Mondragon Unibertsitatea
Faculty of Engineering

# Robotic Control Systems

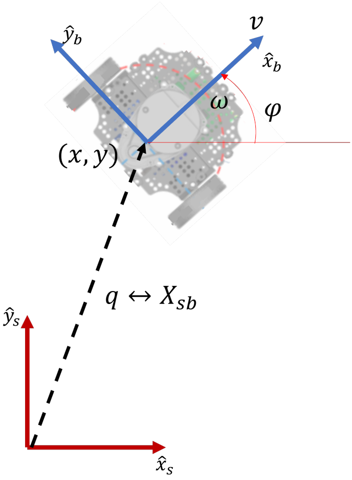
## PBL: Control of a Turtlebot mobile robot

# Outline

1. Modelling the Turtlebot
2. Simulink blocks and files
3. Implementation of the controller on the Turtlebot
4. Time-discretisation of PID controllers

# 1 Modelling the Turtlebot

# Choice of coordinates

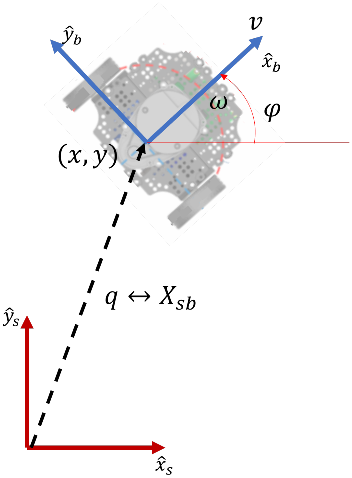



- Fix a reference frame $\{s\}$ and attach a reference frame $\{b\}$ to the robot (e.g. in the middle point between the wheels).
- The robot position can be represented either by a configuration $X_{sb} \in SE(3)$ or a minimal set of coordinates $q_{sb} = (x, y, \phi)$.
- The two representations are linked:

$$X_{sb} = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 & x \\ \sin(\phi) & \cos(\phi) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Kinematics model






- Call $v$ the forward velocity and $\omega$ the angular velocity.
- The kinematics are given by

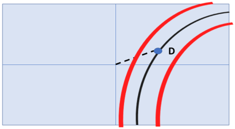
$$\dot{q}_{sb} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos(\phi) & 0 \\ \sin(\phi) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

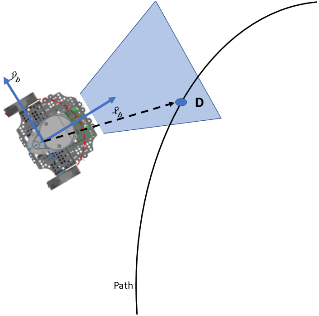
- NOTE: ROS allows the user to set $v$ and $\omega$ for the Turtlebot, with the following limits:

$$|v| \leq 0.22 \frac{\mathrm{m}}{\mathrm{s}}$$

$$|\omega| \leq 2.84 \frac{\mathrm{rad}}{\mathrm{s}}$$

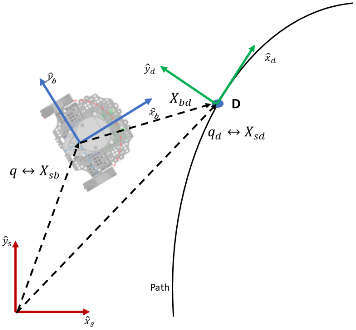
# Modelling the vision system




- The aim of the control system is to follow a given path: starting from the actual position ($q_{sb}$) the robot should be driven to a point $D$ on the path.
- The vision system is in charge to detect the point $D$ on the path.
- Problem: how can we represent the point $D$?

# Reference frames




- Fix a frame $\{d\}$ at $D$, represented either by $X_{sd}$ or $q_{sd} = (x', y', \phi')$ with respect to $\{s\}$ ($\hat{x}_d$ is taken tangent to the path).
- Call $q_{bd} = (x'_b, y'_b, \phi'_b)$ the coordinates of $D$ in $\{b\}$.
- $X_{bd}$ will be in the form

$$X_{bd} = \begin{bmatrix} \cos(\phi'_b) & -\sin(\phi'_b) & 0 & x'_b \\ \sin(\phi'_b) & \cos(\phi'_b) & 0 & y'_b \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Then, we have:

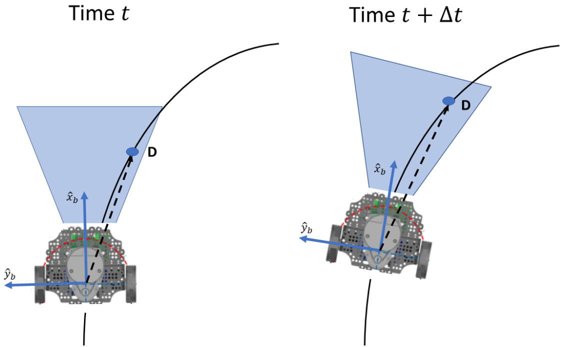
$$X_{bd} = X_{bs} X_{sd} = X_{sb}^{-1} X_{sd}$$

# Control system design

The aim of the control system is to make $q_{bd} = (x'_b, y'_b, \phi'_b) = 0$ (or equivalently $X_{bd} = I$).

- The output of the control system are the signals $(v, \omega)$.
- The choice of the inputs of the control system is up to the students. Possible choices are:
  - $\log(X_{bd})$: error in the spatial directions and in angle.
  - $(x'_b, y'_b)$: error in the spatial directions.
  - $y'_b$: error in the spatial direction perpendicular to the velocity $v$.
- A possible control scheme is PID (or a combination of PIDs).
  - HINT: At least at first set $v = 0.1$ and use a PID only to control $\omega$.

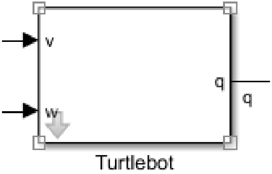
# Tracking error




- It is impossible to obtain $q_{bd}(t) = 0$: every time the robot comes close to the point $D$ the point itself will move ahead on the path.
- A better path tracking performance indicator is the distance between the frame $\{b\}$ and the path.

# 2 Simulink blocks and files

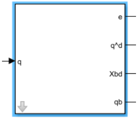
# Turtlebot model




- Simple model of a Turtlebot.
- Inputs:
  - v: Linear velocity [m/s]
  - w: Angular velocity [rad/s]
- Outputs:
  - qsb: Configuration vector $q_{sb} = [x, y, \phi]^T$

# Trajectory generation






Trajectory generation
(Vision system)

- Generate a trajectory by simulating a vision system that detects a path.
- The path can be chosen to be a circle (radius=5$\mathrm{m}$) or a straight line ($y = x$).
- Input:
  - qsb: Configuration vector of the Turtlebot; $q_{sb} = [x, y, \phi]^T$.
- Outputs:
  - e: Distance with respect to the path (only for visualisation).
  - qsd: Desired trajectory $q_{sd}$ in space frame $\{s\}$ (only for visualisation).
  - Xbd: SE(3) configuration of the desired position in the trurtlebot body frame $\{b\}$.
  - qbd: Minimal set of coordinates that represents the desired position in the trurtlebot body frame $\{b\}$ (same information as Xbd but in different format); $q_{bd} = [x_b', y_b', \phi_b']^T$.

# Provided files in MUdle

Mondragon Unibertsitatea

- *TurtlebotOpenLoop.slx*: Open-loop simulation of the robot kinematics.
- *SimulateOL.mlx*: Live editor that allows to simulate the Turtlebot in open-loop. It contains a code that allows you a graphical plot of the movement of the Turtlebot.
- *TurtlebotClosedLoop_template.slx*: Template file to build the closed-loop simulation model.
- *ModernRoboticsLibrary*: Companion code of the Modern Robotics book (in case you need it. . . ).

# 3 Implementation of the controller on the Turtlebot

# Comments on the implementation

## Some comments

- The actual test path is a combination of straight and curved lines.
- The actual vision system is up to you to implement and it may be different from the provided model.
- The parameters of the controller (e.g. PID gains) will not be the same
  - Different inputs (due to different vision system).
  - Discrete-time (Turtlebot) vs Continuous-time (Simulink).
- In the real environment you don't have an external reference frame $\{s\}$, since you don't have external sensors (e.g. cameras).
- The controller can be implemented either in the Turtlebot (best option) or in the remote PC (Ubuntu virtual machine; suboptimal option). Both options will be accepted.
- The integration of the controller with other modules (e.g. deep learning, signal processing) is up to you.

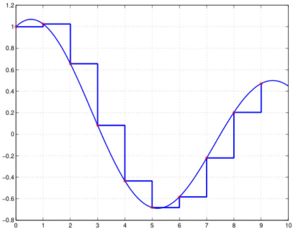
# 4 Time-discretisation of PID controllers

# Continuous and discrete-time signals

- Let the digital controller operate with sampling time $T$.
- Then a discrete-time signal is defined as:

$$u_k \triangleq u(kT) \quad k \in \{0, 1, \dots\}$$

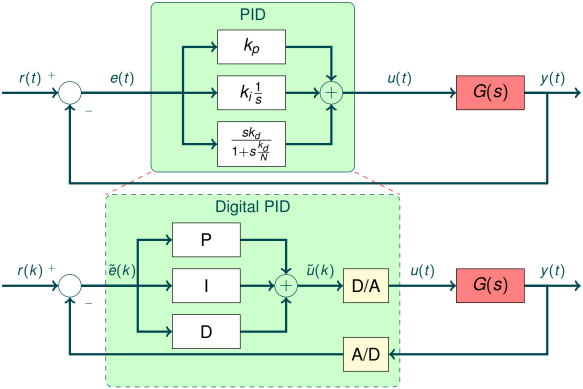
- $u_k$ is a time-series, that is a sequence of points.[1]
- A similar definition can be used for the other signals: $x_k$, $y_k$, $e_k$, etc.




[1] Notice that the subindex $k$ does not indicate the $k$-th component of a vector. In this case it is a time subindex used to indicate $k$-th element in a series.

# Time-discretisation of PID controllers






- We will analyse each part P, I, and D separately.

$$u(kT) = P(kT) + I(kT) + D(kT) = P_k + I_k + D_k$$

# Proportional and integral terms

- The proportional term $k_p e(t)$ is implemented simply by replacing the continuous variables with their sampled versions:

$$P_k = k_p e_k$$

- The integral term is obtained by approximating the integral with a sum (forward Euler method):

$$I_{k+1} = I_k + k_i T e_k$$

- If an anti-windup scheme is used it should be taken into account. For example, with back-calculation:

$$I_{k+1} = I_k + k_i T e_k + \frac{T}{T_t}(sat(u_k) - u_k)$$

# Derivative term

- The filtered derivative term $D(t) = \frac{sk_d}{1+s\frac{k_d}{N}}e(t)$ corresponds to the following differential equation

$$\frac{k_d}{N}\dot{D}(t) + D(t) = k_d\dot{e}(t)$$

- This differential equation can be integrated with backward Euler method:

$$\frac{k_d}{N}\frac{D_k - D_{k-1}}{T} + D_k = k_d\frac{e_k - e_{k-1}}{T}$$

and thus we obtain:

$$D_k = \frac{\frac{k_d}{N}}{\frac{k_d}{N} + T}D_{k-1} + \frac{k_d}{\frac{k_d}{N} + T}\left(e_k - e_{k-1}\right)$$

- The advantage of using a backward Euler is that the parameter $\frac{\frac{k_d}{N}}{\frac{k_d}{N}+T}$ is nonnegative and less than 1 for all $T > 0$, which guarantees that the difference equation is stable.

# Digital PID pseudo-code

```
//Precompute controller coefficients and initialise
Tf = Td/N;
ad = Tf/(Tf + T);
bd = kd/(Tf + T);
I = 0;
D = 0;
//Control algorithm - main loop
while (running) do
    r = adc(ch1) ;                          //read setpoint from ch1
    y = adc(ch2) ;                     //read plant output from ch2
    e = r - y ;                                    //compute error
    P = kp * e ;                       //compute proportional part
    D = ad * D + bd * (e - eold) ;      //update derivative part
    v = P + I + D ;                     //compute temporary output
    u = sat(v, ulow, uhigh) ;   //simulate actuator saturation
    dac(u, ch1)) ;                       //set analogue output ch1
    I = I + ki * T * e + T/Tt * (u - v)) ;      //update integral
    eold = e ;                  //simulate actuator saturation
    sleep(T) ;                //wait until next update interval
end
```

Mondragon Unibertsitatea
Faculty of Engineering

# Questions?