How to Land an Orbital Rocket Booster with kOS

Jul 24, 2024 • Christopher Kalitin

Tell me where I'm wrong or just give compliments here.

Read the code here.

One of the reasons Casey Handmer cites when telling people to write blogs is that they are proof of work. Many of my early projects will not be impressive at all, but it's worth documenting it for a few reasons. (1) To share my thought process throughout the project. (2) To force myself into documenting it. (3) To be able to reflect in the future. Forcing yourself to document something means it's much harder to be satisfied with the shitty way of solving a problem. This is why everyone should write a blog.

Can't think of a good way to write this, so I'll just detail all the mistakes I made and stupid things I did. You have to start somewhere!

I could write many thousands more words about this about the suicide burn, flight phases, printing, pitch mulitplier, and more. But all those things are less interesting and not what I want to cover. I'm not going to explain the simple stuff, this isn't a tutorial.

Initial Aero Control Approach

```
// Get distance between two positions without considering the altitude
// Eg. LatLngDist(V(SHIP:GEOPOSITION:LAT, SHIP:GEOPOSITION:LNG, 0), V(-0.09729775,-74.55
function LatLngDist {
    // Only x and y are used for lat/long. z is to be ignored
    Parameter pos1.
    Parameter pos2.

    // 10471.975 is the length of one degree lat/long on Kerbin. 3769911/360
    return (pos1 - pos2):MAG * 10471.975.
}

// Return direction to position in degrees starting from 0 at north
function DirToPos {
    // Only x and y are used for lat/long. z is to be ignored
    Parameter pos1.
    Parameter pos2.

    SET diff to pos2 - pos1.

    // atan2 resolves arctan ambiguity (ASTC quadrants)
    // Reversing x and y to rotate by 90 degrees so we start at 0 degrees at north, usua
    SET result to arcTan2(diff:X, diff:Y).

    // Keep degress between 0 and 360
    if result < 0 { SET result to result + 360. }

    return result.
}

// Return east/west and north/south components of velocity
function GetVelocityInCompassDirections {
    // https://www.reddit.com/r/Kos/comments/bwy79n/clarifications_on_shipvelocitysurfac
    SET vEast to vDot(ship:velocity:surface, ship:north:starvector).
    SET vNorth to vDot(ship:velocity:surface, ship:north:forevector).
    return v(vEast, vNorth, 0).
}
```

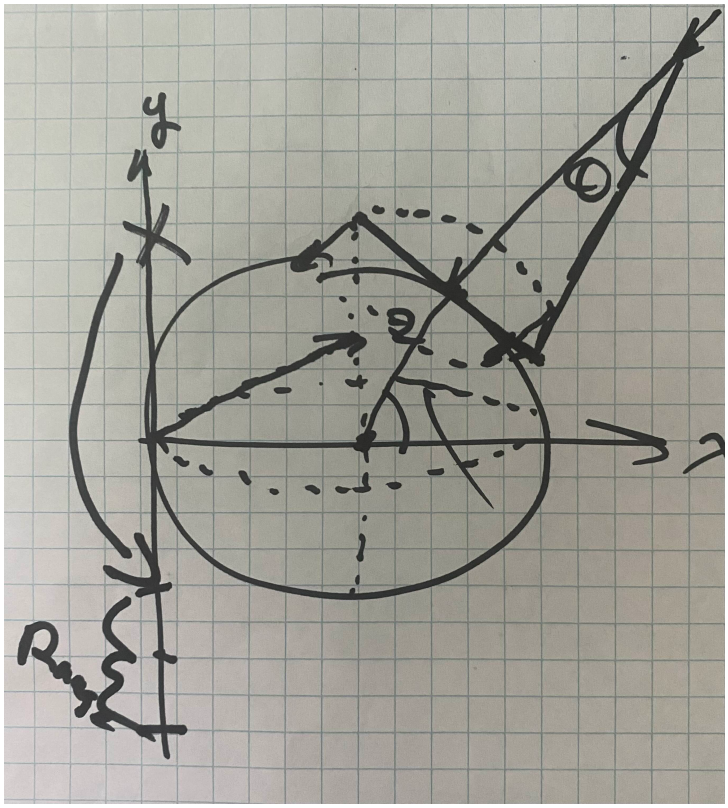
When I first attempted to land a booster with Kerbal Operating System about 2-3 years ago I got stuck trying to implement aerodynamic control. So, this is where I started.

In the intervening years I became a much better programmer and quickly implemented the helper functions above, kids stuff. The principle is to get the direction from the target point to the impact point and pitch in that direction depending on what the required change in distance per second is.

The horrible problem emerges when you realize that the DirToPos function returns the direction from one point to another on the surface of Kerbin. This value is then used as the bearing (degrees relative to north) for the booster.

Anyone who has attempted to land a booster in KSP knows that to adjust your landing site you have to adjust your pitch relative to the retrograde vector. The method above pitches relative to north (bearing relative to north, pitch is used as “amplitude”). At the scale I was testing at (~5000m above the Kerbal Space Center), this problem was not obvious. However, when attempting to reenter after a boostback burn the error became clear.

Failing to Clamp Angle Relative to Retrograde



In kOS (Kerbal Operating System, the scripting mod/language), you control the direction of the craft by inputting a heading which consists of a bearing and pitch value. This is easy to conceptualize for beginners as it’s similar to the Nav Ball in KSP. However, what I didn’t realize fast enough is that you don’t want to do any operations on the heading. There’s a reason why you use heading as a pilot and not when learning trig.

Imagine you're trying to clamp your booster's pitch to within 10 degrees of the retrograde vector. You only have the heading (bearing, pitch) value to work with. Pitch = 0 when straight up, bearing = 0 when pointing north. Stop and think how you would do it. Don't be lazy, do it.

Well if you're rocket is horizontal, clamping the raw bearing and pitch values will work great. A 10 degree offset in either bearing or pitch will result in a displacement of equal magnitude. However, if you're not horizontal (on the equator), a 10 degree change in bearing will result in a smaller displacement than a 10 degree change in pitch. This is the same reason why Vancouver is rotating around the Earth slower than Equador. Hopefully the diagram above makes this as clear as the diagram should be. If you increase pitch (closer to a pole), a single degree of bearing becomes shorter. When you're point straight up, a single degree of bearing is 0.

This problem took a few days to solve (because I don't know that much math, university solves this) and it all stemmed from the initial approach I used for heading control.

tl;dr this is very stupid:

```
function GlideToTarget {
    local aproxTimeRemaining to (SHIP:altitude - TargetPosAltituide) / (SHIP:velocity:su
    local targetChangeInDistanceToTargetPerSecond to impactToTargetDistance/aproxTimeRem

    // If impact dist < 50, do fine control that asymptotically approaches the target (b
    local pitchMultiplier to targetChangeInDistanceToTargetPerSecond * 2.
    if impactToTargetDistance < 50 { SET pitchMultiplier to (impactToTargetDistance^1.6)

    if RetrogradePitch > 70 AND ship:velocity:surface:mag < 450 { SET bearingLimit to 36
    else SET bearingLimit to pitchLimit.

    local shipDirToTarget to impactToTargetDir - 180 - RetrogradeBearing.
    local bearingAndPitch to GetBearingAndPitchFromDir(shipDirToTarget, pitchMultiplier)
    LOCK STEERING TO HEADING(bearingAndPitch:x, bearingAndPitch:y).

    // If retrograde pitch is nearing straight up, behaviour is not correct, so, lock to
    if RetrogradePitch > 80 {
        LOCK STEERING to HEADING(shipDirToTarget + RetrogradeBearing, pitchMultiplier).
    }
}
```

How to Properly Control Heading

Hopefully it's clear that bearing and pitch are horrible values that only pilots should ever use. In

math class you use Eulers, Cartesians, etc. for a reason. The solution is to convert the bearing and pitch to a more suitable rotation system, clamp it, then convert back to bearing and pitch.

A very important insight I heard about learning to code is that half your time should be spent coding and the other half should be spent reading code. There are tons of people who've solved the problem you're working on, it is far more efficient to learn from them. This doesn't mean copy and pasting code, but truly understanding the problem and the solution.

I went back to the video that prompted me to try kOS again and found the solution. Turns out Donies did things the shitty way instead of truly learning (well everyone starts somewhere) and copied the code Edwin Robert wrote here (Video).

```
// I overengineered for 5 wasted days, this is the solution from: https://github.com/Dor
// This functions steers the ship relative to retrograde towards the target position, it
function GetSteeringRelativeToRetrograde {
    local Parameter pitchMultiplierLocalSteerRetrograde. // Local variable naming like t

    // Retrograde vector is in the SHIP-RAW Reference Frame https://ksp-kos.github.io/KC
    local retrogradeVector to -ship:velocity:surface.

    // :position converts from latlng to SHIP-RAW reference frame
    // Refactoring needed to minimize transforming values like LatLng
    local targetVector to ImpactPos:position - LATLNG(TargetPos:x, TargetPos:y):positior
    local targetDirection to retrogradeVector + targetVector * pitchMultiplierLocalSteer

    // If relative angle is too high, limit it.
    // Normalize the vectors, then multiply the target direction by the tan of pitch lim
    local angleDifference to vAng(targetDirection, retrogradeVector). // Angle of two ca
    if angleDifference > PitchLimit { SET targetDirection to retrogradeVector:normalizec

    return lookDirUp(targetDirection, facing:topvector).
}
```

Converting to the SHIP-RAW reference frame is the key. This allows for standard operations to be done on the rotation vectors and to use functions included in kOS like vAng.

The vAng function abstracts away some concepts I don't yet understand. Without it I would've had to study trig for a few weeks to properly implement it. Projects like this are great because they clearly show the extent of your knowledge. "retrogradeVector:normalized + targetDirection:normalized * tan(PitchLimit)." makes perfect sense to me and I can draw the diagram for you, but what goes on inside vAng is a mystery for now.

Failed Refactor

From the use of periods instead of semi-colons and other quirks like “local Parameter”, you might be able to tell that kOS is not a language meant for programmers, but rather for KSP players. This means a new mental framework is required to use kOS efficiently.

When implementing the intial helper functions and first attempt at landing I decided to use the techniques I was aware of and refactor in the future. This is immensely stupid and leads to a lot of wasted time that you will have to deal with in the future. Just write the code properly the first time.

The problem with the existing code is it used SET instead of LOCK on variables. In kOS you can declare a variable the way you’re used to by using SET. LOCK is used to update the value of a variable every physics tick. kOS is obviously supposed to be used with LOCK and my attempt at a refactor changed the code to use this different paradigm. The fundamental solution here is - of course - to rewrite kOS to be a proper language, but I ain’t doing that.

This refactor took more time than expected because I had to port the code to an entirely different execution paradigm.

Previous SET paradigm:

1. Create an infinite “UNTIL false” loop and keep a variable to track flightPhase.
2. Depending on the current flight phase, execute the appropriate function.
3. Break the loop when we’ve landed.

```
// directionError is SET in OrientForBoostback()

UNTIL false {
    if NOT ADDONS:TR:HASIMPACT { LOCK THROTTLE TO 0. CLEARSCREEN. BREAK. }

    UpdateFlightVariables().

    if flightPhase = 0 {
        PRINT "Flight Phase: Orient For Boostback (1/6)" at (0, 0).

        OrientForBoostback().

        PRINT "Flight Variables: " + numberHere at (0, 2).

        if directionError < 30 { StartBoostbackBurn(). }
    }
```

```
}

function StartBoostbackBurn {
    LOCK throttle to 1.

    SET flightPhase to 1.
    CLEARSCREEN.
}
```

New LOCK paradigm:

1. Lock global variables that are needed very often.
2. Call the first flight function (OrientForBoostbackBurn()).
3. Inside OrientForBoostbackBurn(), lock the appropriate variables and wait for completion condition.
4. When the completion condition is met, call the next function.

```
LOCK ImpactToTargetDir to DirToPoint(ImpactPos, TargetPos).

OrientForBoostbackBurn().

function OrientForBoostbackBurn {
    LOCK TargetHeading to Heading(ImpactToTargetDir, 0).

    WAIT UNTIL vAng(targetHeading:vector, ship:facing:vector) < 30 { BoostbackBurn(). }
}
```

The code above was my first attempt at the refactor. It failed because when WAIT UNTIL is called, all other execution stops. In the previous SET paradigm, I used WAIT(0.1) to control the tick speed (Which itself is flawed because the code needs time to run, so Hertz is actually <10). In the new LOCK paradigm, "WAIT UNTIL(completion condition)" simply pauses the program until the condition is met, which is never because it is never updated. Apart from this glaring flaw, this approach also doesn't allow printing variables continuously.

The solution is to add a loop (eg. 10Hz) to the end of the flight functions with this line: "WHEN (completion condition) { RunNextFlightFunction(). }". The loop can also be used to print variables continuously or kOS GUI widgets can be used (better and proper). This will also make the code far more readable. I would've done this but I was on week 3 and wanted to finish the project, maybe will in the future when bored.

No Unifed Solution To Cancel Horizontal Velocity and Minimize Landing Error

In the second part of the video at the top of this post, you can see the booster landing with the UI active. Unlike the first cinematic landing, this one barely makes it onto the landing pad.

The approach I implemented to have a soft touchdown has two phases. First, the Suicide Burn is started and it targets a point ~30 meters away from the landing pad in the opposite direction of the rocket. Second, when the rocket is <40 m/s or <25m altitude, the SoftTouchdown() function is called. This cancels out horizontal velocity and slowly decreases vertical velocity until touchdown (lerp between 10 m/s to 2 m/s, t=altitude/50). While the suicide burn is performed, the horizontal velocity changes and by extension the landing location. This is why aiming ~30 meters off is necessary in the beginning (shitty solution).

```
// Extra code not included, this gets the point across
function StartSuicideBurn {
    local magnitude to -(GetHorizationVelocity():mag^1.67) / 45. // Offset by multiple c
    SET TargetPos to AddMetersToGeoPos(targetSite, GetOffsetPosFromTargetSite(magnitude)
}

function SoftTouchdown {
    local t to TrueAltitude / 50.
    SET TargetVerticalVelocity to Lerp(-2, -10, CLAMP(t, 0, 1)).

    local aproxTimeRemaining to (TrueAltitude - TargetPosAltitude) / (SHIP:velocity:surf
    SET aproxTimeRemaining to CLAMP(aproxTimeRemaining, 5, 10). // Clamp to 10 seconds,

    local pitchMultiplier to Lerp(0, pitchLimit, CLAMP(GetHorizationVelocity():MAG/3, 0,
    LOCK STEERING TO HEADING(RetrogradeBearing, 90 - pitchMultiplier, 0).

    local baseThrottle to SHIP:Mass/(SHIP:MAXTHRUST / 9.964016384)-0.02. // Hover, Kn tc

    local vertVelError to TargetVerticalVelocity - GetVerticalVelocity().
    local throttleChange to CLAMP(vertVelError^1.7/50, 0.01, 0.25) * (vertVelError/ABS(v

    LOCK throttle to CLAMP(baseThrottle + throttleChange, 0, 1).
}
```

This approach has a poor success rate. With our two data points in the video, only 50% make it to the inner circle of the landing pad. A unified solution that both cancels horizontal velocity and minimizes landing error is needed.

There is a shitty solution here that many people have used. You can do an entry burn to cancel horizontal velocity far above the landing site, then land. This approach is shitty because it's unrealistic, uses extra fuel, and is avoiding a really fun problem.

I imagine the solution is to track the estimated net displacement in landing position during the Suicide Burn. With the estimated time to touchdown, current pitch, and current horizontal velocity you could approximate the net displacement. Add this to the target landing location and it should be a much more accurate landing. However, even this is a slightly shitty solution, maybe Rafael (Best kOS landing script I've ever seen) knows the right way.

The Fundamental Insight

The fundamental insight I learned in the past month of doing this project and watching Deep Learning lectures is that to build things properly you need sufficient knowledge of the underlying fields. The most important insights are often the most obvious ones, truly understanding and applying them is the important step. This project could have taken a few days if I was good at rotation/vector math and knew more about GNC. Skill acquisition is the most important goal all young people must have. After you've acquired the skills, building becomes exponentially easier.

"I may not be able to do it the good way, but I sure can do it the shitty way." Don't be a lazy fuck, you won't be a good programmer (or actually good at anything) by doing things the shitty way.

If I had good skills, I would've written the code like this (Video). I have a ton of respect for the man that wrote that code. He didn't do it the shitty way.

Subscribe

CKalitin
x.com/CKalitin

Geohot made a blog too. You should be working on hardware