Ham Sandwich Cuts

Math 480b SAGE

Andy Barr, Maddie Romansic, Lauren Bobel, Chloe Huber, Matthew Junge

March 12th, 2010

Abstract

In eighteenth century England John Montagu, the fourth Earl of Sandwich, forever changed the course of humanity when he developed a way to simultaneously satiate his ravenous appetite, play cards with his chums and keep his hands clean. No one knows how the epiphany dawned upon the Earl, however most historians agree it was probably divine guidance. Inspiration aside, the moment Montagu requested his valet bring him meat tucked between two pieces of bread, the scope and direction of mathematics would be irreversibly altered.

Ever the egalitarian, Montagu became obsessed with being able to share his sandwiches without bias between him and his friends. From this flurry of creativity and scientific curiosity arose the following question:

> *Given a sandwich consisting of bread, cheese and ham constructed in any manner, does there exist a cut that will divide all three ingredients equally between slices?*

It was not until Borsuk-Ulam came along and proved a theorem whose corollary answered the Earl's question with a resounding, "Yes!". The statement of the Ham Sandwich Theorem is as follows:

Theorem 0.1 (Ham Sandwich)**.** *If $A_1, \ldots, A_{n+1}$ are bounded measurable sets in $\mathbb{R}^{n+1}$, there exists an n-plane in $\mathbb{R}^{n+1}$ that bisects each of them simultaneously.*

We see that the $\mathbb{R}^3$ case of the theorem completely answers the Earl's question. However, it does not yield any insight into how one finds such a cut. Our group set out to answer a special case of the theorem. Namely, we will address the following theorem.

Theorem 0.2 ($\mathbb{R}^2$ Polygon Case)**.** *Given two polygons in $\mathbb{R}^2$ there exists a line that simultaneously bisects the area of both.*

To answer this question we have developed a sage interact that takes as input vertices of two polygons and an error bound. With these it plots the polygons and uses a binary search to generate a line that cuts the area of the two polygons to fit within the error bound.

Contents

1 Proving the Theorem **4**

2 Coding out the Code **5**
2.1 Three Teams are Better than One 5

3 The Teams/Tasks **6**
3.1 P-Team 6
3.1.1 Building Polygons 6
3.1.2 Let There be Line 7
3.2 The L-Team Experience 8
3.2.1 Gearing Up 8
3.2.2 Translate and Rotate 8
3.2.3 The Binary Search 9
3.2.4 Epiphany 9
3.2.5 To Class Conversion 9
3.2.6 Fine Tuning: Error Bound Implementation 10
3.3 A-Team 10
3.3.1 The Task 11
3.3.2 Classes versus Functions 11
3.3.3 Lines - How to write them? 11
3.3.4 Reverse Entropy Comes to Fruition 12

4 Conclusion **12**

5 Appendix **13**

1 Proving the Theorem

Though not the most glamorous part of our project, we feel it important to give a little exposition pertaining to pure mathematics of the theorem. First we state the Borsuk-Ulam Theorem:

Theorem 1.1 (Borsuk-Ulam)**.** *Let $S^2 \subset \mathbb{R}^3$ denote the unit sphere. Given a continuous map $f : S^2 \to \mathbb{R}^2$, there is a point of $x \in S^2$ such that $f(x) = f(-x)$.*

The proof of the theorem relies on algebraic topology pertaining to antipode preserving maps and homotopy theory. We will omit these details and assume that Borsuk-Ulam holds. With this, we offer a proof of the two dimensional polygon case of the Ham Sandwich Theorem:

Theorem 1.2. *Given two polygons in $\mathbb{R}^2$ there exists a line that simultaneously bisects the area of both.*

Proof. We take two bounded polygonal regions A_1 and A_2 in the plane $\mathbb{R}^2 \times 1$ contained in $\mathbb{R}^3$, we wish to show there exists a line L in this plane that bisects the area of both polygons.

Given a point $u \in S^2$, let us consider the plane P in $\mathbb{R}^3$ that passes through the origin that has u as its unit normal vector. This plane divides $\mathbb{R}^3$ into two half-spaces; let $f_i(u)$ equal the area of that portion of A_i that lies on the same side of P as does the vector u.

If u is the unit vector $\mathbf{k} =< 0, 0, 1 >$, then $f_i(u) = \text{area}(A_i)$ and if $u = -\mathbf{k}$ then $f_i(u) = 0$. This is true since our polygons lie in the plane $\mathbb{R}^2 \times 1$ determined by $\mathbf{k}$. Thus, if $u \neq k$ and $u \neq -k$ we have the plane P intersects the plane $\mathbb{R}^2 \times 1$ in a line L that splits $\mathbb{R}^2 \times 1$ into two half planes, and $f_i(u)$ is the area of that part of A_i that lies on one side of this line.

Replacing u by $-u$ gives us the same plane P, but the other half space, so that $f_i(-u)$ is the area of that part of A_i that lies on the other side of P from u. This gives us the following:

$$f_i(u) + f_i(-u) = \text{area}(A_i)$$

That is to say we piece together the two halves of A_i by taking both $f_i(u)$ and $f_i(-u)$.

Now consider the map $F : S^2 \to \mathbb{R}^2$ given by

$$F(u) = (f_1(u), f_2(u))$$

The Borsuk-Theorem gives us a point $u \in S^2$ for which $F(u) = F(-u)$. This implies that $f_i(u) = f_i(-u)$ for $i = 1, 2$, Hence $f_i(u) = \frac{1}{2}A_i$ and we have bisected both polygons as desired. □

2 Coding out the Code

Unfortunately the previous theorem is only a result that proves existence of such a line L. Though the result is very pretty and somewhat surprising, the Earl of Sandwich would probably be left unsatisfied. For although we now know such a cut exists, we are still left scratching our heads with no information about how to make such a cut. The great Earl no doubt spent hours upon hours, days upon days, years upon years holed up in his kitchen working his chefs to the bone and tediously implementing the most basic of utensil algorithms to find his cut. Fortunately for us, the advent of the computer allows our team to make more cuts than our friend Montagu could ever dream of making.

Our project very naturally separated into three subgoals:

1. Draw and find the areas of two polygons and be able to find the areas of the sub-polygons generated by an intersecting line.
2. Implement a line-finding algorithm to approximately bisect our polygons.
3. Create a nice interact in Sage that showcases our work and makes the Earl proud.

2.1 Three Teams are Better than One

To attack these areas of our code we decided to split our group of five into two task force 'teams':

P-Team who would deal with the **p**olygons and the **p**oints obtained from intersecting lines with said polygons. This team consists of Maddie, Lauren and Matt.

L-Team would deal with the **l**ines, in particular the area-bisecting line. This team is Andy and Chloe.

Our construction of these two tasks mostly proved to be disjoint pieces of code in which L-Team could pass their arguments into P-Team's area and polygon generating functions. However, there was some work in the gluing together of code which transitions into the final team in our group.

A-Team (Short for All-Team and included all five members.) A-Teams job was to make everything work and implement the interact.

3 The Teams/Tasks

The natural partitioning of our project led to an equally natural partitioning of our paper. We will divide the exposition on our code into three sections, each corresponding to one of the teams in our group.

3.1 P-Team

P-Team essentially had two tasks. First, we needed a nice way to interact with polygons, i.e. plotting their vertices, finding their area and computing their centroids. Second, we had to make our polygons interact well with lines so we could easily implement the cutting.

Let us first turn out attention to the construction of our polygons and the computing of their essential information.

3.1.1 Building Polygons

Drawing polygons turned out to be a simple task thanks to the polygon class in sage. This allows the user to input vertices (in counterclockwise order) and sage will plot the polygon. Unfortunately, the class does not contain area or centroid functions. So, we had to write our own. Luckily the area of a polygon with vertices $P = [(x_0, y_{,0}), (x_1, y_1), \ldots, (x_n, y_n)]$ is well known and we were able to write a function that utilizes the formula:

$$A(P) = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$$

Additionally, we had some interest in locating the centroid since that seemed like a good starting point for generating a bisecting line. The centroid formula is:

$$Centroid(P) = (C_x, C_y)$$

$$C_x(P) = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$C_y(P) = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

With the above formulas coded and a nice way to draw polygons we were able to generate an interact that took as input vertices that plots two polygons and the line through their centroids. While providing the areas of each.

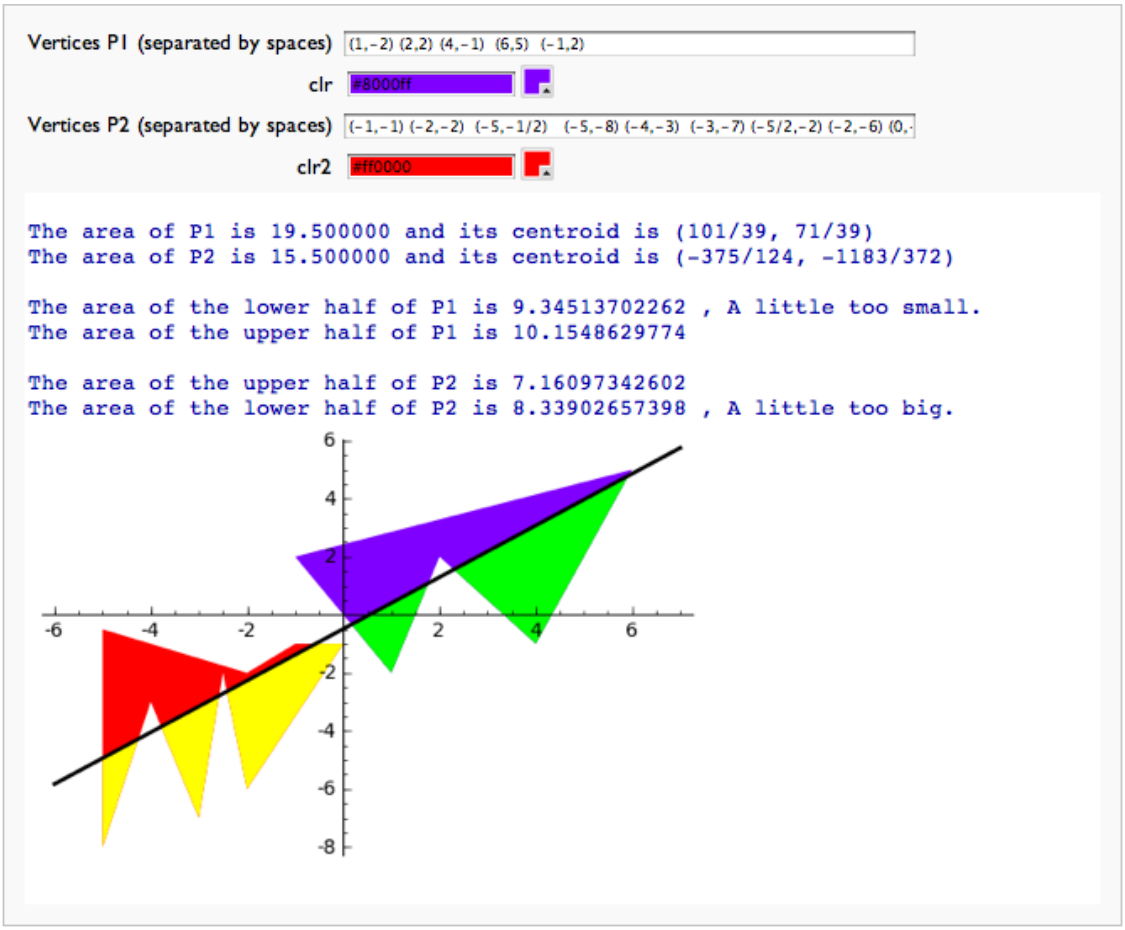
This still left us with the task of generating four new polygons from an intersecting line.

3.1.2 Let There be Line

It proved somewhat difficult to figure out how to find the lower and upper polygons that result from intersecting a polygon with our dividing line. The method that we finally decided on was to write the line in the form $ax + by = c$ and represent each edge of our polygon as a line segment. Thus, we need only calculate whether the line intersects each edge and place all of the points below the intersection into a list creating the lower polygon. To do this we utilized matrix.rightsolve in sage and implemented a careful tracking loop to keep our new vertices in order.

The end result of P-Team's labors is the following interact:

Figure 1: **The Pride and Joy of P-Team**



Notice that we have input boxes for the vertices. And, upon entering two poly-

gons, we deliver the area and centroid of each as well as the dividing line that intersects both centroids. With the line in place we calculate the lower polygons of our two major polygons and display them in a different color. Lastly, we print the area and run a simple output of "too big" or "too small" which decides if the polygon has been bisected or not.

3.2 The L-Team Experience

Like the river winds through the forest floor only to emerge, flowing with an inner grace and powerful beauty, in the bosom of the ocean, our coding objectives wound through the floors of our hearts, twisting and turning, everchanging, only to break out of the constraints of our souls and into the sage notebook as a functioning line generating algorithm.

The point of the preceding poorly punctuated run-on sentence poetry is to illustrate that our process for finding a bisecting line was not mechanical and anything but direct. Below we give an outline of our thought process in making the hamline.

3.2.1 Gearing Up

It was L Team's job to deal with code that would eventually generate a line approximately bisecting the area of any two polygons. Our thought was that we would begin with the line passing through the centroids of the two polygons, and then by comparing the relative areas of the four new polygons created by the intersection of the two original polygons and that line, make a series of informed translations and rotations of the intersecting line: the "hamline".

This prompted the creation of a hamline class, a class of line which could be initialized as a line passing through the two centroids of our polygons, and represented by a 3-tuple (a, b, c) representative of the equation of a line $ax + by = c$. We originally had a $y = mx + b$ hamline which freaked out when our lines were vertical. Handling vertical and horizontal lines would always be hassle.

Hamline would also know other useful information about itself, including its normal and later how to translate and rotate.

3.2.2 Translate and Rotate

The original plan was to compare the areas of the four polygons in our plane, and make the decision to:

1. translate or rotate
2. in which direction, i.e. positive vs. negative, clockwise or counterclockwise
3. to what magnitude

To what magnitude to translate hamline along it's normal, and to what degree to rotate hamline about its "pivot point" - the so-called point equidistant from the two centroids - would be a calculation based on the ratio of the four areas. However, half way through the formation of this calculation, Andy had an epiphany...

3.2.3 The Binary Search

Instead of comparing the areas of four polygons to make an informed translation or rotation, we would use a binary search inspired algorithm until our translate or rotate was within a certain margin of error or within a certain number of steps. At this stage, we were mainly concerned with our rotate method. Rotate would turn hamline into a vector, and place it normal to the original line. Then, it would make the decision to either rotate by progressively half of the angle it had rotated before in either a clockwise or counterclockwise direction.

Translate would go through a similar process: beginning with the line parallel to hamline and passing through either the largest or smallest value y-coordinate vertex, then translating along the normal.

However, we soon became bogged down in how to finish execution and how to alternate effectively between translate and rotate. If we did not set a fixed number of steps (number of rotations or number of translations) and let binary search continue until our hamline was effectively dividing the polygons within a certain margin or error our search might never end (since during the course of one translation or rotation it might be impossible to get close enough to the ideal line). But if a number of steps is set arbitrarily, it is possible that translate or rotate might leave us with a less accurate hamline. We also started to question our choice of pivot point, and whether it should be fixed or dynamic.

3.2.4 Epiphany

After all the thought put into the translate and rotate methods, ultimately we decided the best way to find a line bisecting two polygons was to effectively draw a rectangle of best fit around the polygons and draw lines from one point on the perimeter of the rectangle to another point on the perimeter.

This approximation is not very nice, however, if the dimensions of the rectangle are very different from each other, since the number of points taken from the 2 pairs of sides is the same.

3.2.5 To Class Conversion

We had so many methods manipulating lines and polygons that it made more sense that the lines and polygons knew how to store and give information about themselves. Making a hamline class and polygon class of our own also solved serious scope and hierarchical issues within the code. It also kept our information organized and

retrievable. It made flow control possible to keep track of in a reasonable manner. Before, our code was a mess, but now we have 3 classes and a main method that takes less than 30 lines. Yay!

3.2.6 Fine Tuning: Error Bound Implementation

The error bound was created because in order to achieve machine precision we would have had to created a bunch of lines. In order to put an upper bound on our calculations, we decided to allow the user to input an error bound. We used this error to determine the size of our steps between our two points. See, we partitioned the line segment between the maxima and minima of our x and y and created lines with those steps based on approaching the area with a tolerance given by the error bound.

We also use the error bound to finally decide upon which hamline to accept. Let $A(P$ represent the area function on a polygon, P_1 and P_2 designate our starting polygons and BP_1 and BP_2 designate the lower polygons formed by intersecting a line and let $\epsilon \in (0, .5)$ denote our error bound. We implemented the following requirement:

$$\left|\frac{A(BP_1)}{A(P_1)} - \frac{1}{2}\right| < \epsilon$$

and

$$\left|\frac{A(BP_2)}{A(P_2)} - \frac{1}{2}\right| < \epsilon$$

This way we require that both polygons be very nearly sliced in half!

3.3 A-Team

We begin with a quote from an anonymous, but very wise computer scientist master:

> "No one knows how code comes together - It could be intense labor, it could be careful thought and mechanical resolve or it could be a violation of the chaos law of the universe and a stunning example of reverse entropy. Indeed, no one knows and no one could know... but it probably is reverse entropy."

Sure, L-Team and P-team were great ideas and each squad had a lot of fun coding out their specific code, but when it came down to putting these two seemingly disjoint tasks together, the real work started.

Since we do not want to write an encyclopedia length paper we will only include *some* of the major issues we encountered in compiling the code and describe our ultimate success.

3.3.1 The Task

To reiterate, we had two teams with two nicely working spots of code. P-Team could plot polygons and given a line calculate the upper and lower area of the intersection. L-Team could, given the areas of the upper and lower polygons and an error bound, run a binary search that found a line satisfying the two desired accuracy.

So, our only task was to get these two chunks of code to talk to each other and get along well. Unfortunately, we were overconfident in how disjoint the code was and ended up having several communication errors that compounded upon themselves. Here we go...

3.3.2 Classes versus Functions

The first major issue was that L-Team wrote all of their code inside of classes while P-Team was entirely function based. Upon learning the ignorance of their ways, P-Team realized that classes were far superior. However, no one on P-Team knew how to write class code (all three members of P-Team were very novice programmers). This resulted in many glitches and bugs in the transition. The most traumatizing of which was an indentation error in the code that determined intersects. So, although P-Team's original function code ran beautifully and robustly, the class code was wrought with errors.

3.3.3 Lines - How to write them?

Vertical lines, standard lines, slope-intercept lines and horizontal lines... So many different ways to write lines.

P-Team had an epiphany in their code for solving intersects with polygon vertices by using matrices and thus representing lines in the form $[a, b, c]$ for the standard equation $ax + by = c$. Unfortunately, L-Team decided to call all of their lines as tuples, (a, b, c) in the other standard form $ay + bx = c$. The tuples did not prove to be a major problem, but notice that each team had their own idea of what the standard form of a line is:

$$\text{P-Team:} \qquad ax + by = c$$

and

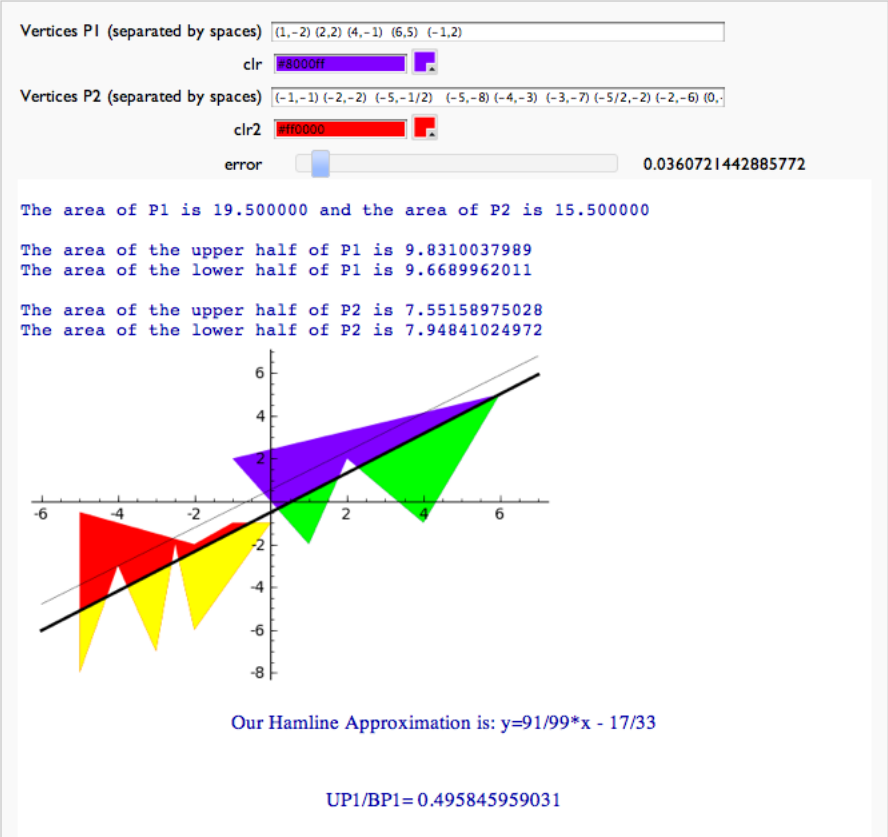
$$\text{L-Team:} \qquad ay + bx = c$$

This error went uncaught for some time in the A-Team code and resulted in many wonky lines and compiling crashes. Fortunately the fix was simple and once we caught this we just had to address several million typos and robustness problems. Still, in the end we finally had something that worked.

3.3.4 Reverse Entropy Comes to Fruition

Pictured below is the fruit of A-Team's Labor:

Figure 2: **The Pride and Joy of A-Team**



4 Conclusion

The Borsuk-Ulam theorem gives us the breathtaking corollary that we can bisect any quantity in our plane, which means there is a plethora of exciting side projects to be implemented - a line exists such that the perimeter and the number of vertices could also be evenly split on either side of the line. We could even venture into the realm of non-polygons or 3 dimensions.

One important lesson we take with us as the quarter comes to a close is that object-oriented is the way to go, if code is to be split up amongst group members more communication is necessary during development and debugging is an agonizing painful process.

It might be nice, in the future, to streamline our code to make finding hamline faster, to make our code more robust as to withstand much smaller error, and perhaps to revive our rotate and translate methods to find a more sophisticated method of bisection. Another useful feature might track the computational expense vs. the given error or send information regarding the number of steps and accuracy of the resulting line.

Though our dear friend John Montagu, the Earl, is no longer with us, his legacy lives on. Moreover, we hope that the work we have done will spark a resurgence of interest in the quest to equitably divide sandwiches.

5 Appendix

Here are some super neat polygons Maddie made that would be a shame to not include:

Figure 3: **Toast**

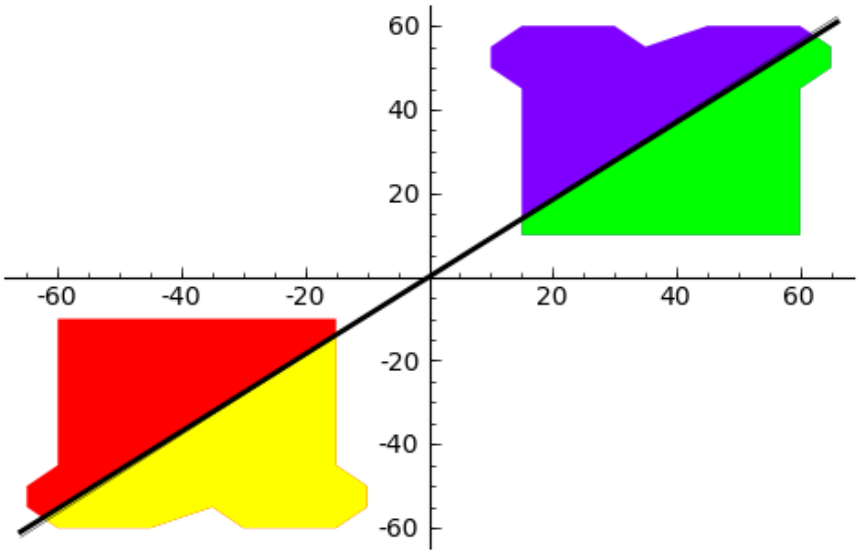


Figure 4: **HAMBONE**

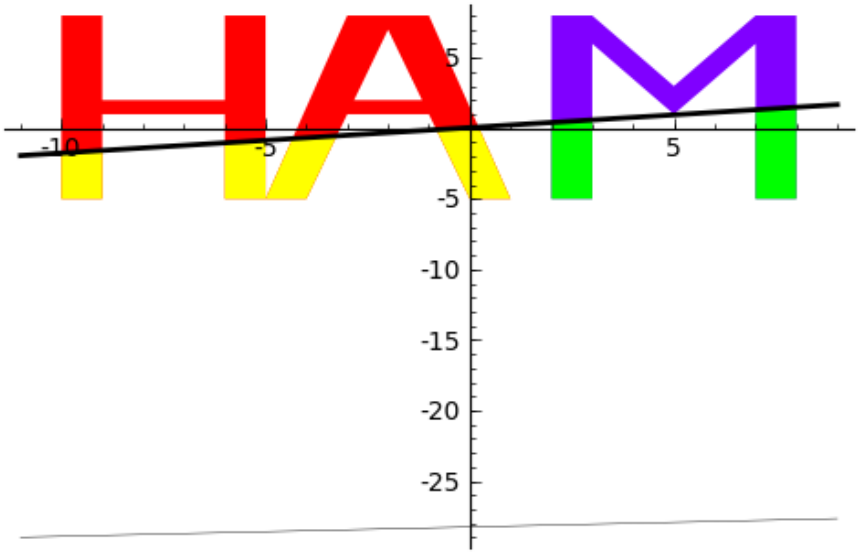


Figure 5: **DINOLOVE**