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 running time of programs.

1. Project overview

The project has two parts, convex hull and line 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. (It is acceptable to choose the coding option for one part and the experimental option for the other.)

If you choose a coding option you may use any approved programming language. The following languages (and versions) are pre-approved:

- C
- C++
- Python(2.x)
- Perl(5.x)
- Java(SE6/1.6.x)

Contact me for approval if you want to use another language. Your code must not use any computational geometry functions from built-in or add-on libraries. That is, you must implement all of the geometric primitives that your project requires. You may use built-in support for basic data structures and related operations (lists, sorting algorithms, etc.).

You may also choose the coding option and write your program in a computer algebra system (CAS) such as Sage, Mathematica, Matlab, or Maple, but if you intend to pursue this option contact me for details and approval.

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.
- A key component of the experimental option will be answering the question: *How can you distinguish between growth rates O(n) and O(n log n) using a collection of function values?*

2. Convex Hull

**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

Project description v1.0 (January 16, 2012)
In each case, measure the running time for various values of \( n \) and report the results (as a table or graph). Use a range of values so that running times vary from less than 0.01 seconds to more than 10 seconds. Do you see \( n \log n \) asymptotics? Is the running time sensitive to the type of input? If so, why?

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 standard input, 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.

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 C++ program that calls CGAL functions and reports their running times. Alternatively, you could write a script (in the shell or with an auxiliary programming language) to automate the execution of the CGAL example program and collection of running time data. However, the total number of runs is expected to be small enough so that it is possible to collect the data in a reasonable amount of time even without any automation.

**Coding option.** Implement a convex hull algorithm that has running time \( O(n \log n) \) or \( O(nh) \), where \( n \) is the number of input points and \( h \) is the number of convex hull vertices. You should probably choose either Graham’s scan, the Graham-Andrew variation from the textbook, or the Jarvis march from problem 1.7. (A naive algorithm that checks all pairs of points will be accepted for partial credit.)

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. Line 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 $\epsilon$ (so all segments come very close to the origin)

In each case, study a number of values of $n$ and record both the running time 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 standard input, find their intersections using floating-point arithmetic, and print the number and location of intersection points to standard output. See the source code listing at the end of this document for details.

**Coding option.** Implement an algorithm for reporting all intersection points for a set of $n$ segments $S$ and listing the segments containing each intersection point. You can use any algorithm you like, e.g. 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. Keep in mind that you will first need to implement the geometric primitive for intersecting two line segments.

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**

Email submission (ddumas@math.uic.edu) is required for source code and preferred for all materials.

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

- A report (PDF or plain text) with:
  - Table or graph of running time data
  - Interpretation of the results (answer questions from the description above)
- 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 report (PDF or plain text) 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
- An archive (.tar.gz or .zip by email) of:
  - Source code of your implementation
  - Input files for the test cases
Requirements for all source code submissions:

- For each source file that you write, one of the first five lines must be a comment of the following form (adapted to the comment syntax of the language you use):

  // MCS 481 Project 1, Spring 2012, by NAME

- Each source file that you modify from the CGAL examples or the code included with this project description should have:
  - A descriptive file name.
    For example, if you modify rand-disk-points.cpp so that it produces random points on the circle, you should change the name to something like rand-circle-points.cpp
  - A comment on one of the first five lines indicating that you modified it, e.g.

// CGAL example modified for MCS 481 Project 1, Spring 2012, by NAME

5. Extra credit opportunities

You should focus on completion of the required tasks described above, but a modest amount of extra credit may be awarded for including any of the following:

- Completion of the convex hull experimental option for at least two different convex hull algorithms that CGAL offers, with comparative analysis of the results.

- Mathematical analysis of the expected number of convex hull vertices for random input of types (1) and (2), i.e. random points in the square or disk.

- Thorough experimental analysis of the distribution (histogram) of numbers of convex hull vertices for random input of types (1) and (2).

- When choosing a coding option, automating testing of your program by feeding the same input to it and to a corresponding program that uses CGAL, then comparing the results.
6. Code listings

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

```cpp
#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 not an efficient algorithm (it has infinite worst-case
        // running time!) but we use it because the code is particularly
        // simple and efficiency of this component is not a focus of the
        // project.

        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;
    }
}
```
6.2. 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.

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

```cpp
// sweep-line-cin.cpp
// MCS 481 project 1 description version 1.0
// Modified version of CGAL example 'sweep_line.cpp'
//
// Read segments from stdin, one per line in format: 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 original CGAL example does more computations, but we stop here.
    return 0;
}
```