Linear Programming III

In this lecture we describe a nice algorithm due to Seidel for Linear Programming in 2D (i.e., linear programs with just two variables). We will also briefly discuss using linear programming to model some non-linear problems, such as ones containing absolute values.

Objectives of this lecture

In this lecture, we will

- See an example of using linear programming to model some non-linear problems.
- See Seidel's algorithm for linear programming, an example of a *randomized incremental algorithm*
- Learn how to analyze randomized incremental algorithms

1 Modeling non-linear problems as linear programs

Sometimes we encounter non-linear objectives or constraints in problems we would like to solve, so we can't use linear programming directly, but, we might be able to model them as a linear program anyway by cleverly converting the non-linear stuff into linear stuff! We will look at one example here where we can sometimes model problems containing *absolute values*, even though absolute values are not linear and hence can not be part of a legal linear program.

Problem: L_1 regression

We are given an $n \times d$ matrix A where n is larger than d, and an $n \times 1$ vector b. We'd like to find an x such that $Ax = b$, but since $n > d$, it is not likely that there exists any such x. Instead, we can try to find x that is *close* to satisfying $Ax = b$, or, more specifically

$$\underset{x}{\text{minimize}} \ \|Ax - b\|_1 \coloneqq \underset{x}{\text{minimize}} \ \sum_{i=1}^{n} |A_i \cdot x - b_i|,$$

where $A_i \cdot x \coloneqq \sum_j A_{ij} x_j$.

The definition of the problem almost looks like an LP, except that it contains absolute values which are not legal in an LP. Let's try to convert the expressions containing absolute values into

legal linear constraints instead. We can start by defining variables s_i for $1 \leq i \leq n$. The idea is that s_i is going to model the expression $|A_i \cdot x - b_i|$. So we would love to write the constraint

$$s_i = |A_i \cdot x - b_i|,$$

but of course *we can't because this isn't a valid linear constraint* (absolute value is not a linear function). But remember what the absolute value function means. If $A_i \cdot x - b_i$ is positive, then $s_i = A_i x - b_i$. Otherwise, if it is negative, then $s_i = -(A_i \cdot x - b_i)$. The absolute value is just the larger of these two, so we could therefore rewrite our constraint for s_i as

$$s_i = \max(A_i x - b_i, -(A_i \cdot x - b_i)).$$

Is this any better? Well not quite, because we still can not legally write max in a constraint of a linear program, its not a linear function either. However, one more step will get us to a valid linear program. max just means the larger of the two, so what if we wrote a *pair* of constraints:

$$s_i \geq A_i \cdot x - b_i,$$
$$s_i \geq -(A_i \cdot x - b_i).$$

This is equivalent to $s_i \geq |A_i \cdot x - b_i|$, so we've successfully eliminated the absolute value, but we have also changed the desired equality into an inequality. Could this be bad and lead to s_i being too large? No! Because our objective function is now

$$\underset{x,s}{\text{minimize}} \sum_{i=1}^{n} s_i,$$

so if s_i was ever larger than $|A_i \cdot x - b_i|$, we could make it smaller and lower the objective, so for any optimal solution, the inequalities are all tight and we have $s_i = |A_i \cdot x - b_i|$ as desired. So we can solve L_1 regression as a linear program like so. Note that the x_i variables still exist in the program and its constraints even though they are not preset in the objective.

Variables: Create variables x_i and s_i for $1 \leq i \leq n$

Objective: $\underset{x,s}{\text{minimize}} \sum_{i=1}^{n} s_i,$

Constraints:

- $s_i \geq A_i \cdot x - b_i$ for all $1 \leq i \leq n$
- $s_i \geq -(A_i \cdot x - b_i)$ for all $1 \leq i \leq n$

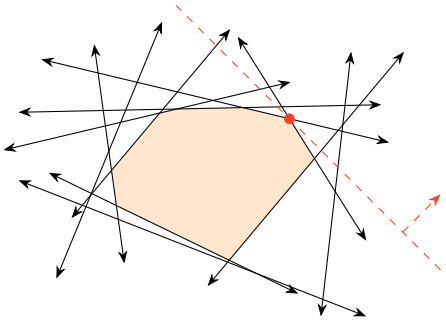
Remark: Absolute values are not legal in linear programs

Its worth repeating a few times just to make sure. Remember that absolute values are not legal linear constraints or objectives for an LP. In this particular scenario, we were able to convert the absolute values into something manageable, but this is not possible for every such problem.

2 Seidel's LP algorithm

We now describe a linear-programming algorithm due to Raimund Seidel that solves the 2-dimensional (i.e., 2-variable) LP problem in $O(m)$ expected time (recall, m is the number of constraints), and more generally solves the d-dimensional LP problem in expected time $O(d!m)$.

Setup In the problem, we are given 2 variables x_1 and x_2. Our objective is to maximize $\mathbf{c} \cdot \mathbf{x}$ for some $\mathbf{c}$ vector [1]. We also have m linear constraints $\mathbf{a}_1 \cdot \mathbf{x} \leq b_1, \ldots, \mathbf{a}_m \cdot \mathbf{x} \leq b_m$. Our goal is to find a solution $\mathbf{x} = (x_1, x_2)$ that satisfies all constraints and maximizes the objective. In the example below, the region satisfying all constraints is given in orange, the dashed line and arrow indicates the direction in which we want to maximize, and the red circle is the feasible point with the maximum objective.



(You should think of sweeping the red dashed line, to which the vector $\mathbf{c}$ is perpendicular, in the direction of $\mathbf{c}$, until you reach the last point that satisfies the constraints. This is the point you are seeking.)

Key Idea: Seidel's algorithm / incremental algorithms

The idea behind Seidel's algorithm is to add in the constraints one at a time, keeping track of the optimal solution for the constraints added so far. This is more broadly called an *incremental algorithm*.

Bounding the feasible region To keep things simple, we'd like to be able to assume that the problem is always bounded. Of course it might not be bounded when we have only added some of the constraints so far, or worse, the problem might not actually be bounded at all! So here's what we can do, let's just add a giant *bounding box* around the feasible region, of the form $-M \leq x_i \leq M$ for a sufficiently large value of M. As long as we pick a large enough M, we can ensure that it encloses all of the vertices of the feasible region, so this won't remove any feasible points unless the problem is unbounded. We can then choose one of the four corners of the box (the one with maximum objective value) as the starting optimal point. If we end up finding

[1] If we wanted to minimize an objective, that is equivalent to maximizing its negation, thus we can always assume a maximizing objective.

that the corner of the box is still the answer after adding all of the constraints, then we know the original problem is unbounded.

The vertices of the feasible region can be written as the intersection of two constraints $\alpha_1 x_1 + \beta_1 x_2 = \gamma_1$ and $\alpha_2 x_1 + \beta_2 x_2 = \gamma_2$. The solution $\mathbf{x} = (x_1, x_2)$ is the solution of

$$\begin{bmatrix} \alpha_1 & \beta_1 \\ \alpha_2 & \beta_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix}$$

If the input numbers are all L-bit numbers, then the solution to this equation can be specified with $(2L+1)$-bit numbers (you could convince yourself that this is true with Cramer's rule, for example), so we can pick an M that is larger than any $(2L+1)$-bit number and we know that it will definitely bound the vertices of the feasible region (and the introduction of M will not change the bit complexity of the problem).

2.1 The algorithm

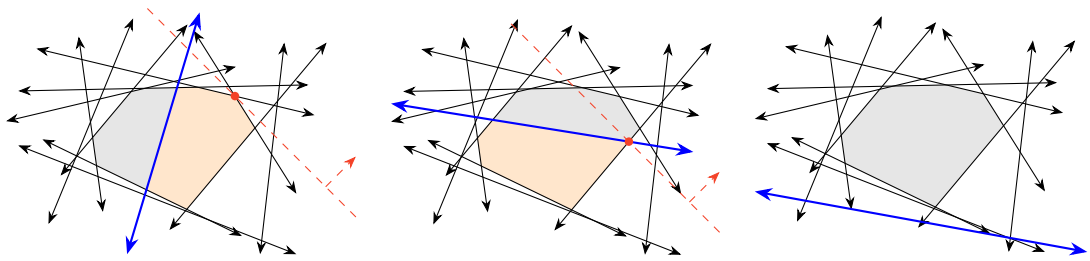
Now, we will talk about the actual algorithm. Remember that this is an incremental algorithm, and the idea is to add the constraints one at a time. So, we will look at a generic iteration of the algorithm to understand how each steps works. We also walk through a small example in Appendix A.

Algorithm iteration Suppose that we are in the middle of running the algorithm. So far, we have found the optimal solution $\mathbf{x}^*$ given the first $m-1$ constraints. Now, we add in the m^{th} constraint $\mathbf{a}_m \cdot \mathbf{x} \leq b_m$, let's call it c_m. We now have two cases for which the algorithm might proceed, based on whether $\mathbf{x}^*$ satisfies c_m. We can test this in $O(1)$ by simply plugging $\mathbf{x}^*$ into c_m.

Case 1: If $\mathbf{x}^*$ satisfies c_m, then $\mathbf{x}^*$ is still optimal.

Case 2: If $\mathbf{x}^*$ doesn't satisfy c_m, then either there is a new optimal point which we need to find, or the LP is no longer feasible.

As an example, we can consider the LP from earlier. We'll denote the new constraint with blue, the old feasible area with gray, and the new feasible are by orange. Adding the constraint on the left example will give us a smaller feasible region, but $\mathbf{x}^*$ will remain optimal. Adding the constraint in the middle example will mean $\mathbf{x}^*$ is no longer in the feasible region, meaning that we will have a new optimal point. Adding the constraint on the right example will also mean that $\mathbf{x}^*$ is no longer in the feasible region, however the entire LP will now be infeasible.



If we are in Case 1, we are done with the iteration. However, we need to do more work for Case 2, to figure out the new optimal point (and if there even is one). Here, we will claim that if the new LP is feasible, the new optimal point, which we will call $\mathbf{x}^{**}$ will be on the new constraint line, that is it will be a point $\mathbf{x}^{**}$ such that $\mathbf{a}_m \cdot \mathbf{x}^{**} = b_m$.

Claim: New optimal points will always be on the new constraint

When adding a new constraint $\mathbf{a}_m \cdot \mathbf{x} = b_m$ that moves the optimal point of the feasible region, the new optimal point $\mathbf{x}^{**}$ will never not end up on the new constraint, meaning

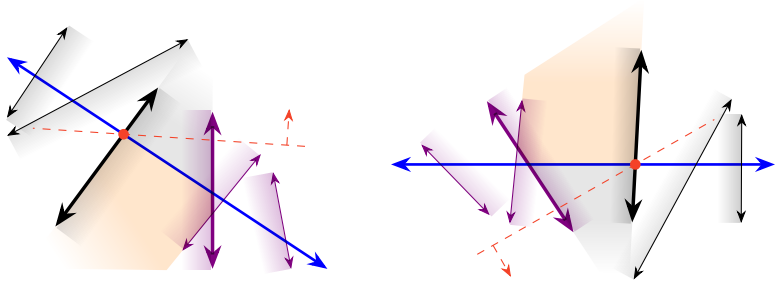
$$\mathbf{a}_m \cdot \mathbf{x}^{**} = b_m.$$

Proof. Assume for the sake of contradiction that the new optimal point $\mathbf{x}^{**}$ is not on the new constraint, meaning it has an objective value higher than any feasible point on the constraint line. Since this point was not optimal before adding our new constraint, it must have a value at most the previous optimal point $\mathbf{x}^{*}$.

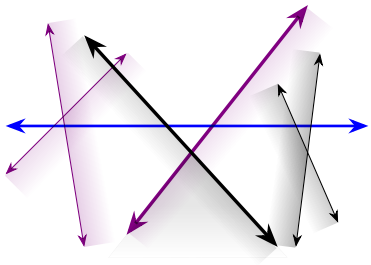
Now consider the line segment from $\mathbf{x}^{**}$ to $\mathbf{x}^{*}$. Since $\mathbf{x}^{**}$ satisfies the newly added constraint but $\mathbf{x}^{*}$ doesn't, there must be a point on this line segment that is on the constraint line. Finally, note that since the objective function is linear, the value of the objective on this line we drew must be monotone nondecreasing. Therefore, the point on the constraint line cannot have a lower value than $\mathbf{x}^{**}$, giving us our desired contradiction. □

By our claim, we know that it suffices to check the points on the new constraint line to find our new optimum, or if none of the points on our line are feasible, to find that the LP is now infeasible. But which points on the constraint line do we need to check? A useful fact for us is that we only need to check the intersections of our new constraint with the previous constraints. This is because it can never be the case that a non-intersection has higher objective value that all of the intersections. Any non-intersection point can be slid along the line in both directions until it hits an intersection. Because of the linearity of our objective, both of these direections *cannot* decrease the value of the objective.

For the rest of this proof, we will call the two directions along the constraint line "left" and "right". For this to make visual sense, you can imagine rotating the entire LP around so that the new constraint is horizontal with its feasible area facing up. An example is provided below, where the right image is rotated as described. Each constraint has a shaded gradient towards its feasible area, and the LP's feasible area before and after the newest constraint is shaded gray and orange respectively.



Now, we can scan through all the other constraints, and for each one, compute its intersection with the new line, and whether the feasible area is to the "right" or to the "left" of the intersection. In our example, the purple constraints would be "right" intersections, as their feasible area in the rotated LP is to the right of the intersection. Similarly, the black constraints would be "left" intersections. We will find the rightmost "right" intersection and the leftmost "left intersection". These are bolded in our example. The area between these two gives us the feasible region for the new optimal point. We can therefore pick which of these two intersections gives a better objective value as the new optimal point (if they give the same value, i.e., the objective is perpendicular to the new constraint, we arbitrarily take the point to the right). Consequently, if these intersections cross with no feasible region, i.e., if the *rightmost* "right" intersection is further right than the *leftmost* "left" intersection, the LP is now infeasible. We can see this happening in the example below, which is already rotated so that the new constraint has its feasible region facing up.



Runtime The worst case time taken per iteratiom is $O(m)$ (when we are in Case 2) since we have $m-1$ constraints to scan through and it takes $O(1)$ time to process each one. Right now, this doesn't look good, since if an iteration takes $O(m)$ time and there are m constraints to iterate over adding, it seems that this algorithm is $O(m^2)$. In fact, in the worst case it is. However, we have a very strong tool to help us with this problem: randomization!

2.2 Improving the algorithm with randomization

Key Idea: Use randomization with incremental algorithms

Adding the constraints one at a time could lead to $O(m^2)$ worst-case time if they happen to always change the objective value. So lets lower the probability of this happening by *randomly shuffling* the constraints and adding them in a random order! This is called a *randomized incremental algorithm.*

So suppose we add the constraints in *a random order*? What is the expected running time of the algorithm? Consider a random permutation $\pi_1, \ldots, \pi_m$ of the constraints, and denote the prefix of the first i constraints in the shuffled order as P_i.

The runtime to insert the i^{th} constraint is $O(1)$ if the optimum does not change, or $O(i)$ if the optimum does change. So let's define an indicator random variable

$$X_i = \begin{cases} 1 & \text{if the optimum given constraints } P_i \text{ is different from } P_{i-1}, \\ 0 & \text{otherwise.} \end{cases}$$

This leads to a running time of

$$T = O\left(\sum_{i=2}^{m}(1 + X_i \cdot i)\right).$$

By linearity of expectation, the expected running time is

$$\mathbb{E}[T] = O\left(m + \sum_{i=2}^{m} i \cdot \mathbb{E}[X_i]\right) = O\left(m + \sum_{i=2}^{m} i \cdot \Pr[X_i = 1]\right).$$

So, what is the probability that the optimum changes when we insert the i^{th} constraint? i.e., what is the probability that we are in Case 2?

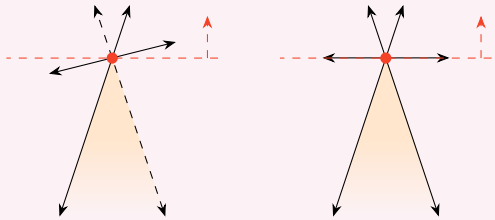
Notice that the optimum (assuming the LP is feasible and bounded) is at a corner of the feasible region, and this corner is defined by two constraints, namely the two sides of the polygon that meet at that point. If both of those two constraints are in P_{i-1}, then the optimum can't change so the probability is zero. So, since we are inserting constraints in a random order, the probability we are in Case 2 when we get to constraint i is at most the probability that the i^{th} constraint is one of these two corner constraints, which is at most $2/i$.

This means that the *expected* runtime of the algorithm is is at most:

$$\mathbb{E}[T] = O\left(m + \sum_{i=2}^{m} i \cdot \Pr[X_i = 1]\right) = O\left(n + \sum_{i=2}^{m} i \cdot \frac{2}{i}\right) = O(m + 2m) = O(m).$$

Remark: More than two constraints in the intersection?

In the proof above, we noted that the optimum is defined by the intersection of two constraints, but what if more than two constraints intersect at that point? Does this hurt our analysis? No, if many constraints intersect at the same point, at most two of them, if removed, would change the answer. Here are two examples to illustrate. The objective direction is up, and the feasible region is shaded.



In the first, three constraints intersect at the optimal point (red), but only one of them (the dashed one) would change the optimum if removed. In the second example, none of them would change the optimum if removed! So the possible probabilities of changing the optimum when removing a random constraint are $2/m$, $1/m$, or 0, which are all at most $2/m$. Extra constraints help us, not hurt us!

2.3 Handling Special Cases, and Extension to Higher Dimensions

Optional content — Not required knowledge for the exams

Handling infeasability What if the LP is infeasible? There are two ways we can analyze this. One is that if the LP is infeasible, then it turns out this is determined by at most 3 constraints. So we get the same as above with $2/m$ replaced by $3/m$. Another way to analyze this is imagine we have a separate account we can use to pay for the event that we get to Case 2 and find that the LP is infeasible. Since that can only happen once in the entire process (once we determine the LP is infeasible, we stop), this just provides an additive $O(m)$ term. To put it another way, if the system is infeasible, then there will be two cases for the final constraint: (a) it was feasible until then, in which case we pay $O(m)$ out of the extra budget (but the above analysis applies to the the (feasible) first $m-1$ constraints), or (b) it was infeasible already in which case we already halted so we pay 0.

Handling unboundedness symbolically What about unboundedness? We had said for simplicity we could put everything inside a bounding box $-M \le x_i \le M$. E.g., if all c_i are positive then the initial $\mathbf{x}^* = (M, \ldots, M)$. In the 2D case we argued that we can find an appropriate M by looking at the bit complexity of the input numbers. In higher dimensions, however, its not clear how to do the same. Instead, we could actually do the calculations viewing M symbolically as a limiting quantity which is arbitrarily large. For example, in 2-dimensions, if $\mathbf{c} = (0,1)$ and we have a constraint like $2x_1 + x_2 \le 8$, then we would see it is not satisfied by (M, M), and hence intersect the constraint with the box and update to $\mathbf{x}^* = (4 - M/2, M)$.

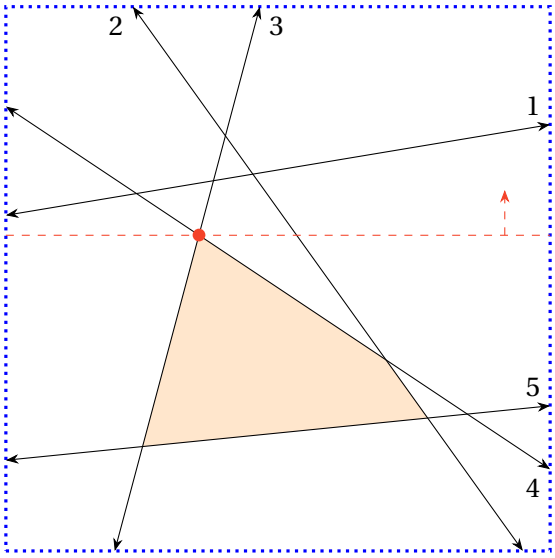
Runtime in higher dimensions So far we have shown that for $d = 2$, the expected running time of the algorithm is $O(m)$. For general values of d, there are two main changes. First, the probability that constraint m enters Case 2 is now d/m rather than $2/m$. Second, we need to compute the time to perform the update in Case 2. Notice, however, that this is a $(d-1)$-dimensional linear programming problem, and so we can use the same algorithm recursively, after we have spent $O(dm)$ time to project each of the $m-1$ constraints onto the $(d-1)$-dimensional hyperplane $\mathbf{a}_m \cdot \mathbf{x} = b_m$. Putting this together we have a recurrence for expected running time:

$$T(d,m) \le T(d,m-1) + O(d) + \frac{d}{m}[O(dm) + T(d-1,m-1)].$$

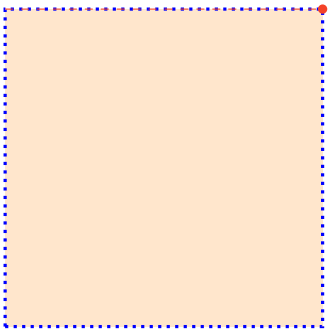
This solves to $T(d,m) = O(d!m)$. Good for very small values of d, but completely impractical otherwise.

A Seidel's Algorithm Example

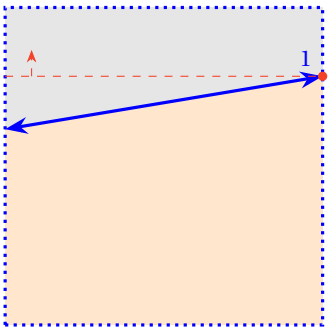
In this section, we will walk through an example application of Seidel's algorithm. Namely, we will be using the following algoirthm, where the order the constraints are added are numbered, the final feasible region is shaded, and the objective direction is up. The starting bounding box is drawn with dotted lines.



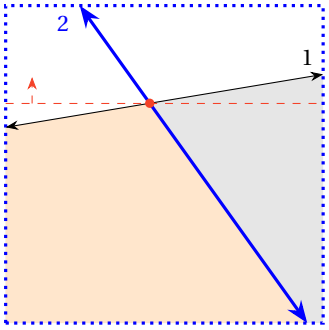
Now, let's run through this example constraint by constraint. Generally, we will make the newly added constraint blue and heavier, as well as making the previous feasible region gray, while the current feasible region is orange.



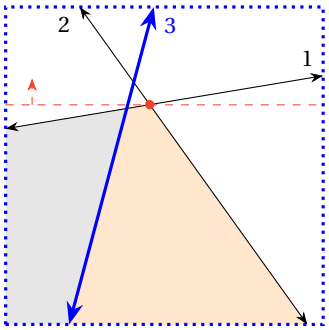
We begin with no constraints added. At this point, the optimal point is at a corner of the bounding box, we will pick the corner (M, M).



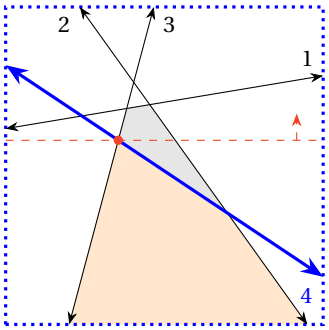
When we add constraint 1, we need to move the optimal point. We check the intersection of the new constraint and all four edges of the bounding box (since those are constraints we've added at the start), and find that the right and left edges give the most constraining intersections, the one of these with higher objective value becomes our new optimal point.



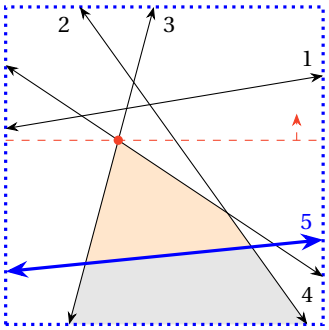
Adding constraint 2, we again need to move the optimal point. This time, we check against the bounding box edges and constraint 1. In relation to constraint 2, the bottom and right edges of the bounding box would be "right" constraints, and the top and left edges of the bounding box as well as constraint 1 would be "left" constraints. Our tightest intersection pair is the bottom of the bounding box and constraint 1, and so we pick the better one as the new optimal point.



Adding constraint 3, we get a smaller feasible region, however we do not need to move the optimal point, since it satisfies the new constraint.



Adding constraint 4, we need to move the constraint yet again. We find that in relation to constraint 4, the bottom and right edges of the bounding box and constraint 2 are "right" constraints, and the top and left edges of the bounding box and constraints 1 and 3 are "left" constraints. Our tightest intersection pair is constraints 2 and 3, and so we pick the better one.
Note that this shows why we need to check against all previous constraints, as constraint 3 had not previously defined an optimal point, but it does now.



Adding constraint 5, we again get a smaller feasible region, while not having to move the optimal point. We have now added all the constraints, and so we see that we have ended up at the correct optimal point.

Exercises: Linear Programming III

Problem 1. Show that if we allowed arbitrary use of absolute value constraints in a linear program, then we would be able to solve NP-Complete problems (Hint: consider constraints like $|x| = 1$.)

This shows that we can not hope to always handle absolute value problems with linear programming (unless $P = NP$), even though we saw some scenarios above where we could.