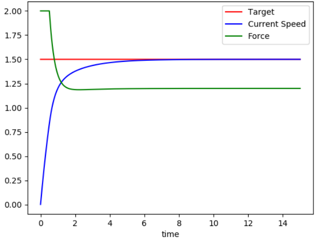
Controlling Position and Speed using Profiles

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
`santoro@dmi.unict.it`

Robotic Systems

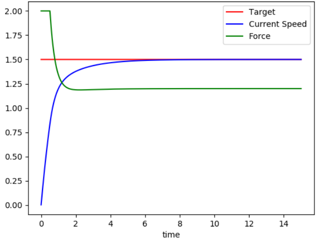
Speed Control



The Trend of the Speed Controller

- We know that, by modulating controller constants, we can change the system response, in terms of **setup-time**, i.e. the time required by the system to reach the target
- However... this setup-time is a **consequence** of constant tuning, and can be determined sperimentally by analysing the trend of the controlled variable
- It is **not** an **input design parameter**

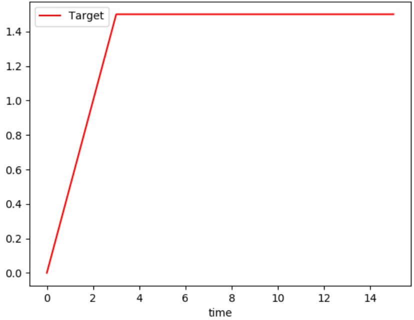
Speed Control



The Trend of the Speed Controller (2)

- We suddenly gave to the system a non-zero set-point (with an “high value”) when system is a “quiet state” and this is unrealistic
- Real systems instead feature an acceleration phase that then leads to the final speed
- Can we control also the acceleration and thus the **time** employed to reach the final speed?

Speed Control



The Trend of the Speed Controller (3)

- Can we control also the acceleration and thus the **time** employed to reach the final speed?
- Rather than give suddenly the final speed, let's **increase** the (set-point) target speed, starting from 0 up to the final value, according to a given **acceleration**

Speed Control with Acceleration

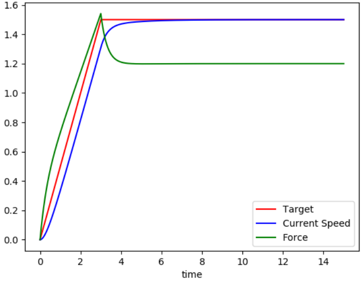
```
def __init__(self):
    ...
    self.final_target_speed = 1.5 # 1.5 m/s
    self.current_target_speed = 0
    self.acceleration = 0.5 # 0.5 m/s^2

def run(self):
    F = self.controller.evaluate(self.delta_t,
                self.current_target_speed, self.get_speed())
    self.cart.evaluate(self.delta_t, F)

    # now accelerate
    self.current_target_speed += self.acceleration * self.delta_t
    if self.current_target_speed > self.final_target_speed:
        # do not overcome the final speed
        self.current_target_speed = self.final_target_speed
```

Speed Control with Acceleration

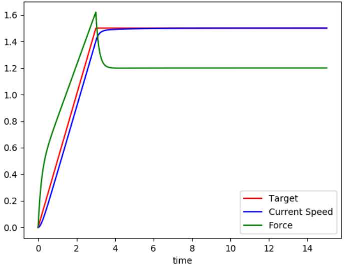
$K_P = 3, K_I = 2, SAT = 2\ N$



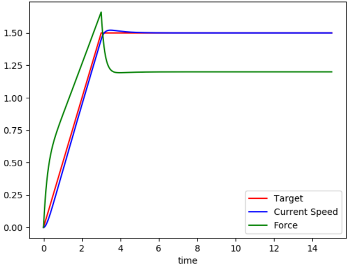
Let's tune constants

- Using the constants above (and considering a saturation value of 2 N) the trend of the real speed does not follow adequately the trend of the target
- Moreover, the system is not in saturation, so we can surely increase the constants

$K_P = 6, K_I = 4$

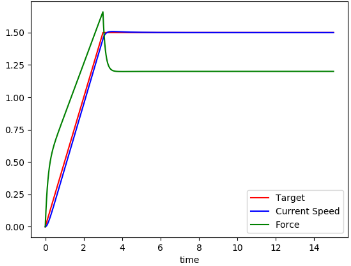


$K_P = 6, K_I = 8$

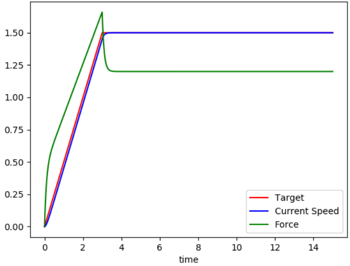


Cart Speed Control with Acceleration

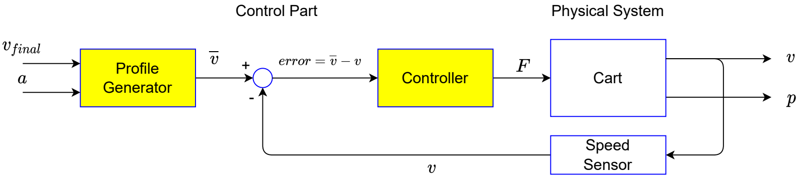
$K_P = 8, K_I = 8$



$K_P = 10, K_I = 8$



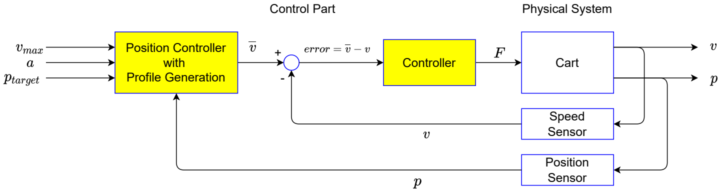
Speed Control with Acceleration



The final schema is made of:

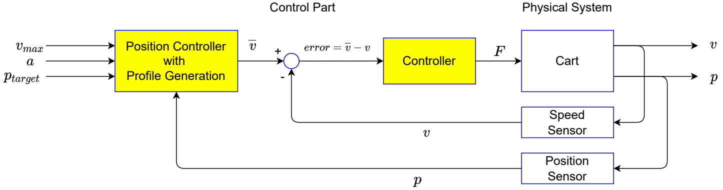
- A classical **speed (PI) controller** with saturation and anti-wind-up
- A **profile generator** that, according to the final value v_{final} and the acceleration a provides (for each time instant) the target speed to be reached in **that** time instant

A Step Further



- Now we have an important key: we discovered that we can modulate the speed set-point $\overline{v}$ of the controller and make the system follow it
- The way in which $\overline{v}$ is modulated depends of the specific application
- It can be reaching a final v according to a certain accelartion, or ...
- reaching a **target position** according to an acceleration (and deceleration) a and a maximum speed v_{max}

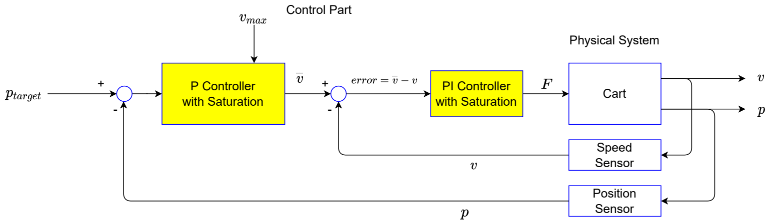
Cascade Controllers



- We can imagine two controllers in **cascade**, one driving the other one
- The **Position Controller** that, according to a certain algorithm, the target position p_{target}, the current position p and other parameters gives (outputs) the speed set-point $\overline{v}$
- The **Speed Controller** that provides the push needed to make the cart reaching the speed set-point instant by instant

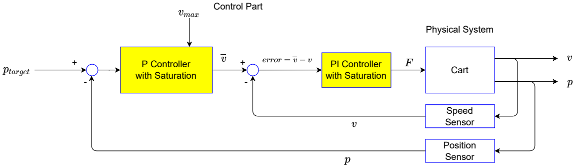
The “Simple” P-Controller

Position and Speed Control



- In this schema, the **Position controller** acts on the basis of the position error
- It is a **P-Controller** with **saturation** and generates a desired travel speed $\overline{v}$ proportional to the error but never greater than v_{max}
- The system first travels at the maximum speed and then reduces the speed proportionally as soon as the target position is approached

Position and Speed Control

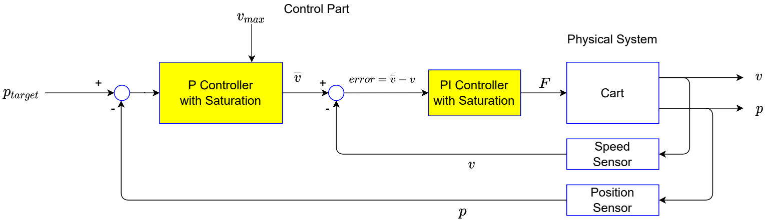


(see **test_speed_pi_control_cart_gui_plot.py**)

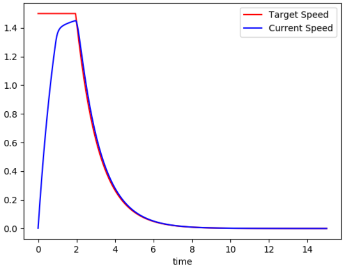
```
def __init__(self):
    ...
    self.speed_controller = PIDSat(10.0, 8.0, 0.0, 2.0, True)
    # Kp = 3, KI = 2, Sat = 2 N
    self.position_controller = PIDSat(0.8, 0.0, 0.0, 1.5)
    # Kp = 0.8, vmax = 1.5 m/s
    self.target_position = 4 # 4 m/s

def run(self):
   v_target = self.position_controller.evaluate(self.delta_t,
                                 self.target_position, self.get_pose())
   F = self.speed_controller.evaluate(self.delta_t,
                                 v_target, self.get_speed())
   self.cart.evaluate(self.delta_t, F)
   ...
```

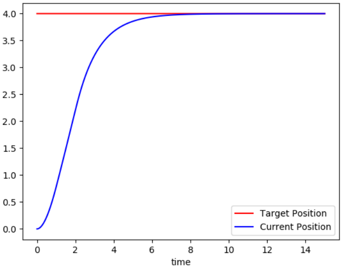
Cart Speed Control with Acceleration



Speed

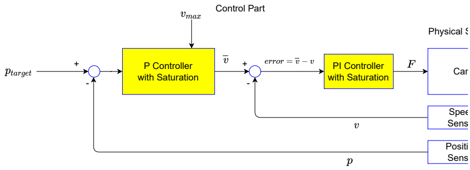


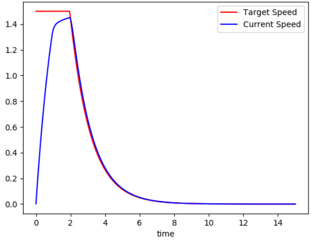
Position



The Speed Profile

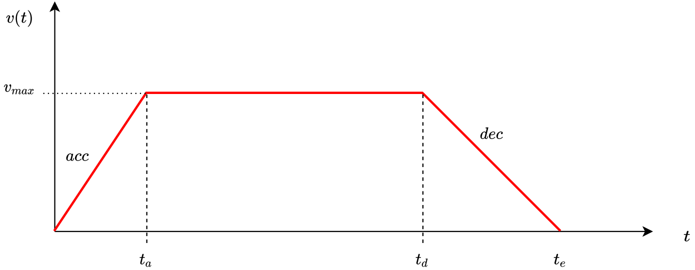
The Simple "P" Controller





- In the "simple-P" controller, the (real) speed shows a specific trend:
  - It has an initial acceleration phase
  - then there is a "cruise" phase at the maximum speed (saturation, v_{max}
  - And, when the P controller exits from saturation, the speed gradually decreases (deceleration phase)
- While the controller works (i.e. the target position is reached), we have **no control** over acceleration and deceleration: in some cases this is undesirable!

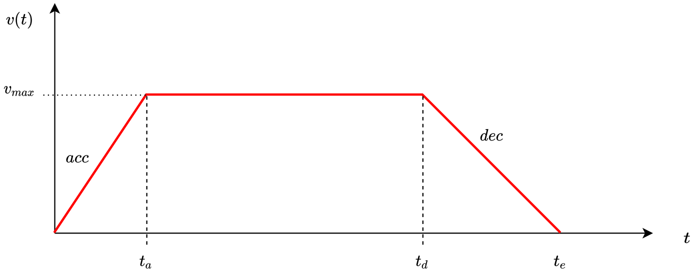
The Speed Profile



- Indeed, a more desirable situation is the one in which we can decide:
  - The final/target position p_{target}
  - The value of the acceleration *acc*
  - The maximum/cruise speed v_{max}
  - The value of decelration *dec* (that could be even equal to acceleration)
- In such a case, the aim of the controller is to ensure that when the deceleration phase ends the robot is **exactly in position** p_{target}

The Virtual Robot

The Virtual Robot



- Rather than dealing with the problem of "control", let us concentrate on how to create the profile above
- To this aim, let us consider an "ideal" (virtual) robot that has to travel a certain distance p_{target} by following that speed profile
- To model such a motion, we consider the cinematic equations related to uniform motion and uniformly accelerated motion

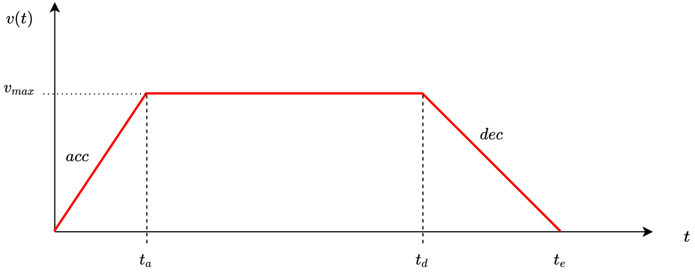
Uniformly Accelerated Motion

$$
\begin{aligned}
a(t) &= a \ \ (= const) \\
v(t) &= v(t_0) + a \cdot (t - t_0) \\
p(t) &= p(t_0) + v(t_0) \cdot (t - t_0) + \frac{1}{2} \cdot a \cdot (t - t_0)^2
\end{aligned}
$$

Uniform Motion

$$
\begin{aligned}
v(t) &= v \ \ (= const) \\
p(t) &= p(t_0) + v \cdot (t - t_0)
\end{aligned}
$$

The Virtual Robot



- We must simulate the motion of the ideal robot by applying the equation above
- However, we must identify **when to change the motion** (from acceleration to cruise, and from cruise to deceleration)
- In other words, we should determine the time instants t_a and t_d in which the regime changes
- This can be done by using the equations, however we must remember that we then act in a "discretized" world!!

The Virtual Robot

Let's implement the virtual robot

- We can write a class that receives the desired parameters of the motion and acts accordingly to the speed profile
- The class embeds, in its attributes, the current speed and position of the robot
- Moreover, we need to somehow encode the phase in which our motion is

```
class VirtualRobot:
   ACCEL = 0
   CRUISE = 1
   DECEL = 2
   TARGET = 3
   def __init__(self, _p_target, _vmax, _acc, _dec):
       self.p_target = _p_target
       self.vmax = _vmax
       self.accel = _acc
       self.decel = _dec
       self.v = 0 # current speed
       self.p = 0 # current position
       self.phase = VirtualRobot.ACCEL
```

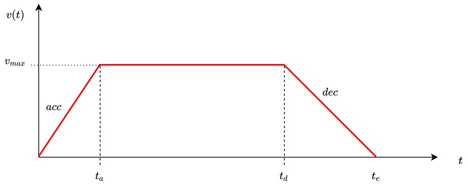
Let's implement the virtual robot

- In the **evaluate** method, let's implement the behavour of the motion
- acceleration and cruise phases are easy to implement, and also their transition can be easily idenfied
- but... when we should start the deceleration?

```
def evaluate(self, delta_t):
    if self.phase == VirtualRobot.ACCEL:
        self.p = self.p + self.v * delta_t \
                 + self.accel * delta_t * delta_t / 2
        self.v = self.v + self.accel * delta_t
        if self.v >= self.vmax:
            self.v = self.vmax
            self.phase = VirtualRobot.CRUISE

    elif self.phase == VirtualRobot.CRUISE:
        self.p = self.p + self.vmax * delta\_t
        if ?????:
            self.phase = VirtualRobot.DECEL
    ...
```

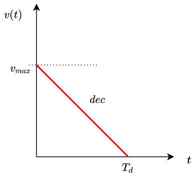
The Deceleration Distance



- Let's consider the final part of the motion, from t_d to the end t_e
- We start at speed v_{max}, at time t_d
- We end at speed 0, at time t_e
- Let us apply the formulae of the uniformly accelerated (decelerated) motion (let's suppose that dec is positive)

$$\begin{aligned}
v(t) &= v(t_0) + a \cdot (t - t_0) \\
v(t_e) &= v(t_d) - dec \cdot (t_e - t_d) \\
0 &= v_{max} - dec \cdot (t_e - t_d) \\
(t_e - t_d) &= \frac{v_{max}}{dec}
\end{aligned}$$

The Deceleration Distance



- Now let's everything but final part of the motion
- Its duration is $T_d = t_e - t_d = \frac{v_{max}}{dec}$
- Let's suppose that it starts at position 0 and ends a position D

$$
\begin{aligned}
p(t) &= p(t_0) + v(t_0) \cdot (t - t_0) + \frac{1}{2} \cdot a \cdot (t - t_0)^2 \\
D &= 0 + v_{max} \cdot T_d - \frac{1}{2} \cdot dec \cdot T_d^2 \\
D &= v_{max} \cdot \frac{v_{max}}{dec} - \frac{1}{2} \cdot dec \cdot \frac{v_{max}^2}{dec^2} \\
D &= \frac{1}{2} \cdot \frac{v_{max}^2}{dec}
\end{aligned}
$$

The Deceleration Distance

$$D = \frac{1}{2} \cdot \frac{v_{max}^2}{dec}$$

- We obtained the **deceleration distance**
- It is the **distance from the target** at which we must start the deceleration phase
- Therefore, if $p_{target} - p_{current} \leq D$, we are in the deceleration phase

```
class VirtualRobot:
    ...
    def __init__(self, _p_target, _vmax, _acc, _dec):
        ...
        self.decel_distance = 0.5 * _vmax * _vmax / _dec

    def evaluate(self, delta_t):
        ...
        elif self.phase == VirtualRobot.CRUISE:
            self.p = self.p + self.vmax * delta\_t
            if self.p_target - self.p <= self.decel_distance:
                self.phase = VirtualRobot.DECEL
        ...
```

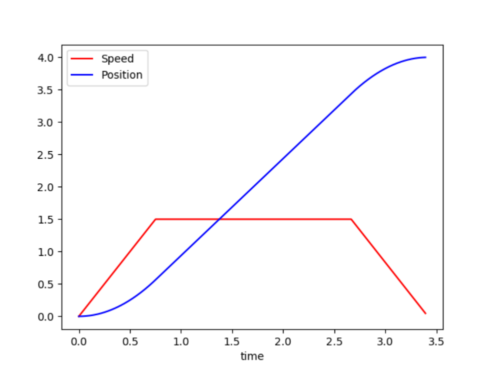
The Virtual Robot

- And finally let's implement the deceleration phase

```
def evaluate(self, delta_t):
    ...
    elif self.phase == VirtualRobot.DECEL:
        self.p = self.p + self.v * delta_t \
                 - self.decel * delta_t * delta_t / 2
        self.v = self.v - self.decel * delta_t
        if self.p >= self.p_target:
            self.v = 0
            self.p = self.p_target
            self.phase = VirtualRobot.TARGET
    ...
```

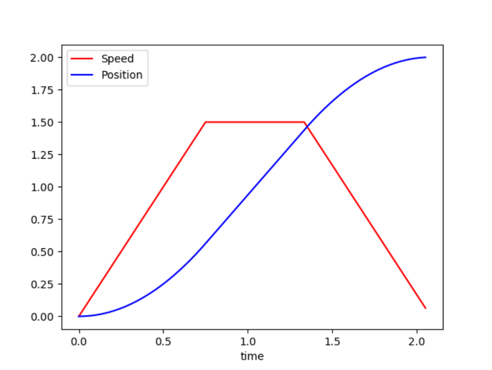
Testing the Code

```
rob =   VirtualRobot(   4, # distance 4 m
                        1.5, # max speed 1.5 m/s
                        2.0, # accel 2 m/s2
                        2.0) # decel 2 m/s2
```



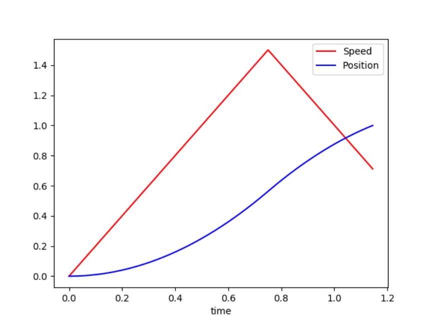
Testing the Code

```
rob =   VirtualRobot(   2, # distance 2 m
                        1.5, # max speed 1.5 m/s
                        2.0, # accel 2 m/s2
                        2.0) # decel 2 m/s2
```



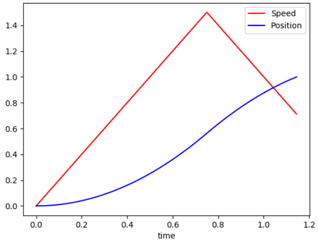
Phase Overappling

```
rob =   VirtualRobot(   1, # distance 2 m
                        1.5, # max speed 1.5 m/s
                        2.0, # accel 2 m/s2
                        2.0) # decel 2 m/s2
```



The target is reached but the final speed **is not** 0!!

Virtual Robot



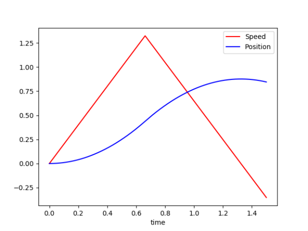
Phase Overlapping

- When the distance is too short, phases may overlap
- The deceleration distance is such that the deceleration phase should begin **before** the acceleration phase is ended
- So we should consider this particular case in our code

Testing the Code

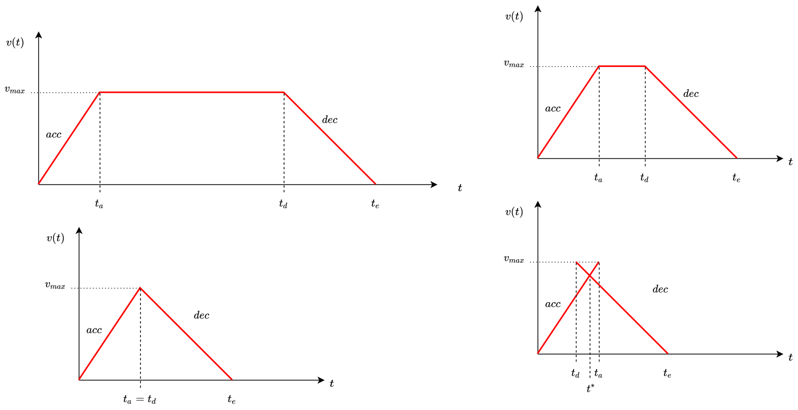
- At first sight, the code should be patched as follows:

```
def evaluate(self, delta_t):
    if self.phase == VirtualRobot.ACCEL:
        self.p = self.p + self.v * delta_t \
                 + self.accel * delta_t * delta_t / 2
        self.v = self.v + self.accel * delta_t
        if self.v >= self.vmax:
            self.v = self.vmax
            self.phase = VirtualRobot.CRUISE
        elif self.p_target - self.p <= self.decel_distance:
            self.phase = VirtualRobot.DECEL
```



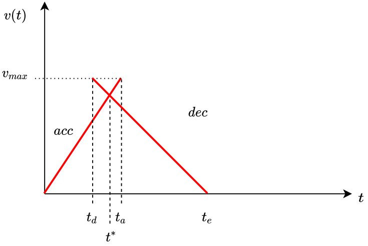
The target is never reached!! Why??

Speed Profile and Distance



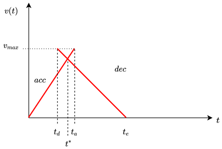
- As soon as the the target distance decreases, the cruise phase is shortened and the deceleration phase "approaches" the acceleration phase
- Until the acceleration and deceleration phases **overlap!**

Speed Profile and Distance



- In this case, the deceleration distance **is not** the one computed before
- But we must find the place in which the acceleration and deceleration lines **meet**

Speed Profile and Distance



Where do the acc and dec phases meet?

- Let's consider once again only the deceleration phase
- Let us suppose that, at a certain time instant, we are at a distance d from the target
- Here we will start travelling at a certain speed v_d and we will have the distance d to cover
- According to dec that distance will be covered in certain time t'
- We have:

$$d = 0 + v_d \cdot t' - \frac{1}{2} \cdot dec \cdot t'^2$$

Where do the acc and dec phases meet?

- We have:

$$d = 0 + v_d \cdot \Delta t' - \frac{1}{2} \cdot dec \cdot \Delta t'^2 \qquad (1)$$

- In the same time interval $\Delta t'$, our speed will go from v_d (unknown) to 0, so:

$$0 = v_d - dec \cdot \Delta t' \qquad (2)$$

- Let's compute $\Delta t'$ from (2) and substitute in (1):

$$d = v_d \cdot \frac{v_d}{dec} - \frac{1}{2} \cdot dec \cdot (\frac{v_d}{dec})^2 \qquad (3)$$

Speed Profile and Distance

Where do the acc and dec phases meet?

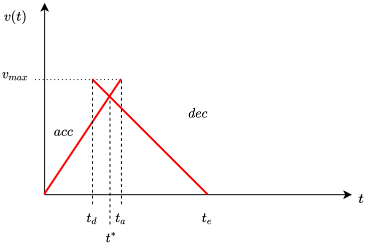
$$d = v_d \cdot \frac{v_d}{dec} - \frac{1}{2} \cdot dec \cdot \frac{v_d^2}{dec^2} \quad (4)$$

- Let's determine v_d from (4):

$$v_d = \sqrt{2 \cdot dec \cdot d} \quad (5)$$

- Formula (5) gives the **expected speed** v_d when we are at a distance d from the end of the motion

Speed Profile and Distance



Resolving the Overlapping

- Now, we are in the acceleration phase, and our speed is v
- According to our initial computation of the **deceleration distance** we have that, **hypothetically**, our deceleration should start at t_d, can we really enter in that phase?
- Since we know the distance to be travelled d, let's determine the expected speed v_d
- if $v_d > v$, **we are still in the acceleration phase**, so continue to accelerate until the condition becomes false

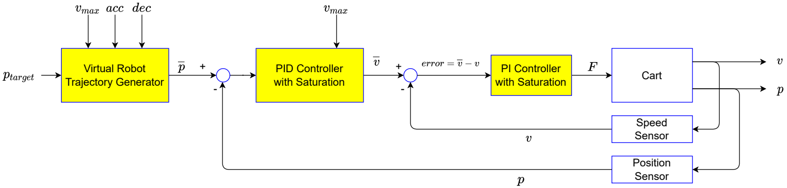
The final code

```
def evaluate(self, delta_t):
    if self.phase == VirtualRobot.ACCEL:
        self.p = self.p + self.v * delta_t \
                 + self.accel * delta_t * delta_t / 2
        self.v = self.v + self.accel * delta_t
        distance = self.p_target - self.p
        if self.v >= self.vmax:
            self.v = self.vmax
            self.phase = VirtualRobot.CRUISE
        elif distance <= self.decel_distance:
            v_exp = math.sqrt(2 * self.decel * distance)
            if v_exp < self.v:
                self.phase = VirtualRobot.DECEL

    elif self.phase == VirtualRobot.CRUISE:
        self.p = self.p + self.vmax * delta_t
        distance = self.p_target - self.p
        if distance <= self.decel_distance:
            self.phase = VirtualRobot.DECEL

    elif self.phase == VirtualRobot.DECEL:
        self.p = self.p + self.v * delta_t \
                 - self.decel * delta_t * delta_t / 2
        self.v = self.v - self.decel * delta_t
        if self.p >= self.p_target:
            self.v = 0
            self.p = self.p_target
            self.phase = VirtualRobot.TARGET
```

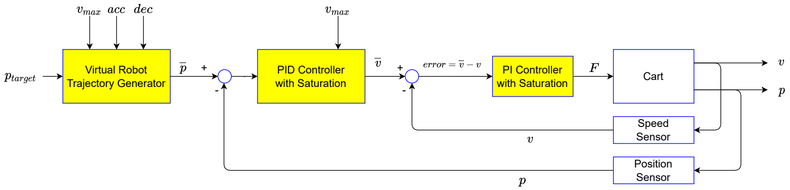
Back to Position Control



From Virtual to Real

- Now we have our virtual robot that travels according to a "path" generated from our initial requirements (distance, maximum speed, acceleration and deceleration)
- How can we use it in our real **position control**?
- The idea is to let the real robot "catch" the virtual robot

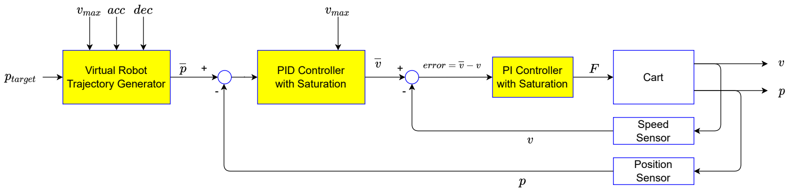
Back to Position Control



Catching the Virtual Robot

- The trajectory generator (our VirtualRobot class) gives the position $\overline{p}$ of the virtual robot time-by-time
- $\overline{p}$ is the position in which we expect to find also the real robot, but this will not be the case
- Let's determine the error $\overline{p} - p$ between expected and real position of the real robot and use a PID controller to compute the speed needed to reach $\overline{p}$
- In other words, the control system works in order to keep the error $\overline{p} - p$ as **non-zero** in order to output a travelling speed (until $\overline{p} = p_{target}$)

Catching the Virtual Robot



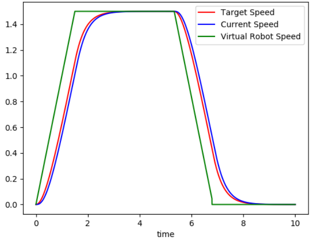
The Code

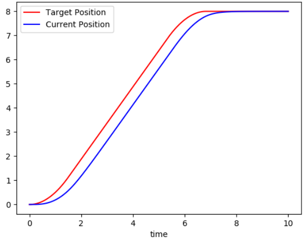
```
class CartRobot(RoboticSystem):

    def __init__(self):
        ...
        self.trajectory = VirtualRobot( 8, # distance 8 m
                                        1.5, # max speed 1.5 m/s
                                        1.0, # accel 1 m/s2
                                        1.0) # decel 1 m/s2
        self.speed_controller = PIDSat(10.0, 8.0, 0.0, 2.0, True)
        # Kp = 3, KI = 2, Sat = 2 N
        self.position_controller = PIDSat(???, 0.0, 0.0, 1.5)
        # Kp = ???, vmax = 1.5 m/s

    def run(self):
        self.trajectory.evaluate(self.delta_t)
        v_target = self.position_controller.evaluate(self.delta_t,
                                         self.trajectory.p, self.get_pose())
        F = self.speed_controller.evaluate(self.delta_t, v_target, self.get_speed())
        self.cart.evaluate(self.delta_t, F)
```

$K_P = 2.0$

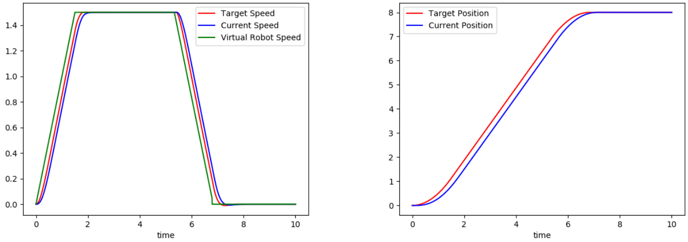




The Role of Constants of the Position Controller

- K_P controls the **delay** of the real robot with respect to the virtual robot
- It is only a delay not an error, since the target position is (sooner or later) reached

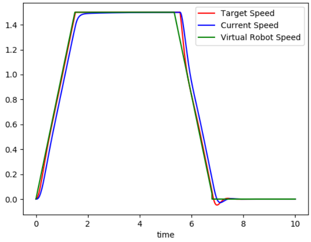
$$K_P = 4.0$$

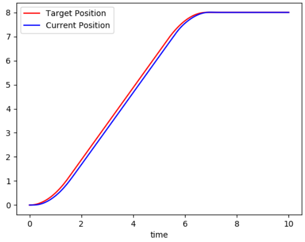


The Role of Constants of the Position Controller

- Interesting.... but still slow

$K_P = 8.0$

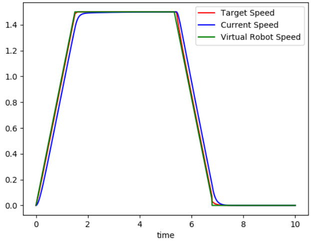


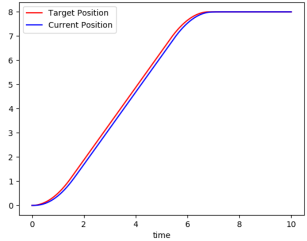


The Role of Constants of the Position Controller

- Very nice!! But there is an overshot
- Let's add a small derivative contribution ...

$$K_P = 8.0, K_D = 0.8$$

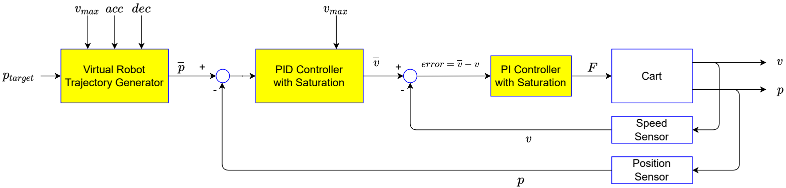




The Role of Constants of the Position Controller

- **It's OK!!**

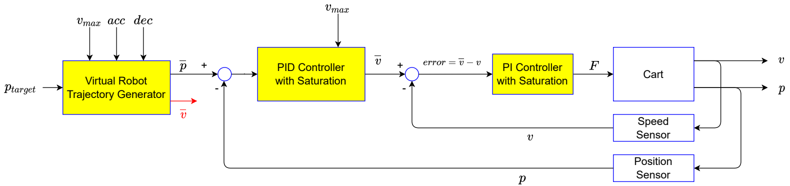
The Virtual Robot



Lesson Learned

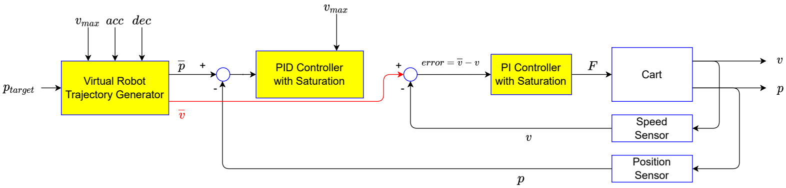
- The virtual robot is indeed a generator of the **theoretical trajectory** that, during time, must be followed by the real system
- Here we have a case with mono-dimensional motion and thus a single (position) variable to control
- However the same concepts can be applied when the trajectory is in a plane or in space

The Speed Profile Generator



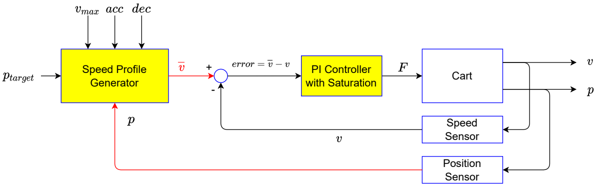
A small patch...

- The Virtual Robot Trajectory Generator outputs not only the theoretical position $\overline{p}$ but also the **theoretical speed** $\overline{v}$ that, during time, must be followed by the real system
- And the plots show that indeed the real speed is in accordance with the speed profile generated by the Virtual Robot
- Well...but, instead of using the position, could we consider directly the theoretical speed as the set-point the **speed controller**?



A small patch...

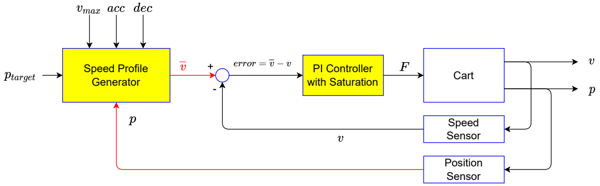
- Indeed we can consider to connect the **theoretical speed** $\overline{v}$ directly to the (final) speed controller
- The Position Controller is now useless and can be removed
- But the current position p cannot be left unconnected: it must be always sent in feedback, otherwise the concept of "control" does not apply and the control does work work



A small patch...

- The profile generator must posses the current position to output the right $\overline{v}$
- But, since we are not considering a virtual robot moving, we must (and can!) use directly to the real postion

The Speed Profile Generator



The Code (1)

```
class SpeedProfileGenerator:

    ACCEL = 0
    CRUISE = 1
    DECEL = 2
    TARGET = 3
    def __init__(self, _p_target, _vmax, _acc, _dec):
        self.p_target = _p_target
        self.vmax = _vmax
        self.accel = _acc
        self.decel = _dec
        self.v = 0 # current speed
        self.phase = SpeedProfileGenerator.ACCEL
        self.decel_distance = 0.5 * _vmax * _vmax / _dec
```

The Code (2)

```
class SpeedProfileGenerator:
    ...
    def evaluate(self, delta_t, current_pos):
        # Indeed this is not correct!!
        # We should consider that, if the target is overcome, we must go back!!
        if current_pos >= self.p_target:
            self.v = 0
            self.phase = SpeedProfileGenerator.TARGET
            return

        distance = self.p_target - current_pos
        if self.phase == SpeedProfileGenerator.ACCEL:
            self.v = self.v + self.accel * delta_t
            if self.v >= self.vmax:
                self.v = self.vmax
                self.phase = SpeedProfileGenerator.CRUISE
            elif distance <= self.decel_distance:
                v_exp = math.sqrt(2 * self.decel * distance)
                if v_exp < self.v:
                    self.phase = SpeedProfileGenerator.DECEL

        elif self.phase == SpeedProfileGenerator.CRUISE:
            if distance <= self.decel_distance:
                self.phase = SpeedProfileGenerator.DECEL

        elif self.phase == SpeedProfileGenerator.DECEL:
            self.v = math.sqrt(2 * self.decel * distance)
```

The Speed Profile Generator

Its Usage

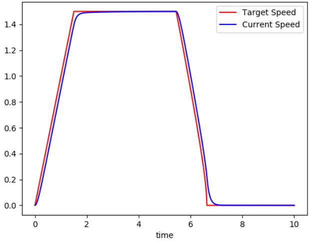
(see `test_position_control_with_profile_gui.py`)

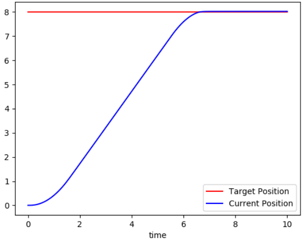
```
class CartRobot(RoboticSystem):

    def __init__(self):
        super().__init__(1e-3) # delta_t = 1e-3
        # Mass = 1kg
        # friction = 0.8
        self.cart = Cart(1, 0.8)
        self.plotter = DataPlotter()
        self.profile = SpeedProfileGenerator( 8, # distance 8 m
                                              1.5, # max speed 1.5 m/s
                                              1.0, # accel 1 m/s2
                                              1.0) # decel 1 m/s2
        self.speed_controller = PIDSat(10.0, 8.0, 0.0, 2.0, True)

    def run(self):
        self.profile.evaluate(self.delta_t, self.get_pose())
        F = self.speed_controller.evaluate(self.delta_t,
                                           self.profile.v, self.get_speed())
        self.cart.evaluate(self.delta_t, F)
```

The Speed Profile Generator





Controlling Position and Speed using Profiles

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
`santoro@dmi.unict.it`

Robotic Systems