Improved PID with Approximated Motion Profiling

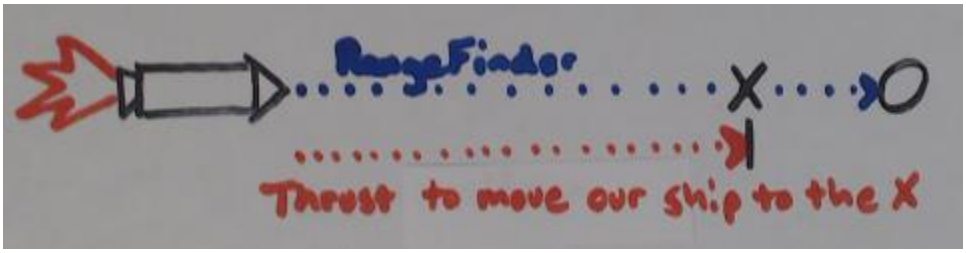
for the Starbase game Asteroid Approach Problem

Starbase Name: Darkyshadow

By: Eric

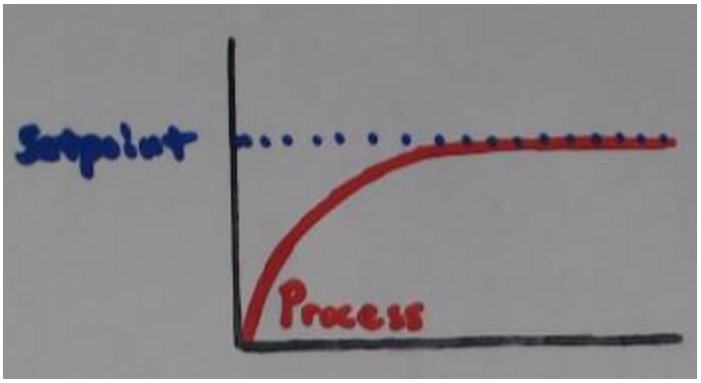
The Problem:

We want to move our spaceship to a fixed position in front of an asteroid quickly without overshooting and hitting the asteroid. We do this by continuously monitoring a RangeFinder that gives us a distance measurement to the asteroid, then throttling the Thrusters accordingly



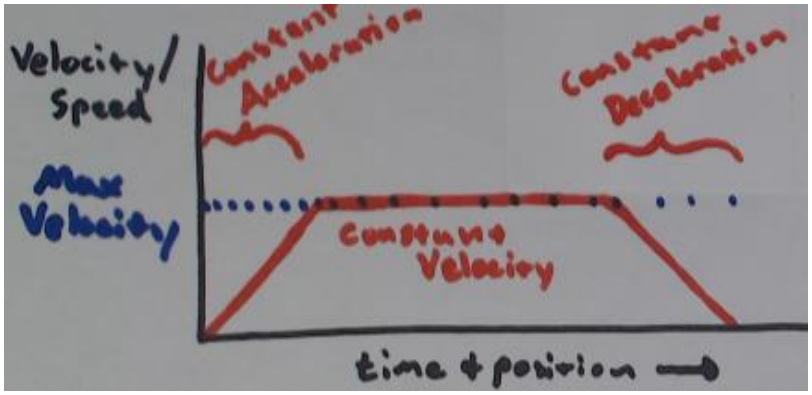
Why PID Alone is a very poor solution for this problem…

A PID closed loop control system's goal is to get the Process (A ship flying forward due to thrust) to the Setpoint as fast as physically possible.



This is a poor problem for PID alone to solve because PID does not take into account many of the physical constraints that a physically moving object must undergo, instead a Profiled Motion PID controller **NEEDS** be used…

Any positional change of a real body involves 3 things. This applies to a car, a person, or in our case a video game spaceship… =)



A period of **Acceleration** (constant acceleration to make it easy), a period of **Constant Velocity**, and a period of **Deceleration** (again, constant Deceleration to make it easy.

The point here, is that PID alone has <u>NO KNOWLEDGE</u> that a physical system should start slowly, ramp up to full speed, coast for a period of time then slow down gracefully before stopping. All PID "Knows" how to do is to get you to your setpoint quickly…. This **much** better solution would consist of a Velocity PID controller that forces the ship to a specific velocity, and then a motion profile controller that would feed the desired velocity values into the Velocity PID controller.

Implementing the above is doable of course, but will be difficult given the constraints of the in game programming language yolol, i.e. require multiple chips, global variables and process synchronization etc etc etc… All the BS I don't want to deal with for this problem right now, but may do in the future!!! But I think we can make a pretty damn good approximation of the above by hacking up a standard PID algorithm. But only if we understand where PID has it's strengths, it's weaknesses, and by bending, manipulating, and coercing it to our WILL!!!!!!

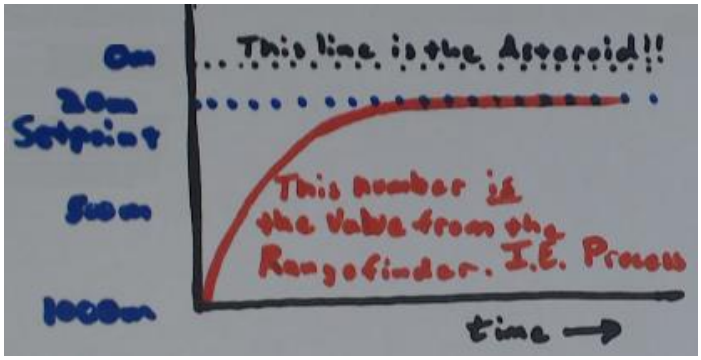
Ok, world domination aside… Lets describe a crudely applied PID algorithm to the problem, how it "kind of works" and how we can manipulate it to work much better now that we know what an optimal (profiled) motion approach would look like.

<u>This Next Part is ALL INTRO PID Control System Theory STUFF and is perfect for the beginner, even if you don't care about approaching asteroids in a game!</u>

OK, we are going to need some terminology to speak this language of control theory… I will keep using the Asteroid Approach as our example.

Setpoint = desired distance from the asteroid. Where we want to stop…

Process = actual distance from asteroid



As time progresses the PID Closed Loop Control Algorithm, looks at our current position (**Process**), looks at our desired position (**Setpoint**) and adds or subtracts

thrust accordingly. The value that PID generates to change our thrust will be called the **Manipulated Variable**. So now we have 3 variables thus far in our algorithm…

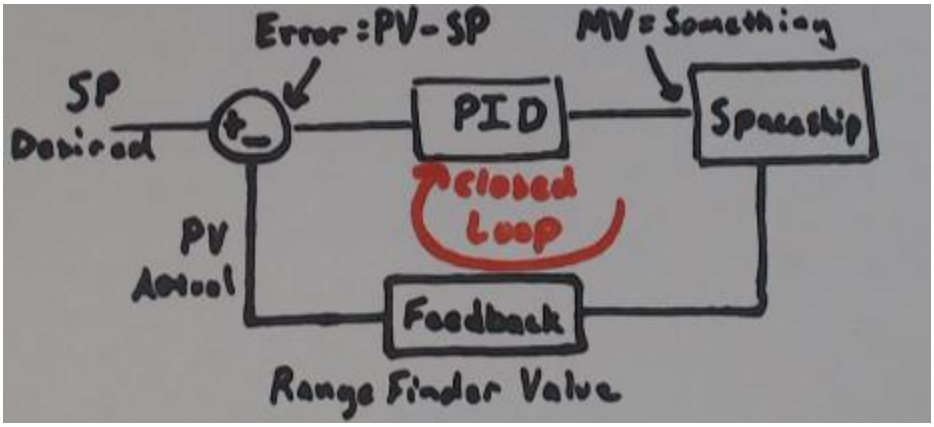
SP: Setpoint i.e. desired position

PV: Process Variable i.e. actual position

MV: Manipulated Variable i.e. how much to drive the thrusters to cause **PV** to move closer to **SP** and eventually equal eachother…

A **PID Closed Loop Control System** is a type of:

Closed Loop Control System: This just means that it looks at the desired position (**SP: Setpoint**), Looks at the actual position (**PV: Process Variable**) AKA **Feedback**, and then decides how to manipulate the output (**MV: Manipulated Variable**) in order to achieve **PV** equal to **SP.**



The **PID** in PID Controller stands for **Proportional Integral Derivative**. Don't FREAK OUT about the calculus part. I promise you, you don't need to be able to do calculus to follow…

Error: is the difference between where we are (**PV**) vs. where we want to be (**SP**). In other words it quantifies "how far away" we are. Or just

like when we were whining children on a road trip, “are we there yet? How long? Are we there yet?”

Error = Process_Variable – Setpoint

When we are at our destination, **Process_Variable = Setpoint**, hence **Error = 0**, when we start at the furthest distance away, like in the case our asteroid approach problem. **Error = 1000 – 20 = 800**

From this **Error** term we will calculate 3 values, a **Proportional** term, an **Integral** term, and a **Derivative** term. These 3 terms will get added together and become our **MV.** Ultimately this **MV** is our thruster value and when we arrive at our destination, **PV = SP** and all is well… you wish it was that easy, it’s NEVER that easy.... BUT I PROMISE, the **Integral/Derivative part is EXTREMELY easy**

Lets write a program in pseudo code to describe this loop before we get into the P + I + D Calculations…

Look at the following code, then look at the previous image, keep going back and forth until it makes sense! If it doesn’t make sense, then email / call / discord me or any other local engineer for help. Nothing else will make sense if this does not!

loop:

SP=20

PV=RangeFinderDistance

Error=PV-SP

--Do Something (calculate PID Terms based off of Error)—

MV= output from PID, the PID terms

Ship Thrusters = MV

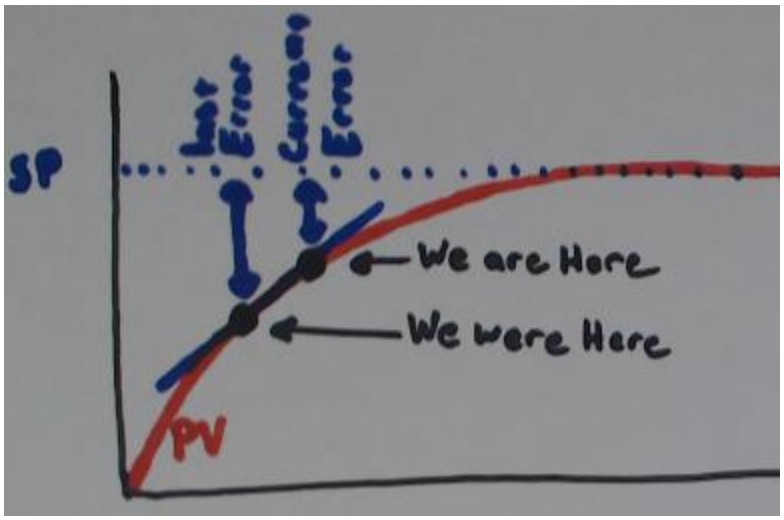
GOTO Loop

See what's going on. The code is doing what the previous page closed loop control flow diagram is describing. Don't continue until this part makes complete sense. OK GOT IT? Lets calculate our terms…

From now on, when I say WE, I'm talking about the spaceship at it's current position.

The Proportional term: Is just a value telling us how far away we are from our destination… WELL… We already have a value for that, it's called the **Error**. The **Error <u>IS</u>** our **Proportional term**. That's easy, done!

The Derivate term: Is much easier to see visually. The derivate on the first day of a calculus class is told to us as the "Slope of the Tangent Line". AWWW SHIT!!!!, there's a tangent line? It has a slope? And what the hell is this line on?….. Chill dude!!!! We already saw the graph that this tangent line is on, and all we need to do is calculate it.



At any point in time, i.e. at any iteration of our loop we are somewhere on this red curve of **PV** towards victory **SP**. We can approximate a derivative

very easily with the following line of code…

Derivative = Current Error – Last Error.

It is sometimes said that the Derivative term is a predictive value of what the system looks like it's going to do in the future. I think this is a valid statement but there is more to it than that. Notice something incredibly important here… If **PV** is getting Closer to **SP**, then **Error** is getting smaller, if **Error** is getting smaller, then **Derivate equals a NEGATIVE VALUE!!!!! Proportional** pushes you forward, **Derivative** says: "But not too much!!!". In other words the polarity of **Derivative** is always such that it slows the large changes that **Proportional** would cause. Let's add this and the **Proportional term** to our code.

loop:

SP = 20

PV = RangeFinderDistance

Error = PV - SP

P = Error //Proportional Term

D = Error - Last_Error //Derivative Term

Last_Error = Error //Save a Copy of this Error to calculate

// the derivative on the next iteration

MV = P + D

Ship Thrusters = MV

GOTO Loop

Notice how easy it was to add our proportional term and our derivative term calculations. To calculate Derivative, we needed a new variable to temporarily save the current Error for the next iteration. The next time this loop executes that **Last_Error** variable is holding the error value from the previous execution. Also notice that we are now setting the Manipulated variable to our 2 terms P and D added together. BUT… do we really want 100% of those 2 terms. This algorithm works for a lot of different systems, Temperature Control, Car Cruise Control, many kinds of industrial processes, and apparently Spaceships Approaching Asteroids in a Video Game. It is **necessary** to tune this algorithm for different processes… We can add tuning parameters, we call them Coefficients…

Kp: Coefficient of Proportional

Kd: Coefficient of Derivative

These are values that will typically be on a scale of 0 to 1, i.e. 0% to 100%. If I want 100% of something, I multiply it by 1, if I want 50% of something, I multiply it by 0.5 etc… right….

Kp = 0.4

Kd = 0.7

loop:

SP = 20

PV = RangeFinderDistance

Error = PV - SP

P = Error //Proportional Term

D = Error - Last_Error //Derivative Term

Last_Error = D //Save a Copy of this Derivative Term for

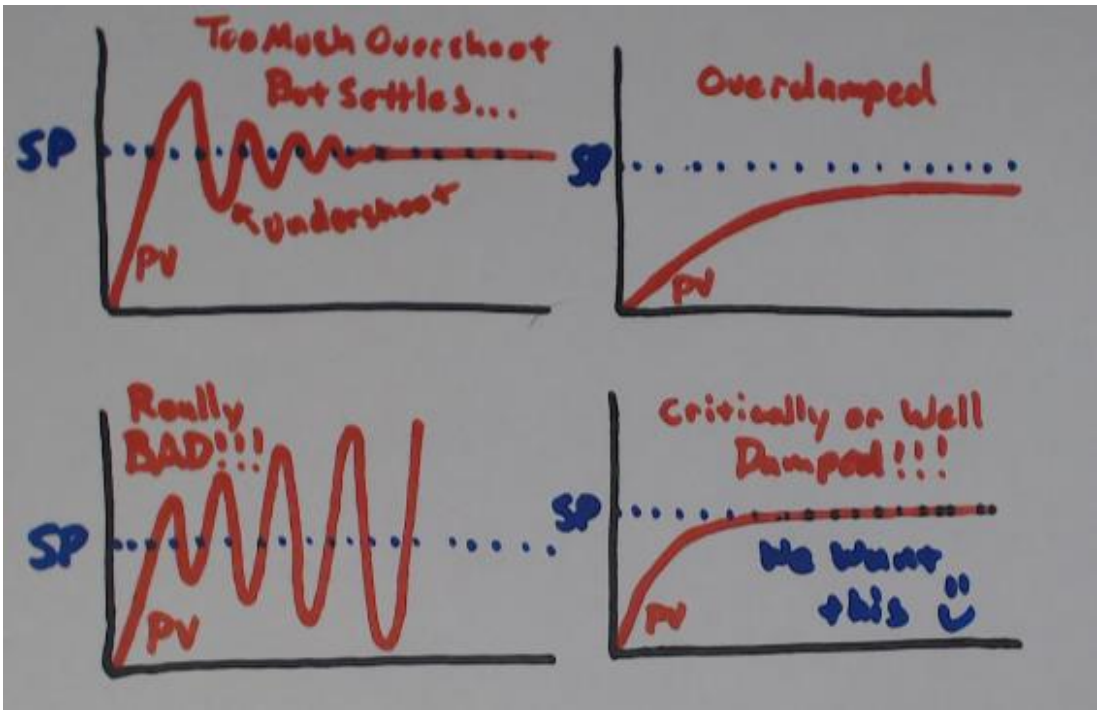
// the next iteration

MV = P*__Kp__ + D*__Kd__

Ship Thrusters = MV

GOTO Loop

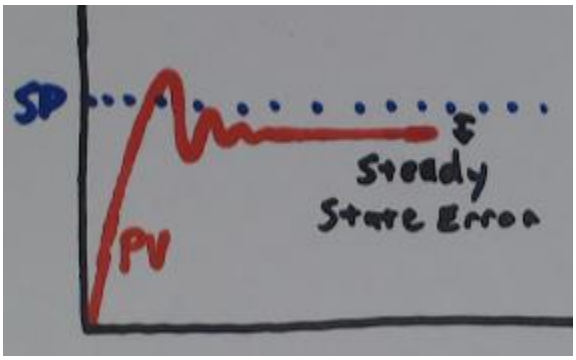
Now we have the ability to “TUNE” our algorithm by setting **Kp** and **Kd** before our loop starts running. This will allow us to adjust the performance, i.e. response of the system to be desirable, in other words, get us to our desired setpoint quickly while minimizing **Overshoot** and **Undershoot**. Some images will help us out here!!



You probably have experienced all of these if you have played with any of the available “Approach” Scripts in the game. Upper Left will look and feel like rubberbanding in your spaceship once you get to the destination, or probably even includes hitting an asteroid during the overshoot before you even get a chance to undershoot. You may have even experienced “really bad”, where you overshooted, thought that things looked good, then you never settled and oscillated yourself back into the asteroid or flew off to the side. Now that I think about it, the ability to first-hand experience and ride the PID response is about the coolest thing about control systems in this game. I knew there was a reason I liked it. I guess you could say the same thing about turning on your cruise control, hopefully it’s never as “exciting” as an under-damped overshooting ride straight to hell!!! =)

I have left out a pretty useful term thus far, **Integral**…. I have purposefully left out the mysterious Integral term because it is the most misunderstood. Realistically and in all seriousness, the above code will work pretty well for most situations. It is known as a PD Controller. YES!!! We have the flexibility to create P-Only Controllers, PD Controllers, PID Controllers, PI Controllers etc… Each one has its strengths and weaknesses in different situations, and we will see a little bit about that much later.

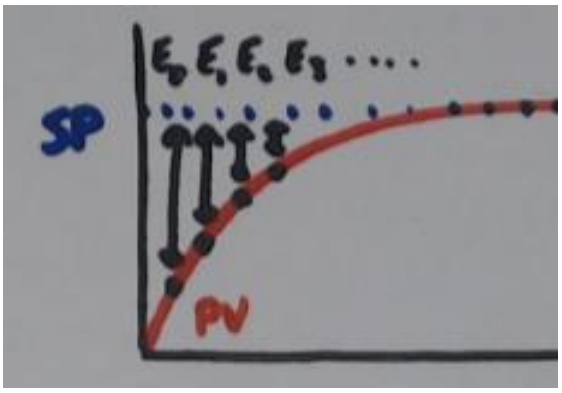
However, **PD** Controllers like above, suffer from one major problem. They have non-zero steady state errors… OK WHAT??? Now we need some context. To illustrate this, let’s look at a slightly underdamped PD Controller’s response…



Notice we settled, but not at the Setpoint. Now, this may actually be tolerable and acceptable depending on the situation or application you are designing for. But… This illustrates the problem that the **Integral term** will solve for us…

Also, it's worth mentioning that in regards to the Asteroid Approach problem, this results in settling at a value too short or too close to the Asteroid.

The Integral term: Now given what we have learned thus far, taking the next step with the **I** Term is actually not too difficult. And on the 2nd day of calculus the professor said "The **Integral** is the area under the curve". Alright, so let's approximate that next with another image and then we will see how to do it easily in code.



To get the **Integral** term all we have to do is keep a running accumulation of all the error terms, this will approximate the "Area above? The curve…." Meh, tomato tomato… One person's negative is another upside-down-person's positive…. As long as when you add the **Integral** term to the **MV** it causes the system to move forward when you are behind, it's GOOD!!!! Again, the purpose of the **Integral** term Is to "find" the value of the steady state error of a PD controller. We can also call this a bias. And before I forget, let's show the I term calculation code:

I = I + Error

Just keep a running accumulation of all error. There are issues with I, it can overflow and underflow numerically, i.e. the number gets too big or too small to fit in a variable. You can numerically clamp this at the extremes to prevent overflow, and maybe there are other things you could try and do too, but this can lead down a whole rabbit hole of unnecessary conjecture and introduced problems… If we design a system by correctly applying all these terms, we can avoid many issues that we would introduce by unnecessarily "tweaking" things…

Now lets put it all together in the Code, This is the final generalized PID code for our problem

Kp = 0.4

Kd = 0.7

Ki = 0.1

loop:

```
SP = 20
PV = RangeFinderDistance
Error = PV - SP
P = Error                  //Proportional Term
D = Error - Last_Error     //Derivative Term
Last_Error = D             //Save a Copy of this Derivative Term for
                           //   the next iteration
```

I = I + Error

MV = P*Kp + D*Kd + I*Ki

Ship Thrusters = MV

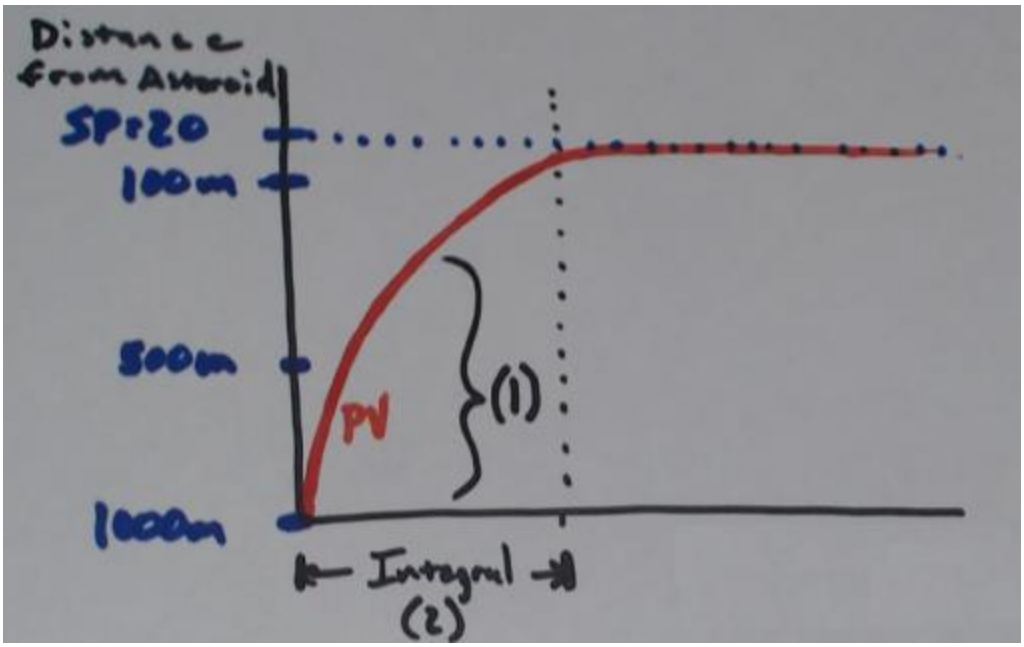
GOTO Loop

And that's it, run the code and hopefully it works after tuning the Coefficients…. Seriously though, if you have made it this far, you now understand more than most people who have ever uttered the word PID Control…..

The Problem of Simply Applying PID to Asteroid Approach Problem and the Solution to Modifying PID to approximate an Ideal Profiled Motion Controller

Alright I already outlined a bit of this on the first few pages. Go read it again, seriously… I'm still waiting… No joke, go read it!!!!!

Now armed with our knowledge of **PID**, we can actually clearly see the problem. There are 2 Primary Problems that PID without profiling causes for us, and there is 1 more problem inherent to the system of Asteroid Approach that is independent of PID. Lets look at the PID issues first.



(1) **TOO much acceleration:** Look at (1), PID wants to keep accelerating you. No matter how low of a Kp you select, the proportional term mathematically won't make sense at the beginning of the flight when compared to the end of the flight. You could try to create a bias, but then that would get in the way of my solution for problem 3, which I haven't mentioned yet. The "proper" solution to this problem would involve a motion profile, remember that trapezoid "curve" picture from the 1st couple of pages, constant acceleration -> constant velocity -> constant deceleration. But as I mentioned, I wasn't interested in all the overhead and added complexity…

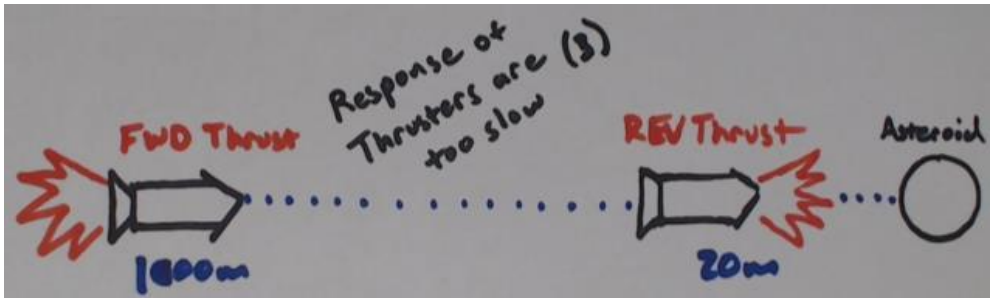
Easy Solution: Limit the effect of the P term for the first x number seconds in order to slow our acceleration in the beginning of the flight. This solution is the easiest to implement given the yolol constraints and you'll see it in code later… THIS IS HOW WE FLATTEN THE CURVE!!!!

(2) WAY TOO much Integral: Why are we accumulating an offset for a steady state error when we are 500m or 900m away from where we

need it. Remember the purpose of the **I** term is to “search” for an offset value that makes up the remainder value that Proportional and Derivative only cannot find. This can also be called a bias. In other words, the **I** term is accumulating errors in the hope of finding a bias that leads to an **Error = 0**. All we are doing by performing any amount of **Integration** back at 800m or 500m all the way to 50m or 30m is rolling a giant snowball down a hill that is numerically going to overwhelm our manipulated variable..

Easy Solution: Only use Integration for the last 5 to 10meters of the flight. Again, all our solutions have to be simple given the constraints of yolol, 70 char per line and 200ms execution time of each line. We don’t have much room to work with and we need our control loop to execute as fast as possible. Currently with 2 lines, that’s 0.4seconds per iteration. When I have created industrial process control designs in the past, I generally execute control loops like this 1000 times per second, 2.5 iterations per second is just silly, I’m shocked this works at all!!!!! But that just means our solution has to be as good as possible to pull it off….

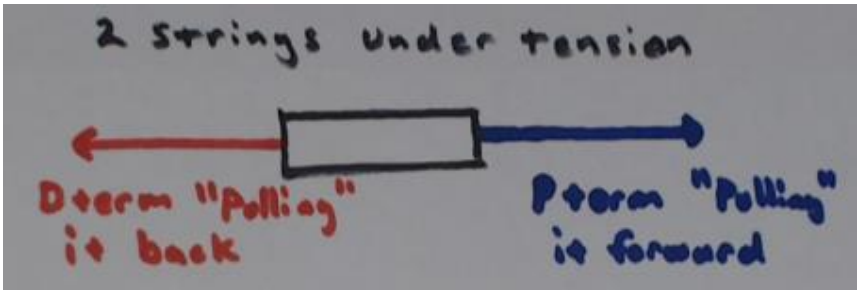
Problem 3 – Our thrusters are WAYY too slow to respond: Have you tried approaching an asteroid manually, it’s rough. It’s even harder for a numerical computing system to do it.



But there is a SUPER EASY way to increase the responsiveness of our thrusters, while leveraging the **P** and **D** terms of our **PID** algorithm… I hinted at it earlier….

Remember, if I'm trying to make something move with PID. The **Proportional** term is like hitting it with a hammer to get it moving while the **Derivative** says: "But not too fast!!!".

Instead of doing the above numerically **FWDThrusters = P+D**, BREAK IT UP… **FWDThrusters = P term REVThrusters = D term.** Let the physical system do the adding for you. A good analogy of this is a sled being pulled by 2 strings under tension. As long as P > D we will get a very well controlled forward motion.



This means our thrusters are already thrusting at a value capable of producing a change in the system!!! We have almost instant responsiveness anytime a thruster value changes!!!

OK SO A LOT OF TALK, WHERE IS THE CODE!!!!!!

```
s=17 m=500 ib=5 t=20 p=0 i=0 x=ib+s u=t g=0.001
if :Ap*(:M<m) then goto3+:insideSafeZone end goto1
Kp=0.056 Ki=0.001 Kd=5.4 h=0.50 l=1-h p=:M-s goto5 //NSZ Conservative
Kp=0.080 Ki=0.001 Kd=5.8 h=0.50 l=1-h p=:M-s goto5 //SZ  Aggressive
e=:M-s d=(p-e)*Kd*h+d*l i+=e*Ki*(:M<x) o=Kp*e+i f=1-t/u t-=1*(t>0) p=e
z/=:Ap*(:M<m)*(d*d>g+e*e>1) :fcuforward=o*f :fcubackward=(d-i)*f goto5
if (d*d>g+e*e>1)==0 then :Ap=0 end :fcuforward=0 :fcubackward=0 goto1

//Improved PID - Approximated Profiled Motion PID
// by: Darkyshadow built and tuned for Ghidorah of HLA
// tips by in game mail are not necessary but appreciated

//Tested Tunings
//Kp=0.056 Ki=0.001 Kd=5.4 h=0.50 //Super Conservative
//Kp=0.080 Ki=0.001 Kd=5.8 h=0.50 //Conservative Bulletproof
//Kp=0.090 Ki=0.001 Kd=5.8 h=0.80 //Slightly More Aggressive
```

Code Explanation:

e=:M-s

Error calculation, Error = PV - SP

d=(p-e)*Kd*h+d*l

Derivative Calculation, D = Last_Error – Error, this has been reversed in order to flip the polarity. This let's us apply the value to **fcubackward** without the need to put a minus sign in front, save a character. The **Kd** is the tuning coefficient. ***h+d*l** is a low pass filter. Specifically an Exponential Moving Average filter. I'm using this here because since we are applying the D term directly to the thrusters, there is a tendency for the D term to have fast changes. This tendency is called "Derivative Kickback". To help mitigate this, a filter really helps, but it also slows the response of the D term. That's why **h** is a tunable value as well… Higher values of **h** cause a faster response at the cost of slightly more Derivative Kickback, whereas lower values of **h** cause a much smoother application of reverse thrust. The Exponential Moving Average filter

was chosen because of it's low computational complexity and can be implemented with 1 variable instead of an array/queue that's generally required for a moving average.

i+=e*Ki*(:M<x)
Integral Calculation, we only want the integral to be used for the last **ib** meters of flight, x is calculated from **ib.** You may be wondering why we don't have any windup protection, clamping or other. Well… 1) we don't have the code space for it, 2) by accumulating the **error** already multiplied by **Ki**, we are already reducing the need to clamp by A LOT, 1000 times if Ki=0.001, 3) If we use I properly, meaning we only engage the **I** term during the last 5-10m of flight, Overflow or Underflow should be next to physically impossible. Basically if we need to clamp, something really really really bad has happened, and again… we don't have the space for it.

o=Kp*e+i
Manipulated output calculation, This is the manipulated output applied only to fcuforward. It only needs the Proportional term and the Integral term. I is not multiplied by Ki because it was already taken into account when I was calculated previously.

f=1-t/u t-=1*(t>0)
This is our Softstart or Early Acceleration Limiter. This is what allows us to flatten the curve of the steep acceleration we would have early on at the beginning of flight with a PID only. We keep a countdown **t** of the # of iterations we would like to softstart for. This gets calculated into an inverted percentage that when multiplied to our Manipulated Variables will provide a slow linear increase of applied Manipulated Output to our thrusters. This prevents an initial jerk that would be experienced in the beginning of flight. This also makes Kp responsible for max velocity during the middle of flight, where we "should" have relatively constant

velocity. Again… This whole system is an approximation to a Profiled Motion PID controller, it's not exactly one =)

p=e
Save the Current Error so that on the next iteration it becomes the previous error (Last_Error).

:fcuforward=o*f :fcubackward=(d-i)*f
Here we apply the Manipulated Variable/s to the thrusters responsible to affecting change of our system. The **fcuforward** gets **o** which has **Proportional** and **Integral**. The **fcubackward** gets **Derivative** and **Integral. Integral** need to get subtracted from fcubackward, because when I is positive, it means we need to go forward, and if it is negative it mean we need to go backward, i.e a negative value means apply positive thrust to reverse thrusters. The ***f** is the percentage from our Early Acceleration Limiter.

There is one line that I can't take credit for and I think is BRILLIANT…

(d*d>g+e*e>1)
It's the done condition for when we are at the end of the approach and our ship is no longer moving… **d*d**: make derivative position, **e*e**: make error positive. **g = 0.001** is a really small value. If the error is greater than **1** we are not at the setpoint yet, if the derivative is greater than 0.001 then we are still moving! And the + acts like an OR operation when used in the full line **z/=:Ap*(:M<m)*(d*d>g+e*e>1)** because a divide by zero causes a failure to execute and the line of code skips…

This is a great line that I took from another player named: **Whitestrake**

I want him to get credit here because he put in a lot of work into an earlier version of a PID Approach and made it open source. His version was what caused me to make this version… Thanks!

Installation:

Make sure to set up your Range Finder's Fields

RangeFinderOnState can be renamed to Ap

RangeFinderSearchLength can be maxed at 1000, note that Approach will not begin until you within the m range

RangeFinderDistance should be renamed to M

This script assumes a transponder has been installed so that you can have 2 different tunings on line 3 and 4 for whether you are inside the safezone or out of the safezone. If there is no Transponder present on the ship then you should remove the ":insideSafeZone" on line 2 with "0", alternatively you can just delete the entire "+:insideSafeZone".

Tuning Parameters: Only Change These Values on lines 1, 3, and 4

I have placed these in order of importance for you to get your desired results

s=17 Setpoint, what distance from asteroid you want to approach to

m=500 Required minimum distance to asteroid that approach will work/begin

Kp=0.06 Coefficient of Proportional term, increase for a faster more aggressive approaches

Kd=5.8 Coefficient of Derivative term, need to increase this when increasing Kp. Remember that Kd is the opposition to Kp, like my sled being pulled from both sides to keep it controlled while moving forward analogy. If you don't fly forward when initially testing, you need to reduce this. If you overshoot your target, you need to increase this.

H=0.5 This value should always be between 0 and 1. Lower values reduce oscillations in the d term at the cost of losing responsiveness. Higher values cause the d term to be more responsive. For more aggressive tunings you will have to use a higher value here. If you see your rev thrusters pulsing too much during

flight and especially if it causes you to veer off throughout the middle of the approach, decreasing this will help at the cost of not being able to user larger Kp values. This value affects a lowpass filter that has been applied to the D term on line 5. If unsure keep between 0.5 and 0.8.

t=20 Time spent to limit/rampup acceleration, in multiples of 0.4s, i.e. 20=8sec reducing this number will cause a faster initial rampup. If you veer off course during the initial start increasing this number will more gracefully start the approach. Think of this as a softstart.

ib=5 Distance within (meters), where Integration will begin to be performed. Another way to decrease the amount of time that you spend centering on the setpoint at the very end is to start the integration a little sooner, you could try changing this to 6,7,10 etc…

Ki=0.001 Coefficient of Integral term. This term is what performs the end centering on the setpoint. If you feel that at the end it takes too long to hit the setpoint, you can try increasing this to 0.002 or even further