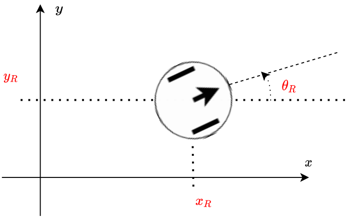
Model and Control of a Mobile Robot in a 2D Space

Corrado Santoro

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

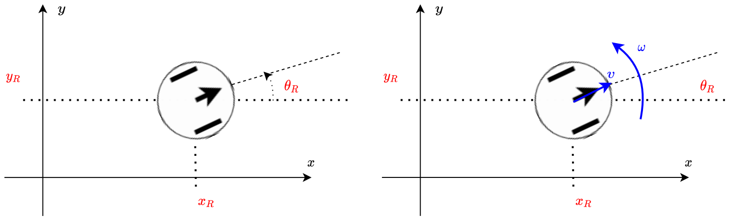
Robotic Systems

A Robot in a 2-Dimensional Space



Model

- Let us consider a mobile robot acting in 2D space
- Let us make no hypothesis on the driving system (two wheels, three wheels, four wheels, omnidirectional wheels, etc.)
- Let us consider the robot a **rigid body** with its mass center placed in the geometric center, e.g. a cylinder with uniform density



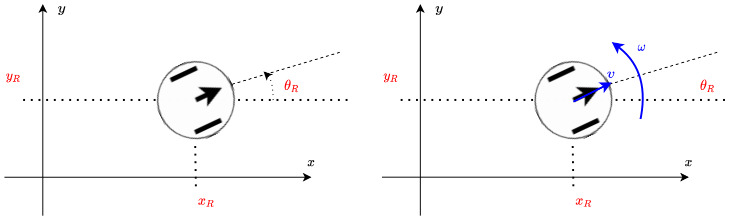
Cinematic Model

- From the cinematic point of view, the **pose** of the robot (position) is characterised by the 2D coordinates and the orientation:

$$\{x_R, y_R, \theta_R\}$$

- The robot's speed is characterised by the **linear speed of the mass' center** v and the **rotational speed** of the body ω

A Robot in a 2-Dimensional Space



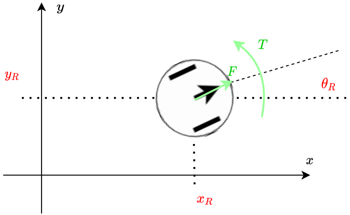
Cinematic Model

- The following relations hold:

$$\begin{aligned}
\dot{x}_R(t) &= v(t)\cos\theta(t) \\
\dot{y}_R(t) &= v(t)\sin\theta(t) \\
\dot{\theta}(t) &= \omega(t)
\end{aligned}$$

- where $\dot{x}_R(t)$ and $\dot{y}_R(t)$ the component of the linear speed v along x and y axis

A Robot in a 2-Dimensional Space

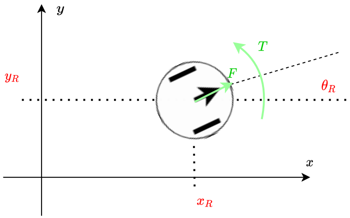


Dynamic Model

From the dynamic point of view, we consider that the driving system is able to apply:

- A **force** F to the mass' center
- A **torque** T for rotation along the vertical axis passing from the mass' center

A Robot in a 2-Dimensional Space



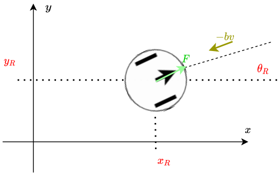
Dynamic Model

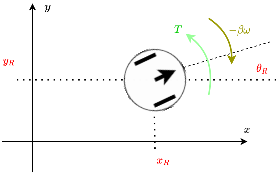
To model the physics of a rigid body, we must consider the second Newton's law in both linear and rotational aspects:

- **Linear:** $\sum F_i = Ma = M\dot{v}$
- **Rotational:** $\sum T_i = I\dot{\omega}$ where
  - T_i is the i^{th} torque applied
  - I is the **moment of inertia**
  - $\dot{\omega}$ is the **angular acceleration**

(see `https://en.wikipedia.org/wiki/List_of_moments_of_inertia`)

A Robot in a 2-Dimensional Space





Dynamic Model

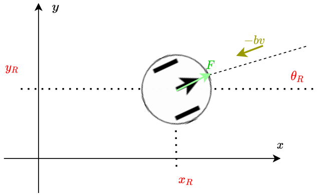
$$
\begin{aligned}
F - bv &= M\dot{v} \\
T - \beta\omega &= I\dot{\omega}
\end{aligned}
$$

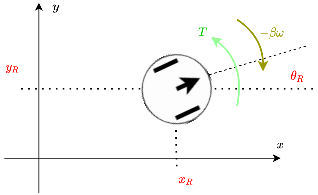
where β is the rotational friction coefficient

Since $I = \frac{1}{2}Mr^2$ for a cylindric robot, we have:

$$
\begin{aligned}
F - bv &= M\dot{v} \\
T - \beta\omega &= \frac{1}{2}Mr^2\dot{\omega}
\end{aligned}
$$

A Robot in a 2-Dimensional Space

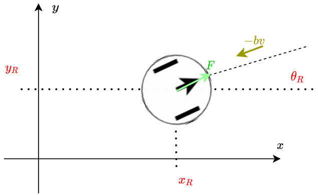




Dynamic Model

$$\dot{v} = -\frac{b}{M} v + \frac{1}{M} F$$

$$\dot{\omega} = -\frac{2\beta}{Mr^2}\omega + \frac{2}{Mr^2} T$$

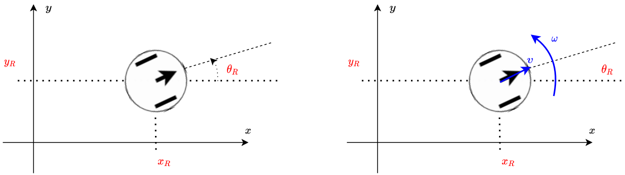


Dynamic Model

$$\begin{bmatrix} \dot{v} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} -\frac{b}{M} & 0 \\ 0 & -\frac{2\beta}{Mr^2} \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} + \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{2}{Mr^2} \end{bmatrix} \begin{bmatrix} F \\ T \end{bmatrix}$$

It is a **linear system** with two inputs

A Robot in a 2-Dimensional Space



Model Discretization

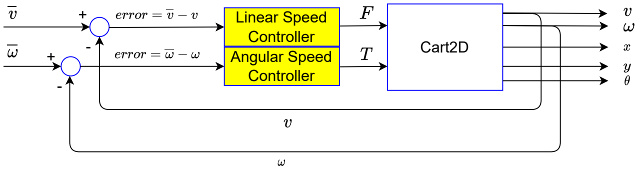
$$
\begin{aligned}
v(k+1) &= v(k) - \frac{b\Delta T}{M} v(k) + \frac{\Delta T}{M} F(k) \\
\omega(k+1) &= \omega(k) - \frac{2\beta\Delta T}{Mr^2}\omega(k) + \frac{2\Delta T}{Mr^2} T(k) \\
x_R(k+1) &= x_R(k) + v(k)\Delta T \cos\theta(k) \\
y_R(k+1) &= y_R(k) + v(k)\Delta T \sin\theta(k) \\
\theta(k+1) &= \theta(k) + \omega(t)\Delta T
\end{aligned}
$$

The Code

```
class Cart2D:

    def __init__(self, _mass, _radius, _lin_friction, _ang_friction):
        self.M = _mass
        self.b = _lin_friction
        self.beta = _ang_friction
        self.Iz = 0.5 * _mass * _radius * _radius
        # Iz = moment of inertia (the robot is a cylinder)
        self.v = 0
        self.w = 0
        self.x = 0
        self.y = 0
        self.theta = 0

    def evaluate(self, delta_t, _force, _torque):
        new_v = self.v * (1 - self.b * delta_t / self.M) \
                + delta_t * _force / self.M
        new_w = self.w * (1 - self.beta * delta_t / self.Iz) \
                + delta_t * _torque / self.Iz
        self.x = self.x + self.v * delta_t * math.cos(self.theta)
        self.y = self.y + self.v * delta_t * math.sin(self.theta)
        self.theta = self.theta + delta_t * self.w
        self.v = new_v
        self.w = new_w
```

Controlling the Speed

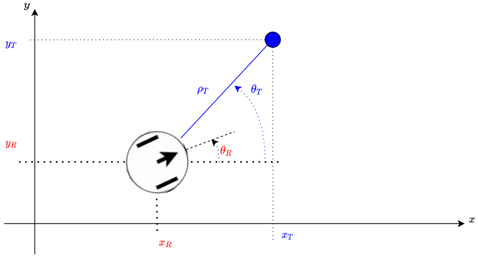
Speed Control in a 2-Dimensional Space



- Controlling the speed is quite straightforward
- v and ω are independent
- v depends only on F
- ω depends only on T
- We can use **two independent** speed controllers, one for each speed
- They can also be **tuned independently**

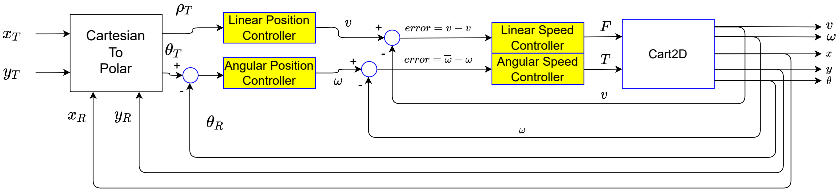
Controlling the Position

The Polar Position Control



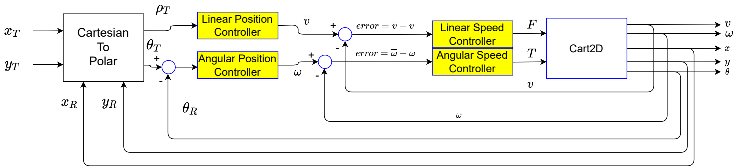
- Let us consider a robot with pose $\{x_R, y_R, \theta_R\}$
- We want the robot reach position $\{x_T, y_T\}$
- The theoretical trajectory is the blue line
- So we can consider **two different targets**
  - The **distance** ρ_T
  - The **heading** θ_T
- And we want to control both simultaneously

The Polar Position Control



- We can consider
  - ρ_T as a **distance error**
  - The heading difference $\theta_T - \theta_R$ as the **heading error**
- ρ_T can drive a linear position controller giving the target v
- $\theta_T - \theta_R$ can drive a linear angular controller giving the target ω

The Polar Position Control



Cartesian to Polar

$$\rho_T = \sqrt{(x_T - x_R)^2 + (y_T - y_R)^2}$$

$$\theta_T = \arctan \frac{y_T - y_R}{x_T - x_R}$$

$$\theta_{error} = \theta_T \ominus \theta_R$$

The Sign of the Distance

$$\begin{aligned}\rho_T &= \sqrt{(x_T - x_R)^2 + (y_T - y_R)^2}\\ \theta_T &= \arctan \frac{y_T - y_R}{x_T - x_R}\\ \theta_{error} &= \theta_T \ominus \theta_R\end{aligned}$$

- According to the formula above, the distance is always positive
- But, what does it happen if the robot **overcomes** the target?
- We expect that the distance becomes **negative**, but, with those formulas, this is not the case!
- We can instead use the **heading error**: if the target (and thus θ_{error}) is in the second or third quadrant, the target is **behind** the robot, and we can change:
  - The sign of ρ_T
  - θ_T by adding π

(see `Polar2DController` in `libs/controllers/control2d.py`)

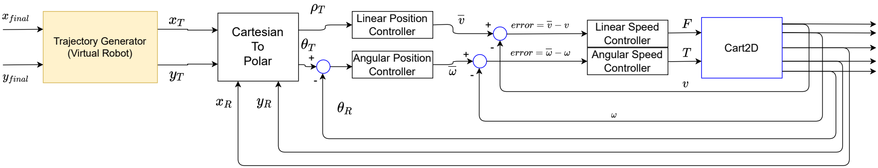
Implementing the Polar Controller

lib/controllers/control2d.py

```
class Polar2DController:

    def __init__(self, KP_linear, v_max, KP_heading, w_max):
        self.linear  = PIDSat(KP_linear, 0, 0, v_max)
        self.angular = PIDSat(KP_heading, 0, 0, w_max)

    def evaluate(self, delta_t, xt, yt, current_pose):
        (x, y, theta) = current_pose

        dx = xt - x
        dy = yt - y

        target_heading = math.atan2(dy , dx)

        distance = math.sqrt(dx*dx + dy*dy)
        heading_error = normalize_angle(target_heading - theta)

        if (heading_error > math.pi/2)or(heading_error < -math.pi/2):
            distance = -distance
            heading_error = normalize_angle(heading_error + math.pi)

        v_target = self.linear.evaluate_error(delta_t, distance)
        w_target = self.angular.evaluate_error(delta_t, heading_error)

        return (v_target, w_target)
```

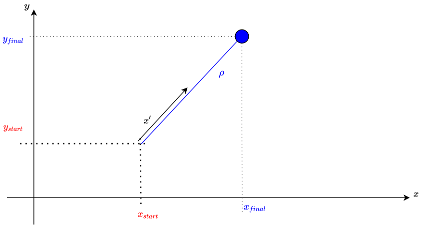
Following a Trajectory

Following a Trajectory



- The polar control uses two P controllers to control position, therefore it does not give the possibiity to specify acceleration or deceleration ramps
- In these cases, a **trajectory generator** can be used to give the (moving) target position (virtual robot) that has to be reached time-by-time by the (real) robot

Following a Trajectory



- Let us consider the robot in the initial position $\{x_{start}, y_{start}\}$ and that we want to reach position $\{x_{final}, y_{final}\}$ using a straight line
- We can consider a change in the reference frame with the x' along the straight line and a **virtual robot** moving along such a line
- The 1D-virtual robot algorithm gives the position $x'(t)$ of the virtual robot at time instant t
- Then it is **roto-translated** to the $\{x, y\}$ frame thus generating the couple $\{x_T, y_T\}$ to be provided to the Polar Controller

Implementing Virtual Robot in 2D

lib/controllers/control2d.py

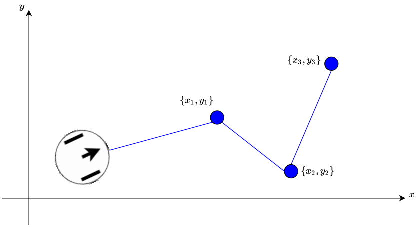
```
class StraightLine2DMotion:

    def __init__(self, _vmax, _acc, _dec):
        self.vmax = _vmax
        self.accel = _acc
        self.decel = _dec

    def start_motion(self, start, end):
        (self.xs,self.ys) = start
        (self.xe,self.ye) = end

        dx = self.xe - self.xs
        dy = self.ye - self.ys

        self.heading = math.atan2(dy , dx)
        self.distance = math.sqrt(dx*dx + dy*dy)

        self.virtual_robot = VirtualRobot(self.distance,
                                          self.vmax, self.accel, self.decel)

    def evaluate(self, delta_t):
        self.virtual_robot.evaluate(delta_t)

        xt = self.xs + self.virtual_robot.p * math.cos(self.heading)
        yt = self.ys + self.virtual_robot.p * math.sin(self.heading)

        return (xt, yt)
```

Using the Virtual Robot in 2D

tests/cart_2d/test_robot_trajectory.py

```
class Cart2DRobot(RoboticSystem):

  def __init__(self):
      super().__init__(1e-3) # delta_t = 1e-3
      # Mass = 1kg, radius = 15cm, friction = 0.8
      self.cart = Cart2D(1, 0.15, 0.8, 0.8)

      self.linear_speed_controller = PIDSat(10, 3.5, 0, 5) # 5 newton
      self.angular_speed_controller = PIDSat(6, 10, 0, 4)

      self.polar_controller = Polar2DController(0.5, 2, 2.0 , 2)
      self.trajectory = StraightLine2DMotion(1.5, 2, 2)

      (x,y,_) = self.get_pose()
      self.trajectory.start_motion( (x,y), (0.5, 0.2) )

  def run(self):
      (x_target, y_target) = self.trajectory.evaluate(self.delta_t)
      (v_target, w_target) = self.polar_controller.evaluate(self.delta_t,
                                   x_target, y_target, self.get_pose())
      Force = self.linear_speed_controller.evaluate(self.delta_t,
                                                v_target, self.cart.v)
      Torque = self.angular_speed_controller.evaluate(self.delta_t,
                                                w_target, self.cart.w)
      self.cart.evaluate(self.delta_t, Force, Torque)
      return True
```

Following a More Complex Trajectory



- But if we want to follow a **generic** path?
- A basic solution is to **split** the path into a sequence of segments and follow each segment
- Once an intermediate point is reached, we start following the next segment
- However, in checking the arrival to a point, a threshold is always needed

The Path Follower 2D

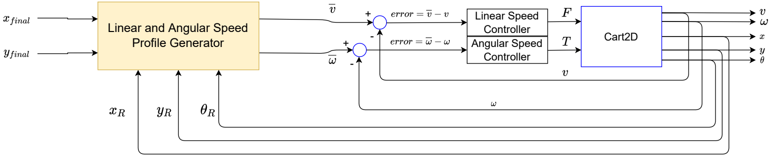
lib/controllers/control2d.py

```
class Path2D:

    def __init__(self, _vmax, _acc, _dec, _threshold):
        self.threshold = _threshold
        self.path = [ ]
        self.trajectory = StraightLine2DMotion(_vmax, _acc, _dec)

    def set_path(self, path):
        self.path = path

    def start(self, start_pos):
        self.current_target = self.path.pop(0)
        self.trajectory.start_motion(start_pos, self.current_target)

    def evaluate(self, delta_t, pose):
        (x, y) = self.trajectory.evaluate(delta_t)
        target_distance = math.hypot(pose[0] - self.current_target[0],
                                     pose[1] - self.current_target[1])
        if target_distance < self.threshold:
            if len(self.path) == 0:
                return None
            else:
                self.start( (x,y) )

        return (x,y)
```

Using the Path Follower

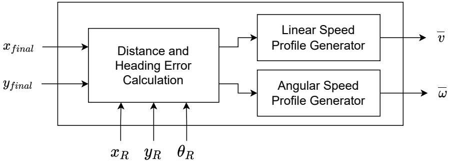
tests/cart_2d/test_robot_path.py

```
class Cart2DRobot(RoboticSystem):

    def __init__(self):
        super().__init__(1e-3) # delta_t = 1e-3
        self.cart = Cart2D(1, 0.15, 0.8, 0.8)
        self.linear_speed_controller = PIDSat(10, 3.5, 0, 5) # 5 newton
        self.angular_speed_controller = PIDSat(6, 10, 0, 4) # 4 newton * metro
        self.polar_controller = Polar2DController(0.5, 2, 2.0 , 2)
        self.path_controller = Path2D(1.5, 2, 2, 0.01) # tolerance 1cm
        self.path_controller.set_path( [ (0.5, 0.2),
                                         (0.5, 0.4),
                                         (0.2, 0.2) ] )
        (x, y, _) = self.get_pose()
        self.path_controller.start( (x,y) )

    def run(self):
        target = self.path_controller.evaluate(self.delta_t,
                                               self.get_pose())
        if target is None:
            return False
        (x_target, y_target) = target
        (v_target, w_target) = self.polar_controller.evaluate(self.delta_t, x_ta
        Force = self.linear_speed_controller.evaluate(self.delta_t, v_target, se
        Torque = self.angular_speed_controller.evaluate(self.delta_t, w_target,
        self.cart.evaluate(self.delta_t, Force, Torque)
        return True
```



- Similarly to the 1-D case, we can generate the $\overline{v}$ and $\overline{\omega}$ directly from distance and heading errors
- This implies merging the Trajectory Generator, Cartesian-To-Polar and Position Controllers into a unique control block

Following a Speed Profile



- Starting from the comparison between the target and the pose, the **distance** and the **heading error** are computed
- They are then passed to the two blocks that (according to the error) generate the proper speed using the trapezoidal profile

(see class **`SpeedProfileGenerator2D`** in **`lib/models/virtual_robot.py`** and **`tests/cart_2d/test_robot_speed_profile.py`**)

Model and Control of a Mobile Robot in a 2D Space

Corrado Santoro

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

Robotic Systems