A foolish consistency is the hobgoblin of little minds

— Ralph Waldo Emerson

Lecture 8
DEGENERACY AND PERTURBATION

The central concern of theoretical algorithms is to understand the inherent complexity of computational problems. As such, most algorithms on paper are formulated to facilitate their analysis, not meant to be literally implemented. In computational geometry, the reader will often encounter qualifications of the following sort:

> *"Our algorithm accepts an input set S of points. For simplicity, we assume that the points in S are distinct, no 2 points are co-vertical, no 3 points are collinear, no 4 points are co-circular, etc. The reader can easily extend our algorithm to handle inputs that fail these assumptions."*

The inputs that satisfy these assumptions may be said to be **non-degenerate**; otherwise they are **degenerate**. Implementors are left to their own devices to handle degenerate cases. Some papers may not even list the degenerate inputs. Thus, Forrest [2] deplores the "plethora of special cases ... glossed over by theoretical algorithms". Forrest, and also Sedgewick [6], calls this the "bugbear" of geometric algorithm implementation.

In this lecture, we explore some general techniques that can help bridge *this* particular gap between theoretical algorithms and their implementation. Notice we do not address other possible gaps, such as the under-specification of data representation.

§1. A Perturbation Framework

Given an algorithm, let us assume that there is a notion of **valid inputs**. Generally speaking, we expect it to be easy to verify if a given inputs are valid. Among the valid inputs, certain inputs are defined as **degenerate**. Let us call an algorithm **generic** if it works correctly only for non-degenerate inputs; it is **general** if it works correctly for all valid inputs, degenerate or not.

As a simple example, suppose the valid inputs are any sequence of points. If there are n points, $p_1, \ldots, p_n$, we view the input as a sequence

$$\mathbf{a} = (x_1, y_1, x_2, y_2, \ldots, x_n, y_n) \in \mathbb{R}^{2n}$$

where $p_i = (x_i, y_i)$. Thus the set of valid inputs can be identified with $\mathbb{R}^{2n}$ (for each n). But in many algorithms, we assume the input points are distinct. In this case, the valid inputs of size n would be

$$\mathbb{R}^{2n} \setminus \{\mathbf{a} : (\exists i, j)[i \neq j, x_i = x_j, y_i = y_j\}.$$

Checking valid inputs in this case is relatively easy, though still non-trivial. Alternatively, the algorithm may detect this error and reject during computation.

The gap between theoretical algorithms and implementable algorithms amounts to converting generic algorithms into general algorithms. Of course, one way to bridge this gap is to explicitly detect all the degenerate cases and handle them explicitly. There is a practical problem with this suggestion:

> (B) The number of degenerate cases can be overwhelming and non-trivial to exhaustively enumerate.

Roughly speaking, the number of special cases for handling degeneracies increases exponentially with the dimension d of the underlying geometry (e.g., $S \in \mathbb{R}^d$). So the explicit treatment of degeneracies can increase the size of the code by a large factor: we end up with a program where, say 10% of the code uses 90% of the machine cycles, with a large portion of the remaining code rarely executed if at all. This makes the automatic generation of this 90% of the code an attractive goal, if it were possible.

Some feel that the degenerate cases are rare. This is true in some measure-theoretic sense, but ignoring these cases is not an option if we want a robust implementation. Another impulse is to require the algorithm designer to supply details for the degenerate cases as well. Indeed, this has been suggested as a standard for publications in algorithms. This is[1] neither desirable nor enforceable. Such details may not make the most instructive reading for non-implementors. In short, the gap between theoretical algorithms and practical algorithms will not go away: the literature will continue produce some algorithms spelled out in detail, and others less so. What should implementors do?

In many applications situations, the user would not mind if their input is slightly perturbed (this is clearly acceptable if the inputs are inexact to begin with). What does perturbation mean? Recall geometric objects live in a parametrized space, and the input is just the set of parameters (associated with some combinatorial structure). If G is the combinatorial structure (e.g., a graph) and $x = (x_1, \ldots, x_n)$ are the parameters, then the object is $G(x)$. In general, only certain choices of parameters x are **consistent** with G. For any $\varepsilon > 0$, we define an ε-perturbation of $G(x)$ to be another object of the form $G(x')$ where $\|x - x'\| < \varepsilon$. Note that $G(x')$ shares the same combinatorial structure (G) with $G(x)$, and and x' must be consistent with G. In numerical analysis, this ε is called **backwards error** bound.

So we want to exploit this freedom to do perturbation. In particular, we would like to consider automatic tools to choose some $G(x')$ that is non-degenerate. The framework is as follows:

Begin with a generic algorithm A for a problem P. We provide schemes for transforming A into a general algorithm A' for P. But what is A'? It cannot really be a full-fledged algorithm for P since, by definition, it does not really handle degenerate cases. Yet as a general algorithm it does produce an output, even when the input is degenerate. We call this output a "perturbed output". Perturbation calls for another **postprocessing algorithm** B. The algorithm B is supposed to fixed the perturbed output so that it is the correct output for the unperturbed input. This framework is illustrated in figure 1.

The simplistic view of perturbation is to add a small non-zero value to each numerical parameter (within some $\epsilon > 0$ bound). We call this **real perturbation**. But we can also use **infinitesimal perturbation** which can be viewed as perturbation that is smaller than any real perturbation $\epsilon > 0$. This notion can be justified using the theory of infinitesimal numbers. We will consider both these kinds of perturbations.

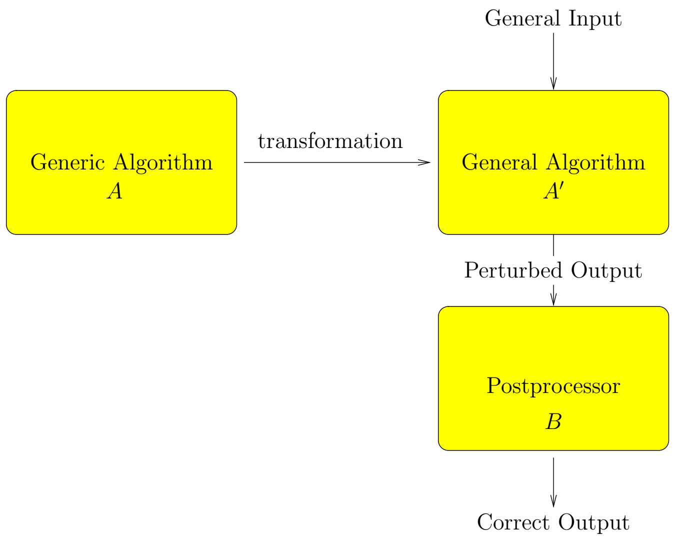

Figure 1: Generic to General Transformation

[1]The algorithmic activity, like all human activity, is multifaceted and serves no single overriding concern. In theoretical papers, implementation details are usually of secondary interest.

¶1. Example. Let us illustrate infinitesimal perturbation with the problem of computing the Voronoi diagram of a point set S. The input S is degenerate if any four points of S are cocircular. If A is a generic algorithm, what does the transformed A' compute? Well, in case a, b, c, d are cocircular (in this order) about an empty circle C of S, then the output of A' will include two Voronoi vertices, say $v(a, b, c)$ and $v(c, d, a)$, which are connected by a Voronoi edge of zero length. Then the postprocessor B has to detect such zero-length edges and remove them.

¶2. Discussion. It has been pointed out that handling degeneracies explicitly may be a good thing (e.g. [1]). This is based on two facts (i) programming the degenerate cases may not be onerous, and (ii) perturbation can increase the run-time of specific instances by an exponential amount. A simple example is the convex hull of a set of n points in d dimensions. If all n points are coincident, the runtime is constant, but the convex hull of n points in general position is $\Theta(n^{\lfloor d/2 \rfloor})$.

Such examples supply some caution when using the generic perturbation framework. On the other hand, examples of complete handling of degeneracies in algorithms are not common and there has been no evidence that they scale easily to harder problems. As for the potential of increasing the worst case cost dramatically, it is clear that such examples (all points coincident) are special. It is possible that general techniques may be developed to automatically reduce such complexity.

The examples do not negate the fact that generic algorithms are much easier to design and code than general algorithms. So when perturbations are acceptable, and there are general tools to cross the chasm from generic to general, users ought to consider this option seriously. A more potentially more serious objection arises: by not constructing a general algorithm in the first place, aren't we doing more work in the end? Instead of one general algorithm, we need to construct a generic A, transform it to A' and finally, provide a postprocessor B. We see in some examples that this is not necessarily harder. First of all, the transformation $A \mapsto A'$ may be quite simple. Basically the idea is to perturbed any input so that it looks nondegenerate. Second, in view of the possible code bloat in a general algorithm, the work involved in constructing A' is negligible. An example of this is the problem of constructing hyperplane arrangements (see below). Finally, the postprocessing algorithm B in these examples turns out to be relatively simple. More importantly, B is not specific to A', but can be used with *any* other algorithm for P. For instance, if P is computing Voronoi diagrams of a point set, then B amounts to detecting Voronoi edges of length 0 and collapsing their end points. Thus B is not specific to A or A'.

¶3. An Example. It is instructive to work out the degenerate cases of some of the algorithms we have encountered so far in these lectures. These are not too difficult, but it can also serve to indicate the kind of tedium that our propose framework can eliminate.

Let us begin with the Bentley-Ottmann algorithm.

Exercises

Exercise 1.1: Take any Voronoi diagram algorithm for a point set (e.g., Fortune's sweep).
(a) Describe the generic version of this algorithm.
(b) Discuss how the possibility of zero-length edges would affect how the arithmetic is implemented. One basically has to beware of dividing by zero. ♢

Exercise 1.2: The exercise will help you appreciate the practical difficulties of enumerating all combinatorial cases, including degenerate cases. In each part (a)-(e) of this question, we consider the combinatorial intersection patterns involving two simple geometric objects, s and t in $\mathbb{R}^d$ ($d = 2$ or 3). A segment is represented by a pair of distinct points in $\mathbb{R}^d$, and a triangle is represented by three non-collinear points in $\mathbb{R}^d$. In each case, you need to do three things:
(1) Classify all non-degenerate (s, t)-intersection patterns.
(2) Classify all degenerate (s, t)-intersection patterns. Do not forget that, for instance, if s, t are both triangles, they might be identical. Summarize your classifications in (1) and (2) by filling in the Table 1.
(3) Give a classification algorithm which, given s and t, will correctly classify the intersection patterns. This classification algorithm must detect non-intersection as well, and must be "parsimonious". A parsimonious (or non-redundant algorithm) is one that does not any comparisons that could have been deduced from earlier comparisons. ♢

	Dim	Type of s	Type of t	No. Nondegen.	No. Degenerate
(a)	2	segment	segment		
(b)	2	segment	triangle		
(c)	2	triangle	triangle		
(d)	3	segment	triangle		
(e)	3	triangle	triangle		

Table 1: Intersection patterns of simple objects

Exercise 1.3: Classify the combinatorial intersection pattern for three intersecting triangles in space. As in the previous exercise, list (1) the non-degenerate cases (2) the degenerate cases, and (3) give a parsimonious classification algorithm. $\diamondsuit$

Exercise 1.4: Work out the degenerate cases for for the problem of constructing a hyperplane arrangement. $\diamondsuit$

Exercise 1.5: Work out the degenerate cases for the beneath-beyond algorithm for 3-D convex hull. $\diamondsuit$

Exercise 1.6: Fix any given combinatorial structure G, and suppose $\mathbb{R}^n$ is the set of valid parameters for G, and there is a polynomial $p(x_1, \ldots, x_n)$ such that for all $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, if x is degenerate for G then $p(x) = 0$. Prove that for any $\varepsilon > 0$ and any $x \in \mathbb{R}^n$, we can always $x' \in \mathbb{R}$ such that $\|x - x'\| < \varepsilon$ which is non-degenerate. $\diamondsuit$

End Exercises

§2. Model of Degeneracy

Our definition of degeneracies in Section 1 ought to called **algorithm-induced degeneracies**. In contrast, an input is **inherently degenerate** if it is degenerate for all algorithms for the problem. An example is an input set S of points for the convex hull problem. If three of the points in S are collinear and they lie on the boundary of the convex hull of S, we would say that S is **inherently degenerate** because any convex hull algorithm must be able to detect and handle this degeneracy. However, if S has three collinear points but all such triples do not on the convex hull of S, then S is not inherently degenerate. S may still be degenerate for some algorithms (e.g., an incremental algorithm might see such a collinear triple). We focus on algorithm-induced degeneracy, as this is more important in practice.

In the introduction, we described some typical non-degeneracy assumptions on input points (coincident points, covertical pairs, collinear triples, cocircular quadruples, etc). Here are more examples of such assumptions, in case the input is a set of lines:

- a vertical line
- 2 parallel lines
- 2 perpendicular lines
- 3 concurrent lines
- 2 intersecting lines (when the lines are embedded in dimension greater that 2)
- the intersection points defined by pairs of lines satisfy the degeneracy conditions specified for point sets.

In case the input set is a set S of points and a set T of lines, we may introduce more such assumptions: (1) an input point lying on an input line, (2) an input line parallel to the line through 2 input points, (3) an input line perpendicular to the line through 2 input points, etc.

¶4. Computational Model. Our algorithm is essentially a Real RAM whose instructions are classified into one of two types: **comparison steps** or **construction steps**. A typical comparison step has the form

$$\text{if } (P(y_1, \ldots, y_m))\text{then goto } L \tag{1}$$

where $P(y_1, \ldots, y_m)$ is a predicate of the variables $y_1, \ldots, y_m$, and L a label of an instruction. If P is true, the the next instruction has label L, otherwise it will be the instruction following the current one in the program. But we also consider a sign comparison step of the form

$$OnSign(g(y_1, \ldots, y_m)) \text{ goto } (g_-, L_0, L_+) \tag{2}$$

where g is a real function applied to variables $y_1, \ldots, y_m$. Depending on the sign of $g(y_1, \ldots, y_m)$, the next instruction will be labeled by one of L_-, L_0, or $L+$. A construction step has the form

$$y \leftarrow g(y_1, \ldots, y_m) \tag{3}$$

where y_i are variables and g is some function. Construction steps do not cause any branching.

Note that assume the variables of our RAM are of various types that include reals, integers, Booleans, and possibly other types. We allow algorithm-dependent predicates (e.g., P in (1)) and functions (e.g., g in (3)) to be used directly in our RAM programs.

Assume that the input is a geometric object of the form $G(x)$ where $x \in \mathbb{R}^n$ and G is a directed graph. Our algorithm will have programming variables, and these are of two kinds: real variables that depend on x are **critical**, and all others are non-critical. How do we determine if a real variable y is critical? Each of the input variables $x_1, \ldots, x_n$ are critical. If y ever appears as the right hand side of a construction step as in (3) where any of the variables $y_1, \ldots, y_m$ is critical, then y is critical. A comparison step of the form (2) where any $y_1, \ldots, y_m$ is critical is called a **critical comparison**. The input is deemed **degenerate** if the algorithm takes any zero-branch in any critical comparison. The g used in a critical comparison of the form (2) is called a **critical function**.

¶5. Precise Problems for Imprecise Points. It should be emphasized that inputs are assumed to be exact whenever we discuss perturbation methods. In any case, what do we mean that an input is "imprecise"? What is the algorithm supposed to compute?

Suppose we want to compute the convex hull of a set of "imprecise points". The input is a set S of points together with an $\varepsilon > 0$ parameter. What we want to output is the **precise** convex hull of any set S' of points such that each point in S' is within ε distance from a corresponding point in S. We see that this new problem is as precise as anything we have considered so far. So in some sense, there is really no such thing as an "imprecise input". In any case, we can solve the (precise) problem of computing the convex hull of "imprecise points" just by computing the convex hull S. Of course, we now have the option of not doing so (Exercise).

Note that all such degeneracy conditions can be specified as a polynomial $p(x_1, \ldots, x_m)$ vanishing on some of the input parameters. We may call these **á priori degeneracy conditions**. Usually, these á priori conditions are meant to prevent the occurrence of algorithm induced degeneracies (i.e., **á posteriori degeneracies**). Informally, á priori and á posteriori degeneracies is analogous to compile-time and run-time errors.

Exercises

Exercise 2.1: Provide polynomials whose vanishing corresponds to each to the degeneracy condition listed above. ♢

Exercise 2.2: Discuss how we can solve the (precise) problem of computing the convex hull of imprecise points by taking advantage of the ε parameter. Is it true that a generic convex hull algorithm can always solve such a problem? ♢

End Exercises

§3. Black Box Perturbation Scheme

A critical value is the value of a critical variable. The **depth** of a critical value y is inductively defined: the value of an input parameter $x_1, \ldots, x_n$ has depth 0. The depth of a value y is 1 more than the maximum depth of $y_1, \ldots, y_m$ where $y = g(y_1, \ldots, y_m)$. Note that we use the same symbol 'y' to denote a programming variable as well as its value which is time-dependent – the context should make it clear which is intended. If the values $y_1, \ldots, y_m$ in a critical comparison (3) is never more than d, we say the algorithm has **depth** of d.

We will first describe a scheme described in [8] which works under the following conditions:

- The critical functions are polynomial.
- The test depth is 0.

Later we note possibilities to relax some of these conditions. The idea is to construct a "black box" which, given any non-zero polynomial $p = p(x_1, \ldots, x_m)$ and any sequence of depth 0 parameters $\mathbf{b} = (b_1, \ldots, b_m)$, to output a non-zero sign

$$\mathtt{sign}_{\mathbf{b}}(p) \in \{-1, +1\}.$$

Note that a depth 0 sequence $\mathbf{b}$ is essentially a subsequence of $\mathbf{a}$ in (**??**). Moreover, if $p(\mathbf{b}) = p(b_1, \ldots, b_m)$ is non-zero, then $\mathtt{sign}_{\mathbf{b}}(p)$ is just the sign of $p(\mathbf{b})$. Thus, the interesting case happens when $p(\mathbf{b}) = 0$: in effect, we are computing a "signed zero", $+0$ or -0.

¶6. Using the Black Box. As in figure 1, we begin with a generic algorithm A. We construct the general algorithm A' by replacing each polynomial-test $p(\mathbf{b})$ in A with a call to this black box. The algorithm then continues with the positive or negative branch after the test, depending on $\mathtt{sign}_{\mathbf{b}}(p)$. Even if the input is degenerate, no zero-branch is ever taken.

Before describing our solution, consider the following possible approach. Randomly perturb the inputs $\mathbf{b}$ to some $\mathbf{b}'$ (with $\|\mathbf{b} - \mathbf{b}'\| < \varepsilon$) and evaluate the sign of $p(\mathbf{x}')$. With high probability, $p(\mathbf{b}')$ is non-zero, as desired. But what if the result is zero? Perturb again. Unfortunately, we need to make sure that the new perturbation preserves the signs of all previously evaluated polynomials. This seems hard. Moreover, we must ensure that the perturbation $\mathbf{b}'$ is sufficiently small that it does not change the sign of any non-zero $p(\mathbf{b})$. We abandon this approach for now, although the idea of numerical perturbation can be made to work under special conditions (see Section 4 below).

¶7. Admissible Black Box Schemes. Our scheme is based on the concept of "admissible orderings" which originally arose in Gröbner bases theory.

Let $\mathbf{x} = (x_1, \ldots, x_n)$ be real variables (corresponding to the input parameters (**??**)). Let $\mathrm{PP} = \mathrm{PP}(x_1, \ldots, x_n)$ denote the set of all power products over $x_1, \ldots, x_n$. Thus a typical power product has the form

$$w = \prod_{i=1}^{n} x_i^{e_i}, \qquad e_i \geq 0$$

where the e_i are natural numbers. Note that $1 \in \mathrm{PP}$ corresponding to $e_1 = e_2 = \cdots = e_n = 0$. If $e = (e_1, \ldots, e_n)$ we also write $w = x^e$. The **degree** of x^e is $\sum_{i=1}^{n} e_i$, also denoted $|e|$. We say a total ordering $\leq_A$ on PP is **admissible** if for all $u, v, w \in \mathrm{PP}$, (1) $1 \leq_A w$, and (2) $u \leq_A v$ implies $uw \leq_A vw$.

There are infinitely many admissible orderings, but the two most familiar ones are **(pure) lexicographical ordering** $\leq_{\mathtt{LEX}}$ and **total degree ordering** $\leq_{\mathtt{TOT}}$. These are defined as follows: let $w = x^e$ and $v = x^d$ where $e = (e_1, \ldots, d_n)$ and $d = (d_1, \ldots, d_n)$.

- $w \leq_{\mathtt{LEX}} v$ iff either $e = d$ or else $e_i < d_i$ where i is the first index where $e_i \neq d_i$.
- $w \leq_{\mathtt{TOT}} v$ iff $|e| < |d|$ or else $w \leq_{\mathtt{LEX}} v$.

Notice that both of these orderings depend on a choice of an ordering of the variables $x_1, \ldots, x_n$. So there are really $n!$ lexicographical (resp. total degree) orderings.

Now let $p(\mathbf{x}) \subseteq \mathbb{R}[\mathbf{x}]$ be a real polynomial. For $w \in \mathrm{PP}$, we let $\partial_w(p)$ denote the partial derivative of $p(\mathbf{x})$ with respect to w. More precisely, if $w = x^e$ and $e = (e_1, \ldots, e_n)$ then

$$\partial_w(p) := \frac{\partial^{|e|} p}{\partial^{e_1} x_1 \partial^{e_2} x_2 \cdots \partial^{e_n} x_n}.$$

For instance, $w = x^3yz^3$ (we typically let $x = x_1, y = x_2, z = x_3$) then $\partial_w(p) := \frac{\partial^6 p}{(\partial x)^3(\partial y)(\partial z)^2}$. If $p(x,y,z) = 2x^2z^3 - 7xy + y^5 + 10$ then $\partial_x(p) = 4xz^3 - 7y$, $\partial_{xy}(p) = 7$, $\partial_{z^2}(p) = 24xz$, $\partial_{xyz}(p) = \partial_{yz^2}(p) = 0$.

Fix any admissible ordering $\leq_A$. We describe a blackbox sign function corresponding to $\leq_A$. The ordering $\leq_A$ gives an enumeration

$$(w_0, w_1, w_2, \ldots), \qquad w_i \leq_A w_{i+1}$$

of all power products in PP. Clearly, $w_0 = 1$. Define the sequence of polynomials

$$\partial(p) := (\partial_{w_0}(p), \partial_{w_1}(p), \cdots).$$

If $\mathbf{a} = (a_1, \ldots, a_n)$ then define

$$\partial(p; \mathbf{a}) := (\partial_{w_0}(p)(\mathbf{a}), \partial_{w_1}(p)(\mathbf{a}), \cdots).$$

So $\partial(p; \mathbf{a})$ is an infinite sequence of numbers obtained by evaluating each polynomial of $\partial(p)$ at $\mathbf{a}$. We finally define our "black box sign" $\mathtt{sign}_{\mathbf{a}}(p)$ to be the sign of the first non-zero value in $\partial(p; \mathbf{a})$. Note that if $p \neq 0$ then $\partial_{w_i}(p)$ is a non-zero constant for some a. Hence $\mathtt{sign}_{\mathbf{a}}(p)$ is well-defined for all non-zero p. By definition, $\mathtt{sign}_{\mathbf{a}}(p) = 0$ when p is the zero polynomial. Also write

$$\mathtt{sign}_{\leq_A, \mathbf{a}}(p) \tag{4}$$

for $\mathtt{sign}_{\mathbf{a}}(p)$ to indicate the dependence on $\leq_A$. We will call $\mathtt{sign}_{\leq_A, \mathbf{a}}$ an **admissible sign function**.

Let us first give two actual examples of perturbation that can be understood as special cases of our blackbox scheme.

¶8. Example: SoS Scheme of Edelsbrunner and Mücke. Suppose H is a set of n lines in the plane. Each $h_i \in H$ $(i = 1, \ldots, n)$ has the equation $y = a_i x + b_i$. Thus the input parameters are $(a_1, b_1, a_2, b_2, \ldots, b_n)$. The problem is to compute the arrangement of cells defined by these lines: each cells has dimensions 0, 1 or 2 where 0-cells are just points of intersection defined by the lines, 1-cells are open line segments bounded by 0-cells, and 2-cells are the open connected polygonal regions bounded by the lines. The algorithm for computing an arrangement is fairly easy in the generic case. But dealing with degeneracies is complicated, and it gets much worse in higher dimensions. Hence there is strong motivation to use a generic algorithm, and to perturb the input to ensure non-degeneracy. Suppose 3 distinct lines h_i, h_j, h_k are concurrent (passes through a common point). This is a degeneracy. It amounts to the vanishing of the following determinant:

$$\delta_{ijk} = \det \begin{bmatrix} a_i & b_i & 1 \\ a_j & b_j & 1 \\ a_k & b_k & 1 \end{bmatrix}. \tag{5}$$

We say H is **simple** if no 3 distinct lines are concurrent. We want to perturb the input parameters so that H is simple. For this purpose, let $\varepsilon > 0$ be a small value (to be clarified) and define $h_i(\varepsilon)$ with equation

$$\begin{aligned} y &= a_i(\varepsilon)x + b_i(\varepsilon), \\ a_i(\varepsilon) &= a_i + \varepsilon^{2^{2i}}, \\ b_i(\varepsilon) &= b_i + \varepsilon^{2^{2i-1}}. \end{aligned}$$

The corresponding determinant, expanded as a polynomial in ε is given by

$$\begin{aligned}
\delta_{ijk}(\varepsilon) &= \det\begin{bmatrix} a_i(\varepsilon) & b_i(\varepsilon) & 1 \\ a_j(\varepsilon) & b_j(\varepsilon) & 1 \\ a_k(\varepsilon) & b_k(\varepsilon) & 1 \end{bmatrix} \\
&= \det\begin{bmatrix} a_i & b_i & 1 \\ a_j & b_j & 1 \\ a_k & b_k & 1 \end{bmatrix} \\
&\quad -\varepsilon^{2^{2i-1}} \det\begin{bmatrix} a_j & 1 \\ a_k & 1 \end{bmatrix} \\
&\quad +\varepsilon^{2^{2i}} \det\begin{bmatrix} b_j & 1 \\ b_k & 1 \end{bmatrix} \\
&\quad +\varepsilon^{2^{2j-1}} \det\begin{bmatrix} a_i & 1 \\ a_k & 1 \end{bmatrix} \\
&\quad +\varepsilon^{2^{2j-1}+2^{2k-1}} \\
&\quad +\cdots .
\end{aligned}$$

The key observation is that each coefficient of this polynomial is a subdeterminant of Δ. Moreover, if ε is sufficiently small, and $i < j < k$, then the sign of $\delta_{ijk}(\varepsilon)$ is obtained by evaluating these coefficients in the order shown above (in order of increasing power of ε). The first term is therefore Δ. We stop at the first non-zero coefficient, as this is the sign of the overall polynomial.

This ε-scheme is called **Simulation of Simplicity** (SoS) and clearly works in any dimension. We leave it as an exercise to show that SoS amounts to an admissible sign function, based on a suitable lexicographic ordering $\le_{\mathtt{LEX}}$.

¶9. Example: Euclidean Maximum Spanning Tree. The second example arises in the computation of Euclidean maximum spanning tree [5]. Given n points $p_1, \ldots, p_n$ where $p_i = (x_i, y_i)$, we want to totally sort the set

$$\{D_{ij} : i, j = 1, \ldots, n, (i \neq j)\}$$

where $D_{ij} = \|p_i - p_j\|^2 = (x_i - x_j)^2 + (y_i - y_j)^2$. When $D_{ij} = D_{k\ell}$, Monma et al [5] break ties using the following rule: let $i_1 = \min\{i, j, k, \ell\}$ and relabel $\{i, j, k, \ell\} \setminus \{i_1\}$ as $\{j_1, i_2, j_2\}$ such that

$$\{\{i_1, j_1\}, \{i_2, j_2\}\} = \{\{i, j\}, \{k, \ell\}\}.$$

There are two cases:
(A) If $i_1 \notin \{i, j\} \cap \{k, \ell\}$ then make $D_{i_1,j_1} > D_{i_2,j_2}$ iff $x_{i_1} \ge x_{j_1}$.
(B) If $\{i_1\} = \{i, j\} \cap \{k, \ell\}$ then, without loss of generality, assume $i_1 = i_2$. Make $D_{i_1,j_1} > D_{i_2,j_2}$ iff $2x_{i_1} \ge x_{j_1} + x_{j_2}$.

Let us define the function σ which assigns to polynomials of the form $p = D_{ij} - D_{k\ell}$ a sign $\sigma(p) \in \{-1, +1\}$. The above scheme amounts to defining $\sigma(D_{ij} - D_{k\ell}) = +1$ iff $D_{ij} > D_{k\ell}$ (breaking ties as needed). An exercise below shows that σ arises from an admissible sign function.

¶10. Issues In the next sections, we address three key issues:

- The $\mathtt{sign}(p(\mathbf{x}), \mathbf{b})$ must satisfy some notion of consistency. What is it?
- How should we interprete the output from the generic algorithm when the input is degenerate?
- How can we implement such a black box?

The answer to the first question is that, for any finite set of polynomials $A \subseteq \mathbb{R}[\mathbf{x}]$, and for any $\mathbf{b} \in \mathbb{R}^n$ and $\varepsilon > 0$, there exists $\mathbf{b}' \in \mathbb{R}^n$ such that $\|\mathbf{b} - \mathbf{b}'\| < \varepsilon$ such that $\mathtt{sign}(p(\mathbf{x}), \mathbf{b}) = \mathtt{sign}(p(\mathbf{b}'))$. This proves that the sign function $\mathtt{sign}(p(\mathbf{x}), \mathbf{b})$ is "consistent".

Exercises

Exercise 3.1: Show that the sign given to $\Delta_{ijk}(\varepsilon)$ is exactly the sign given by a suitable admissible sign function. In fact, the admissible ordering is $\leq_{\texttt{LEX}}$ for a suitable ordering of the input parameters. $\diamondsuit$

Exercise 3.2: Generalize the SoS scheme to the computation of the sign of any $n \times n$ determinant. Show that this generalization of SoS is again case of our black-box sign, under a suitable $\leq_{\texttt{LEX}}$ ordering. $\diamondsuit$

Exercise 3.3: In the above analysis, we assumed that lines has the form $y = ax + b$. Suppose we now assume lines have equations $ax + by + c = 0$, with $a^2 + b^2 > 0$. What has to be modified in the SoS scheme? $\diamondsuit$

End Exercises

§4. Implementation Issues

We consider the implementation of Black Boxes. Such a scheme would be of general utility. For instance, it could have been used in place of SoS for perturbing sign of determinant, and in the Euclidean Maximum Spanning Tree application. An example from Paterson shows not all perturbation schemes arise from black boxes (Exercise).

One way to implement our black box is to choose a particular admissible ordering, and implement the corresponding black box. For instance, the SoS Scheme has been implemented by Mücke. If we choose the lexicographic ordering $\leq_{\texttt{LEX}}$, we would have a generalization of Mücke's implementation. However, we now want to motivate the need for a more general implementation. This example is suggested by the experience of F. Sullivan and I. Beichl (at the Center for Computing Sciences, Bowie, Maryland) with using the SoS scheme on points on a regular three dimensional grid. The symbolic perturbation may cause our triangulation algorithm to behave very badly. What do we mean by badly? Ideally, we want the triangles to have no sharp angles. For instance, any triangle formed by 3 points in a row of the grid is degenerate, and should be avoided. But our greedy algorithm (guided by any symbolic perturbation scheme) may not know how to avoid such situations.

We first need to fix the algorithm for triangulation. For the present study, assume the the **greedy algorithm**: given a set S of n points, form all $m = \binom{n}{2}$ edges from pairs of points in S. Sort these edges by non-decreasing order of their lengths:

$$e_1, e_2, \ldots, e_m$$

where $|e_i| \leq |e_j|$. We go through this list in order, and put e_i into our triangulation T as long as e_i does not intersect any of the previously picked edges. To break ties, we use our blackbox with pure lexicographic ordering. Consider the performance of the greedy algorithm on the following input sequence

$$(x_1, y_2, x_2, y_2, \ldots, x_{25}, y_{25})$$

representing the points $p_i = (x_i, y_i)$ $(i = 1, \ldots, 25)$. Suppose the points lie on the 5×5 grid:

$$\begin{array}{ccccc} p_1 = (1,1), & p_2 = (1,2), & p_3 = (1,3), & p_4 = (1,4), & p_5 = (1,5) \\ p_6 = (2,1), & p_7 = (2,2), & & \cdots & p_{10} = (2,5) \\ p_{11} = (3,1), & p_{12} = (3,2), & & \cdots & p_{15} = (3,5) \\ p_{16} = (4,1), & p_{17} = (4,2), & & \cdots & p_{20} = (4,5) \\ p_{21} = (5,1), & p_{22} = (5,2), & & \cdots & p_{25} = (5,5). \end{array}$$

Figure 2 shows the result of applying the pure lexicographic perturbation. It is not a good triangulation by any measure.

THIS IS A PLACE HOLDER – NO FIGURE YET

Figure 2: Triangulation of points on a grid

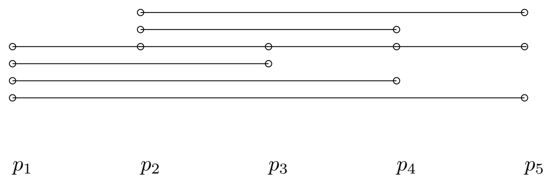

Figure 3: Schematic triangulation of a set of points on a line.

To understand analytically this behavior, let us analyze the output when there is only one row of points $p_1 = (1,1), p_2 = (1,2), \ldots, p_n = (1,n)$. Let d_{ij} be the distance between p_i and p_j. Clearly, $d_{ij} = |i - j|$. What is the result of comparing $d_{i,i+2} : d_{j,j+2}$?

So we would like to have general orderings at our disposal. It turns out that there is a beautiful theory of admissible orderings that can help us do this.

Another way to prevent triangulations such as Figure 2 is to use randomization. Suppose we use lexicographic ordering, but we first randomly reorder the variables first.

Exercises

Exercise 4.1: Assume the total degree ordering. How can the successive evaluations of a polynomial be sped up at the cost of some space (to store intermediate results)? Quantify this speedup. ♢

Exercise 4.2: (Paterson) Give an example of a perturbation scheme that does not arise from our black box. ♢

End Exercises

§5. Algebraic Consistency

We address the consistency of admissible sign functions. We will use an algebraic approach here.

Let R be any commutative ring with unit 1. For instance, $R = \mathbb{Z}$ or $R = \mathbb{R}$. A function $\sigma : R \to \{-1, 0, +1\}$ is called a **sign function** for R provided it satisfies these 4 axioms. For all $a, b \in R$:

(A0) $\sigma(a) = 0$ iff $a = 0$.

(A1) $\sigma(-a) = -\sigma(a)$.

(A2) $\sigma(a) > 0$ and $\sigma(b) > 0$ implies $\sigma(a+b) > 0$.

(A3) $\sigma(ab) = \sigma(a)\sigma(b)$.

If σ is a sign function for R, then it induces a total ordering $<_\sigma$ on R where

$$a <_\sigma b \Rightarrow \sigma(a-b) = -1.$$

It is easy to verify that $<_\sigma$ is indeed a total ordering: if a, b are distinct elements in R, then $a <_\sigma b$ or $b <_\sigma a$, by (A0). Moreover, (A1) says that only one of $a <_\sigma b$ or $b <_\sigma a$ must hold. Transitivity follows from (A2).

We need the following lemma about admissible orderings $\leq_A$:

Lemma 1. *Let $u, v, u', v' \in \mathrm{PP}$ and either $u \neq u'$ or $v \neq v'$. Then $uv \leq_A u'v'$ implies $u <_A u'$ or $v <_A v'$.*

See [8] for a proof. The next theorem constitute our "algebraic consistency" claim.

Theorem 2. *Let R be any ordered ring and $S = R[x_1, \ldots, x_n]$. Fix $\mathbf{a} = (a_1, \ldots, a_n) \in R^n$ and any admissible ordering $\leq_A$. Let $\sigma = \sigma_{\leq_A, \mathbf{a}}$ be the function that assigns to each polynomial $p \in S$ the sign $\mathtt{sign}_{\leq_A}(p; \mathbf{a})$ (see (4)). Then σ is a sign function for S.*

Proof. Recall that for a non-zero $p \in S$, our black box sign, $\sigma(p)$, is the first non-zero value in the sequence $\partial(p; \mathbf{a})$. Axiom (A0) is immediate. Axiom (A1) comes from the fact that $\partial(-p; \mathbf{a})$ is obtained from $\partial(p; \mathbf{a})$ by negating every value. Axiom (A2) comes from the fact that $\partial(p+q; \mathbf{a}) = \partial(p; \mathbf{a}) + \partial(q; \mathbf{a})$ (componentwise addition). To show (A3), let the first non-zero entry of $\partial(p; \mathbf{a})$ be $\partial_u(p)(\mathbf{a})$ and the first non-zero entry of $\partial(q; \mathbf{a})$ be $\partial_v(q)(\mathbf{a})$, for some $u, v \in \mathrm{PP}$. Then

$$\partial_{uv}(pq) = \sum_{u'} \partial_{u'}(p)\partial_{v'}(q)$$

where $u' \in \mathrm{PP}$ ranges over all divisors of uv and $v' = uv/u'$. But the previous lemma, unless $u = u'$ and $v = v'$, $u'v' \leq_A uv$ implies $u >_A u'$ or $v >_A v'$. Hence $\partial_{u'}(p)(\mathbf{a}) = 0$ or $\partial_{v'}(p)(\mathbf{a}) = 0$. This proves

$$\partial_{uv}(pq)(\mathbf{a}) = \partial_u(p)(\mathbf{a})\partial_v(q)(\mathbf{a}).$$

The same argument shows that for all $w <_A uv$,

$$\partial_w(pq)(\mathbf{a}) = 0.$$

Thus, the first non-zero entry of $\partial(pq; \mathbf{a})$ has the sign $\partial_u(p)(\mathbf{a})\partial_v(q)(\mathbf{a}) = \sigma(p)\sigma(q)$, as desired. **Q.E.D.**

¶11. Geometric Consistency. To motivate the need for a geometric notion of consistency, suppose that we have n collinear points, $q_1, \ldots, q_n \in \mathbb{R}^2$. Let Δ_{ijk} denote determinant whose sign is the $LeftTurn(q_i, q_j, q_k)$ predicate (Lecture II.1). Since the lines are collinear, $\Delta_{ijk} = 0$ for all i, j, k. On the other hand, our black box will assign a non-zero sign $s_{ijk} \in \{-1, +1\}$ to Δ_{ijk}. The geometric consistency question asks to know if there is an actual perturbation of $q_1, \ldots, q_n$ which achieves these s_{ijk}'s. This question is answered in the affirmative in [7].

It is a consequence of the next theorem. For $\delta > 0$ and $\mathbf{a}, \mathbf{a}' \in \mathbb{R}^n$, we say $\mathbf{a}'$ is a **δ-perturbation** of $\mathbf{a}$ if $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{a}' = (a'_1, \ldots, a'_n)$ and $|a'_i - a_i| \leq \delta$ for all $i = 1, \ldots, n$.

Theorem 3. *Let σ be any black box sign function for $\mathbb{R}[x_1, \ldots, x_n]$ (based on some admissible ordering and a choice of $\mathbf{a} \in \mathbb{R}^n$). For any finite set $P \subseteq \mathbb{R}[x_1, \ldots, x_n]$ of polynomials and any $\delta > 0$, there exists $\mathbf{a}' = (a'_1, a'_2, \ldots, a'_n)$, a δ-perturbation of $\mathbf{a}$, such that for all $p \in P$,*

$$\sigma(p) = \mathtt{sign}(p(\mathbf{a}')).$$

In other words, the signs assigned by our black box to any finite set P of real polynomials can be achieved by an actual perturbation. Suppose A' is the transformed algorithm in figure 1, and $\mathbf{a}$ is any input for A'. Consider the branching (computation) path π through A' that is taken by input $\mathbf{a}$. This result shows that there is an arbitrarily small perturbation $\mathbf{a}'$ of $\mathbf{a}$ such that $\mathbf{a}'$ is non-degenerate for A' and $\mathbf{a}'$ would exact the same path π.

Exercises

Exercise 5.1: Show the following properties of $<_\sigma$ where σ is a sign function on R. For all $a, b, c, d \in R$:
(a) $a <_\sigma b$ implies $a + c <_\sigma b + c$.
(b) $a <_\sigma b$, $\sigma(c) > 0$ implies $ac <_\sigma bc$.
(c) If b^{-1}, a^{-1} exists, then $a <_\sigma b$ iff $b^{-1} <_\sigma a^{-1}$.
(d) $a + b <_\sigma c + d$ implies $a <_\sigma c$ or $b <_\sigma d$.
(e) $\sigma(a) = \sigma(b) = +1$ and $ab <_\sigma cd$ implies $a <_\sigma c$ or $b <_\sigma d$. ♢

Exercise 5.2: Show that the function σ defined by the Monma, et al, scheme arises from an admissible sign function. ♢

End Exercises

§6. Perturbation Framework

Following Seidel, we define a **problem** to be a partial function $f : X \to Y$ with suitable topologies on the input X and output Y spaces. We write $f(x) \downarrow$ and $f(x) \uparrow$ depending on whether $f(x)$ is defined or not. Typically, $X = \mathbb{R}^{nd}$ (a set of n points in d dimensions, with the usual Euclidean topology, and $Y = D \times R$ where D is a finite set with the discrete topology and R is a direct union of real spaces with the Euclidean topology. The domain of f is $dom(f) := \{x \in X : f(x) \downarrow\}$. We say f is **generic** if the closure $\overline{dom(f)} = X$.

Let us give two examples: (1) Convex hull volume and (2) Convex hull facet structure. In both cases, $X = \mathbb{R}^{nd}$, In (1), $Y = \mathbb{R}$. In (2), $Y = D \times R$ where D is the finite set of all the labeled combinatorial structures that may be output. Also, R can be empty, or for our purposes below, we can let R denote the k-dimensional volume of each k-dimensional face of the output. In this way, problem (1) is can be embedded in problem (2). It is imporant to note that the structures are labeled by the index of input points.

Let z be a set of nd variables (called input variables). A **branching straightline program** (BSLP) $A = A[z]$ is a rooted tree in which each non-branching node computes a value (using an input variable or previously computed values) or evaluates a predicate p on the input and then performs a three-way branch according to the sign of $p(x)$. Each non-branching node is associated with a (programming) variable. Note that we assume that predicates p directly operates on the input x. For any input x, the computation on x is a path that is either non-terminating or terminates at a leaf which by definition is a non-branching node. The output of A on x, denoted $A(x)$, is undefined if the path taken by x is non-terminating, and otherwise is the value computed at last node of the path. We say A is an algorithm for f if for all $x \in X$, $A(x) = f(x)$. An algorithm is **generic** if is an algorithm for a generic problem. It is **general** if it defines a total function on X.

If $x \in X$, a perturbed instance of x is a curve x^* rooted at x: $x^* : \mathbb{R}_{\geq 0} \to X$ such that $x^*(0) = x$. A perturbation scheme Q defines a perturbed instance for each $x \in X$. Once the perturbation is fixed, any problem f is transformed into a perturbed problem

$$f^* : X \to Y$$

where $f^*(x) = \lim_{n\to\infty} f(x(1/n))$ where $f^*(x)$ may be undefined if the limit does not exist. We say that the perturbation scheme Q is **valid** for f if $f^*(x)$ is defined for all $x \in X$ and is non-zero.

We similarly say Q is **valid** for A if $p^*(x) = \lim_{n\to\infty} p(x(1/n))$ is defined and non-zero for all predicates in A. If Q is valid for A, we can transform A to some A^* in the obvious way.

Theorem: Let A be an algorithm for f. Assume Q is valid for f and for algorithm A. Then A^* solves f^* and for each $x \in X$, if f is continuous at x then $A^*(x) = f(x)$. Moreover, A^* is a general algorithm.

Pf: Since Q is valid for f, the problem f^* is total. Also, f^* must be computed by A^*. If f

We can reformulate the above theorem in terms of $\mathbb{R}(\Omega)$ where $\mathbb{R}(\Omega) = \mathbb{R}(\omega)$, ω is an infimaximal.

§7. Numerical Perturbation

We present some efficient numerical schemes for perturbation. In particular, we are interested in numerical linear perturbations: $x \mapsto x + \varepsilon x^*$ where x^*

§8. Perturbing towards Non-Degenerate Instances

A fundamentally different idea for perturbation is proposed by Raimund Seidel: for any problem P and for each input size n, suppose we can find a non-degenerate instance G_n. Then for any input instance I of size n, we want to use the perturbed instance

$$I + \varepsilon G_n$$

where $\varepsilon > 0$ is a small constant.

Recall that one of our basic goals figure 1 is to transform a generic A to a general A'. In this case, we need to be able to generate G_n on the fly. This turns out to be easy in some problems. Consider the problem P of computing the convex hull of points in $\mathbb{R}^d$. For each n, there are well-known constructions for a set G_n of n points in general position. Basically, G_n can be any n distinct points chosen on the moment curve,

$$C = \{(t, t^2, \ldots, t^d) : t \in \mathbb{R}\}$$

Another direction in perturbation research is known as "controlled perturbation", as opposed to random perturbation (or perturbation where we don't care how the perturbation behaves as long as it is bounded). An example of why we want this is in the intersection of two convex polyhedron. We want the perturned polygons to grow outwardly, so that any tangential intersection between a vertex and a face can be detected. In random perturbation, we may not see such tangential intersections.

Removing Degeneracies: Gomez et al [4, 3] consider algorithms for finding orthogonal and perspective projections (respective) to remove degenaracies in point sets.

References

[1] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 16–23, 1995.

[2] A. R. Forrest. Foreword. *ACM Trans. on Graphics*, 3& 4, 1984. Two Special Issues on Computational Geometry. Guest Editors: R. Forrest, L.Guibas, J.Nievergelt.

[3] F. Gomez, F. Hurtado, T. Sellares, and G. Toussaint. On degeneracies removable by perspective projections. *Int. J. of Mathematical Algorithms*, 2:227–248, 2001.

[4] F. Gomez, S. Ramaswami, and G. Toussaint. On removing non-degeneracies assumptions in computational geometry. In *Proc. Italian Conf. on Algorithms*, pages 52–63, 1997. March 12-14, 1997, Rome, Italy.

[5] C. Monma, M. Paterson, S. Suri, and F. Yao. Computing Euclidean maximum spanning trees. *Algorithmica*, 5:407–419, 1990. Also: 4th ACM Symp.Comp.Geom. (1988).

[6] R. Sedgewick. *Algorithms in C: Part 5, Graph Algorithms.* Addison-Wesley, Boston, MA, 3rd edition edition, 2002.

[7] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *J. of Computer and System Sciences*, 40(1):2–18, 1990. Also: *Proc. 4th ACM Symp.Comp.Geom*, 1988, pp.134–142.

[8] C. K. Yap. Symbolic treatment of geometric degeneracies. *J. of Symbolic Computation*, 10:349–370, 1990. Also, *Proc. International IFIPS Conf. on System Modelling and Optimization,* Tokyo, 1987, Springer Lecture Notes in Control and Information Science, Vol.113, pp.348-358.