Introduction to Control Theory: The Control Problem

In [ ]:
```
#| default_exp intro_to_control_theory
```

In [ ]:
```
#| hide
%load_ext autoreload
%autoreload 2
```

In [ ]:
```
#| hide
from IPython.display import Image
from IPython.display import HTML

# To support animations
from matplotlib import animation, rc
from matplotlib.animation import FuncAnimation
```

In [ ]:
```
#| hide
# This is equivalent to rcParams['animation.html'] = 'html5'
rc('animation', html='html5')
```

In [ ]:
```
#| export
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Notebook Objectives

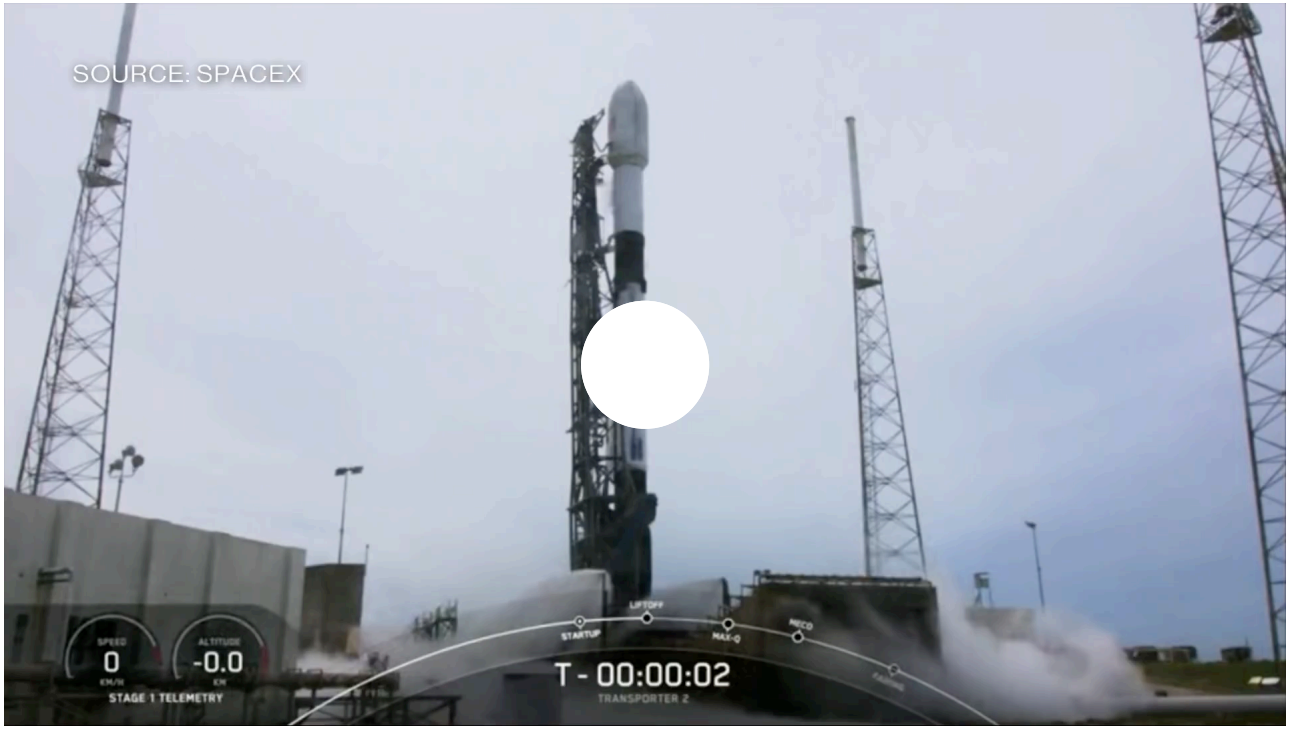
This notebook provides an overview of the big picture problem that we're trying to solve as control system engineers.

- Control theory makes it possible to solve a number of engineering problems, not only as control engineers but as any engineer
  - Switching power regulators
  - Automatic gain control circuits that automatically increase or decrease the gain of a signal
  - Isolation system in a motor mount that is sensitive to vibrations
  - Industrial robotics
  - etc.

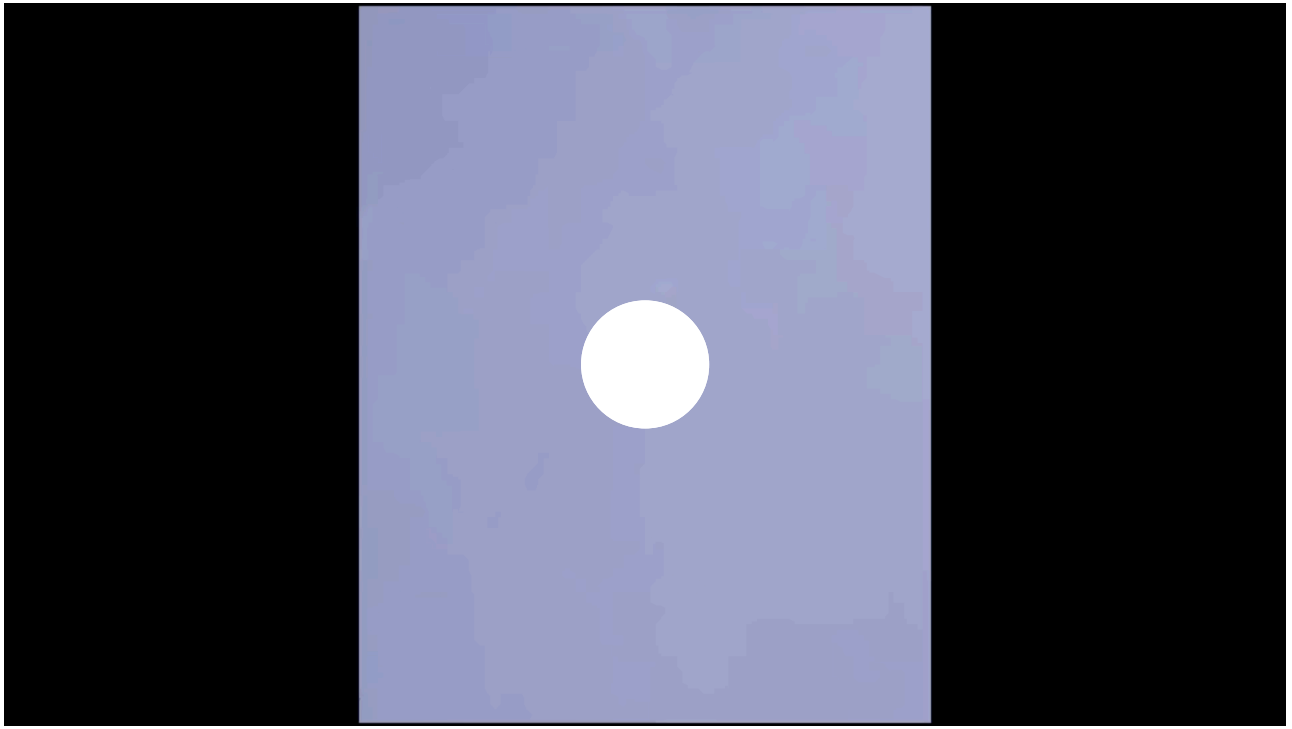
Control system is:

- Building models of your system
- Simulate them to make predictions
- Understanding the dynamics and how they interact with the rest of the system
- Filtering and rejecting noise
- Selecting or designing hardware (sensors, actuators)
- Testing the system in expected and unexpected environments
- It is understanding your system!

Some (cool) examples



SpaceX Nails Landing of Reusable Rocket on Land, From Bloomberg Technology

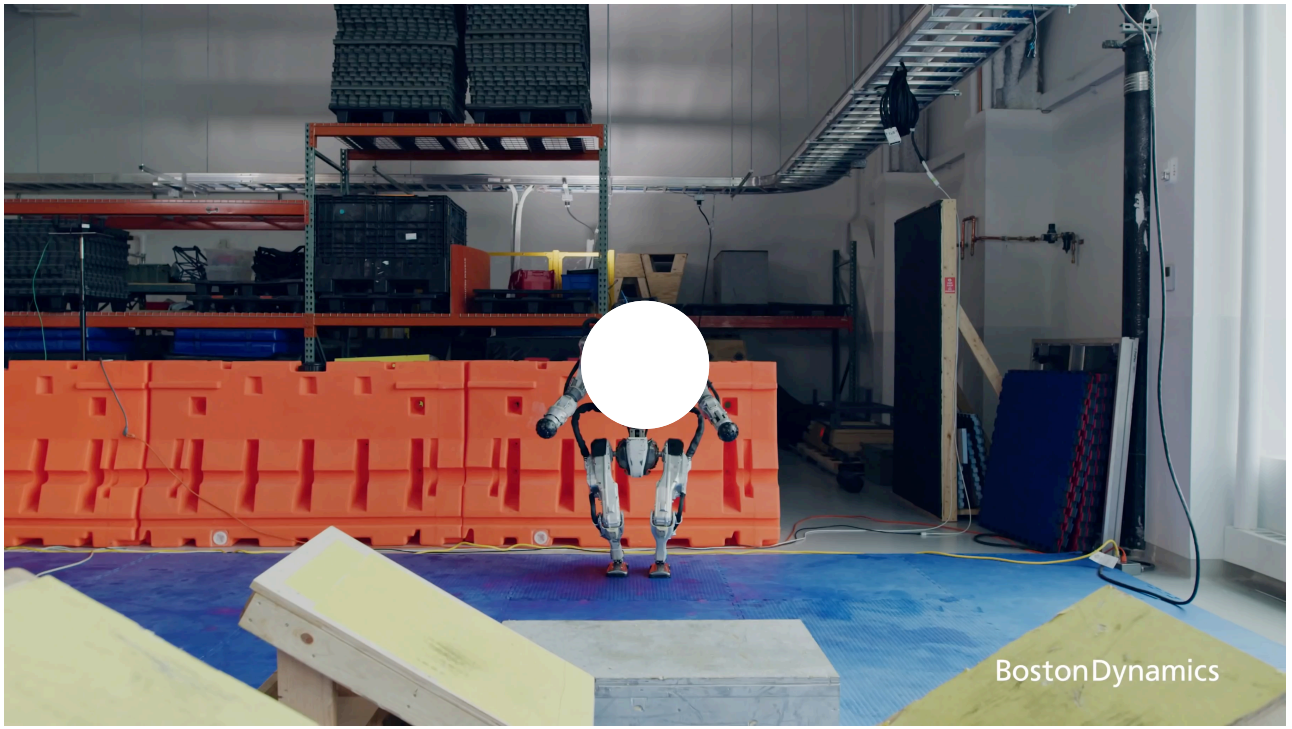


SpaceX Nails Landing of Reusable Rocket on Land (close up)

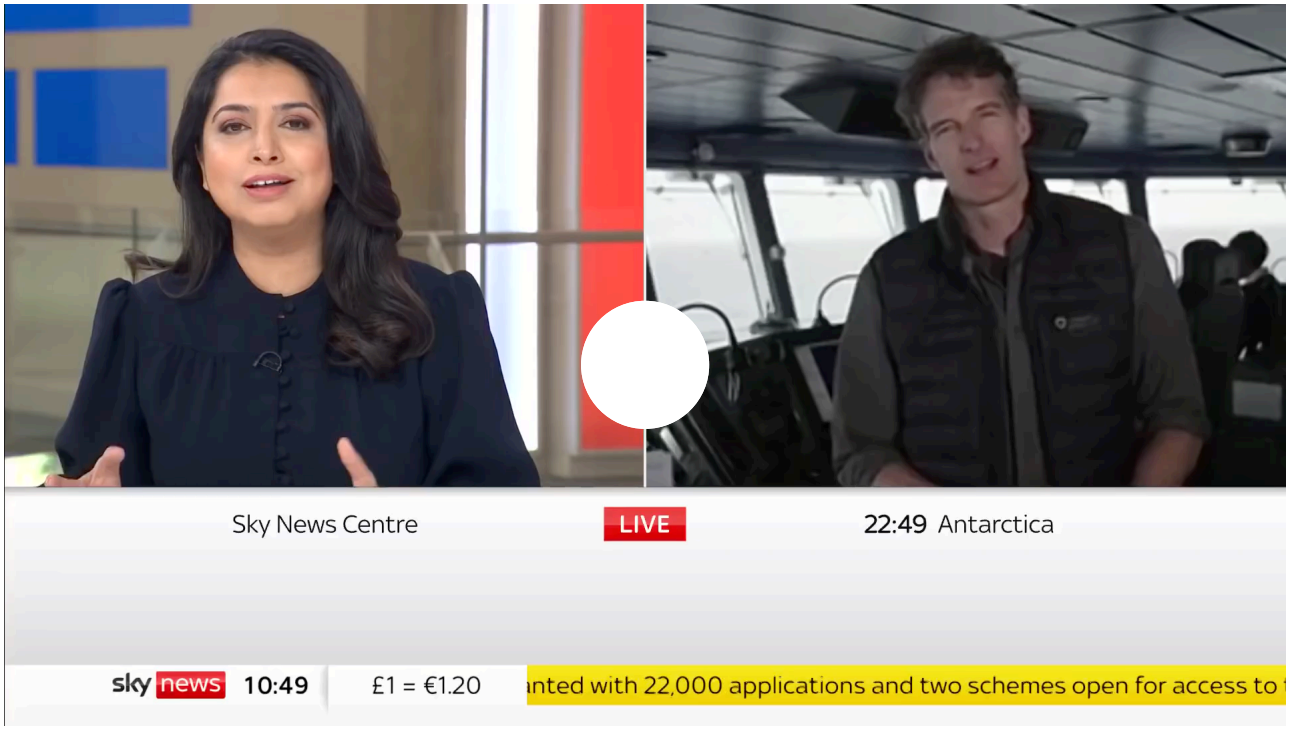
Where does Control System Engineering come into place?

- Attitude control
- Landing control
- Trajectory tracking
- Land control (e.g., antennna tracking)

Autonomous Ferries (Rolls-Royce)



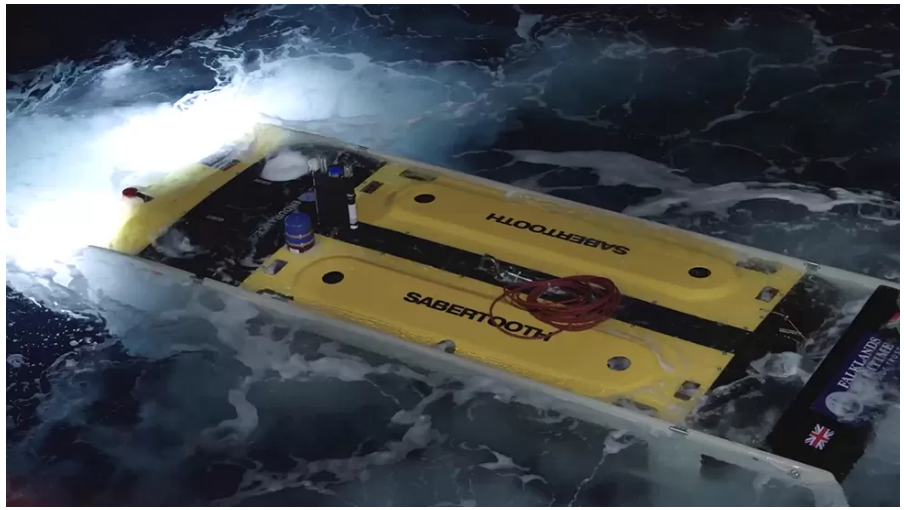
Atlas (Boston Dynamics)


Sky News Centre
LIVE
22:49 Antarctica
sky news 10:49
£1 = €1.20
nted with 22,000 applications and two schemes open for access to

Last voyage of Shackleton's Endurance ship

Source: Endurance22 Expedition

BBC



How do you get a system to do what you want?

Control Theory is a mathematical framework that gives us the tools to develop autonomous systems

---

What is a system?

The concept is straigthforward, but the term is sometime applied very generically

For us:

> A **system** is a collection of interconnected parts that form a larger more complex whole

Engineering problems are usually complex.

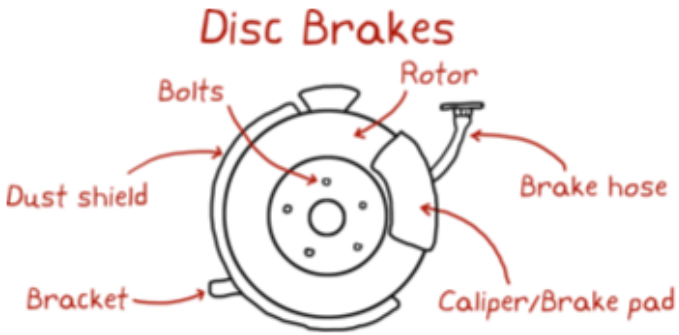
- Divide complex projects into smaller parts, or *systems* makes it possible to simplify the problem and to specialise in specific areas
- Usually more than one layer of complexity
- Each one of the interconnected parts that form a larger system can be a complex system in its own right!

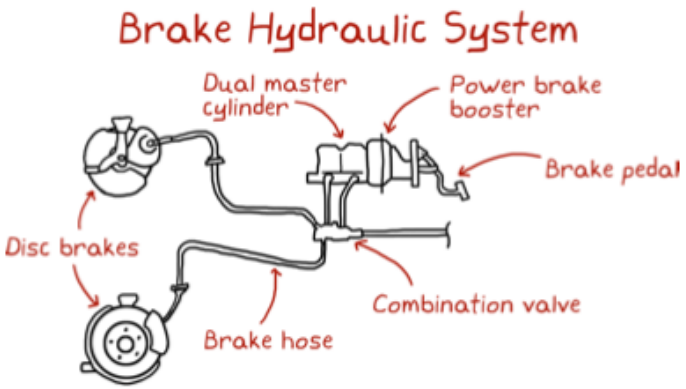
Control systems

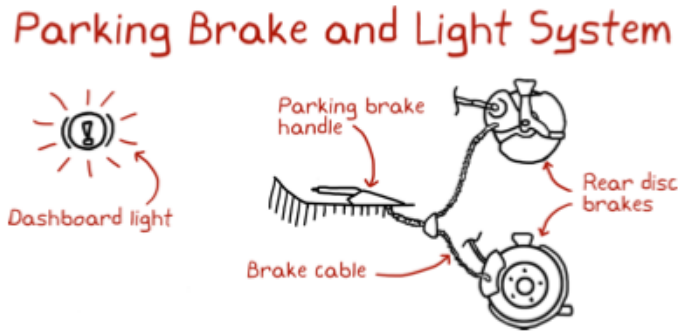
As a control engineer:

- Goal is to create something that meets the functional or performance requirements you set for the project.
- The collection of the interconnected parts that are created specifically to meet these requirements are the *control system* (at least in general)
- More specifically: **A control system is a mechanism (hardware or software) that alters the future state of a system**
- For any project however the control system again might be a collection of interconnected parts that require specialists:
    - sensor experts,
    - actuators experts,
    - digital signal processing experts,
    - state estimation experts,
    - etc.

For example, *car braking system*:



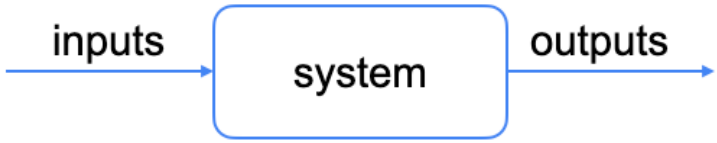




And of course, the braking system itself is just one of the main interconnected parts that create the car.

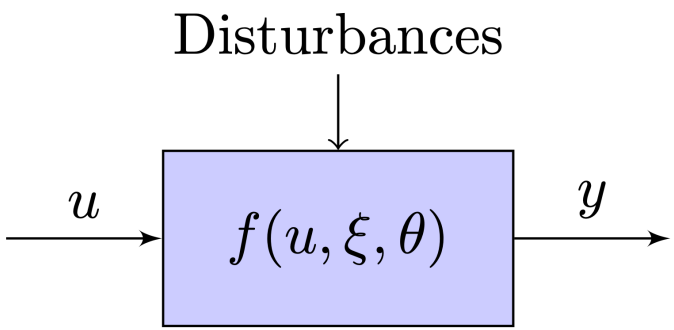
System as a box

- We represent systems graphically as a box
- Arrows going into the box are inputs
- Arrows going out of the box are output



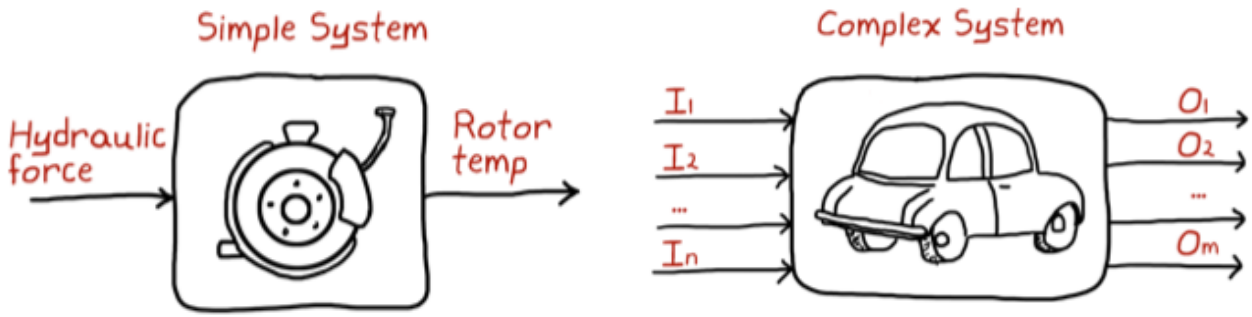
- The system inside the box is described through a math model (e.g. equation of motion)
- The "box" can be used to describe systems that are simple or complex

So if we want to start representing it more formally:



- u: control inputs
- ξ: disturbances
- f: system (physics is involved here!)
- θ: system parameters (e.g., mass, inertia, spring constants, etc.)
- y: output

A system can be anything..



- Sometimes we also talk about parts, components, subsystems and processes (or plants)

Three different problems

- Three parts to our simple block diagram
  - The system
  - The inputs (that drive the system)
  - The output (that the system generates)

At any given time one of the three parts are unknown, and the part that is unknown defines the problem that you are solving.

The system identification problem

- As a practicing engineer you won't always be given a model of your system
- You will need to determine it yourself
- This is done through a process called **System Identification**

- You might be doing **System Identification** if you are asking yourself these questions:
  - What are the mathematical equations that will convert my known inputs into my measured outputs?
  - What should I model?
  - What are the relevant dynamics of my system?
  - What is my system?

Black box vs White box

In system identification, black box and white box identification are two different approaches for modeling a system.

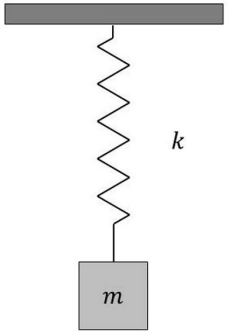
Black box:

- You are given a box that you cannot open, but you are asked to model what is inside
- Subject what is inside the box to various known inputs, measure the output and then infer what is inside the box based on the relationship between the input and the output.

White box

- Now you can see exactly what is inside the box (hardware, software).
- You can write the mathematical equations of the dynamics directly (e.g. Netwon's equations of motion).

For example, for a sping-mass system:



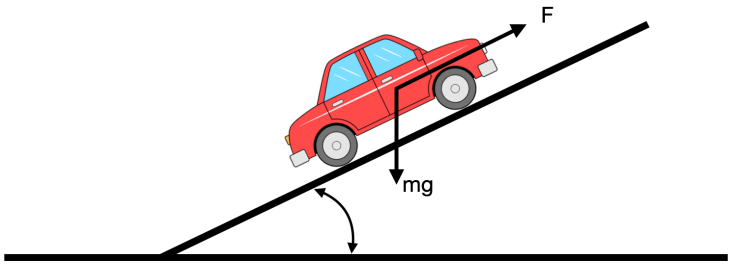
$$F = m\ddot{x} + kx$$

- Even the while box method might require to run tests and applying inputs to measure outputs to calculate the parameters of your system.
- E.g., modeling a linear spring: you know the equation but what is the exact spring constant?

Usually, you need to do a bit of both.

Example: modeling a car

Let's consider an example: car cruise control, using a simplified mathematical model.



$$m\frac{dv}{dt} + \alpha|v|v + \beta v = \gamma u - mgsin(\theta)$$

- Input: gas pedal
- Output: speed (speedometer)
- Disturbance: the slope of the road

Assumptions to simplify the model:

- Thrust is proportional to how much pressure we put on the pedal and this directly translates into how much is open the gas valve
- Frictions and drags are linear with speed
- Small angles: $\theta < 30^o$ so that $sin(\theta) \approx \theta$

Using our assumptions, the previous model can be simplified as:

$$m\frac{dv}{dt} + \alpha|v|v + \beta v = \gamma u - mgsin(\theta) \approx m\frac{dv}{dt} + \beta v = \gamma u - mg\theta$$

Even in those cases where we want to simplify the model, it can often be useful to have it in a **standard representation**, in this case *state space representation*:

$$\dot{\mathbf{x}} = [\dot{x_1}, \dot{x_2}]^T$$

$$\begin{cases} \dot{x_1} & = x_2 \\ \dot{x_2} & = -\frac{\alpha}{m}|x_2|x_2 - \frac{\beta}{m}x_2 + \frac{\gamma}{m}u - gsin(\theta) \\ y & = x_2 \end{cases}$$

$$\mathbf{x}(t_0) = x_0$$

- This is a system of first-order ordinary differential equations (ODEs) that describes the motion of our car along a road.
- Velocity is the derivative of the position
- Two state variables: position x_1 and velocity x_2
- Two inputs: u (Force applied to the car) and θ (angle of the road with respect to the horizontal)
- One output: velocity of the car
- The initial condition $\mathbf{x}(t_0) = x_0$ specifies the initial position and velocity of the car at time t_0.

- The state $\mathbf{x}$ includes the minimal amount of information at the current time to predict its behaviour in the future, only based on the knowledge of its future inputs.
- In the case where we have differential equations from actual physical systems we also need the initial conditions.
- Note that there is no time in the equations above: **time-invariant** system

State equations - standard form:

$$\dot{x} = f(x, u, t; \theta)$$

$$y = g(x, u, t; \theta)$$

$$x(t_0) = x_0$$

Standard representations

- "Standard representations" are the bulding blocks of the engineering language
- Analysis and design are based on having the system in a standard representation
- Specific problems typically need to be translated into a standard representation
- Control System Software typically assumes that the system is already in a standard representation (and expects inputs based on that)

This looks like something we can code up!

In [ ]:
```
#| export
class Car:
    _g = 9.8 # Gravity

    def __init__(self, x0, params):
        self._x_1 = x0[0] # position (along the road)
        self._x_2 = x0[1] # velocity (along the road)
        self._m, self._alpha, self._beta, self._gamma = params

    def step(self, dt, u, theta):
        self.theta = theta
        self._x_1 = self._x_1 + dt*self._x_2
        self._x_2 = self._x_2 + dt*(-self._alpha/self._m*abs(self._x_2)*self._x_2 - \
                                    self._beta/self._m*self._x_2 + self._gamma/self._m*u - \
                                    Car._g*np.sin(theta))

    def speedometer(self):
        v = self._x_2
        return (v,)

    # Utility function to simplify plotting
    def sensor_i(self):
        # Rotation matrix to get back to the main frame.
        R = np.array(((np.cos(self.theta), -np.sin(self.theta)), (np.sin(self.theta), np.cos(self.theta))))
        x_i, y_i = R.dot(np.array([[self._x_1],[0]]))
        v = self._x_2
        return (x_i, y_i, v)
```

Model Linearisation (intro)

- The previous model was relatively complicated (e.g., included sin and cos, module of...)
- We can simplify it to make it more mathematically manageble
- This is called *Linearisation*

- *Linearisation involves creating a linear approximation of a nonlinear system that is valid in a small region around the operating or trim point, a steady-state condition in which all model states are constant.*
- *Linearisation is needed to design a control system using classical design techniques (e.g., Bode plot and root locus design) and analyse system behavior, such as system stability, disturbance rejection, and reference tracking.*

Equilibrium

Given a system:

$$\dot{x} = f(x, u, t; \theta)$$

and given an input $u(t)$,

a state $\bar{x}$ is said to be an equilibrium if:

$$0 = f(\bar{x}, u, t; \theta)$$

the system does not move from $\bar{x}$.

Linearising around an equilibrium

- We need to first assume that the system operates near an operating point or equilibrium point.
- At this point, nonlinear effects can be neglected.

- For simplicity:

  - we neglect time dependency
  - we omit parameters θ
- Given input $\bar{u}$

- Equilibrium $\bar{x}$

we can write:

$$\dot{x} = f(x, u)$$

$$0 = f(\bar{x}, \bar{u})$$

We can also define new variables $\tilde{x} = x - \bar{x}$, which represents the deviation of the state from the equilibrium point and $\tilde{u} = u - \bar{u}$, which represents the deviation of the input from the equilibrium input:

$$\tilde{x} = x - \bar{x}$$

$$\tilde{u} = u - \bar{u}$$

Taylor expansion around the equilibrium

- Approximate a function as an infinite sum of its derivatives evaluated at a single point
- Given a function f(x), the Taylor series expansion around a point a is:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

where $f^{(n)}(a)$ denotes the nth derivative of f evaluated at a, and $n!$ is the factorial of n.

- We apply Taylor expansion to the state equations around the equilibrium and we keep only the linear terms

$$\dot{x} = f(x,u) = f(\bar{x},\bar{u}) + \frac{\partial f}{\partial x}\Big|_{x=\bar{x},u=\bar{u}}(x-\bar{x}) + \frac{\partial f}{\partial u}\Big|_{x=\bar{x},u=\bar{u}}(u-\bar{u}) + \mathcal{O}(\Delta^2 x) + \mathcal{O}(\Delta^2 u)$$

To linearize the system described by the following equation:

$$m\frac{dv}{dt} + \alpha|v|v + \beta v = \gamma u - mgsin(\theta)$$

- We need to first assume that the system operates near an operating point or equilibrium point. At this point, the velocity and input are assumed to be small enough such that their nonlinear effects can be neglected.
- We can then use Taylor series expansion to approximate the nonlinear terms around this equilibrium point, keeping only the linear terms.

- Assuming an equilibrium point of $v = v_0$ and $u = u_0$, the Taylor series expansion of the nonlinear terms around this point is:

$$\alpha|v|v \approx \alpha|v_0|v + \frac{\alpha v_0^2}{2}\left(\frac{v - v_0}{|v_0|}\right)$$

- Substituting this approximation into the original equation, we get:

$$m\frac{dv}{dt} + \alpha|v_0|v + \frac{\alpha v_0^2}{2}\left(\frac{v - v_0}{|v_0|}\right) + \beta v = \gamma u - mgsin(\theta)$$

- We can then define a new variable $\tilde{v} = v - v_0$, which represents the deviation of the velocity from the equilibrium point. Using this variable, we can rewrite the equation as:

$$m\frac{d\tilde{v}}{dt} + (\alpha|v_0| + \frac{\alpha v_0^2}{2|v_0|})\tilde{v} + \beta v_0 = \gamma\tilde{u}$$

where $\tilde{u} = u - u_0$ is the deviation of the input from its equilibrium point.

- This linearised equation is in the form of a standard linear system, with $\tilde{v}$ as the state variable, $\tilde{u}$ as the input, and m, $(\alpha|v_0| + \frac{\alpha v_0^2}{2|v_0|})$, and βv_0 as the system coefficients.

In control system we tend to use standard representations and in the case of our state space model, and apply the Taylor expansion to that:

$$\dot{\tilde{x}} = \dot{x} - \dot{\bar{x}} = f(x,u) - f(\bar{x},\bar{u})$$

$$\dot{x} = f(x,u) = f(\bar{x},\bar{u}) + \frac{\partial f}{\partial x}\Big|_{\substack{x=\bar{x} \\ u=\bar{u}}}(x-\bar{x}) + \frac{\partial f}{\partial u}\Big|_{\substack{x=\bar{x} \\ u=\bar{u}}}(u-\bar{u}) + \mathcal{O}(||\Delta x||^2) + \mathcal{O}(||\Delta u||^2)$$

In this case, the matrices $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial u}$ are called **Jacobians**. These are matrix of numbers.

$$A = \frac{\partial f}{dx}\Big|_{\bar{x},\bar{u}} = \begin{bmatrix} \frac{\partial f_1}{dx_1} & \frac{\partial f_1}{dx_2} & \cdots \\ \frac{\partial f_2}{dx_1} & \frac{\partial f_2}{dx_2} & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix}_{\bar{x},\bar{u}} \tag{1}$$

$$B = \frac{\partial f}{du}\Big|_{\bar{x},\bar{u}} == \begin{bmatrix} \frac{\partial f_1}{du_1} & \frac{\partial f_1}{du_2} & \cdots \\ \frac{\partial f_2}{du_1} & \frac{\partial f_2}{du_2} & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix}_{\bar{x},\bar{u}} \tag{2}$$

$A \in \mathcal{R}^{n \cdot n}$, $B \in \mathcal{R}^{n \cdot m}$

This means that we can write:

$$\dot{x} = Ax + Bu$$

- Note that when we linearise, x and u are the deviations w.r.t. the equilibrium.
- We can also calculate the Jacobian for the output equation which leads the output equation is linear form:

$$y = Cx + Du$$

Let's go back to our car example.

We need to compute the Jacobian matrix of the system evaluated at the operating point $(\bar{x}, \bar{u})$, where $\bar{x}$ and $\bar{u}$ are the constant equilibrium values of x and u, respectively.

$$\dot{x} = f(x,u) = \begin{cases} \dot{x_1} & = x_2 \\ \dot{x_2} & = -\frac{\alpha}{m}|x_2|x_2 - \frac{\beta}{m}x_2 + \frac{\gamma}{m}u - gsin(\theta) \end{cases} \tag{3}$$

$$A = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{\alpha}{m}|\bar{x}_2| - \frac{\beta}{m} \end{bmatrix} \tag{4}$$

$$B = \frac{df}{\partial u} = \begin{bmatrix} \frac{\partial f_1}{\partial u} & \frac{\partial f_1}{\partial \theta} \\ \frac{\partial f_2}{\partial u} & \frac{\partial f_2}{\partial \theta} \end{bmatrix} \begin{bmatrix} 0 & 0 \\ \frac{\gamma}{m} & -gcos(\theta) \end{bmatrix} \tag{5}$$

- and if we choose input $\theta = 0$ and $u = 0$
- then an equilibrium is $\forall x_1, x_2 = 0$ (x_1 does not affect the result):

$$A = \frac{\partial f}{\partial x}\Big|_{\substack{x_1=0 \\ x_2=0}} = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{\beta}{m} \end{bmatrix} \tag{6}$$

$$B = \frac{\partial f}{\partial u}\Big|_{\substack{u=0 \\ \theta=0}} = \begin{bmatrix} 0 & 0 \\ \frac{\gamma}{m} & -g \end{bmatrix} \tag{7}$$

and bringing everything together for our system:

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$$

$$y = [0\ 1]\mathbf{x}$$

This is also something we can code up!

In [ ]:

```
#| export
class LinearCar:
    _g = 9.8

    def __init__(self, x0, params):
        self._x_1 = x0[0] # position (along the road)
        self._x_2 = x0[1] # velocity
        self._m, self._alpha, self._beta, self._gamma = params

    def step(self, dt, u, theta):
        # u: gas pedal
        # theta: slope of the road
        self._theta = theta
        A = np.array([[0, 1], [0, -self._beta/self._m]])
        B = np.array([[0, 0], [self._gamma/self._m, -LinearCar._g]])

        x = np.array([[self._x_1],[self._x_2]])
        U = np.array([[u],[theta]])

        self._x_1 = (self._x_1 + dt*(A[0,np.newaxis,:].dot(x) + B[0,np.newaxis,:].dot(U))).item()
        self._x_2 = (self._x_2 + dt*(A[1,np.newaxis,:].dot(x) + B[1,np.newaxis,:].dot(U))).item()

    def speedometer(self):
        v = self._x_2
        return (v,)

    def sensor_i(self): # Utility function to simplify plotting
        # Rotation matrix to get back to the inertial frame..
        R = np.array(((np.cos(self._theta), -np.sin(self._theta)),
                      (np.sin(self._theta), np.cos(self._theta))))
        x_i, y_i = R.dot(np.array([[self._x_1],[0]]))
        v = self._x_2
        return (x_i, y_i, v)
```

Comments

- Multiple models can be used to represent a physical system
- There are a number of parameters in our models (m, α, β, γ)
- Their value depend on the specific car you are modeling
- It might not be easy to have good values for these parameters

The simulation problem

- Predict how your outputs change given a known set of inputs and the mathematical model of the system
- Usually a lot of design time is spent in this stage
- It can be difficult to figure out the set of inputs and their range to characterise the operational envelope of the system

You need to run a simulation to answer:

- Does my system model match my test data?
- Will my system work in all operating environments?
- What happens when...
- Simulations can also be used to limit the number of field tests (which are usually expensive or might let you avoid dangerous maneouvers)

Let's go back to our Python car. We can now easily simulate it.

In [ ]:
```
# We assume we have identified the model parameters
m = 10
alpha = 1
beta = 1
gamma = 1
params = (m, alpha, beta, gamma)

# We select the car initial conditions (position and velocity)
x_0 = (0,0)

# We create our car
car = Car(x_0, params)
```

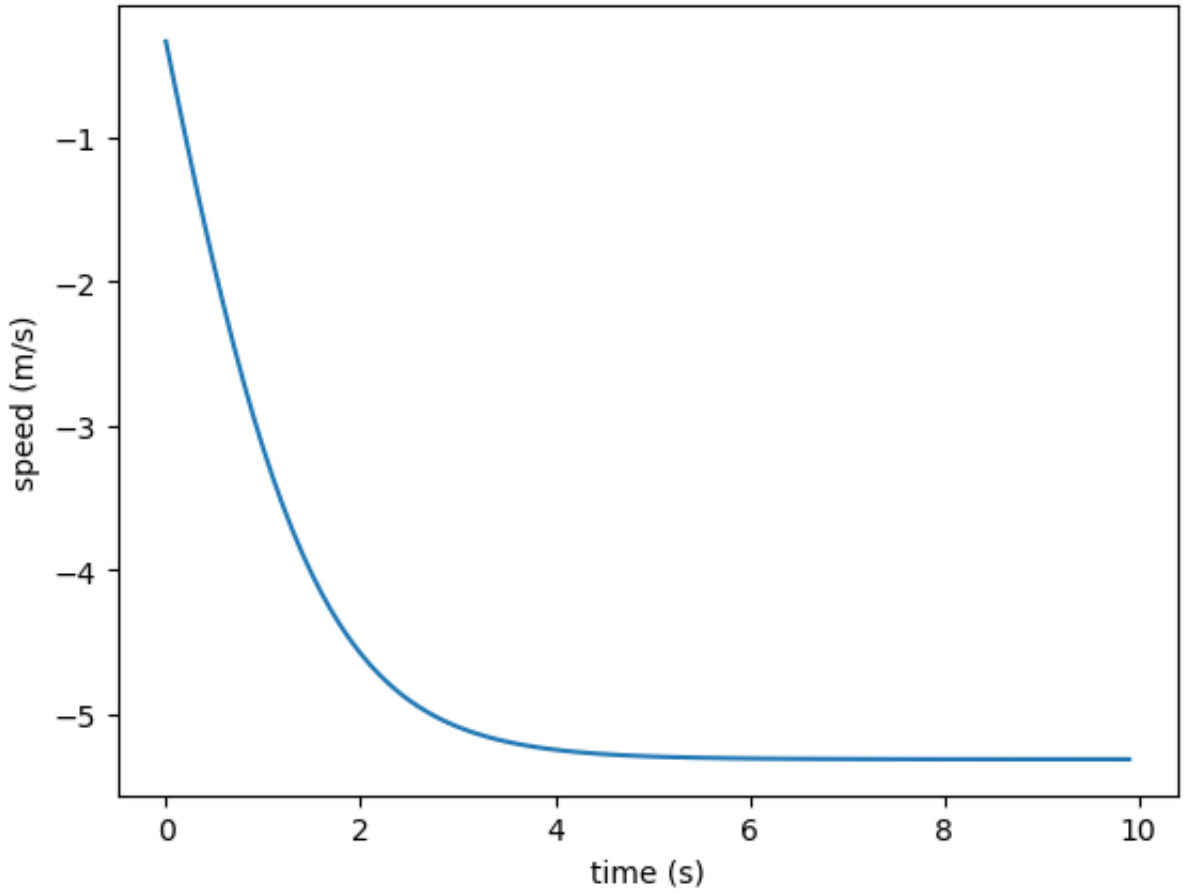
In [ ]:
```
# We define out inputs:
theta = np.radians(20) # Disturbance
u = 0                  # Input

# And finally we define the simulation parameters
t0, tf, dt = 0, 10, 0.1 # time

position = []
velocity = []
time = []
for t in np.arange(t0, tf, dt):
    print('.', end ='')
    car.step(dt, u, theta)
    x, y, v = car.sensor_i()
    position.append((x,y))
    velocity.append(v)
    time.append(t)

print('\n[DONE] Simulation is now complete.')
```

```
....................................................................................................
[DONE] Simulation is now complete.
```

Done! Now we can see what happened..

For example we can plot the values of a few interesting variables.

In [ ]:
```
fig, ax = plt.subplots();

plt.plot(time, velocity)
plt.xlabel('time (s)')
plt.ylabel('speed (m/s)');
```

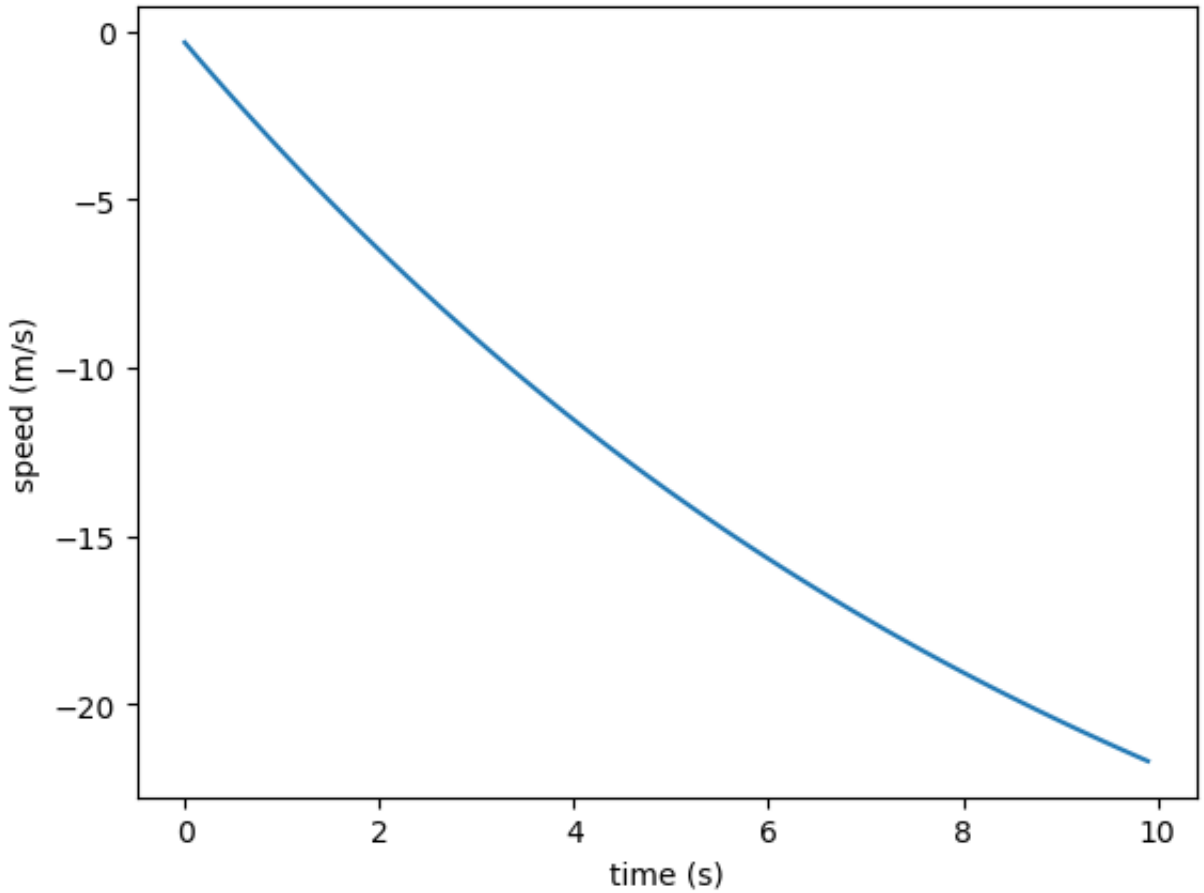


What happens if we simulate our LinearCar?

In [ ]:
```
# We select the car initial conditions (position and velocity)
x_0 = (0,0)

# We create our car
# HERE WE USE LINEARCAR!
car = LinearCar(x_0, params)

# We define out inputs:
theta = np.radians(20) # disturbance
u = 0                  # Input

# And finally we define the simulation parameters
t0, tf, dt = 0, 10, 0.1 # time

lin_position = []
lin_velocity = []
time = []
for t in np.arange(t0, tf, dt):
    print('.', end ='')
    car.step(dt, u, theta)
    x, y, v = car.sensor_i()
    lin_position.append((x,y)), lin_velocity.append(v)
    time.append(t)

print('\n[DONE] Simulation is now complete.')
```

```
....................................................................................................
[DONE] Simulation is now complete.
```

In [ ]:
```
fig, ax = plt.subplots();

plt.plot(time, lin_velocity)
plt.xlabel('time (s)')
plt.ylabel('speed (m/s)');
```



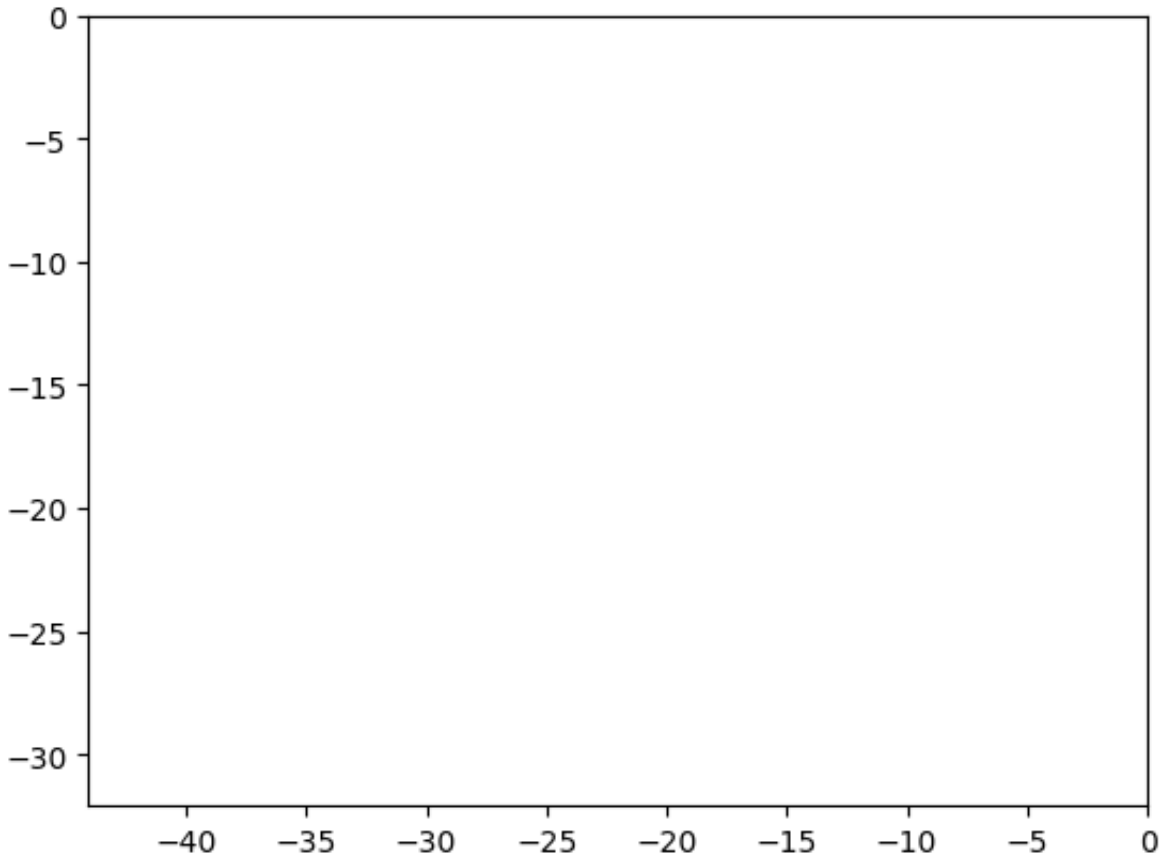
Comments

- Multiple models can be used to represent a physical system
- The complexity of the model depends on the objectives
- Different models might lead to different results

More complex representation

- Our initial simulation was relatively simple
- We can have much more complex representations of our simulation

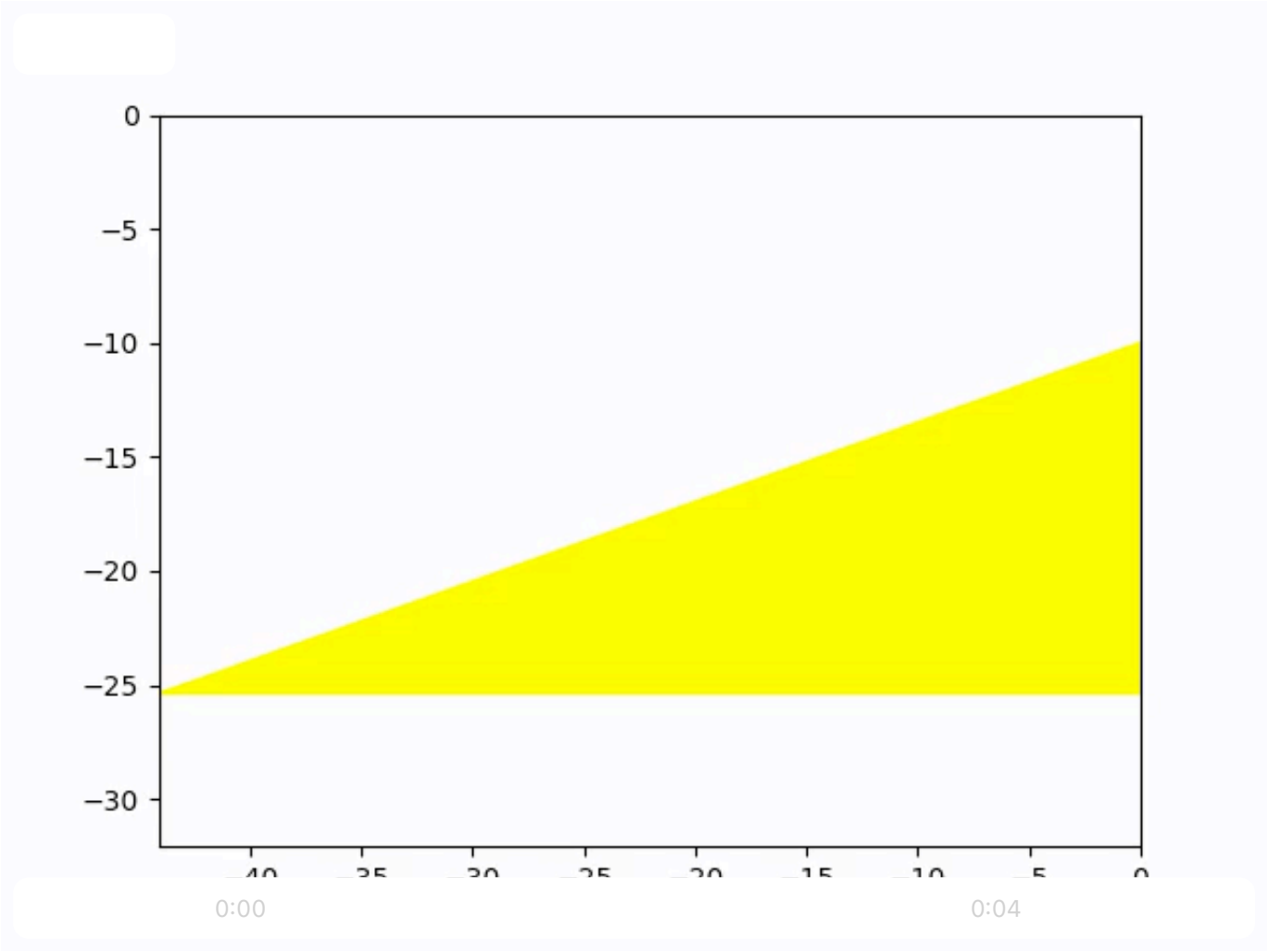
In [ ]:
```
# First set up the figure, the axis, and the plot elements we want to animate
fig, ax = plt.subplots();

ax.set_xlim((min(position)[0], max(position)[0]))
ax.set_ylim((min(position)[1]*2, max(position)[1]))
line, = ax.plot([], [], lw=4);
```



In [ ]:

```
# draw terrain
def terrain(theta_rad, x_0, x_range):
    y_range = theta*(x_range-x_0[0])+x_0[1]-10
    close_triangle = (x_range[0], y_range[-1])
    x_range = np.append(x_range, close_triangle[0])
    y_range = np.append(y_range, close_triangle[1])
    X = np.matrix([x_range, y_range]).transpose()
    patch = plt.Polygon(X, color='yellow')
    return patch

x_range = np.linspace(int(position[0][0]), int(position[-1][0]), num=20)
patch = terrain(theta_rad=theta, x_0=x_0, x_range=x_range)

# initialization function: plot the background of each frame
def init():
    ax.add_patch(patch)
    return patch,

# animation function. This is called sequentially
def animate(i):
    #line.set_data(time[max(0,i-2):i], position[max(0,i-2):i])
    x_min, x_max = position[max(0,i-2)][0], position[i][0]
    y_min, y_max = position[max(0,i-2)][1], position[i][1]
    line.set_data([x_min, x_max], [y_min, y_max])
    return (line,)


# call the animator. blit=True means only re-draw the parts that have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=len(time), interval=40, blit=True,
                               repeat_delay=10000, repeat=True);

HTML(anim.to_html5_video())
```

Out[ ]:



The Control Problem

- Control theory gives you the tools needed to answer this question.
- Without control theory, the designer is relegated to choosing a control system through trial and error.

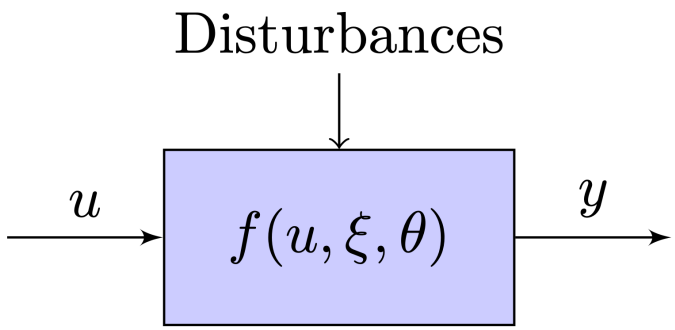
- How can I get my system to meet my performance requirements?
- How can I automate a process that currently involves humans in the loop?
- How can my system operate in a dynamic and noisy environment?

Why do we need feedback control?

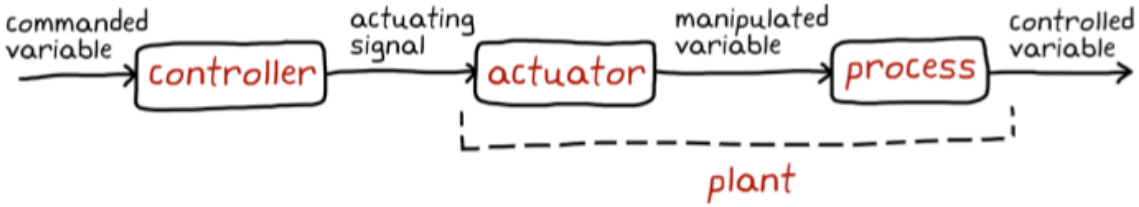
- We know the model of the system
- We know how we want the system outputs to behave
- We can determine the appropriate inputs through various control methods.

Control problem - how do we generate the appropriate system input that will produce the desired output?

Open-loop control



- Now the box isn't just any system, it's specifically a system that we want to control.
- We call the system that is being controlled as the **process** (or controlled system).
- The **inputs** into the process are variables that we have access to and can change based on whichever control scheme we choose (manipulated variables).
- Actuators manipulate these variables. An actuator is a generic term that refers to a device or motor that is responsible for controlling a system.
- The actuators are driven by an actuating signal that is generated by the controller.



This type of control system is referred to as open-loop since the inputs into the controller are not fed back from the output of the process.

- Open loop control is reserved for simple processes that have well-defined input to output behaviors

Some examples:

- Dishwasher
- Lawn sprinklers

Feedback-control

For any arbitrary process, an open-loop control system is typically not sufficient.

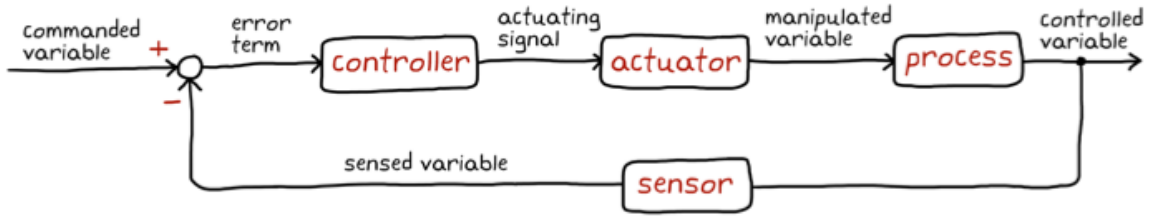
This is because:

- there are disturbances that affect your system that are random by nature and beyond your control;
- the process itself might have variations (e.g., variation of resistence due to temperature)

Since an open-loop control has no knowledge of the process output has no way of responding to these variations.

So what can we do about this?

- We add feedback to our system!
- We accept the fact that disturbances and process variations are going to influence the controlled variable.
- Instead of living with the resulting error, we add a sensor that will measure the controlled variable and pass it along to our controller.



A feedback control system is able to react to changes to the controlled variable automatically by constantly driving the error term to zero.

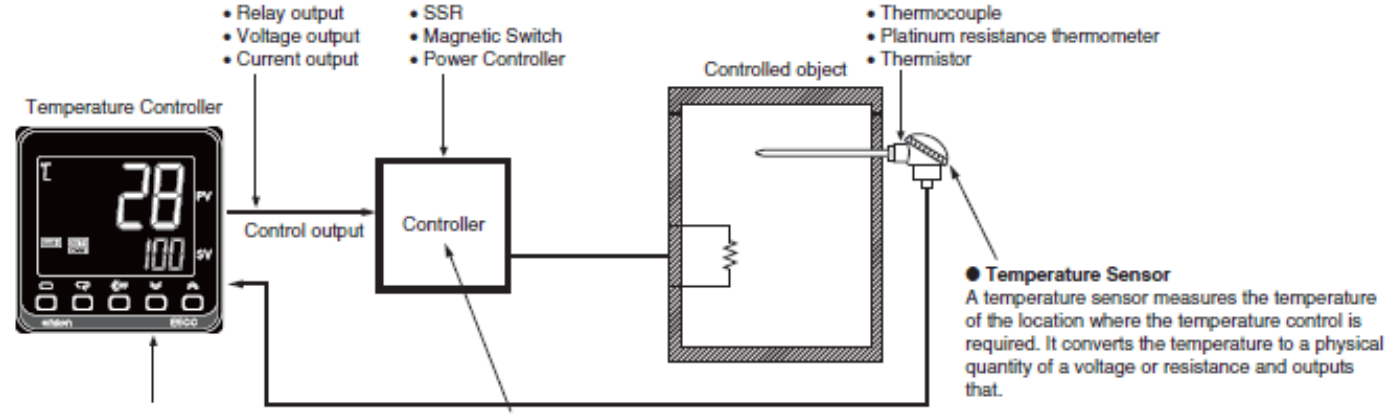
- The feedback structure is very powerful and robust
- With the addition of the feedback structure we also have new problems
- We need to think about:
    - the accuracy of the controlled variable at steady state,
    - the speed with which the system can respond to changes and reject disturbances,
    - the stability of the system as a whole.
- We have also added sensors, and they have noise and other inaccuracies that get injected into our loop and affect the performance.
    - We can add redundant sensors (so that we can measure different state variables)
    - We filter the output of the sensors to reduce the noise
    - We fuse them together to create a more accurate estimate of the true state.

What is a control system?

- A control system is a mechanism that alters the behavior (or the future state) of a system;
- The future behavior of the system must tend towards a state that is desired.
- This means that you have to know what you want your system to do and then design your control system to generate that desired outcome.

A Simple Example

simulation of a basic temperature control system using a simple regulator



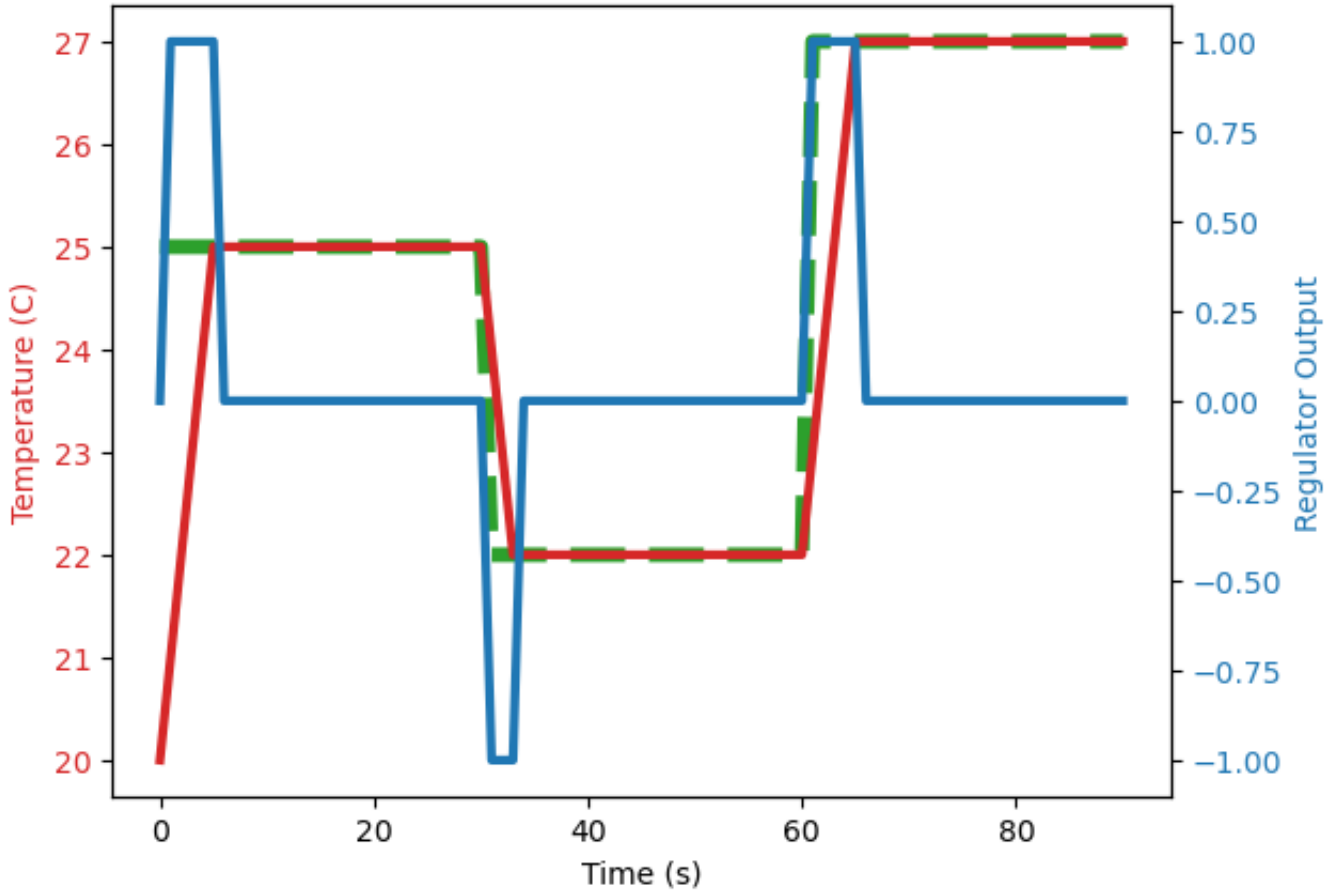
In [ ]:

```python
import time
import matplotlib.pyplot as plt

# Define the regulator function
def temperature_controller(current_temp, setpoint):
    # Set the allowable error range
    error_range = 0.5

    # Calculate the error
    error = setpoint - current_temp

    # If the error is within the allowable range, do nothing
    if abs(error) <= error_range: return 0.0

    # If the error is positive, turn on the heater
    elif error > 0: return 1.0

    # If the error is negative, turn on the cooler
    else: return -1.0
```

In [ ]:

```
# Set the simulation parameters
initial_temp = 20.0
setpoints = [25.0, 22.0, 27.0]
duration = 30.0  # seconds for each setpoint

# Initialize the temperature and time lists
temperature = [initial_temp]
time_list = [0.0]
setpoint_list = [setpoints[0]]

# Initialize the regulator output list
regulator_output = [0.0]

# Simulate the temperature control system
for i, setpoint in enumerate(setpoints):
    t = 0.0
    setpoint = setpoints[i]
    while t < duration:
        # Measure the current temperature
        current_temp = temperature[-1]
        # Use the regulator to calculate the control output
        control_output = temperature_controller(current_temp, setpoint)
        # Apply the control output to the temperature
        temperature.append(current_temp + control_output)
        # Add the control output to the regulator output list
        regulator_output.append(control_output)
        # Add the current setpoint to the setpoint list
        setpoint_list.append(setpoint)
        # Update the time
        t += 1.0
        # Add the time to the time list
        time_list.append(i * duration + t)
```

In [ ]:
```
# Plot the results
fig, ax1 = plt.subplots()
color = 'tab:red'
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Temperature (C)', color=color)
ax1.plot(time_list, setpoint_list, color='tab:green', linestyle='--', linewidth=5)
ax1.plot(time_list, temperature, color=color, linewidth=3)
ax1.tick_params(axis='y', labelcolor=color)
ax2 = ax1.twinx()
color = 'tab:blue'
ax2.set_ylabel('Regulator Output', color=color)
ax2.plot(time_list, regulator_output, color=color, linewidth=3)
ax2.tick_params(axis='y', labelcolor=color)
```



Linear Time Invariant (LTI) Systems

General form of a system:

$$\begin{cases} \dot{x}(t) = f(x(t), u(t), t) \\ y(t) = g(x(t), u(t), t) \end{cases}$$

When f and g are linear in x and u then the system can be written in the form:

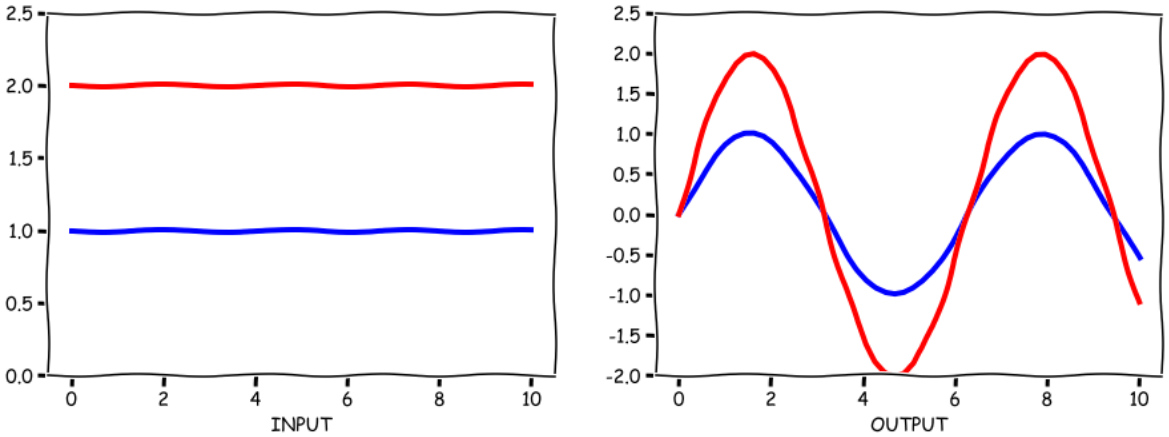
$$\begin{cases} \dot{x}(t) = A(t)x(t) + B(t)u(t) \\ y(t) = C(t)x(t) + D(t)u(t) \end{cases}$$

- The system is said *linear* in this case,
- Matrices $A(t), B(t), C(t), D(t)$ can be in general function of time,
- If they are not, we talk about **Linear Time-Invariant Systems**

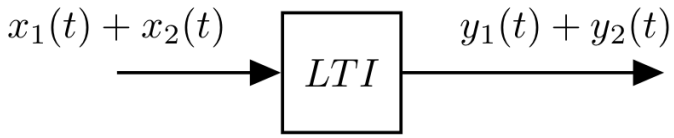
$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t) + Du(t) \end{cases}$$

All LTI systems have the following defining properties:

- Homogeneity:
  - If you scale the input $u(t)$ then the output will be scaled by the same factor:
  - $au(t) \Rightarrow ay(t)$ (e.g. if you double the input, the output would also double).
  - ex. step input of amplitude 1 and 2:



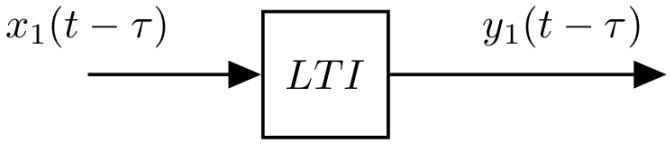
- Superposition (Additivity)
  - Suppose that:
    - input $x_1(t)$ produces output $y_1(t)$
    - input $x_2(t)$ produces output $y_2(t)$
  - If you add two inputs together, then the output is the superposition of the two separate outputs, i.e. the sum of the individual outputs:



Sometimes, these two properties are stated together: "if the input is scaled and summed, then the output will also be scaled and summed by the same amount".

- A system that respect these two property is a *linear system*

- Time Invariance
  - The system behaves the same regardless of when the action takes place
  - Formally there is no explicit dependency of time in the equations
  - The same input translated in time produces the same output also translated in time:
  - An input of $x(t-\tau)$ produces an output of $y(t-\tau)$.



- These conditions are very restritive and practically no real world system meets them.
- However, linear systems can be solved!
- Wide range of systems can be approximated accurately by an LTI model

---

"Linear systems are important because we can solve them." — Richard Feynman.

Impulse Response

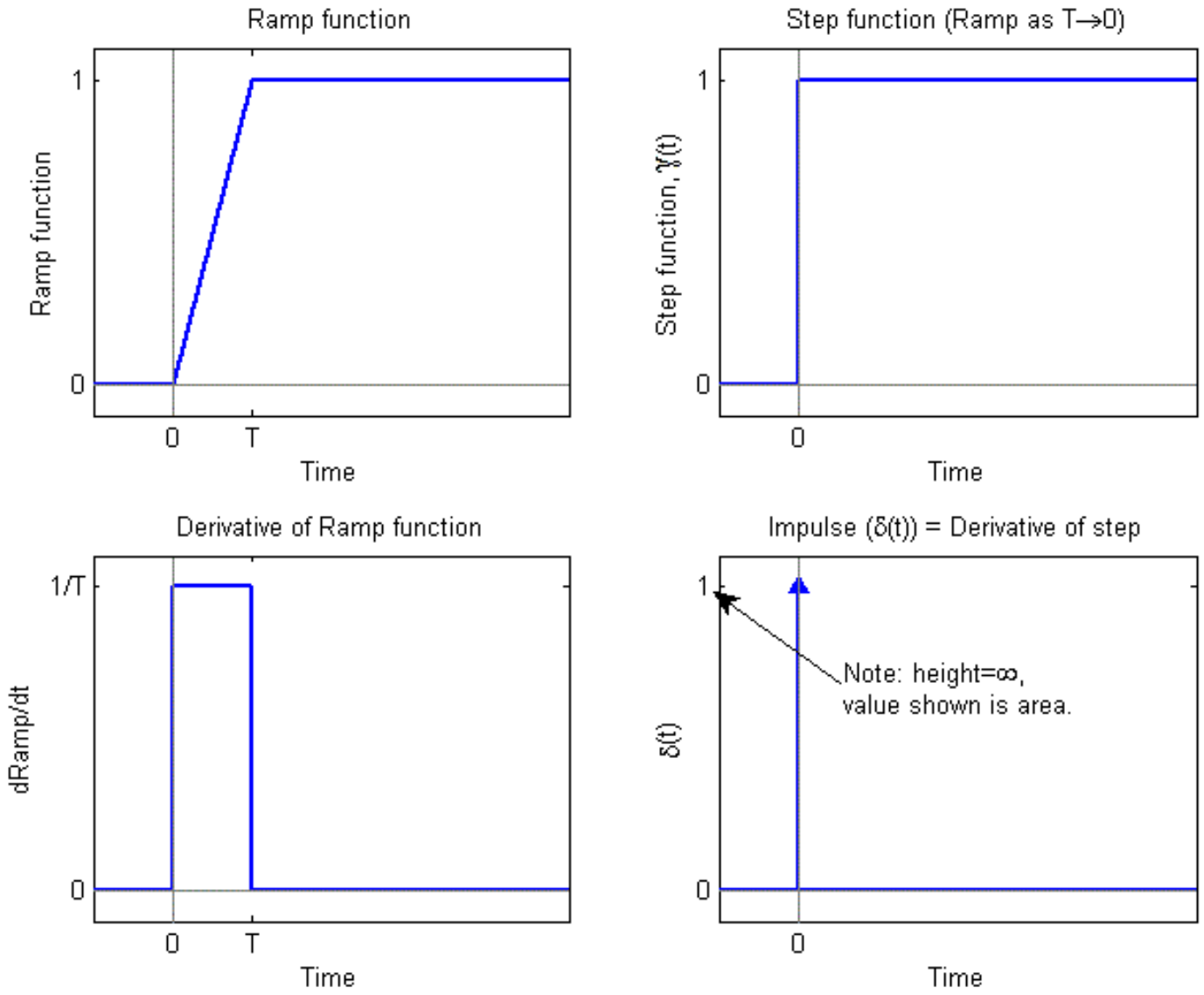
- LTI systems can be characterised by their response to an impulse function
- This is the output of the system when presented with a brief input signal

More formally, somewhat: The impulse response is the output of a system when an input that is an impulse function is applied to it. An impulse function is a theoretical input signal that is very short in duration and has a very large magnitude, but its area under the curve (integral) is equal to 1.

- Also known as Dirac delta function

- The impulse response of a system can be used to determine how the system will respond to other inputs.

An ideal impulse function is a function that is zero everywhere but at the origin, where it is infinitely high.



Consider first the ramp function shown in the upper left. It is zero for $t < 0$ and one for $t > T$, and goes linearly from 0 to 1 as time goes from 0 to T.

If we let $T \to 0$, we get a unit step function, $\gamma(t)$ (upper right).

If we take the derivative of our ramp function (lower left), we get a rectangular pulse with height 1/T (the slope of the line) and width

T.
This rectangular pulse has area (height·width) of one.
If we take the limit as $T \to 0$, we get a pulse of infinite height, zero width, but still with an area of one; this is the unit impulse and we represent it by $\delta(t)$.
Since we can't show the height of the impulse on our graph, we use the vertical axis to show the area. The unit impulse has area=1, so that is the shown height.

Let's see what happens when we apply an impulse to our car.

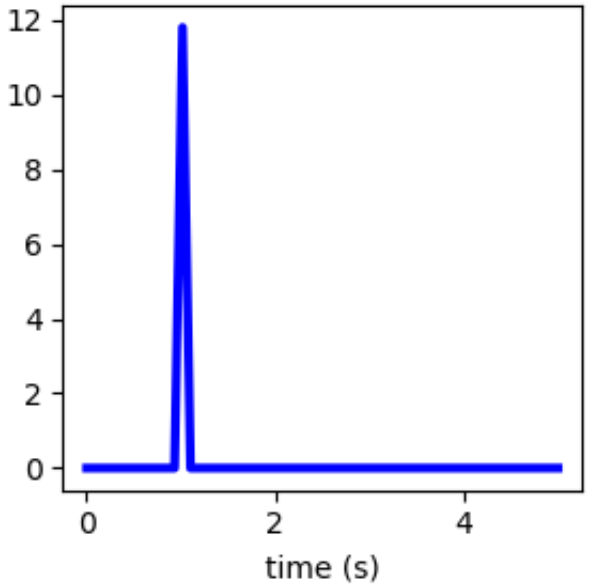
...Before we do that, we need to define an impulse function in Python.

...and before we do that, let's note that in the real world, an impulse function is a pulse that is much shorter than the time response of the system (sometime also called Dirac pulse).

In [ ]:
```python
#| export
def step(t, step_time=0):
    """Heaviside step function"""
    return 1 * (t >= step_time)

def delta(t, delta_t=0, eps=None):
    """
    Dirac Pulse

    The function delta computes an approximation of the impulse function (also called Dirac delta function)
    over a given time range t.
    The impulse function is a theoretical function that is zero everywhere except at t=0, where it is infinite.
    In practice, we cannot generate an infinite impulse, so we use a finite approximation.

    The function takes three arguments:
    t:       a scalar or an array of time values at which to compute the function.
    delta_t: the time at which the impulse occurs. This is an optional argument with a default value of 0, which
    eps:     the width of the pulse used to approximate the impulse.
             This is an optional argument with a default value of None,
             which means the function will try to estimate it automatically based on the time values in t.
             If eps is not defined and t has more than one element,
             eps is estimated as the difference between the first two elements of t.
             Otherwise, eps is set to the value of the eps argument.

    Computation of the impulse approximation:
        It uses the step function to create a pulse that has a width of eps centered at delta_t,
        and then subtracts two of these pulses to create a pulse that approximates the impulse function.

    The function returns the computed pulse as an array of the same shape as t.
    """
    if np.isscalar(t) and eps is None:
        raise Exception('eps must be defined for scalar values.')
    if eps is None and len(t) > 1:
        _eps=t[1]-t[0]
    else:
        _eps = eps
    return 1/_eps*(step(t, delta_t-_eps/2) - step(t, delta_t+_eps/2)) # area 1
```

And here is what it looks like:

In [ ]:
```
fig, ax = plt.subplots(1, 1, figsize=(3, 3))

t = np.linspace(0, 5, 60)
ax.plot(t, delta(t, delta_t=1), linewidth=3, color='blue')
ax.set_xlabel('time (s)')
fig.tight_layout()
```

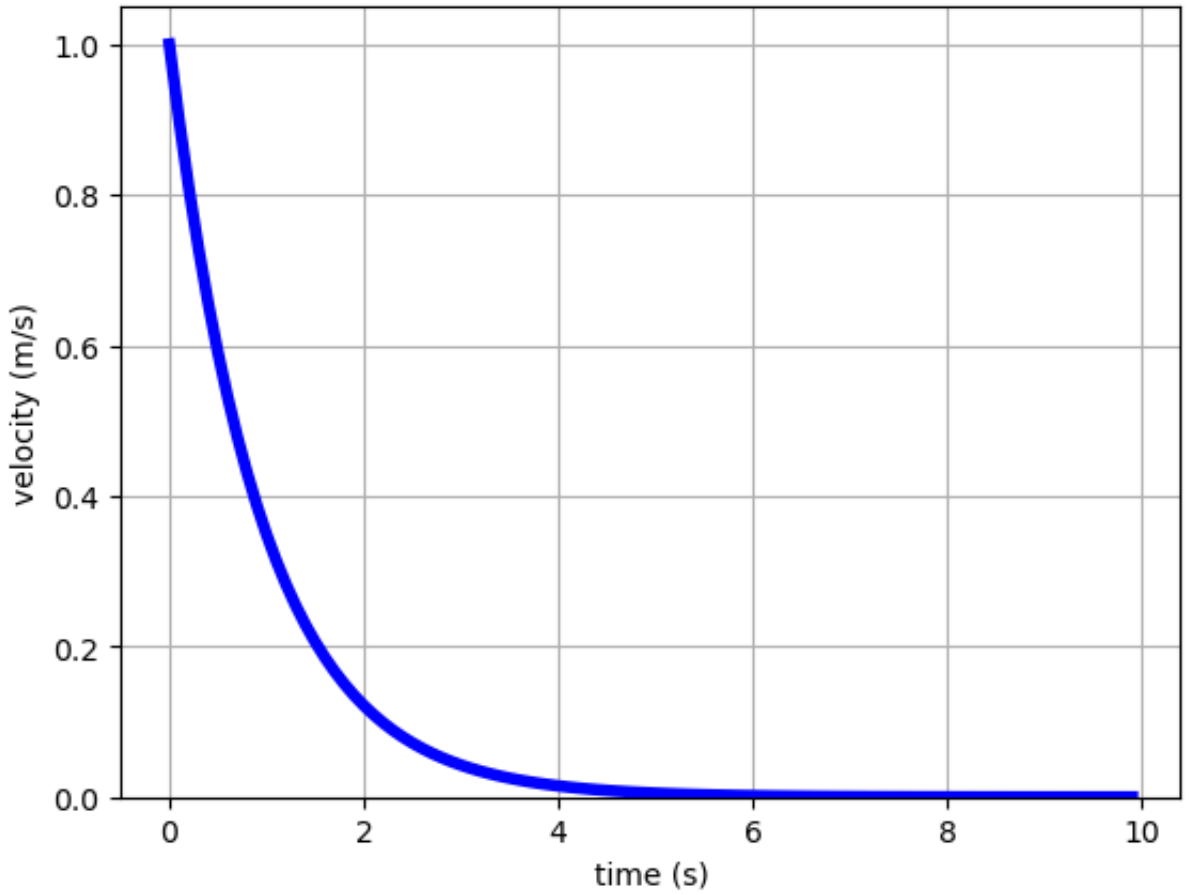


Now we can apply this impulse as input to our car.

In [ ]:
```
# We assume we have identified the model parameters
m = 10
alpha = 1 # it is not used in the linear system
beta = 10 # friction
gamma = 1 # coeff. applied to the input.
params = (m, alpha, beta, gamma)

# We select the car initial conditions (position and velocity)
x_0 = (0,0)

# We create our car
car = LinearCar(x_0, params)

theta = np.radians(0) # disturbance (deg) - we can also try np.radians(20)

# And finally we define the simulation parameters
t0, tf, dt = 0, 10, 0.1 # time

position = []
velocity = []
input_values = []
time = []
for t in np.arange(t0, tf, dt):
    print('.', end='')
    # HERE, we apply the impulse
    u = delta(t, delta_t=0, eps=0.01)
    car.step(dt, u, theta)
    x, y, v = car.sensor_i()
    position.append((x,y)), velocity.append(v)
    time.append(t)
    input_values.append(u)

print('\n[DONE] Simulation is now complete')
```

```
....................................................................................
[DONE] Simulation is now complete
```

In [ ]:
```
plt.plot(np.arange(t0, tf, dt), velocity, linewidth=4, color='blue', label='t_0')
plt.xlabel('time (s)'), plt.ylabel('velocity (m/s)');
plt.ylim(0,)
plt.grid();
```



Thanks to time invariance:

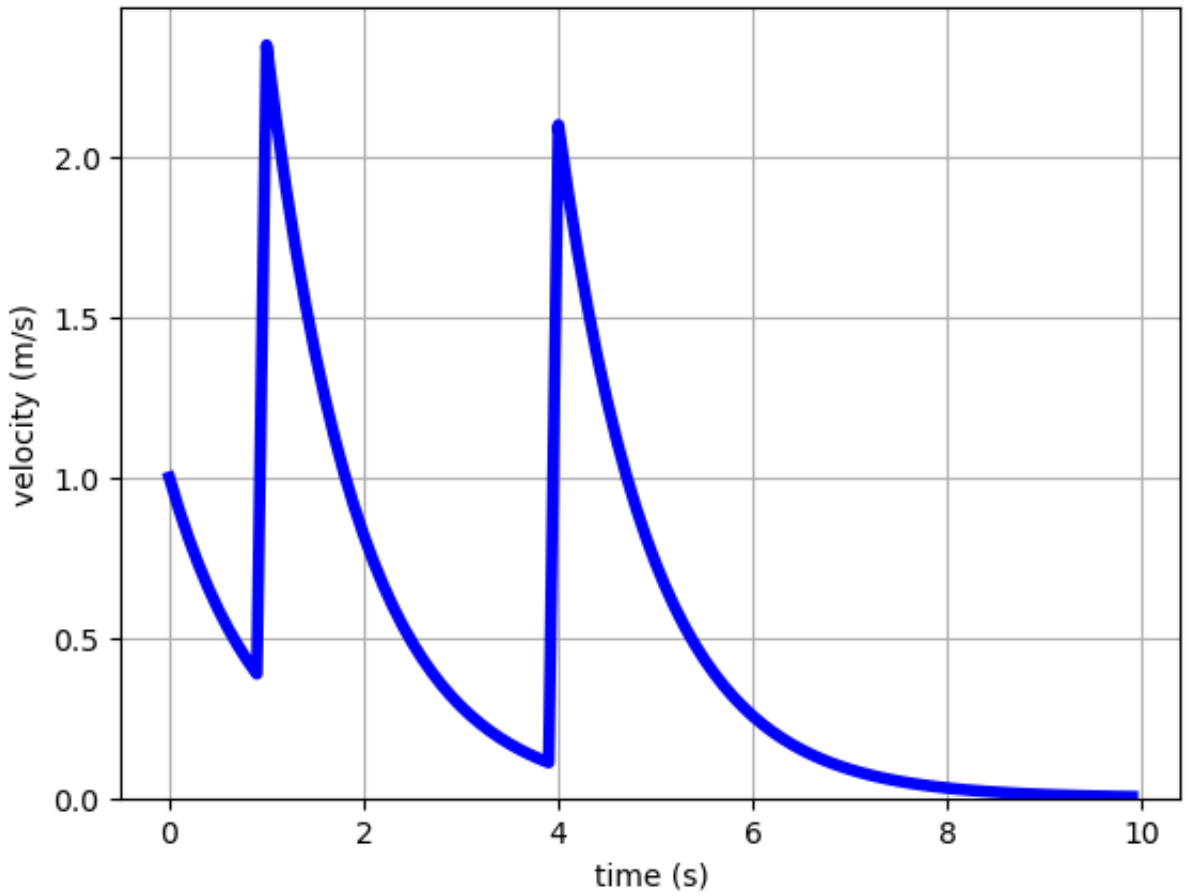
- if we have another impulse we can expect the same response at this second time,
- and if we hit twice as hard, then the response is twice as large

In [ ]:
```
def multiple_impulses(t):
    u1 = 1*delta(t, delta_t=0, eps=0.01)
    u2 = 2*delta(t, delta_t=1, eps=0.01)
    u3 = 2*delta(t, delta_t=4, eps=0.01)

    return u1 + u2 + u3
```

In [ ]:
```
car = LinearCar(x0=(0,0), params=params)

position = []
velocity = []
input_values = []
time = []
for t in np.arange(t0, tf, dt):
    print('.', end='')
    # HERE, we apply the impulse
    u = multiple_impulses(t)

    car.step(dt, u, theta)
    x, y, v = car.sensor_i()
    position.append((x,y)), velocity.append(v)
    time.append(t)
    input_values.append(u)

print('\n[DONE] Simulation is now complete')
```

```
.....................................................................................
[DONE] Simulation is now complete
```

In [ ]:
```
plt.plot(np.arange(t0, tf, dt), velocity, linewidth=4, color='blue', label='t_0')
plt.xlabel('time (s)'), plt.ylabel('velocity (m/s)');
plt.ylim(0,)
plt.grid();
```



Thanks to the superposition principle

- The full response is the summation of the signals.
- If we have a more complex signal (e.g. ramp) we can break it down to a sequence of impulses (the output is the response to each of the impulses)

In [ ]:
```
#| export
def ramp_as_impulses(t, time_vector):
    u = t*delta(time_vector, delta_t=t, eps=.01)
    return u
```

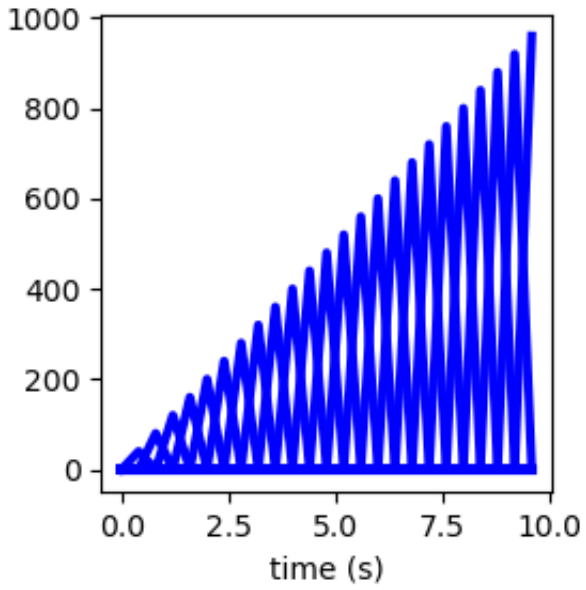
And we can verify that this is a ramp:

In [ ]:
```
input_u = []

for t in np.arange(t0, tf, .4):
    input_u.append(ramp_as_impulses(t, np.arange(t0, tf, .4)))
```

In [ ]:
```
len(input_u)
```

Out[ ]:
```
25
```

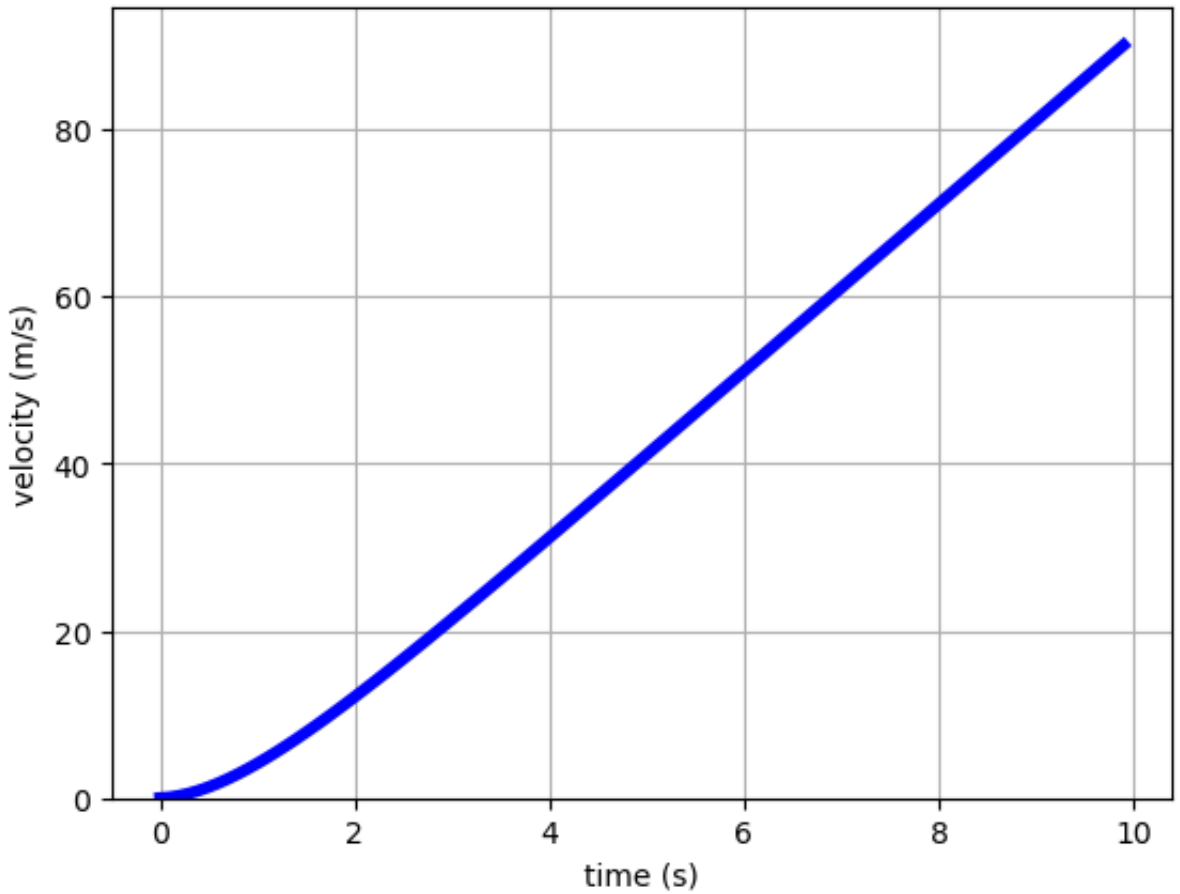
In [ ]:
```
fig, ax = plt.subplots(1, 1, figsize=(3, 3))

ax.plot(np.arange(t0, tf, .4), input_u, linewidth=3, color='blue')
ax.set_xlabel('time (s)')
fig.tight_layout()
```



In [ ]:
```
car = LinearCar(x0=(0,0), params=params)

position = []
velocity = []
input_values = []
time = []
for index, t in enumerate(np.arange(t0, tf, dt)):
    print('.', end='')
    # HERE, we apply the ramp as a sequence of impulses
    u = ramp_as_impulses(t, np.arange(t0, tf, dt))
    #print('time:', t, '-> index:', index, '->u[i]:', u[index])
    car.step(dt, u[index], theta)
    x, y, v = car.sensor_i()
    position.append((x,y)), velocity.append(v)
    time.append(t)
    input_values.append(u)


print('\n[DONE] Simulation is now complete')
```

```
.....................................................................................
[DONE] Simulation is now complete
```

In [ ]:
```
plt.plot(np.arange(t0, tf, dt), velocity, linewidth=4, color='blue', label='t_0')
plt.xlabel('time (s)'), plt.ylabel('velocity (m/s)');
plt.ylim(0,)
plt.grid();
```


velocity (m/s)
0
20
40
60
80
0
2
4
6
8
10
time (s)

- In reality, the impulses and responses will be infinitesimal
- In this case, the summation in the time domain is a convolution $u(t) \circledast H(t)$, where $H(t)$ is the impulse response of the system

$$(f \circledast g)(t) := \int_{-\infty}^{\infty} f(\tau) g(t-\tau)\, d\tau.$$

Convolution is a mathematical operation of two functions that describes how the shape of one is modified by the shape of the other. The result of this operation is the integral of their product, after folding one and shifted across the other.

- The impulse response of a system is important in control theory because it allows us to predict the behavior of the system under various inputs, without having to actually apply those inputs.
- We can simply convolve the input with the impulse response to obtain the output.
- This can be a difficult integration..

Fortunately we have a few aces up our sleeve!

- If we use the Laplace transform and move to the s-domain however, convolutions become multiplications, and things are simpler.
- As a control engineers we would like to design our systems to be as close as possible to be LTI, or so that the non LTI parts can be ignored.
- So that the standard LTI tools can be used.
    - Think of operating regions of a spring. What happens to the relation between force and distance when you pull it too much or compress it too tight vs what happens when you use it in its operating regime

In [ ]:
```
%matplotlib notebook

"""
Nice animation here: http://195.134.76.37/applets/AppletConvol/Appl_Convol2.html
"""

# Define time
t = np.linspace(0, 29, 30)
```

```
# Define the impulse response
impulse_response = np.zeros(30)
impulse_response[5:25] = 1

# Define the input signal
input_signal = np.zeros(30)
input_signal[10:20] = 1

# Calculate the convolution of the input signal and the impulse response
output_signal = np.convolve(input_signal, impulse_response, mode='full')

# Set up the plot
fig, axs = plt.subplots(3, 1, figsize=(8, 8))

# Plot the impulse response
axs[0].plot(t, impulse_response, color='g')
axs[0].set_xlim(0, len(impulse_response))
axs[0].set_ylim(0, 1.1)
axs[0].set_xlabel('Time')
axs[0].set_ylabel('Amplitude')
axs[0].set_title('Impulse Response')

# Plot the input signal
axs[1].plot(t, input_signal, color='r')
axs[1].set_xlim(0, len(input_signal))
axs[1].set_ylim(0, 1.1)
axs[1].set_xlabel('Time')
axs[1].set_ylabel('Amplitude')
axs[1].set_title('Input Signal')

# Plot the output signal (initially empty)
axs[2].plot(output_signal, color='k')
axs[2].set_xlim(0, len(output_signal))
axs[2].set_ylim(0, 11)
axs[2].set_xlabel('Time')
axs[2].set_ylabel('Amplitude')
axs[2].set_title('Output Signal')
line, = axs[2].plot([], [], 'k', label='Sliding window')

# Define the animation function
```

```python
def animate(i):
    # Calculate the convolution for the current time step
    print('hee')
    convolved_signal = np.convolve(input_signal, impulse_response[:i], mode='full')

    # Update the plot for the output signal
    line.set_data(range(len(convolved_signal)), convolved_signal)
    return (line,)

# Set up the animation
anim = FuncAnimation(fig, animate, frames=len(impulse_response)+1, interval=20)

# Show the plot
plt.show()
```