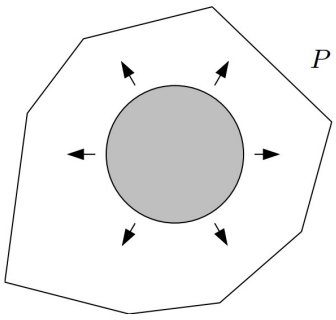
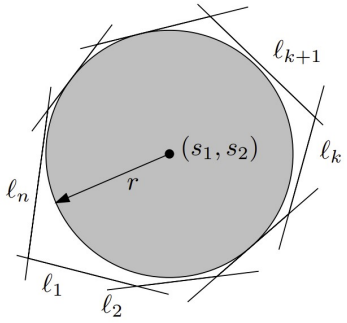
Introduction This project was an introduction to Linear Programming using the Largest Disk in a Convex Polygon problem. We will first introduce the problem, explore my implementation, and finally analyze the runtime of a parallel algorithm.

Largest Disk in a Convex Polygon

Let us call the given convex polygon P, and let us assume that it has n sides. As we said, we want to find the largest circular disk contained in P.



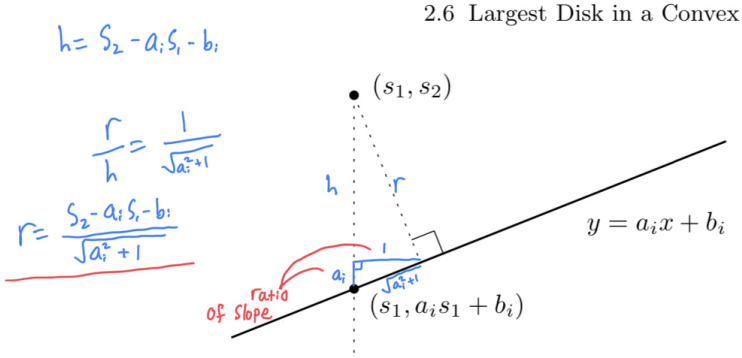
For simplicity let us assume that none of the sides of P is vertical. Let the ith side of P lie on a line ℓ_i with equation $y = a_i x + b_i$, $i = 1, 2, \ldots, n$, and let us choose the numbering of the sides in such a way that the first, second, up to the kth side bound P from below, while the $(k+1)$st through nth side bound it from above.



[1]

[1]Matousek Jiri, amp; Gartner Bernd. (2007). Understanding and using linear programming. Springer.

From this point, we can maximize the length of the radius based on 2 sets of equations: Lines l_1 to l_k bounding r from below and lines l_{k+1} to l_n bounding r from above. We may use the following diagram and similar triangles in order to find distance inequalities for which we will maximize the radius r.



The disk of radius r centered at $\mathbf{s}$ thus lies inside P exactly if the following system of inequalities is satisfied:

$$\frac{s_2 - a_i s_1 - b_i}{\sqrt{a_i^2 + 1}} \geq r, \qquad i = 1, 2, \ldots, k$$
$$\frac{s_2 - a_i s_1 - b_i}{\sqrt{a_i^2 + 1}} \leq -r, \qquad i = k+1, k+2, \ldots, n.$$

Therefore, we want to find the largest r such that there exist s_1 and s_2 so that all the constraints are satisfied. This yields a linear program! (Some might be frightened by the square roots, but these can be computed in advance, since all the a_i are concrete numbers.)

$$\begin{aligned} \text{Maximize} \quad & r \\ \text{subject to} \quad & \frac{s_2 - a_i s_1 - b_i}{\sqrt{a_i^2 + 1}} \geq r \quad \text{for } i = 1, 2, \ldots, k \\ & \frac{s_2 - a_i s_1 - b_i}{\sqrt{a_i^2 + 1}} \leq -r \quad \text{for } i = k+1, k+2, \ldots, n. \end{aligned}$$

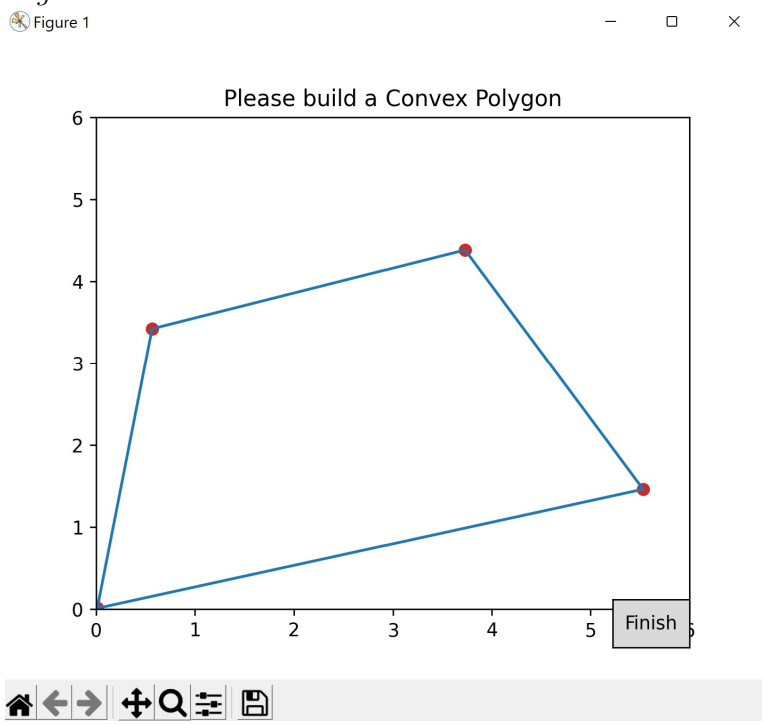
There are three variables: s_1, s_2, and r. An optimal solution yields the desired largest disk contained in P.

In order to code this problem, I will lay out the steps needed:

- Use user input to construct a polygon
- Turn the polygon (x,y) points into line equations of form y = ax + b
- Separate line equations into upper and lower bounds
- turn upper and lower bounds into distance inequality equations
- combine those equations and plug into linear programmer
- output final circle

Constructing the polygon: I used matplotlib to record user clicks into a set of (x,y) inputs. 2 consecutive inputs will plot a line, begining with a point set at the origin and ending with the same point. Users press the "Finish" button in order to place this last line.
As shown in Figure 1, the first user input was the upper leftmost red dot, and a line was constructed between the origin and that point. The last user input was the bottom right most point and the "Finish" button creates a line from this point to the origin.

Figure 1



Line equations: At this point I turned the (x,y) input points into line equations by converting double point line equations into the form $y = a_i x + b_i$. I stored this data through a $nx2$ sized matrix with the first column holding a_i values and the second column holding b_i values.

Upper and Lower Bounds: At this point, I had to determine which line equations bounded the circle from above and below. This was a bit of a difficult task to do simply so I used a trick in order to simplify. However, it excludes a subset of polygons from the acceptable input. I defined the first lower bound line equation, l_k, as the first positively sloped line equation after a negatively sloped line equation. This parameter fails for polygons whose lines all have negative slopes. However, it works for the vast majority of inputs (see if you can make the input fail) [2].

Linear Programming Inequalities: Now that we have our two sets of line equations separated, we can put them into the two sets of inequalities used to constrain the optimization. *NOTE* The linear programmer I used only accepts minimization problems, so we multiplied the distance equations by -1 in order to make this a minimization problem.
We can write our inequalities of the form:
Note: $a^{'} = \sqrt{a_i^2 + 1}$

For the upper bounds: $k + 1 \leq i \leq n$

$$r - \frac{a_i s_1}{a^{'}} + \frac{s_2}{a^{'}} \leq \frac{b_i}{a^{'}}$$

For lower bounds: $1 \leq i \leq k$

$$r + \frac{a_i s_1}{a^{'}} - \frac{s_2}{a^{'}} \leq -\frac{b_i}{a^{'}}$$

Additionally, we are optimizing for 3 variables as:

$$\text{Minimize} - 1 * r + 0 * s_1 + 0 * s_2$$

[2] Diamond like shapes fail the input. Since first data point is predetermined as the origin (0,0), any diamond shape created in the first quadrant will have 4 lines all with negative slopes

Matricies as Linear Programming inputs: At this point we have our data to optimize over, but we must put it into a specific matrix form to be accepted by the linear programmer. First we set our objective function, our goal is simply to minimize the value of:

$$c^T x$$

In this specific problem, we can represent these vectors as:

$$c = [-1, 0, 0] \text{ and } x = [r, s_1, s_2]$$

Next we must define the optimization constraints. We will combine the lower and upper bounded sets of inequalities into a single set of linear inequalities of the form:

$$Ax \leq b$$

$A \in n\text{x}m$ $x \in \mathbb{R}^{\text{m}}$ and $b \in \mathbb{R}^{\text{n}}$ Where $m = 3$ since were optimizing for 3 variables and n is the number of (x,y) input coordinates that construct the convex polygon. For example, a singular row of A (as an upper bounded equation) would be of the form:

$$[1, -\frac{a_i}{a'}, \frac{1}{a'}]$$

with the vector x:

$$x = [r, s_1, s_2]$$

and the coresponding row of vector B would be of the form:

$$[\frac{b_i}{a'}]$$

combined, we have the original inequality defined in the paragraph before: *for upper bounding line equations*

$$r - \frac{a_i s_1}{a'} + \frac{s_2}{a'} \leq \frac{b_i}{a'}$$

We will also define some bounds for our variables to avoid invalid conclusions.

$$0 \leq r < \infty$$

$$0 \leq s_1 \leq 6$$

$$0 \leq s_2 \leq 6$$

The coordinate bounds are constructed from the coordinate axis I defined and were mostly there for debugging purposes.

The Linear Programmer: Interior-Point-Method and Karmarkar's Projective Scaling Algorithm

Since our data was put into standard form, any type of Linear Programming algorithm can be used to solve this problem. I used SciPy's linprog from their optimize library. SciPy does not provide in depth explanation of their specific implementation, they only say which general algorithm they used. For this program, they used an Interior-Point-Method algorithm. There are many interior point linear programming algorithms, so after doing some research, I'd like to explain my favorite one.

Karmarkar's Algorithm Background: this algorithm is a bit famous in the LP world as it was the first feasably implementable polynomial time algorithm. Up until this point, the simplex method was the best known approach to LP problems, with an unfortunate exponential worst case. Before Karmarkar's, the polynomial time Ellipsoid Method was discovered with great excitement, but in practice dissapointed the world of LP solvers due to the large coeficient in its big O complexity. It proved less efficient than the simplex method even with the simplex methods exponential time worst case. But in the fall of 1984, Karmarkar's algorithm out of Bell Labritories made wake in the LP world due to its polynomial complexity and real world feasability. [3] It was the first algorithm to beat the simplex method in many real world applications. Each algorithm takes a vastly different approach to solving LP problems, however, both turn the system of linear inequalities into a convex hull which represents the solution space. The simplex method utalizes the the perimeter of this convex hull and traverses along this edge until an optimal value is reached. With a large solution space, this path can be very long. In light of reducing the path traveled to the optimal solution, Karmarkar's algorithm traverses instead within the interior of the convex hull solution space. It was the first Interior Point Method. The algorithm reduces the path length taken by a clever method of traversing inside the space towards the optimal solution. The algorithm combines gradient descent to move in the direction of "steepest descent" and Linear Transforms in order to maximize the possible movement at each iteration. As I'm taking Calc 3 and Linear Algebra this semester, it was very cool to see how this algorithm combines the two. Realizing that the combination of these two concepts are the engine behind so many optimization tequniques, namely machine learning, has been a extremely stimulating endeavor. This field will probably be a large concern throughout my education. But enough about me...

[3] https://www.ise.ncsu.edu/fuzzy-neural/wp-content/uploads/sites/9/2021/07/LPchapter-6.pdf

The Algorithm: Karmarkar's Projective Scalling Algorithm.
minimize

$$c^T x$$

subject to

$$Ax \leq b$$

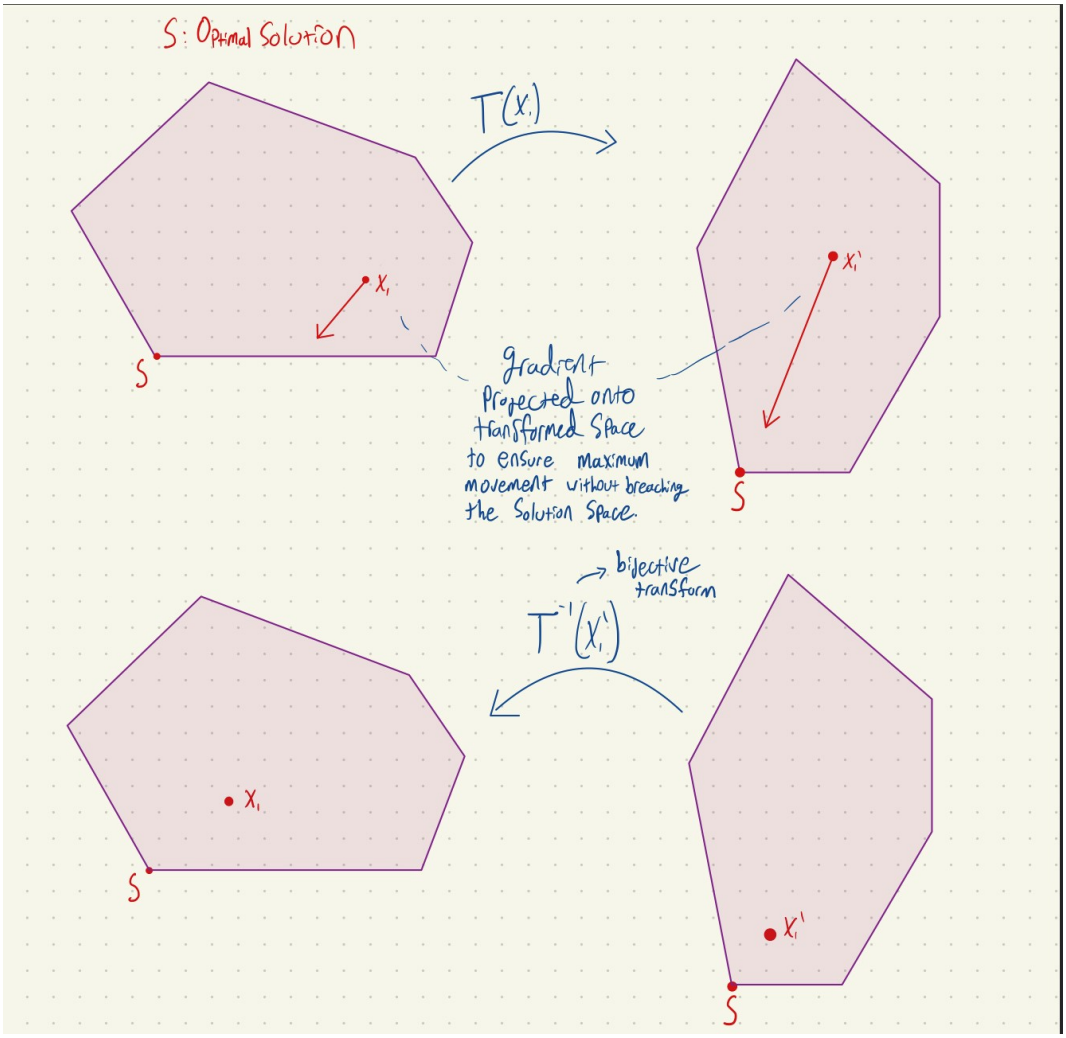
As I stated above, the meat of Karmarkar's algorithm is finding a clever way to optimally traverse the interior of the solution space.
In general, the algorithm is an iterative process. For our current interior point, we transform the solution space so that this point can take the largest possible movement in the direction of the solution. This is achieved by moving in the direction of the gradient of the objective function. This movement is transformed back into the real solution space so that we can generate a new point closer to our optimal solution. The iteration is performed again on this new interior point. This algorithm runs until the minimum solution is reached.

Karmarkar's algorithm is based on two key insights:

- If the current interior point is near the center of the polytope, then moving in the direction of stepest descent of the objective function will approach a minimum value (remember this is a minimization problem).
- An appropreate linear transform can be applied to the solution space such that the current interior point is placed near the center in the transformed space.

Now that we have these insights, lets get a little more specific. The algorithm starts at some known interior point within the solution space.

Figure 2 outlines the following process:
In each iteration we perform a bijective linear transfrom (it is bijective since we will will need to map back to our real solution space) that puts our current interior point at the center (or as close as possible) of the transformed solution space. At this point, we will project the gradient of the objective function onto this new space in order to move the greatest distance towards the optimal solution without breaching the feasable region. We will use the inverse of our original transform in order to plot this movement in our real solution space. We will start the process again with this new interior point. We will iterate until a more optimal solution cannot be reached (the optimal solution is found).
Figure 2



I will not go into the proof of this algorithm as it was more than 2 pages long. However, I will state it's conclusion. With n the number of variables to optimize for, and L the number of bits of input to the algorithm. The algorithm requres $\mathrm{O}(nL)$ iterations to terminate and $\mathrm{O}(n^{2.5} * L * logL * loglogL)$ where $\mathrm{O}(L * logL * loglogL)$ is the linear algebra matrix multiplication required on L input bits. Thus the total complexity $\mathrm{O}(n^{3.5} * L^2 * logL * loglogL)$.