \text{double pgon}[][2], \text{int numverts, double point}[2] ); \\
\text{int} \\
\text{SpackmanTest}( \text{double anchor}[2], \text{pSpackmanSet p_spackman_set}, \\
\text{int numrec, double point}[2] ); \\
\text{int} \\
\text{GridTest}( \text{pGridSet p_gs, double point}[2] );
\]
# Table of Contents

1 Introduction ............................................. 1

1.1 How To (Re)Run The Experiments .......................... 2

2 Practical Point-in-Polygon Strategies ......................... 2

2.1 Polygon Concept and Models .............................. 3

2.2 Some Shared Code Fragments .............................. 4

2.3 crossings ............................................ 5

2.4 weiler ............................................. 6

2.5 halfplane .......................................... 6

2.6 barycentric ......................................... 7

2.7 spackman .......................................... 8

2.8 grid ............................................... 9

2.9 PNPOLY ........................................... 10

2.10 csg ............................................... 11

2.11 no-sort-csg ...................................... 12

2.12 RTR .............................................. 12

2.13 CGAL with simple Cartesian kernel and double precision floats ............... 13

2.14 LEDA’s rational kernel ................................ 13

2.15 CGAL’s exact kernel .................................. 14

3 Simple Polygons ........................................ 14

4 Query Points .......................................... 15

4.1 Random Query Points .................................. 16

4.2 Query Points (almost) on the Polygon Edges ................ 17

4.3 Query Points on the Supporting Lines of Polygon Edges ................ 19

4.4 Query Points on Triangle-Fan Diagonals ................... 19

4.5 Query Points at Cross Lines at Polygon Vertices ............. 20

4.6 Query Points near Polygon Vertices ........................ 21

4.7 Little Helpers ...................................... 21