$C++$ Automated Tracking Turret in 3 Dimensions

Ethan J. Eldridge

April 18, 2011

1 The Problem at Hand

Given an object **O** at position O_p with velocity vector of O_v and a turret **T** located at the origin of the plane. **T** will quickly compute the required angle to fire upon, and how much delay to apply (if any) to the projectile to be fired. This projectile has some velocity P_v that is known to the calculations of **T** and will be launched at the angle computed by **T** after t_{delay} computed by **T** .

2 Considered with 2 dimensions

Given an object **O** at position (x, y) with velocity vector of (dx, dy) and a turret **T** located at the origin of the xy plane. **T** will quickly compute the required angle to fire upon, and how much delay to apply (if any) to the projectile to be fired. This problem can be tackled in sequence:

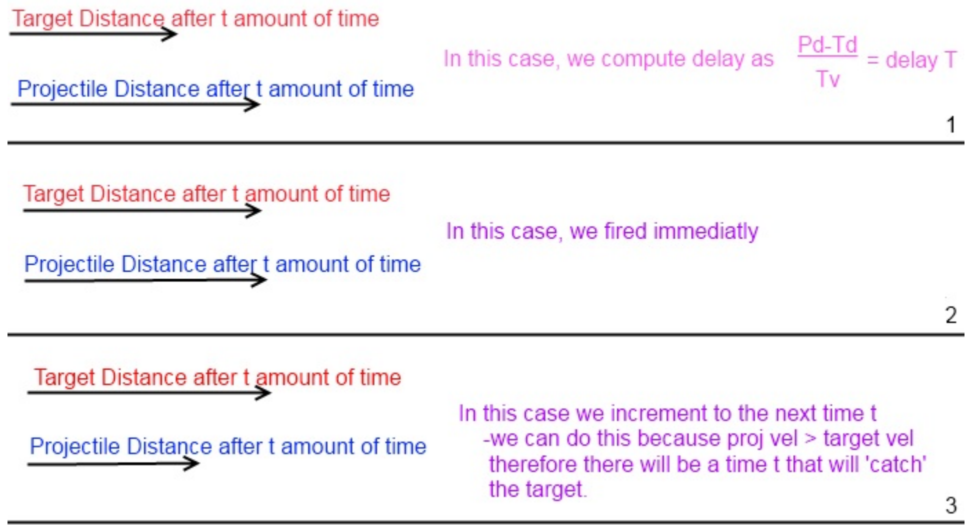
1. Find the angle θ that aligns **T** and **O** at time t
2. Find the possible distance projectile **P** can travel within t, this is defined as P_d
3. Compare P_d and O_d if $P_d \leq O_d$ increase t and repeat previous steps
4. $\frac{P_d - O_d}{v} = t_{delay}$ on principle of $v = \frac{d}{t}$

Steps 1,2 and 3 are implemented using a simple logic loop and running sentinal values for t. Also, a window of opportunity may be specified, such that **T** will only compute n steps to prevent a divergent loop. This could result from $O_v > P_v$ and in which case the logic should abort it's process. This is a simple technical detail though, and such cases are not considered in this paper.

2.1 Details of Steps 1-3

Step 1 can be computed by two dimensional vector arithmatic. First, since **T** is located at the origin point of $(0, 0)$, subtracting the point specifying the position **O** (denoted O_p) from the point specifying **T** has no effect. That is, $O_p - T_p = O_p$**,** O_p is also the vector connecting the two objects. The cosine of the angle **T** needs to turn to face **O**'s expected position (denoted O_{ep}) is the dot product of the vectors of T_p and O_{ep} provided that both vectors are normalized. This product, N_p can be used to determine the angle, by taking the $arccos(\mathbf{N_p})$ and adding it to the current angle **T** is facing.

Step 2 and 3 are defined by a loop that exits once $P_d \geq O_d$ or some specified maximum look ahead value is reached. Step 2 is found by taking the magnitude of O_p and Step 3 compares it to $P_v \cdot t$ where t is increasing as the loop runs. The cases which the comparison may find are detailed in the figure on the next page.


Target Distance after t amount of time
Projectile Distance after t amount of time
In this case, we compute delay as Pd-Td / Tv = delay T
1
Target Distance after t amount of time
Projectile Distance after t amount of time
In this case, we fired immediatly
2
Target Distance after t amount of time
Projectile Distance after t amount of time
In this case we increment to the next time t -we can do this because proj vel > target vel therefore there will be a time t that will 'catch' the target.
3

2.2 Details of Step 4 and after

In essence, step 4 is a simple calculation of $\frac{P_d - O_d}{v} = t_{delay}$ however since this algorithm is included inside of a larger program, it stands to reason with how to deal with our results. Here is the function prototype:

T::P target()

where T is our Turret class and P is our objects we've defined as **T** and **P** respectfully.

Since our function returns and P type we need to create this object, P's constructor requires 4 parameters. the x and y components of it's position vector, and it's x and y components of it's velocity vector. The position is given simply by $(0, 0)$ because the projectile is fired from the origin. The velocity vector can be given by $P_v \cdot sin(\theta)$ for x and $P_v \cdot cos(\theta)$ for y. With θ being the heading computed by Step 1.

2.3 Within the Program Itself

The program takes command line arguments of the target's position and velocity, as well as how many simulation steps to run. So calling the exe from a window's command line looks like this: `consoleturret2d.exe` 10 10 -1 -1 20 and this would create **O** with $O_p = (10, 10)$ and $O_v = (-1, -1)$ for a runtime simulation of 20 updates.

The runtime simulation variable makes more sense when you view the program from it's procedure. This is defined as:

1. Begin For Loop running for the runtime simulation amount (in this case, 20)
2. Call world.simulate() (Steps 3-6 are within world.simulate())
3. update positions by some t amount of time.
4. if the turret is armed, call the targeting function[1]
5. if the turret is locked onto the target (meaning the previous step successfully targeted **O** increment the simulation by the delay time and fire the projectile.
6. Display the updated positions to the console.

[1] described by Steps in the §2.1

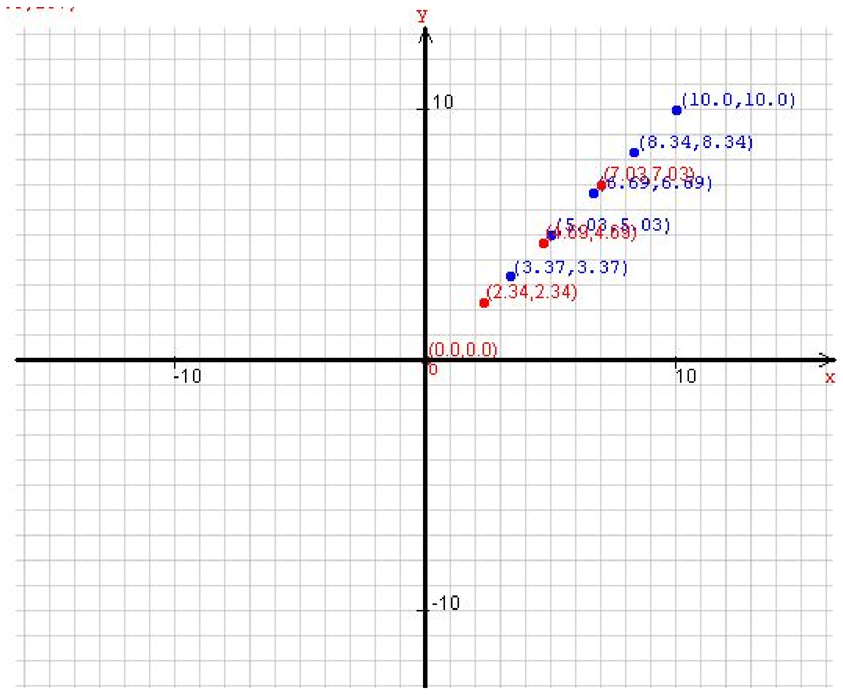
2.4 An Example Output From the Program

```
Turret Program 2D

Turret Theta: 0.785398 Turret Delay: -1
Turret Vectr: 0.707107,0.707107
Turret Armed: 1 Locked: 0 Fired: 0
Turret Estimation of UFO: 0,0

UFO: Pos: 10,10 Vel: -1,-1 (1.41421 units / s)
----------------------------------------------------------------------
Turret Theta: 0.785398 Turret Delay: -1
Turret Vectr: 0.707107,0.707107
Turret Armed: 1 Locked: 0 Fired: 0
Turret Estimation of UFO: 10,10

UFO: Pos: 10,10 Vel: -1,-1 (1.41421 units / s)
----------------------------------------------------------------------
Turret Theta: 0.785583 Turret Delay: 0.656854
Turret Vectr: 0.707238,0.706976
Turret Armed: 0 Locked: 1 Fired: 1
Turret Estimation of UFO: 8.34315,8.34315

UFO: Pos: 8.34315,8.34315 Vel: -1,-1 (1.41421 units / s)
Fired Projectile: Pos: 0,0 Vel: 1.41448,1.41395 (2 units / s)
----------------------------------------------------------------------
Turret Theta: 0.785583 Turret Delay: 0.656854
Turret Vectr: 0.707238,0.706976
Turret Armed: 0 Locked: 1 Fired: 1
Turret Estimation of UFO: 6.68629,6.68629

UFO: Pos: 6.68629,6.68629 Vel: -1,-1 (1.41421 units / s)
Fired Projectile: Pos: 2.34358,2.34271 Vel: 1.41448,1.41395 (2 units / s)
----------------------------------------------------------------------
Turret Theta: 0.785583 Turret Delay: 0.656854
Turret Vectr: 0.707238,0.706976
Turret Armed: 0 Locked: 1 Fired: 1
Turret Estimation of UFO: 5.02944,5.02944

UFO: Pos: 5.02944,5.02944 Vel: -1,-1 (1.41421 units / s)
Fired Projectile: Pos: 4.68716,4.68542 Vel: 1.41448,1.41395 (2 units / s)
----------------------------------------------------------------------
Turret Theta: 0.785583 Turret Delay: 0.656854
Turret Vectr: 0.707238,0.706976
Turret Armed: 0 Locked: 1 Fired: 1
Turret Estimation of UFO: 3.37258,3.37258

UFO: Pos: 3.37258,3.37258 Vel: -1,-1 (1.41421 units / s)
Fired Projectile: Pos: 7.03074,7.02814 Vel: 1.41448,1.41395 (2 units / s)
----------------------------------------------------------------------
```

2.5 Explanation Of Program Output

There are some values and points of interest I would like to clarify from the output on the previous page.

- Turret Delay Value: -1 is the null value of delay time. When the delay is not calculated yet, this negative time appears.
- Armed, Locked, and Fired: These are boolean values that control whether the targeting function and the fired projectiles are incremented or not.
- The command line values passed to the program are as follows, 10 10 -1 -1 10[2]
- Fired Projectile does not appear until the 3rd display because the display command does not display this until **T** has targeted **O**
- The **T** delay value is not reset to 0 after **P** is launched, this is because the target function is what updates that value, and the target function is not called while **T** is not armed.
- all angles are in radians.
- the below figure shows a plot of the points given from the program. As you can, the points cross inbetween the third and fourth points which you can also see in the sixth display of the simulation output.



As you can see, the strategy described in the previous section works well. And for a simple two dimensional problem these methods are adequate. However, for a problem of three dimension involving gravity, these methods must be redesigned. This leads us to…

[2]Note that the output shown is only for 6 displays of the simulation

3 Considered with 3 dimensions

The fundamental algorithm used in §2 is reusable for the problem in three dimensions, however each of the steps involve more complex calculations. The increased complexity is not neccesary fully grasped from the following description of the algorithm.

1. Check if **O** is within range.
2. Calculate the angle to face **T** towards an estimation of **O** at t in the XY plane[3], this angle is denoted $P_{\Theta_{xy}}$
3. Calculate the angle of elevation θ_z, call the line that connects **T** and **O** *ox*, a line perpindicular to this is denoted as *oy*. These two lines create what we call the *ox*-*oy*-plane.
4. Use $\theta = arctan\left(\frac{{O_v}^2 \pm \sqrt[2]{{O_v}^4 - g(gx^2 + 2y{O_v}^2)}}{gx}\right)$[4] with respect to the *ox*-*oy* plane to create an angle θ to fire on the *ox*-*oy*-plane. Using this θ, specifying it as $P_{oxy\theta}$ find the time of flight using $t = \frac{2P_v sin(\theta)}{g}$
5. compare the time of flight to the original t considered. If $t_{flight} \leq t$ then we may move out of our loop, if not, we increment t and start again.
6. Compute the delay time, this is simply $t - t_{flight}$
7. $P_{oxy\theta} + \theta_z$ will provide us with our final angle to launch our projectile on, this angle is called P_{Θ_z}
8. Using P_{Θ_z} and $P_{\Theta_{xy}}$ we can created out launch vector for **P**
9. Fire the projectile after the delay time is passed

While this algorithm is twice as long in step number as the two dimensional one, A lot of the calculations require less simulation and checking then the two dimensional problem. For example, when computing the delay time, it is now a simple subtraction instead of solving $v = \frac{d}{t}$ for t. The main complication that one runs into when figuring in three-dimensions is the effects of gravity and their effect on the height of an object through a continuous modification of the objects velocity.

3.1 Dealing With Gravity

As we can already see, our equations governing motion are no longer simple $O_p = O_v \cdot t + O_p$ expressions. But must be determined with respect to gravity. A solution might be to use two 2-dimensional vectors, the first being composed of the x and y components, and the second being composed of the magnitude of the first vector and the z component. That way the $\vec{xy}$ can be updated by our simple expression, and then the second may be updated with respect to velocity. However, this is a gross inflamation of what we can do, not to mention overcomplicating what a few lines of code can do for us.

The actual method used in this project is to create a 3-Dimensional vector $\vec{A}$ and when updating, update with our simple expression first, and then compute $\vec{P_z}$ as $-\frac{1}{2}gt^2 + v \cdot t$[5]. $\vec{A_z}$ can then be updated by subtracting $g \cdot t$ from the current velocity. These two vectors, $\vec{P}$ and $\vec{A}$ describe the position and velocity of **O** as well as **P** . A quick scan of the steps

[3]This is the 'two-dimensional' plane which you can envision as the Z axis's base

[4]We use x,y here instead of $x = rcos\theta$ and $y = rsin\theta$ because of the use of the *ox*-*oy*-plane instead of polar coordinates. This is done because for the sake of this project, cartesian coordinates are used because, in general, most people visualize problems in cartesian coordinates easier than polar.

[5]The full form of this equation is $-\frac{1}{2}gt^2 + v \cdot t + v_0$ however, because v_0 is 0 we omit it.

in §3's algorithm description shows that the other two equations we use both make use of g and therefore take gravity into effect without any more modification by our code.

3.2 Steps 1-5 in Detail

The first step in both algorithms is to determine if **O** is within range. Determining this in 3-dimensions is more complex than the 2-dimensional version. Specifically because the maximum range is determined by the angle of **T** . At first, intuition tells us that an angle of 45° will give us the maximum *horizontal* distance; and an angle of 90° will give us **P**'s maximum height. Obviously, we can test **O** to see if it qualifies in this idealistic case, but more than likely, O_p will fall between 0 and $\frac{\pi}{2}$[6] and this case is expensive to try for each change in angle between **O** and **T** . We would have to compute the angle, and then the magnitude of O_p , and compare this to the maximum height attainable by **P** . However, this maximum height only happens at some certain time t_{ideal}, t_{ideal} and t would have to be either very close, or equal, in order to attain a hit. All these calculations and checks would be, as I said, expensive.

A solution to this problem, is to use the equation from Step 4, this equation will return imaginary roots if the object is out of range. While the square root and exponentiation operations are generally considered expensive, compared to the alternative, I feel their use is justified. Note that the check is the 1st Step, while the equation is part of Step 4. Within our loop, a break can occur whenever **O** is determined to be out of range.

Also, we can consider what out of range means, we can be out of range in three different ways, too far out with respect x, too far out with respect to y, and too far out with respect to z .an object with position $(254, 0, 0)$[7] can be hit, while an object $(254, 1, 0)$ can't because $x^2 + y^2 + z^2 > 254.929$. From this we could slowly gather that the magnitude of the entire vector could not exceed that of the maximum distance coverable by **P** with some fixed P_v. This would be less expensive than many of the methods we've describe, especially if we store the maximum distance in some global variable, which we could then make our checks against. Then the only operation left to do is take the magnitude of O_p and compare them.

This last method is the one we use in our actual implementation of the algorithm, because it does not involve dependency on another step before or after it as the second strategy did. From here, we move on to Step 2 which is a far easier problem. The calculation of $P_{\Theta_{xy}}$ is effectively the same as described in §2.1 description of Step 1. Because this angle is strictly on a horizontal basis, gravity is not considered in our equations. If we considered wind speed and direction this step would become much more difficult than it is. Causing us to face at an angle not perfectly alligned with **T** estimated position. However, since this is not considered in this project, we move on to finding P_{Θ_z}.

To find P_{Θ_z}we first need to create an *ox-oy*-plane, and in order to do this we need to compute the angle of elevation. We do this using the following equation

$$\theta_z = arctan\left(\frac{O_{p_z}}{\sqrt{{O_{p_x}}^2 + {O_{p_y}}^2}}\right)^{8}$$

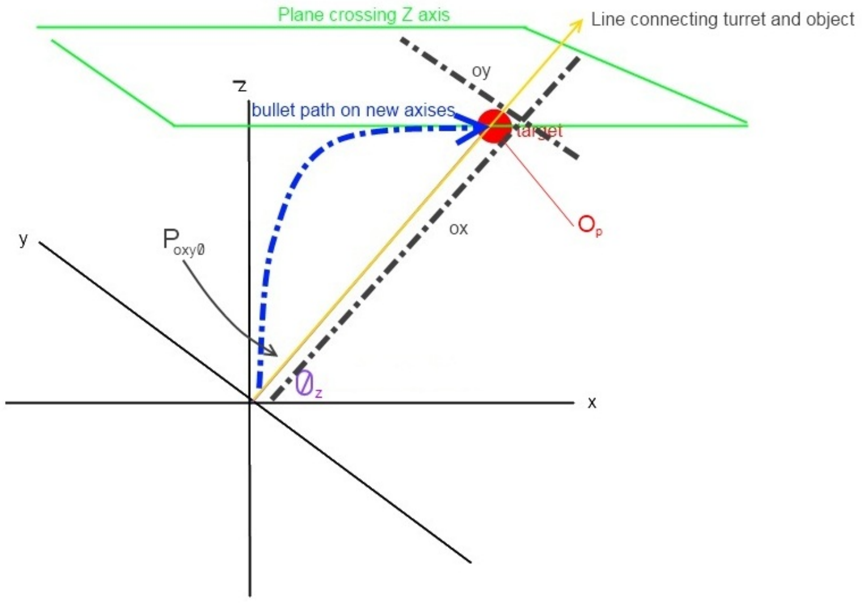
Using θ_z we create our *ox-oy*-plane and solve the equation in Step 4 with each x being it's

[6]From this point on, all angles will be given in radians, the previous use of degrees was because most people have a more intuitive grasp of degrees. Specifically 90° as oppose to $\frac{\pi}{2}$

[7]In this example, $P_v$is 50 units per second. Giving a maximum distance coverable of 254.929 in a single direction given 45° as T_{Θ_z}

[8]this is effectively the standard defination of $arctan\frac{opp}{adj} = \theta$ the more complicated bottom is because when computing the angle of elevation, one can think of the problem in 2 dimensions with an xy axis and a z axis, where the xy piece is found from the hypotenuse of the x and y axis of the base plane.

ox counterpart and *y* likewise. From that equation we calculate angle $P_{oxy\theta}$. We then use the $P_{oxy\theta}$ to estimate t_{flight} (Steps 2-4) and then using t_{flight}, we compare it to t and if $t_{flight} \leq t$ then we have found the angle which we will fire on. All that remains from here is to shift from the *ox*-*oy*-plane back to the 3-dimensional space that **T** lives in. As seen in the following illustration, $P_{oxy\theta}$ must be added to θ_z in order to obtain the actual angle which we are to launch on.



The visualization of the *ox*-*oy*-plane created by θ_z and **O**

3.3 Steps 6-9 in Detail

Step 6, due to our use of an equation to find t_{flight} is made almost as simple as t_{delay} in 2-dimensions, a simple subtraction computes any delay time we might require. After this, we only need to do one more step before being able to create our launch vector for **P** . This step is simpler than the previous, as it is a quick calculation of $P_{oxy\theta} + \theta_z = P_{\Theta_z}$. Now that we have both P_{Θ_z} and $P_{\Theta_{xy}}$ we create the launch vector.

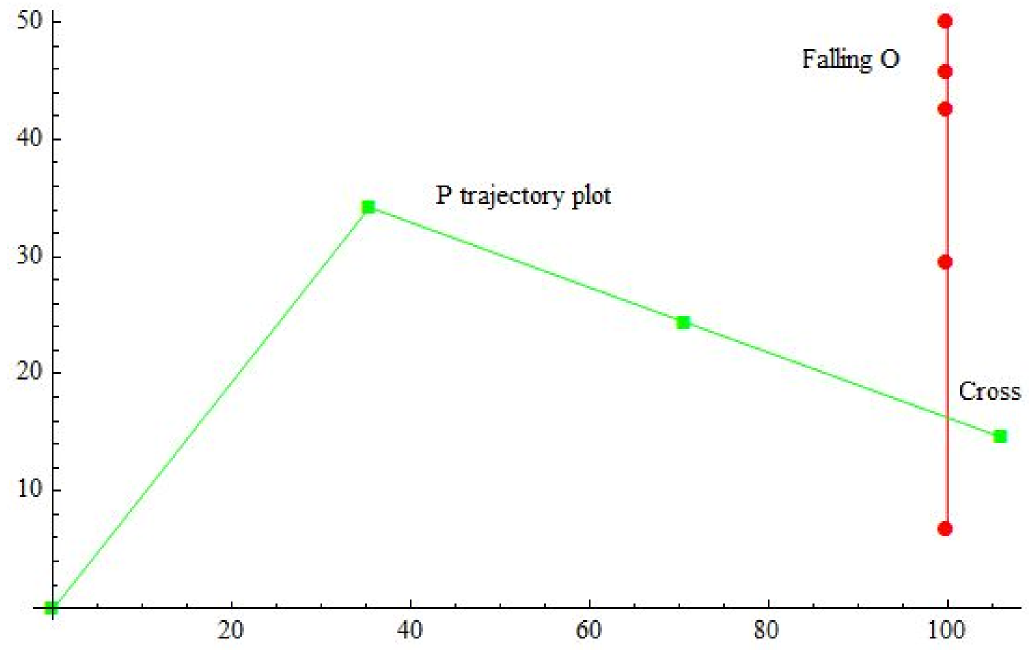
$$\vec{P}_{launch\,vector} = (P_v \cdot sin(P_{\Theta_{xy}}), P_v \cdot cos(P_{\Theta_{xy}}), P_v \cdot sin(P_{\Theta_z}))$$

And all that is left is to let the simulation know that **T** is locked and ready to fire as soon as t_{delay} units of time have passed by.

3.4 Differences between the 2D and 3D versions of the program

[In regards to the program command arguments]

Similarily to the 2-dimensional program, this version takes command line arguments of the target's position and velocity, and how many simulation steps to run. The only difference between §2.3's explanation and this one is that the command line would be given arguments of the form `consoleturret2d.exe` 100 100 50 -1 -1 10 20 instead of `consoleturret2d.exe` 10 10 -1 -1 20. That is, the *z* components of **O** are included into the parameters. Besides this, the procedure is the same.



A plot of the z components and the xy components by Wolfram Mathematica using the programs output.

4 Complexity and Optimization

4.1 Time Complexity

The algorithm for targeting only has a single loop.

```
while(true){
    if(lookAhead < t) {  break; }
    if(Td > maxD ) { break; }
    if(t > flightTime ) { break; }
    t++
 }
calculation of heading
estimation of delay
```

This while loop can also be made into a for loop

```
 for(t = 0; lookahead > t && Td < maxD && t < flightTime; t++){
 simulation and calculations
 }
calculation of heading
estimation of delay
```

At worse, the algorithm will find that the t necessary to hit O is just before the look ahead value. The lookahead value can be computed by $\frac{v \cdot sin(\frac{\pi}{2}) + \sqrt[2]{(v * sin(\frac{\pi}{2})^2)}_9}{g}$ and if

9 $\frac{v \cdot sin(\theta) + \sqrt[2]{(v \cdot sin(\theta)^2) + 2gy_0}}{g}$ y_0 is the initial height, and since $\mathbf{T}$ lies at at the origin, the value $+2gy_0$ will always be 0. Also, $\frac{\pi}{2}$ is used because the maximum time of flight can be attained when an object is launched straight into the air

we denote the result of this calculation as N_t then the worse runtime for the algorithm can be expressed as $\mathbf{O}(N_t)$.

$\mathbf{O}(N_t)$ is a very broad analysis of the algorithm, however we can be far more specific and state the runtime as

$$\mathbf{O}(\ [5 \cdot (\mathbf{O}(\sqrt{N_i} \cdot \mathbf{M}(N_d))) + 26 \cdot \mathbf{M}(N_d) + 20 \cdot \Theta(1)] \cdot N_t\ )$$

this can be computed from studying the source code and counting the number of calculations performed.

The first term comes from the complexity of trigonometric functions, the second from the amount of multiplication and divisions, and the third from addition,subtraction and boolean comparison operations. $\mathbf{M}(N_d)$ stands for the time to perform multiplication (or division) based on the amount of digits in the number being operated on. $\mathbf{M}(N_i)$ stands for the time to perform multiplication (or division) based on the decimal precision.

4.2 Complexity of individual components of the algorithm

Component	Operations	Complexity
Magnitude of a Vector	$\sqrt{x^2 + y^2 + z^2}$	$\text{O}(4\cdot\ M(N_d)\ +\ 2 \cdot \Theta(1))$
Angle to hit xy	$arctan\left(\frac{{O_v}^2 \pm \sqrt[2]{{O_v}^4 - g(gx^2 + 2y{O_v}^2)}}{gx}\right)$	$\mathbf{O}(\sqrt{N_i}\cdot\mathbf{M}(N_d)\ + 3\Theta(1) + 8\cdot\mathbf{M}(N_d))$
Time of Flight	$t = \frac{2P_v sin(\theta)}{g}$	$\text{O}(2\cdot\ \text{M}(N_d) +\ \text{O}(\sqrt{N_i} \cdot \text{M}(N_d)))$ [10]
θ_{xy}	$arctan\left(\frac{O_{py}}{O_{px}}\right)$	$\mathbf{O}(\sqrt{N_i} \cdot \mathbf{M}(N_d) + \mathbf{M}(N_d))$
θ_z	$arctan\left(\frac{O_{p_z}}{\sqrt{{O_{p_x}}^2 + {O_{p_y}}^2}}\right)$	$\mathbf{O}(\sqrt{N_i}\cdot\mathbf{M}(N_d) + 4\cdot\mathbf{M}(N_d) + \Theta(1))$
$t++$	$t = t + 1$	$\Theta(1)$
updatePosition Function	Function code omitted	$\mathbf{O}(7\cdot\mathbf{M}(N_d)\ +\ 10 \cdot \Theta(1))$

The runtime complexity is simple to derive once you have the individual components mapped out, and the reader is invited to double check the addition and simplification of terms into it's final form. Although, if the reader does this, they may notice a discrepency between the number of constant time operations in the final complexity and individual summation. This is due to the boolean checks against terminating the loop.

Also, the amount of trigonometric functions differs by one. This is because within the code, the time of flight is computed twice within the loop. Once to check against terminating, and the other to assign the variable inside the structure to keep track of it. Also, the function code for the updatePosition function is omitted, the numerous multiplications, additions, and boolean checks within that function are due to corrections to the position of **O** or **P** with respect to gravity.

The runtime of the algorithm is dominated by multiplication operations, and therefore will be affected by the precision of the calculations as well as how large the numbers are. Generalizing the notation, we could simply remove the constant time operations because they are, with respect to multiplication and trigonometric functions, negligable based on their time. They are included here for details sake.

4.3 Optimizations

1. Within the angle to hit xy operation, both v^2 and v^4 are stored into variables at the beginning setup of the program, this reduces the number of multiplications within that function from 13 to

[10] O_{py} is the y position of **O** and similarily defined for O_{px}

8

2. Within the time of flight, storing $2 \cdot P_v$ as a variable at the beginning setup of the program reduces multiplications within that function by one (from three to two operations that take $\mathbf{M(}N_d\mathbf{)}$.
3. also, within the overall runtime we could reduce the number of trigonometric functions by one by storing the time of flight into a temporary variable and checking it against t and then assigning it. This was not done within the program to reduce storage complexity.
4. An optimization that was not made was to replace the while loop with a for loop. But this was done to improve the actual empirical runtime, because each of the conditions are tested at different points within the loop, if a satistfactory condition is found, the loop exits before any unnecessary code is reached. This allows for expensive operations not to be done, while using a for loop would result in this occuring, or for the amount of boolean checks to increase.

5 Credits

- Main program source code written by Ethan Eldridge
- Vector2d class found online at Terathon Software, Vector 3d created by Ethan Eldridge through modifications of 2d vector class.
- Formulas used in §3 Step 4 (Hitting (x, y) given initial velocity)
- Formula used in §3 Step 4 (Time of Flight)
- Height of a projectile after some t with intial velocity v
- Formula in §3.2 derived by Ethan Eldridge, explained in footnote 8 in the same secton
- §3.2 Creation of the *ox*-*oy*-plane suggested by Adam Thibault during correspondence with Ethan Eldridge on 29 March 2011
- Computational complexity of addition, multiplication, and trigonometric functions