* [Skip navigation](#content)
* [Skip to search form](#q)

# **April Column: Computational Geometry**

By [George T. Heineman](http://www.oreillynet.com/pub/au/3414)  
May 13, 2009 | [Comments: 0](http://broadcast.oreilly.com/2009/05/april-column-computational-geo.html#comments)

This is the sixth of a series of monthly columns in the blog associated with the [**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/) book, published October 2008 by [**O'Reilly Media, Inc.**](http://oreilly.com/)

[**In our last column**](http://broadcast.oreilly.com/2009/04/march-network-flow-algorithms.html) we described key algorithms used to solve network flow problems.

In this column we present an algorithm from Computational Geometry showing how to detect all intersections among a set of line segments in the two-dimensional plane. I first saw this algorithm when I was an undergraduate and knew that I had to include it in our book. There are important details of the algorithm to be worked out, and we refer you to the book and the ADK for these issues. For this column we focus on the following points:

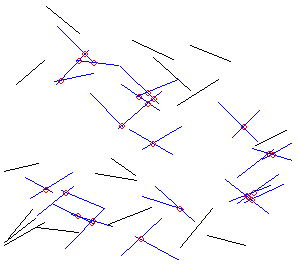
* An O(n2) **brute force** algorithm exists. We show a LineSweep algorithm that does better.
* Under what [**input sets**](#InputSets) will LineSweep outperform the **brute force** approach? When will it be inefficient?
* How do **brute force** and LineSweep perform under [**stress tests**](#stressTests) that attempt larger and larger problems?

We end with a general discussion of how to evaluate and test algorithm implementations that seem to defy a straightforward unit test approach.

**Download April Code Samples**

You can download the code samples described in this column from the *code.zip* file found at [**code.zip**](http://examples.oreilly.com/9780596516246/Blogs/April_2009/code.zip) (83,495 bytes). The following examples were tested on a standard Windows desktop computer running [**Eclipse Version 3.4.1**](http://www.eclipse.org/). The [**JDK version**](http://java.sun.com/javase/6/) is 1.6.0\_13. In fact, the *code.zip* file is actually an exported Eclipse project, which means you will be able to easily load this month's code into your Eclipse workspace. Should you choose to compile and execute these examples outside of Eclipse, simply ensure that your CLASSPATH variable is properly configured to use the compiled sources of the ADK.

To bring this month's code into Eclipse, simply unzip the code.zip file found at code.zip. Once you have unzipped the files, in Eclipse choose to create a New Java Project. For the "Project name" enter "April 2009". Then choose the "Create project from existing source" radio button and browse to the location where you unzip'd the code.zip file. Make sure you select the directory named "April\_2009". Then click Finish. If you have already imported the JavaCode project from the ADK into this Eclipse workspace, then the code should compile cleanly; if you have not, then you will need to tell this Eclipse project about the location of the compiled Jar file for the ADK in which the implemented algorithms can be found.

  
**Fig. 1** Sample line segments with intersections marked.

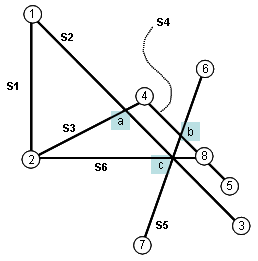
**Intersecting Line Segments**

**Fig. 1** contains an example. Given a set of n two-dimensional line segments, a **brute force** algorithm can detect all intersection points by simply comparing all possible line segment pairs; unfortunately, **brute force** must check n \* (n - 1)/2 such pairs. The **brute force** algorithm doesn't take any advantage of the placement of the lines and thus will repeatedly compare lines that clearly have no chance of intersecting. Is this wasted effort a big deal? Well, given a problem that is 1,000-times larger, you must perform 499,500 times as much effort. Such an algorithm can be classified as O(n2) since it will require on the order of n2 computations. Throughout the book we have sought to find O(n log n) algorithms, because when a problem is 1,000-times larger, you only have to perform about 10,000 times as much work.

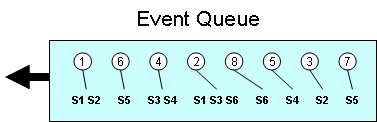
The problem for intersecting line segments is that there doesn't seem to be an obvious way to decompose the problem into smaller problems as we have shown in past columns for [**sorting**](http://broadcast.oreilly.com/2008/10/welcome-to-algorithms-in-a-nut.html) and [**searching**](http://broadcast.oreilly.com/2008/12/december-column-searching-algo.html). A powerful technique developed by [**Shamos and Hoey**](http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=4567905) in the 1970s shows how to "sweep" an infinitesimally thin line over the plane of line segments to locate and report all k intersections in O((n+k) \* log n)). In this column we sketch the details of the implementation which can be found in the [**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/) book and the ADK. This algorithm has many special cases and is challenging to test. Indeed we realized that the implementation still reported unexpected results when the book was ready for publication, and we lamely chalked that up to "issues with floating point analysis". Only recently we investigated the implementation further and detected the underlying reasons for the apparent discrepancies between the **brute force** algorithm and LineSweep. This investigation prompted the topics for this column.

**LineSweep algorithm**

To describe the LineSweep algorithm we need to describe two concepts: (a) an event queue of points that dictates the coordinates of the sweeping line; and (b) the state of the sweeping line. The example described in **Fig. 2** is drawn from the book. Can you count the number of intersections in the figure below?

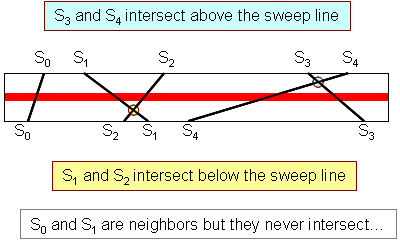
  
**Fig. 2** Sample set of six intersecting line segments.

You can verify that there are seven intersections. Indeed, as you were counting these intersections, you likely "swept" down the image, identifying intersection points along the way in systematic fashion so you were sure not to miss any. One observation to make is that we only need to be concerned with "interesting points", namely the beginning or ending point of a line segment or an actual intersection point. Again, from **Fig. 2** there are 11 interesting points. How can we process these points systematically? Let's consider using a priority queue as an event queue, where the y-coordinate of a point is used to determine priority; should two points have the same y-coordinate value, then break ties using the x-coordinate (from left to right). Initially, the event queue will contain up to 2\*n points, as drawn from the beginning and ending points of the set of line segments. There may be less than 2\*n points because the line segments may actually have intersections among these end points (note in **Fig. 2** that there are eight points, labeled 1-8, out of a potential 12 one would have with six line segments). The initial state of the event queue is shown in **Fig. 3**

  
**Fig. 3** Initial state of event queue.

The key idea is that every point p in the event queue has three pieces of associated information: (a) the line segments whose upper endpoint is p; (b) the line segments whose lower endpoint is p; and (c) those line segments that actually intersect exactly at p. For each line segment, we consider the endpoint with higher y-coordinate as the "upper" and the endpoint with lower y-coordinate as the "lower". For horizontal lines, we break ties by considering the left-most end point as the "upper" and the right-most end point as the "lower".

The LineSweep algorithm maintains a designated line state as the line sweeps vertically from the top of the plane down to the bottom. The sweep point of the line is determined by the next point extracted from the event queue. The line state stores those lines that are "active" at the coordinate of the sweeping line. The key observation is that if two segments intersect then at some point they were neighbors on sweep line. However, if two line segments are neighbors on the sweep line, it doesn't automatically guarantee that the line segments will intersect. In **Fig. 4**, for example, there are five line segments that "touch" the sweeping line at the given sweep point identified by the horizontal red line.

  
**Fig. 4** Sample line sweep state.

The state of the line can be read from left to right as a sequence of line segments. Thus in **Fig. 4**, the state before the two identified interactions is "S0 - S1 - S2 - S3 - S4". Note that after an intersection, the relative position of two line segments on the line state swaps. Thus the state after both intersections is S0 - S2 - S1 - S4 - S3".

We now have sufficient information to sketch out the algorithm, whose details can be found in the book. Once the initial event queue is constructed, the coordinate of the "sweep line" is defined by the next point p extracted from the event queue. If there are any associated line segments whose upper endpoint is p, the line state is updated to include them. If there are any associated line segments whose lower endpoint is p, the line state is updated to remove them. All intersections are computed by inspecting within the line state to find the neighboring line segment to the left of p and the neighboring line segment to the right of p. Using this strategy, only neighboring line segments are ever inspected for intersections.

When an intersection point ip below the sweep line is detected between two line segments, the point ip is simply inserted into the event queue to be processed by the sweep line in due course. In this way, no intersection point is skipped, and if more than one pair of line segments intersects at the exact same point, the event queue will merge the information to ensure that no duplicate points are reported.

The algorithm performs in O((n+k) log n) time by using a balanced binary tree to represent the event queue and, similarly, a balanced binary tree to represent the line state during processing. The increased complexity of the code is balanced by the improved performance bounds. We hope you are motivated by this brief sketch to learn more about the full implementation, which shows the full details.

**Evaluating** **Input Sets to compare performance of LineSweep and brute force**

The key factor to determine the performance of the algorithm variations is k, the number of intersections. Since this value cannot be known in advance one must try to estimate the likelihood of an intersection. This is especially important since k can be as high as n\*(n-1)/2.

When the number of intersections begins to approach the maximum O(n2), you should simply use the **brute force** search. Even when the number of intersections is as large as n, the LineSweep algorithm is the one to choose.

**Stress Tests**

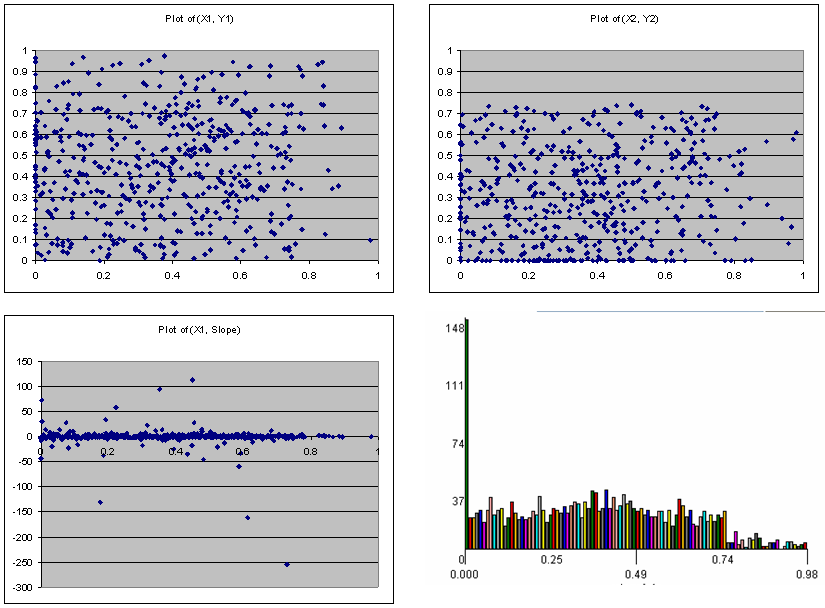
After all unit testing showed the algorithms performed as expected on sample trial data, we generated sets of random line segments of length 0.25 drawn randomly from the [0,1] unit square (run the Generator program included in this month's blog code). We ran this test to benchmark the performance of the algorithms with data sets of increasing size. We were surprised, to say the least, that as the number of line segments increased, the two algorithms occasionally returned different results (with increasing frequency as n increased in size); LineSweep even started throwing NullPointerExceptions for large n.

|  |  |  |
| --- | --- | --- |
| n | Brute Force | LineSweep |
| 128 | 435 | 435 |
| 256 | 1545 | 1541 |
| 512 | 6,412 | 6,410 |
| 1,024 | 25,804 | 25,794 |
| 2,048 | 108,306 | 108,276 |

**Table 1:** Detected intersections for same input set with increasing n.

Repeated debugging attempts failed to locate the problem; as the book went to publication in October 2008, we simply modified the code to trap the exceptions and report "LineSweep produced invalid computation". The documentation for the LineSweep implementation was modified to report that "floating point calculations may lead to inconsistent state".

As I prepared this month's column, I decided to investigate this problem further. Was it the number of line segments that caused the problem? or did the number of line segments increase enough that some configuration of line segments appeared in the data to thwart the LineSweep algorithm? or was there some special property of the line segments themselves? I recorded one of the 512-sized sample data sets that caused the problem and constructed several images. For each line segment (X1,Y1,X2,Y2) I plotted (X1 vs. Y1), (X2 vs. Y2) and (X1 vs. slope); I also constructed a histogram of the 2,048 coordinate values. These visualizations were manually constructed in Microsoft Excel.

  
**Fig. 5** Visualizing the input set.

Viewing the images in **Fig. 5**, there seems to be no apparent pattern in the line segment coordinates themselves, but note how in the lower-left image, there are about ten outlier points whose slope is completely different. One line segment in particular (0.729318413, 0.514685003, 0.73029922, 0.264686927) has a slope of -254.89; also note in the histogram (in the lower-right image) that there appear to be 148 coordinates that fall in the left-most bin [0, 0.01). Which of these factors is the reason for the discrepancy?

It turns out that "0.0" is far and away the most common coordinate in the input set and six line segments all share (0.0,0.0) as an endpoint; clearly, the "random generation" of line segments fails to uniformly distribute the endpoints properly. Removing these six line segments from the input data set ensures that both algorithms return the same result. Why did this happen? The issue occurs because of floating point arithmetic. To confirm this point, run the Validate class included in the code for this month's blog entry. The code checks for intersections among three line segments:

* (0.145721433,0.105568686) to (0,0)
* (0.079389892,0.043921955) to (0,0)
* (0.094178956,0.036624491) to (0,0)

The first output is from LineSweep while the second is from **brute force**. Because of the small errors that accumulate within floating point computations, **brute force** mistakenly determines there are two intersections.

1 intersection points.

0.0,0.0:

<0.145721433,0.105568686,0.0,0.0>

<0.079389892,0.043921955,0.0,0.0>

<0.094178956,0.036624491,0.0,0.0>

2 intersection points.

2.7755575615628914E-17,1.3877787807814457E-17:

<0.079389892,0.043921955,0.0,0.0>

<0.094178956,0.036624491,0.0,0.0>

-2.7755575615628914E-17,-2.7755575615628914E-17:

<0.145721433,0.105568686,0.0,0.0>

<0.079389892,0.043921955,0.0,0.0>

<0.094178956,0.036624491,0.0,0.0>

The only way to prevent this is to put in a post-processing step which revisits each of the found intersection points and merges those points whose coordinates are simply "too close" to each other. The obvious implementation of such a post-processing step requires O(k2) where k is the number of intersections. It is likely that one could construct a more efficient approach.

Was the outlying slope a red herring? Well it reveals another underlying problem, but one that only appears when the absolute value of the slope is excessive. To confirm this point, run the NearVerticalProblem class included in the code for this month's blog entry. In this small example, a line appears whose slope is -231,477,104! When executing, the following output appears, showing how **brute force** continues to find two intersection points

slope of 0th line is -2.31477104788506E8

slope of 1th line is -1.86755802238522

slope of 2th line is -0.0724060246888423

0 intersection points.

LineSweep produced invalid computation for:<0.21,0.0: up:, low:<0.079389892,0.243921955,0.21,0.0>, inter:>

2 intersection points.

0.14572143293746567,0.12004395358506237:

<0.14572143,0.8,0.145721433,0.105568686>

<0.079389892,0.243921955,0.21,0.0>

0.14572143115392622,0.5328925052714808:

<0.14572143,0.8,0.145721433,0.105568686>

<0.094178956,0.536624491,0.6,0.5>

Fortunately, these sort of issues can be resolved by pre-processing the data set to "normalize" line segments that are nearly vertical (based upon some pre-defined maximum negative or positive slope value). The impact on the actual intersecting points is going to be minimal and this pre-processing step can be performed in O(n) time.

In general, you can use both pre- and post-processing steps to improve upon the performance of the algorithms and the quality of the results.

**Next Column**

We have now completed one blog entry for each core chapter of the book. We encourage you to go back and read these entries:

* [**March Column: Network Flow Algorithms**](http://broadcast.oreilly.com/2009/04/march-network-flow-algorithms.html)
* [**February Column: Improving Performance of Algorithms**](http://broadcast.oreilly.com/2009/02/february-column-improving-algo.html)
* [**January Column: Algorithm to Solve FreeCell Solitaire Games**](http://broadcast.oreilly.com/2009/01/january-column-graph-algorithm.html)
* [**December Column: Searching Algorithms**](http://broadcast.oreilly.com/2008/12/december-column-searching-algo.html)
* [**November Column: Welcome to Algorithms in a Nutshell**](http://broadcast.oreilly.com/2008/10/welcome-to-algorithms-in-a-nut.html)

In next Month's May column, we will investigate the IntroSort sorting algorithm which did not make it into the actual book itself. Until next time, we hope you take the opportunity to investigate the numerous algorithms in the Algorithms in a Nutshell book as well as to explore the examples provided in the ADK.

[**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/)  
George T. Heineman, Gary Pollice, Stanley Selkow