PRINCETON UNIV. F'18 COS 521: ADVANCED ALGORITHM DESIGN

Lecture 16: Seperation, optimization, and the ellipsoid method

Lecturer: *Christopher Musco*

This section of the course focus on solving the optimization problems of the form:

$$\min_x f(x) \qquad \text{such that} \qquad x \in \mathcal{K},$$

where f is a convex function and $\mathcal{K}$ is a convex set. Recall that any convex f satisfies the following equivalent inequalities:

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \qquad \forall x, y, \lambda \in [0,1] \tag{1}$$

$$f(x) - f(y) \leq \nabla f(x)^T (x - y) \qquad \forall x, y. \tag{2}$$

Also recall that a convex set is any set where, for all $x, y \in \mathcal{K}$, if $z = \lambda x + (1-\lambda)y$ for some $\lambda \in [0,1]$, then $z \in \mathcal{K}$.

1 Gradient descent recap

Last lecture we analyzed the gradient descent procedure for solving a convex optimization problem over a convex set.

GRADIENT DESCENT FOR CONSTRAINED OPTIMIZATION
Let $\eta == \frac{D}{G\sqrt{T}}$.
Let x_0 be any point in $\mathcal{K}$.
Repeat for $i = 0$ **to** T
$y_{i+1} \leftarrow x_i - \eta \nabla f(x_i)$
$x_{i+1} \leftarrow$ Projection of y_{i+1} on $\mathcal{K}$.
At the end output $\bar{x} = \frac{1}{T}\sum_{i=0}^{T} x_i$.

D is the diameter of $\mathcal{K}$ (or, if our problem is unconstrained, simply an upper bound on $\|x_0 - x^\star\|$). G is an upper bound on the size of f's gradient, i.e. $\|\nabla f(x)\|_2 \leq G$, $\forall x$.

Last lecture we proved:

Lemma 1. *Let $x^\star \ \arg\min_{x \in \mathcal{K}} f(x)$. After T steps of gradient descent,*

$$f(\bar{x}) - f(x^\star) \leq \frac{DG}{2\sqrt{T}}.$$

So after $T = \frac{4D^2G^2}{\epsilon^2}$ steps, we have $f(\bar{x}) - f(x\star) \leq \epsilon$.

2 Online Gradient Descent

If you look back to last lecture, you will see that we actually proved something a bit stronger. We showed that in every step,

$$f(x_i) - f(x^\star) \leq \frac{1}{2\eta}\left(\|x_i - x^\star\|_2^2 - \|x_{i+1} - x^\star\|_2^2\right) + \frac{\eta}{2}G^2. \tag{3}$$

The only thing our proof used about f was the $\|\nabla f(x_i)\|_2$. In particular, it could have been that f differed at every iteration! If we have functions $f_0, \ldots, f_T$ and run gradient descent with updates equal to $-\eta\nabla f_i(x_i)$ on iteration i, (3) allows us to obtain the bound:

$$\frac{1}{T}\sum_{i=0}^{T} f_i(x_i) - \frac{1}{T}\sum_{i=0}^{T} f_i(x^\star) \leq \frac{DG}{2\sqrt{T}}, \tag{4}$$

for any $x_\star \in \mathcal{K}$.

This is a **regret bound**, just like we saw for the experts problem and multiplicative weights update. I.e. instead of optimizing one fixed function f, suppose our goal is to output a vector x_i at time i, which corresponds to some strategy: e.g. a linear classifier for spam prediction, or a distribution of funds over stocks, bonds, and other assets.

After playing strategy x_i, we incur some loss $f_i(x_i)$. For example, suppose we receive a data point a_i and want to classify a_i as positive or negative (e.g. yes it's spam, no it's not) by looking at sign $a_i^T x_i$. After we play x_i, the true label of a_i is revealed as $b_i \in \{-1, 1\}$ and we pay the convex penalty:

$$f_i(x_i) = \max(0, 1 - a_i^T x \cdot b_i)$$

We update our classifier based on $\nabla f_i(x_i)$ to obtain x_{i+1} and proceed. This procedure is referred to as **online gradient descent** and the problem of minimizing our cumulative loss $\sum_i f_i(x_i)$ is referred to as **online convex optimization**.

The bound of (4) means that our average penalty is no worse than that of the best fixed classifier $x^\star$ by a factor that grows sublinearly in $\sqrt{T}$. This powerful observation, due to [1], has a number of applications, included below (we did not discuss these in lecture).

2.1 Case Study: Online Shortest Paths

The Online Shortest Paths problem models a commuter trying to find the best path with fewest traffic delays. The traffic pattern changes from day to day, and she wishes to have the smallest average delay over many days of experimentation.

We are given a graph $G = (V, E)$ and two nodes s, t. At each time period i, the decision maker selects one path p_i from the set $P_{s,t}$ of all paths that connect s, t (the choice for the day's commute). Then, an adversary independently chooses a weight function $w_i : E \to \mathbb{R}$ (the traffic delays). The decision maker incurs a loss equal to the weight of the path he or she chose: $\sum_{e \in p_i} w_i(e)$.

The problem of finding the best would be natural to consider this problem in the context of expert advice. We could think of every element of $P_{s,t}$ as an expert and apply the multiplicative weights algorithm we have seen before. There is one major flaw with this

approach: there may be exponentially many paths connecting s, t in terms of the number of nodes in the graph. So the updates take exponential time and space in each step, and furthermore the algorithm needs too long to converge to the best solution.

Online gradient descent can solve this problem, once we realize that we can describe the set of all distributions x over paths $P_{s,t}$ as a convex set $\mathcal{K} \in \mathbb{R}^m$, with $O(|E| + |V|)$ constraints. Then the decision maker's expected loss function would be $f_i(x) = w_i^T \cdot x$. The following formulation of the problem as a convex polytope allows for efficient algorithms with provable regret bounds.

$$\sum_{e=(s,w),w\in V} x_e = \sum_{e=(w,t),w\in V} x_e = 1 \qquad \text{Flow value is 1.}$$

$$\forall w \in V, w \neq u, v, \sum_{e \ni w} x_e = 0 \qquad \text{Flow conservation.}$$

$$\forall e \in E, 0 \leq x_e \leq 1 \qquad \text{Capacity constraints.}$$

What is the meaning of the decision maker's move being a distribution over paths? It just means a fractional solution. This can be decomposed into a combination of paths as in the lecture on approximation algorithms. She picks a random path from this distribution; the expected regret is unchanged.

2.2 Case Study: Portfolio Management

Let's return to the portfolio management problem discussed in context of multiplicative weights. We are trying to invest in a set of n stocks and maximize our wealth. For $t = 1, 2, \ldots,$ let $r^{(t)}$ be the vector of relative price increase on day t, in other words

$$r_i^{(t)} = \frac{\text{Price of stock } i \text{ on day } t}{\text{Price of stock } i \text{ on day } t-1}.$$

Some thought shows (confirming conventional wisdom) that it can be very suboptimal to put all money in a single stock. A strategy that works better in practice is *Constant Rebalanced Portfolio* (CRB): decide upon a *fixed* proportion of money to put into each stock, and buy/sell individual stocks each day to maintain this proportion.

Example 1. *Say there are only two assets,* stocks *and* bonds. *One CRB strategy is to put split money equally between these two. Notice what this implies: if an asset's price falls, you tend to* buy more of it, *and if the price rises, you tend to* sell it. *Thus this strategy roughly implements the age-old advice to "buy low, sell high."Concretely, suppose the prices each day fluctuate as follows.*

	Stock $r^{(t)}$	Bond $r^{(t)}$
Day 1	4/3	3/4
Day 2	3/4	4/3
Day 3	4/3	3/4
Day 4	3/4	4/3
...	...	...

Note that the prices go up and down by the same ratio on alternate days, so money parked fully in stocks or fully in bonds earns nothing in the long run. (Aside: This kind of fluctuation is not unusual; it is generally observed that bonds and stocks move in opposite directions.) And what happens if you split your money equally between these two assets? Each day it increases by a factor $0.5 \times (4/3 + 3/4) = 0.5 \times 25/12 \approx 1.04$. *Thus your money grows exponentially!*

Exercise: *Modify the price increases in the above example so that keeping all money in stocks or bonds alone will cause it to drop exponentially, but the 50-50 CRB increases money at an exponential rate.*

CRB uses a fixed split among n assets, but what is this split? Wouldn't it be great to have an angel whisper in our ears on day 1 what this magic split is? Online optimization is precisely such an angel. Suppose the algorithm uses the vector $x^{(t)}$ at time t; the ith coordinate gives the proportion of money in stock i at the start of the tth day. Then the algorithm's wealth increases on t by a factor $r^{(t)} \cdot x^{(t)}$. Thus the goal is to find $x^{(t)}$'s to maximize the final wealth, which is

$$\prod_t r^{(t)} \cdot x^{(t)}.$$

Taking logs, this becomes

$$\sum_t \log(r^{(t)} \cdot x^{(t)}) \tag{5}$$

For any fixed $r^{(1)}, r^{(2)}, \ldots$ this function happens to be concave, but that is fine since we are interested in maximization. Now we can try to run online gradient descent on this objective. By Zinkevich's theorem, the quantity in (5) converges to

$$\sum_t \log(r^{(t)} \cdot x^*) \tag{6}$$

where x^* is the best money allocation in hindsight.

This analysis needs to assume very little about the $r^{(t)}$'s, except a bound on the norm of the gradient at each step, which translates into a weak condition on price movements. In the next homework you will apply this simple algorithm on real stock data.

3 Stochastic Gradient Descent

Beyond it's direct applications, online gradient descent also gives a way of analyzing the ubiquitous stochastic gradient descent method. In the most standard setting, this method applies to convex functions that can be decomposed as the sum of simpler convex function:

$$f(x) = \sum_{j=1}^{n} g_j(x) \qquad \text{where} \qquad \text{each } g_j \text{ is convex .}$$

This is a very typical structure in machine learning, where $f(x)$ sums some convex loss function over n individual data points. We've seen the example of least squares regression:

$$f(x) = \|Ax - b\|_2^2 = \sum_{i=1}^{n} \left(a_i^T x - b\right)^2.$$

Other examples include objective functions for robust function fitting, like ℓ_1 regresssion: $f(x) = \sum_{i=1}^n \left|a_i^T x - b\right|$, and objective functions used in linear classification, like the hinge loss: $f(x) = \sum_{i=1}^n \max(0, 1 - a_i^T x \cdot b_i)$.

The key observation is that, when $f(x) = \sum_{j=1}^n g_j(x)$, $\nabla f(x) = \sum_{j=1}^n \nabla g_j(x)$. So if we select j uniformly at random, $n\nabla g_j(x)$ gives an unbiased estimator for $\nabla f(x)$. Stochastic gradient descent uses this estimate in place of $\nabla f(x)$. The advantage of doing so is that the estimate is much faster to compute, typically saving a factor n. For example, computing $\nabla f(x)$ for any of the objectives we just listed takes $O(nd)$ time, while computing $\nabla g_j(x)$ takes $O(d)$ time.

Let G' be an upper bound on $\|n\nabla g_j(x)\|_2$ for all g_i, x.

STOCHASTIC GRADIENT DESCENT
Let $\eta == \frac{D}{G'\sqrt{T}}$.
Let x_0 be any point in $\mathcal{K}$.
Repeat for $i = 0$ **to** T
Choose j_i uniformly from $\{1, \ldots, n\}$.
$y_{i+1} \leftarrow x_i - \eta \cdot n\nabla g_{j_i}(x_i)$
$x_{i+1} \leftarrow$ Projection of y_{i+1} on $\mathcal{K}$.
At the end output $\bar{x} = \frac{1}{T}\sum_{i=0}^T x_i$.

The output of the SGD algorithm is a random variable, so we give a bound on it's expected performance. First we use convexity and linearity of expectation:

$$\mathbb{E}f(\bar{x}) - f(x^\star) \leq \frac{1}{T}\sum_{i=0}^T \mathbb{E}f(x_i) - f(x^\star) \leq \frac{1}{T}\sum_{i=0}^T \mathbb{E}\nabla f(x_i)^T(x_i - x^\star).$$

Now, for any x_i, the expectation of $n \cdot \nabla g_{j_i}(x_i)$ over our random choice of j_i is equal to $\nabla f(x_i)$. So the expression above is equivalent to:

$$\frac{1}{T}\sum_{i=0}^T \mathbb{E}\nabla f(x_i)^T(x_i - x^\star) = \frac{1}{T}\sum_{i=0}^T \mathbb{E}n\nabla g_{j_i}(x_i)^T(x_i - x^\star).$$

Finally, any particular realization of $g_{j_1}, \ldots, g_{j_T}$, we can use (4) to bound:

$$\frac{1}{T}\sum_{i=0}^T \mathbb{E}n\nabla g_{j_i}(x_i)^T x_i - n\nabla g_{j_i}(x_i)^T x^\star \leq \frac{DG'}{\sqrt{T}}.$$

I.e., we let $f_i(y) = g_{j_i}(x_i)^T y$ be the shifting objective function in online gradient descent. We conclude that $\mathbb{E}f(\bar{x}) - f(x^\star) \leq \frac{DG'}{\sqrt{T}}$, so we obtain error ϵ if we set $T = O(\frac{D^2G'^2}{\epsilon^2})$

How does this bound compare to standard gradient descent? First note that $\|\nabla f(x)\|_2 \leq \sum_{i=1}^n \|\nabla g_i(x)\|_2$ by triangle inequality. And since $\sum_{i=1}^n \|\nabla g_i(x)\|_2 \leq n \max \|\nabla g_i(x)\|_2$, we always have $G' \geq G$. This is expected – in general, using stochastic gradient's will slow down the convergence of our algorithm.

However, in many cases this is more than made up for by how much we save in computing gradients. In the examples given above, as long as $G' \leq \sqrt{n}G$, we get an overall runtime savings by using SGD instead of standard gradient descent.

4 The Ellipsoid Method

The advantage of gradient descent and related methods is cheap per iteration cost – in many cases linear in the size of the input problem. Furthermore, the convergence rate of gradient descent is *dimension independent*, depending loosely on specific problem parameters, and not at all on the problem size. At the same time, at least without making additional assumptions about f, this convergence rate is relatively slow – a dependence on $1/\epsilon^2$ limits the possibility of using gradient descent to obtain high accuracy solutions to constrained convex optimization problems.

In the remainder of this lecture and next lecture, we will look at alternative methods for minimizing convex functions over convex sets which offer far more accurate solutions. In fact, for many special cases like linear programming, these methods can be shown to produce an exact solution in polynomial time[1].

The first such method we consider is the ellipsoid method, which originates in work by Shor (1970), and Yudin and Nemirovskii(1975). In 1979 Khachiyan showed that the ellipsoid method can be used to solve linear programs in polynomial time, giving the first polynomial time solution to the problem. Next lecture we will look at interior point methods, which were shown by Karmarkar to solve LPs in polynomial time in 1984. Generally, interior point methods are considered practically faster than ellipsoid methods (and give better asymptotic runtimes) but they also apply to less general problems.

4.1 Find a point in a convex set

The problem that the ellipsoid method actually solves is as follows:

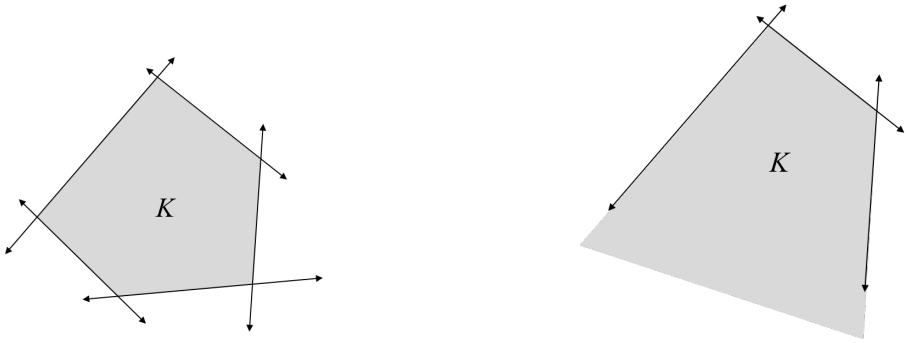
Problem 2. *Given a convex body (i.e., a closed and bounded convex set) $\mathcal{K}$, find some point $x \in \mathcal{K}$ or output EMPTY if $\mathcal{K} = \emptyset$.*

Intuitively, we can see that an algorithm for this problem can be used blackbox to minimize a convex function $f(x)$ over a convex set. In particular, if f is convex, then the set $\{x : f(x) \leq z\}$ is convex. It follows that $\mathcal{K} \cap \{x : f(x) \leq z\}$ is convex. So, we can use an algorithm for Problem 2, run on $\mathcal{K} \cap \{x : f(x) \leq z\}$, to binary searching over values of z and find the minimum z (or close to it) such that $f(x) \leq z$ for some $x \in \mathcal{K}$.

There are a number of details to think about here which we do not have time to cover. For example, we need at least upper and lower bounds on $\min f(x)$ to perform binary search. As suggested in class, it might also be that $\mathcal{K} \cap \{x : f(x) \leq z\}$ is not bounded, which violates the input assumption of Problem 2. Nisheeth Vishnoi's lecture notes on the ellipsoid method [2] are a good place to learn about some of these points in more detail

For now, we restrict our attention to solving Problem 2.

[1] This can be a complicated issue, involving many details. E.g. for linear programming the methods we will look at run in "weakly polynomial time", meaning time polynomial in the input size and in L, which is a bound on the maximum number of of bits required to specify each entry of our constraint matrix if all entries are integers specified in binary (i.e., they can't be specified in floating point).



(a) Closed and bounded convex set $K \subset \mathbb{R}^d$. (b) Unbounded convex set $K \subset \mathbb{R}^d$.

Figure 1: For this lecture, we restrict our attention to finding a point inside a bounded convex set $\mathcal{K}$ that is specified by a separation oracle

4.2 Presenting a convex body: separation oracles

The first question to ask it how the input to Problem 2 is even specified. For some convex optimization problems (e.g. linear programming) we had a convenient way of representing $\mathcal{K}$ as the intersection of many halfspaces (a polytope) but this is not always the case. We need a more generic approach.

One simple way to present a body to the algorithm is via a *membership oracle*: a black-box program that, given a point x, tells us if $x \in \mathcal{K}$. We will work with a stronger version of the oracle, which relies upon the following fact.

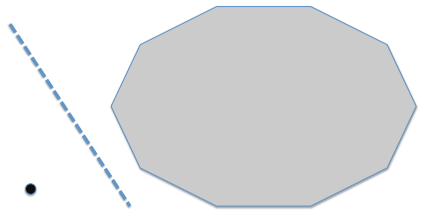


Figure 2: Separating hyperplane theorem/Farka's Lemma: Between every convex body and a point outside it, there is a hyperplane.

Separating hyperplane theorem: If $\mathcal{K} \subseteq \mathbb{R}^n$ is a closed convex set and $x \in \mathbb{R}^n$ is a point, then one of the following holds:

1. $p \in \mathcal{K}$
2. There is a hyperplane that separates x from $\mathcal{K}$. I.e. there is some $c \in \mathbb{R}^d$ such that $c^T x > c^T y \ \forall y \in \mathcal{K}$.

This claim was proven in our lecture on LP duality. It prompts the following definition:

Definition 1. *A* **Separation Oracle** *for a convex set $\mathcal{K}$ is a procedure which given x, either tells that $p \in K$ or returns a vector $c \in \mathbb{R}^d$ which specifies a hyperplane that separates p and all of K.*

A separating hyperplane oracle provides a lot more information than a membership oracle. It tells us not only that x is not in $\mathcal{K}$, by *why* – e.g. roughly what direction we would have to move in to get close to $\mathcal{K}$.

For many convex bodies of practical interest, it is possible to construct an efficient separation oracle. This oracle will be used blackbox by the ellipsoid method, so its runtime will dictate the cost of solving Problem 2.

Example 2. *A polytope is a convex set described by linear constraints:*

$$\mathcal{K} = \{x : Ax \leq b\}.$$

A separation oracle for K simply needs to compute Ax. If this vector is entrywise $\leq b$, x is in $\mathcal{K}$. Alternatively, if $[Ax]_i > b_i$, we simple output the i^{th} for of A, a_i, as our separating vector.

Example 3. *The set of positive semidefinite matrices includes all symmetric matrices $X \in \mathbb{R}^{d \times d}$ such that $w^T X w \geq 0$ for all w. This is a convex set, which we will discuss more in Lecture 18.*

Unlike the polytope example, the set of PSD matrices is defined by infinite linear constraints of the form $\sum_{ij} X_{ij} w_i w_j \geq 0$. We can't check all of these constraints to find a separating hyperplane. However, since any symmetric matrix with all positive eigenvalues is PSD, we can use an eigendecomposition to obtain a separation oracle.

If an input matrix X has any negative eigenvalues ,we take an eigenvector a corresponding to one of these negative eigenvalue and returns the hyperplane $\sum_{ij} X_{ij} a_i a_j = 0$. (Note that a_i's are constants here.)

Example 4. *Not all simple to describe convex sets have efficient (e.g. polynomial time) separation oracles. For example, consider the set of copositive matrices, which includes all (possible asymmetric matrices) $X \in \mathbb{R}^{d \times d}$ with $w^T X w \geq 0$. While similar to the set of PSD matrices, the separation problem for this set is actually NP-hard (and convex optimization over the set of copositive matrices is NP-hard).*

4.3 Main idea

A separation oracle is not sufficient to allow an algorithm to test $\mathcal{K}$ for nonemptiness in finite time. Each time the algorithm questions the oracle about a point x, the oracle could just answer $x \notin \mathcal{K}$, since the convex body could be further from the origin than *all* the (finitely many) points that the algorithm has queried about thus far. After all, space is *infinite*!

Thus the algorithm needs some very rough idea of where $\mathcal{K}$ may lie. It needs $\mathcal{K}$ to lie in some known *bounding box*. The bounding box could be a cube, sphere etc.

The Ellipsoid method will use an ellipsoid as a bounding box. In particular, it solves Problem 2 under the assumption that $\mathcal{K}$ is contained in an n-dimensional ball of finite radius R, and also contains an n-dimensional ball of radius $r < R$ (which ensures that $\mathcal{K}$ can't be arbitrarily small).

Assumption:

1. $\mathcal{K} \subseteq B(\alpha_1, R)$ for some known center α_1 and radius R.
2. $B(\alpha_2, r) \subseteq \mathcal{K}$ for some known center α_2 and radius $r < R$.

It is not always possible to find such bounding balls apriori, but for many problems of practical interest we can. For example, consider a polytope with integer constraints (i.e. every entry in A and b is integral) that takes a total of L bits to represent. Because the polytope is invariant to scaling A and b, this is equivalent to saying that the input constraints have bounded precision. In this case, it is possible to show that $R \lesssim 2^L$ and, either $\mathcal{K}$ is empty, or $r \gtrsim 2^{-L}$. Again, we refer to [2] for more details, and for now assume that R and r exists and can be determined.

The ellipsoid method is an iterative algorithm. It begins with bounding ellipse $B(\alpha_1, R)$ and queries whether or not the *center* of that ellipse is in $\mathcal{K}$. If it is, we are done. If not, is uses the separating hyperplane returned by our separation oracle to cut $B(\alpha_1, R)$ in half, reducing our search space. The new search space is itself represented by an ellipse – we take the minimum volume ellipse that contains the half ellipse $B(\alpha_1, R) \cap \{y : c^T y \leq c^T x\}$.

So overall, we produce a sequence of ellipses:

$$E_0 = B(\alpha, R), E_1, E_2, \ldots, E_T$$

where E_{i+1} is the ellipse of minimal volume containing $E_i \cap \{y : c_i^T y \leq c^T x_i\}$. Here x_i is the center of ellipse E_i, which we pass as a query to our separation oracle, and c_i is the separation vector returned in $x_i \notin \mathcal{K}$.

A representation for E_i can be found analytically in $O(n^2)$ time by using the fact that any ellipse E is defined as containing all points x such that $(x - \alpha)^T A(x - \alpha) \leq 1$ for some symmetric PSD matrix A and center α.

Why do this? Why not just keep track of the half ellipse, query its center, and cut in half again? Why make an approximation at each step by surrounding our search space by a new ellipse? The issue is that it's hard to find the center of a generic convex body, so we very quickly would be unable to decide which point x to query next in our search region.

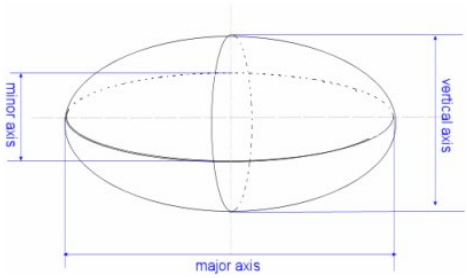


Figure 3: 3D-Ellipsoid and its axes

With our general strategy in place, the only problem is to make sure that the algorithm makes progress at every step. To do so we use the following important lemma:

Lemma 3. *The minimum volume ellipsoid surrounding a half ellipsoid (i.e. $E_i \bigcap H^+$ where H^+ is a halfspace as above) can be calculated in polynomial time and*

$$Vol(E_{i+1}) \leq \left(1 - \frac{1}{2n}\right) Vol(E_i)$$

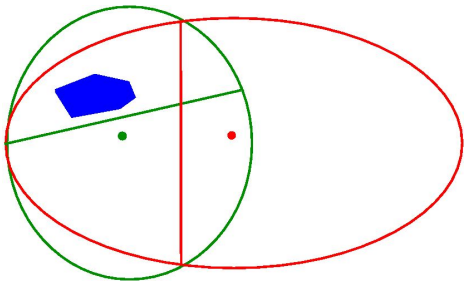


Figure 4: A few of runs of the Ellipsoid method showing the tiny convex set in blue and the containing ellipsoids. The separating hyperplanes do not pass through the centers of the ellipsoids in this figure, although for the algorithm described here they would.

It's an interesting exercise to work through a proof of this lemma – you can follow the proof here: http://people.csail.mit.edu/moitra/docs/6854notes12.pdf.

Thus after T steps the volume of the enclosing ellipsoid has dropped by $(1 - 1/2n)^T \leq \exp(-T/2n)$. Our starting ellipse has volumen $O(R^n)$, so in $T = O(n \log(R^n/r^n) = O(n^2 \log(R/r)$ steps, we can ensure that, either we find a point $x \in \mathcal{K}$, or

$$E_T \leq O(r^n).$$

If this is the case, we can assume that $\mathcal{K}$ is empty since we assumed that if it wasn't, it at least contained a ball of volume $O(r^n)$.

At first glance Lemma 3 might seem weak – ideally we could have hoped to cut the volume of our search region in half with each query. We only cut it by a factor of $(1 - \frac{1}{2n})$, which cost us an extra n factor in iteration complexity. This loss is an inherent cost of approximating our search region by an ellipsoid. Recent work on improving the ellipsoid method seeks to address this issue by maintaining alternative representations of the search region [3].

References

[1] Zinkevich, Martin. Online Convex Programming and Generalized Infinitesimal Gradient Ascent. Proceedings of the International Conference on Machine Learning (ICML), 2013.

[2] Nisheeth K. Vishnoi. Algorithms for Convex Optimization. EPFL lecture notes, 2018. https://nisheethvishnoi.wordpress.com/convex-optimization/.

[3] Yin Tat Lee, Aaron Sidford, and Same Chiu-Wai Wong. A Faster Cutting Plane Method and its Implications for Combinatorial and Convex Optimization. FOCS 2015. 1049-1065.