Modelling and Control of a Self Landing Rocket

By (Abbott) Run Bai

May 2025

Contents

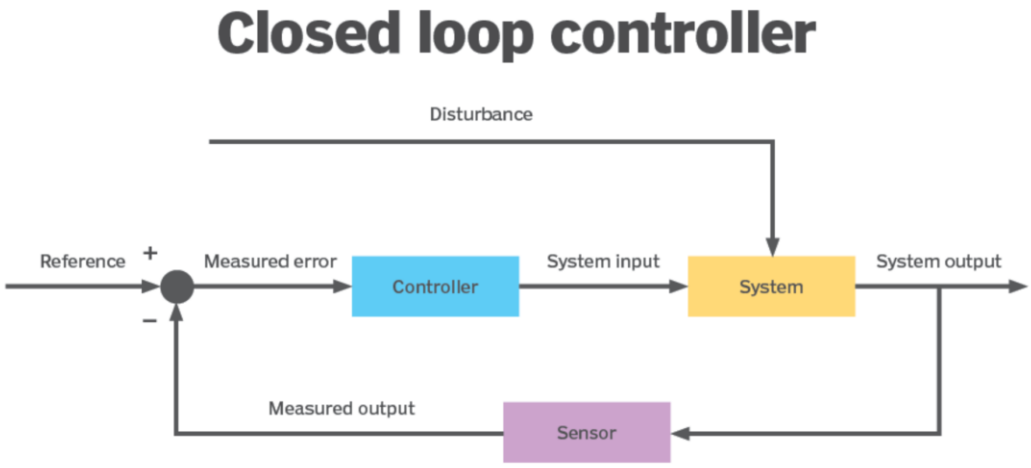
Introduction 3
Github 4
Controllers 4
PID controller 4
Creating a simulator 6
Results 8
LQR controller 9
Overview 9
State space equation 9
Cost function 10
How to deal with a constraint when differentiating? 12
Minimizing cost function 13
Determining state variables 17
Results from simulator 19
MPC controllers 20
Overview 20

Introduction

Half a year ago I built my first prototype rocket, with 4 fins controlled by 4 servos at the tail of the rocket. The rocket had a sensor inside, which had a gyroscope and accelerometer. I used the gyroscope's ability to measure the rate of rotation to calculate the tilt of the rocket and used the accelerometer to reset the values to prevent drift when the magnitude of the downwards acceleration is 9.8 $m \cdot s^{-2}$. After processing the data from the sensor my rocket would change the fins to the same degree as the tilt. Anyone experienced with basic rocketry could probably guess the outcome of the launch.

The rocket swayed uncontrollably in the air and eventually crashed, making me ponder about my life choices but more importantly, control theory…

My initial prototype failed due to a number of reasons, it was trying to react to the error, instead of making a prediction of how the rocket will behave. Hence the plant can become unstable or oscillatory.



[1]

In this paper I will discuss various algorithms and methods involved in controlling a rocket's descent. I will create different models which uses a closed (aka feedback) loop to control the plant (the rocket). I will describe

[1] What is a closed loop control system and how does it work?

the motion of the plant using state space equations, and feed them into a controller, which adjusts the attributes of the plant in order to achieve a desired outcome (land). I will also create a python simulation to test and improve upon my model. The python model will not have a sensor and will mainly be based on the controller and plant, because all external disturbances are accessible via variables.

Github

Before we get started, note that I only provide snippets of my simulator in this paper to limit the overall size. However if you want to test it out yourself, you can find the full version of my code through my github: https://github.com/AbbottBai/self_landing_rocket_control_software

Controllers

Controllers are an essential component of a closed loop system. They directly influence how the plant adjusts to external changes/forces. Controllers take in the state variables of the system such as the current velocity, displacement, tilt, etc... and then outputs a suitable control signal based on the error (how far away the current state is from the ideal state).

There are 2 main types of controllers, classical and modern. This section will cover the description and implementation of PID and LQR controllers and it will briefly touch on MPC controllers in self landing rockets.

PID controller

The PID (Proportional –Integral – Derivative) controller is a type of classical controller, it is the controller that is closes to the one used in my first failed prototype (see introduction). The PID controller does not use state space equations nor complex physics equations to predict how the system will behave and choose the most optimal path, instead it compares the current attributes/readings of the plant (from a sensor) with the reference (ideal

attributes/readings) and produces an output to adjust the plant accordingly. This is pretty similar to ‘changing the angle of fins based on the tilt’ method in my failed prototype, however it includes some extra maths to reduce oscillations/spiralling out of control. PID may be able to land a rocket vertically, but it can't land it in a specific location as well as other controllers such as LQR or MPC. This is because although you could have one PID controller per state variable, it is not able to find the relationship between different state variables and different inputs, which are all interconnected in the real world. Also it would become a nightmare to tune the parameters for PID controllers if multiple state variables are in play.

PID controllers can only be implemented on SISO (single input single output) systems, therefore it does not make use of matrices (which will be covered later).

Here is an equation which defines the error, which we feed into the PID controller:

$$e(t) = r(t) - y(t)$$

Where e is the error, r is the reference and y is the sensor reading.

Now we take the error and feed it into the general PID controller formula:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau), d\tau + K_d \frac{de(t)}{dt}$$

The desired closed loop dynamics can be achieved by iteratively testing and updating the K parameters. And as the name suggests, the PID controller equation purely consists of the sum of the error, integral of error and the derivative of error.

The proportional term is included at the front so a system which is far from the desired state lead to a more vigorous correction.

The integral is used to enlarge the effects of an error which may be small, but persists for a long period of time. It introduces ‘memory’ to the algorithm.

The derivative is used to highlight the rate of change of the error, so an error which is changing quickly will cause the controller to compensate more. But a slow drift in tilt won’t cause the controller to jolt the system back, which may cause oscillations or send system into chaos.

The proportional, integral and derivative term combined provides a smoother transition for the states of the system, instead of hard coded reactions that spins out of control.

Creating a simulator

I coded a 2D simulator (see github subsection from introduction for full code) to demonstrate the implementation of PID controllers in rockets, taking into account the tilt and vertical descent velocity. Since the PID controller equation is identical mathematically for velocity and tilt, I will demonstrate the logic and code behind the tilt only.

```
10  def ut(dt, e_theta, prev_e_theta, i_theta, utI_reset):
11      if abs(e_theta) <= 1 and utI_reset == False:
12          # if the error is small, it will reset the integral to allow small updates
13          # But only once to allow re-build up of small error
14          i_theta = 0
15          utI_reset = True
16      else:
17          i_theta += e_theta * dt
18          if i_theta > 20:
19              i_theta = 20
20          elif i_theta < -20:
21              i_theta = -20
22
23      d_theta = (e_theta - prev_e_theta) / dt
24
25      u_theta = k_theta[0] * e_theta + k_theta[1] * i_theta + k_theta[2] * d_theta
26      return i_theta, u_theta, utI_reset
```

The function, which produces an output u, is called ut. And it takes in parameters dt, current tilt error, previous tilt error, integral of tilt error, and integral of tilt error reset Boolean.

Before diving into the inner workings of the function, we should note that the integral term will accumulate a large error during the early stages of activating the controller. This is called integral windup, and can be fixed by resetting the integral term when the error reduces, thus reducing the chance of overshooting and oscillating about the optimum state. I have implemented this by using an if statement to detect if the absolute value for the error is smaller than one and if the integral has already been reset in the previous iterations. This is so that if the absolute value of error is below one, it is still able to accumulate overtime so that the controller notices it and tries to compensate instead of ignoring errors that are too small.

I have also clamped the integral term (limiting the upper and lower bounds), so that it does not grow indefinitely in size and overshadow the proportional and derivative terms.

dt is passed in from the main loop, which detects the time elapsed since the last iteration and feeds it into the function to be processed.

i theta is returned into the main loop so in the next iteration, it is not reset and will continue accumulating from the last iteration.

```
angular_acceleration = u_theta / mass
angular_velocity += angular_acceleration * dt
# += is used instead of = because you want the current velocity, which is a result of all previous velocities added todether.
d_angle = angular_velocity * dt# total change in angle from start would use += instead of =
current_angle += d_angle
current_angle *= 0.99# Damping factor, prevents infinite oscillation

u_vel += mass * 9.8# u alone wasn't strong enough to overcome gravity, so I have implemented a baseline thrust
acceleration = u_vel / mass
d_velocity = acceleration * dt
current_velocity += -9.8 * dt
current_velocity += d_velocity
current_velocity *= 0.99# Damping factor, prevents infinite oscillation
```

This code is taken directly from the main loop. Note that the output of the PID controller, u, does not directly correspond to the change in angle or velocity, this is because the PID controllers outputs a value (force, acceleration, etc...) that the actuator can apply. It does not directly set angle or velocity, those are state variables that change according to the actuator's output.

Results

The readings are taken once every 20 iterations through the main loop so the output data fits in this document.

```
C:\Users\21304\AppData\Local\Microsoft\WindowsApps\python3.1
Velocity: -265.58509488096706, angle: -20.35103226257266
Velocity: -167.48721799747813, angle: -13.169807955464591
Velocity: -105.6002847468851, angle: -8.171641478566384
Velocity: -66.58167087172059, angle: -4.689013549504915
Velocity: -41.959786149834606, angle: -2.2514278941425867
Velocity: -26.446247189164865, angle: -0.5522345263772452
Velocity: -16.65266030817597, angle: 0.6041289526725085
Velocity: -10.480133920180807, angle: 1.3609198600226995
Velocity: -6.580706314626046, angle: 1.855452541614648
Velocity: -4.12575520847022, angle: 2.143676978427847
Velocity: -2.5770261605242717, angle: 2.2938864238774044
Velocity: -1.6010666748931288, angle: 2.3362623788324477
Velocity: -1.0272959356066502, angle: 2.2998414427002136
Velocity: -0.6696941515848891, angle: 2.2078125371739823
Velocity: -0.44367126433910864, angle: 2.0673683429368954
Velocity: -0.30130479192927284, angle: 1.8892058464838295
Velocity: -0.21208684940107592, angle: 1.6858389856477274
Velocity: -0.15665436567229113, angle: 1.4689045427876573
Velocity: -0.1226624567484742, angle: 1.2455645289360262
Velocity: -0.10252704906680941, angle: 1.0228732363896313
Rocket is in stable descent position

Process finished with exit code 0
```

The angle has overshot a tiny bit, but oscillates back to the desired state because the coefficient K_p of the proportional term may be too large, causing the system to reach the zero state too quickly.

Or this could be because the accumulated error from the integral term continued to push the output beyond the zero state, but this is unlikely as in code I implemented some code which resets the integral when the error is small.

LQR controller

Overview

The LQR (Linear Quadratic Regulator) controller is a type of modern MIMO (multiple input multiple output) controller. It is able to take in multiple state variables such as velocity and tilt into one equation (unlike PID controller where one state variable needs its own equation), and compute an output for all states. It is also capable of drawing connections between different states, for example velocity and height, by using matrix operations.

The system dynamics are described by a linear, time invariant state space equation. Linear means there are no non-linear terms like x^2 or $\sin x$, the output is directly proportional to the input. Time invariant means the system's behaviour does not change with time, applying the same input leads to the same result regardless of time. Of course this is not a very accurate model for actual rockets, where state variables such as mass is constantly changing according to time.

The controller aims to minimize the error, which is described by the quadratic cost function. LQR controllers plans pre-emptively unlike PID controllers, which purely reacts to current state variables instead of predicting future state variables. LQR controllers achieves this by having an integral to infinity for the cost function, meaning the algorithm is able to compute a matrix (K) which drives the cost down over time, eventually ending up at 0, to minimize the overall size of the cost.

LQR performs best in an ideal world: constant system dynamics and no constraints. It computes a value when initiated, so regardless of the time and the plant's current state, will always bring the plant towards the desired state.

State space equation

The state space equation is a first order differential equation which tells you how a matrix containing all state variables will react to changes that the controller applies. The change that the controller applies to the plant will be labelled as $\boldsymbol{U}$. Imagine a very simple rocket system with only two

state variables, position and velocity. Then the state matrix $\boldsymbol{X}$ can be defined as:

$$\boldsymbol{X} = \begin{bmatrix} Position \\ Velocity \end{bmatrix}$$

The derivative of position is velocity, and the derivative of velocity depends on u. So we define a matrix $\boldsymbol{A}$ and apply it to $\boldsymbol{X}$ to find the derivative of $\boldsymbol{X}$.

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{1} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \qquad \boldsymbol{AX} = \begin{bmatrix} Velocity \\ 0 \end{bmatrix}$$

u is defined as a scalar in this case, we multiply it with matrix $\boldsymbol{B}$, which tells us how much influence u has on each state of the system.

$$\boldsymbol{B} = \begin{bmatrix} 0 \\ Some\ number \end{bmatrix} \qquad \boldsymbol{B}u = \begin{bmatrix} 0 \\ Some\ number * u \end{bmatrix}$$

$$\therefore \boldsymbol{X'} = \boldsymbol{AX} + \boldsymbol{B}u + constant(such\ as\ gravity)$$

Cost function

The end goal of the LQR controller is to find a value for u that moves the system closer to a target. This can be achieved by keeping the state $\boldsymbol{X}$ close to the desired state whilst using minimal effort u to preserve fuel and enhance efficiency.

Unlike the PID controller, we will not explicitly define an equation for error, instead we just assume that the desired for the rocket's state variables are 0 (if they aren't, you can always transform them so they are). Therefore our error is just $\boldsymbol{X}$.

However the state variables could be negative, so we square it to reduce negative error. Squaring also penalizes larger errors. So now our error becomes $\boldsymbol{X^2}$.

We introduce a coefficient matrix, $\boldsymbol{Q}$, to our current error so that for example a large tilt is penalized more than a large velocity. Now our error becomes $\boldsymbol{QX^2}$. $\boldsymbol{Q}$ is usually a diagonal matrix, e.g. $\begin{bmatrix}10 & 0\\ 0 & 5\end{bmatrix}$, because we want to apply separate weights to different state variables.

$$\begin{bmatrix}10 & 0\\ 0 & 5\end{bmatrix} \times \begin{bmatrix}Position\\ Velocity\end{bmatrix} = 10\ position^2 + 10\ velocity^2.$$

However due to matrix multiplication rules, we need to transpose $\boldsymbol{X}$ first in order to calculate its square.

So our final error term for the distance to target is $\boldsymbol{X^T QX}$.

The error term for the effort, $\boldsymbol{U}$, follows a similar trend, so it can be defined as $\boldsymbol{U^T RU}$.

We integrate the two errors up to infinity, because we want the controller to compute a matrix which decreases the cost regardless of the time, in other words permanently stabilize the system instead of just for a short period of time. A good controller won't actually integrate to infinity, because the integral would converge to zero over time. Hence we can assemble all the pieces to get our cost function:

$$cost = \int_0^\infty (\boldsymbol{X^T QX} + \boldsymbol{U^T RU})dt$$

A good choice for $\boldsymbol{Q}$ and $\boldsymbol{U}$ ensures that the rocket is able to reach its destination whilst using the least amount of fuel, therefore a rocket's behaviour/performance depends hugely on the choices of parameters.

How to deal with a constraint when differentiating?

This is a prenote for the section below, minimizing cost function.

If we have a function that we want to minimize, $f(x, y) = x^2 + y^2$, but there is a constraint $x + y = 5$. What we might do is to substitute the constraint into the function, then differentiate it to find the minima:

$$y = 5 - x$$

$$f(x) = x^2 + (5 - x)^2$$

$$= 2x^2 - 10x + 25$$

$$\frac{df}{dx} = 4x - 10$$

$$4x - 10 = 0$$

$$\therefore x = \frac{5}{2} \quad y = \frac{5}{2}$$

This is relatively easy, because constraint is not a function, just an equation. However if you have a constraint that is also a function itself, this method may be more troublesome.

Therefore in general, we tackle this kind of problem by using a new function called the Lagrangian:

$$\mathcal{L}(x, y, \lambda) = f(x, y) + \lambda \times g(x, y)$$

Where $g(x, y) = 0$ is the constraint function. λ is like a weight, and it puts emphasis on the importance of the constraint. So if the constraint is not met then λ enlarges the value for the Lagrangian. If it is met then the constraint term becomes zero.

We then calculate the derivative of the Lagrangian equation to get the minimum value for $f(x, y)$ given constraint $g(x, y)$.

Minimizing cost function

To get the rocket to the most optimum state (e.g. to land/hover), we want to minimize the cost. The cost is not a simple function of time, it is a functional. A functional in maths contains a function, so instead of taking a single number as the input, it takes the whole range of the inside function as its domain.

Hence to minimize the cost, we want to find the smallest value that the inside function outputs. So we want to differentiate $\boldsymbol{X^T QX + U^T RU}$. We cannot apply simple calculus rules here, because we have a constraint from when we defined the state space equation above. The constraint is $\boldsymbol{X' = AX + BU}$. Using the rules we described in the section above, how to deal with a constraint when differentiating, we can create a Lagrangian equation to differentiate:

$$\mathcal{L}(\boldsymbol{X, U, \lambda}) = \boldsymbol{X^T QX + U^T RU + \lambda^T (AX + BU)}$$

Note that according to matrix differentiation rules (which will not be covered in detail in this paper), if you have a matrix like $\boldsymbol{A^T BC}$, and you want to differentiate in terms of $\boldsymbol{C}$, then $\frac{d}{d\boldsymbol{C}} = \boldsymbol{B^T A}$.

So we have to differentiate 3 terms:

1. Lets first write out the terms containing $\boldsymbol{X}$:

$$\boldsymbol{X^T QX + \lambda^T AX}$$

Which equals

$$\boldsymbol{QX^2 + \lambda^T AX}$$

Now normal calculus rules apply, see above for matrix differentiation rule.

$$\frac{d\mathcal{L}}{d\boldsymbol{X}} = 2\boldsymbol{QX} + \boldsymbol{A^T\lambda}$$

$\boldsymbol{\lambda}$ tells us how important the constraint is to the cost at an instance, which changes according to the change in $\boldsymbol{X}$, hence $\boldsymbol{\lambda'} \propto \frac{\boldsymbol{d\mathcal{L}}}{\boldsymbol{dx}}$. In reality $\boldsymbol{\lambda'} = -\frac{\boldsymbol{d\mathcal{L}}}{\boldsymbol{dx}}$, but to derive it is way beyond year 2 undergraduate level, so actual derivation will not be covered in this paper. However we still need it for the next steps of the derivation so:

$$-\frac{d\mathcal{L}}{d\boldsymbol{X}} = -2\boldsymbol{QX} - \boldsymbol{A^T\lambda} = \boldsymbol{\lambda'}$$

2. Now lets write out the terms containing $\boldsymbol{U}$:

$$\boldsymbol{U^T RU + \lambda^T BU}$$

Which equals:

$$\boldsymbol{RU^2 + \lambda^T BU}$$

Therefore:

$$\frac{d\mathcal{L}}{d\boldsymbol{U}} = 2\boldsymbol{RU} + \boldsymbol{B^T\lambda}$$

And now setting it to zero for the minima:

$$2\boldsymbol{RU} + \boldsymbol{B^T\lambda} = \boldsymbol{0}$$

Therefore:

$$\boldsymbol{U} = -\frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{R}^{-1}\boldsymbol{B}^{T}\boldsymbol{\lambda}$$

3. Lastly, lets write out the terms containing $\boldsymbol{\lambda}$:

$$\boldsymbol{\lambda}^{T}\boldsymbol{A}\boldsymbol{X} + \boldsymbol{\lambda}^{T}\boldsymbol{B}\boldsymbol{U}$$

After differentiating, we get:

$$\frac{d\mathcal{L}}{d\boldsymbol{U}} = \boldsymbol{A}\boldsymbol{X} + \boldsymbol{B}\boldsymbol{U} = \boldsymbol{X}'$$

Which is the same as our state space equation!

So with these three differential equations in our arsenal, we can move on with the next step. From the second step above, we know that $\boldsymbol{U} = -\frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{R}^{-1}\boldsymbol{B}^{T}\boldsymbol{\lambda}$, but we do not have a value for $\boldsymbol{\lambda}$ yet. We need to include the current state $\boldsymbol{X}$ into the equation for the control variable $\boldsymbol{U}$, so we assume that from step 1 above, $\boldsymbol{\lambda} \propto \boldsymbol{X}$. Hence we can say:

$$\boldsymbol{\lambda} = \mathbf{2}\boldsymbol{P}\boldsymbol{X}$$

There is a 2 in front of the coefficient to simplify our equation for $\boldsymbol{U}$ further during substitution:

$$\boldsymbol{U} = -\frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{R}^{-\mathbf{1}}\boldsymbol{B}^{T}(\mathbf{2}\boldsymbol{P}\boldsymbol{X}) = -\boldsymbol{R}^{-\mathbf{1}}\boldsymbol{B}^{T}\boldsymbol{P}\boldsymbol{X}$$

So if $\boldsymbol{K} = \boldsymbol{R}^{-1}\boldsymbol{B}^{T}\boldsymbol{P}$, then:

$$\boldsymbol{U} = -\boldsymbol{K}\boldsymbol{X}$$

This gives us the control law for LQR controllers. The controller output has a linear relationship with the current state of the plant, which is why LQR is called **linear** quadratic regulator.

Finding P

To find an optimal value for $\boldsymbol{K}$ which will give us a value for $\boldsymbol{U}$ that will minimize the cost, we have to find one unknown variable: $\boldsymbol{P}$.
P is a constant, so:

$$\boldsymbol{\lambda = 2PX} \qquad \therefore \boldsymbol{\lambda' = 2PX'}$$

From $\boldsymbol{U = -R^{-1}B^T PX}$, we can say:

$$\boldsymbol{X' = AX + BU = AX - BR^{-1}B^T PX}$$

$$\therefore \boldsymbol{\lambda' = 2P(A - BR^{-1}B^T P)X}$$

From step 1 of cost minimization, we know that:

$$\boldsymbol{\lambda' = -2QX - A^T\lambda = -2QX - A^T 2PX = -2(Q - A^T 2P)X}$$

So setting the two $\boldsymbol{\lambda'}$ equal to each other:

$$\cancel{\boldsymbol{2}}\boldsymbol{P(A - BR^{-1}B^T P)}\cancel{\boldsymbol{X}} = -\cancel{\boldsymbol{2}}\boldsymbol{(Q - A^T 2P)}\cancel{\boldsymbol{X}}$$

$$\therefore \boldsymbol{PA - PBR^{-1}B^T P = Q - A^T 2P}$$

$$\therefore \boldsymbol{PA - PBR^{-1}B^T P + Q + A^T 2P = 0}$$

And now we have an equation which we can solve with the help of a computer to get $\boldsymbol{P}$, and then substitution into $\boldsymbol{K = R^{-1}B^T P}$ will give us the value of $\boldsymbol{K}$. Therefore we can use the calculated, optimised value of $\boldsymbol{K}$ to

give us the best value for $\boldsymbol{U}$ using $\boldsymbol{U = -KX}$ to manoeuvre our rocket to a state with minimum cost.

Determining state variables

There are 2 types of systems:

1. Static systems – in this type of system the output depends solely on the input, regardless of the previous state of the system. E.g. the voltage across a resistor depends solely on the current.
2. Dynamic systems – in this type of system the output depends on the previous state of the system, plus the current input. The velocity of a car at an instance depends on the previous velocity, and the current throttle position.

State variables are basically the 'memory' of the plant, they represent the previous state of the system. Static systems do not require state variables, because they do no need to 'remember' their past state. Their output is directly connected to their input.

The behaviour/current state of the plant in a dynamic system at $t > 0$ can be found using the **state of the plant at $\mathbf{t = 0}$** and the **input at $\mathbf{t > 0}$** .

The number of state variables is proportional to the number of energy states in the plant. For example, displacement, velocity and acceleration are all linked to kinetic energy. However these are not our final state variables, this is because from Newton's second law we know that F=ma, therefore current acceleration is influenced only by the force, and not the past acceleration. However taking velocity as an example:

$$F = ma = m\frac{dv}{dt} \qquad \therefore dv = \frac{F}{m}dt$$

$$\int_{v_0}^{v(t)} 1 \times dv = \int_{t_0}^{t} \frac{F}{m}dt$$

$$\int_{v_0}^{v(t)} 1 \times dv = v \qquad and \qquad \int_{t_0}^{t} \frac{F}{m} dt = \frac{F}{m} t$$

$$\therefore v(t) - v_0 = \frac{F}{m} t - \frac{F}{m} t_0$$

$$v(t) = \frac{F}{m}(t - t_0) + v_0$$

So the velocity at any point in time depends on the initial velocity and the input (force). Similar steps could be implemented onto displacement, and the result would be the same: both their values at a given time is dependent on their initial values.

Therefore although displacement, velocity and acceleration all somewhat fit under kinetic energy, only displacement and velocity are state variables because they have to be 'derived from themselves' instead of purely on another variable such as thrust.

So coming back to our rocket, it would have kinetic energy, gravitational potential energy and chemical energy whilst in flight. However GPE depends on the height of the rocket, which is included in displacement for KE. And chemical energy is transformed into thermal energy of the gas, which expands and exits the rocket at speed, meaning it eventually transforms to KE. Therefore the only energy store that we should look at for state variables is KE.

From above we know kinetic energy can be split into three main components: displacement, velocity and acceleration. We can immediately exclude acceleration so that we are left with displacement and velocity. However they can be split into different components, because we are concerned with 3 dimensions instead of just one. We can split

velocity and displacement into their translational and rotational components.

So for translational displacement, we would have displacement in the x, y and z axis. For rotational displacement, we would have roll (rotation around x axis, φ), pitch (rotation around y axis, θ) and yaw (rotation around z axis, ψ).

For translational velocity, we would have linear x, y and z velocity. For rotational velocity, we would have angular velocity (ω) in x, y and z plane.

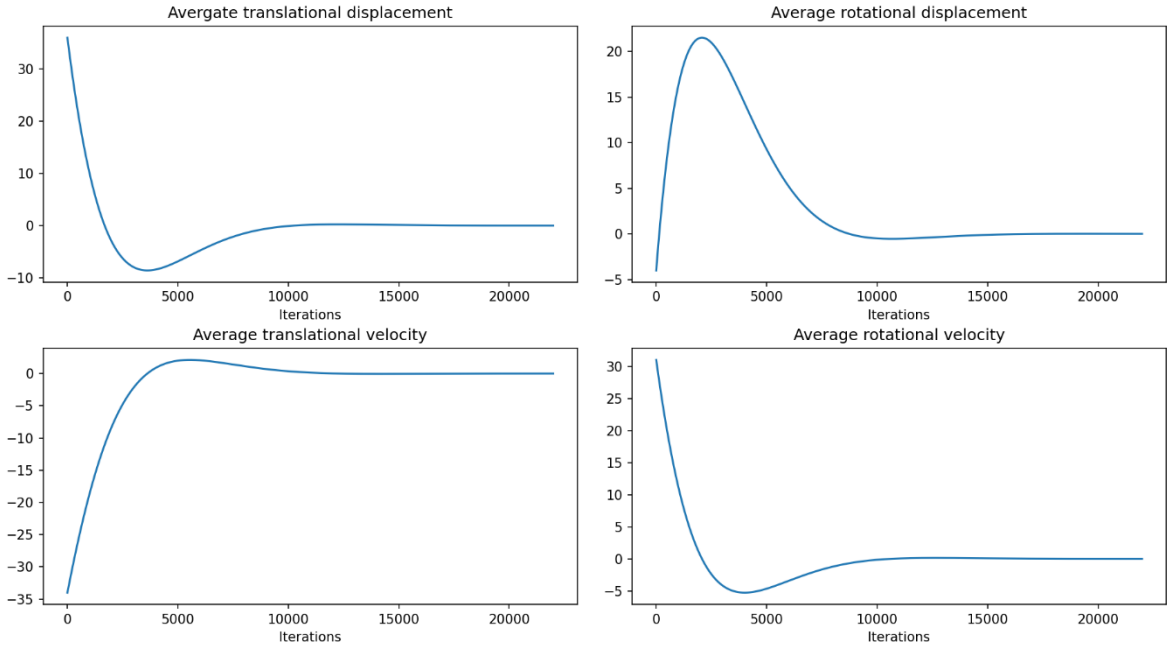
So overall, for a rocket in three dimensions, our state space matrix would look like this:

$$\boldsymbol{X} = \begin{bmatrix} S_x \\ S_y \\ S_z \\ \varphi \\ \theta \\ \psi \\ v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

Results from simulator

I will not include any of my actual code for the LQR controller in this paper, because I have already gone through how to find $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{Q}, \boldsymbol{R}$ above. Additionally, the formula we derived to solve for $\boldsymbol{P}$ is too complex to solve by hand, therefore there are python libraries that facilitate and optimize this step. But if you are interested, you can access the full code on my github (please see github subsection from introduction for the link).

I built a 3D simulator, with the matrix containing state variables, $\boldsymbol{X}$, being identical to the one I described above. Here are the results:



These graphs only shows the average displacement/velocity for the x, y and z axis as otherwise we would need 12 graphs, one for each state variable. The library that I used to plot these graphs, matplotlib, automatically scales the axis based on the x and y values, so 0 on the y axis is in different places.

As you can see, the randomly initialized state variables of the rocket eventually moves towards 0, where error is at its minimum. There is a bit of oscillation, but the controller quickly fixes it afterwards.

MPC controllers

Overview

MPC controllers logically similar to LQR controllers, they both try to minimize the error based on how the plant will react to the controller outputs.

In the real world, MPC (Model Predictive Control) is preferred over LQR because it re-calculates the best action to apply to the plant periodically, meaning it has access to updated system state information. This allows

MPC to handle time varying dynamics, constraints, and non-linear systems.

LQR is like a compass, you initiate the north and south values, and it will always point you towards the poles regardless of where you are at any instance of time. MPC is like a GPS, which calculates the most optimum route periodically to prevent any external factors from affecting the trajectory.

MPC does this by integrating over a finite region, whereas LQR integrates up to infinity to ensure the system always moves towards the desired state regardless of time.

MPC is mathematically more complex than LQR and requires even deeper knowledge in matrix manipulation and calculus, therefore I will not go any more into depth for MPC.