Counting Circles Without Computing Them

Rudolf Fleischer

GUtech, Muscat, Oman; and
Fudan University, Shanghai, China
`rudolf.fleischer@gutech.edu.om`

──── **Abstract** ────

In this paper we engineer a fast algorithm to count the number of triangles defined by three lines out of a set of n lines whose circumcircle contains the origin. The trick is *not* to compute any triangles or circles.

1   Introduction

I am a lousy programmer. I think I have a solid theoretical knowledge of basic and clever data structures and algorithms, but solving problems quickly and implementing them error-free on the first attempt (versus finding logical and programming errors incrementally in repeated rounds of trial-and-error) is a completely different story. I am in good company here, even experienced algorithmicists can, even without the pressure of a relentlessly ticking clock, blunder and design (and sometimes publish) wrong algorithms [5].

Since we recently started to train student teams for the ACM Collegiate Programming Contest [1], We found it appropriate to do a little bit of programming training ourselves to better understand why our students sometimes do not manage in a contest to solve problems that are seemingly straightforward to solve. Simple answer: even problems that appear simple from the elevated viewpoint of a theoretician can be tricky to solve in a contest if the contestant lacks programming experience.

The best way to gain experience is to solve many problems from previous contests, and one platform that provides an excellent training environment is the Codeforces website [10] that not only gives access to many old contest problems but also regularly hosts contests with newly designed problem sets. Usually, no model solutions are provided for the contest problems, and reverse engineering participants solutions to understand the underlying algorithms can be frustratingly difficult because submissions are often highly optimized and nearly always without any comments.

In a recent Codeforces contest [10] the following problem, titled *Ruminations on Ruminants*, was given in category D, the second highest difficulty level [2].

> Kevin Sun is ruminating on the origin of cows while standing at the origin of the Cartesian plane. He notices n lines $\ell_1, , \ell_2, \ldots, \ell_n$ on the plane, each representable by an equation of the form $ax + by = c$. He also observes that no two lines are parallel and that no three lines pass through the same point.
>
> For each triple (i, j, k) such that $1 \le i < j < k \le n$, Kevin considers the triangle formed by the three lines ℓ_i, ℓ_j, ℓ_k. He calls a triangle *original* if the circumcircle of that triangle passes through the origin. Since Kevin believes that the circles of

bovine life are tied directly to such triangles, he wants to know the number of original triangles formed by unordered triples of distinct lines.

Recall that the circumcircle of a triangle is the circle which passes through all the vertices of that triangle.

Of course, it is straightforward to solve this problem by first computing all triangles and then counting the number of circumcircles that pass through the origin. Unfortunately, this $O(n^3)$ time algorithm is too slow to solve problems with $n = 2,000$ lines with integer coefficients $|a_i|, |b_i|, |c_i| \leq 10,000$ representing line ℓ_i within the required time bound of four seconds on the Codeforces server which has a similar performance as a regular desktop PC. A MacBook needs already 2.5 seconds just to enumerate all triples of lines, adding the complex calculations to determine the circumcircle and testing whether it passes through the origin would increase this time by more than a factor of ten.

With these constraints on the problem size and running time it was clear that we need to find at least a quadratic algorithm (it takes only 3 ms on a MacBook to enumerate $\binom{2000}{2}$ pairs of lines). While many Codeforces problems have straightforward solutions based on elementary data structures like, for example, balanced search trees and priority queues, this problem gave me a hard time. None of the standard tricks from the bag of computational geometry algorithms like plane sweep, duality, divide-and-conquer, Voronoi diagram, etc., seemed to work to bring down the running time from cubic to quadratic complexity. There is no apparent locality in the problem, the lines forming original triangles can be close together or far apart, and no clever preprocessing or ordering of the lines seems to give an advantage that would allow us to not compute many of the $\binom{n}{3}$ triangles and circumcircles to obtain a better running time.

In this paper, we will show how to solve this problem in time $O(n^2 \log n)$ (i.e., nearly quadratic) by *not* computing any triangles or circles and only using the basic arithmetic operations $+, -, \star, /$, i.e., without computing roots and trigonometric functions which usually make life miserable for geometrical algorithm designers because of the inherent rounding errors. It is actually a common trick in combinatorics to count X instead of Y if that is what we are really interested in. A classical example from computational geometry is the problem of counting the number of cells in a simple arrangement of n lines in general position [4] by counting the number of vertices, instead. There are $1 + n + \binom{n}{2}$ cells because there are n infinite-downward rays separating the $n + 1$ cells of the arrangement that are unbounded in the downward direction; the remaining cells all have a unique lowest vertex, each vertex is the lowest one of a unique cell, and there are $\binom{n}{2}$ vertices.

This paper is organized as follows. In Section 2, we will introduce the notations and formulas from geometry we need to formulate and solve the problem. In Section 3, we will explain the algorithm and analyze its correctness and run-time. In Section 4, we will shortly discuss how the algorithm was implemented and fine-tuned to run faster by a factor of eight.
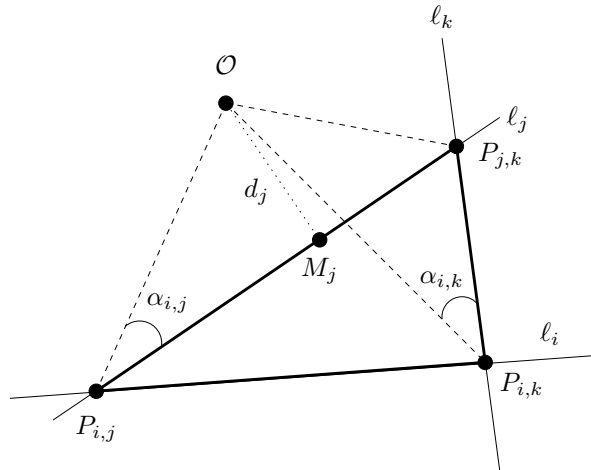
2 Geometry Preliminaries

In this section we define the problem and introduce some notations. The input is a set of n lines $\ell_1, \ldots, \ell_n$. Each line ℓ_i is represented by the equation

$$a_i x + b_i y = c_i , \tag{1}$$

where $a_i^2 + b_i^2 > 0$. We assume the lines are in general position, i.e., no two lines are parallel and no three lines pass through the same point. The first assumption implies

$$a_i b_j \neq a_j b_i \tag{2}$$

Figure 1 The triangle $T_{i,j,k}$ (in bold) formed by the lines ℓ_i, ℓ_k, ℓ_k.

for all $i \neq j$. Any three distinct lines ℓ_i, ℓ_j, ℓ_k, $1 \leq i < j < k \leq n$, therefore define a triangle $T_{i,j,k}$ with a unique circumcircle $C_{i,j,k}$. We call $T_{i,j,k}$ *original* if $C_{i,j,k}$ passes through the origin $\mathcal{O}$. The task in the *Original Triangle Counting Problem (OTC)* is to count the number of original triangles defined by the given set of n lines.

Before we solve this problem efficiently, let us introduce a few more notations and formulas from computational geometry. This also gives the reader the chance to digress and try to find an efficient solution herself before continuing to read this paper. The intersection point $P_{i,j} = (x_{i,j}, y_{i,j})$ of lines ℓ_i and ℓ_j, $1 \leq i < j \leq n$, can be computed as

$$x_{i,j} = \frac{b_i c_j - b_j c_i}{b_i a_j - b_j a_i}, \qquad y_{i,j} = \frac{a_i c_j - a_j c_i}{a_i b_j - a_j b_i} . \tag{3}$$

The point $M_j = (s_j, t_j)$ on line ℓ_j closest to the origin (see Figure 1) can be computed as

$$s_j = \frac{a_j c_j}{a_j^2 + b_j^2}, \qquad t_j = \frac{b_j c_j}{a_j^2 + b_j^2} \tag{4}$$

and the square of its distance d_j from the origin as

$$d_j^2 = \frac{c_j^2}{a_j^2 + b_j^2} . \tag{5}$$

A central element of our algorithm will be the angle under which we can see $\mathcal{O}$ from line ℓ_j in point $P_{i,j}$. We denote this angle by $\alpha_{i,j}$. We can compute the square of the sine of $\alpha_{i,j}$ as

$$\sin^2(\alpha_{i,j}) = \frac{d_j^2}{x_j^2 + y_j^2} . \tag{6}$$

We would like to use the right-hand side of Equation 6 to represent the angle $\alpha_{i,j}$ because we can compute it with basic arithmetic operations. However, since $\sin(\frac{\pi}{2} - \beta) = \sin(\frac{\pi}{2} + \beta)$ for any β, we cannot distinguish between angles smaller and larger than $\frac{\pi}{2}$. We will therefore use $\frac{d_j^2}{x_j^2 + y_j^2}$ to represent $\alpha_{i,j}$ if the three points $\mathcal{O}$, $P_{i,j}$, and M_j are counterclockwise oriented
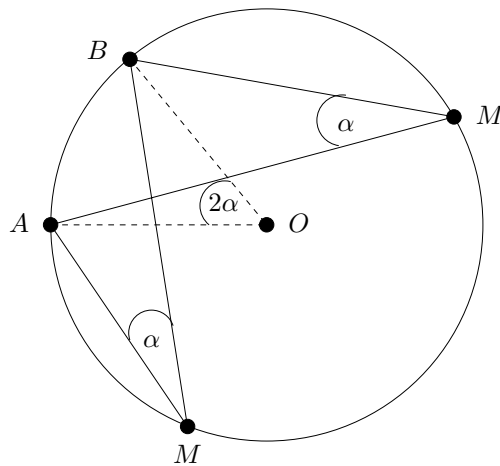
Figure 2 An illustration of the Inscribed Angle Theorem [3].

(i.e., when walking from $\mathcal{O}$ to M_j via $P_{i,j}$ we turn left at $P_{i,j}$, as in Figure 1), and we will use $-\frac{d_j^2}{x_j^2+y_j^2}$ if $\mathcal{O}$, $P_{i,j}$, and M_j are clockwise oriented.

The following theorem is a generalization of Thales's Theorem.

▶ **Theorem 1** (Inscribed Angle Theorem, [3]). *An angle α inscribed in a circle is half of the central angle 2α that subtends the same arc on the circle. Therefore, the angle does not change as its vertex is moved to different positions on the circle.*

We actually need to rephrase the theorem for our purposes, see Figure 2.

▶ **Theorem 2** ([3]). *Given two points A and B, the set of points M in the plane for which the angle AMB is equal to α is an arc of a circle through A and B. The measure of the angle AOB, where O is the center of the circle, is 2α.*

3 The Algorithm

We will now describe an efficient algorithm for solving OTC. It is based on Theorem 2 which we eventually remembered from another paper long ago [9]. It implies the following Corollary, see Figure 1. Note that the validity of the corollary does not depend on a particular way to draw the lines (i.e., the order in which the intersection points appear on the circle), see for example a different configuration in Figure 3. Please remember that we represent the angles $a_{i,j}$ as described after Equation 6.

▶ **Corollary 3.** *The circumcircle $C_{i,j,k}$ of triangle $T_{i,j,k}$ passes through the origin $\mathcal{O}$ if and only if $P_{i,j}$ and $P_{i,k}$ both see the line segment $\overline{\mathcal{O}P_{j,k}}$ under the same angle, i.e., if and only if $\alpha_{i,j} = \alpha_{i,k}$.*

Proof. We need to distinguish two cases, whether $P_{i,j}$, $P_{i,k}$, and $P_{j,k}$ appear in this order clockwise or counterclockwise on the circumcircle $C_{i,j,k}$ of triangle $T_{i,j,k}$. In the former case, $P_{i,j}$ and $P_{i,k}$ lie in different half-spaces defined by the line through $\mathcal{O}$ and $P_{j,k}$ (see Figure 3), while in the latter case they are in the same half-space (see Figure 1). Below we only give the proof of the Corollary in the latter case, from Figure 3 it should be clear that the proof of the other case is similar.
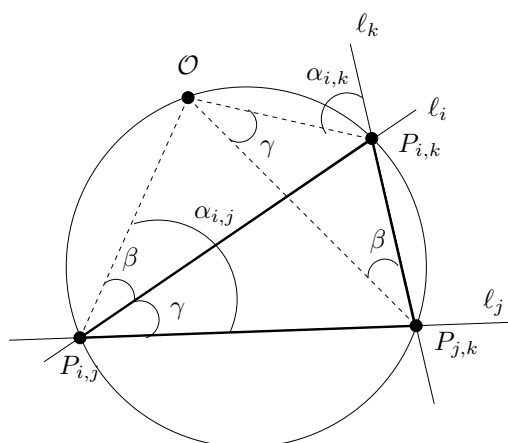
Figure 3 A different configuration for triangle $T_{i,j,k}$ (in bold) formed by the lines ℓ_i, ℓ_k, ℓ_k. Note that $\alpha_{i,j} = \beta + \gamma = \alpha_{i,k}$.

We use the Inscribed Angle Theorem with $A = \mathcal{O}$, $B = P_{j,k}$, and $P_{i,j}$ and $P_{i,k}$ in the role of M. If $\alpha_{i,j} = \alpha_{i,k}$, then $P_{i,j}$ and $P_{i,k}$ see the line segment $\overline{\mathcal{O}P_{j,k}}$ under the same angle. Now Theorem 2 implies that the four points $\mathcal{O}$, $P_{j,k}$, $P_{i,k}$, and $P_{i,j}$ must lie on a circle, which is the circumcircle $C_{i,j,k}$ of triangle $T_{i,j,k}$. That means, $C_{i,j,k}$ contains the origin. ◀

We can now formulate an efficient algorithm for OTC. First note that we can w.l.o.g. assume that no line goes through the origin. If there is only one such line, it cannot contribute to any original triangle and we can delete it. If there are two such lines, any triangle they form with a third line is an original triangle, so we can increase the triangle count by $n - 2$ and delete the two lines through the origin.

1. Compute all intersection points and store point $P_{i,j}$, $1 \leq i < j \leq n$, on line ℓ_i together with the angle $\alpha_{i,j}$ (using Equation 6).
2. Sort for each line ℓ_i, $1 \leq i \leq n$, the intersections points $P_{i,j}$, $j \neq i$, on the line by angle $\alpha_{i,j}$;
3. If the same angle appears k times on line ℓ_i for some $k \geq 2$, then any pair of the corresponding intersection points induce an original triangle, i.e., we can increase the triangle counter by $\binom{k}{2}$.

Although we do not use trigonometric functions, we may experience rounding error problem when computing the squares of the sine values of the angles. However, if the lines have integer coefficients of absolute value at most 10^5, then we can define two angles to be equal if they are less than 10^{-14} apart, for example. As one reviewer pointed out, we may also avoid the rounding problems by using exact rationals instead of error-prone doubles or floats.

The run-time of step 1 is $O(n^2)$, step 2 needs time $O(n^2 \log n)$, and step 3 needs time $O(n^2)$. Thus, the run-time of the algorithm is $O(n^2 \log n)$.

▶ **Theorem 4.** *We can solve OCT in time $O(n^2 \log n)$.*

4 Engineering Speed-Ups

At the moment, there are 127 correct solutions for OTC listed on Codeforces, the fastest one running in 109 ms [11] on the most difficult test case (apparently using a similar algorithm

as the one described in this paper, but submitted three weeks later than our solution). Our first correct submission [6] needed 1,684 ms, well within the time bounds of 4 seconds, but nevertheless a bit disappointing considering the brilliant elegance and simplicity of our algorithm.

So we set out to identify and remove the bottlenecks in our program code. Our first bad design decision had been to actually compute all original triangles three times, namely once for each line bounding the triangle. Computing the original triangles only once reduced the run-time by a factor of two to 826 ms [7]. Note that we do not achieve a speed-up factor of three. The reason is that originally each $P_{i,j}$ was stored twice, once on line ℓ_i and once on line ℓ_j; in the new implementation it was only stored on line ℓ_i if $i < j$, thus saving half of the work.

The second bad design decision had been to sort all intersection point angles in one single batch which may look slightly more elegant on paper but is less efficient in practice. So the next speed-up came from sorting the angles separately for each line. Asymptotically there is no difference in the run-time, it is $O(n^2 \log n)$ in both cases, but the constant factors differ by a factor of four. In the first variant, we sort $\binom{n}{2}$ angles which needs approximately $\binom{n}{2} \log \binom{n}{2} \approx n^2 \log n$ comparisons. In the improved variant, we successively sort $n-1, n-2, \ldots, 1$ values which requires $(n-1)\log(n-1) + (n-2)\log(n-2) + \cdots + 1\log 1 < \frac{1}{2}n^2 \log n$ comparisons. We gain another factor of two because we compare pairs of (line,angle) in the first variant, i.e., each comparison in the sorting step actually requires two comparisons, while the sorting in the second variant only requires one comparison to compare two angles. Consequently, the run-time dropped to 187 ms [8], which is currently the 11th best run-time on the Codeforces server. Actually not too bad for a lousy programmer.

Acknowledgements.   We would like to thank the reviewers for their helpful comments, and also for generously ignoring some shortcomings of the original draft that was written in a great hurry shortly before the submission deadline.

References

1 ACM-ICPC International Collegiate Programming Contest. `https://icpc.baylor.edu`.

2 Problem 603D–36: Ruminations on Ruminants. *Codeforces*, Contest 342, Division 1, 2015, Dec 1. `http://codeforces.com/contest/603/problem/D`.

3 Wikipedia contributors. Inscribed angle, Date retrieved: 20 February 2016 15:40 UTC. Permanent link: `https://en.wikipedia.org/w/index.php?title=Inscribed_angle&oldid=699778165`.

4 Wikipedia contributors. Arrangement of lines, Date retrieved: 29 February 2016 15:24 UTC. Permanent link: `https://en.wikipedia.org/w/index.php?title=Arrangement_of_lines&oldid=702141041`.

5 R. Fleischer. FUN with implementing algorithms. In E. Lodi, L. Pagli, and N. Santoro, editors, *Proceedings of the 1998 International Conference FUN with Algorithms (FUN'98)*, pages 88–98. Carleton Scientific, Proceedings in Informatics 4, 1999.

6 R. Fleischer. Problem 603D–36: Submission 14741117. *Codeforces*, Contest 342, Division 1, 2015, Dec 10. `http://codeforces.com/contest/603/submission/14741117`.

7 R. Fleischer. Problem 603D–36: Submission 16228941. *Codeforces*, Contest 342, Division 1, 2016, Jan 20. `http://codeforces.com/contest/603/submission/16228941`.

8 R. Fleischer. Problem 603D–36: Submission 16231449. *Codeforces*, Contest 342, Division 1, 2016, Jan 20. `http://codeforces.com/contest/603/submission/16231449`.

9 R. Fleischer and Y. Wang. On the camera placement problem. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC'09).* Springer Lecture Notes in Computer Science 5878, pages 255–264, 2009.

10 M. Mirzayanov. Codeforces: The only programming contests web 2.0 platform, 2010–2016. `http://codeforces.com`.

11 Z. Shi. Problem 603D–36: Submission 15094164. *Codeforces*, Contest 342, Division 1, 2015, Dec 30. `http://codeforces.com/contest/603/submission/15094164`.