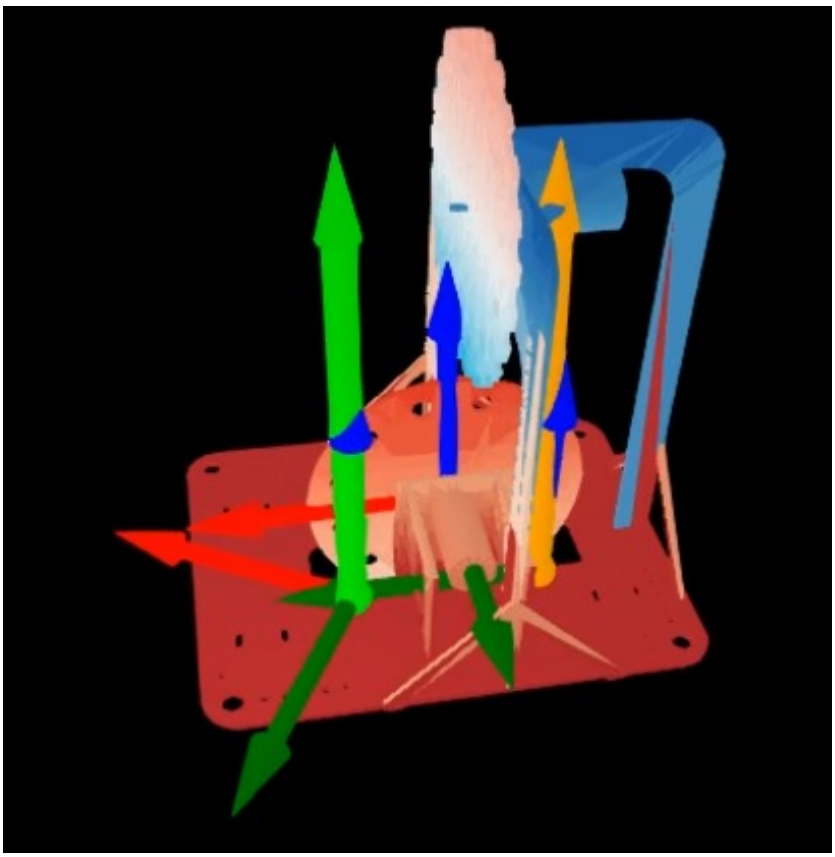
How the Reaction-Wheel Unicycle Works



This is a model of a unicycle with two symmetrically attached rotors. One of the reasons the matrix of inertia is not trivial is that the rotors' axes of rotation do not intersect at a point. The constraint on the system is conservation of angular momentum. The angular velocity of the body is related to the rotor velocities. That relation gives rise to a differential equation in the rotation group Special Orthogonal of real dimension 3 for the robot's body. Using the Euler parameters of $SO(3)$ we obtain a local coordinate description of the differential equation, in terms of the roll, pitch and yaw angles. The robot can be repositioned by controlling the rotor velocities. The Linear Quadratic Regulator regulates the roll and pitch angles by a choice of a suitable input.

The Control of the Balancing Unicycle

Here we deal with non-linear control systems that are described in terms of either differential equations or difference equations. In other words we consider the follwoing two types:

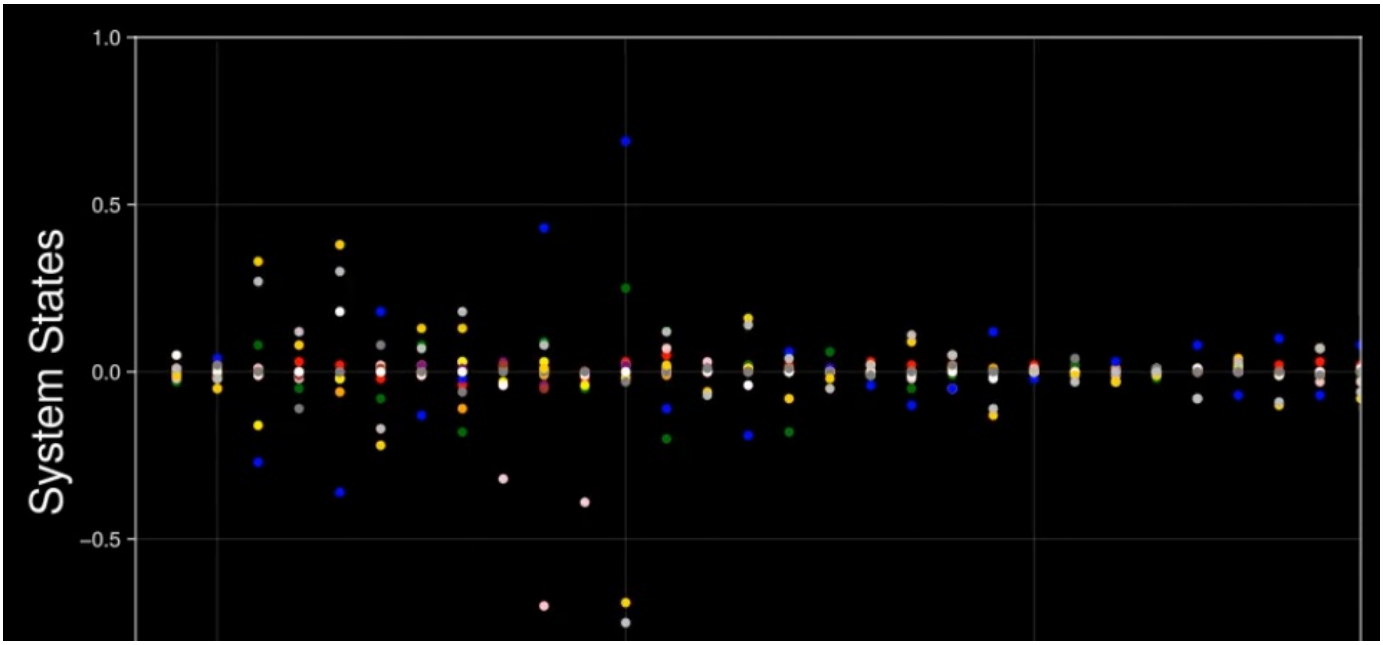
1. Differential equations: $\begin{cases} \dot{x}(t) = f(x(t), u(t)) \\ y(t) = h(x(t), u(t)) \end{cases}$
2. Difference equations: $\begin{cases} x(k+1) = f(x(k), u(k)) \\ y(k) = h(x(k), u(k)) \end{cases}$

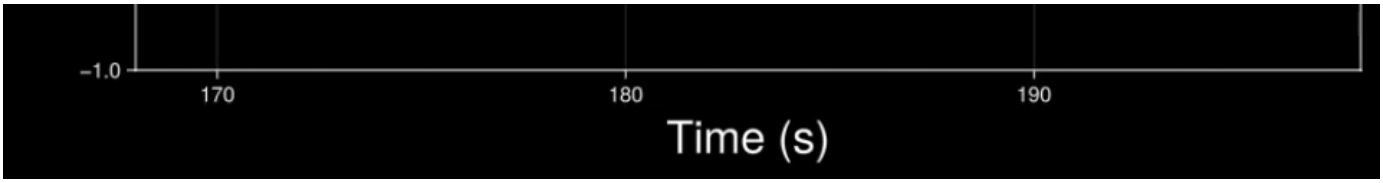
The variable x denotes the system state, the variable u denotes the control input to the system, and finally, the y variable denotes the output of the system. In the following, we explain how the system works through a robotics application. In this section, we see how the system works. But in the next section, we build a regulator to control the system state by generating suitable inputs for a given time frame.

The policy function produces the control inputs $u(t)$ (or $u(k)$ in the descrete case). The feeadback policy is called optimal control whenever it makes the system state x approximately equal to zero as the result of its application. The feeback policy is able to adapt through periodic policy updates. Between each two consecutive policy updates, a recursive relation updates filter coefficients, blocks of which are used to update the policy function. The policy update loop is slower whereas the filter coefficients update loop is faster.

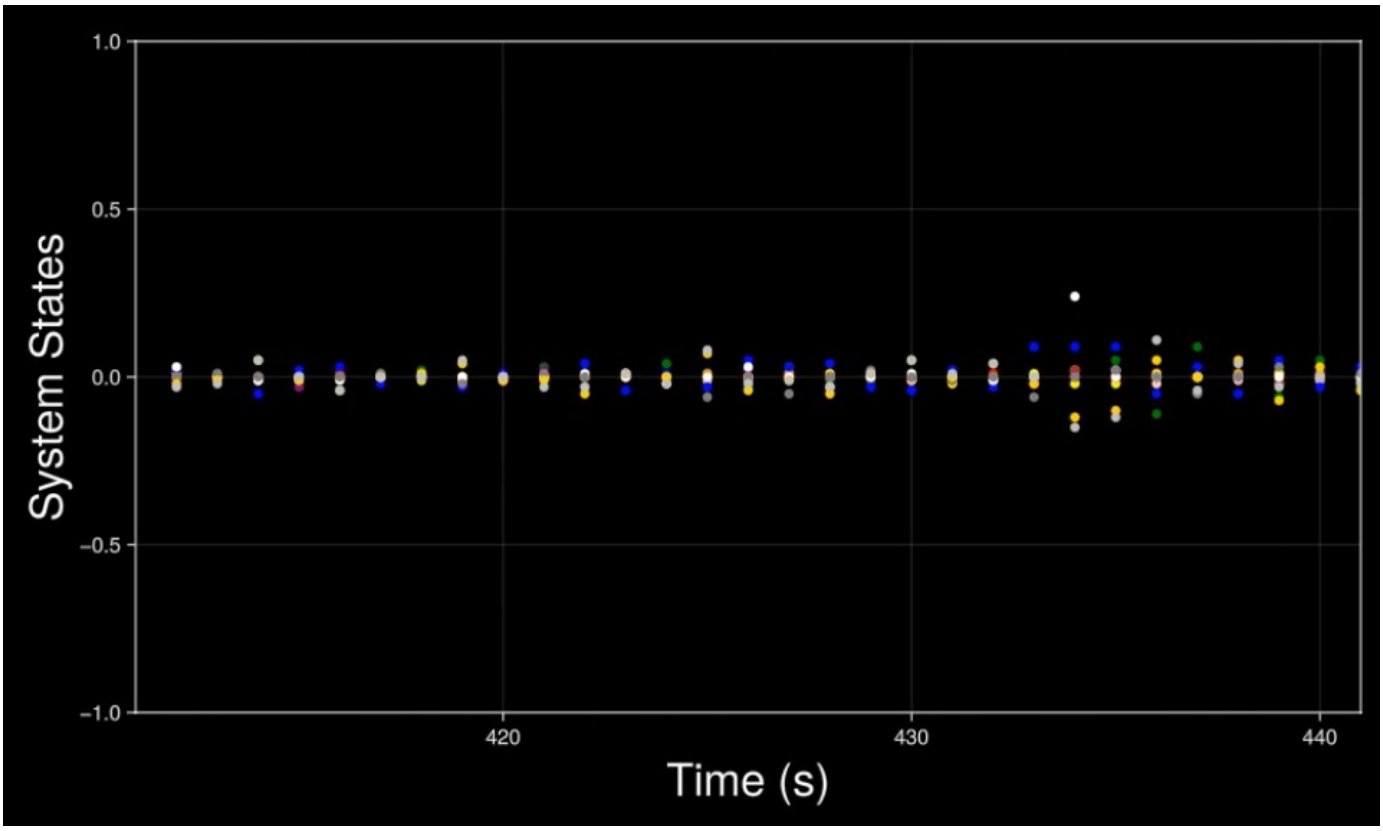
The function h, which produces the output y, and the quality $Q(x, u)$ are similar functions. The similarity of $h(x, u)$ and the quality function $Q(x, u)$ is first because both have the same parameter signature that includes the statem state vector x and the input vector u. Second, because both h and Q produce verifiable statements about the value of the ststem state at the next time step $k + 1$ or t in the immediate future. Later in the next section, we see how a least squares relation can be used to calculate a difference equation between the desired state and the measured state, as a feedback signal for enhancing the quality function Q and producing accurate outputs y over time.

On one hand, the filter coefficients play the role of a critic that evaluates the quality of being in state x and having taken input u. On the other hand, the policy plays the role of an actor that uses the system state x as input to a matrix-vector product that produces the feedback policy u. The Actor/Critic architechture uses the first principles of reinforcement learning for adapting the optimal control inputs. The adpativeness of the controller increases the probability that the quality of the system state x is measured higher as the time variable t in differential equations (or k in the case of difference equations) progresses forward in time.





System states in real time. Even though the matrix of inertia (among other physical parameters) is unknown, the adaptive controller based on value iteration keeps the states stable and regulates them to zero.



Example

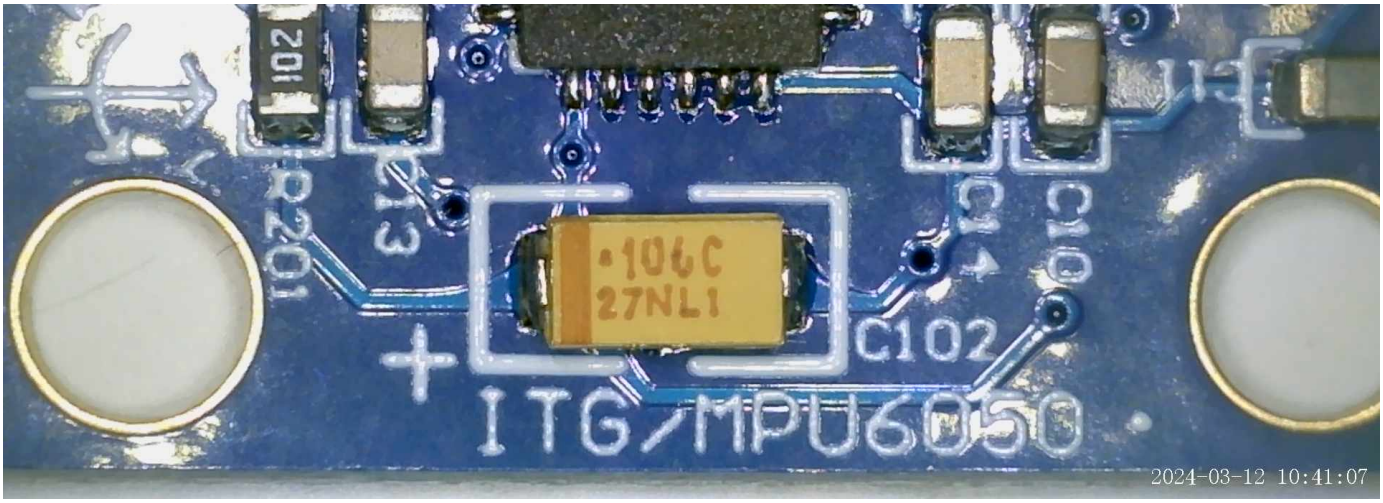
The robot that we build here is a self-balancing reaction wheel unicycle. As it is known from the name, this robot has only one wheel and therefore has one contact point with the ground surface. Just one contact point is the reason it is more complex compared to two-wheel balance robots. In fact, the robot has to preserve it balance in two directions around two perpendicular axes. In this robot, the motion in the forward/backward direction is like balancing a two-wheel balance robots, and so it obeys the same laws of physics. But since the robot has no way of moving to either left or right, in order to balance along the left/right direction we have to use another torque generator. For that purpose, we use a rotating mass, which is also called a reaction wheel. This rotating mass, which is mounted at the top of the robot, works based on the physical principle that if we apply a torque on it until it starts moving, then that mass also applies a torque on the robot's chassis, as large as the torque that is applied to it. One of the physical principles called the preservation of angular momentum explains this phenomenon. According to this principle, the sum of the angular momenta of a rotating set around a specific axis remains constant, unless it is

acted upon by an external torque. So if one of the parts of the rotating set starts rotating using its internal torque, then the complement of the part (the other parts of the set) start rotating in the opposite direction in order to neutralize the internal motion of that part. Otherwise the angular momentum of the entire set will not be preserved. Using this reaction torque we can take control of the angle of the robot's bofy in the left and right directions.

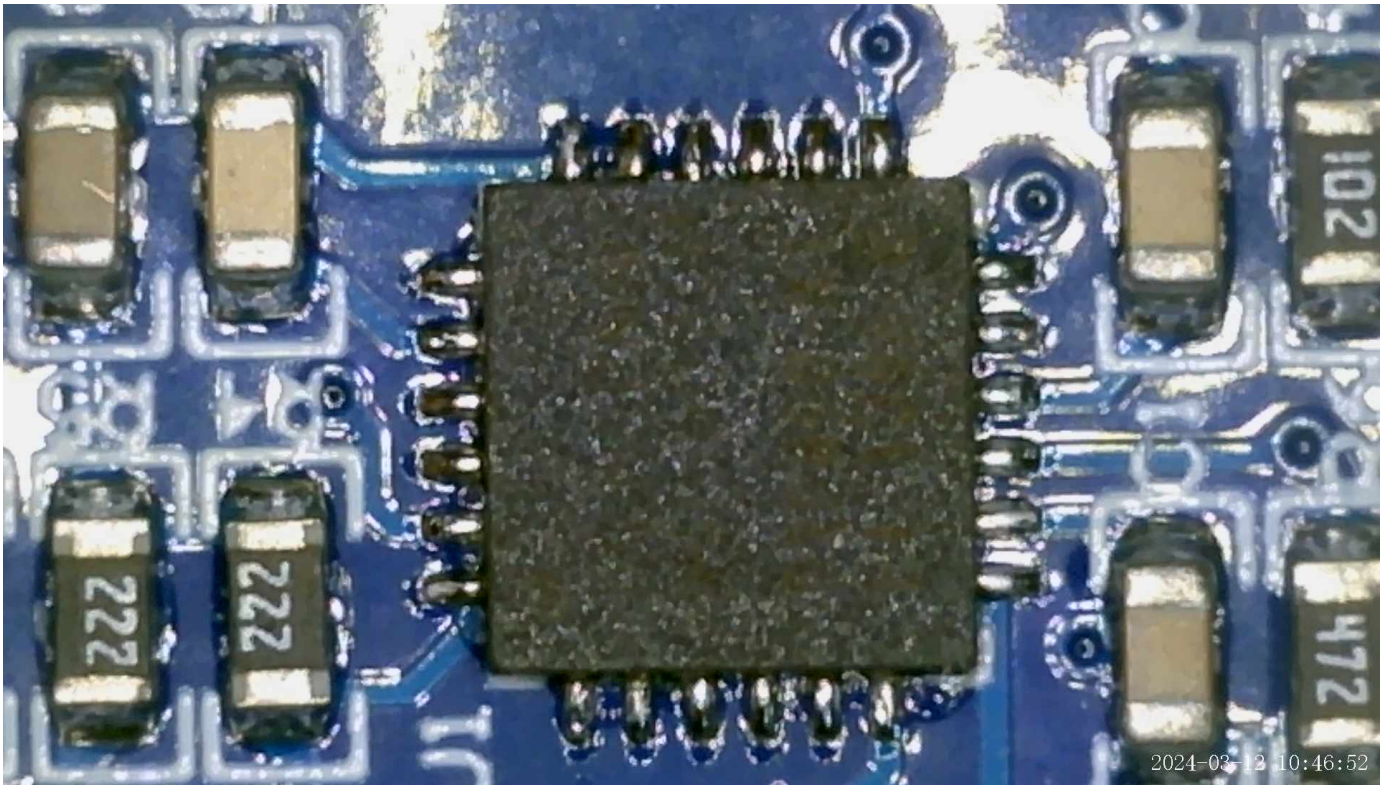
The one-wheel balance robot may not seem so practical at first, but similar to two-wheel balance robots, or different kinds of the inverted pendulum, provides proper conditions for experimenting with various control algorithms. In addition, the most important part of the robot, the reaction wheel has a special application in satellites. After a satellite is placed into orbit around a celestial body, the only force acting on it is caused by the gravitational field. Therefore, it will have no control over its own motion. In order for the satellite to make small maneuvres along its path, or to be able to make small adjustments to its orbit, usually they equip it with three motion systems: the propulsion motor, the electromagnetic torque generator, or the reaction wheel. The first item is outside of the scope of this project. The reaction wheel is applied in satellites by rotating the wheel in the opposite direction for as much as needed for pointing at a specific direction. The amount of rotation is determined based on the ratio of the rotational momenta between the satellite and the wheel. To control the rotation of the satellite in all spatial directions, it is equipped with three wheels that are mounted in mutually perpendicular directions. Professional motorcyclists also take advantage of this property of the preservation of rotational inertia. Whenever a motorcyclist makes a jump and is detatched from the Earth, The only force acting on in that moment is the gravity of the Earth, which is outside of the control of the cyclist. In this situation, the cyclist can set its landing angle (attack vector) by accelerating the rear wheel or decelerating (braking). The effect of the reaction of the rear wheel that is applied to the body of the motorcycle, rotates the whole system in the upward or downward direction. To build this robot, it is required to become familiar with the mechanical structure, mathematical modelling, control algorithms, digital design and electronic circuits.

The Z-Euler Angle Is Not Observable

The control and navigation of mobile robots is not pissible without knowing the position. For positioning, various sensors have been designed and built, which are used based on their specific applications. Among positioning systems one can name a list: accurate accelerometrs used in rockets, existing gyroscopes in flying machines, altimeters, navigation systems based on the magnetic field of the Earth, or even more advanced navigation systems based on the images of stars that are used in satellites and spacecrafts.



Nowadays, the electromechanical sensors are manufactured in small scales (micrometers). This technology is known by the name of Micro-Electro-Mechanical Systems (MEMS). The birth of the MEMS technology has had a great effect on the price, size and the improved precision of different kinds of electronic sensors in the market. This subject makes it possible to use a multiple of such sensors in a small robot. In some cases, manufacturers offer multiple sensors of different types in a unit microchip package.

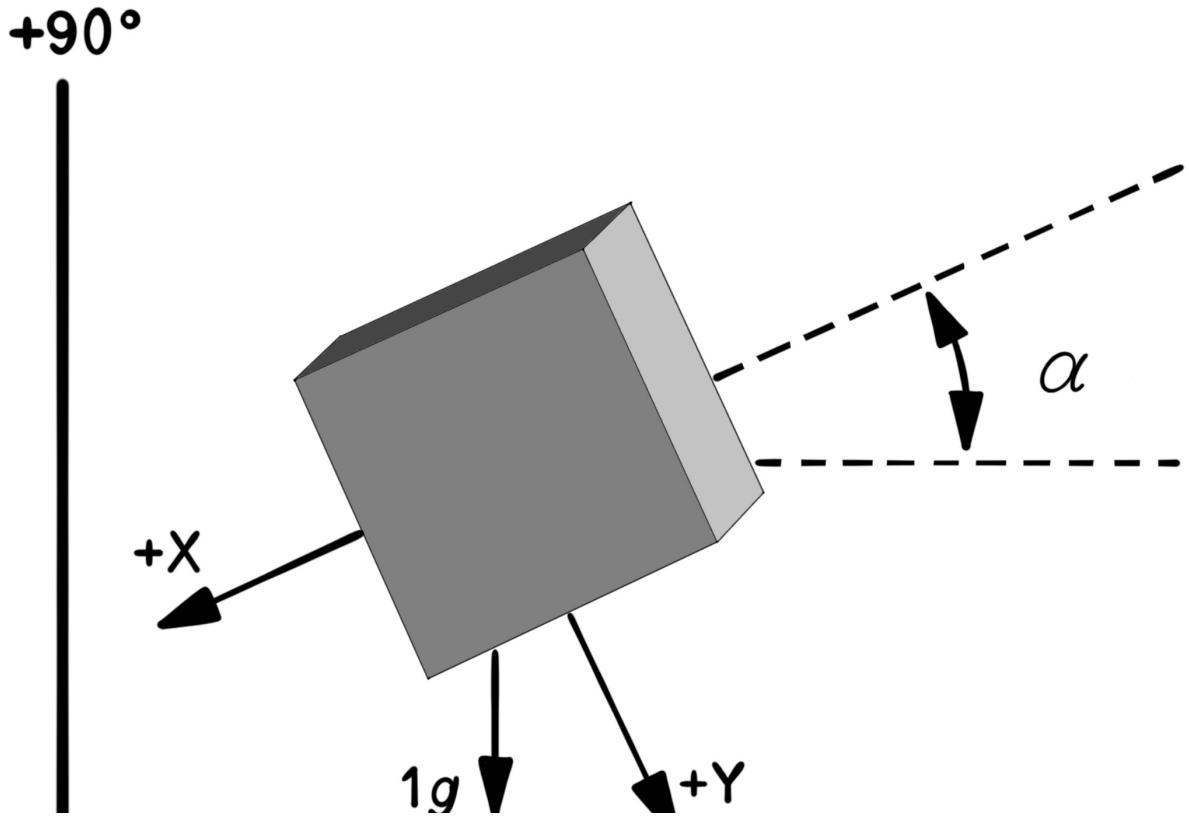


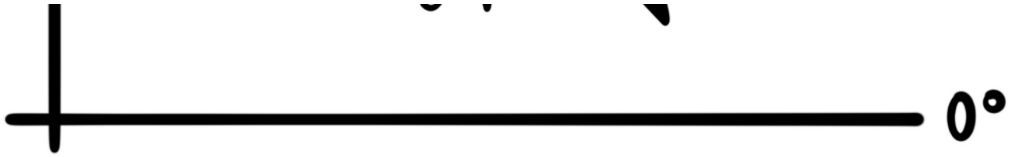
Among positioning peripherals, sensors that measure the acceleration, the rotational velocity (gyroscopes), and the magnetic field are the most applicable in small self-driving robots. This section focuses on the accelerometer and the gyroscope sensors. Using accelerometers you can calculate the accleration of your robot, along with its velocity and position through integration. You should know that the gravitational field of the Earth has an effect on the measurements of an accelerometer. This issue makes it harder to find the position, but it is useful for measuring the deviation from the gravitational direction (the vertical line). Also, the gyroscope essentially

measures the angular velocity, after which it will be possible to calculate the angular position (direction) using integration. This way, with the help of accelerometrs and gyroscopes you have the ability to measure the position and orientation of the motion of your robot, and in particular the estimate of the angle of deviation from the vertical line (the direction of gravity) is possible.

Acclerometers

Every accelerometer based on Micro Electro Mechanical Systems (MEMS) has some sort of moving part inside of it, such that it moves under the influence of external forces. This part is held in place using a spring structure, and the displacement caused by the external force on it, is measured using various methods such as the change in capacitance. The Hooke's law states that the force excerted by a spring is directly proportional to the displacement caused by that force. Then, knowing the spring constant (the force per unit of distance traveled) and the mass of the part, this displacement is transformed to its equivalent acceleration. Therefore, MEMS accelerometers measure an external force excerted on the moving part. That is why these accelerometers measure the static acceleration (the Earth's gravity) and the dynamical acceleration (due to changes in velocity) the same and decomposing the two measurements is your responsibility. In this way, if the direction of the accelerometer is in the direction of the Earth's gravitational field, then the measurement value is the representation of the acceleration due to motion in addition to the gravitational acceleration ($9.8\frac{m}{s^2}$). And if the direction of the measurement of the sensor is in the horizontal direction (perpendicular to the gravitational field of the Earth) then only the dynamical acceleration is measured and gravity will not have any effect on the measurement. So, in case a one-axis accelerometer (capable of measuring in one of the directions of the coordinate system) is used in a system, the orientation of it must be specified with respect to the gravitational direction so that the static acceleration is computable.

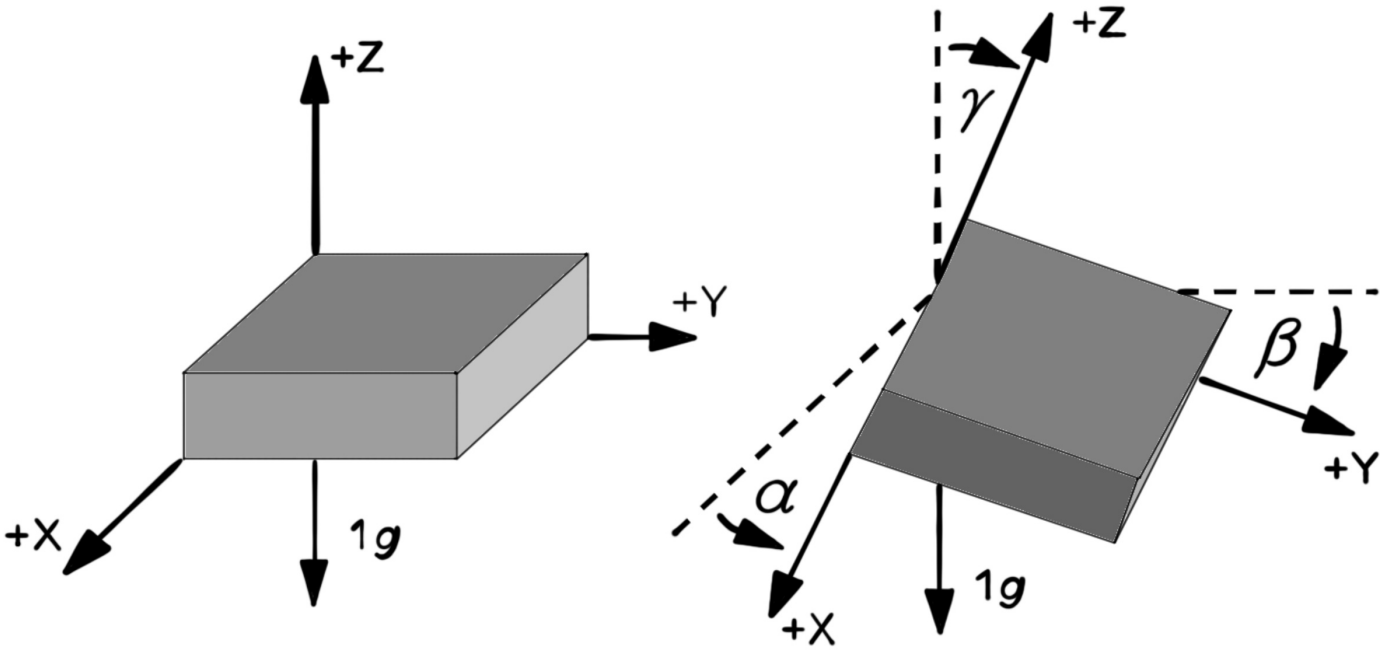




Using a two-axis accelerometer for measuring the direction of the Earth's gravity in the plane perpendicular to the Earth. In this figure, the orientation of the two-axis accelerometer (X-Y) with respect to the horizontal direction is computable using the given relation.

$$\alpha = tan^{-1}(\frac{A_X}{A_Y})$$

Now, imagine that you have two or three accelerometers such that their directions of measurement are mutually orthogonal (like the X, Y, and Z coordinate axes of the standard Cartesian coordinate system). If the velocity of this set is constant and only the static accelration due to gravity acts on it, by comparing the ratio of the measured accelerations across the axes, the orientation angle of the set with respect to the direction of gravity is computable. This is the way that many electronic balances and mobile robots use to measure the angle of orientation with respect to the direction of gravity.



Using a tri-axis accelerometer for measuring the direction of the Earth's gravity in the three-dimensional space. In this figure, the orientation of the three-axis accelerometer (X-Y-Z) with respect to the hirozontal plane and the direction of gravity is computable using the given relations.

$$\alpha = tan^{-1}(\frac{A_{X,OUT}}{\sqrt{A_{Y,OUT}^2+A_{Z,OUT}^2}})$$

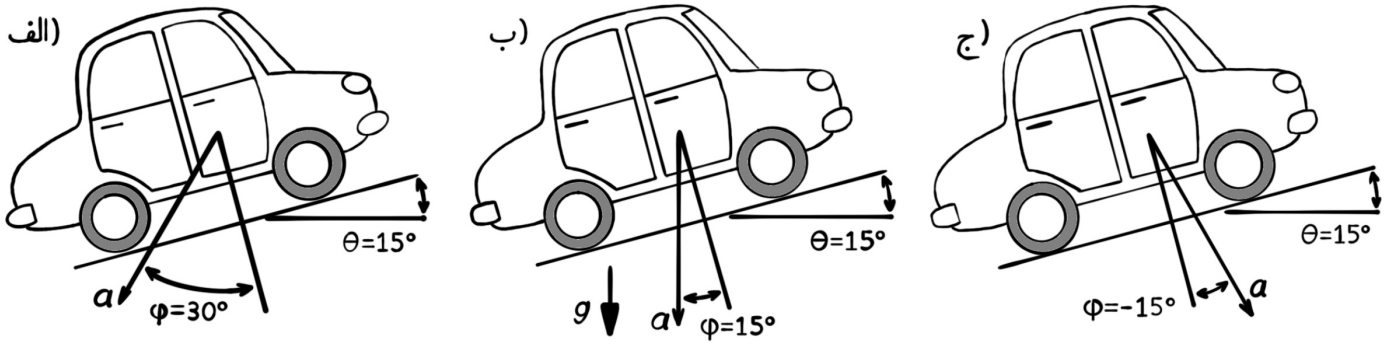
$$\beta = tan^{-1}(\frac{A_{Y,OUT}}{\sqrt{A_{X,OUT}^2+A_{Z,OUT}^2}})$$

$$\gamma = tan^{-1}(\frac{\sqrt{A_{X,OUT}^2+A_{Y,OUT}^2}}{A_{Z,OUT}})$$

Note that knowing the angles between each axis and the direction of gravity, does not give the general angular orientation of the accelerometer in the three-dimensional space. In fact, if this accelerometer is rotated about an axis parallel to gravity, all three axes will measure the same result compared to before the rotation. The reason for the Z-Euler angle not being observable is that the projection of the gravity vector onto the horizontal plane looks like a point (approximately the zero vector). In order to determine the angular orientation of the accelerometer in a complete way, it needs to measure at least two known vectors that are not parallel with respect to each other (the gravity vector and another vector). In every case, using a three-axis accelerometer one can build an electronic balance that can measure tilt in two mutually perpendicular directions.

Many manufacturers, make two-axis and tri-axis accelerometers as one chip, where two and three accelerometers (respectively) are placed in perpendicular directions in a unit package.

One of the fundamental problems of using accelerometrs to measure deviation, is the effect of dynamical accelerations (caused by changes in velocity) on the measurement of direction. For example, if you install such a device on a car and want to measure the slope of the road, the measured direction is correct as long as the vehicle has constant velocity. But when the car's velocity changes, the vector of dynamical acceleration is added to the vector of static acceleration and your measuring device measures the direction of this new vector (which is different from the direction of the Earth's gravity). One other disadvantage of accelerometers is the sensitivity to vibrations and the production of noisy results.



Estimating the slope of the road through measuring the direction of gravity using the accelerometer that is embedded in the car. In (a) the car has positive acceleration (increasing velocity), and in (b) without acceleration (constant velocity), and in (c) the car has negative acceleration (braking). As you can see, only in the figure (b) the direction of gravitational acceleration and the slope of the road are measured correctly.

Sensitivity to vibrations and dependence on dynamical accelerations, make it necessary to get help from other sensors such as gyroscopes and the magnetic field sensor (the electronic compass) for measuring the direction of the Earth's gravity.

To choose an accelerometer, one should pay attention to the measurement range, the sampling rate, the interface (the analog or digital communication protocol) and also the number of axes that are needed in the project (the number dimensions). Other parameters that should be considered in MEMS accelerometers are sensitivity to temperature and supply voltage changes, and the existence of an initial offset (the value read at zero acceleration), which is corrected with calbration. Here, we show how to use three-axis accelerometers.

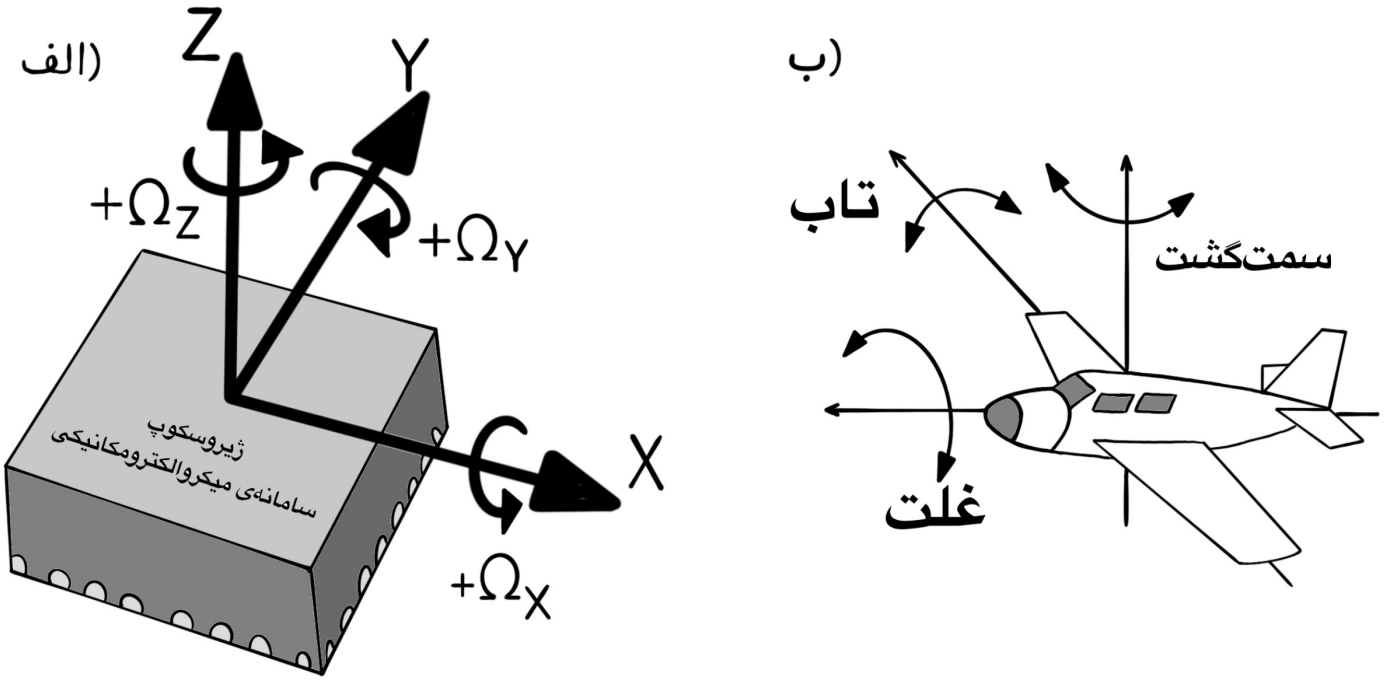
Gyroscopes

As you know, a gyroscope essentially measures the angular velocity about an axis. A rotation about an axis is measured with a specific value (often in terms of degrees per second $\frac{deg}{s}$) and a rotation in the opposite direction results in a value with the opposite sign, and in the case where the rotation is stopped, the value of zero is measured. Mechanical gyroscopes that work based on coriolis forces / the coriolis effect of a rotating mass, had been used in airplanes and rockets for a while until optical gyroscopes and various kinds of MEMS were built. Among gyroscopes, optical gyroscopes are the most accurate, whereas the MEMS gyroscopes are the cheapest and the most applied type of this measuring device.

Unlike accelerometers, a gyroscope is not generally sensitive to vibrations and produces more continuous measurement results. But, since the angular velocity is not that useful on its own, and the angular orientation is more useful to mobile vehicles, the output of this sensor is integrated to extract the angular position. Having an integrator in a positioning system based on gyroscopes causes the accumulation of the smallest offsets and unavoidable permanent errors over time to produce greater errors. As such, the angular position computed by the integrator using the output of the gyroscope drifts away from the real value over time, so much that after the passgae of a few minutes (or even a few seconds) the computed values are no longer valid. This issue forces the use of other sensors such as a detector of the Earth's magnetic field or an acclerometer along with

the gyroscope, unless the goal of measurement is only the angular velocity and not the angular position, in which case the integrator is removed and the output of the gyroscope will be accurate enough.

Like MESMS accelerometers, MEMS gyroscopes are manufactured in small sizes with afforable prices, and many manufacturers provide two or three gyroscopes in a single electronic package for measurements along different directions that are perpendicular with respect to one another.



A three-axis gyroscope measures the angular velocity about three mutually perpenducular axes (X, Y and Z). Ususally, the right-handed rotation about each axis is denoted by the positive sign and the left-handed rotation is denoted by the negative sign. The angular velocity is expressed in terms of degrees per second $\frac{deg}{s}$. In flying vehicles such as rockets and airplanes and also some mobile robots, the names Roll, Yaw and Pitch are used to label the rotation axes. The axes Roll, Yaw and Pitch are not necessarily aligned with the X, Y and Z axes and this fact depends on the assignment of the coordinate system axes to the mobile object.

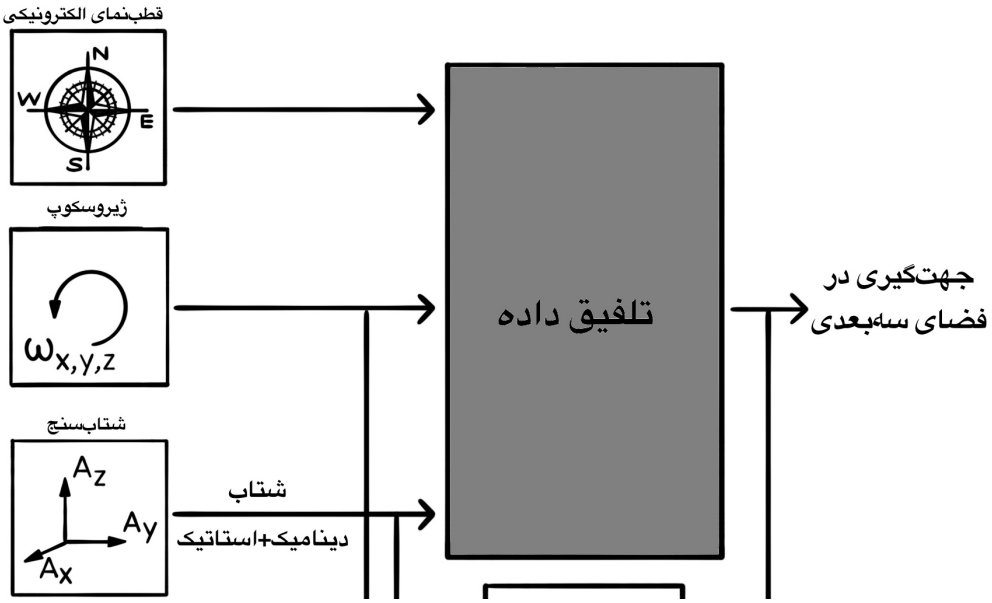
When choosing a gyroscope, one should consider its measurable velocity range, the sampling rate, the way to communicate with it (analog or digital), and also the number of required axes in the project (the number of dimensions). In addition to those parameters, consider issues such as the sensitivity to temperature and supply voltage changes, the initial offset (the value read in the stationary state) and the cross-sxis sensitivity. If a gyroscope is rotated around an axis perpendicular to the measurement axis, it should measure the value zero, but it is not the case in

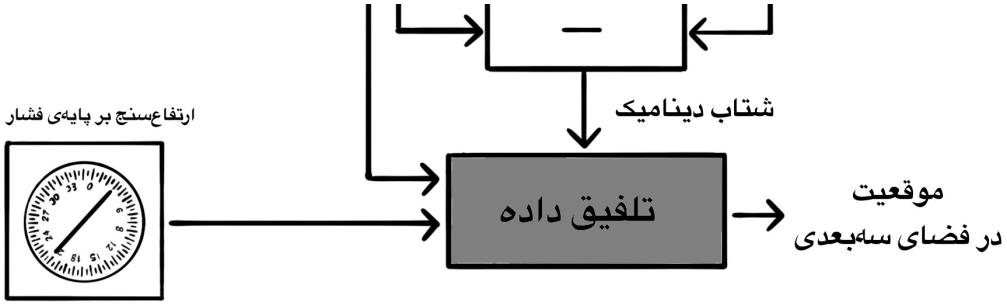
practice. The gyroscope can show sensitivity to rotation in directions other than the one that is to be measured. This value, which must be as small as possible, represents the cross-axis sensitivity and is expressed in terms of the error percentage.

Fusing the Output Data of Accelerometers with that of Gyroscopes

In this section we want to use the data of both accelerometers and gyroscopes in order to estimate the correct direction of gravity. A tri-axis accelerometer alone can be used to find the orientation with respect to the gravity vector. However, this estimate is correct only when there are no accelerations acting on the system other than the static gravitational acceleration. This is not possible in mobile robots. On top of that, an accelerometer is very sensitive to vibrations and due to too much noise, its output does not have much value on its own. In contrast, gyroscopes have their own disadvantages, the most important of which is the gradual drifting of the calculated angle from the real value, which is calculated using integration over time. Fortunately, the errors existing in acclerometers and gyroscopes are completely different in nature, such that by using the sensors in the proper way, one can correct both errors. For an effective application of the two sensors, one must fuse their data together in a way that the fused result is more valid than the result of each sensor on its own.

Assuming that there are no long-term dynamical accelerations acting on the system, and so assuming that the direction of gravity has been calculated in a correct way, you can subtract the gravity vector of the Earth (the direction of which is known and its magnitude is approximately equal to $9.8\frac{m}{s^2}$) from the filtered information of the accelerometer to get the dynamical acceleration of the motion. By integrating the dynamical acceleration you can calculate the velocity of motion and the position of the robot. But these results are valid short-term because of the integral operation. To prevent the accumulation of error over a long period of time, it is necessary to add another positioning sensor such as the Global Positioning System (GPS) to the set.





As you know, in a positioning system including tri-axis gyroscopes and accelerometers (having six degrees of freedom), the gravity vector calculated using the accelerometer is like a yardstick that counteracts the effect of the gradual drift of the gyroscope from the estimation of the direction of gravity. But this vector does not have any projections on the horizontal plane. In addition, knowing the direction and magnitude of a known vector (like the gravity vector) in an orthogonal coordinate system, it is not possible to determine the direction of all three coordinate axes at the same time. In fact, one can show that there are an infinite number of coordinate systems that measure a particular vector the same (if you rotate a coordinate system about an axis parallel to that particular vector, all coordinate systems that are created as a result of rotating the coordinate system through various angles about the axis, measure the said vector the same). To detect the orientation of a coordinate system in the three-dimensional space, we have to have the measurement results of a pair of non-parallel vectors in the coordinate system. The coordinate system that we want to find the orientation of, is our Inertial Measurement Unit (IMU). That is why the output of a data fusion algorithm that uses the information of tri-axis accelerometers and gyroscopes, is not a valid source to find the orientation around the vertical axis (the Z-axis). To solve this problem one must use another vector that has a substantial projection on the horizontal plane. An electronic compass can accomplish this goal. Positioning systems in flying machines generally take advantage of all three sensors: compasses, gyroscopes and accelerometers (having nine degrees of freedom), and the fusion algorithms in them process the information of all of the sensors.

Estimating Tilt Using Accelerometers

In this section, we find an estimator for the roll and pitch angles of a rigid body that has only rotational degrees of freedom. This estimate is based on the measurements of multiple accelerometers that are mounted on the rigid body, such that it is assumed that we know their mounting positions and their directions of orientation. First, we describe the roblem setup in a formal way. Then, we find an estimate of the gravity vector in the frame of the robot. Finally, we use the gravity vector in order to calculate the roll and pitch angles. In this section, we follow the setup described in Sebastian Trimpe and Raffaello D'Andrea (2010) [2].

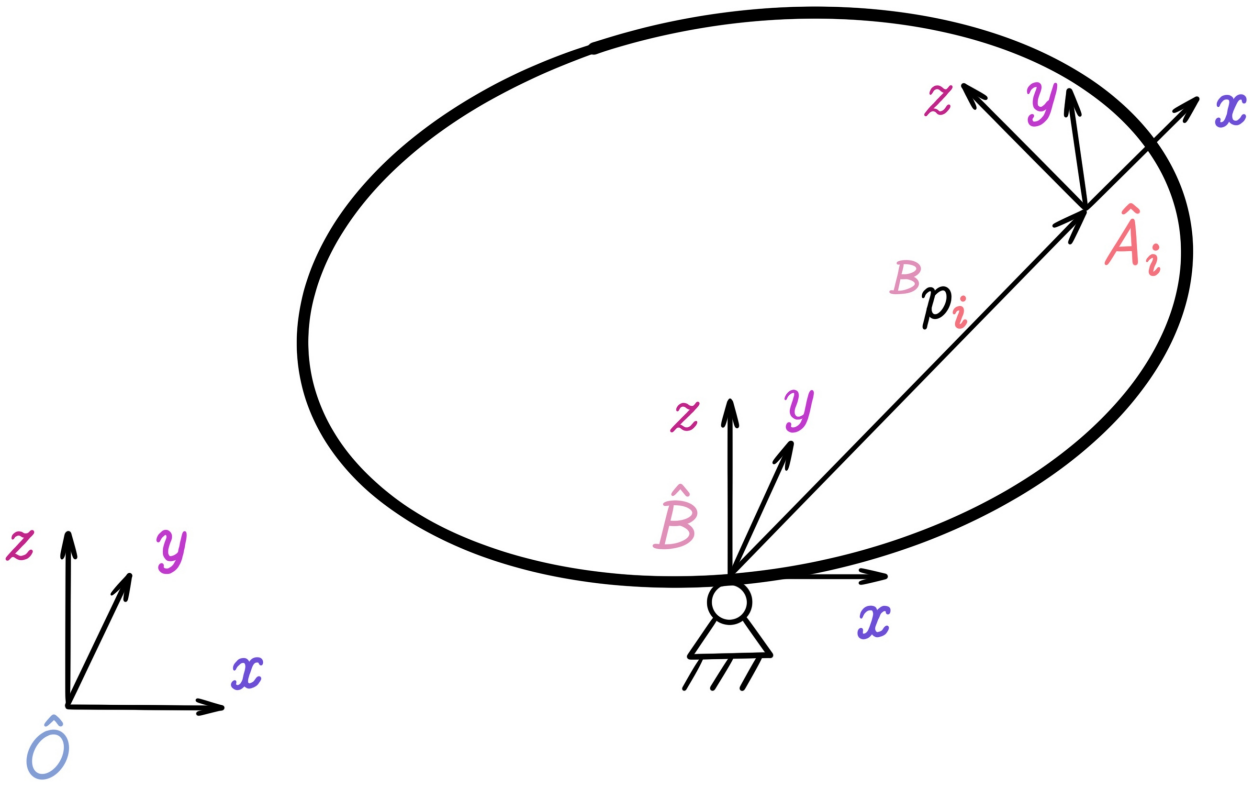
The Problem Setup

Suppose there is a rigid body that is standing on a fixed pivot point without friction. Therefore,

the body has three rotational degrees of freedom, but has no translational degrees of freedom. The origin of the coordinate system of the body, which is denoted by $\hat{B}$, is at the center of rotation. The inertial reference frame is denoted by $\hat{O}$, the origin of which is coincident with the origin of the body's frame $\hat{B}$. On the body, there are L sensors that are mounted at specific locations p_i, for $i = 1, ..., L$. It is assumed that the position of sesnors ${}^{B}p_i$ is known in the reference frame of the body. Each sensor $\hat{A}_i$ measures the local acceleration along three directions in its local frame. Two rotation matrices are introduced to describe the rotation of the rigid body and the mounting orientaton of the sensors: the first matrix describes the rotation of the inertial reference frame $\hat{O}$ with respect to the body's frame $\hat{B}$, and the second matrix determines the rotation of the ith sensor's local reference frame with respect to the body's frame.

$$\begin{cases} {}^{O}_{B}R \\ {}^{A_i}_{B}R \end{cases}$$

For example, a vector ${}^{B}v$ in the body's reference frame can be represented in the inertial reference frame with a matrix-vector multiplication ${}^{O}v = {}^{O}_{B}R\ {}^{B}v$.



An accelerometer, measures ${}^{A_i}m_i$ the acceleration vector ${}^{O}\ddot{p}_i$ at its mounting position in addition to the gravity vector ${}^{O}g$, which is rotated with respect to its local frame $\hat{A}_i$.

${}^{A_i}m_i = {}^{A_i}_{B}R\ {}^{B}_{O}R({}^{O}\ddot{p}_i + {}^{O}g) + {}^{A_i}n_i$ (Equation 1)

In equation 1, the measurement of the accelerometer is denoted by ${}^{A_i}m_i \in \mathbb{R}^3$ and the measurement noise is denoted by ${}^{A_i}n_i \in \mathbb{R}^3$. It is assumed that it is the white noise, and its value $\mathbb{E}[{}^{A}n_i \ ({}^{A}n_i)^T] = \sigma_n^2 I_3$ is limited by the zero mean $\mathbb{E}[{}^{A}n_i] = 0$ and the standard deviation σ_n. Here, the mathematical expected value is denoted by $\mathbb{E}$ and the matrix $I_3 \in \mathbb{R}^{3\times 3}$ is the identity matrix of real dimension three. This model of noise is reasonable for acclerometers that are based on the MEMS technology provided that the bias is subtracted from it.

Using the coordinate transformation ${}^{O}p_i = {}^{O}_{B}\ R {}^{B}p_i$ and the fact that ${}^{B}p_i$ is constant in time $\frac{d\,{}^{B}p_i}{dt} = 0$, we arrive at the conclusion that the acceleration ${}^{O}\ddot{p}_i$ in terms of the inertial reference frame $\hat{O}$ is calculated using the second derivative of the rotation matrix ${}^{O}_{B}\ddot{R}$ with respect to time $\frac{d^2\,{}^{O}_{B}R}{dt^2} = {}^{O}_{B}\ \ddot{R}$. The matrix ${}^{O}_{B}\ddot{R}$ captures the dynamic terms of the rigid body: the rotational and centripetal acceleration terms.

${}^{O}\ddot{p}_i = {}^{O}_{B}\ \ddot{R}\ {}^{B}p_i$ (Equation 2)

Using equation 2, one can rewrite the acceleration measurement in equation 1 as follows:

${}^{A_i}m_i = {}^{A_i}_{B}\ R\ {}^{B}_{O}R({}^{O}_{B}\ddot{R}\ {}^{B}p_i + {}^{O}\ g) + {}^{A_i}\ n_i$ (Equation 3)

Since all of the rotational orientations of the sensors, denoted by ${}^{A_i}_{B}R$, are assumed to be known, we can represent the sensor measurements ${}^{A_i}m_i$ in terms of the body's frame of reference $\hat{B}$ by multiplying equation 3 on the left with the transpose of the rotation matrix ${}^{B}_{A_i}R = {}^{A_i}_{B}\ R^T$.

${}^{B}m_i = \tilde{R}{}^{B}p_i + {}^{B}\ g + {}^{B}\ n_i$ (Equation 4)

In equation 4, the matrix $\tilde{R}$ combines the rotation of the body ${}^{B}_{O}R$ and the dynamic terms of the motion of the body ${}^{O}_{B}\ddot{R}$ using a matrix-matrix product: $\tilde{R} := {}^{B}_{O}\ R\ {}^{O}_{B}\ddot{R}$.

${}^{B}g = {}^{B}_{O}\ R\ {}^{O}g$ (Equation 5)

Also in equation 4, the gravity vector ${}^{B}g$ is represented in terms of the body's reference fame $\hat{B}$, and the noise vector ${}^{B}n_i$ is rotated with respect to the body's frame ${}^{B}n_i = {}^{B}_{A_i}\ R\ {}^{A_i}n_i$. The mean of the noise after the coordinate transformation is still equal to zero $\mathbb{E}[{}^{B}n_i] = 0$, and the standard deviation of the noise is still limited to the scalar multiplication of σ_n^2 with the three-dimensional identity matrix: $\mathbb{E}[{}^{B}n_i({}^{B}n_i)^T] = \sigma_n^2 I_3$.

Suppose that the measurements are done at a rate T and the time index is introduced as k. So we can rewrite equation 4 like the following:

${}^{B}m_i(k) = \tilde{R}(k)\ {}^{B}p_i + {}^{B}\ g(k) + {}^{B}\ n_i(k)$ (Equation 6)

With the given measurements that are done in equation 6 for every sensor from i through L at time k, the objective is to estimate the tilt of the rigid body at time k, which is captured by the matrix ${}^{B}_{O}R$. As an intermediate step, an estimate of the gravity vector ${}^{B}g(k)$ at time k (and as a side product) an estimate of the matrix $\tilde{R}(k)$ is derived. Then, the estimate of the gravity vector is

used to find the roll and pitch angles of the rigid body.

The Optimal Estimation of the Gravity Vector

In this section, the problem of the estimation of the gravity vector ${}^{B}g$ in the reference frame of the robot's body $\hat{B}$ using the acceleration measurements ${}^{B}m_i$ in equation 6, for $i = 1, ..., L$, is described as a least-squares problem. All of the measurements in equation 6, which are L in number, are combined in a matrix equation where the time index k is removed for convenience.

$M = QP + N$ (Equation 7)

In equation 7, the matrix denoted by M combines the measurements of all sensors, Q denotes the matrix of unknown parameters, whereas P denotes the matrix of known parameters.

$M := \begin{bmatrix} {}^{B}m_1 & {}^{B}m_2 & \dots & {}^{B}m_L \end{bmatrix} \in \mathbb{R}^{3\times L}$ (Equation 8)

$Q := \begin{bmatrix} {}^{B}g & \tilde{R} \end{bmatrix} \in \mathbb{R}^{3\times 4}$ (Equation 9)

$P := \begin{bmatrix} 1 & 1 & \dots & 1 \\ {}^{B}p_1 & {}^{B}p_2 & \dots & {}^{B}p_L \end{bmatrix} \in \mathbb{R}^{4\times L}$ (Equation 10)

$N := \begin{bmatrix} {}^{B}n_1 & {}^{B}n_2 & \dots & {}^{B}n_L \end{bmatrix} \in \mathbb{R}^{3\times L}$

The letter N in equation 7 denotes the matrix that combines all of the noise vectors, meaning its expected value is equal to zero $\mathbb{E}[N] = 0$, and its standard deviation is limited by the scalar multiplication of the L-dimensional identity matrix I_L with $\sigma_N^2 := \sqrt{3}\sigma_n$:

$$\mathbb{E}[N^T N] = 3\sigma_n^2 I_L = \sigma_N^2 I_L.$$

Other than the gravity vector ${}^{B}g$, which is what we are after, the unknown parameters matrix Q also contains the matrix $\tilde{R}$, combining the rotation of the body ${}^{B}_{O}R$ and the dynamic terms of the motion of the body ${}^{O}_{B}\ddot{R}$. In the follwoing, a scheme for the optimal estimation of the entire matrix Q is described, even though the gravity vector motivates the tilt estimation. In other applications, one might also be interested in estimating the dynamic terms, denoted by $\ddot{R}$, which is derived from the matrix $\tilde{R} = {}^{B}_{O}R\ {}^{O}_{B}\ddot{R}$ once the rotation matrix ${}^{B}_{O}R$ is found.

The objective is to find an estimate of the optimal matrix $\hat{Q}^*$ of the unknown parameters matrix Q such that the expected value of the optimization error is minimized (see equation 11 where $||.||_F$ denots the Frobenius matrix norm), subjected to the fact that $\mathbb{E}[\hat{Q}] = Q$ the expected value of the matrix $\hat{Q}^*$ equals the matrix Q.

$\min_{\hat{Q}} \mathbb{E}\left[||\hat{Q} - Q||_F^2\right]$ (Equation 11)

The estimate of the matrix $\hat{Q}$ is restricted to linear combinations of the measurements M, that is, we look for an optimal matrix X^* so that we can decompose the matrix Q as a matrix-matrix

product $\hat{Q} = MX$. This way results in a straight-forward implementation: at each time step, the estimate of the matrix Q is calculated by a matrix-matrix product. Note that in equation 7, the matrices M, Q and N are variable in time. At each time k, according to the measurements of the matrix M, we want to find an optimal estimate of the unknown parameters matrix Q. The next lemma, states the best unbiased linear estimate of the full parameter matrix Q.

The Full Estimation Lemma

Given the real matrices $P \in \mathbb{R}^{4\times L}$ and $M \in \mathbb{R}^{3\times L}$ satisfying $M = QP + N$ with unknown matrix $Q \in \mathbb{R}^{3\times 4}$ and the matrix random variable $N \in \mathbb{R}^{3\times L}$ with $\mathbb{E}[N] = 0$, $\mathbb{E}[N^T N] = \sigma_N^2 I_L$. Assuming P has full row rank, the (unique) minimizer $X^* \in \mathbb{R}^{L\times 4}$ of

$\min_X \mathbb{E}\left[||MX - Q||_F^2\right]$ subjected to $\mathbb{E}\left[MX\right] = Q$ (Equation 12)

is given by

$X^* = P^T(PP^T)^{-1}$. (Equation 13)

The minimum estimation error is

$\mathbb{E}\left[||MX^* - Q||_F^2\right] = \sigma_N^2 \sum_{i=1}^{4} \frac{1}{s_i^2(P)}$, (Equation 14)

where $s_i(P)$ denotes the ith largest singular value of P.

The Proof of the Full Estimation Lemma

Since $\mathbb{E}[MX] = \mathbb{E}[M]X = QPX$, it is required that

$PX = I$ (Equation 15)

to satisfy $\mathbb{E}[MX] = Q$. Next, consider the singular value decomposition (SVD) of P,

$P = U \begin{bmatrix}\Sigma & 0\end{bmatrix} \begin{bmatrix}V_1^T \\ V_2^T\end{bmatrix}$, (Equation 16)

with $U \in \mathbb{R}^{4\times 4}$ unitary, $\Sigma \in \mathbb{R}^{4\times 4}$ diagonal, $V_1 \in \mathbb{R}^{L\times 4}$, $V_2 \in \mathbb{R}^{L\times(L-4)}$, and $V = \begin{bmatrix}V_1 & V_2\end{bmatrix}$ unitary. From the full row rank assumption on P, it follows that Σ is positive definite. Therefore, a parameterization of all X that satisfy equation (15) is given by

$X = V_1\Sigma^{-1}U^T + V_2\bar{X}$, (Equation 17)

where $\bar{X} \in \mathbb{R}^{(L-4)\times 4}$ is a free parameter matrix. Thus, $\bar{X}$ needs to be chosen such that equation (12) is minimized: Using equations (7), (15) and basic peoperties of the trace operation, yields

$\mathbb{E}\left[||MX - Q||_F^2\right] = \mathbb{E}\left[||NX||_F^2\right]$

$= \mathbb{E}\left[trace(X^T N^T N X)\right] = trace(\mathbb{E}\left[N^T N\right] X X^T)$

$= \sigma_N^2\, trace(X X^T) = \sigma_N^2\, trace(V^T X (V^T X)^T)$

$= \sigma_N^2 || \begin{bmatrix} \Sigma^{-1} U^T \\ \bar{X} \end{bmatrix} ||_F^2$, (Equation 18)

which is minimized by $\bar{X} = 0$. Therefore,

$X^* = V_1 \Sigma^{-1} U^T = P^T (P P^T)^{-1}$,

which can readily be seen by inserting equation (16) for P, and

$\mathbb{E}[||MX^* - Q||_F^2] = \sigma_N^2 ||\Sigma^{-1} U^T||_F^2 = \sigma_N^2 ||\Sigma^{-1}||_F^2$.

□

The optimal estimate $\hat{Q} = MX^*$ includes both the optimal estimate of the gravity vector ${}^B g$ and of the dynamics matrix $\tilde{R}$. Since, for tilt estimation, only the former is of interest, one needs to ask if X^* is also optimal if one seeks only an estimate of parts of the unknown matrix Q. The following lemma states that this is indeed the case.

The Partitioned Estimation Lemma

Let the matrices Q, P, N and M be defined as in the full estimation lemma. Furthermore, let $Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}$, with

$$\begin{cases} Q_1 \in \mathbb{R}^{3 \times q} \\ Q_2 \in \mathbb{R}^{3 \times (4-q)} \end{cases}$$

where $1 \leq q \leq 4$. Assuming P has full row rank, the (unique) minimizer $Y^* \in \mathbb{R}^{L \times q}$ of

$\underset{Y}{min}\, \mathbb{E}\left[||MY - Q_1||_F^2\right]$ subjected to $\mathbb{E}\left[MY\right] = Q_1$ (Equation 19)

is $Y^* = X_1^*$, where $X^* = \begin{bmatrix} X_1^* & X_2^* \end{bmatrix}$ is the solution of the full estimation lemma.

The Proof of the Partitioned Estimation Lemma

It needs to be shown that $Y = X_1^*$ satisfies equation (19). First, since $X^* = \begin{bmatrix} X_1^* & X_2^* \end{bmatrix}$ satisfies $\mathbb{E}\left[MX\right] = Q$ in equation (12),

$\mathbb{E}\left[\begin{bmatrix} MX_1^* & MX_2^* \end{bmatrix}\right] = \mathbb{E}\left[MX^*\right] = Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \longrightarrow$

$$\begin{cases} \mathbb{E}\left[MX_1^*\right] = Q_1 \\ \mathbb{E}\left[MX_2^*\right] = Q_2 \end{cases} .$$

Then,

$$||MX^* - Q||_F^2 = || \begin{bmatrix} MX_1^* - Q_1 & MX_2^* - Q_2 \end{bmatrix} ||_F^2 = ||MX_1^* - Q_1||_F^2 + ||MX_2^* - Q_2||_F^2$$

i.e. X^* minimizes both terms in the last expression separately and X_1^* thus minimizes $||MX_1^* - Q_1||_F^2$ alone.

□

Applying the partitioned estimation lemma with $q = 1$ yields the optimal gravity vector estimate ${}^B\hat{g}(k)$ at time k given all sensor measurements $M(k)$,

${}^B\hat{g}(k) = M(k)X_1^*$, (Equation 20)

with $X_1^* \in \mathbb{R}^{L \times 1}$. The *optimal fusion vector* X_1^* is static and completely defined by the geometry of the problem (through P) and can thus be computed offline.

Note that the gravity vector estimate in equation (20) is independent of the rigid body dynamics, which are captured in ${}_B^O\ddot{R}$ (and thus in $\tilde{R}$). This can be seen from

$$\begin{aligned}
{}^B\hat{g} &= MX_1^* = QPX_1^* + NX_1^* \\
&= \begin{bmatrix} {}^Bg & \tilde{R} \end{bmatrix} \; U\Sigma V_1^T \; V_1\Sigma^{-1}U_1^T \; + NX_1^* \\
&= \begin{bmatrix} {}^Bg & \tilde{R} \end{bmatrix} \; P \; X_1^* \; + NX_1^* \\
&= \begin{bmatrix} {}^Bg & \tilde{R} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} U_1^T + NX_1^* \\
&= \begin{bmatrix} {}^Bg & \tilde{R} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + NX_1^* = {}^B\,g + NX_1^*,
\end{aligned}$$

where the SVD of P in equation (16) has been used. Clearly, the matrix $\tilde{R}$ does not appear in the estimate, i.e. the gravity vector observation is not corrupted by any dynamic terms. As expected, the sensor noise does enter the estimation equation.

The Physical Interpretation of the Full Row Rank Condition

Both in the full estimation lemma and the partitioned estimation lemma, the matrix P, which contains the sensor locations on the rigid body, is assumed to have full row rank. In the following, a physical interpretation of this rank condition is given.

Consider the case where P does not have full row rank. Then, there exists a nontrivial linear combination of the rows of P,

$\exists \lambda \neq 0 \in \mathbb{R}^4 : \lambda_1 p_x + \lambda_2 p_y + \lambda_3 p_z + \lambda_4 \mathbf{1} = 0$, (Equation 21)

where $p_x^T,\ p_y^T,\ p_z^T \in \mathbb{R}^{1\times L}$, denote the last three rows of P (the vectors of x, y and z-coordinates of all sensor locations, respectively) and $\mathbf{1}^T \in \mathbb{R}^{1\times L}$, the vector of all ones, is the first row of P. Expression (21) is equivalent to

$\exists \lambda \neq 0 \in \mathbb{R}^4 : \forall i = 1, ..., L$

$\lambda_1\ {}^B p_{i,x} + \lambda_2\ {}^B p_{i,y} + \lambda_3\ {}^B p_{i,z} = -\lambda_4$ (Equation 22)

where ${}^B p_{i,x},\ {}^B p_{i,y},\ {}^B p_{i,z} \in \mathbb{R}$ denote the x, y and z-coordinate of the ith sensor location in the body frame. Since the equation $\lambda_1 x + \lambda_2 y + \lambda_3 z = -\lambda_4$ defines a plane in (x, y, z)-space, condition (22) is equivalent to *all* L sensors lying on the same plane. Therefore, the full row rank condition on P is satisfied if and if only *not* all sensors lie on a plane, this also implies that at least four tri-axis accelerometers are required for the proposed method.

Note that the gravity vector estimate given in the partitioned estimation lemma is optimal under the assumption that P has full row rank. These results can be extended and the rank condition on P can be relaxed when one only seeks only the gravity vector. For example, one could directly measure gravity with a single tri-axis accelerometer at the pivot, where the dynamic terms do not enter the measurements.However, this is not possible for the balancing unicycle application.

Tilt Estimation

With the estimate ${}^B\hat{g}$ of the gravity vector in the body frame, one can use equation (5) to estimate the body rotation, since the direction of the gravity vector in the inertial frame is known. In this project, the attitude of the rigid body is represented by z-y-x-Euler angles (yaw, pitch, roll), i.e. the body frame $\hat{B}$ is obtained by rotating the inertial frame $\hat{O}$ succesively about its z-axis, then the resulting y- and x-axis,

${}^O_B R = R_z(\alpha)\ R_y(\beta)\ R_x(\gamma)$, (Equation 23)

$$\begin{cases} R_z(\alpha) := \begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ R_y(\beta) := \begin{bmatrix} cos(\beta) & 0 & sin(\beta) \\ 0 & 1 & 0 \\ -sin(\beta) & 0 & cos(\beta) \end{bmatrix} \\ R_x(\gamma) := \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\gamma) & -sin(\gamma) \\ 0 & sin(\gamma) & cos(\gamma) \end{bmatrix} \end{cases},$$

where α, β and γ are the yaw, pitch and roll Euler angles, respectively. With this representation, the tilt of the rigid body is captured by β and γ. Using equation (23), equation (5) can be written as

${}^{B}g = {}^{O}_{B}R^{T}\ {}^{O}g = R_x^T(\gamma)\ R_y^T(\beta)\ R_z^T(\alpha)\ {}^{O}g$. (Equation 24)

Using ${}^{O}g = \begin{bmatrix} 0 & 0 & g_0 \end{bmatrix}^T$ with gravity constant g_0 and the definitions of the rotation matrices, equation (24) simplifies to

$${}^{B}g = R_x^T(\gamma)\ R_y^T(\beta)\ {}^{O}g = g_0 \begin{bmatrix} -sin(\beta) \\ sin(\gamma)cos(\beta) \\ cos(\gamma)cos(\beta) \end{bmatrix}. \text{ (Equation 25)}$$

It follows that the z-Euler angle α is not observable from the accelerometer measurements.

Given the estimate of the gravity vector in equation (20), the *accelerometric estimates* for the y- and x-Euler angles at time k are:

$$\begin{cases} \hat{\beta}_a(k) = atan2(-{}^{B}\hat{g}_x(k), \sqrt{{}^{B}\hat{g}_y^2(k) + {}^{B}\hat{g}_z^2(k)}) \\ \hat{\gamma}_a(k) = atan2({}^{B}\hat{g}_y(k), {}^{B}\hat{g}_z(k)) \end{cases} \text{, (Equation 26)}$$

where $atan2$ is the four-quadrant inverse tangent. Note that one does not need to know the gravity constant g_0 for estimating tilt.

The variance of the angle estimates can be obtained from the variance of the gravity vector estimate, which can in turn be calculated from equation (18) and the SVD of P in equation (16).

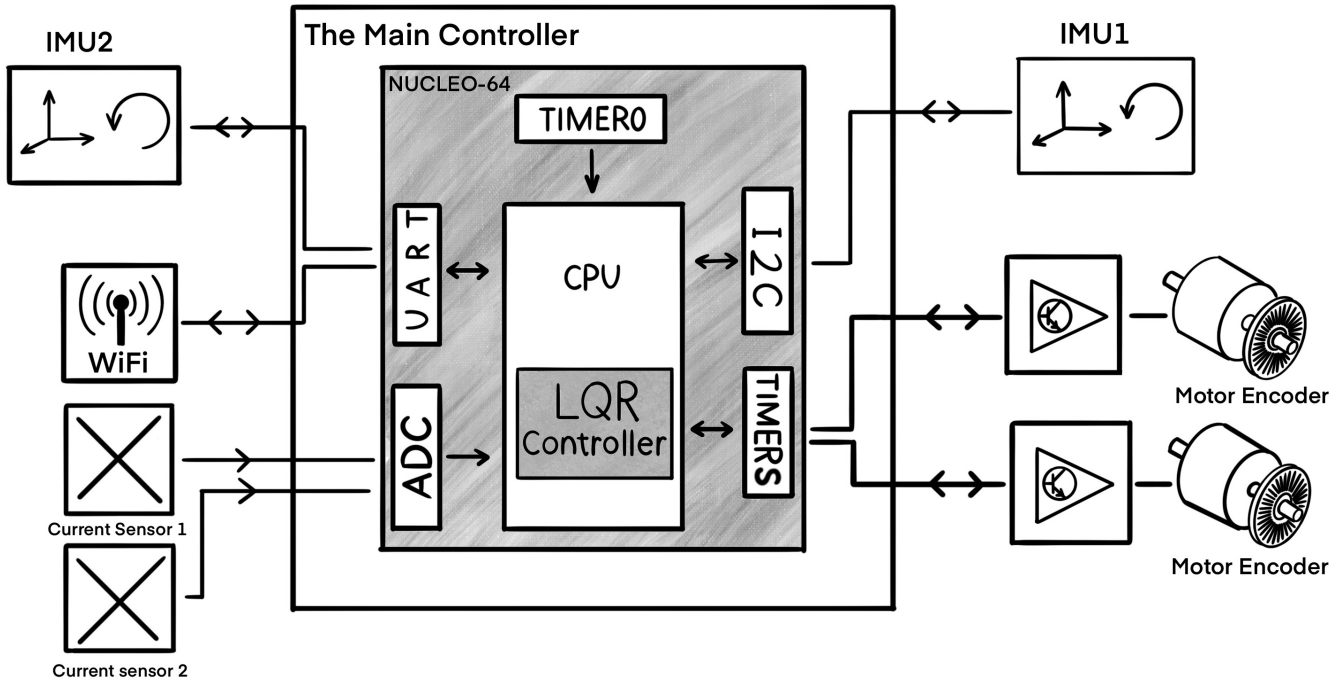
The Application of Tilt Estimation to the Balancing Unicycle

The tilt estimation algorithm is applied to the balancing unicycle robot. The passive structure of the unicycle is balanced on a wheel. There is a reaction wheel that is mounted on top of the rolling wheel, generating torque in the perpendicular direction for keeping the robot in equilibrium. The pitch and roll angles of the robot are estimated from measurements of two inertial measurement units (IMUs) with tri-axis accelerometers and rate gyros using the algorithm presented in the section above. The noise level of the estimate is further reduced by straightforward fusion with data from the rate gyros. Experimental results are provided in the next section.

The balancing unicycle robot consists of a rigid body in the shape of a rectangular cuboid with two orthogonally attached rotors. The objective is to balance the robot on the bottom wheel. In this configuration, the rigid body has three rotational degrees of freedom, and one translational degree of freedom. One can assume that the robot pivot does not slip due to friction, but rolls freely along a straight line. On the robot chassis, a pair of IMUs are mounted that measure accelerations and angular velocities each along three axes.

The wheels are actuated by DC motors and rotate relative to the robot structure. When the wheels rotate, they exert inertia reactional and ground reactional forces (by generating angular momenta and towing force) on the robot structure. An absolute encoder is used on each motor to measure the angle of a wheel relative to its mounting axle. Furthermore, the robot carries its own

power unit and a computer that is connected to the sensors and the DC motors. The computer is also connected to a WiFi module to broadcast all its local sensor measurements to the terminal of a workstation.



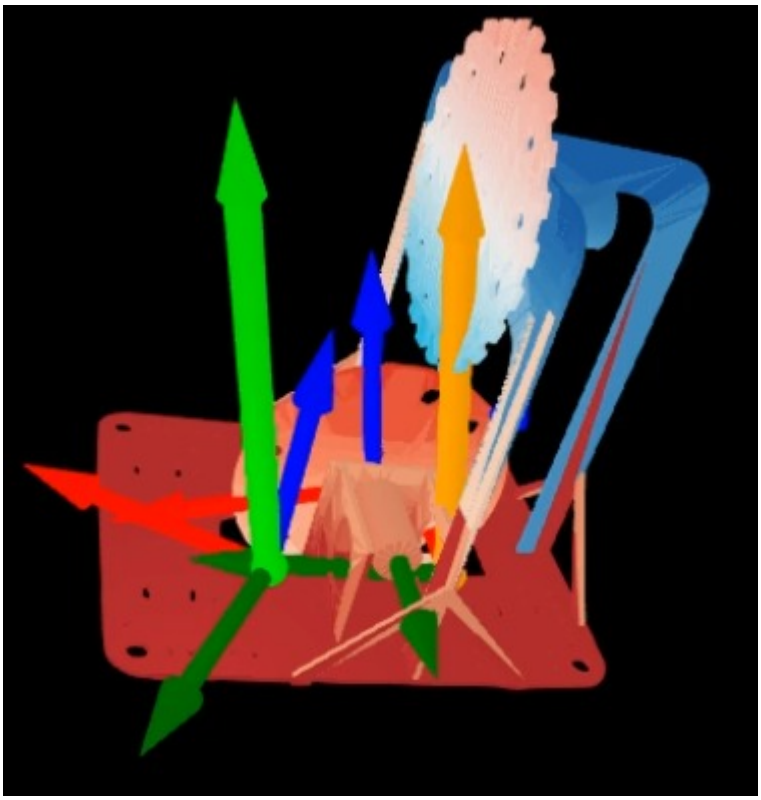
A state feedback controller is designed that stabalizes the unstable equilibrium of an upright standing robot. Estimates for the chassis angles are obtained from the tilt estimation algorithm of the section above. Although the details of the controller design are explained in the next section, a brief explanation of the control system follows.

Foe both wheels, an inner feedback loop is closed for the chassis velocity. The inner-loop controller, which is the critic loop in the Actor/Critic architecture, ensures that velocity commands are tracked at a faster rate than the natural dynamics of the chassis. This way, one can neglect nonlinear effects such as friction and backlash in the actuation mechanism. A linear dynamical model (a linear quadratic regulator) about the equilibrium that takes into account the effect of the inner loop is obtained using a two-timescale technique, where the outer loop updates the actor of the controller. Updating the control policy in the outer-loop results in a stabalizing controller.

The sensors can generate estimates of all system states: the wheels' angular velocities are calculated using measurements by encoders, estimates of the motors' electrical current rates result from a difference equation using measurements by Hall effect sensors, and estimates of the robot's pitch and roll angles and their rates and accelerations are derived from the IMU measurements. Note that for balancing, knowledge of yaw is not required. Hence, a centralized full-state feedback LQR controller can be designed for stabalizing the system.

The Implemetation of the Tilt Estimation Algorithm

The coordinate frame definitions and the locations of the two IMUs and the pivot point on the robot's chassis are indicated in the figure below.



The position vectors of the sensors are

$${}^{O}p_1 = \begin{bmatrix} -0.1400 & -0.0650 & -0.0620 \end{bmatrix}^T \text{ and}$$

$${}^{O}p_2 = \begin{bmatrix} -0.0400 & -0.0600 & -0.0600 \end{bmatrix}^T.$$

Also the position of the pivot point in the inertial coordinate frame is

$${}^{O}pivot = \begin{bmatrix} -0.097 & -0.1 & -0.032 \end{bmatrix}^T$$

With this data, matrix P can be constructed as in equation (10).

$$P \in \mathbb{R}^{4 \times L}$$

$$L = 2$$

$$P \in \mathbb{R}^{4 \times 2}$$

the unicycle graphical dashboard for taking the measurements

$${}^{O}p_1 = \begin{bmatrix} p_{1x} \\ p_{1y} \\ p_{1z} \end{bmatrix}$$

$$\begin{bmatrix} p_{2x} \end{bmatrix}$$

$$^{O}p_2 = \begin{bmatrix} p_{2y} \\ p_{2z} \end{bmatrix}$$

$$^{O}pivot = \begin{bmatrix} pivot_x \\ pivot_y \\ pivot_z \end{bmatrix}$$

$$P = \begin{bmatrix} 1 & 1 \\ p_{1x} - pivot_x & p_{2x} - pivot_x \\ p_{1y} - pivot_y & p_{2y} - pivot_y \\ p_{1z} - pivot_z & p_{2z} - pivot_z \end{bmatrix}$$

Applying the partitioned estimation lemma yields the optimal fusion matrix for estimating the gravity vector in the robot's body frame,

$$X = P^T(PP^T)^{-1} \in \mathbb{R}^{L \times 4} \longrightarrow X \in \mathbb{R}^{2 \times 4},$$

$$X_1^* = \begin{bmatrix} 0.586913 & -11.3087 & 0.747681 & 0.0 \\ 0.446183 & 8.92749 & -3.54337 & 0.0 \end{bmatrix}.$$

In the implementation of the algorithm, all accelerometer measurements are rotated to the body frame and stacked into the matrix $M(k)$ as in equation (8) at each time step. Then, equations (20) and (26) are implemented to obtain the accelerometric estimates for the pitch and roll angles, $\hat{\beta}_a(k)$ and $\hat{\gamma}_a(k)$.

In order to reduce the noise level of the accelerometer-based estimates, a straightforward scheme for data fusion with the tri-axis rate gyro measurements may be used. Let $r(k) \in \mathbb{R}^3$ denote the body angular rate at time k, which is directly measured by a gyro that is mounted on the body. Thus, an estimate $\hat{r}(k)$ of this quantity may be obtained by averaging the measurements of all two gyros. The body rates are transformed to Euler angular rates by

$$\begin{bmatrix} \dot{\hat{\alpha}}(k) \\ \dot{\hat{\beta}}(k) \\ \dot{\hat{\gamma}}(k) \end{bmatrix} = \begin{bmatrix} 0 & sin(\hat{\gamma})/cos(\hat{\beta}) & cos(\hat{\gamma})/cos(\hat{\beta}) \\ 0 & cos(\hat{\gamma}) & -sin(\hat{\gamma}) \\ 1 & sin(\hat{\gamma})tan(\hat{\beta}) & cos(\hat{\gamma})tan(\hat{\beta}) \end{bmatrix} \hat{r}(k), \text{ (Equation 27)}$$

which requires estimates of the Euler angles $\hat{\beta}$ and $\hat{\gamma}$. For a straightforward implementation, the most recent estimate may be used as an approximation, i.e. $\hat{\beta} = \hat{\beta}(k-1)$ and $\hat{\gamma} = \hat{\gamma}(k-1)$.

Integrating the rate estimates in equation (27) yields estimates for the Euler angles that are based on the rate gyro measurements. Thus, the accelerometer- and gyro-based estimates can be fused to obtain a better overall estimate of the robot pitch and roll angles,

$$\begin{cases} \hat{\beta}(k) = \kappa_1 \hat{\beta}_a(k) + (1 - \kappa_1)(\hat{\beta}(k-1) + T\dot{\hat{\beta}}(k)) \\ \hat{\gamma}(k) = \kappa_2 \hat{\gamma}_a(k) + (1 - \kappa_2)(\hat{\gamma}(k-1) + T\dot{\hat{\gamma}}(k)) \end{cases} \text{, (Equation 28)}$$

where T is the sampling time (the same as `dt` in the LQR struct) and κ_1 and κ_2 are tuning parameters that may be chosen such that the variance of the estimate is minimized given the noise specifications of accelerometers and rate gyros. For the application presented in this project, $\kappa_1 = \kappa_2 = 0.01$ was used.

The Experimental Results

The accelerometer-based tilt estimator was implemented on the Balancing Unicycle as described above. In the Balancing Unicycle application, the bias of the Euler angle estimates is corrected prior to operation in a calibration procedure, which accounts to first order for the accelerometer biases (see the struct fields `accXOffset`, `accYOffset` and `accZOffset`).

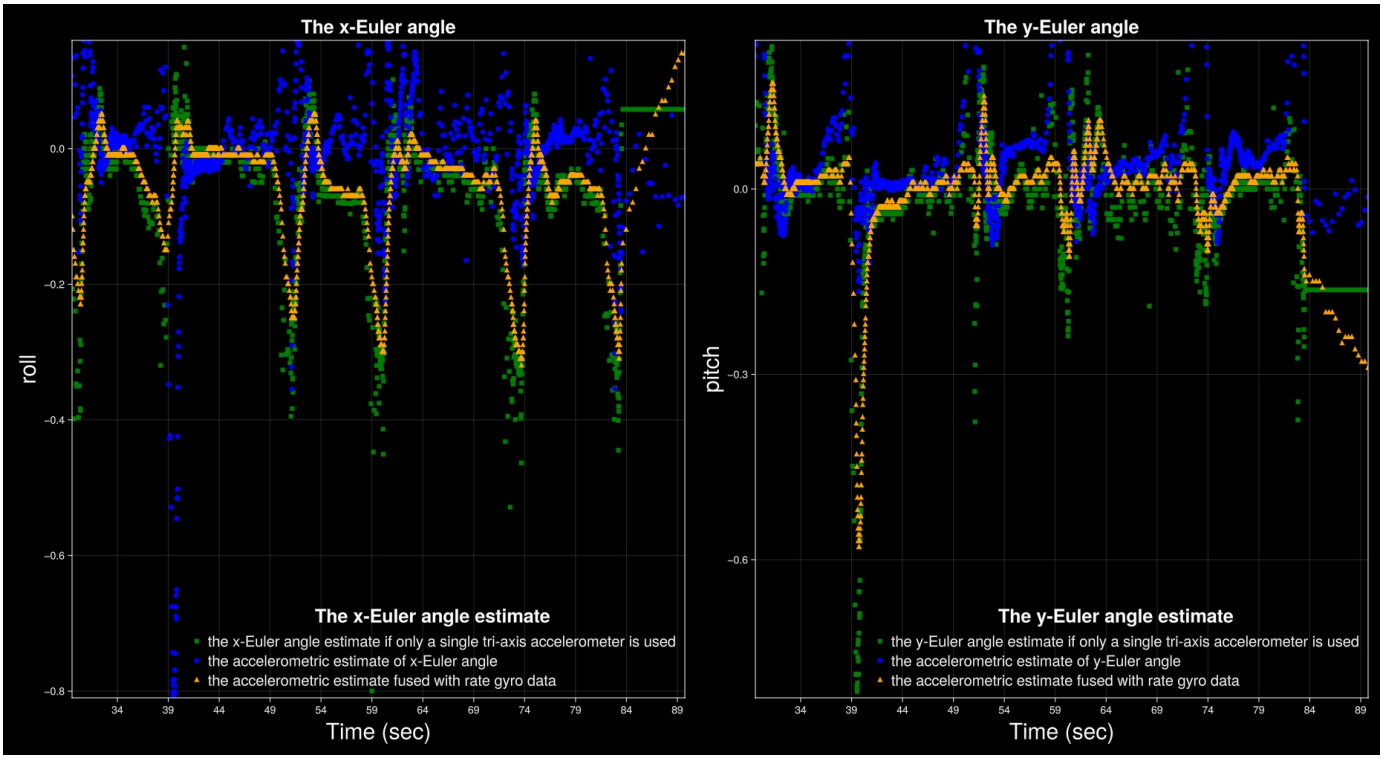
```
typedef struct
{
  int16_t accXOffset;
  int16_t accYOffset;
  int16_t accZOffset;
  float accXScale;
  float accYScale;
  float accZScale;
  int16_t gyrXOffset;
  int16_t gyrYOffset;
  int16_t gyrZOffset;
  float gyrXScale;
  float gyrYScale;
  float gyrZScale;
  int16_t rawAccX;
  int16_t rawAccY;
  int16_t rawAccZ;
  int16_t rawGyrX;
  int16_t rawGyrY;
  int16_t rawGyrZ;
  float accX;
  float accY;
  float accZ;
  float gyrX;
  float gyrY;
  float gyrZ;
  float roll;
  float pitch;
  float yaw;
  float roll_velocity;
  float pitch_velocity;
  float yaw_velocity;
  float roll_acceleration;
  float pitch_acceleration;
```

```
  float yaw_acceleration;
  Mat3 B_A_R; // The rotation of the local frame of the sensor i to the robot fr
  Vec3 R;     // accelerometer sensor measurements in the local frame of the sen
  Vec3 _R;    // accelerometer sensor measurements in the robot body frame
  Vec3 G;     // gyro sensor measurements in the local frame of the sensors
  Vec3 _G;    // gyro sensor measurements in the robot body frame
} IMU;
```

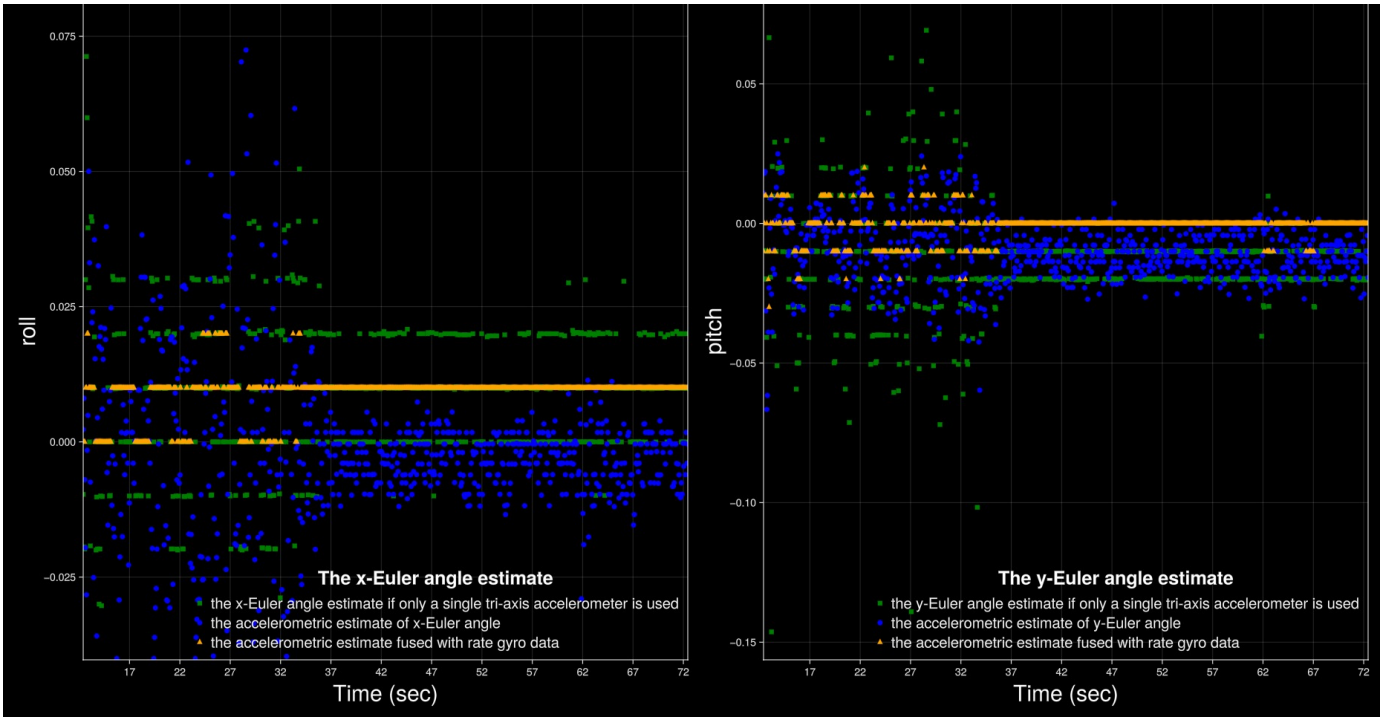
To demonstrate the accelerometer-based tilt estimator, the robot was put on the rolling wheel and moved by hand (and also by the LQR controller) about the upright equilibrium position, which corresponds to the Euler angles $\gamma \approx 0$ and $\beta \approx 0$. The results are shown in the following figures. Clearly, the accelerometric estimate is accurate both for slow and fast motion of the robot. In contrast to the tilt estimation method, the same experiment is shown, but now only a single tri-axis accelerometer (sensor i = 1) is used to observe the gravity vector. When the robot is static (away from peaks and valleys), the estimate is accurate. However, when the robot is being moved the estimate suffers from the dynamic terms that act as disturbances to the static estimator. This demonstrates that the dynamics are not negligible. For the closed-loop operation of the system, i.e. for balancing the robot using the LQR model, the improved estimate for the robot angles (up triangles in orange) using both accelerometer and rate gyro data was used.

Estimation data during balancing of the robot: comparison of Euler angle estimates if only a single tri-axis accelerometer is used (green) to the accelerometric estimate of x- and y-Euler angles (blue), and to the accelerometric estimate fused with rate gyro data (orange).

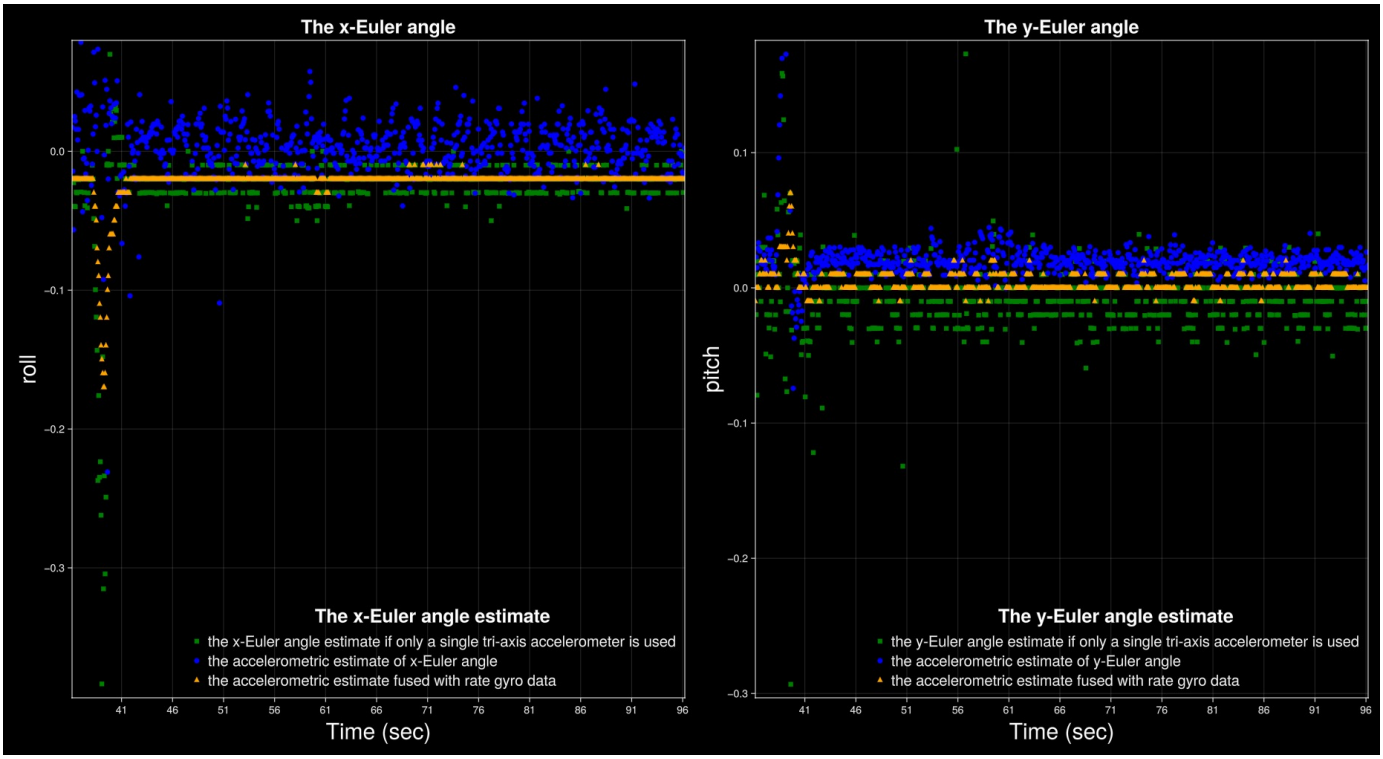
Data sample 1:



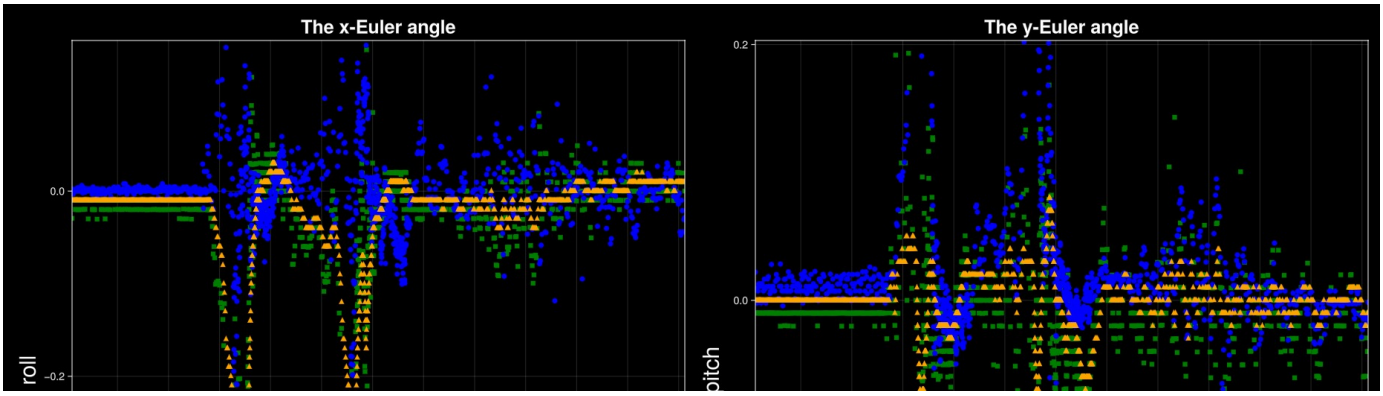
Data sample 2:

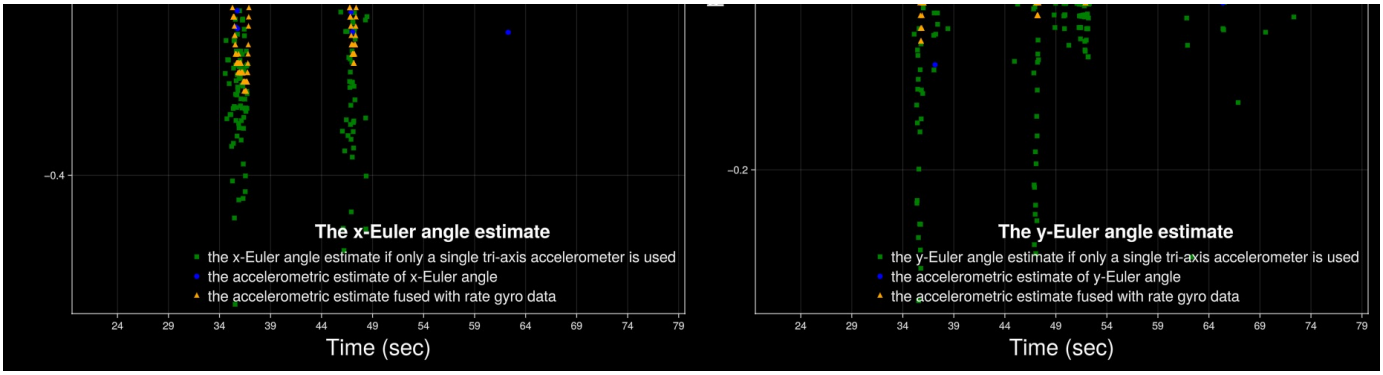


Data sample 3:



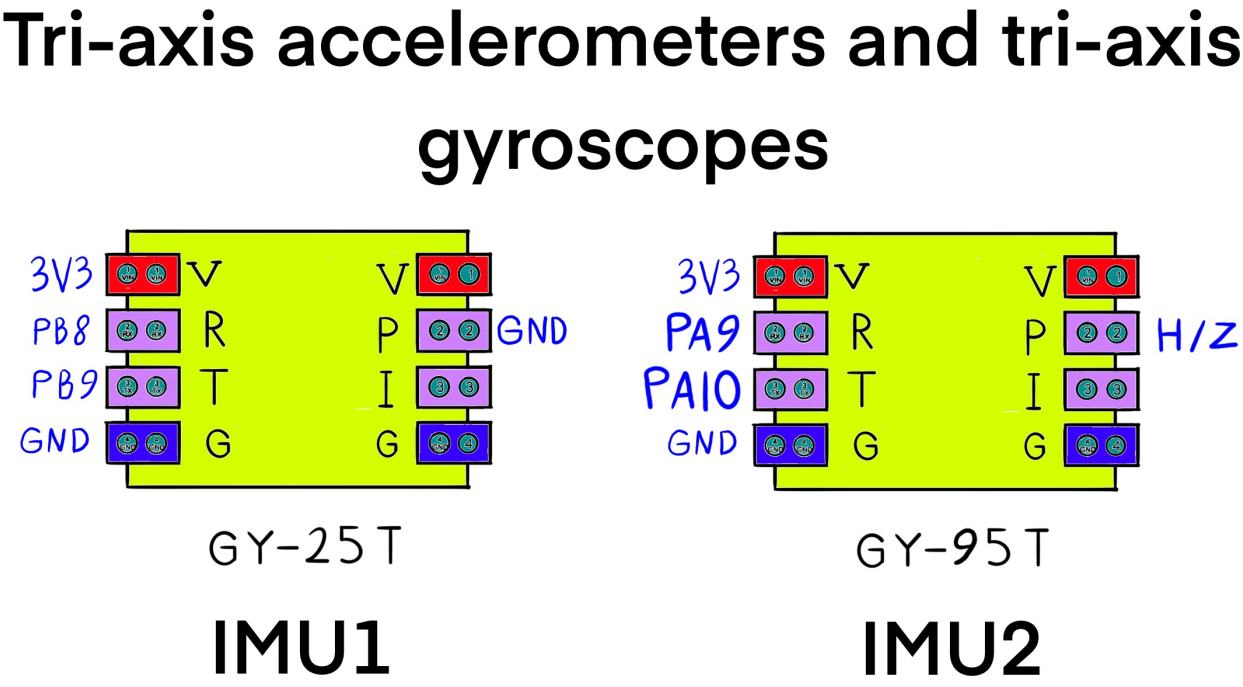
Data sample 4:





The Micro-Controller Program

The function `updateIMU` provides the main source of data for the objective of the system. Through this function, the Micro-Controller Unit (MCU) talks to the Inertial Measurement Unit (IMU) modules 1 and 2 for updating the roll and pitch angles along with their first and second derivatives. This is done by calling the function and giving it a pointer to the Linear Quadratic Regulator (LQR) model object. Although both IMUs are used for tilt estimation, the final result is assigned to the field of IMU 1. This function encapsulates matrix-vector multiplications for coordinate transformations, the singular value decomposition for obtaining the gravity vector, and sensor fusion between the tri-axis accelerometers and the tri-axis gyroscopes. Knowing about the position and orientation of each IMU with respect to the body, the function excludes linear accelerations from calculations. So, `UpdateIMU` gathers the latest inertial measurements form multiple sensor units and computes the roll and pitch angles using known parameters of the system configuration.



In terms of connectivity, The MCU peripheral USART1 is used to talk to IMU #2 (GY-95T). Set the baudrate of uart1 to 115200 Bits/s for the GY-95 IMU module. Set the Pin6 (PS: IIC/USART output mode selection) of IMU #2 (GY-25T) to zero, in order to use the I2C protocol. The I2C clock speed is set at 100000 Hz in the stanard mode. For saving MCU clock cycles and time, added a DMA request with USART1_RX and DMA2 Stream 2 from peripheral to memory and low priority. The mode is circular and the request call is made once in the main function by passing the usart1 handle and the receive buffer. The request increments the address of memory. The data width is one Byte for both the preipheral and memory.

```
void updateIMU(LinearQuadraticRegulator *model)
{
  updateIMU1(&(model->imu1));
  updateIMU2(&(model->imu2));
  setIndexVec3(&(model->imu1.R), 0, model->imu1.accX);
  setIndexVec3(&(model->imu1.R), 1, model->imu1.accY);
  setIndexVec3(&(model->imu1.R), 2, model->imu1.accZ);
  setIndexVec3(&(model->imu2.R), 0, model->imu2.accX);
  setIndexVec3(&(model->imu2.R), 1, model->imu2.accY);
  setIndexVec3(&(model->imu2.R), 2, model->imu2.accZ);

  for (int i = 0; i < 3; i++)
  {
    setIndexVec3(&(model->imu1._R), i, 0.0);
    setIndexVec3(&(model->imu2._R), i, 0.0);
  }

  for (int i = 0; i < 3; i++)
  {
    for (int j = 0; j < 3; j++)
    {
      setIndexVec3(&(model->imu1._R), i, getIndexVec3(model->imu1._R, i) + getIn
      setIndexVec3(&(model->imu2._R), i, getIndexVec3(model->imu2._R, i) + getIn
    }
  }

  for (int i = 0; i < 3; i++)
  {
    setIndexMat32(&(model->Matrix), i, 0, getIndexVec3(model->imu1._R, i));
    setIndexMat32(&(model->Matrix), i, 1, getIndexVec3(model->imu2._R, i));
  }

  for (int i = 0; i < 3; i++)
  {
    for (int j = 0; j < 4; j++)
    {
```

```
      setIndexMat34(&(model->Q), i, j, 0.0);
      for (int k = 0; k < 2; k++)
      {
        setIndexMat34(&(model->Q), i, j, getIndexMat34(model->Q, i, j) + getInde
      }
    }
  }
  setIndexVec3(&(model->g), 0, getIndexMat34(model->Q, 0, 0));
  setIndexVec3(&(model->g), 1, getIndexMat34(model->Q, 1, 0));
  setIndexVec3(&(model->g), 2, getIndexMat34(model->Q, 2, 0));
  model->beta = atan2(-getIndexVec3(model->g, 0), sqrt(pow(getIndexVec3(model->g
  model->gamma = atan2(getIndexVec3(model->g, 1), getIndexVec3(model->g, 2));

  setIndexVec3(&(model->imu1.G), 0, model->imu1.gyrX);
  setIndexVec3(&(model->imu1.G), 1, model->imu1.gyrY);
  setIndexVec3(&(model->imu1.G), 2, model->imu1.gyrZ);
  setIndexVec3(&(model->imu2.G), 0, model->imu2.gyrX);
  setIndexVec3(&(model->imu2.G), 1, model->imu2.gyrY);
  setIndexVec3(&(model->imu2.G), 2, model->imu2.gyrZ);

  for (int i = 0; i < 3; i++)
  {
    setIndexVec3(&(model->imu1._G), i, 0.0);
    setIndexVec3(&(model->imu2._G), i, 0.0);
  }

  for (int i = 0; i < 3; i++)
  {
    for (int j = 0; j < 3; j++)
    {
      setIndexVec3(&(model->imu1._G), i, getIndexVec3(model->imu1._G, i) + getIn
      setIndexVec3(&(model->imu2._G), i, getIndexVec3(model->imu2._G, i) + getIn
    }
  }
  for (int i = 0; i < 3; i++)
  {
    setIndexVec3(&(model->r), i, (getIndexVec3(model->imu1._G, i) + getIndexVec3
  }

  setIndexMat3(&(model->E), 0, 0, 0.0);
  setIndexMat3(&(model->E), 0, 1, sin(model->gamma) / cos(model->beta));
  setIndexMat3(&(model->E), 0, 2, cos(model->gamma) / cos(model->beta));
  setIndexMat3(&(model->E), 1, 0, 0.0);
  setIndexMat3(&(model->E), 1, 1, cos(model->gamma));
  setIndexMat3(&(model->E), 1, 2, -sin(model->gamma));
  setIndexMat3(&(model->E), 2, 0, 1.0);
  setIndexMat3(&(model->E), 2, 1, sin(model->gamma) * tan(model->beta));
  setIndexMat3(&(model->E), 2, 2, cos(model->gamma) * tan(model->beta));
```

```
  for (int i = 0; i < 3; i++)
  {
    setIndexVec3(&(model->rDot), i, 0.0);
  }

  for (int i = 0; i < 3; i++)
  {
    for (int j = 0; j < 3; j++)
    {
      setIndexVec3(&(model->rDot), i, getIndexVec3(model->rDot, i) + getIndexMat
    }
  }

  model->fusedBeta = model->kappa1 * model->beta + (1.0 - model->kappa1) * (mode
  model->fusedGamma = model->kappa2 * model->gamma + (1.0 - model->kappa2) * (mo
  model->imu1.yaw += model->dt * getIndexVec3(model->rDot, 0);

  float _roll = model->fusedBeta;
  float _pitch = -model->fusedGamma;
  float _roll_velocity = ((getIndexVec3(model->rDot, 1) / 180.0 * M_PI) + (_roll
  float _pitch_velocity = ((-getIndexVec3(model->rDot, 2) / 180.0 * M_PI) + (_pi
  model->imu1.roll_acceleration = _roll_velocity - model->imu1.roll_velocity;
  model->imu1.pitch_acceleration = _pitch_velocity - model->imu1.pitch_velocity;
  model->imu1.roll_velocity = _roll_velocity;
  model->imu1.pitch_velocity = _pitch_velocity;
  model->imu1.roll = _roll;
  model->imu1.pitch = _pitch;
}
```

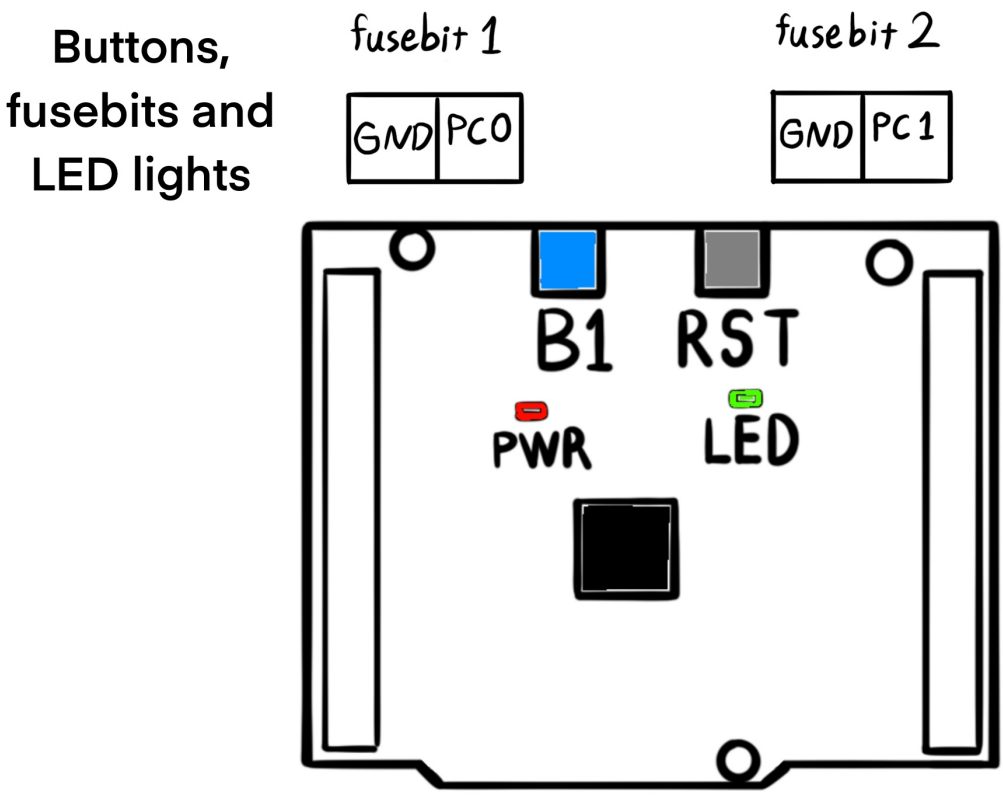
Stepping Through the Implementation

In this section, we step through the implementation of the robot's controller in the order of execution. The controller is implemented in the C programming language. It runs on a STM32F401RE mictocontroller, which is clocked at 84 MHz. Starting from first principles, there are at least two loops in a reinforcement learning program: the actor loop and the critic loop. The critic loop operates at a faster timescale and finds filter coefficients by taking actions, making mesurements and computing a recursive algorithm. In contrast, the actor loop is slower and updates the control policy, which is a function that produces actions. Even though actions are taken in the critic loop, the feedback policy function is the same across multiple runs of the loop. The actor loop is where the feedback policy is updated as a function of the latest set of filter coefficients. The state estimations and matrix parameters of the controller are stored in a data structure. The `LinearQuadraticRegulator` type is instantiated and initialized once, before either of the loops begin execution.

```
// Represents a Linear Quadratic Regulator (LQR) model.
typedef struct
{
  Mat12 W_n;                          // filter matrix
  Mat12 P_n;                          // inverse autocorrelation matrix
  Mat210 K_j;                         // feedback policy
  Vec12 dataset;                      // (x_k, u_k)
  Vec12 z_n;                          // z_n in RLS
  Vec12 g_n;                          // g_n in RLS
  Vec12 alpha_n;                      // alpha_n in RLS
  float x_n_dot_z_n;                  // the inner product of the x_n (dataset)
  int j;                              // step number
  int k;                              // time k
  int n;                              // x_k ∈ R^n
  int m;                              // u_k ∈ R^m
  float lambda;                       // exponential wighting factor
  float delta;                        // value used to intialize P(0)
  int active;                         // is the model controller active
  float CPUClock;                     // the CPU clock
  float dt;                           // period in seconds
  float reactionDutyCycle;            // reaction wheel's motor PWM duty cycle
  float rollingDutyCycle;             // rolling wheel's motor PWM duty cycle
  float reactionDutyCycleChange;      // the maximum incremental change in the
  float rollingDutyCycleChnage;       // the maximum incremental change in the
  float clippingValue;                // the clipping value for any of the P ma
  float clippingFactor;               // the coefficient by which the P matrix
  float rollSafetyAngle;              // the roll angle in radian beyond which
  float pitchSafetyAngle;             // the pitch angle in radian beyond which
  float kappa1;                       // tuning parameters to minimize estimate
  float kappa2;                       // tuning parameters to minimize estimate
  int maxEpisodeLength;               // the maximum number of interactions wit
  int logPeriod;                      // the period between printing two log me
  int logCounter;                     // the number of control cycles elpased s
  int maxOutOfBounds;                 // the maximum number of consecutive cycl
  int outOfBoundsCounter;             // the number of consecutive times when e
  float beta;                         // y-Euler angle (pitch)
  float gamma;                        // x-Euler angle (roll)
  float fusedBeta;                    // y-Euler angle (pitch) as the result of
  float fusedGamma;                   // x-Euler angle (roll) as the result of
  int noiseQuotient;                  // the quotient of the random number for
  float noiseScale;                   // the scale of by which the remainder of
  float time;                         // the time that has elapsed since the st
  float changes;                      // the magnitude of the changes to the fi
  float convergenceThreshold;         // the threshold value of the changes to
  Mat34 Q;                            // The matrix of unknown parameters
  Vec3 r;                             // the average of the body angular rate f
  Vec3 rDot;                          // the average of the body angular rate i
  Mat3 E;                             // a matrix transfom from body rates to E
```

```
  Mat24 X;                                // The optimal fusion matrix
  Mat32 Matrix;                           // all sensor measurements combined
  Vec3 g;                                 // The gravity vector
  Mat2 Suu;                               // The input-input kernel
  Mat2 SuuInverse;                        // the inverse of the input-input kernel
  Mat210 Sux;                             // the input-state kernel
  Vec2 u_k;                               // the input vector
  IMU imu1;                               // the first inertial measurement unit
  IMU imu2;                               // the second inertial measurement unit
  Encoder reactionEncoder;                // the reaction wheel encoder
  Encoder rollingEncoder;                 // the rolling wheel encoder
  CurrentSensor reactionCurrentSensor; // the reaction wheel's motor current sen
  CurrentSensor rollingCurrentSensor;  // the rolling wheel's motor current sens
} LinearQuadraticRegulator;
```

There are two fuse bits on the robot for configuration without flashing a program. The first one is connected to the port C of the general purpose input / output, pin 0. The fuse bit is active whenever the connected pin is grounded. The fuse bit deactivates the linear quadratic regulator by clearing the `active` field as a flag in the model structure. Even though the status of the fuse bit 0 is necessary to activate the model, it is not a sufficient condition. The user must connect the fuse bit and also push a blue push button once on the robot for activating the model. The push button is the same blue button that is found on the NUCLEOF401RE board. These two conditions are chained together for safety reasons. If the model is not active, then the robot must stop moving by calling the function `resetActuators`.



```
elapsedTime1 = DWT->CYCCNT;
if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_0) == 0)
{
  if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == 0)
  {
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
    model.active = 1;
  }
}
else
{
  model.active = 0;
  resetActuators(&model);
}
```

When the reaction wheel unicycle falls over, the roll and pitch angles of the chassis with respect to the pivot point exceed ten degrees. It makes sense to disable the actuators after a fall has been detected to both save energy and minimize physical shock to gearboxes. The lower and upper bounds on the roll and pitch angles are combined using the logical "or" operator `||` with the episode counter so that the model stops running after the maximum number of interactions with the environment, the total steps in an episode. A fall or a certain number of interactions, whichever comes first, must cause the model to deactivate on its own. When the model is deactivated, the green light on the NUCLEOF401RE turns on to signify that the controller is no longer active. The user has four options whenever the green LED lights up:

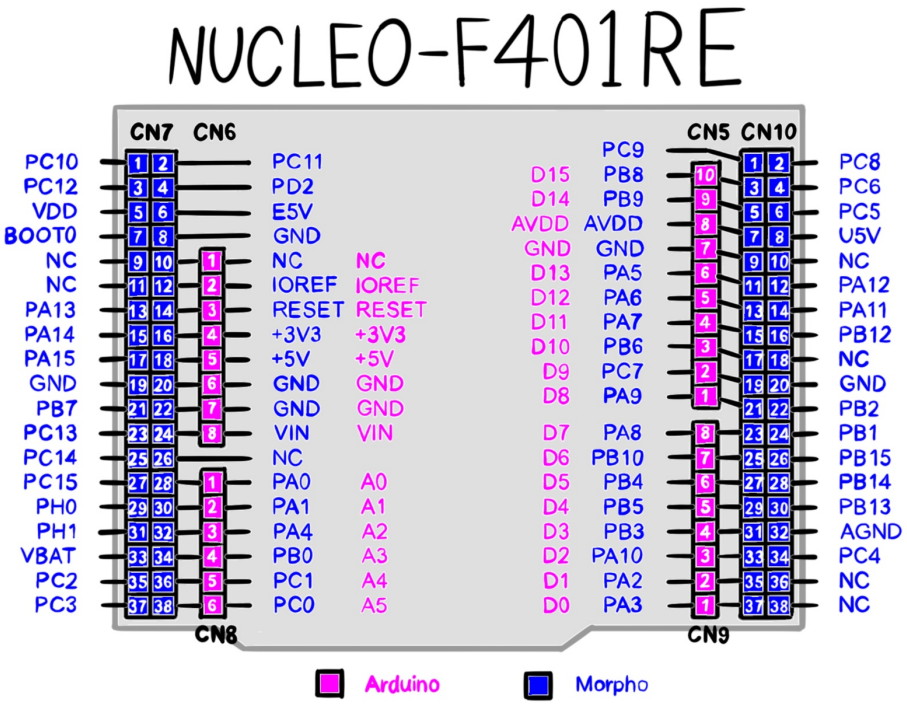
1. Pick the robot up and make it stand upright, before pushing the blue push button to run again.
2. Switch the power button on the chassis to condition zero, in order to power off the robot.
3. Connect to the robot WiFi network and execute the following command in the terminal for printing the logs. Print `uart6` serial messages by executing: `nc 192.168.4.1 10000`
4. Activate the Porta.jl environment in a Julia REPL and then run the linked script for visualizing the logs: Unicycle

Since the robot is portable and has a feedback loop related to the motion of its body, violating the safety angle bounds does not immediately disable the model. Instead, the `outOfBoundsCounter` is incremented every time the safety conditions are violated and is decremented otherwise. Then the model is deactivated if the out of bounds counter is greater than `maxOutOfBounds`. This approach reduces the probability that a discontinous state estimation is able to trigger deactivation.

```
if (fabs(model.imu1.roll) > model.rollSafetyAngle || fabs(model.imu1.pitch) > mo
{
```

```
  model.outOfBoundsCounter = model.outOfBoundsCounter + 1;
}
else
{
  model.outOfBoundsCounter = fmax(0, model.outOfBoundsCounter - 1);
}

if (model.outOfBoundsCounter > model.maxOutOfBounds)
{
  model.active = 0;
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
}
```

The microcontroller is built around a Cortex-M4 with Floating Point Unit (FPU) core, which contains hardware extensions for debugging features. The debug extensions allow the core to be stopped either on a given instruction fetch (breakpoint), or on data access (watchpoint). When stopped, the core's internal state and the system's external state may be examined. Once examination is complete, the core and the system may be restored and program execution resumed.



The ARM Cortex-M4 with FPU core provides integrated on-chip debug support. One of the debug features is called Data Watchpoint Trigger (DWT). The DWT unit provides a means to give the

number of clock cycles. The DWT register `CYCCNT` counts the number of clock cycles. The period of a control loop is required in the application for integrating the gyroscopic angle rates. If we count the number of clocks twice: one time before the loop begins and one time after the loop ends, then we can find the time period that it takes to complete a control loop. In the beginning, we count the number of clocks by assigning the register value to a local variable called `t1`.

At the end of the control loop, where the model has taken one step forward, it is time to count the number of the processor's clock cycles for a second time for measuring delta `t`. At this point, by assigning the value of the `DWT` counter register to the variable `t2` we can know how many cycles are there between `t1` and `t2`. Then divide the difference by the number of Central Processing Unit (CPU) clock cycles per second `cpuClock` for finding the period of the control loop. The field `dt` of the model struct saves the control period.

If the model is set to active, then the controller takes one step forward. The function `stepForward` takes as argument a pointer to the model, mainly because two of its fields require persistent memory across runs: the filter matrix `W_n` and the inverse autocorrelation matrix `P_n`. The system state estimation is done by calling `updateSensors`. But since we extend the meaning of the value function to the quality function $Q(x, u)$, the states x_k are appended by the inputs u_k, which in turn are computed by calling `computeFeedbackPolicy`. The function call `applyFeedbackPolicy` applies the action of the feedback policy, changing the angular velocity of the motors.

$$Q(x, u) = Q(z) = W^T \phi(z)$$

$$x_k \in \mathbb{R}^n,\ u_k \in \mathbb{R}^m$$

In the case where the model is active, the critic loop is run for a few times before the policy is updated by calling the `updateControlPolicy` function. However, when the model in not active, neither the critic loop nor the actor loop are executed. During the inactive mode of operation, the program makes measurements by calling the functions `updateSensors` and `computeFeedbackPolicy`, and resets the actuators by calling the function `resetActuators`.

```
if (model.active == 1)
{
  t1 = DWT->CYCCNT;
  updateSensors(&model);
  computeFeedbackPolicy(&model);
  applyFeedbackPolicy(&model);
  stepForward(&model);
  if (fabs(model.changes) < 2.0)
  {
    updateControlPolicy(&model);
  }
  model.logCounter = model.logCounter + 1;
  t2 = DWT->CYCCNT;
  diff = t2 - t1;
```

```
  model.dt = (float)diff / model.CPUClock;
}
else
{
  model.logPeriod = 80;
  t1 = DWT->CYCCNT;
  resetActuators(&model);
  updateSensors(&model);
  computeFeedbackPolicy(&model);
  model.logCounter = model.logCounter + 1;
  t2 = DWT->CYCCNT;
  diff = t2 - t1;
  model.dt = (float)diff / model.cpuClock;
}
```

In order to monitor the controller and debug issues, we write the logs periodically to the standard input / output console. The variable `log_counter` is incremented by one every control cycle. Then, the log counter variable `logCounter` is compared to the constnt `logPeriod` for finding out if a cycle should be logged. But it is not a sufficient condition for logging, because a second fuse bit is also rquired for permission to log. The second fuse bit is connected to the port C of the general purpose input / output, pin 1. Whenever the fuse bit pin is grounded, it is activated. Once the fuse bit is active, the local transmission flag `transmit` is set at the relevant control cycle count. The reason for `logPeriod` is to limit the total number of logs per second, as the Micro-Controller Unit (MCU) is too fast for a continuous report. And the second fuse bit is there to turn off logging for saving time, as log transmission takes time away from the control processes. So by using the logical "and" operator `&&` we can combine the logging period condition with the logging fuse bit, in order to manage the frequnecy of transmissions.

```
if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_1) == 0)
{
  model.logPeriod = 5;
}
else
{
  model.logPeriod = 40;
}
if (model.logCounter > model.logPeriod)
{
  transmit = 1;
}
```

If the `transmit` variable is equal to one, then the log counter `logCounter` is cleard along with the `transmit` variable, before transmission. The function `sprintf` is called with a message buffer `MSG` and a formatted string to populate the buffer with numbers. A log message can be anything, but for finding the matrix of known parameters in tilt estimation, the accelerometrs data must be

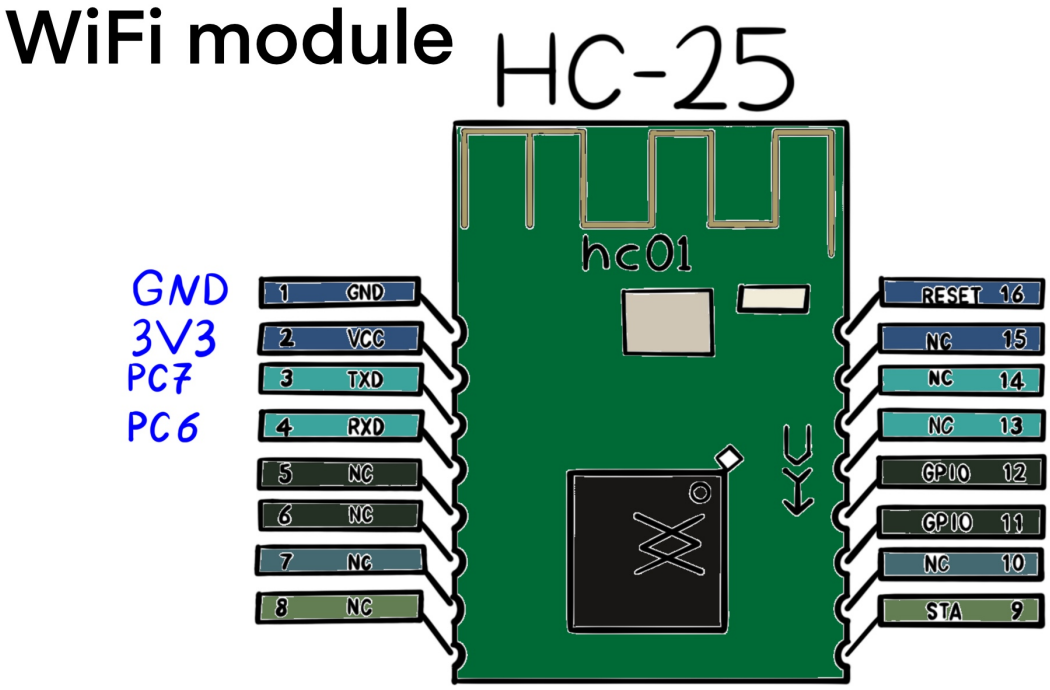
included. After the message is composed, it is given as an argument to the function `HAL_UART_Transmit`, which stands for: Hardware Abstraction Layer, Universal Asynchronous Receiver / Transmitter, Transmit. The function also requires a pointer to `uart6`, which is a micro-controller peripheral for serial communications, and the size of the message buffer, along with a time out delay. Printing and transmitting the log finishes the actor loop. The console on the other side of tranmission should receive a line like this: `AX1: -0.01, AY1: 1.03, AZ1: 0.00, | AX2: -0.05, AY2: 0.97, AZ2: -0.04, | roll: 0.03, pitch: -1.59, | encT: 0.46, encB: 4.31, | j: 1038.000000, | x0: -0.02, x1: -0.12, x2: 0.03, x3: -1.22, x4: -0.90, x5: 0.05, x6: 0.09, x7: -0.04, x8: 0.00, x9: -0.01, x10: 0.00, x11: 0.00, | P0: -96.08, P1: 0.27, P2: 2.04, P3: 1.18, P4: 1.36, P5: 0.34, P6: 0.03, P7: 0.77, P8: 0.08, P9: 0.57, P10: 62.48, P11: 62.48, dt: 0.001947`.

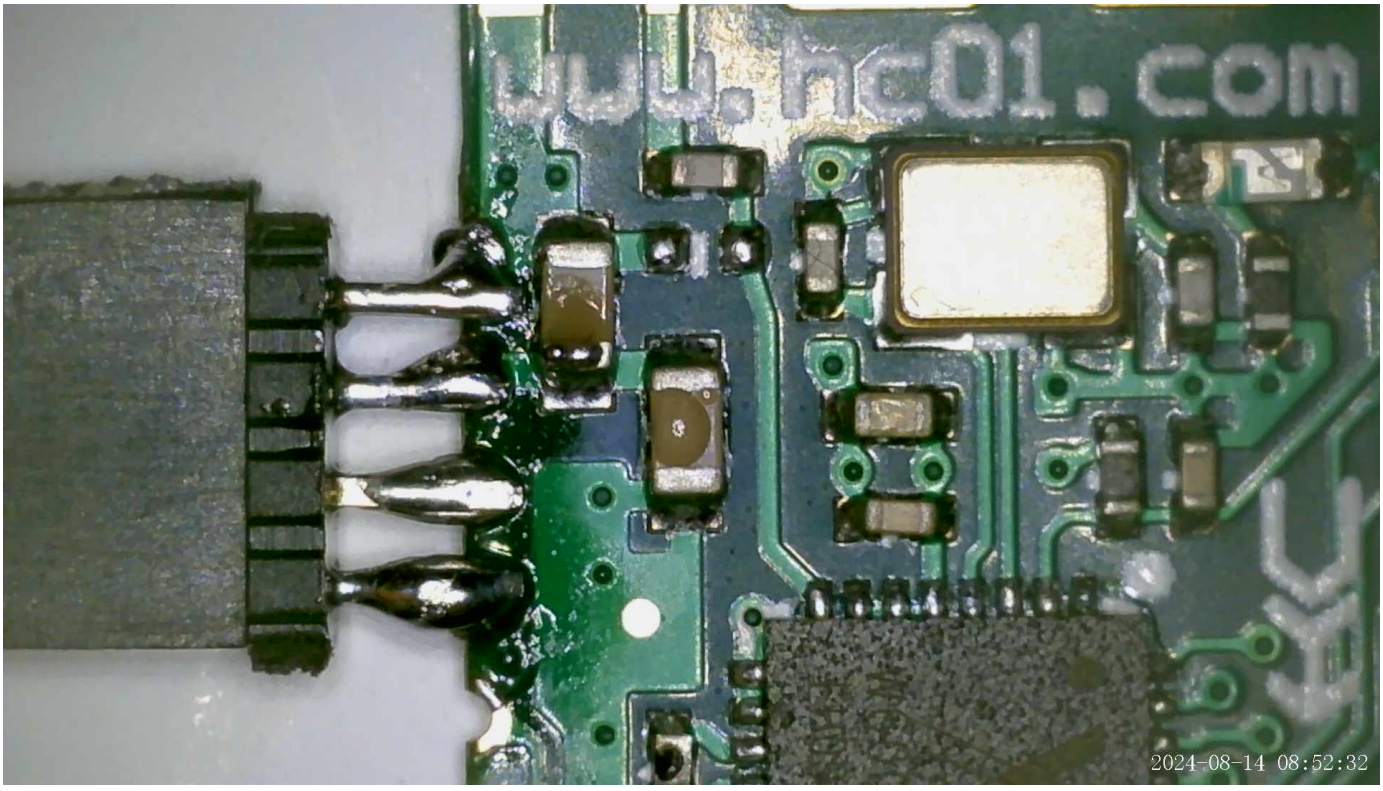
You can visualize this example message using the Unicycle script. The example includes: the tri-axis acceleromer measurements of IMU 1 and IMU 2, the roll and pitch angles after the primary and secondary sensor fusions, the absolute position of both rotary encoders, and the diagonal entries of the inverse auto-correlation matrix `P_n`. Different messages can be composed for different use cases, for example printing raw sensor readings for calibrating the zero point and the scale of the accelerometers axes.

```
if (transmit == 1)
{
  t1 = DWT->CYCCNT;
  transmit = 0;
  model.logCounter = 0;

  sprintf(MSG,
          "active: %0.1f, changes: %0.2f, AX1: %0.2f, AY1: %0.2f, AZ1: %0.2f, |
          (float)model.active, model.changes, model.imu1.accX, model.imu1.accY,

  HAL_UART_Transmit(&huart6, MSG, sizeof(MSG), 1000);
  t2 = DWT->CYCCNT;
  diff = t2 - t1;
  model.dt += (float)diff / model.CPUClock;
}
// Rinse and repeat :)
elapsedTime2 = DWT->CYCCNT;
elapsedTime = elapsedTime2 - elapsedTime1;
model.time += (float)elapsedTime / model.CPUClock;
```

In order to enable the function `sprintf` to use floating point numbers, do the following steps:

1. Open the file `gcc-arm-none-eabi.cmake` that is created by CubeMX.
2. Add the option `-u _printf_float` to `CMAKE_C_FLAGS`.





Set the baudrate of `uart6` to 921600, for the wifi module HC-25. The HC-25 module settings are on the IP address `192.168.4.1` as a web page. Here is the checklist to set up a new module:

1. The password is not set for new modules. So just login without a password to access the settings page.

2. Set a username and password for the robot's access point.
3. Set the *WiFi Mode* to **AP** for Access Point.
4. Change the port number from *8080* to *10000*.
5. Set the *Baud Rate* parameter to 921600 Bits/s.

Step Forward

The function `stepForward` identifies the Q function using RLS with the given pointer to the `model`. The algorithm updates the Q function at each step. As a result, the filter matrix `W_n` and the inverse auto-correlation matrix `P_n` are updated. Performs a one-step update in the parameter vector W by applying RLS to equation.

$$W_{j+1}^T(\phi(z_k) - \gamma\phi(z_{k+1})) = r(x_k, h_j(x_k))$$

$$W_{j+1}^T(\phi(z_k) - \phi(z_{k+1})) = \tfrac{1}{2}(x_k^T Q x_k + u_k^T R u_k)$$

```
void stepForward(LinearQuadraticRegulator *model)
```

The vector of filter coefficients $\mathbf{w}_n = \begin{bmatrix} w_n(0) & w_n(1) & \dots & w_n(p) \end{bmatrix}^T$ at time n minimizes the weighted least squares error. The weighted least squares error is equal to the squared norm of the error at time i times an exponential weighting factor, sumed over the observation interval. $\mathrm{E}(n) = \sum_{i=0}^{n} \lambda^{n-i} |e(i)|^2$. The exponential weighting (forgetting) factor λ is greater than zero and, less than or equal to one. $0 < \lambda \leq 1$. The error at time i is equal to the difference between the desired signal and the filter output. $e(i) = d(i) - y(i) = d(i) - \mathbf{w}_n^T x(i)$. In the definition of the error, $d(i)$ is the desired signal at time i, and $y(i)$ is the filter output at time i. The filter output is the result of the matrix-vector product of the filter coefficients and the new data vector. The latest set of filter coefficients $\mathbf{w}_n(k)$ is used for minimizing the weighted least squares error $\mathrm{E}(n)$. Also, it is assumed that the weights $\mathbf{w}_n$ are constant over the observation interval $[0, n]$ with end points zero and n.

For the error minimization objective, the partial derivative of the weighted least squares error with respect to the filter coefficients must be equal to zero. Setting the partial derivative equal to zero minimizes the weighted least squares error $\frac{\partial \mathrm{E}(n)}{\partial \mathbf{w}_n^*(k)} = 0$ for $k = 0, 1, ..., p$, where p denotes the filter order. Following the implications of the partial derivative equation, the filter coefficients are transformed by the exponentially weighted deterministic autocorrelation matrix $\mathbf{R}_x(n)\mathbf{w}_n = \mathbf{r}_{dx}(n)$ in order to produce the deterministic cross-correlation between the desired signal $d(n) = \begin{bmatrix} d(n) & d(n-1) & \dots & d(0) \end{bmatrix}^T$ and the new data vector $\mathbf{x}(i) = \begin{bmatrix} x(i) & x(i-1) & \dots & x(i-p) \end{bmatrix}^T$.

Therefore, the deterministic normal equations define the optimum filter coefficients. The exponentially weighted deterministic autocorrelation matrix $\mathbf{R}_x(n) \in \mathbb{R}^{(p+1)\times(p+1)}$ for the new data vector $\mathbf{x}(n)$ is defined as the sum of the outer product of the data vector with itself, in an

exponential way with the given exponential factor λ. In contrast, the deterministic cross-correlation $\mathbf{r}_{dx}(n)$ is the outer product between the desired signal $d(n)$ and the data vector $\mathbf{x}(n)$.

$$\begin{cases} \mathbf{r}_{dx}(n) = \sum_{i=0}^{n} \lambda^{n-i} d(i) \mathbf{x}^*(i) \\ \mathbf{R}_x(n) = \sum_{i=0}^{n} \lambda^{n-i} \mathbf{x}^*(i) \mathbf{x}^T(i) \end{cases}$$

Multiplying the filter coefficients with the deterministic cross-correlation on the left and then subtracting the result from the weighted norm of the desired signal $||d(n)||_{\lambda}^2$ yields the minimum error $\{\mathrm{E}(n)\}_{min} = ||d(n)||_{\lambda}^2 - \mathbf{r}_{dx}^H(n) \mathbf{w}_n$.

Both the deterministic autocorrelation matrix $\mathbf{R}_x(n)$ and the deterministic cross-correlation $\mathbf{r}_{dx}(n)$ depend on the time variable n. So instead of directly solving the deterministic normal equations $\mathbf{R}_x(n) \mathbf{w}_n = \mathbf{r}_{dx}(n)$, it is easier to derive a recursive solution for the filter coefficients $\mathbf{w}_n$. A correction $\Delta \mathbf{w}_{n-1}$ that is applied to the solution at time $n-1$ results in the filter coefficients $\mathbf{w}_n = \mathbf{w}_{n-1} + \Delta \mathbf{w}_{n-1}$ at time n.

For a recursive equation, first derive the deterministic cross-correlation $\mathbf{r}_{dx}(n)$ at time n in terms of the cross-correlation $\mathbf{r}_{dx}(n-1)$ at time $n-1$. To solve for the filter coefficients, multiply both sides of the deterministic normal equations on the left by the inverse of the deterministic autocorrelation matrix: $\mathbf{w}_n = \mathbf{R}_x^{-1}(n) \mathbf{r}_{dx}(n)$. Second, derive the inverse of the deterministic autocorrelation matrix $\mathbf{R}_x^{-1}(n)$ in terms of the inverse autocorrelation at the previous time $\mathbf{R}_x^{-1}(n-1)$ and the new data vector $\mathbf{x}(n)$. On the one hand, the cross-correlation $\mathbf{r}_{dx}(n) = \sum_{i=0}^{n} \lambda^{n-i} d(i) \mathbf{x}^*(i)$ may be updated recursively as the sum of the previous cross-correlation times the weighting factor, and the desired signal times new data. On the other hand, the autocorrelation matrix $\mathbf{R}_x(n)$ may be updated recursively from the autocorrlation of the previous time $\mathbf{R}_x(n-1)$ and the new data vector $\mathbf{x}(n)$ in this way: multiply the previous autocorrelation by the weighting factor and then add the result to the outer product of the new data vector with itself.

$$\begin{cases} \mathbf{r}_{dx}(n) = \lambda \mathbf{r}_{dx}(n-1) + d(n) \mathbf{x}^*(n) \\ \mathbf{R}_x(n) = \lambda \mathbf{R}_x(n-1) + \mathbf{x}^*(n) \mathbf{x}^T(n) \end{cases}$$

Since we are interested in the inverse of the autocorrelation matrix $\mathbf{R}_x(n)$ we use the Woodbury's Identity. The Woodbury matrix identity $(\mathbf{A} + \mathbf{u}\mathbf{v}^H)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^H\mathbf{A}^{-1}}{1+\mathbf{v}^H\mathbf{A}^{-1}\mathbf{u}}$ says that the inverse of a rank-k correction of some matrix can be computed by doing a rank-k correction to the inverse of the original matrix. Therefore, the multiplication of the weighting factor λ and the autocorrelation matrix $\mathbf{R}_x(n-1)$ at time $n-1$ should be the original matrix $\mathbf{A}$, whereas the pair of vectors $\mathbf{u}$ and $\mathbf{v}$ in the identity are both equal to the new data vector $\mathbf{x}^*(n)$.

$$\begin{cases} \mathbf{A} = \lambda \mathbf{R}_x(n-1) \\ \mathbf{u} = \mathbf{v} = \mathbf{x}^*(n) \end{cases}$$

$$\mathbf{R}_x^{-1}(n) = \lambda^{-1}\mathbf{R}_x^{-1}(n-1) - \frac{\lambda^{-2}\mathbf{R}_x^{-1}(n-1)\mathbf{x}^*(n)\mathbf{x}^T(n)\mathbf{R}_x^{-1}(n-1)}{1+\lambda^{-1}\mathbf{x}^T(n)\mathbf{R}_x^{-1}(n-1)\mathbf{x}^*(n)}$$

Writing the inverse of the deterministic autocorrelation matrix using Woodbury's identity, $\mathbf{P}(n) = \lambda^{-1}[\mathbf{P}(n-1) - \mathbf{g}(n)\mathbf{x}^T(n)\mathbf{P}(n-1)]$, we find that the inverse of the autocorrelation matrix $\mathbf{P}(n) = \mathbf{R}_x^{-1}(n)$ at time n involves two terms, one of which is the weighted version of the matrix at time $n-1$. The other term in the calculation of the inverse matrix is the multiplication of the gain vector and the data vector and the inverse matrix at time $n-1$. The gain vector is the solution to the equation $\mathbf{R}_x(n)\mathbf{g}(n) = \mathbf{x}^*(n)$. As such, the gain vector $\mathbf{g}(n) = \frac{\lambda^{-1}\mathbf{P}(n-1)\mathbf{x}^*(n)}{1+\lambda^{-1}\mathbf{x}^T(n)\mathbf{P}(n-1)\mathbf{x}^*(n)}$ is transformed by the deterministic autocorrelation matrix to yield the new data vector. Alternatively one can say that the new data vector is transformed by the inverse of the deterministic autocorrelation matrix in order to produce the gain vector $\mathbf{g}(n) = \mathbf{P}(n)\mathbf{x}^*(n)$. This is the same as the deterministic normal equations, but the cross-correlation vector $\mathbf{r}_{dx}(n)$ is replaced with the data vector $\mathbf{x}^*(n)$:

$$\begin{cases} \mathbf{R}_x(n)\mathbf{w}_n = \mathbf{r}_{dx}(n) \\ \mathbf{R}_x(n)\mathbf{g}(n) = \mathbf{x}^*(n) \end{cases}$$

```
model->x_n_dot_z_n = 0.0;
float buffer = 0.0;
for (int i = 0; i < (model->n + model->m); i++)
{
  buffer = getIndexVec12(model->dataset, i) * getIndexVec12(model->z_n, i);
  if (isnanf(buffer) == 0)
  {
    model->x_n_dot_z_n += buffer;
  }
}
if (fabs(model->lambda + model->x_n_dot_z_n) > 0)
{
  for (int i = 0; i < (model->n + model->m); i++)
  {
    setIndexVec12(&(model->g_n), i, (1.0 / (model->lambda + model->x_n_dot_z_n))
  }
}
else
{
  for (int i = 0; i < (model->n + model->m); i++)
  {
    setIndexVec12(&(model->g_n), i, (1.0 / model->lambda) * getIndexVec12(model-
  }
}
```

The derivation of the time-update equation for the coefficient vector $\mathbf{w}_n$ completes the recursion. Start with the fact that the transformation of the cross-correlation by the inverse

autoccorrelation results in the filter coefficients. Then, replace the cross-correlation with the recursive solution.

$$\begin{cases} \mathbf{w}_n = \mathbf{P}(n)\mathbf{r}_{dx}(n) \\ \mathbf{r}_{dx}(n) = \lambda \mathbf{r}_{dx}(n-1) + d(n)\mathbf{x}^*(n) \end{cases}$$

Replacing the solution with a recursive term gets us half of the way, because in the time-update equation we still need to replace the inverse autocorrelation matrix with a recursive inverse autocorrelation: $\mathbf{w}_n = \lambda \mathbf{P}(n)\mathbf{r}_{dx}(n-1) + d(n)\mathbf{P}(n)\mathbf{x}^*(n)$. Recall that the transformation of the data vector by the inverse autocorrelation matrix gives us the gain vector. But we also established that the recursive relation of the inverse autocorrelation matrix includes the gain vector.

$$\begin{cases} \mathbf{P}(n)\mathbf{x}^*(n) = \mathbf{g}(n) \\ \mathbf{P}(n) = \lambda^{-1}[\mathbf{P}(n-1) - \mathbf{g}(n)\mathbf{x}^T(n)\mathbf{P}(n-1)] \end{cases}$$

Given these two facts, the time-update of the filter coefficients at time n is rewritten, $\mathbf{w}_n = [\mathbf{P}(n-1) - \mathbf{g}(n)\mathbf{x}^T(n)\mathbf{P}(n-1)]\mathbf{r}_{dx}(n-1) + d(n)\mathbf{g}(n)$, so that it becomes dependent on the inverse autocorrelation matrix at time $n-1$. So far, the filter coefficients vector is derived in terms of the following: the filter coefficients at time $n-1$, the gain vector at time n, the desired signal at time n, and the new data vector: $\mathbf{w}_n = \mathbf{w}_{n-1} + \mathbf{g}(n)[d(n) - \mathbf{w}_{n-1}^T\mathbf{x}(n)]$. To simplify the time-update equation, we use the filter coefficients at time $n-1$ in place of the transformation of the cross-correlation by the inverse autocorrelation: $\mathbf{P}(n-1)\mathbf{r}_{dx}(n-1) = \mathbf{w}_{n-1}$.

```
for (int i = 0; i < (model->n + model->m); i++)
{
  setIndexVec12(&(model->alpha_n), i, 0.0);
}
for (int i = 0; i < (model->n + model->m); i++)
{
  for (int j = 0; j < (model->n + model->m); j++)
  {
    setIndexVec12(&(model->alpha_n), i, getIndexVec12(model->alpha_n, i) + 0.0 -
  }
}
```

Here is yet another simplification for defining the correction to the filter coefficients as the *a priori error* transforming the gain vector: $\mathbf{w}_n = \mathbf{w}_{n-1} + \alpha(n)\mathbf{g}(n)$. The *a priori error* $\alpha(n) = d(n) - \mathbf{w}_{n-1}^T\mathbf{x}(n)$ is the difference relation between the desired signal $d(n)$ at time n and the estimate of the desired signal $\mathbf{w}_{n-1}^T\mathbf{x}(n)$ using the previous set of filter coefficients $\mathbf{w}_{n-1}$ at time $n-1$.

$$\begin{cases} \alpha(n) = d(n) - \mathbf{w}_{n-1}^T \mathbf{x}(n) \\ e(n) = d(n) - \mathbf{w}_n^T \mathbf{x}(n) \end{cases}$$

The *a priori error* $\alpha(n)$ is defined as the error that would occur if the filter coefficients were not updated, whereas the *a posteriori error* $e(n)$ is defined as the error that occurs after the weight vector $\mathbf{w}_n$ is updated.

```
for (int i = 0; i < (model->n + model->m); i++)
{
  setIndexVec12(&(model->z_n), i, 0.0);
}
for (int i = 0; i < (model->n + model->m); i++)
{
  for (int j = 0; j < (model->n + model->m); j++)
  {
    setIndexVec12(&(model->z_n), i, getIndexVec12(model->z_n, i) + getIndexMat12
  }
}
```

The definition of the filtered information vector $\mathbf{z}(n) = \mathbf{P}(n-1)\mathbf{x}^*(n)$ makes the equations for the gain vector and the inverse of the deterministic autocorrelation matrix simple.

$$\begin{cases} \mathbf{g}(n) = \frac{\lambda^{-1}\mathbf{P}(n-1)\mathbf{x}^*(n)}{1+\lambda^{-1}\mathbf{x}^T(n)\mathbf{P}(n-1)\mathbf{x}^*(n)} \\ \mathbf{P}(n) = \lambda^{-1}[\mathbf{P}(n-1) - \mathbf{g}(n)\mathbf{x}^T(n)\mathbf{P}(n-1)] \end{cases}$$

The filtered information vector $z(n)$ at time n is the transformation of the new data vector at time n by the inverse of the autocorrelation matrix at time $n-1$.

$$\begin{cases} \mathbf{g}(n) = \frac{1}{\lambda+\mathbf{x}^T(n)\mathbf{z}(n)}\mathbf{z}(n) \\ \mathbf{P}(n) = \frac{1}{\lambda}[\mathbf{P}(n-1) - \mathbf{g}(n)\mathbf{z}^H(n)] \end{cases}$$

```
// a backup of old filter coefficients before updating for calculating the magni
Mat12 W_1;
for (int i = 0; i < (model->n + model->m); i++)
{
  for (int j = 0; j < (model->n + model->m); j++)
  {
    setIndexMat12(&W_1, i, j, getIndexMat12(model->W_n, i, j));
    buffer = getIndexMat12(model->W_n, i, j) + getIndexVec12(model->alpha_n, i)
    if (isnanf(buffer) == 0)
    {
      setIndexMat12(&(model->W_n), i, j, buffer);
    }
  }
}
```

```
model->changes = calculateChanges(W_1, model->W_n);
```

In short, we derived five equations for minimizing the weighted least squares error $\mathrm{E}(n)$ in a recursive way: the filtered information vector, the *a priori error*, the gain vector, the filter coefficients, and the inverse of the autocorrelation matrix. These equations are parts of what is called the exponentially weighted Recursive Least Squares (RLS) algorithm.

$$\begin{cases} \mathbf{z}(n) = \mathbf{P}(n-1)\mathbf{x}^*(n) \\ \alpha(n) = d(n) - \mathbf{w}_{n-1}^T\mathbf{x}(n) \\ \mathbf{g}(n) = \frac{1}{\lambda+\mathbf{x}^T(n)\mathbf{z}(n)}\mathbf{z}(n) \\ \mathbf{w}_n = \mathbf{w}_{n-1} + \alpha(n)\mathbf{g}(n) \\ \mathbf{P}(n) = \frac{1}{\lambda}[\mathbf{P}(n-1) - \mathbf{g}(n)\mathbf{z}^H(n)] \end{cases}$$

```
int scaleFlag = 0;
for (int i = 0; i < (model->n + model->m); i++)
{
  for (int j = 0; j < (model->n + model->m); j++)
  {
    buffer = (1.0 / model->lambda) * (getIndexMat12(model->P_n, i, j) - getIndex
    if (isnanf(buffer) == 0)
    {
      if (fabs(buffer) > model->clippingValue)
      {
        scaleFlag = 1;
      }
      setIndexMat12(&(model->P_n), i, j, buffer);
    }
  }
}
if (scaleFlag == 1)
{
  for (int i = 0; i < (model->n + model->m); i++)
  {
    for (int j = 0; j < (model->n + model->m); j++)
    {
      setIndexMat12(&(model->P_n), i, j, model->clippingFactor * getIndexMat12(m
    }
  }
}
```

Whenever the weighting factor is equal to one, $\lambda = 1$, the algorithm is called the growing window RLS algorithm, because it has infinite memory of the system's trajectory. Meaning the algorithm does not forget outliers, even though it can make the effects of older data less significant over time with a forgetting facter λ less than one. But a sliding window variant of the

algorithm can forget outlier at the cost of doubling the computation.

The recursive updating of the filter coefficients $\mathbf{w}_n$ and the inverse autocorrelation matrix $\mathbf{P}(n)$ requires initial conditions for both terms. A suggestion would be to initialize the deterministic autocorrelation matrix with the identity matrix multiplied by a small positive constant δ.

$$\begin{cases} \mathbf{R}_x(0) = \delta\mathbf{I} \\ \mathbf{P}(0) = \delta^{-1}\mathbf{I} \\ \mathbf{w}_0 = \mathbf{0} \end{cases}$$

But setting an initial zero vector for the filter coefficients does not minimize the weighted least squares error $\mathrm{E}(0)$, and so $\mathbf{w}_0$ is not an optimal initial vector. However, with an exponential weighting factor less than one, $\lambda < 1$, the bias in the least squares solution goes to zero as n increases.

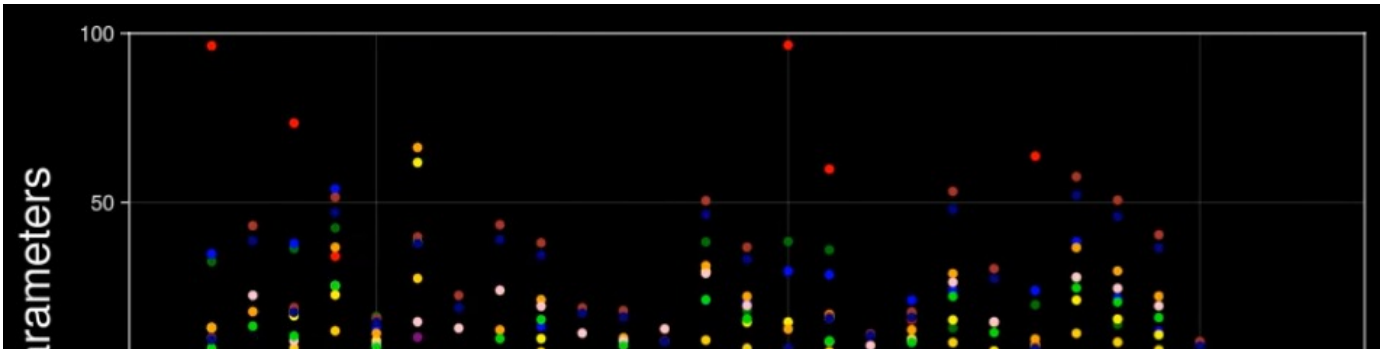
An adaptive control algorithm based on Q learning that converges to the solution to the discrete-time LQR problem. This is accomplished by solving athe algebraic Riccati equation in real time without knowing the system dynamics by using data measured along the system trajectories.

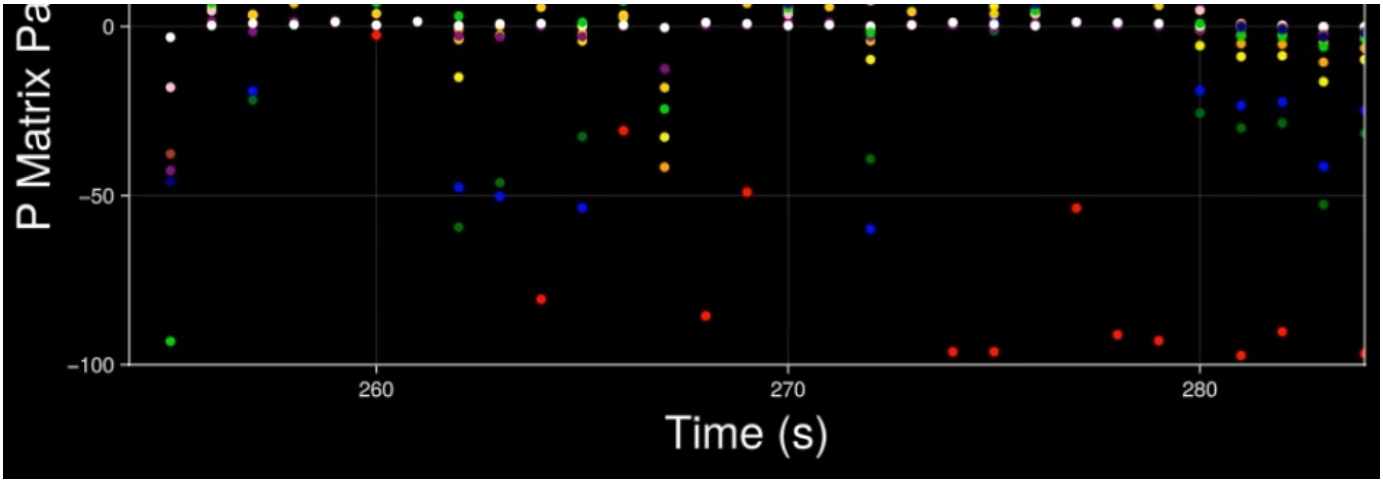
Q learning is implemented by repeatedly performing the iterations $W_{j+1}^T(\phi(z_k) - \gamma\phi(z_{k+1})) = r(x_k, h_j(x_k))$ and $h_{j+1}(x_k) = arg\ \underset{u}{min}(W_{j+1}^T\phi(x_k, u))$, for all $x \in X$. In it is seen that the LQR Q function is quadratic in the states and inputs so that $Q(x_k, u_k) = Q(z_k) \equiv (\frac{1}{2})z_k^T S z_k$ where $z_k = \begin{bmatrix} x_k^T \\ u_k^T \end{bmatrix}$.

The counter variable `k` is incremented every time the `stepForward` function is called for keeping track of the number of steps in an episode. This reminds us of the counter variable `j`, which counts the number of policy updates in the function `updateControlPolicy`. In the `stepForward` function, the variable `k` is incremented before returning to the `main` function. Repeat at the next time `k + 1` and continue until RLS converges and the new parameter vector W_{j+1} is found.

```
model->k = model->k + 1;
```

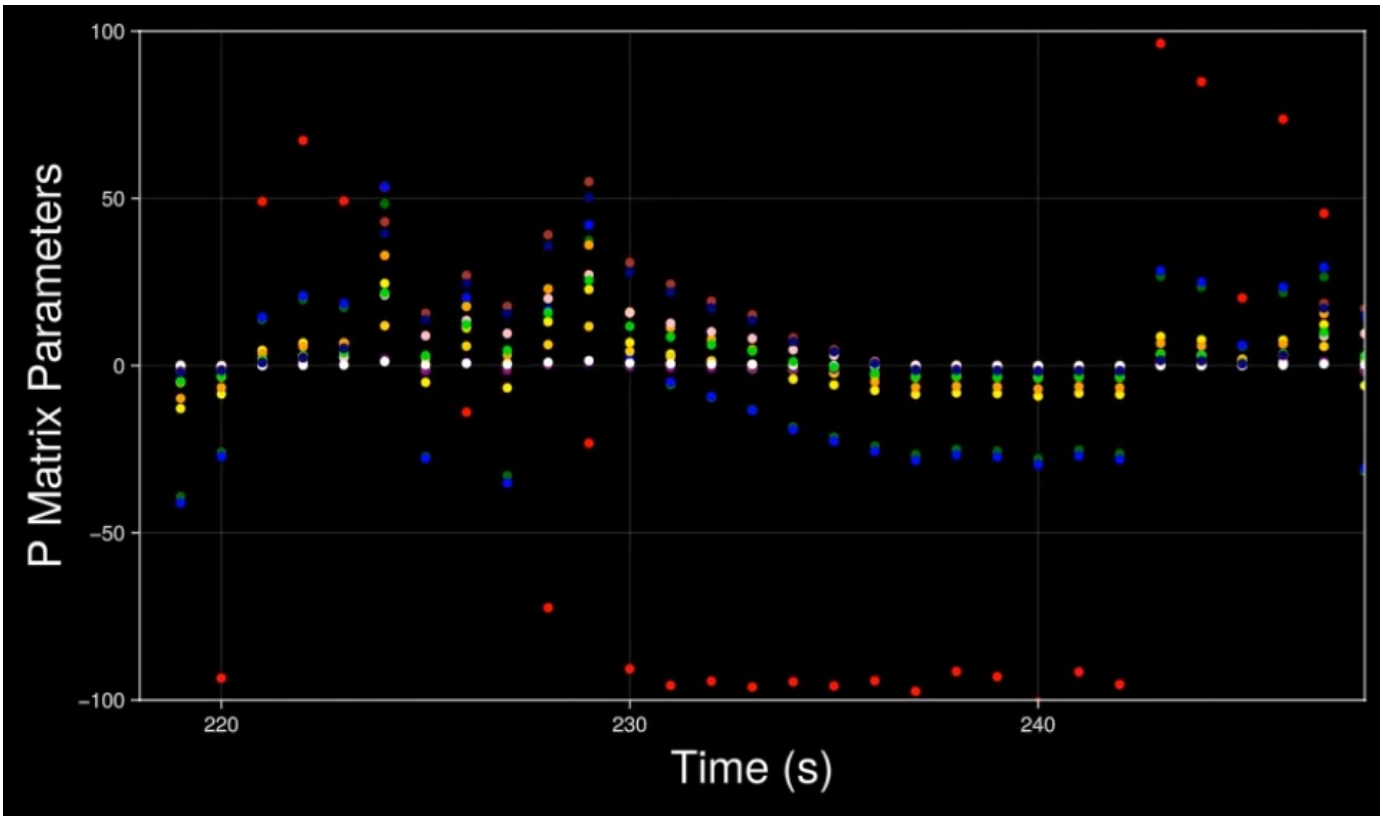
The Convergence of Selected Algebraic Riccati Equation Solution Parameters





Convergence of selected algebraic Riccati equation solution parameters. The adaptive controller based on value iteration converges to the ARE solution in real time without knowing the system matrix (including the inertia matrix, and the torque and the electromotive force constants of the motors.)

$$Q(x_k, u_k) = \frac{1}{2} \begin{bmatrix} x_k \\ u_k \end{bmatrix} \begin{bmatrix} A^T P A + Q & B^T P A \\ A^T P B & B^T P B + R \end{bmatrix} \begin{bmatrix} x_k \\ u_k \end{bmatrix}$$




```
void updateControlPolicy(LinearQuadraticRegulator *model)
{
  // unpack the vector Wj+1 into the kernel matrix
  // Q(xk, uk) ≡ 0.5 * transpose([xk; uk]) * S * [xk; uk] = 0.5 * transpose([xk;
  model->k = 1;
  model->j = model->j + 1;
```

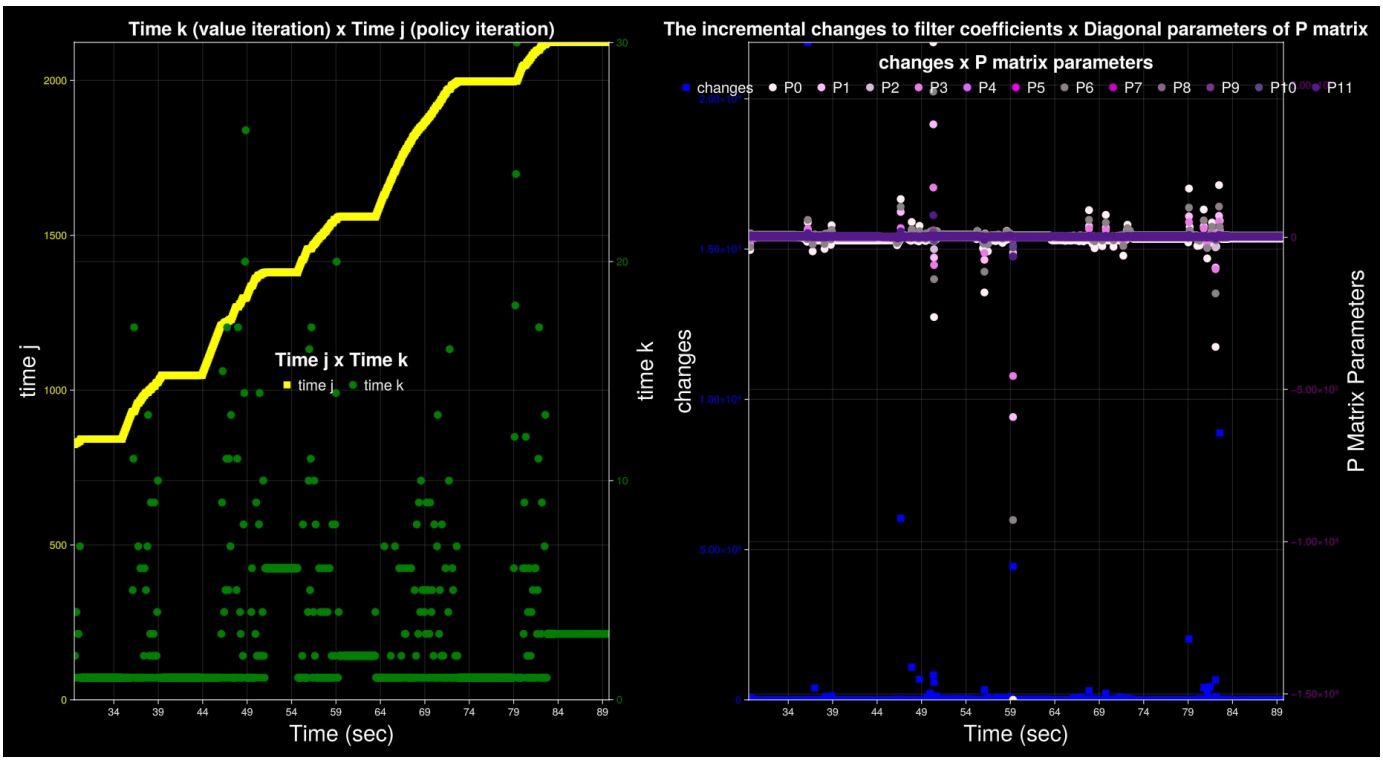
```
  for (int i = 0; i < model->m; i++)
  {
    for (int j = 0; j < model->n; j++)
    {
      setIndexMat210(&(model->Sux), i, j, getIndexMat12(model->W_n, model->n + i
    }
  }
  for (int i = 0; i < model->m; i++)
  {
    for (int j = 0; j < model->m; j++)
    {
      setIndexMat2(&(model->Suu), i, j, getIndexMat12(model->W_n, model->n + i,
    }
  }

  // Perform the control update using (S24), which is uk = -S^-1uu * Sux * xk
  // uk = -S^-1uu * Sux * xk
  float determinant = getIndexMat2(model->Suu, 1, 1) * getIndexMat2(model->Suu,
  // check the rank of S_uu to see if it's equal to 2 (invertible matrix)
  if (fabs(determinant) > 0.001) // greater than zero
  {
    setIndexMat2(&(model->SuuInverse), 0, 0, getIndexMat2(model->Suu, 1, 1) / de
    setIndexMat2(&(model->SuuInverse), 0, 1, -getIndexMat2(model->Suu, 0, 1) / d
    setIndexMat2(&(model->SuuInverse), 1, 0, -getIndexMat2(model->Suu, 1, 0) / d
    setIndexMat2(&(model->SuuInverse), 1, 1, getIndexMat2(model->Suu, 0, 0) / de
    // initialize the gain matrix
    for (int i = 0; i < model->m; i++)
    {
      for (int j = 0; j < model->n; j++)
      {
        setIndexMat210(&(model->K_j), i, j, 0.0);
      }
    }
    for (int i = 0; i < model->m; i++)
    {
      for (int j = 0; j < model->n; j++)
      {
        for (int k = 0; k < model->m; k++)
        {
          setIndexMat210(&(model->K_j), i, j, getIndexMat210(model->K_j, i, j) +
        }
      }
    }
  }
  return;
}
```

$$Q(x_k, u_k) = \frac{1}{2}\begin{bmatrix} x_k \\ u_k \end{bmatrix}^T S \begin{bmatrix} x_k \\ u_k \end{bmatrix} = \frac{1}{2}\begin{bmatrix} x_k \\ u_k \end{bmatrix}^T \begin{bmatrix} S_{xx} & S_{xu} \\ S_{ux} & S_{uu} \end{bmatrix} \begin{bmatrix} x_k \\ u_k \end{bmatrix}$$

The Relation Between Time k and the Incremental Changes to Filter Coefficients

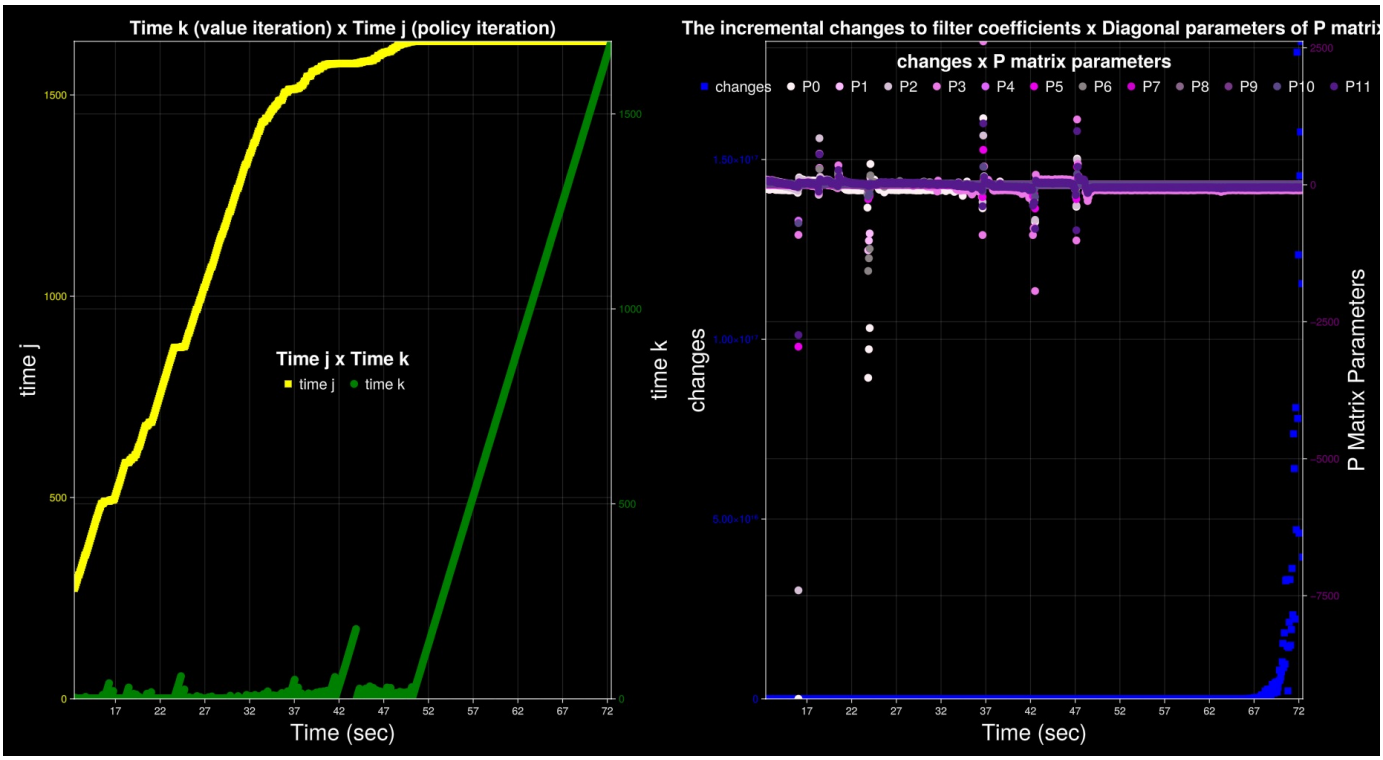
Time *j* is incremented if only time *k* is reset to the value of 1. But, time k is reset after the RLS algorithm converges. The incremental *changes* that are made to the filter coefficients $\Delta W_n = \Sigma(W_n - W_{n-1})$ is calculated by summing up the absolute value of the incremental updates to the filter coefficients.

```
float calculateChanges(Mat12 W_1, Mat12 W_2)
{
  float changes = 0.0;
  for (int i = 0; i < 12; i++)
  {
    for (int j = 0; j < 12; j++)
    {
      changes += fabs(getIndexMat12(W_2, i, j) - getIndexMat12(W_1, i, j));
    }
  }
  return changes;
}
```

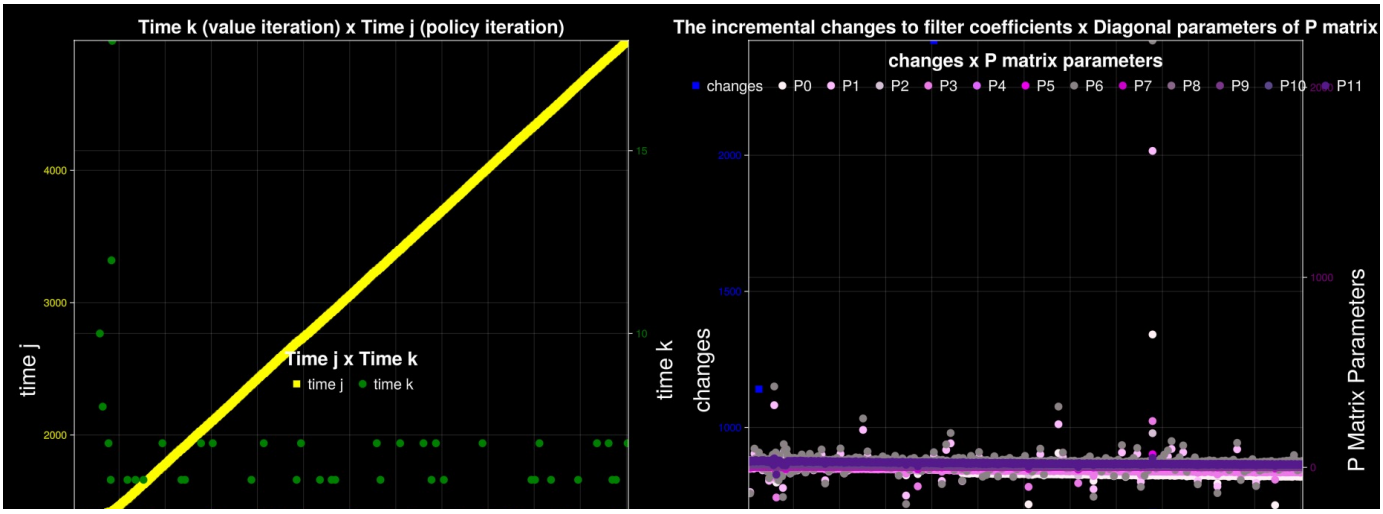


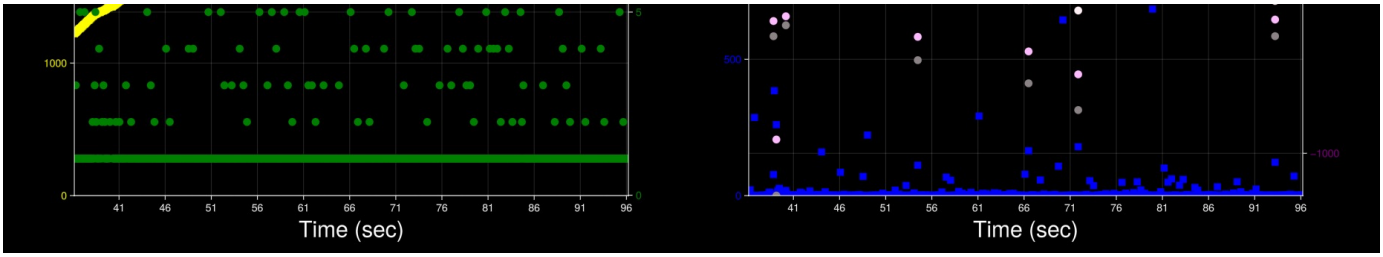
In the plot below, we see that time *j* stops incrementing before 52 seconds. The direct cause is that time *k* keeps increasing strictly at the same time stamp. Time *k* is reset back to 1 whenever

the incremental *changes* are less than 2.0, $\Delta W_n < 2.0$. The large *changes* that we see in the plot beginning from 62 seconds should be a consequence of not updating the policy for an extended period of time (for about 20 seconds).

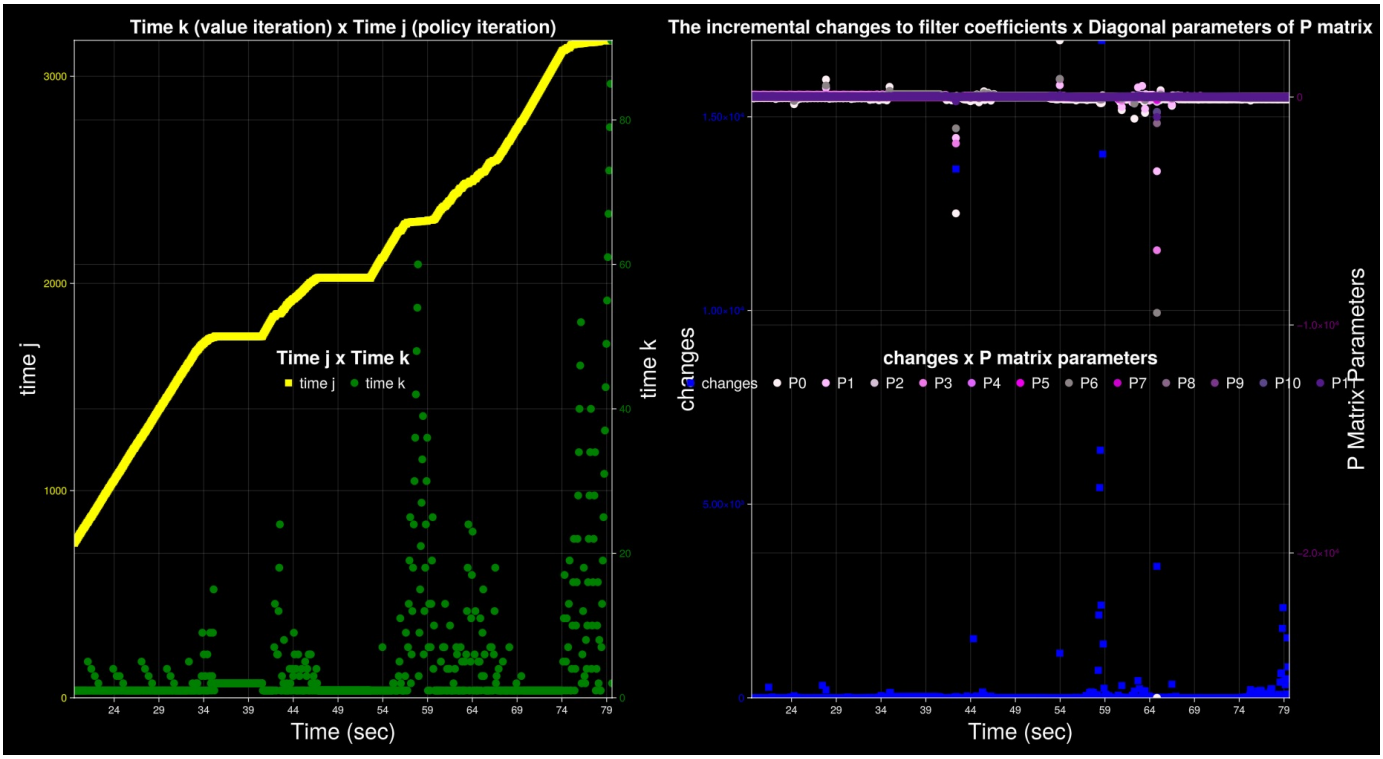


In the following figure, time *k* has a large value increase at about 40 seconds. Immediately after that, we see that time *j* has an steady increase over a 55-second interval. However, from 41 seconds until 96 seconds, time *k* deviates from the lower bound 1 for 7 steps at maximum, which are shown as 7 horizontal lines in green. The behavior of the graph of time *k* is reasonable in this case because the value of *k* changes in a regular way. Except for three data points before 41 seconds, the deviation of time *k* is standard as we add probing noise to the input for persistent excitation. The *changes* (as in ΔW_n) are scattered predominantly around the zero point. For most anomolies in the filter *changes* we have a corresponding anomoly in P matrix parameters. The anomolies of *changes* and *P matrix parameters* occur at the same time, because at those times the model is surprised (a great error value) and so adjusts the P matrix parameters to minimize the error.





In the left-side graph of the figure below, time *k* has sudden increases in steps at approximately: 34, 44, 54, 59, 64, 75 and 79 seconds. The same time stamps also show anomolies in the right-hand side graph of *changes*. But the *changes* are made zero, and that's why the greater *changes* snap back to approximately zero quickly. As a fact to be expected, the P matrix parameters have anomolous data points at approximately 35, 42, 54 and 65 seconds, which are coincident with great *changes* in the incremental changes in the filter coefficients. So the figure below shows simultaneous anomolies in time *k*, incremental *changes* in filter coefficients, and *P matrix parameters*. It is normal since a drop in time *k* means small *changes* and that means a policy update to then decrease the least squares error.



The criteria to determine when the RLS algorithm converges seems to work when the threshold of the *changes* ($\Delta W_n < 2.0$) is set equal to 2.0. A threshold equal to 1.0 results in suboptimal policies and performance, whereas a threshold of 3.0 produces useless policies.

The Controllability of the Z-Euler Angle

Nonholonomic Motion Planning

Steering Using Sinusoids

Steering Second-Order Canonical Systems

Attitude Control of A Space Platform / Manipulator System Using Internal Motion

In this section we separately model the motion of the robot in two different directions, which have different dynamics. In principle, an inverted pendulum is defined as a similar mechincal structure that is not stable in an intrinsic way (or has an unstable balance) and becomes balanced using a control system. Inverted pendulums have been interesting to control and systems labs for years, since their unstable and non-linear nature provides the opportunity to investigate the effectiveness of control algorithms. Similar structures are found in nature as well, for example the walking of humans. Because the cross section of the bottom of the human foot is not that big, the human body is not balanced in a very stable way, and what keeps the balance while walking, running, or even standing, is a series of control commands sent from the brain to muscles.

To model and perform dynamical calculations it is necessary to know the center of mass of the robot. For doing complementary calculations, it is also required to know the inertial momentum of the set of the wheel and the motor, and the rotational inertia of the robot's chassis about the horizontal axis passing through the center of mass, and the parameters of the motor. Finding the center of mass of the built robot is easy. Since the structure of the robot has bilateral symmetry, the center of mass of the robot is located on the axis of symmetry. Now, for determining the exact location of the center of mass, it is sufficient to put the robot on a triangular shape (like a seesaw) and then try to balance the robot. Having found the position of the center of mass, for subsequent calculations one can assume that the entire mass of the robot is located at that position, which is a certain distance away from the axis of the main wheel and the ground surface.

The center of mass of a multi-point mass, which is located at a determined position in a reference coordinate frame, can be calculated as $r_{CM} = \frac{\Sigma m_k r_k}{\Sigma m_k}$, where m_k denotes mass number k, and r_k denotes the position vector of that mass in terms of the reference frame. The rotational moment of inertia of the wheel is estimated approximately by measuring the mass and the radius. The moment of inertial of a uniform disk about its axis is equal to $I = \frac{mr^2}{2}$. The moment of inertial of a uniform circular loop about its axis is equal to $I = mr^2$. Becasue the wheel that we use has a structure between the two shapes (a disk and a circle), the moment of inertia is approximately equal to a value between the two equations. Besides, one can build the model of the wheel using a mechanical design software such as Catia, or perform precise experiments to find the exact value.

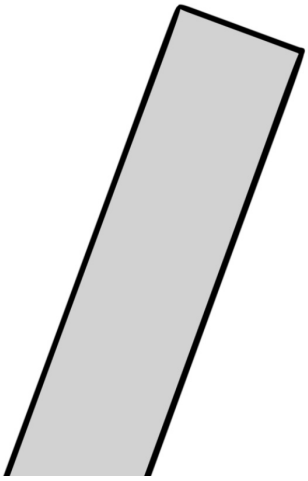
In modeling a balncing robot, the torques of the motors enter the equations describing the motion of the robot, rather than their velocities. In fact, describing the balance of the robot and providing

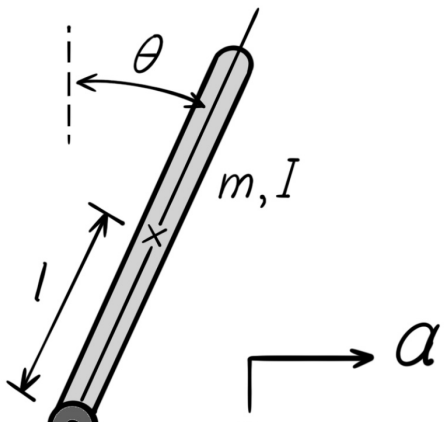
a state transition funcation or a state space model based on the velocity of the motor is specially complicated. That is why most robot makers use Direct Current (DC) motors in their balancing robots in order to make the mathematical modeling simple. In addition, direct current motors have higher power compared to stepper motors of similar dimensions and mass.

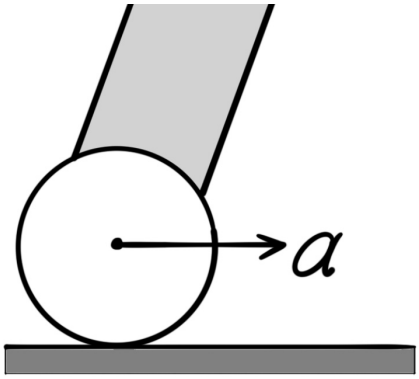
As the rotor is small in size and mass, its moment of inertia is much less than that of the wheel. However, because of the gearbox, the rotor rotates at a higher velocity than the wheel. Therefore, the moment of inertia of the rotor is multiplied by the gearbox transformation ratio to measure the effective moment of inetia of the rotor from outside of the gearbox. This value is not negligible usually. But, for calculating its effect on the motion of the robot one should know the direction of rotation of the rotor with respect to the wheel (the same direction or the opposite direction) since because of the gearbox the direction of rotation is not necessarily the same. Here, the moment of inertia of the rotor has been ignored in calculations. But you can consider its effect to make the calculations more accurate. The moment of inertia of the rotor and the internal parts of the gearbox are provided in the specifications sheet of the motor and gearbox, and are also measurable through experimentation. Here, we have ignored the moment of inertia of the set of the rotor and the gearbox parts. For calculating the rotational moment of inertial of the robot, we consider a simple model of the chassis and the motors, which represent the position of the mass of each part.

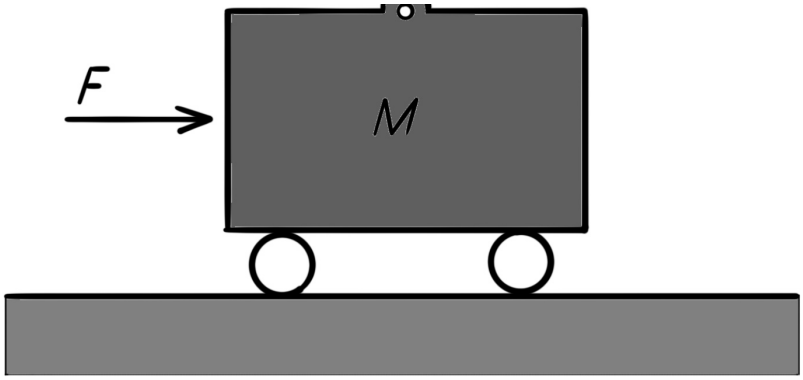
Some robot makers design the robot in mechanical simulation softwares such as Modelica, Dymola, Simulink, and Simscape Multibody. They consider a mathematical model according to the behavior of the robot based on the motor speed. The calculations for finding the mathematical model of balancing robots are gnerally based on either of the two methods: the Lagrange energy method or the analysis of Newtonian forces. In the inverted pendulum model that is connected to a moving system using a joint, the internal torque between the robot and its wheel does not have much effect on the motion of the robot. This case is made when the rotational moment of inertia of the wheel is negligible compared to that of the robot.

A Balancing Robot Compared to an Inverted Pendulum on a Cart









Deriving the Equations of the Robot to See How It Works

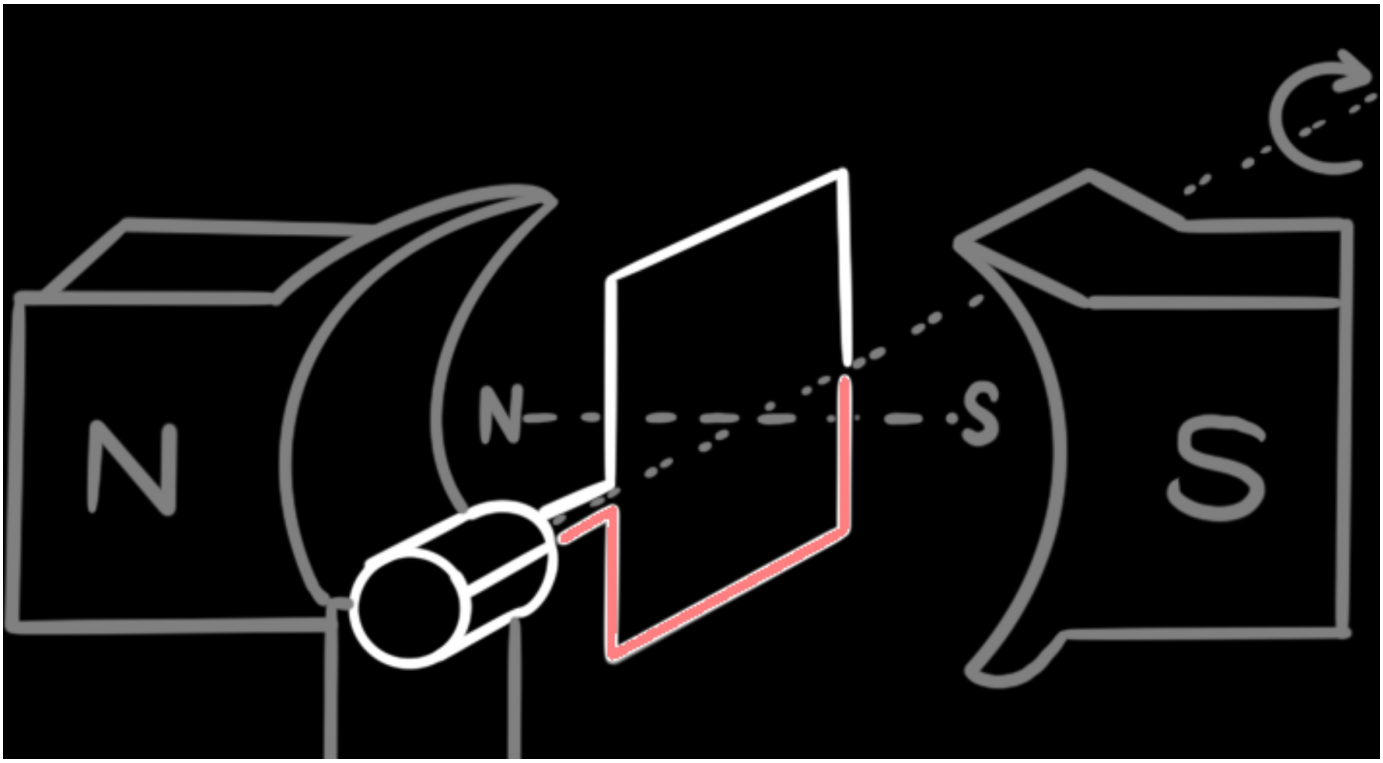
The relations between the supply voltage and the output torque of the DC motor is summarized as follows:

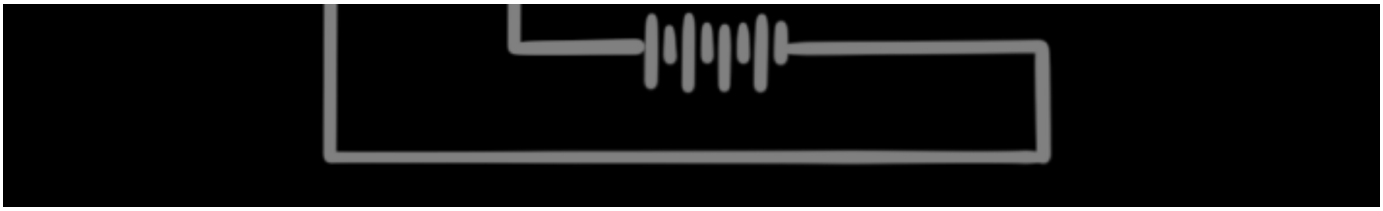
$$\tau = k_\tau i$$

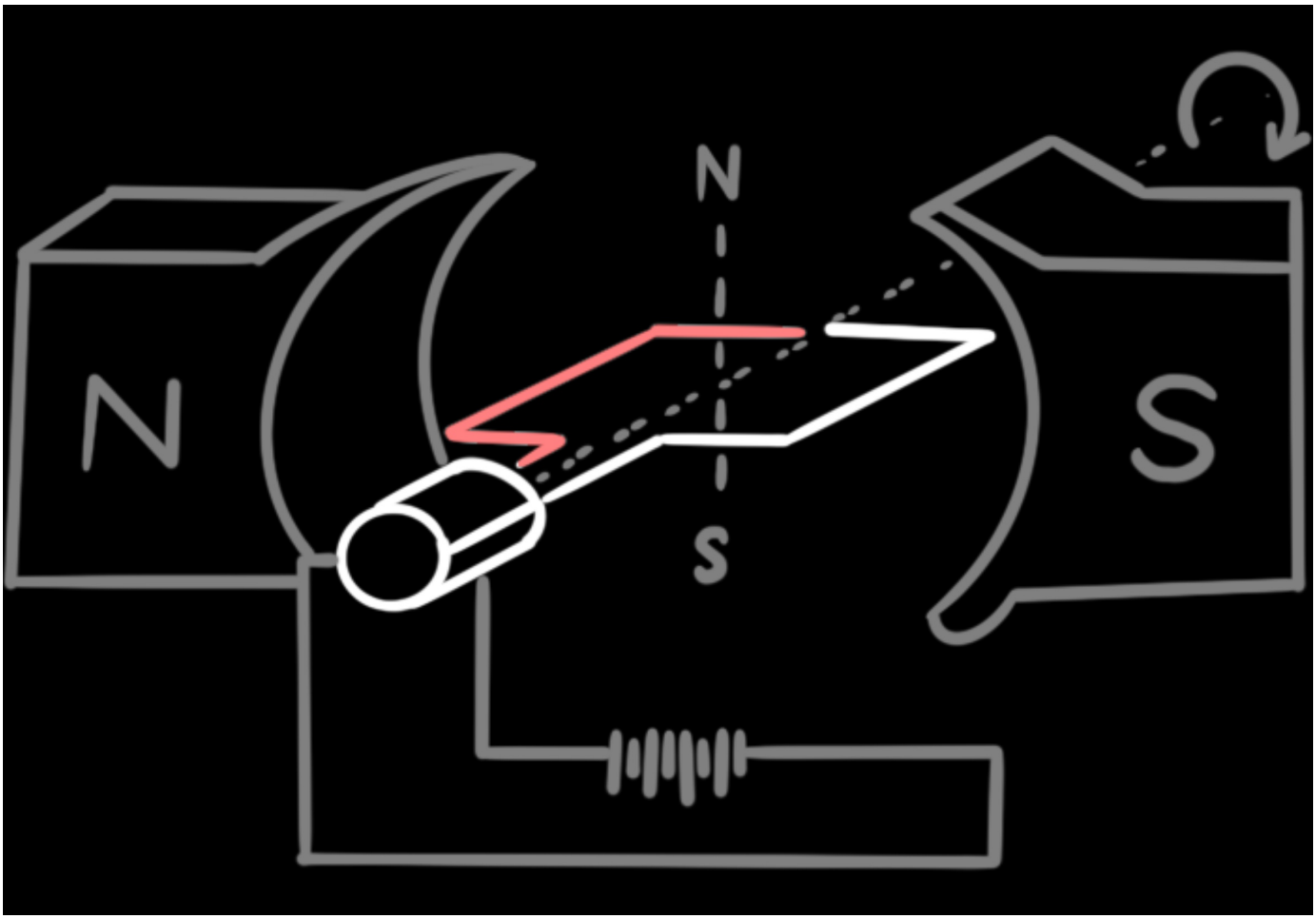
$$v = Ri + L\frac{di}{dt} + e \approx Ri + e$$

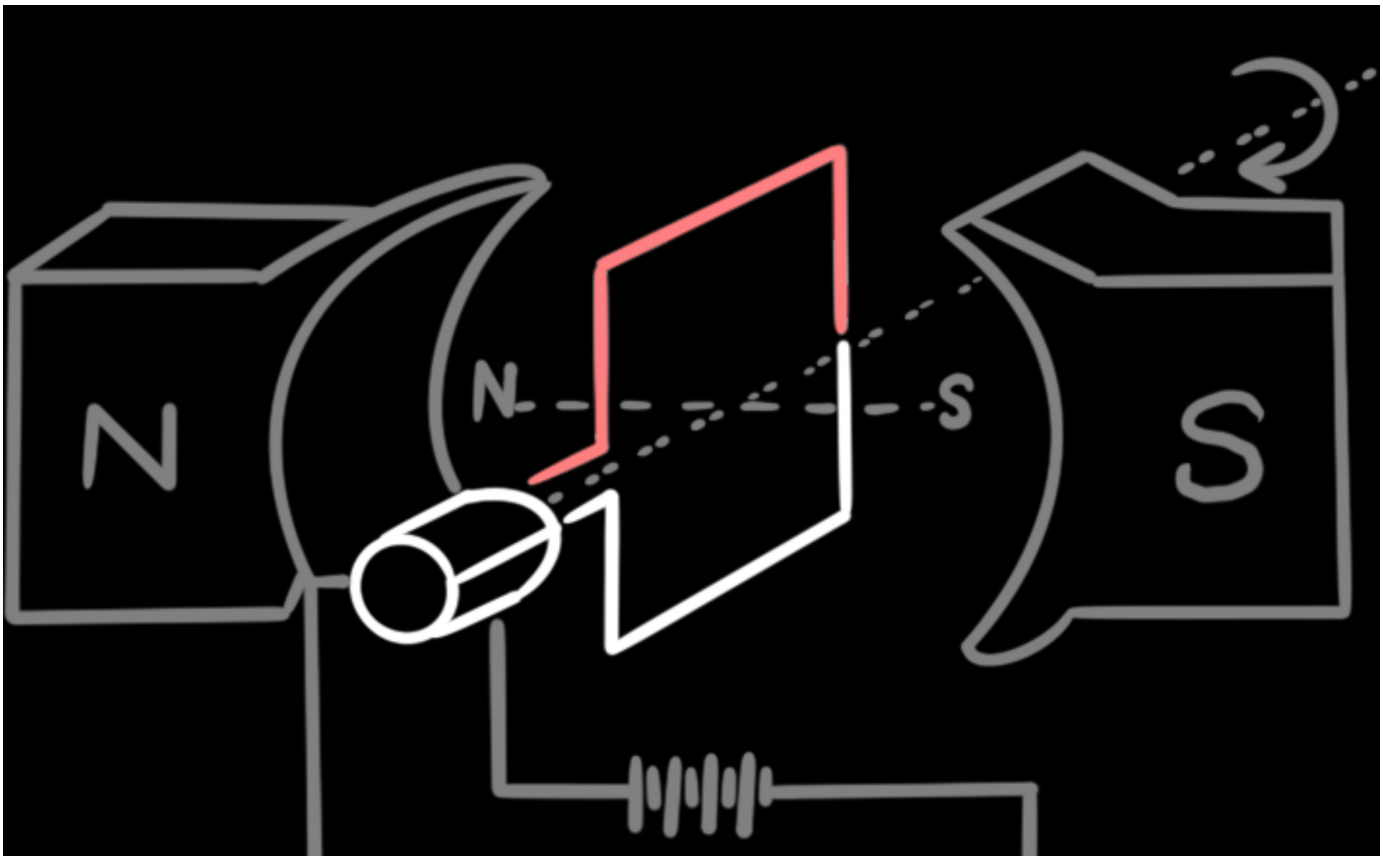
$$e = k_e \omega_{enc}$$

When the motor's current changes, due to the motor's inductance, it generates a voltage in opposition to the changing current via Faraday's law. While the motor's armature is spinning in a magnetic field with constant current, the Back-ElectroMotive Force (Back-EMF) is non-zero, because the motor acts as a generator while moving. The back-EMF is a function of the magnitude of the motor's rotational velocity (speed). So, the back-EMF is zero when the rotation speed is zero.






N
N
S
S


N
N
S
S

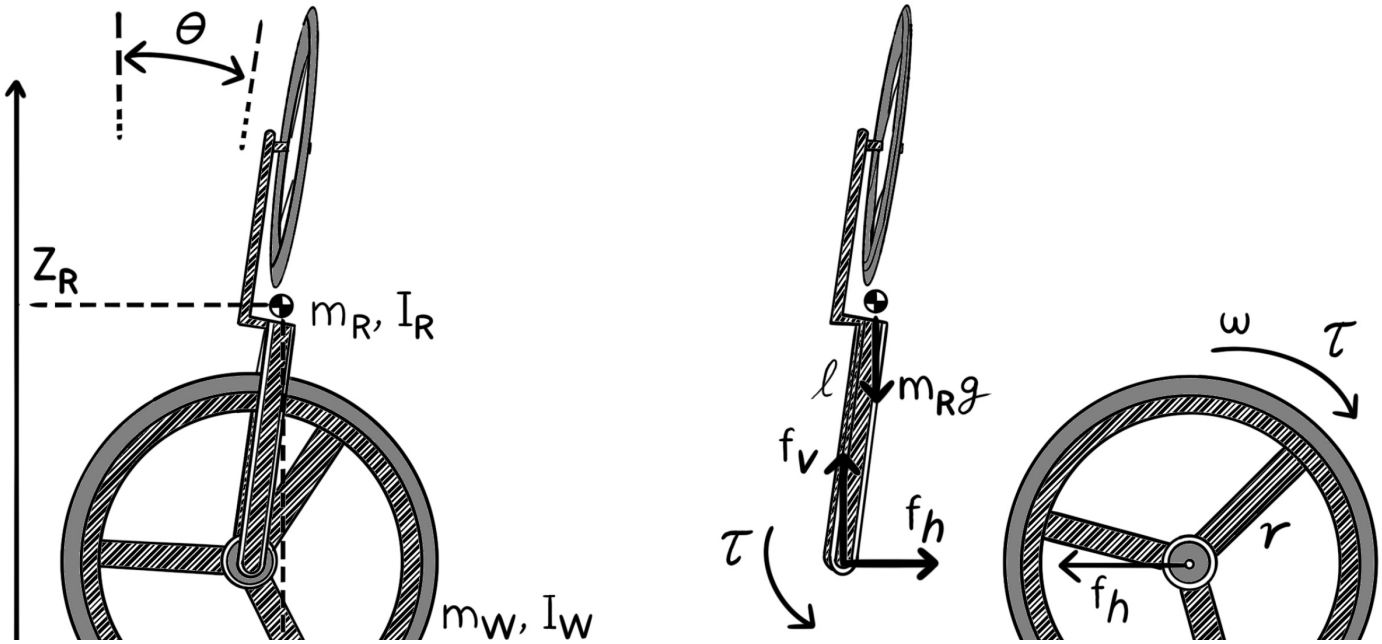
$v \approx R\frac{\tau}{k_\tau} + k_e\omega_{enc}$ (Equation 1)

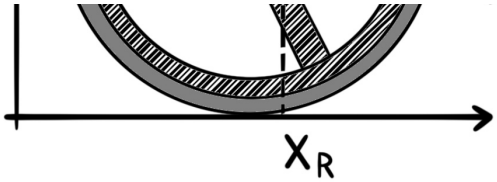
In equation 1 the term k_tau is called the torque constant, and k_e is called the back-EMF constant. The parameters R and L denote the resistance and the inductance of the motor's coils in the armature. Since the electrical behavior of the robot is much faster than its mechanical behavior, lower time constant, the inductance property that underlies the electrodynamics of the motor's coils is negligible and has been removed from the equation. The angular velocity of the output axis of the motor relative to the body's coordinate frame $\hat{B}$ is denoted by ω_{enc} and is equal to the difference between the angular velocity of the wheel and the chassis (or the motor's body) in the inertial coordinate frame $\hat{O}$.

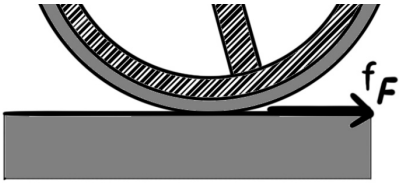
After removing the inductance property of the motor L, equation 1 becomes an approximation of the applied voltage in terms of torque and velocity. In an ideal motor, the torque constant k_τ and the back-electromotive constant k_e are equal. But, in reality this isn't the case and they have different values to be measured in an experimental way (might require opening the motor's casing). Based to physical features of the robot and the laws of Newtonian motion, we derive the equations that explain how the robot works. Also in this section, the method of Lagrangian mechanics is useful. In the configuration diagrams, the total mass of the robot is asumed to be located at its center of mass.

Modeling the Main Wheel's Behavior

First we derive the mathematical model of the robot's motion in the forward/backward direction, and its deviation from the upright equilibrium position. This motion is controlled by the robot's main wheel and is similar to the dynamics of a two-wheel balancing robot.







Using Newton's laws for the acceleration of the main wheel and the robot's body in the xz-plane, we have:

$m_W\ddot{x} = f_F - f_h \longrightarrow f_F = m_W\ddot{x} + f_h$ (Equation 2)

In the figure above, the mass of the robot and its main wheel are denoted by m_R and m_W, and their rotational moment of inertia are denoted by I_R and I_w, respectively.

The name m_R denotes the mass of all of the robot's parts, excluding the main wheel, and is located at the center of mass.

$\tau - f_F r_W = I_W\dot{\omega}_W \overset{\omega=\frac{\dot{x}}{r}}{\longrightarrow} \tau - f_F r_W = I_W\frac{\ddot{x}}{r}$ (Equation 3)

The parameter I_R denotes the rotational moment of inertia of all of the robot's parts, except for the main wheel, about the axis through the center of mass (parallel to the axis passing through the main wheel). For calculating m_R and I_R, the reaction wheel is assumed to be a static part of the robot's body. The names m_W and I_W represent the main wheel's mass and rotational moment of inertia, respectively.

The letters H, V and F stand for Horizontal, Vertical and Friction, respectively. They represent the horizontal, vertical and friction forces.

$m_R\ddot{x}_R = f_h \overset{x_R=x+lsin(\theta),\ sin(\theta)\approx\theta}{\longrightarrow} m_R(\ddot{x} + l\ddot{\theta}) = f_h$ (Equation 4)

$m_R\ddot{z}_R = f_v - m_R g \overset{z_R=l(1-cos(\theta)),\ cos(\theta)\approx 1}{\longrightarrow} f_v = m_R g$ (Equation 5)

$f_v l sin(\theta) - f_h l cos(\theta) - \tau = I_R\ddot{\theta} \overset{sin(\theta)\approx\theta,\ cos(\theta)\approx 1}{\longrightarrow} f_v l\theta - f_h l - \tau = I_R\ddot{\theta}$ (Equation 6)

By solving equations 1-6 simultaneously, we find the equations that describe the system. To extract the state equations, begin with equation 1:

$V \approx R\frac{\tau}{k_\tau} + k_e\omega_{enc}$

$-R\frac{\tau}{k_\tau} \approx k_e\omega_{enc} - V$

$\tau \approx \frac{Vk_\tau}{R} - \frac{k_e k_\tau \omega_{enc}}{R}$

Insert that into equation 3:

$\frac{Vk_\tau - k_e k_\tau \omega_{enc}}{R} - f_F r_W = \frac{I_W}{r}\ddot{x}$

$\overset{\omega_{enc} = \omega_W - \dot{\theta} = \frac{\dot{x}}{r} - \dot{\theta}}{\longrightarrow} \frac{Vk_\tau - k_e k_\tau (\frac{\dot{x}}{r} - \dot{\theta})}{R} - f_F r_W = \frac{I_W}{r}\ddot{x}$

$$Vrk_\tau - k_e k_\tau \dot{x} + rk_e k_\tau \dot{\theta} - f_F r_W rR = RI_W \ddot{x}$$

Then, substitute equation 2:

$$Vrk_\tau - k_e k_\tau \dot{x} + k_e k_\tau \dot{\theta} r - rRm_W \ddot{x} r_W - rRr_W f_h = RI_W \ddot{x}$$

Then, substitute equation 4:

$$Vrk_\tau - k_e k_\tau \dot{x} + k_e k_\tau \dot{\theta} r - m_W \ddot{x} r_W rR - I_W \ddot{x} R = m_R(\ddot{x} + l\ddot{\theta}) r_W rR$$

$$(-k_e k_\tau)\dot{x} + (k_e k_\tau r)\dot{\theta} - (rRm_W r_W + rRm_R r_W + I_W R)\ddot{x} - (m_R r_W L r R)\ddot{\theta} = -(rk_\tau)V$$

Then, divide both sides of the equation by $-r\ R$:

$\frac{k_\tau}{R}V = \frac{k_e k_\tau}{rR}\dot{x} - \frac{k_e k_\tau}{R}\dot{\theta} + (m_W r_W + m_R r_W + \frac{I_W}{r})\ddot{x} + (m_R r_W l)\ddot{\theta}$ (Equation 7)

That is one of the state equations. Now, begin with equation 6 again, for finding the other equation.

$$f_v l\theta - f_h l - \tau = I_R \ddot{\theta}$$

Next, insert equations 4 and 5 into equation 6:

$$\begin{cases} m_R(\ddot{x} + l\ddot{\theta}) = f_h \\ f_v = m_R g \end{cases} \longrightarrow m_R g l\theta - l(m_R(\ddot{x} + l\dot{\theta})) - \tau = I_R\ddot{\theta}$$

From equation 1 we have:

$\tau \approx \frac{Vk_\tau}{R} - \frac{k_e k_\tau \omega_{enc}}{R}$

Therefore,

$$m_R g l\theta - l(m_R(\ddot{x} + l\dot{\theta})) - I_R\ddot{\theta} = \tau$$

$$m_R g l\theta - l m_R(\ddot{x} + l\dot{\theta}) - I_R\ddot{\theta} = \frac{Vk_\tau}{R} - \frac{k_e k_\tau \omega_{enc}}{R}$$

$\overset{\omega_{enc} = \omega_W - \dot{\theta} = \frac{\dot{x}}{r} - \dot{\theta}}{\longrightarrow} m_R g l\theta - l m_R(\ddot{x} + l\dot{\theta}) - I_R\ddot{\theta} = \frac{Vk_\tau}{R} - \frac{k_e k_\tau (\frac{\dot{x}}{r} - \dot{\theta})}{R}$

Looks like we need to replace the value of V. But, from using equation 7 we can write the following equation.

$$V = \frac{k_e}{r_W}\dot{x} - k_e\dot{\theta} + \frac{Rm_W r_W + Rm_R r_W + \frac{R}{r_W}I_W}{k_\tau}\ddot{x} + \frac{Rm_R r_W l}{k_\tau}\ddot{\theta}$$

Therefore the following equation can take its final form.

$$m_R g l\theta - l m_R(\ddot{x} + l\dot{\theta}) - I_R\ddot{\theta} = \frac{V k_\tau}{R} - \frac{k_e k_\tau(\frac{\dot{x}}{r} - \dot{\theta})}{R}$$

$$m_R g l\theta - l m_R(\ddot{x} + l\dot{\theta}) - I_R\ddot{\theta} = \frac{k_\tau}{R}(\frac{k_e}{r}\dot{x} - k_e\dot{\theta} + \frac{R m_W r_W + R m_R r_W + \frac{R}{r} I_W}{k_\tau}\ddot{x} + \frac{R m_R r_W l}{k_\tau}\ddot{\theta}) - \frac{k_e k_\tau(\frac{\dot{x}}{r} - \dot{\theta})}{R}$$

After a rearrangement.

$$\ddot{x}(-l m_R - m_W r_W - m_R r_W - \frac{I_W}{r}) + \dot{x}(\frac{k_e k_\tau}{r_W R} - \frac{k_e k_\tau}{r_W R}) + \theta(m_R g l) + \ddot{\theta}(-I_R - m_R r_W l) + \dot{\theta}(-l^2 m_R + \frac{k_e k_\tau}{R} - \frac{k_e k_\tau}{R}) = 0$$

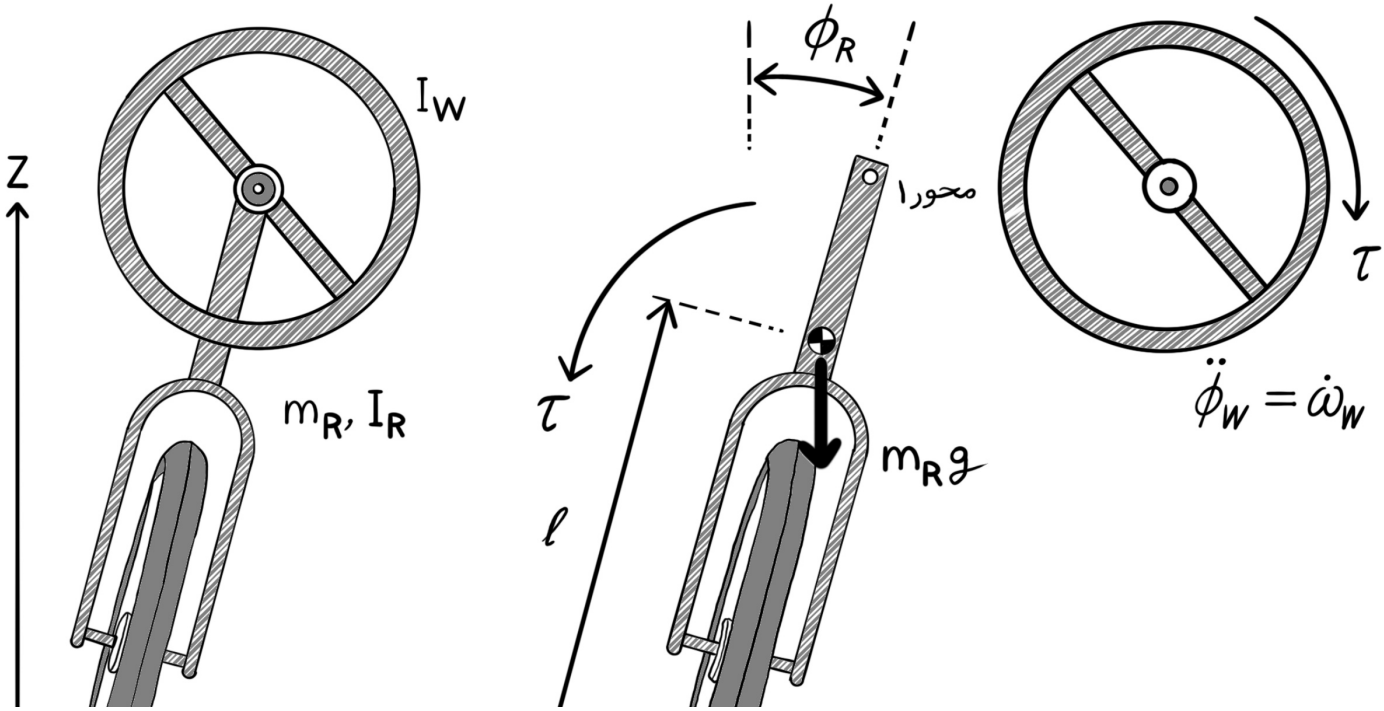
The final form of the main wheel's second equation:

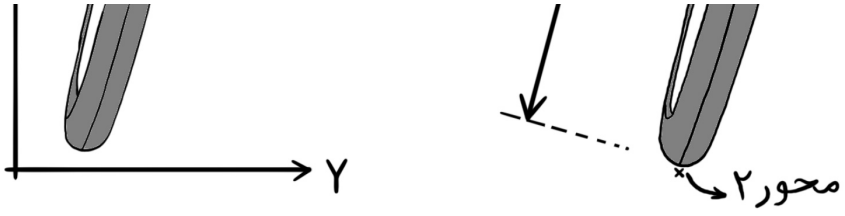
$$-(r_W m_W + r_W m_R + l m_R + \frac{I_W}{r_W})\ddot{x} + (m_R g l)\theta - (l^2 m_R)\dot{\theta} - (I_R + r_W m_R l)\ddot{\theta} = 0$$
(Equation 8)

Equations 7 and 8 determine the robot's behavior in the direction of the x-axis and its deviation in rotating about the y-axis (angle θ) in terms of the main motor's supply voltage. Next, we derive the mathematical model of the robot rotating around the x-axis (angle ϕ) and the effect of the reaction wheel.

Modeling the Reaction Wheel's Behavior

In this figure, the mass of all of the robot's parts (including the body, the main wheel and the reaction wheel) is denoted by m_R and is concentrated at the center of mass. The rotational inertia of all of the parts of the robot, excluding the reaction wheel, is denoted by I_R. And the rotational inertia of the reaction wheel is identified with I_W.





The parameter I_W is calculated about the reaction wheel's axis. And the parameter I_R is calculated about an axis parallel to the axis of the reaction wheel, but at the ground contact of the main wheel (axis 2). Note that the whole body of the robot oscillates about this axis.

In addition to gravity, we have a thin flexible band over the circumference of the main wheel at the ground contact of the wheel. The flexible belt effectively acts as a spring that resists side-to-side motions of the robot (angle ϕ) and tries to keep the robot upright. The force from this spring is given by the Hooke's law:

$f_{spring} = -k_s x = -k_s l \phi$ (Equation 9)

In equation 9, k_s denotes the spring constant and l denotes the length between the center of mass of the robot's body and the contact point on axis 2. The equations that govern the reaction wheel's driver are similar to those of the main motor, as equation 10 suggests.

$\tau \approx \frac{V - k_e \omega_{enc}}{R} k_\tau - f_{spring}$ (Equation 10)

$$\begin{cases} \omega_{enc} = \dot{\phi}_W - \dot{\phi}_R \\ f_{spring} = -k_s l \phi_R \end{cases}$$

Here is the equation governing the angular motion of the reaction wheel about its axis.

$\tau \approx I_W \ddot{\phi}_W$ (Equation 11)

The equation ruling the rotational motion of the entire robot about axis 2 at the ground contact point:

$mgl sin(\phi_R) - \tau \approx I_R \ddot{\phi}_R$ (Equation 12)

In the equations above the parameters m_R and I_R are written for the whole robot, including the reaction wheel since gravity applies to all parts. The parameters m_W and I_W are specifically related to the reaction wheel. Note that the rotational inertia I_R and I_W are calculated about two different axes. By solving equations 9 through 12 simultaneously, and assuming the linear approximation that $sin(\phi)$ is approximately equal to ϕ for small angles, the equations describing the system state are derived.

$\frac{v - k_e \omega_{enc}}{R} k_\tau - f_{spring} = I_W \ddot{\phi}_W$

$\frac{v - k_e (\dot{\phi}_W - \dot{\phi}_R)}{R} k_\tau - k_s l \phi_R = I_W \ddot{\phi}_W$

$\frac{-k_e\dot{\phi}_W+k_e\dot{\phi}_R}{R}k_\tau - k_s l\phi_R + \frac{V}{R}k_\tau = I_W\ddot{\phi}_W$

$\dot{\phi}_W(\frac{-k_e}{R}k_\tau) + \dot{\phi}_R(\frac{k_e}{R}k_\tau) + \phi_R(-k_s l) + \frac{V}{R}k_\tau = I_W\ddot{\phi}_W$

$\dot{\phi}_W(-k_e) + \dot{\phi}_R(k_e) + \phi_R(\frac{-k_s lR}{k_\tau}) + V = \frac{RI_W}{k_\tau}\ddot{\phi}_W$

$-k_e\dot{\phi}_W + k_e\dot{\phi}_R - \frac{k_s lR}{k_\tau}\phi_R - \frac{RI_W}{k_\tau}\ddot{\phi}_W = -V$

$k_e\dot{\phi}_W - k_e\dot{\phi}_R + \frac{k_s lR}{k_\tau}\phi_R + \frac{RI_W}{k_\tau}\ddot{\phi}_W = V$ (Equation 13)

Equation 13 without the force of the spring takes a different form, which makes sense because it should depend on the angle ϕ.

$k_e\dot{\phi}_R - k_e\dot{\phi}_W - \frac{RI_W}{k_\tau}\ddot{\phi}_W = -V$

Also, equation 13 defines the first state equation of the reaction wheel's controller. Next, we find the other half of the system state equations.

$mglsin(\phi_R) - \frac{V-k_e(\dot{\phi}_W-\dot{\phi}_R)}{R}k_\tau + k_s l\phi_R = I_R\ddot{\phi}_R$

$\overset{sin(\phi)\approx\phi}{\longrightarrow}$

$mgl\phi_R + \frac{k_e(\dot{\phi}_W-\dot{\phi}_R)}{R}k_\tau + k_s l\phi_R - \frac{Vk_\tau}{R} \approx I_R\ddot{\phi}_R$

$mgl\phi_R + \frac{k_e\dot{\phi_W}-k_e\dot{\phi}_R}{R}k_\tau + k_s l\phi_R - \frac{Vk_\tau}{R} \approx I_R\ddot{\phi}_R$

$\phi_R(mgl + k_s l) - \dot{\phi}_R(\frac{k_e}{R}k_\tau) + \dot{\phi}_W(\frac{k_e}{R}k_\tau) - \frac{Vk_\tau}{R} - I_R\ddot{\phi}_R \approx 0$

$\phi_R\frac{mglR+k_s lR}{k_\tau} - \dot{\phi}_R k_e + k_e\dot{\phi}_W - V - \frac{I_R R}{k_\tau}\ddot{\phi}_R \approx 0$

$\phi_R(\frac{mglR+k_s lR}{k_\tau}) - k_e\dot{\phi}_R + k_e\dot{\phi}_W - V - \frac{I_R R}{k_\tau}\ddot{\phi}_R \approx 0$

$$\begin{cases} \phi_R\frac{mglR+k_s lR}{k_\tau} - k_e\dot{\phi}_R + k_e\dot{\phi}_W - \frac{I_R R}{k_\tau}\ddot{\phi}_R - V \approx 0 \\ k_e\dot{\phi}_R - k_e\dot{\phi}_W - \frac{k_s lR}{k_\tau}\phi_R - \frac{RI_W}{k_\tau}\ddot{\phi}_W = -V \end{cases}$$

$\phi_R\frac{mglR+k_s lR}{k_\tau} - k_e\dot{\phi}_R + k_e\dot{\phi}_W - \frac{I_R R}{k_\tau}\ddot{\phi}_R + (k_e\dot{\phi}_R - k_e\dot{\phi}_W - \frac{k_s lR}{k_\tau}\phi_R - \frac{RI_W}{k_\tau}\ddot{\phi}_W) \approx 0$

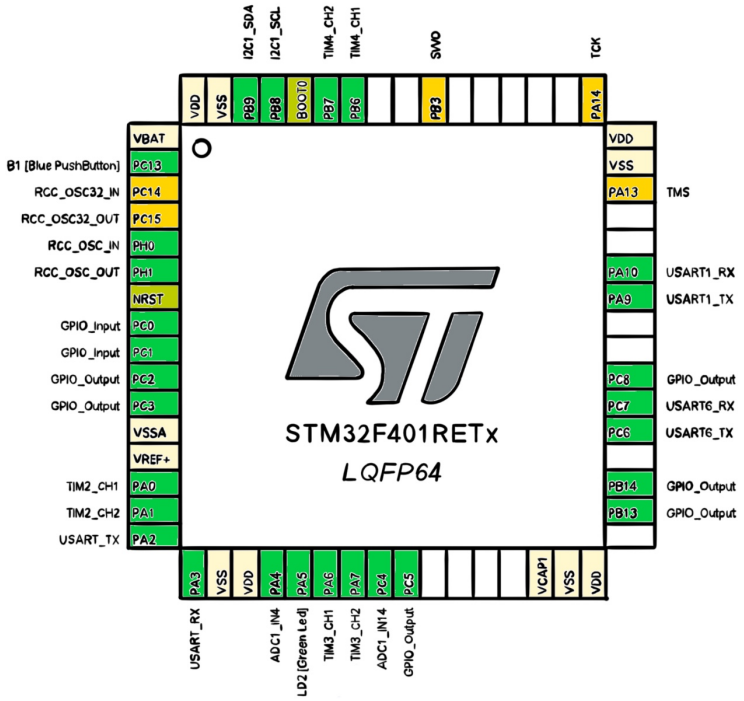
The result is equation 14 after simplification, completing the second half of the system state equations.

$I_R\ddot{\phi}_R + I_W\ddot{\phi}_W - (mgl)\phi_R \approx 0$ (Equation 14)

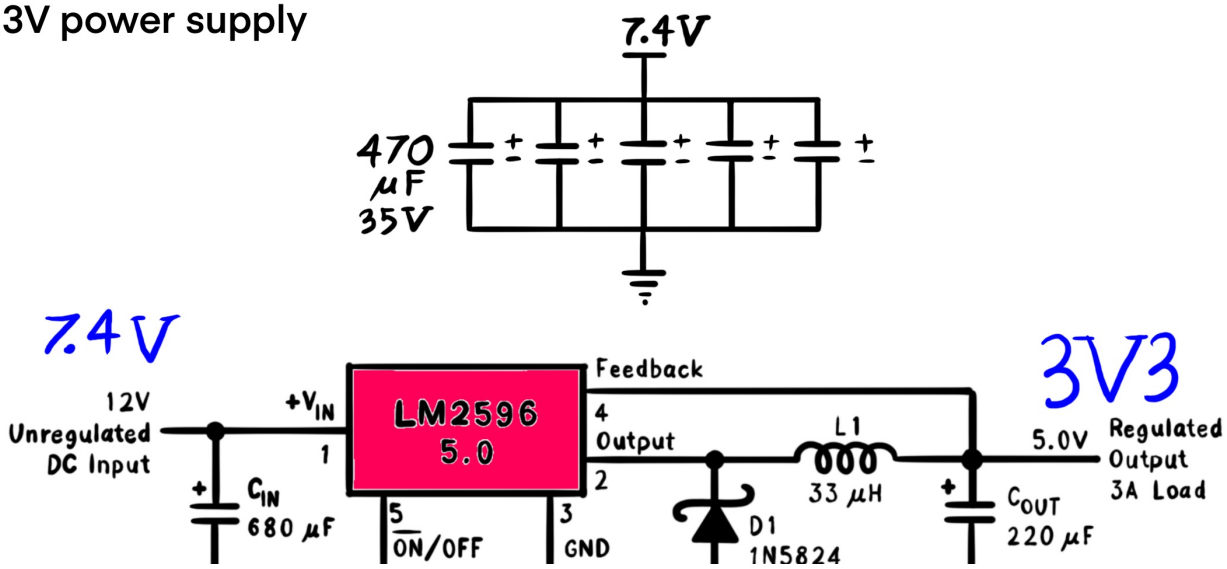
Here, we derived two different mathematical models for the motion of the robot about two mutually perpendicular axes. Using the equations that describe these motions (equations 7 and 8 or equations 13 and 14) we can extract the system's transform matrix of that motion, or find its state space. The system state at time-step n equals the transform of the system state at time-step

$n-1$. Finally, keeping balance in two different directions based on these equations requires two separate controllers. The controller that controls angle θ and motion along the x-axis commands The main motor, whereas the controller that controls angle ϕ commands the motor of the reaction wheel.

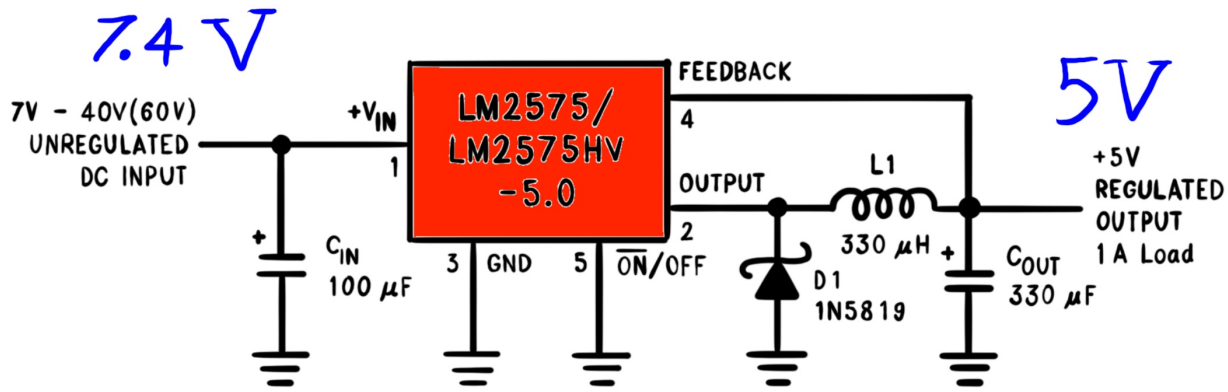
The Electronic Circuits and Power Supply



3.3V power supply

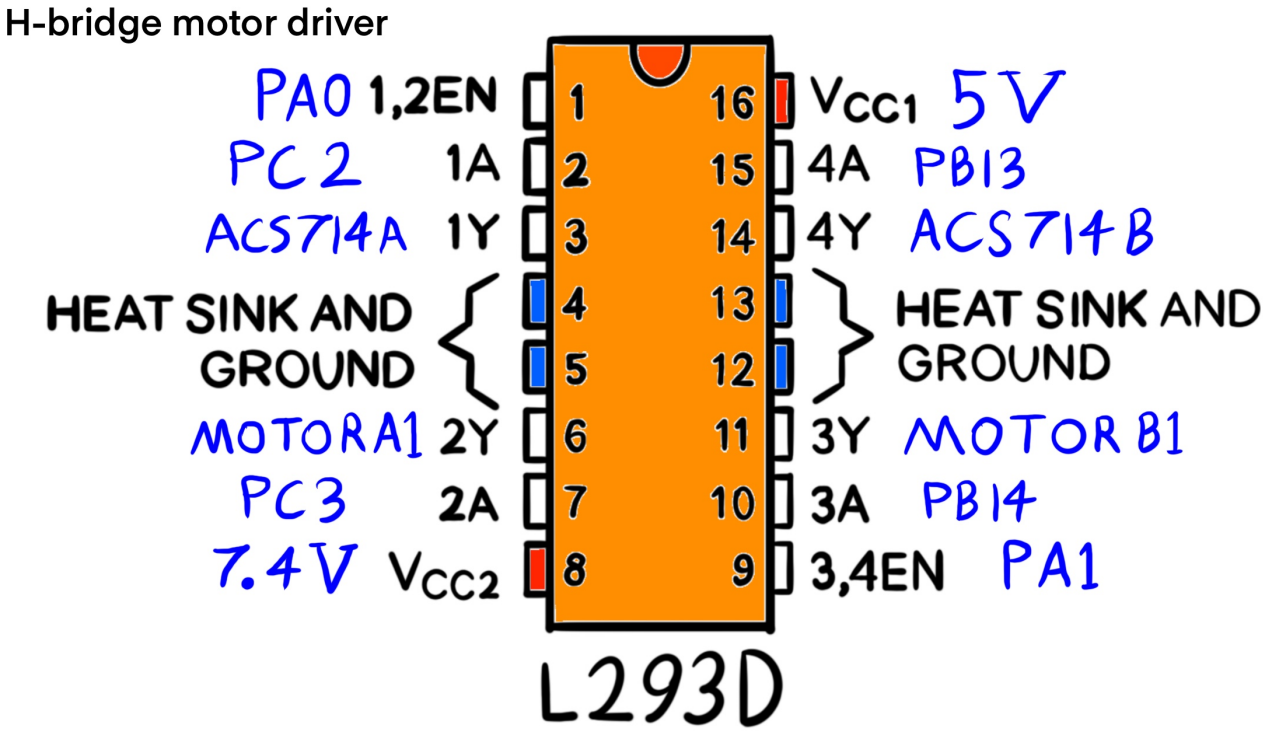


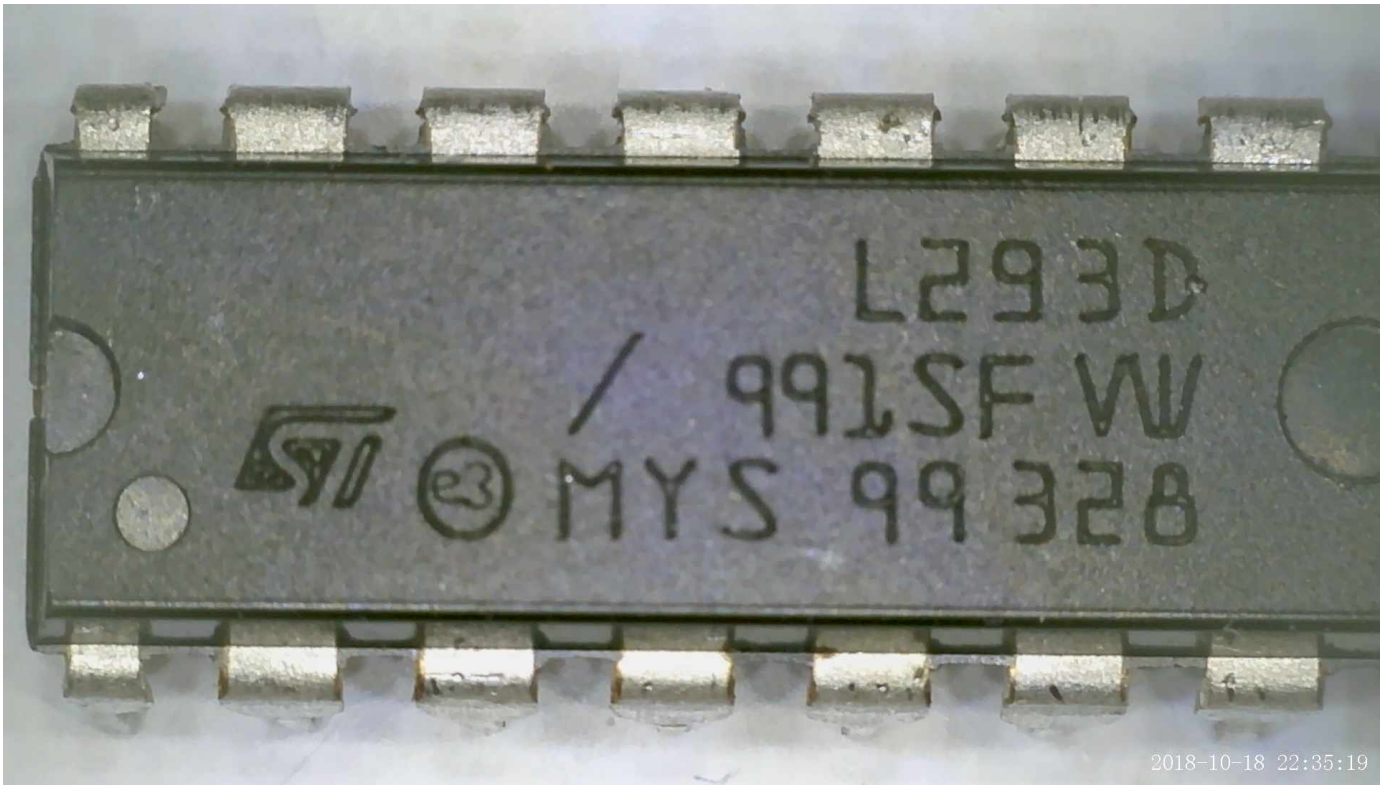
5V power supply



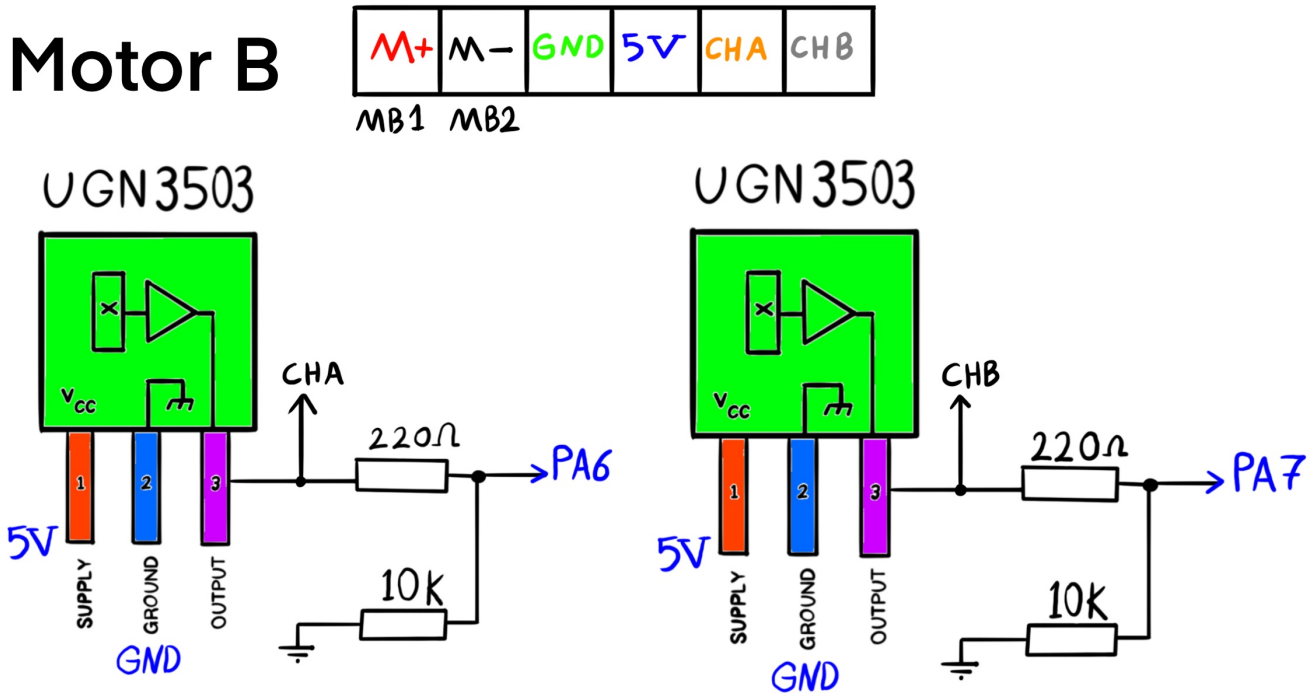
The LQR inputs are bidirectional and analog. The LQR regulates the roll and pitch angles by a choice of a suitable input. There are four digital input pins: **1A**, **2A**, **3A**, **4A**, and a pair of analog enable pins: **1,2En** and **3,4EN**. The enable pins control the speeds of rotation with a 16-bit resolution, whereas the logical input pins: **1A**, **2A**, **3A** and **4A** control the direction of rotation. A motor rotates in reverse by swapping the values of Input 1 with Input 2, switching 2 values in the memory. Therefore, LQR controls the roll and pitch angles by making changes to two variables: `rollingPWM` corresponding to **1,2EN** and `reactionPWM` corresponding to **3,4EN**. LQR adds / subtracts from the two variables when it acts in the environment. To drive a direct current actuator, the driver generates an electric potential at the two ends of the actuator's coil. An electric current in the power terminals (**1Y** and **2Y**, or, **3Y** and **4Y**) occurs whenever the electric potential at the two end points are sufficiently different in intensity. The duty cycle of a PWM signal shapes the line graph of an analog voltage. In a Voltage versus Time graph, the PWM signal is a point on the graph and varies with time. There are two independent PWM signals: the reaction wheel's motor enable pin and the rolling wheel's motor enable pin. In turn, the duty cycles of the PWM signals, tell the Integrated Circuit (IC) to adjust the electric potential at the

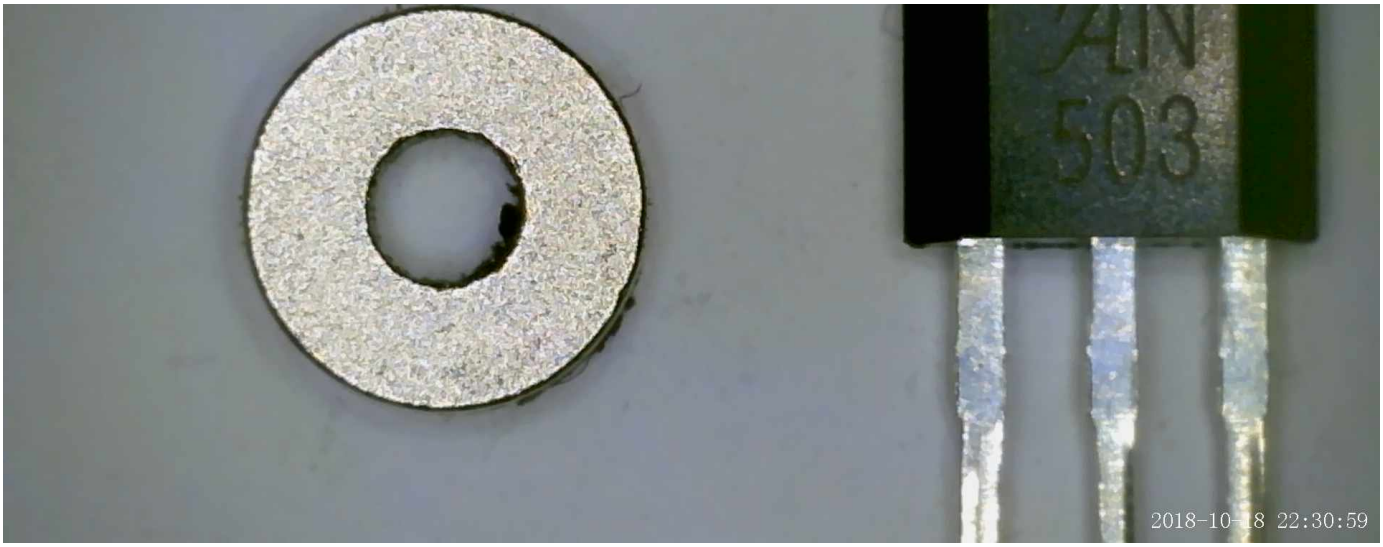
output pins of the IC: **1Y**, **2Y**, **3Y** and **4Y**. Making changes to the duty cycles with the given feedback policy `u_k`, the registers of channels one and two of Timer 2 are changed after scaling the variables and casting them to integer values.

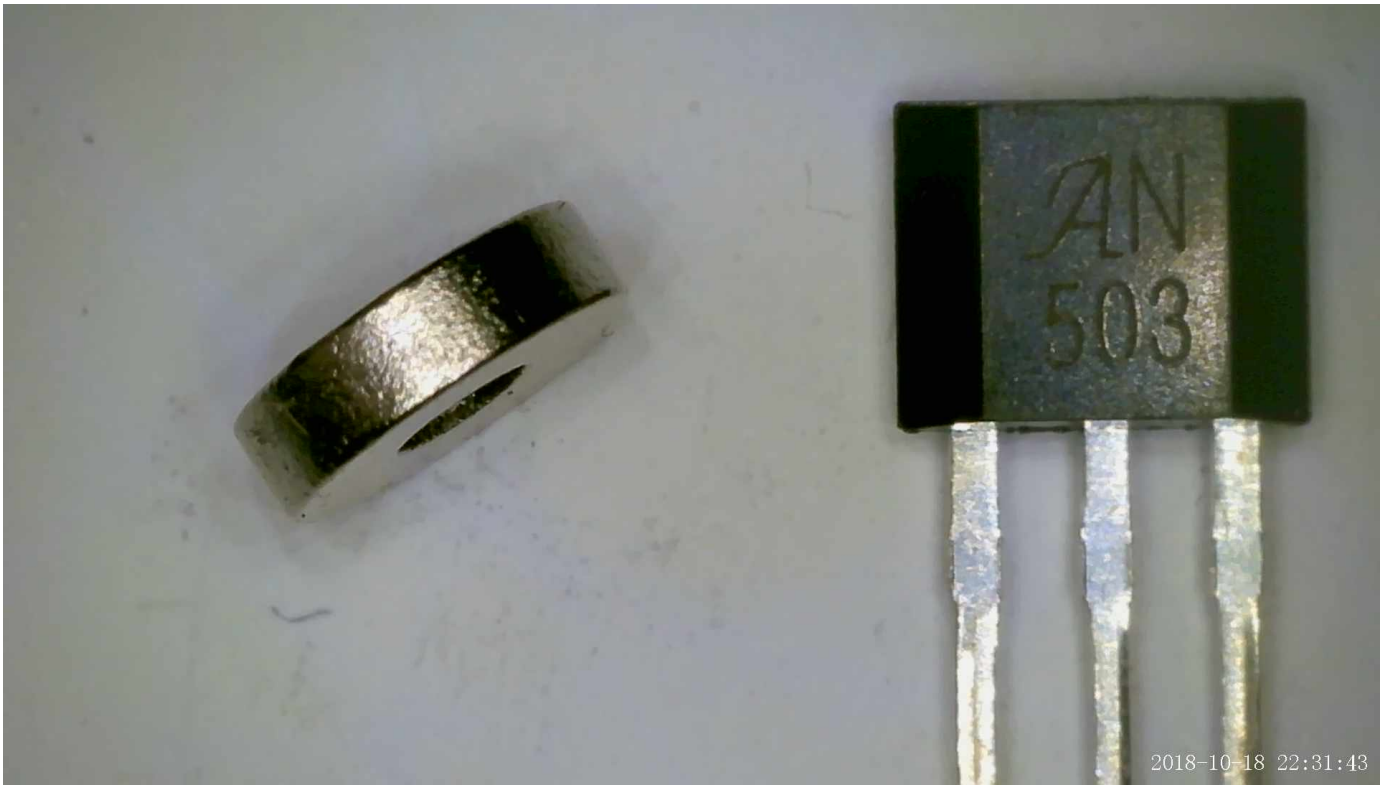




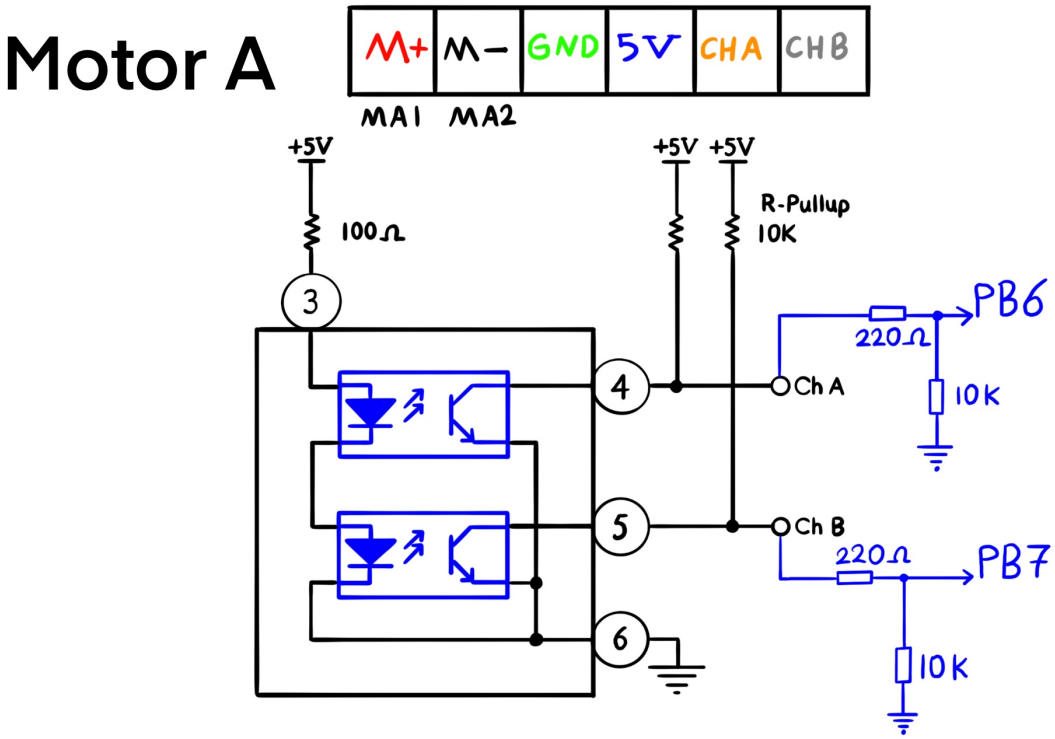
There is an encoder wheel at the opposite the end of the reaction wheel's motor. Since the gearbox reduces the speed of rotation of the reaction wheel in exchange for multiplying the output torque, the encoder's wheel rotates faster than the reaction wheel. The difference in the speed of rotation between the output reaction wheel and the input encoder wheel allows the encoder to be more precise. Also in the motor / encoder assembley, there is an array of small magnets on the circumference of the encoder's wheel. The resolution of the encoder depends on the number of magnets in the circular array and the gearbox ratio. A Hall effect sensor produces a voltage proportional to an axial component of the magnetic field vector produced by the magnetic array. The encoder measures the absolute position of the wheel using two channels. A pair of Hall effect sensors are mounted near the surface of the encoder's wheel, such that the magnets pass by the Hall effect sensors. Timer 3 of the MCU is set up to work in encoder mode, with a register of the absolute position of the encoder's wheel. When Timer 3 is in the encoder mode, it compares the pair of channels at each rising edge of the signals to find the position. Therefore, we call the `encodeWheel` function with a pointer to the `Encoder` object of the reaction wheel along with the value of the counter register of Timer 3. The function `encodeWheel` updates the velocity field of the reaction wheel's encoder struct to be used in the LQR model as a system state.





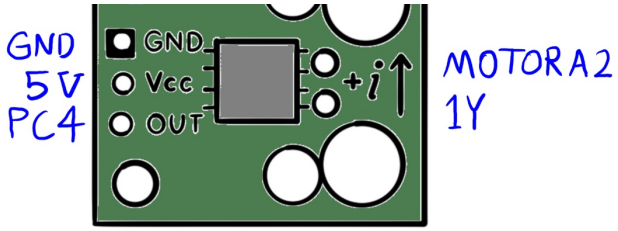


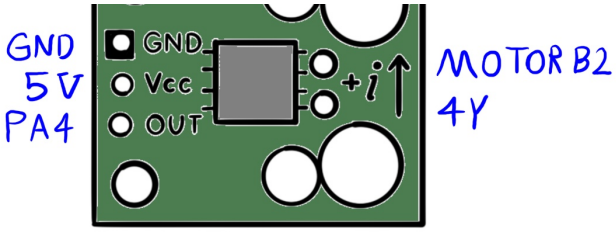
The rolling wheel's encoder works the same as the reaction wheel's encoder, except for the fact that the hardware of the channels sensors is photonic rather than magnetic. The rolling wheel encoder's disk has an alternating pattern of stripes on it for a pair of infrared light emmiting diodes and a pair of photo transistors to sense its rotation. The IR LEDs send light from one side of the wheel to be received by photo transistors on the other side through the alternating pattern. The amount of light received by the photo transistors is translated to two channels of representative electrical signals, which are fed to Timer 4 of the MCU. Supplying the `encodeWheel` function with a pointer to the rolling wheel's encoder object and the value of the counter register of Timer 4, the function call updates the wheel velocity. Before connecting the encoder signals to the MCU timer, a voltage division is applied for making sure the amplitudes of the signals do not exceed 3.3 volts.

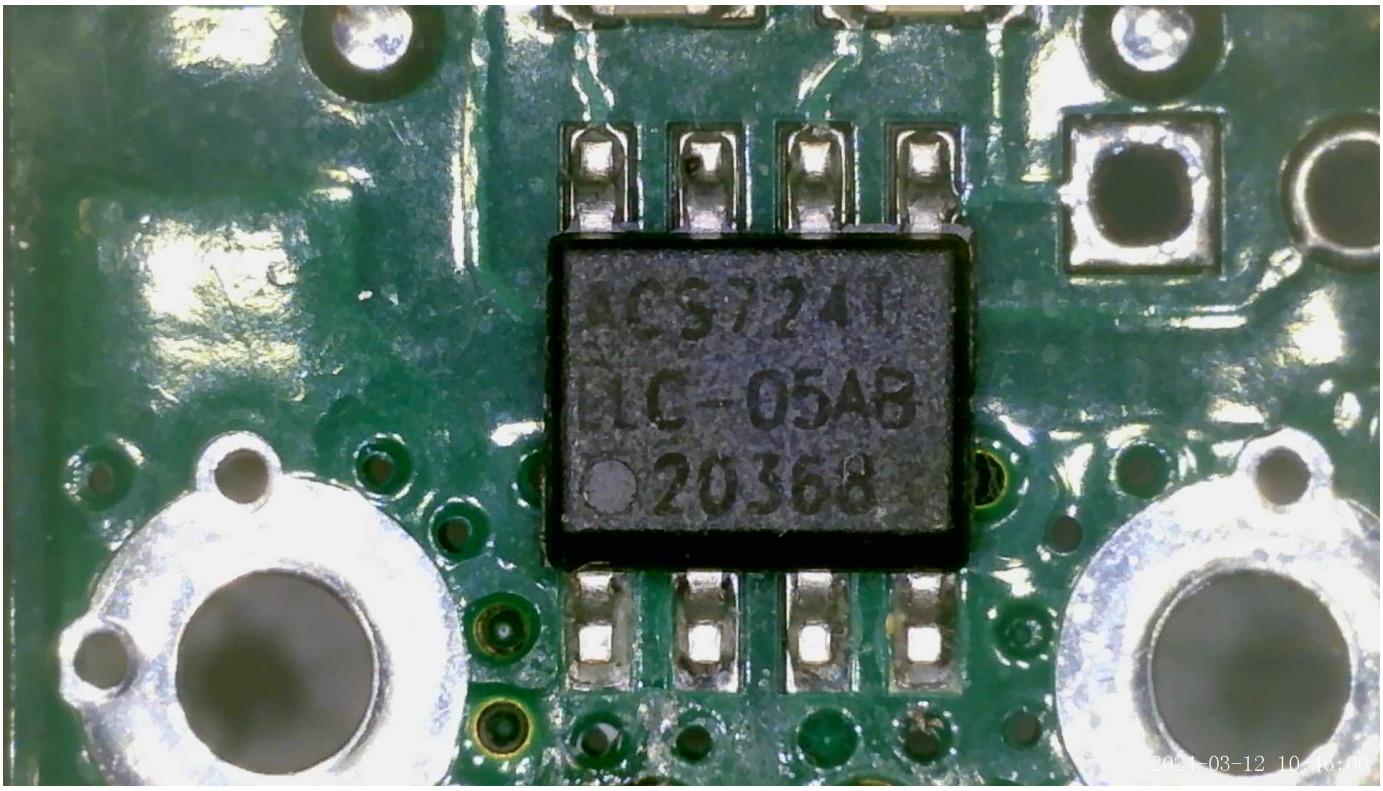


The `senseCurrent` function computes the current rates of the reaction and rolling motors. The function accepts two pointers of the `CurrentSensor` type and updates the `currentVelocity` field of the respective arguments. Measuring the electric current rate is the result of two Analog to Digital Conversion (ADC) channels, as peripherals of the MCU (pins `PC4` and `PA4` of the ADC unit). A pair of Hall effect sensors are powered using a regulated 5-Volt direct current source. To measure the current rate of the driver's output, two of the wires that connect the driver IC pins (**1Y** and **4Y**) to the respective motor coils (**MotorA2** and **MotorB2**), are routed in such a way that they pass by the respective current sensing Hall effect sensors. When the IC drives a voltage across the motor terminals (**MA1** and **MA2**, or **MB1** and **MB2**), the Hall effect sensors measure the magnetic field vector that is caused by the electric field inside the wires between the motors and the driver. The ratio-metric readings from the magnetic field vectors represent the flows of the electric current of the reaction motor's and the rolling motor's coils. The LQR model observes the current rates and regulates them to zero by generating suitable inputs.

Current sensing







Fiber Optic Gyroscopes

Resources

1. Yohanes Daud, Abdullah Al Mamun and Jian-Xin Xu, *Dynamic modeling and characteristics analysis of lateral-pendulum unicycle robot*, Robotica (2017) volume 35, pp. 537–568. Cambridge University Press 2015, doi: 10.1017/S0263574715000703.
2. Sebastian Trimpe and Raffaello D'Andrea, *Accelerometer-based Tilt Estimation of a Rigid Body with only Rotational Degrees of Freedom*, 2010 IEEE International Conference on Robotics and Automation, Anchorage Convention District, May 3-8, 2010, Anchorage, Alaska, USA.
3. K. G. Vamvoudakis, D. Vrabie and F. L. Lewis, "Online adaptive learning of optimal control solutions using integral reinforcement learning," 2011 IEEE Symposium on Adaptive Dynamic

Programming and Reinforcement Learning (ADPRL), Paris, France, 2011, pp. 250-257, doi: 10.1109/ADPRL.2011.5967359.

4. Y. Engel, S. Mannor, and R. Meir, “The kernel recursive least-squares algorithm,” IEEE Transactions on Signal Processing, vol. 52, no. 8, pp. 2275–2285, 2004.
5. C. Fernandes, L. Gurvits and Z. X. Li, "Attitude control of space platform/manipulator system using internal motion," Proceedings 1992 IEEE International Conference on Robotics and Automation, Nice, France, 1992, pp. 893-898 vol.1, doi: 10.1109/ROBOT.1992.220183.
6. G. C. Walsh and S. S. Sastry, "On reorienting linked rigid bodies using internal motions," in IEEE Transactions on Robotics and Automation, vol. 11, no. 1, pp. 139-146, Feb. 1995, doi: 10.1109/70.345946.
7. Hayes, Monson H. (1996). "9.4: Recursive Least Squares". Statistical Digital Signal Processing and Modeling. Wiley. p. 541. ISBN 0-471-59431-8.
8. Richard M. Murray, Zexiang Li, and S. Shankar Sastry, *A Mathematical Introduction to Robotic Manipulation*, CRC-Press, March 22, 1994, ISBN 9780849379819, 0849379814.
9. S. Haykin, Adaptive Filter Theory, Prentice-Hall, Englewood-Cliffs, NJ, 1986.
10. Richard S. Sutton and Andrew G. Barto, Reinforcement Learning (An Introduction), second edition, 2018, The MIT Press, Cambridge, Massachusetts, London, England, ISBN: 978-0-262-19398-6.