MCS 481 / David Dumas / Spring 2011

Project 1: Convex hulls and segment intersection
Due at 2pm on Monday, February 7

0. Requirements

In order to complete this project you will need CGAL and a compatible C++ build environment. For some experiments you will need the ability to accurately measure the runtime of programs.

1. Project overview

The project has two parts, *convex hulls* and *segment intersection*. For each part you can choose either an *experimental option* in which you will work with CGAL and its example programs, or a *coding option* involving the implementation of an algorithm in a computer programming language.

If you choose a coding option you may use any approved programming language or computer algebra system (CAS) for your implementation. The following languages and CASs are pre-approved:

```
C C++ Python Perl Java
Sage Octave Mathematica Maple Matlab
```

Contact me for approval if you want to use another language or CAS. Your code should not use any built-in computational geometry functions. You may use built-in list manipulation functions, such as a sorting algorithm.

A bit of advice:

- If you have limited programming experience, think carefully before choosing a coding option. With the experimental option, you start with a working program and analyze it, whereas in the coding option you need to write and debug your own program.
- Decide which options you intend to complete as soon as possible.

2. Convex Hulls

Experimental option. Test one of the convex hull algorithms available in CGAL for four classes of input data:

(1) n random points in the unit square
(2) n random points in the unit disk
(3) n random points on edges of the unit square
(4) n random points on the unit circle

In each case, measure the runtime for various values of n and report the results (as a table or graph). Use a range of values so that runtimes vary from milliseconds to more than 10 seconds. Do you see $n \log n$ asymptotics? Is the runtime sensitive to the type of input?

You can use the example program `ch_from_cin_to_cout` from CGAL for these experiments. This program is located in the `examples/Convex_hull_2` subdirectory of the

source distribution. (The example programs may be located elsewhere if you installed CGAL as a binary package rather than compiling from source.) This program reads a list of coordinates from stdin, formatted like this:

```
1.2 3.4
5.6 7.8
-2.2 0.1
0.5 0.9
```

Thus, for example, in a linux terminal the command

```
./ch_from_cin_to_cout < input.txt > output.txt
```

reads points from `input.txt` and writes the convex hull vertices to `output.txt`, and

```
time ./ch_from_cin_to_cout < input.txt > output.txt
```

will do the same but also report the amount of time used by the program. In both cases it is assumed that you have compiled the sample programs using `cmake`.

For hints on how to generate suitable input files, see the source listings at the end of this document.

Note: If you prefer, you can automate these experiments by writing your own program that calls CGAL functions. With instructor approval, it is also permissible to use CGAL bindings to another programming language.

Coding option. Implement the algorithm ConvexHull from pp6–7 of the textbook. Write your code "from scratch", that is, do not base it on an existing implementation. Test your implementation on several small non-degenerate datasets (e.g. two points, three non-collinear points, five points on the unit circle, ...) and verify the correctness of the output. Analyze the robustness of your algorithm by testing on several degenerate cases (e.g. points that lie on a single vertical line, points on a non-vertical line, 10 points on the edges of a triangle).

3. Segment intersection

Experimental option. Study the CGAL implementation of the plane sweep segment intersection algorithm. Experiment with each of the following classes of input:

(1) n segments defined by $2n$ random points on the unit circle
(2) n random diameters of the unit circle (so all segments intersect at the origin, and nowhere else)
(3) n random diameters of the unit circle shifted by small, random vectors of a fixed small norm ϵ (so all segments come very close to the origin)

In each case, study a number of values of n and record both the runtime and the total number of intersections I. Analyze the results and attempt to determine:

- For which classes of input do you observe $O((n + I) \log n)$ asymptotics?
- For each class, how does I behave as a function of n?

The CGAL example program `sweep_line.cpp` (located in `examples/Arrangement_2/` or `examples/Arrangement_on_surface_2/` depending on CGAL version) computes all intersections between a sample collection of segments and prints the results. The dataset is fixed in the source code, and a rational number library is used for exact arithmetic. With minor modifications this example can be adapted to the experiments described

above, i.e. to read a list of segments from stdin, find their intersections using floating-point arithmetic, and print the number and location of intersection points to stdout. See the source code listing at the end of this document for details.

Coding option. Implement the naïve algorithm for reporting all intersection points for a set of n segments S and listing the segments containing each intersection point. Your algorithm should simply check all pairs of segments, record which pairs intersect and where, and then sort the results by intersection point in order to give the desired output.

Test your algorithm on several small configurations where you can compute the correct output by hand (e.g. segments with one common endpoint, disjoint segments, several vertical segments intersecting one horizontal segment, etc.). Verify that the output is correct in these cases. Also test the robustness of your algorithm for some types of degenerate input (e.g. all segments contained in a line, or several disjoint parallel segments).

4. How to submit your project

For an **experimental option**, you must submit:

- A document (printed or email PDF) with:
  - Table or graph of runtime data
  - Your interpretation of the results
  - Source code for all programs you wrote or modified for these experiments
- An archive (.tar.gz or .zip by email) of:
  - Source code for all programs you wrote or modified for these experiments

For a **coding option**, you must submit:

- A document (printed or email PDF) with:
  - A description of the algorithm you use, including pseudocode
  - A brief discussion of how you implemented the algorithm
  - Input and output listings for the test cases you used
  - A statement to the effect that you are the sole author of the code you are submitting
  - Source code of your implementation
- An archive (.tar.gz or .zip by email) of:
  - Source code of your implementation
  - Input files for the test cases

5. Code listings

5.1. **Random points in the unit disk.** The following program prints a list of 50 random points in the unit disk.

```
// rand-disk-points.cpp
// MCS 481 project 1 description version 1.0
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <iostream>

using namespace std;

double dblrand()  // return pseudorandom double in [0.0,1.0).
{
  return rand()/(double(RAND_MAX) + 1);
}

int main()
{
  srand((unsigned)time(0)); // Seed the random generator with current time

  const int numpoints = 50;
  for (int i=0; i<numpoints; i++) {
    // Generate points in the square [-1,1)x[-1,1) until we find one
    // that lies in the unit disk.

    // This is a bad algorithm, in general, but we use it because both
    // the code and the underlying mathematics are very simple.
    double x,y;
    do {
      x = 2.0*dblrand() - 1.0;  // rescale to get random number in [-1,1)
      y = 2.0*dblrand() - 1.0;
    } while (fabs(x*x + y*y) > 1.0);
    cout << x << " " << y << endl;
  }
}
```

5.2. **Random points on the edges of the unit square.** The following program prints a list of 50 random points on the edges of the unit square.

```
// rand-square-edge-points.cpp
// MCS 481 project 1 description version 1.0
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <iostream>

using namespace std;

double dblrand()  // return pseudorandom double in [0.0,1.0).
{
  return rand()/(double(RAND_MAX) + 1);
}
```

```
bool flip() // flip a coin
{
  double x = dblrand();
  return (x < 0.5);
}

int main()
{
  srand((unsigned)time(0)); // Seed the random generator with current time

  const int numpoints = 50;
  for (int i=0; i<numpoints; i++) {
    double x,y;
    // First generate a pair:
    // x = a random number in [0,1)
    // y = either 0 or 1, equal probability
    x = dblrand();
    if (flip())
      y = 1.0;
    else
      y = 0.0;
    // Now flip a coin to decide whether or not to swap x and y
    if (flip()) {
      double z = x;
      x = y;
      y = z;
    }
    cout << x << " " << y << endl;
  }
}
```

5.3. **Intersections from a list of segments.** This modified version of the CGAL example program `sweep_line.cpp` reads segments from stdin instead of using a fixed dataset, and it works with floating-point types instead of exact rational numbers.

```
// sweep-line-cin.cpp
// Modified version of CGAL example 'sweep_line.cpp'
// MCS 481 project 1 description version 1.0
//
// Read segments from stdin -- one segment per line, formatted as
// px py qx qy
//
// Print number of segments read, list of intersection points, and
// total number of intersection points.


#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Sweep_line_2_algorithms.h>
#include <list>

// Use "inexact" kernel for approximate results with floating-point input
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_2                                     Point_2;
typedef CGAL::Arr_segment_traits_2<Kernel>                  Traits_2;
typedef Traits_2::Curve_2                                   Segment_2;
```

```
int main()
{
  CGAL::set_ascii_mode(std::cin);

  // Read the segments from stdin and store them in a list.
  std::list< Segment_2 > segments;
  Point_2 p,q;

  for (std::istream_iterator< Point_2 > i(std::cin);
       i != std::istream_iterator< Point_2 >();
       i++) {
    p = *i; i++; q = *i;
    Segment_2 s(p,q);
    segments.push_back(s);
  }
  std::cout << "Read " << segments.size() << " segments." << std::endl;

  // Compute all intersection points.
  std::list<Point_2>     pts;

  CGAL::compute_intersection_points (segments.begin(), segments.end(),
                                     std::back_inserter (pts));

  // Print the result.
  std::cout << "Intersection points: " << std::endl;
  std::copy (pts.begin(), pts.end(),
             std::ostream_iterator<Point_2>(std::cout, "\n"));

  std::cout << "Total number of intersection points: " << pts.size() << std::endl;

  // The CGAL example continues to compute non-intersecting
  // sub-segments, but we stop here.

  return 0;
}
```