Reinforcement Learning Intro

Reading

- Sutton, Richard S., and Barto, Andrew G. Reinforcement learning: An introduction. MIT press, 2018.
  - http://www.incompleteideas.net/book/the-book-2nd.html
  - Chapter 1
- E.A. Lee and S.A. Seshia, Introduction to Embedded Systems: CPS Approach, Second Edition, MIT Press, 2017
  - https://ptolemy.berkeley.edu/books/leeseshia/releases/LeeSeshia_DigitalV2_2.pdf
  - Chapter 2
- Puterman, Martin L. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
  - Chapter 1

Overview

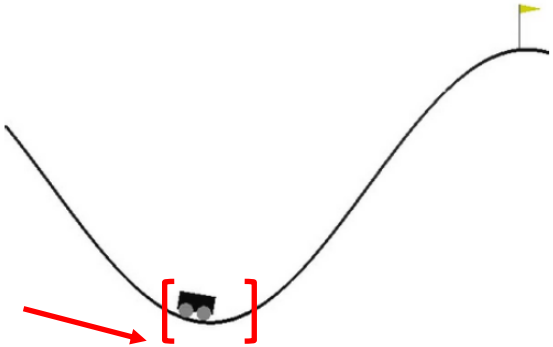
- RL is learning what to do, i.e., map situations to actions
  - Typically in the form of maximizing a numerical reward
- The learner is not told what to do
  - Need to explore the space and discover which actions yield the most return
- RL can be used in many settings
  - Control, scheduling of tasks, training language models
- Control is most relevant to this course
  - An alternative to standard control theoretic methods, especially in complex environments, such as image-based control

Comparison with other types of learning

- Different from supervised learning
  - No access to carefully collected labeled data
- Different from unsupervised learning
  - Not trying to learn relationships between unlabeled data
- Similar to unsupervised learning
  - Learning is largely “unsupervised”, agent must explore and learn on its own
- Similar to supervised learning
  - Over time, labeled state-action-reward pairs are collected
- Overall, RL considered a different learning paradigm

Example, Mountain Car

- A benchmark reinforcement learning problem
- Learn a controller to get an underpowered car up a hill
  - Need to go up left hill first
  - Small negative reward after each step (smaller for higher inputs)
  - Big positive reward if goal is reached

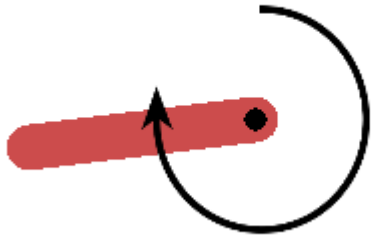


Initial condition chosen randomly from this range

- Learning problem considered “solved” if average reward over 100 random trials is over 90
  - Go up the hill *fast* while conserving energy

Example, Inverted Pendulum

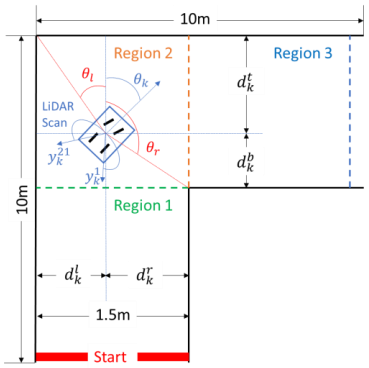
- A benchmark reinforcement learning problem
- Learn a controller to stabilize the pendulum vertically
  - Need to swing it to one side first and then swing the other way
  - Small negative reward after each step (smaller for higher inputs)
  - The longer it takes you to stabilize the pendulum, the lower the reward



- There is no “solved” threshold, an average reward above -200 is generally a good sign

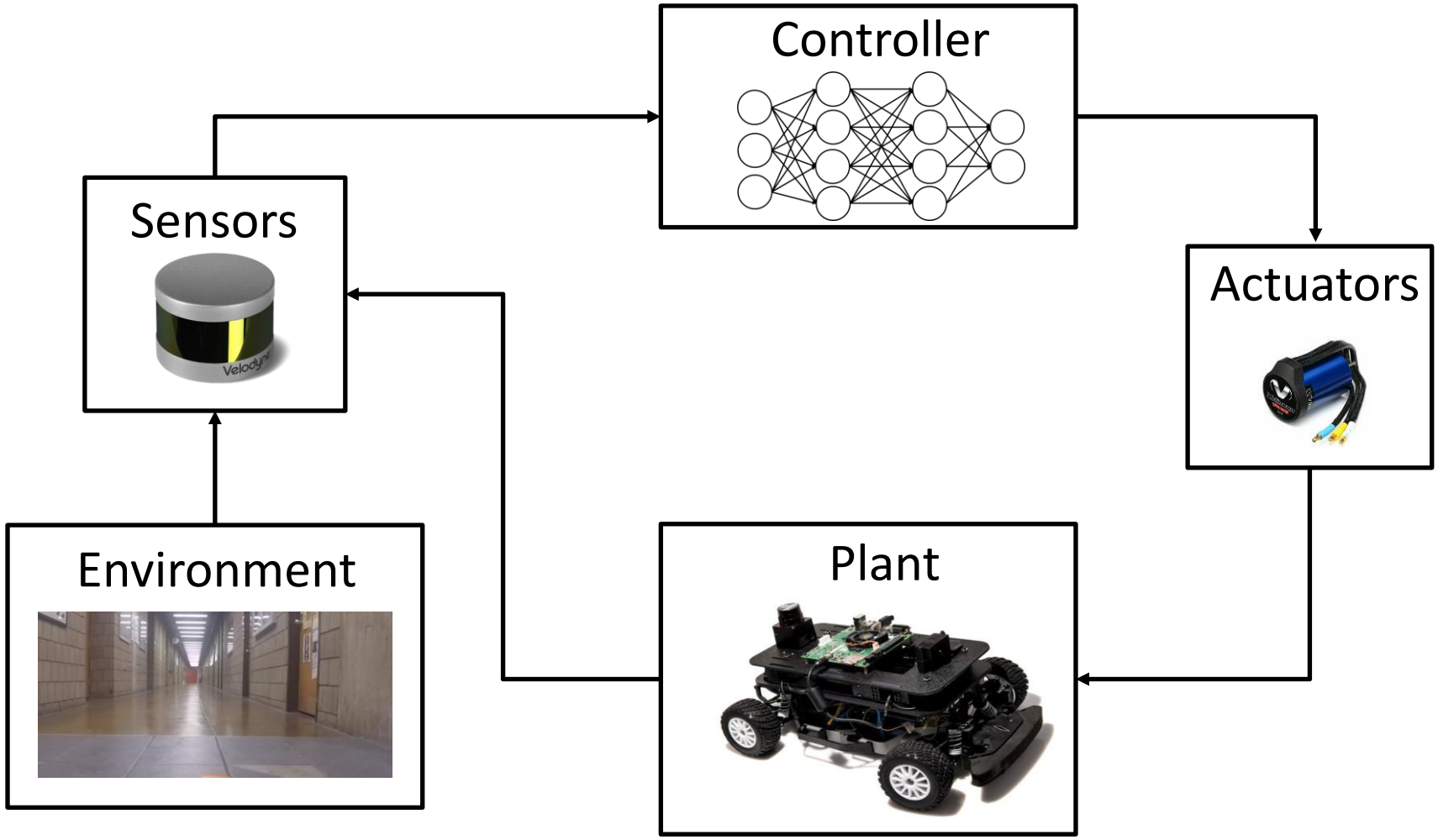
Example, F1/10

- (Soon-to-be) A benchmark reinforcement learning problem
- Learn a controller to navigate a hallway environment
  - Get a small positive reward after each step with no crash
  - Get a big negative reward upon crash
  - Over time, learn to avoid walls



- This problem can be solved with standard control techniques but only for known environments with regular shapes

Standard Control Loop



Other Examples

- Chess (and other games)
  - Select a (sequence of) move that leads to victory
- Learning to walk (in simulation)
  - Select joint/muscle actions that lead to stability and movement
- Learn to flip pancakes
- Fold proteins
- Many, many, many more

Elements of RL

- Agent
  - Robot, decision maker who is learning the task
- Environment
  - Agent's environment, e.g., obstacles, other objects, other agents
- Policy
  - A mapping from perceived states (measurements) to actions
  - i.e., a controller
- Reward signal
  - Defines the goal of the RL problem
  - Observe a reward after each action and corresponding state change
  - Easier for some tasks than for others – need to be able to quantify the conceptual goal (e.g., walking, driving safely)

Standard Control Approach

- Model system/environment
  - Typically, in the form of automata/differential equations
- Encode goal as a cost function over some time horizon
- Design controller to minimize cost function
- This paradigm is known as model predictive control (MPC)
- Can also build (simpler) controllers without needing models
  - More later

A Simple Dynamics Model

- Suppose a car is moving in a straight line at $v\ m/s$
- How much will the car have travelled after $T\ s$?

$$vT\ m$$

- Let the car's position at time 0 be p_0 and at time T be p_T

$$p_T = p_0 + vT$$

- Suppose every T seconds velocity jumps up by $a\ m/s$
- How do we adapt the model (for discrete times when velocity is changed)?

$$p_{kT} = p_{(k-1)T} + v_{(k-1)T}T$$
$$v_{kT} = v_{(k-1)T} + a$$

  - where $k = 1,2, \ldots$

Elements of a dynamical system model

- Note: notation will change when we get to RL proper
- System has a state, denoted by $\boldsymbol{x} \in \mathbb{R}^n$
  - Captures position, velocity, acceleration, etc.
- Control inputs are denoted by $\boldsymbol{u} \in \mathbb{R}^p$
  - Captures throttle, steering, etc.
- Measurements are denoted by $\boldsymbol{y} \in \mathbb{R}^q$
  - Could measure states directly, e.g., odometry, GPS
  - Could be high-dimensional such as camera, LiDAR

State Evolution

- As time passes, the system state evolves based on the previous state and the current control inputs
- We typically model the state as a signal:

$$\boldsymbol{x}: \mathbb{R}_+ \to \mathbb{R}^n$$

  - i.e., for a given time t, $\boldsymbol{x}(t)$ returns the state at that time
- If we want to model the evolution of $\boldsymbol{x}$ in continuous time, we describe it with ordinary differential equations:

$$\dot{\boldsymbol{x}} := \frac{\partial \boldsymbol{x}(t)}{\partial t} = f\big(\boldsymbol{x}(t), \boldsymbol{u}(t)\big)$$

- Modern systems are digital, so a discrete-time model makes more sense (since controller is sampled at discrete times)

$$\boldsymbol{x}_{k+1} = f(\boldsymbol{x}_k, \boldsymbol{u}_k)$$

  - where k is incremented with the sampling rate (e.g., 10Hz)

State Evolution Example

- Going back to the position/velocity example:

$$p_{kT} = p_{(k-1)T} + v_{(k-1)T}T$$
$$v_{kT} = v_{(k-1)T} + a$$

- This is a discrete-time model where $\boldsymbol{x} = [p, v]^T, u_k = a$, so

$$f([x_1, x_2], u) = \begin{bmatrix} x_1 + x_2 T \\ x_2 + u \end{bmatrix}$$

- In this case, f is linear, so the system can also be written as

$$\boldsymbol{x}_{k+1} = \boldsymbol{A}\boldsymbol{x}_k + \boldsymbol{B}u_k$$

  - where $\boldsymbol{A} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}, \boldsymbol{B} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

- Note that we implicitly dropped the T in the subscript
  - It is redundant, since k is chosen for a sampling rate of T

Measurement model

- Measurements are typically modeled as a function of the state:

$$\boldsymbol{y}_k = g(\boldsymbol{x}_k)$$

- In our example, if we can only measure position, then

$$\boldsymbol{y}_k = \boldsymbol{C}\boldsymbol{x}_k$$

  - where $\boldsymbol{C} = [1 \quad 0]$

- In case of more complex measurements, g may be quite complex or (as is often the case) unknown
  - In the F1/10 case, LiDAR measurements can be modeled as a function of the car state and the hallway dimensions
  - Modeling a camera would be significantly harder

General Model Form

- In its most general form, the model can be written as

$$\boldsymbol{x}_{k+1} = f(\boldsymbol{x}_k, \boldsymbol{u}_k)$$
$$\boldsymbol{y}_k = g(\boldsymbol{x}_k)$$
$$\boldsymbol{u}_k = h(\boldsymbol{y}_k)$$

  - This model has the Markov property, i.e., the current state depends only on the previous state and control
    - It doesn't matter how we got to the previous state

Designing Controllers

- Suppose we focus on the state-feedback problem first, i.e.,

$$\boldsymbol{x}_{k+1} = f(\boldsymbol{x}_k, \boldsymbol{u}_k)$$
$$\boldsymbol{u}_k = h(\boldsymbol{x}_k)$$

- Given f, one needs to design a controller $\boldsymbol{u}_k = h(\boldsymbol{x}_k)$
  - E.g., to navigate the track as fast as possible
  - How do we pick the controls $\boldsymbol{u}_k$?
  - Minimize a cost function (surprise, surprise), e.g.,

$$J = \boldsymbol{x}_{k+H}^T \boldsymbol{Q} \boldsymbol{x}_{k+H} + \sum_{j=0}^{H-1} \boldsymbol{x}_{k+j}^T \boldsymbol{Q} \boldsymbol{x}_{k+j} + \boldsymbol{u}_{k+j}^T \boldsymbol{R} \boldsymbol{u}_{k+j} + \boldsymbol{x}_{k+j}^T \boldsymbol{N} \boldsymbol{u}_{k+j}$$

  - where H is a time horizon, $\boldsymbol{Q}$, $\boldsymbol{R}$ and $\boldsymbol{N}$ are user-defined matrices

Cost Function Considerations

- The cost function

$$J = \boldsymbol{x}_{k+H}^T \boldsymbol{Q} \boldsymbol{x}_{k+H} + \sum_{j=0}^{H-1} \boldsymbol{x}_{k+j}^T \boldsymbol{Q} \boldsymbol{x}_{k+j} + \boldsymbol{u}_{k+j}^T \boldsymbol{R} \boldsymbol{u}_{k+j} + \boldsymbol{x}_{k+j}^T \boldsymbol{N} \boldsymbol{u}_{k+j}$$

  is known as the linear quadratic regulator (LQR)
  - Can be solved iteratively for linear systems
- Matrices $\boldsymbol{Q}$, $\boldsymbol{R}$ and $\boldsymbol{N}$ chosen to satisfy control requirements
  - e.g., reach a target, minimize fuel consumption
- Having a horizon allows to plan more complex strategies
  - E.g., mountain car is easily solved
- Optimal control is extremely well studied
  - Strong theory and optimality guarantees for linear systems
  - However, non-linear systems have no general solutions

RL vs. Control

- In many existing settings, standard control is superior to RL
  - Easier to understand, requires (significantly) less computation and easier to adapt/modify
- However, complex controllers require good models
  - Hard to model complex systems (e.g., a quadruped robot)
- Furthermore, building symbolic algorithms to analyze high-dimensional measurements is very hard
  - E.g., recognize objects in images
  - This is known as the perception problem
- In such cases, it makes sense to try to directly learn the controller from data

Exploration vs Exploitation

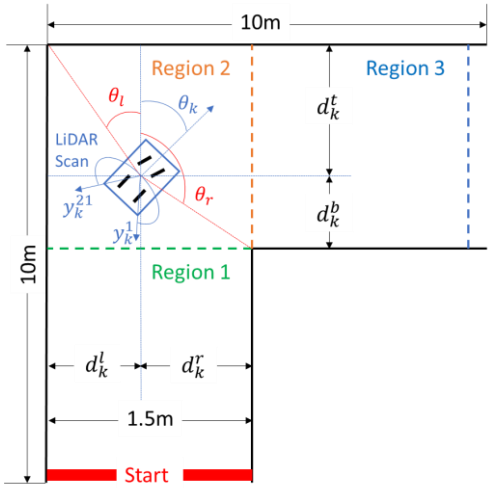
- One of the major challenges in RL
- More exploration allows the agent to observe larger parts of the state space and discover higher-reward actions
  - Essentially learn the unknown system model
  - At the expense of more random actions and failures
- More exploitation allows the agent to perform actions that are already known to produce good rewards
  - At the expense of getting stuck in a local minimum
- Fundamental trade-off that does not have an obvious solution
  - Solution is typically task-specific

RL vs Control, cont'd

- RL works best when we have a structured task with the ability to generate a lot of data
  - E.g., games, protein folding
  - In such cases, it is likely to be superior to standard control
- However, RL is computationally very expensive
  - A lot of iterations necessary and typically no convergence guarantees
  - Often not easy to identify the issue (insufficient exploration, small models, not enough training)

F1/10 Car Simulator

- Developed a model for the F1/10 car as part of my research
- Car navigates a hallway environment while avoiding collisions
  - Has access to LiDAR measurements (laser scan)
- Modeled the car dynamics as well as the LiDAR measurements
- Control inputs are throttle and steering

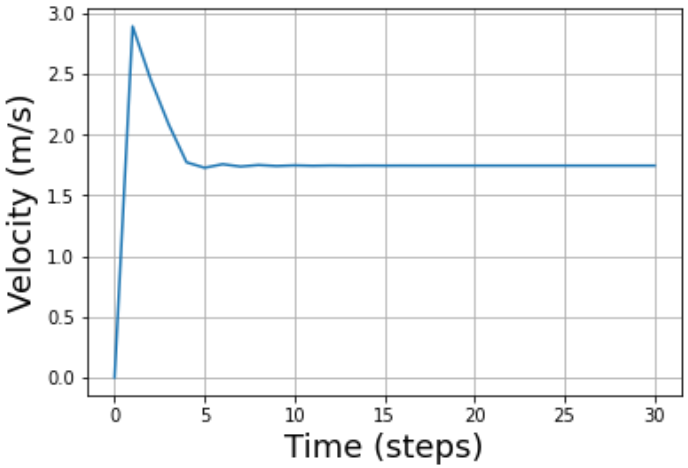


F1/10: control velocity

- Suppose we would like to achieve a target velocity of 2 m/s
- What is a simple approach to achieve that velocity?
  - Try some throttle and observe the error
  - If your velocity is under the target, increase thrust
  - It is enough to know that there is a positive relationship between thrust and velocity
- Attempt 1: apply thrust that is proportionate to the error, i.e., difference between current and target velocity
  - Suppose we observe velocity $v = 1$
  - Error is $e = v_T - v = 1$
  - Apply throttle proportionate to error, e.g.,

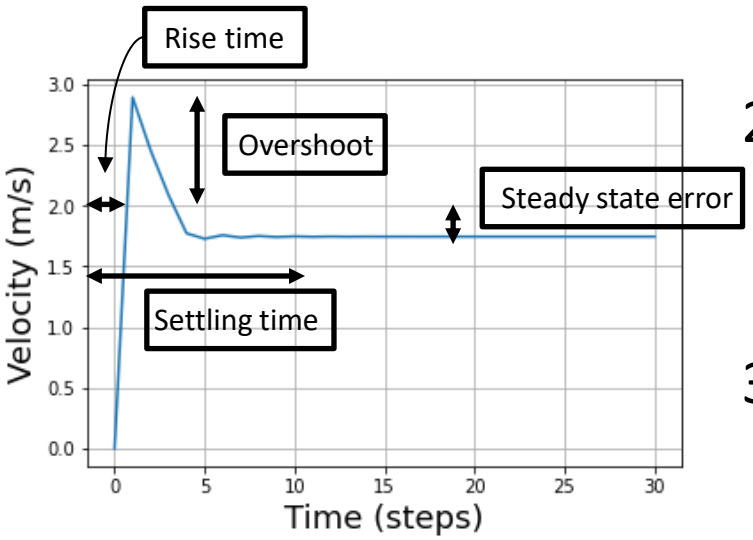
$$u = K_p e$$

Response of Proportionate Controller



- Step response: How will system output change if at time 0, with $v = 0$, we change reference input to 2?
- Beyond convergence, what are desired characteristics of the response?

Characteristics of the Step Response

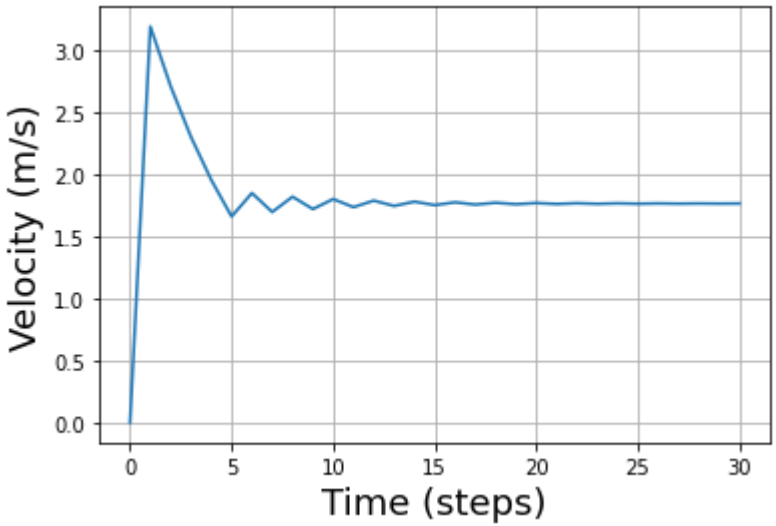


1. Overshoot: Difference between maximum output value and reference value
2. Rise Time: Time at which the output value crosses reference value
3. Settling Time: Time at which output value reaches steady-state value
4. Steady State Error: Difference between steady-state output value and reference

Why is there steady-state error?

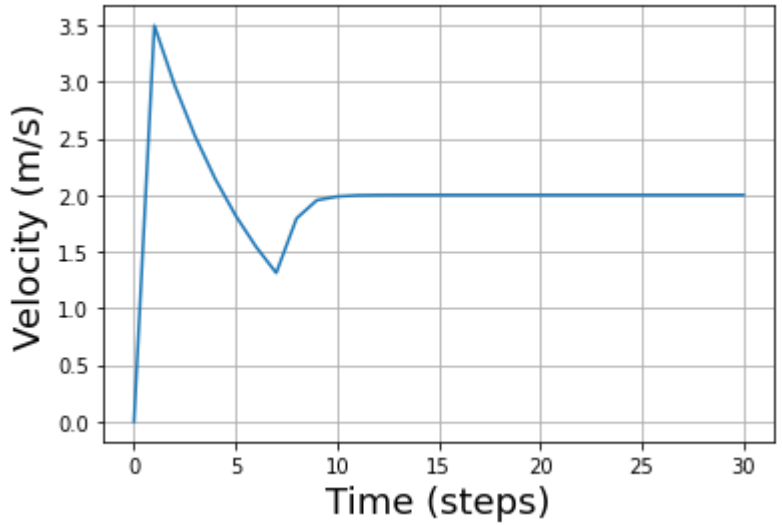
Eventually error becomes small enough so that a proportional controller can't remove it

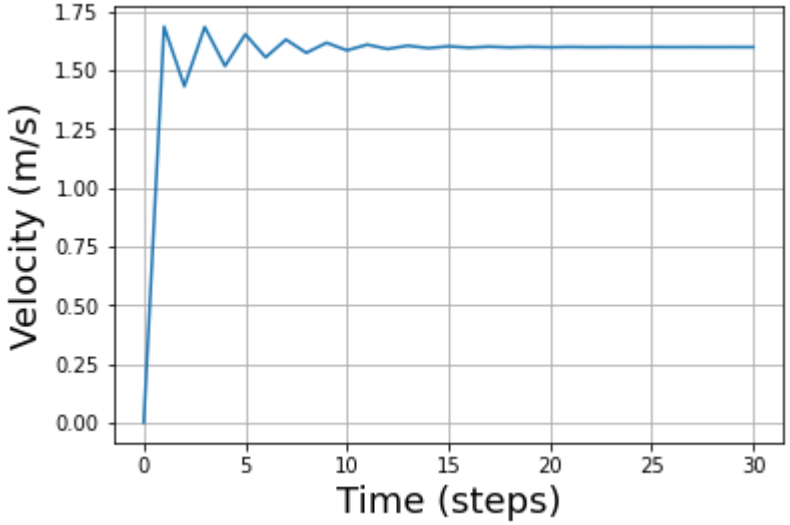
Improving the Step Response



- Performance of the P-controller depends on the value of the proportional gain constant K_P
- What happens if we increase it?
- Rise time decreases, but overshoot increases
- Steady-state error remains!
- How do we get rid of steady-state error?

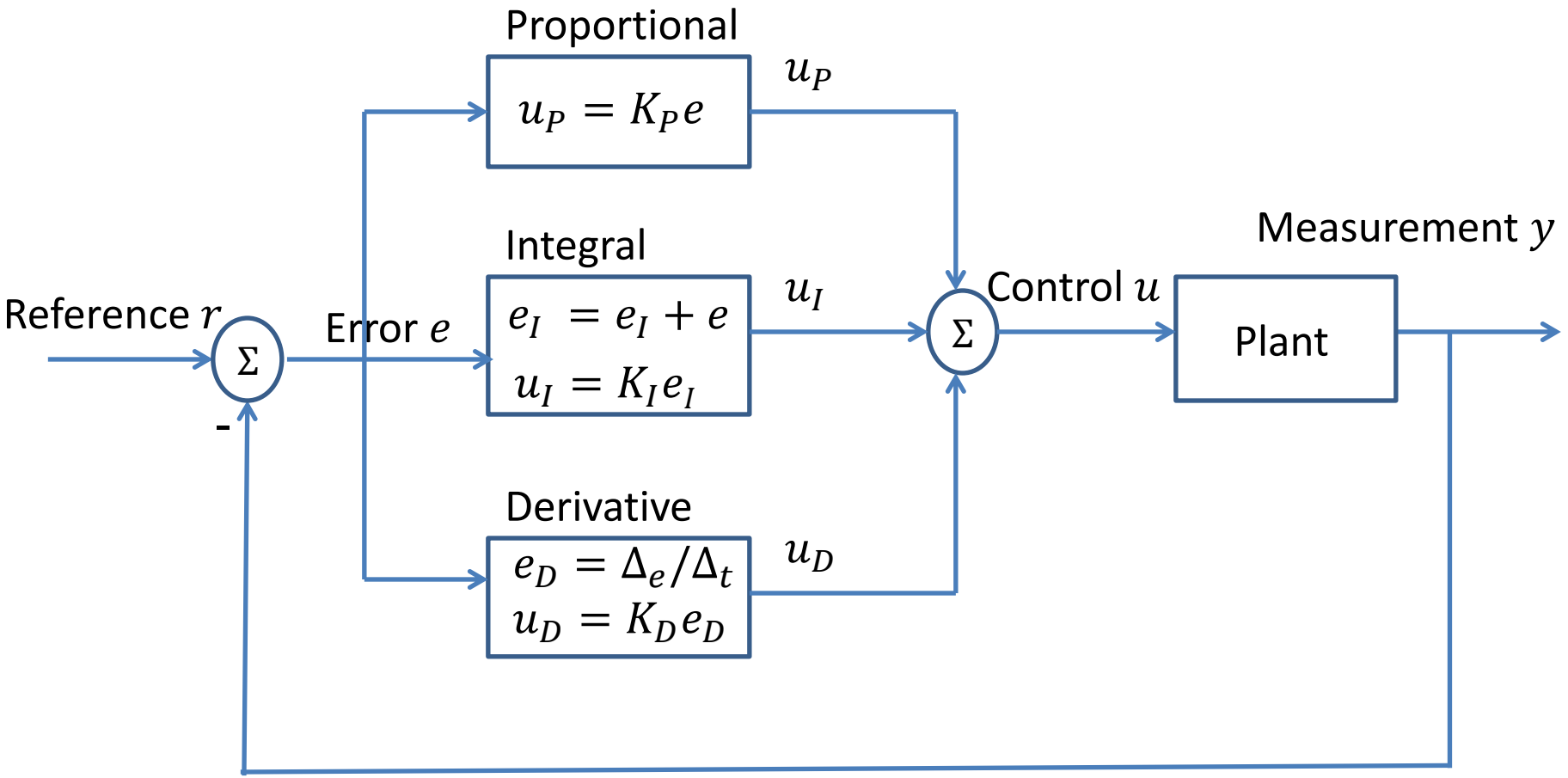
Adding up errors over time





- PI Controller: add up errors over time and adjust throttle accordingly
  - Even if steady-state error is very small, it will eventually accumulate and be corrected
  - Overshoot, rise time, settling time increase (why?)
- PD controller: adding derivative term to proportional controller gets rid of overshoot
  - Steady state error remains

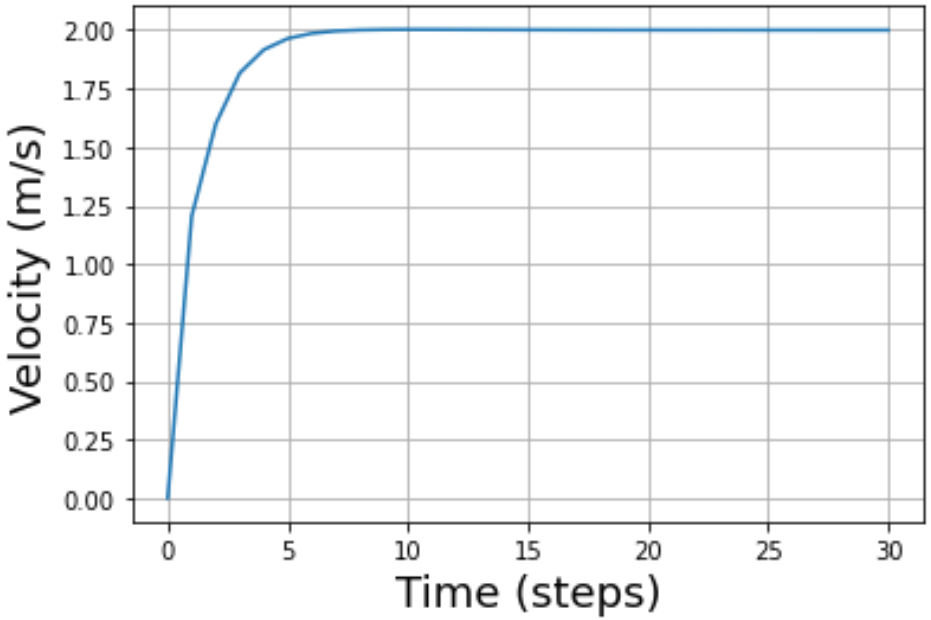
PID Controller



PID Controller

- If $e(t)$ is the error signal, then the output $u(t)$ of the PID controller is the sum of 3 terms:
  - Proportional term: $K_P e(t)$, K_P is called proportional gain (response to current error)
  - Integral term: $K_I \int_0^1 e(t)dt$, K_I is integral gain (response to error accumulated so far)
  - Derivative term: $K_D \dot{e}$, K_D is derivative gain (response to current rate of change of error)
- Special cases of controllers: P, PD, PI
  - You rarely need all 3

PID Controller for F1/10 Car Velocity



- Excellent performance on all metrics
  - $K_P = 18, K_D = 0.2, K_I = 4$
- Small rise time, settling time, negligible steady state error, no overshoot

Deficiencies of PID controller

- When is the PID controller not sufficient?
  - For example, can you solve Mountain Car?
  - No, because you need to get farther from the goal first
- PID controller is only good when the error provides enough information
  - Sometimes, you need to plan ahead
  - Need to know how your control affects the plant
  - Need to know the dynamics of the plant!
- For more sophisticated control, we need to model the plant
- For even more complex tasks and in case of bad models, need RL