Simple Controllers

Robotics applies specifically to systems that have to interact with the physical world. Because of this, all robotics systems have to effect a change in their own state and/or the state of physical quantities in the surrounding environment. The specific change required depends on the robot's task. The central problem in this is that the physical world is a complex and noisy place. The robot's own mechanisms for affecting its environment are also noisy and imperfect.

We can illustrate this with an example. Consider the problem of implementing software to control the speed of a car travelling in the highway. This is a fairly straightforward task: The car has a target speed it needs to maintain, and the software needs to ensure that this speed is constant (or as close to constant as possible) independently of conditions on the road. The car can effect a change in its speed by accelerating or braking.

This may seem like a very simple task, until we start considering all the external forces that affect the car's velocity that we can't either completely predict, or incorporate into our software. For instance, the condition of the road, whether it's dry or wet, whether there is snow, or gravel, or dust. The wind's direction and speed, changes in inclination, or turns. The car itself is a changing system – as fuel is consumed the mass of the car changes. The temperature of the tires changes which affects their grip, the pressure of the air inside the tires changes as their temperature increases, which also changes their grip. The engine's performance is dependent on velocity and which gear the car is currently in.

All of these factors, as well as untold many which we didn't even think of, will result in unwanted changes to the car's speed – the quantity we want to control.

Controllers

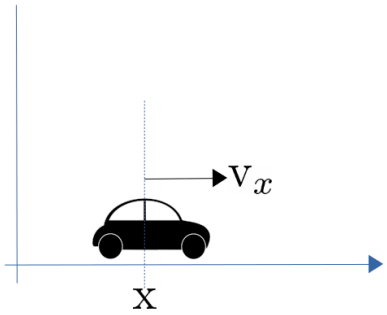
Our general task, in terms of a robotic system that is tasked with changing the state of a set of physical quantities in its environment, is to write software that will achieve and maintain the desired value for these controlled quantities despite any disturbances (whether they are external, internal, or caused by actions of other entities in the environment). A ***controller*** is a system component whose task is to achieve this.

In more formal terms, we are interested in studying a ***dynamical system.*** This is a simply a system that changes over time. In order to study such a system, we characterize the system's ***state*** in terms of a set of ***state variables***. These state variables correspond to any quantities – often corresponding to physical quantities in the environment, but also including any internal values that affect the system's operation.

We will represent state variables as a vector

$$\vec{s} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

each variable in this vector correspond to a quantity that is required to ***model the system*** and understand how to effect a specific change in the variables of interest. As an example, consider the system below which corresponds to a car moving horizontally. The system is described by two state variables, the position of the car, and its velocity:



$$\vec{s} = \begin{bmatrix} x \\ v_x \end{bmatrix}$$

A couple of very important things need to be noted at this point:

* We always work with ***a model*** of a system, this means we study how the system works, and find a suitable set of state variables which helps us work with the system – but ***this is not the same as the system***. There will be variables we didn't include, our model will be an approximation of how the system actually works, there are external effects we can't model. So bear in mind – our model is just a model, not the real thing.

* Often, the state variables themselves can not be directly known. All we have access to is either a measurement from some kind of sensor, or a value that has been computed from sensor measurements, the model, or both. This means we are always working with ***incomplete, noisy, and imperfect*** knowledge about the actual state of the system.

For the purposes of this course, we will study the simplest kind of model for a system we want to control. The system is ***linear***, which makes it easy to study and manipulate mathematically. Many real-world systems are non-linear, but they often can be approximated with a linear model well enough for the purpose of implementing a simple controller. Here's what a general linear model for a control system looks like:

$$\dot{\vec{s}} = \frac{d\vec{s}}{dt} = A \cdot \vec{s}$$

Where ***A*** is a matrix and ***s*** is our vector of state variables. The notation with a ***dot*** on top of a variable is often usef in Physics as a shorthand for the derivative with respect to time.

This is a good point to introduce an important idea you must practice for the rest of the term: ***understanding what an equation actually tells you about the world***. All equations we will study in this course relate to real-world physical systems, as such, they are not just a convenient bunch of symbols put together so we can do some algebra and calculus on them – they are a compact and accurate description of something we think is true about the world we're modelling. Here's how we translate the equation above into a statement about the world that we can understand:

$$\dot{\vec{s}} = \frac{d\vec{s}}{dt} = A \cdot \vec{s}$$

The change in our state variables over time — is given by — A linear function applied to our state variables

- or -

is the result of

Let's look at an example of how the model above can be used to understand the simple system above, with the car travelling horizontally in 1D. With two state variables for position and velocity, our model looks like so:

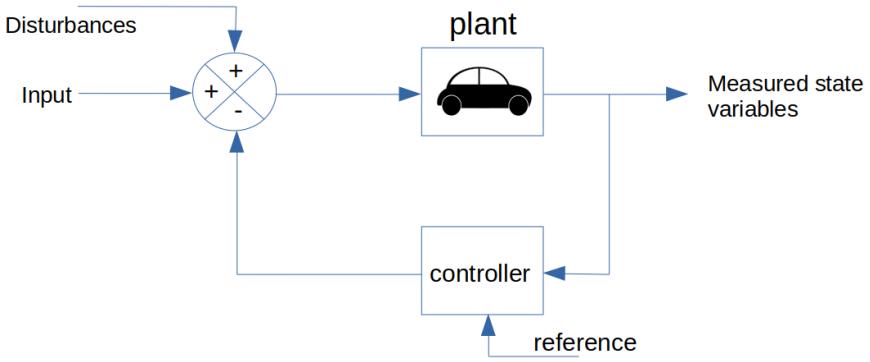
$$\frac{d\vec{s}}{dt} = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dv_x}{dt} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ v_x \end{bmatrix}$$

The first row of this model tells us that the change in position over time is equal to the velocity of the car – which makes sense. The second row of our model tells us that the change in velocity over time is zero, that is, the velocity is constant. How could we model a car whose velocity also changes? We can do this by adding one more state variable and expanding our model as follows:

$$\frac{d\vec{s}}{dt} = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dv_x}{dt} \\ \frac{da_x}{dt} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ v_x \\ a_x \end{bmatrix}$$

Now our model states that the change in position over time is given by the velocity of the car, and the change in velocity over time is given by the acceleration. The acceleration is zero, so the velocity is constant.

Why is this model useful? Our controller's function is to ensure the system's state variables reach a desired value and remain there. In order to achieve this, we need to understand how these state variables are changing (that's what the system tells us), then figure out the ***difference between their desired value and their current value –*** which we call the ***error,*** and then have the controller effect a ***change in their value in a direction that reduces error***.



A block diagram of our system is shown above. Traditionally, the system being controlled is called the ***plant***, and the controller's job is to ***read the state variables***, ***compare*** them ***with the desired reference*** value, and apply a feedback signal that will drive the system toward the reference value. The ***input*** here represents any externally applied, but intentional change to the system (in the example of the car, maybe the driver is pressing the accelerator, or the brake, and the system will respond to that input). At the same time, there are external, un-intentional, nuisance forces (called ***disturbances***) that affect the system and will tend to drive it away from the desired reference.

The diagram above shows a ***feedback loop***. Any useful controller must use a feedback loop to effect the change in the system. Notice the sign is negative in the diagram above. This ***does not mean the controller only applies feedback with a negative sign!*** The diagram represents a ***negative feedback loop***. Such a loop is intended to reduce (make smaller) the changes to the system, whereas a positive feedback loop would amplify (make bigger) the changes to the system.

Useful controllers are always in the form of a ***negative feedback loop*** – they drive the value of the state variables toward their desired reference ***while also reducing the amount of change*** (specifically, change due to any disturbance to the system) so that the variables remain stable once the reference is reached.

Mathematically, we can now write down our system model with a controller as shown below:

$$\dot{\vec{s}} = \frac{d\vec{s}}{dt} = A \cdot \vec{s} + B \cdot \vec{u}$$

The equation above now states that the change in state variables is the result of ***a linear function applied to the state variables and a linear function applied to the control signals*** $\vec{u}$ **.** Our controller has to figure out the correct control signal that will move the system toward the desired reference.

To make this concrete, let's look again at the example of the very simple car above. We can expand the system to include a control signal that either causes the car to accelerate (***acc***), or brake (***brk***).

$$\frac{d\vec{s}}{dt} = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dv_x}{dt} \\ \frac{da_x}{dt} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ v_x \\ a_x \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} acc \\ brk \end{bmatrix}$$

Before introducing the controller, the system had a constant acceleration of zero, now the controller effects a change in state variables by changing the acceleration of the car (matrix ***B***)– which in turn changes the velocity, which then affects the position (matrix ***A***).

Simple Controllers

Let's look at how we may implement the simplest of controllers. In the example above, suppose we want a controller that will ensure the car's velocity is kept at a constant value (most cars have this, it's called *cruise control)* despite disturbances. Our controller can cause the car to accelerate or to brake, and for simplicity let's assume that the amount of acceleration or braking that is applied is a constant.

The simplest controller we could think of works like this:

```
accel=0;
brake=0;
if (velocity<reference)
   accel=1;
else if (velocity>reference)
   brake=1;
```

This is called a ***bang-bang*** controller. It turns a control signal on/off depending on the values of state variables. You can imagine the effect such a controller will have on the car (and the people riding it). It will ensure that the velocity is reasonably close to the reference, but aside from very brief moments when the velocity is actually equal to the reference, the car will be either accelerating or braking most of the time – not a smooth ride.

We can introduce a ***tolerance*** value to reduce this problem, that is, we can make sure that as long as the state variable's value is within some small distance from its reference value, the controller will take no action. A more general form of a ***bang-bang*** controller would thus look like this:

```
err=(ref-x)
if  (err < -k)
   u=a1
else if (err > k)
   u=a2
else
   u=0
```

Here ***x*** is the state variable we're controlling, ***k*** is a small constant that determines how big the error ***err*** can be before the controller takes action, ***u*** is a control signal we apply to the system, and the two constant values ***a1*** and ***a2*** are suitable control amounts to use when the state variable's value is below, or above the reference ***ref*** value respectively.

This kind of controller is actually suitable to some real-world applications. Think for instance of a thermostat (like you would find in an air-conditioning unit, or a kitchen oven). It will do the job when the variable we're controlling changes relatively slowly in response to inputs. But for situations like we

had with the car, they cause the system to oscillate around the reference value and they waste energy because they are applying a control input for significant amounts of time.

One fundamental problem with the ***bang-bang*** controller is that it doesn't at all use the value of the error term when deciding what control input to provide. In the case of the car, one could imagine a controller that applies the accelerator much more strongly if the current velocity is far away from the desired reference, but the acceleration becomes more gentle as the difference decreases. This would produce a smoother ride.

A controller that provides a control input that depends on the magnitude of the error is called a ***P*** controller. This is simply because the control input is ***proportional*** to the difference between our state variable and its intended reference. The implementation of a ***P*** controller is straightforward:

```
err=(ref-x)
u_p=k_p * err
```

Here ***err*** is the error term, k_p is a tunable constant, and u_p is the control input applied by the ***P*** controller.

PID Controllers

The ***P*** controller will do a reasonable job with fairly general control situations. However, it does have a tendency to ***overshoot*** and ***oscillate***. This happens because all physical systems have a certain amount of inertia, and their response to a control input is not instantaneous, there is always ***lag***. Therefore the ***P*** controller keeps applying a control signal well after the system (left alone) would have reached the reference value due to previously applied control inputs.

Once the system overshoots the reference, the ***P*** controller will apply an opposite control input. This new input first has to counteract the inertia in the system, then has to drive the system back toward the reference value. Often, the controller will end up overshooting again – but in the opposite direction. The overall effect is that the system oscillates around the reference value.

To avoid this, one could use a ***P*** controller with a small constant k_p, but this has the effect of increasing the time it takes for the controller to bring the system to the reference value. If the constant is too small, the controller may not be able to compensate for disturbances.

A more robust controller, one that allows us to bring a system to the desired reference smoothly, reasonably quickly, and with minimal oscillations, is the ***PID controller.***

The ***PID*** has three components. The ***P term*** is just the same proportional term we saw above. It does most of the heavy lifting. But, we now add two additional components to the final control input.

Firstly, we add a term that is proportional to the ***difference (or derivative in the continuous case) of the error (the D term).*** This means that we take the difference between ***the error as it was on the previous call to the controller***, and ***the current error***. This term is useful because it can help avoid

overshoot. Let's look at an implementation of a ***PD controller*** which contains both the ***P*** and the ***D*** terms:

```
err=(ref-x)
diff=(err - prev_err)
u_p=k_p*err
u_d=k_d*err
u=u_p+u_d
prev_err=err
```

Here the current error is ***err***, the error from the previous call to the controller is ***prev_err***, the proportional term provides control input u_p, the difference term provides control input u_d, and the combined control input that will be applied by the controller is u. The values of k_p and k_d have to be tuned to achieve a good balance for the specific application we're working with but k_d is typically significantly smaller than k_p.

How does this work? Suppose the error is large and positive, but it's decreasing because of the action of the controller. The ***P*** term will provide a large and positive control signal, the ***D*** term will provide a control input with the opposite sign to that of ***P*** because the error is decreasing (***diff*** is negative). Because the constant k_p is significantly larger than k_d and $|diff|$ is generally less than $|err|$, the overall result is still a large, positive control input to reduce the overall error.

Now suppose the system is approaching the reference value, the error is still positive, but not so large, the control input from the ***P*** term is still positive. ***If |diff| is large***, we run the risk of overshooting – the error is changing quickly even though its magnitude is small. In this situation, the control input provided by the ***D*** term can dampen, cancel, or even counter-act the effect of the ***P*** term – hopefully helping the system avoid overshoot (or at least reducing its amplitude).

This all depends on carefully tuning the constants for the ***P*** and ***D*** terms, if they are not well balanced, the interaction between the two terms can result in a controller that is worse than a properly tuned ***P*** controller!

A ***PD controller*** is a solid general purpose controller, and you can write one for pretty much any application for which you need a good controller. However there is one possible remaining problem. On some situations, the controller can achieve equilibrium at a value of the state variable that is close but not exactly the desired reference value. That is, the error is not actually zero (or close enough as to make no difference).

If we care about this (we don't always do! In a care, we don't really mind if the velocity is 100.1 Km/h instead of 100.0), then we need to add the third term in a ***PID*** controller. The ***I*** term uses the ***integral*** of the error over time, and provides a control input whose goal is to ensure that the error actually reaches zero in the long run.

The complete ***PID*** controller is shown below:

```
err=(ref-x)
diff=(err - prev_err)
intg=sum of the previous T error values
u_p=k_p*err
u_d=k_d*diff
u_i=k_i*intg
u=u_p+u_d+u_i
prev_err=err
```

The ***I term*** uses the ***sum of the errors from the previous T calls to the controller***. If the error is so small that the ***P*** and ***D*** terms aren't providing sufficient control input to drive it toward zero, the accumulation of error will allow the ***I*** term to provide the required push. The value of ***T*** must be sufficiently large (in the literature, you will find this includes ***all*** error terms since the controller was activated). In practice, you don't need to carry around an ever increasing list of error values, but you must ensure the ***I*** term has access to enough error information that its control signal will have a reasonable magnitude when the errors are small but not zero for some time.

Like the ***PD*** controller, the ***PID*** controller requires carefully tuned constants for each of its three terms. Badly tuned ***PID*** constants will result in a controller that may fail to bring the system under control, may oscillate too much, or in the opposite situation, may be quite slow to achieve the desired reference.

Unfortunately, the particular constants that will work best for a specific application can often only be found empirically, by tuning the controller, testing it, then tuning it some more until the controller works reasonably well. This is a minor disadvantage of a controller that is otherwise very easy to implement and works well on a wide range of real-world situations.