A Mathematical Approach to Auto-Aim

FRC Team VorTX 3735

Cole M. Huffine, Klein High School, cole.huffine@gmail.com
Chirag Tripathi, Klein High School, chirag.tripathi@outlook.com
Ethan Lee, Klein High School, elee012345 on GitHub

Abstract

VorTX's 2024 FRC season robot, SoundByte, is designed to be able to shoot game pieces from around the field. With our shooter being on an articulating arm, the robot is capable of shooting from farther away, although manually aiming the arm during a game by hand would turn out to be inconsistent. As a workaround to this problem, we created an auto-aim function utilizing kinematics, Newton's Method, trigonometry, and some calculus. This paper presents an overview of the iterative design process, from conceptualization to implementation. We started with sitting down and establishing the problem, then went through the math on paper to get an idea of what to do, and then created code based on the math. We took a robust approach to integrating the code by creating a to-do list to work through along with a visual simulation of our code before deploying it to the physical bot. The design process laid out in this paper prioritizes understandable and reliable code while having a straightforward and easy-to-follow process for other teams to replicate.

© 2024 VorTX. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC-BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.
See http://creativecommons.org/licenses/by/4.0/.
OPEN ACCESS

1. Contextualization

To start, we need to establish the problem, and our approach to the solution. Our current problem is that aiming requires precise angles which cannot be achieved using manual control. Ideally, the solution would be an "auto-aim" function which takes our Robot's position and calculates the required angle of the arm to successfully shoot into the goal.

Before diving into the code, there are a few hardware constraints to be considered. First would be the method of getting the distance from the robot to the target. In our robot, we use a LimeLight, a vision-based camera system designed for FRC robots. The LimeLight is able to accurately track the distance from the robot to an AprilTag on the target. It is also very much possible to use drivebase telemetry to track the position of the robot throughout the field, but, however, the complexity of the calculations would increase dramatically and would also bring a sacrifice to accuracy as it depends heavily on the starting position of the robot. Along with that, it's important to consider the processing power of the robot, and having the latest RoboRio is helpful in ensuring the calculations don't bog down the computer.

2. The Fundamental Aiming Calculations

Our Fundamental Calculations began with some inspiration from a YouTube video created by b2studios.[1] We highly recommend watching the video in order to have an understanding of the calculations. The video explains the physics of aiming, and there is an even more in-depth paper that details the calculations, however, for our purposes in this paper, we will just focus on a few formulas. We can start with this basic equation of motion.

$$x = x_0 + vt + \frac{1}{2}at^2$$

"This expression describes displacement over time given initial position (*x_0*), velocity (*v*) and acceleration (*a*). We use this to describe the motion of our target relative to our shooter."[2] This can be extended to 2 dimensions by simply applying the equation separately to 2 axes:

$$x = x_0 + v_x t + \frac{1}{2}a_x t^2$$

$$y = y_0 + v_y t + \frac{1}{2}a_y t^2$$

All position (*p*), velocity (*v*), and acceleration (*a*) here are from the frame of reference of the target.

*On a side note, we won't be encorporating the z-axis into our calculations as it would be much easier to implement a separate method which say, tracks an april tag to align the robot to face the speaker.

Continuing on, we can utilize Pythagorean's Theorem to create a 2D vector containing *x* and *y* components.

$$|(x, y)| = \sqrt{x^2 + y^2}$$

$$|(x, y)| = \sqrt{(x_0 + v_x t + \tfrac{1}{2} a_x t^2)^2 + (y_0 + v_y t + \tfrac{1}{2} a_y t^2)^2}$$

[1] YouTube Video by b2studios https://www.youtube.com/watch?v=aKd32I0uwAQ

[2] b2studios White Paper https://docs.google.com/document/d/1TKhiXzLMHVjDPX3a-3U0uMvaiW1jWQWUmYpICjIDeMSA/

Now let's propose that our shot can be represented by an explosion, where the outer curves of the explosion represent the possible trajectories of the object being shot. Using the distance-speed-time formula, we can find the distance from the origin to any point on the outer shell of the explosion at a certain time *t*. We can utilize this formula for our fundamental aiming, however, it does not incorporate important variables such as velocity and acceleration. We can solve that by setting it equal to the equation from earlier.

$$d = st$$

$$\sqrt{(x_0 + v_x t + \frac{1}{2} a_x t^2)^2 + (y_0 + v_y t + \frac{1}{2} a_y t^2)^2} = st$$

We can then square it and algebraically expand it out giving us our final equation.

$$(x_0 + v_x t + \frac{1}{2} a_x t^2)^2 + (y_0 + v_y t + \frac{1}{2} a_y t^2)^2 = s^2 t^2$$

$$\frac{a_x^2 \cdot a_y^2}{4} t^4 + (a_x \cdot v_x + a_y \cdot v_y) t^3 + (v_x^2 + x_0 \cdot a_x + v_y^2 + y_0 \cdot a_y - s^2) t^2 + 2(x_0 \cdot v_x + y_0 \cdot v_y) t + (p_x^2 + p_y^2) = 0$$

This is our final auto-aim function. This equation may look daunting, but we have values for all variables except for time, which can be solved. Our goal is to find the solutions to this equation, which we can then plug into another formula later on. Since most of the variables are numbers we know, to make it look less daunting, we can rewrite it as coefficents for each power of *t*, and then solve for them to make the equation more readable.

$$t^4 \Rightarrow \frac{a_x^2 \cdot a_y^2}{4}$$

$$t^3 \Rightarrow a_x \cdot v_x + a_y \cdot v_y$$

$$t^2 \Rightarrow v_x^2 + x_0 \cdot a_x + v_y^2 + y_0 \cdot a_y - s^2$$

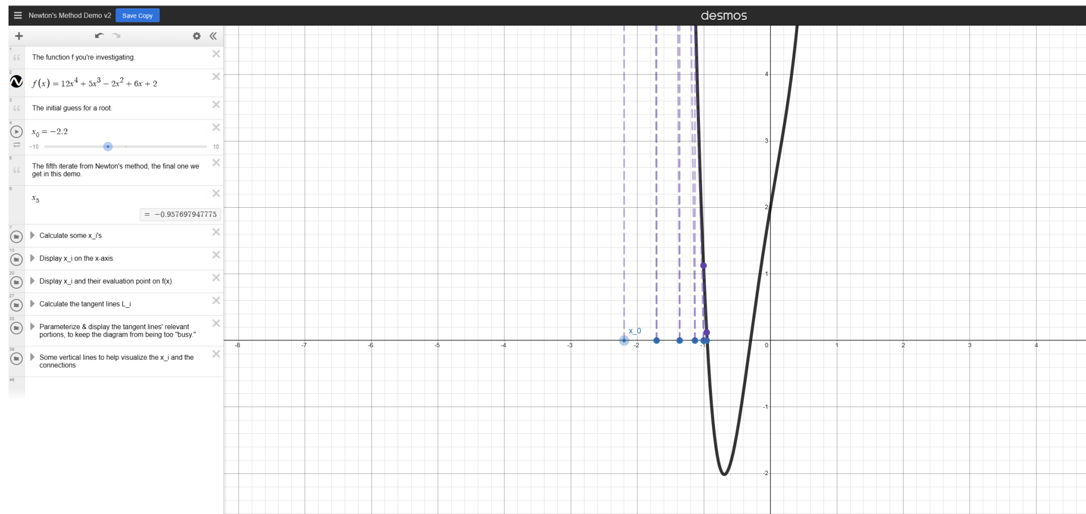
$$t^1 \Rightarrow 2(x_0 \cdot v_x + y_0 \cdot v_y)$$

$$t^0 \Rightarrow (p_x^2 + p_y^2)$$

Now we need to find the solutions to this equation. It may seem simple on paper, but it's a bit more complicated, and was one of the main challenges we ran into when working through this. This quartic equation is in terms of *t* and has 4 solutions. Considering our function is in terms of time, we can rule out all negative solutions as time cannot be negative. Along with that, there is the possibility for multiple positive solutions. In our case (the 2024 FRC season), we want the smallest positive solution as that will provide the shot which reaches the target the fastest and provide the trajectory we need. In some cases, you wouldn't want the smallest possible solution and it depends on the desired goal. For example, if the goal target is on the top of an object, you'd want a trajectory that has a longer air time whose path follows into the air and falls back down to hit the target. We also need to consider the efficiency of the method of finding that finds the solution. We cannot do a simple search of the domain of (-∞, ∞) as that would quickly utilize all of the computer's memory. Considering all of these factors, we decided to use Newton's method to find the roots as it can efficiently and accurately estimate the roots of a function.

3. Using Newton's Method to find Roots

So before we start, a quick explanation of Newton's Method might help with understanding. Newton's Method is a way to accurately estimate the root of a function by using a starting guess and recursively repeating with the step size determined by the derivative until the root is found. I won't bore into the specifics, but there is a desmos we found which has a great visualization of the method.



https://www.desmos.com/calculator/0rqvw1idkx

With an explanation out of the way, let's get on to our usage of Newton's Method. Our first step would be to find the value of our first guess. We need a value that's somewhat close to the root we are looking for with some wiggle room while also not being too far to avoid unnecessary calculations. For our method, we utilized finding a point on the graph where there is an inflection point and then subtracting some constant value as "wiggle room". We chose to use the inflection point as that can be considered as a point where the graph has "activity" and won't be near any of the areas of the graph where it diverges towards a value of infinity. There are numerous correct ways to find a "good guess" and the correct method varies with desired usages, so don't feel you are constrained to our method. So, let's start with how to find the inflection points. Writing out the full formula is unnecessary so we'll use some example constants. We can start by finding the second derivative by using the Power Rule.

$$t^4 \Rightarrow 12$$

$$t^3 \Rightarrow 5$$

$$t^2 \Rightarrow -2$$

$$t^1 \Rightarrow 6$$

$$t^0 \Rightarrow 2$$

$$f(x) = 12x^4 + 5x^3 - 2x^2 + 6x + 2$$

$$f'(x) = 48x^3 + 15x^2 - 4x + 6$$

$$f''(x) = 144x^2 + 30x - 4$$

Now that we have our second derivative, we can use the fact that zeros on the second derivative represent the x-values of inflection points (or in this case, t-values). We can use the quadratic formula to find these zeros.

$$f''(x) = 144x^2 + 30x - 4$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$x = \frac{-30 \pm \sqrt{(30)^2 - 4(144)(-4)}}{2(144)}$$

$$x = 0.09237460, -0.30070794$$

With the values of these zeros, we can now subtract our constant value and use it as our first guess in Newton's Method. To be safe, we decided to subtract 20 as it gives us plenty of wiggle room for edge case scenarios while also not overloading the computer with unnecessary calculations. We can now plug this guess into Newton's method and recursively work through it until finding the root of our quadratic.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

4. Final Aiming Calculations

We are now in the final stretch. Now that we've used Newton's Method to find the roots of our quadratic, we can use the equation from earlier and repurpose it to solve for our shooting vector and angle. *(p_target is the distance to the goal)

$$p_{aim} = p_{target} + vt + \frac{1}{2}at^2$$

$$p_{aim_x} = p_{target_x} + v_x t + \frac{1}{2}a_x t^2$$

$$p_{aim_y} = p_{target_y} + v_y t + \frac{1}{2}a_y t^2$$

Using these equations, we have all the values and can use the value of *t* we found earlier to plug in to find the *x* and *y* components of our vector. To find our angle, we can simply use trigonometry.

$$\theta_{shooting_angle} = \tan^{-1}(\frac{p_{aim_y}}{P_{aim_x}})$$

5. Finding the Shooting Velocity

Compared to finding the aiming angle, shooting velocity is much easier. For our calculations, we will assume the horizontal acceleration is zero because we are high school students who are not able to calculate air resistance and the vertical acceleration is -9.8 because we are on Earth. We can start with solving for the y component with the physics equation below.

$$v_{f_y}^2 = v_{i_y}^2 + 2a_y(\Delta y)$$

We can rearrange the equation to solve for v_f, and remove the v_i^2 because initial velocity is 0.

$$v_{f_y} = \sqrt{2a_y(\Delta y)}$$

Now that we have the y component, we can solve for the *x* component. Since the *x* component is a little more diffcult to solve for, let's lay out what we know

	x component	y component
vf	Solving for this	Solving for this
vi	equal to v_f	0
Δx	x_dist from shooter to goal	y_dist from shooter to goal
t	unknown	unknown
a	0	-9.81

From what we have, we don't have enough information to solve for the final velocity of the *x* component. However, we can solve for *t* on the *y* component and use that for the *x* component because *t* is an indpendent variable. To solve for *t*, we can start with the physics equation below

$$v_{f_y} = v_{i_y} + a_y t$$

$$t = \frac{v_{f_y}}{a_y}$$

Now that we have time, we are able to solve for the velocity of the *x* component

$$\Delta x = \frac{1}{2}(v_i + v_f)t$$

$$v_f = v_i \therefore \Delta x = \frac{1}{2}(2v_f)t$$

$$\Delta x = v_f t$$

$$v_f = \frac{\Delta x}{t}$$

$$v_f = \frac{\Delta x}{(\frac{v_{f_y}}{a_y})}$$

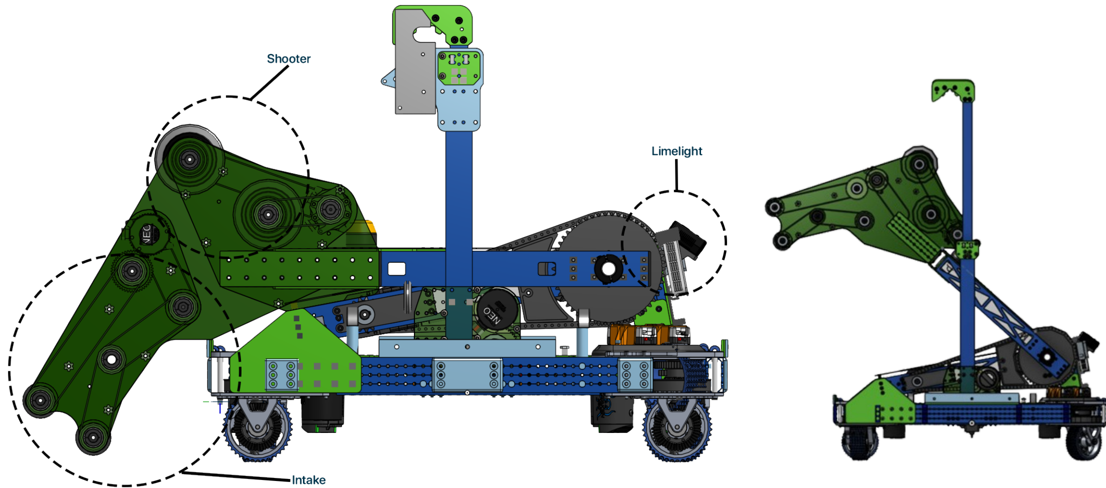
To convert to RPM which our motors need, we need to first combine the *x* and *y* velocities into a vector, then convert that velocity into radians. (*r* is radius)

$$v_{shooter} = \sqrt{v_{f_x}^2 + v_{f_y}^2}$$

$$rpm = \frac{60 \cdot v_{shooter}}{2\pi \cdot r_{shooter}}$$

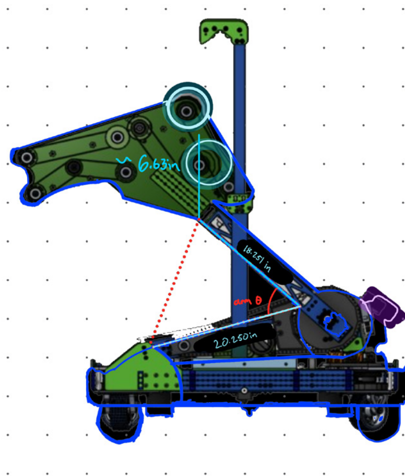
6. Calculating Offsets

Before we can implement these calculations into our code, there is one oversight to address. These calculations do not take in the robot position, but the position of the shooter. So, before implementation, we must create a formula which allows us to calculate an offset between the shooter and limelight. Keep in mind, this will be different for every robot as every shooter design is different. This is a good time to introduce our robot, SoundByte.



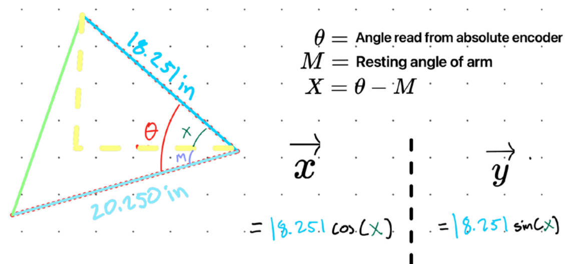
As seen by the CAD, there is a significant distance between our limelight and shooter. This offset is not negligible and must be factored into the calculations in order to create a consistent and accurate shot to the target. Another issue arises, however; the distance between the shooter and limelight is constantly changing due to movement of the arm. So, in summary, we'll have to make a function of theta (arm angle) which outputs shooter distance.

To start, let's trace the robot and draw a picture to establish the problem.



Looking at the traced picture, with cyan values being constant and red values being variables, we can create an *x* and *y* component relative to "arm theta". With arm theta coming from an Absolute Encoder on our robot.

6. Calculating Offsets (continued)



After creating an *x* and *y* component, we can then use *sin* and *cos* functions to calculate the arm distance based on the angle of the arm. After we calculate those values, we'll just have to add constant values and then we will have both and *x* and *y* distance from our shooter to the target. These values can be used in p_aim from earlier for their respective axes.

7. Implementation in Code

At this point, we have all the necessary math calculated and constants measured. Now we need to implement it in code. We can start by creating a basic to-do list and working from there.

1. Define constants/values we know (create variables).
2. Calculate our offsets from section 6 to plug into our final aiming equation
3. Create a variable for our final aiming equation
4. Find the roots of the aiming equation using Newton's Method
5. Plug the roots back into the physics equation to create a shooting vector
6. Use trig to find shooting theta.

We won't get too much into the specifics of the code as it's not the focus of this document and it's different for every team's application, but as always, our code is all open-source and can be found on

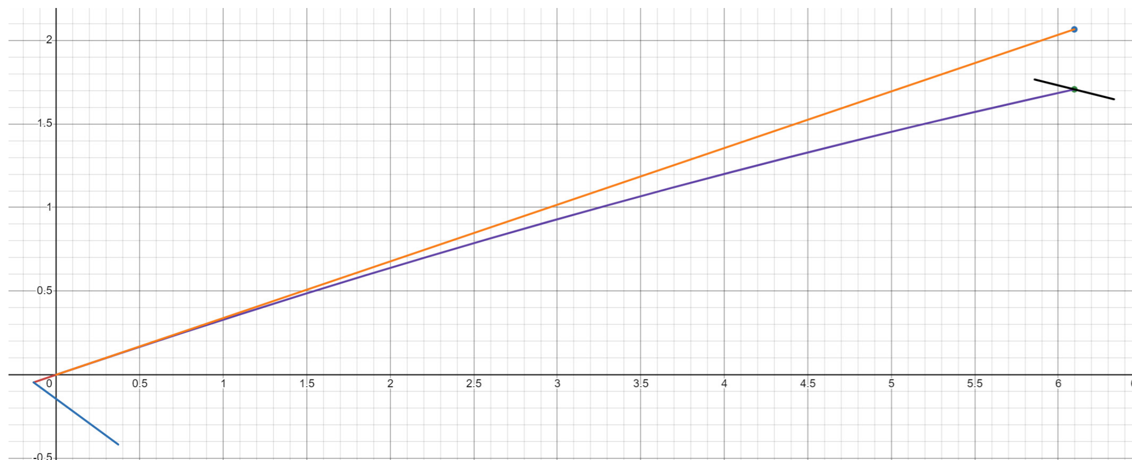
Auto-aim Code: https://github.com/Udunen/frc-ring-shooter

Our code implemented in the 2024 Robot: https://github.com/Vortx3735/2024-Bot/blob/main/src/main/java/frc/robot/commands/AutoAim.java

There is one specific concept in the implementation that took us a while to get right, however. Our "auto-aim function" takes the shooter *x* and *y* distance and gives us the required angle on the arm. The downside to this approach to implementation is that our shooter *x* and *y* distances is always changing as the arm moves up and down. To counter this, we decided to run a PID loop (Proportional, Integral, Derivative Loop). The PID loop runs until the actual arm angle equals the required arm angle given by the function and there is a trajectory which hits the speaker.

8. Visualization of Auto-Aim

To visualize our auto-aim, we decided to use Desmos, a capable online graphing calculator software. Why Desmos? Because we are familiar with it and were able to avoid wasting time by having to learn new software. We achieved a simulation in desmos by running our function stand-alone and creating a desmos file from real-world constants of different measurements from the game-field.



The purple line in this visualization is the projected trajectory of the game piece and the black line is the target

This visualization would prove to be very useful, not only in creating the code, but also allowing us to easily share and show others our methodology.