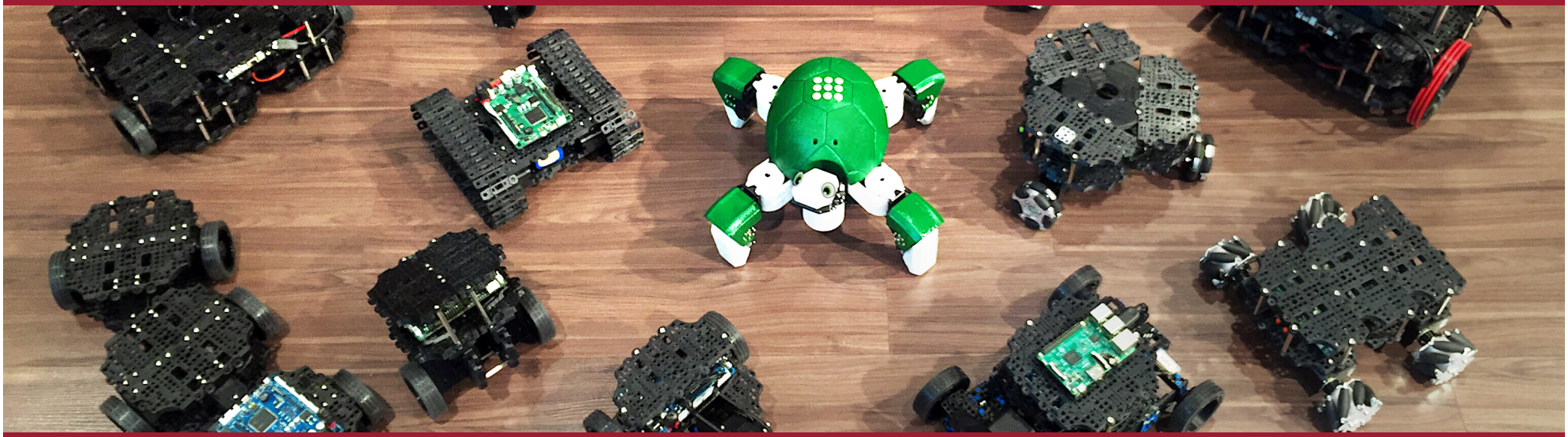
Controls (PID)



Intro. to Autonomous Robotics

Dr. Chris S. Crawford
Dept. of Computer Science
University of Alabama

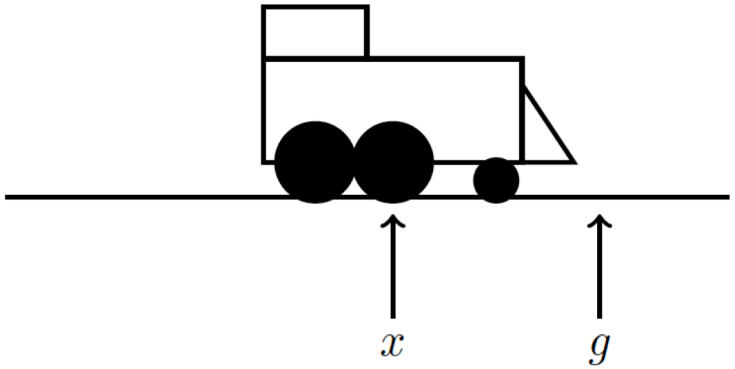
The Role of Control

- Robotics tasks are often defined by *achievement goals:*
  - Ex. *Go to (x , y)*

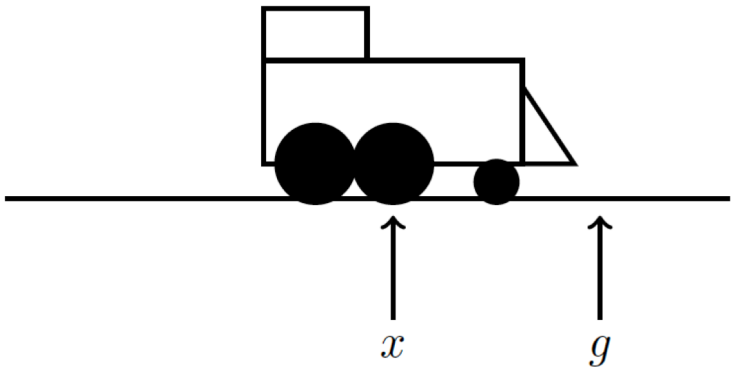
- Other tasks in robotics are defined by maintenance goals:
  - Ex. Drive at 0.5m/s

- Two common types of control systems
  - **Open-loop (feed-forward) control**
  - **Closed-loop (feedback) control**

Open-loop control



Open-loop control



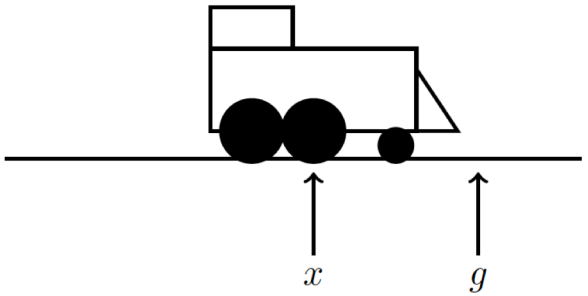

```
def openLoop(x, g):

    # Calculate the distance to travel:
    d = g - x

    # Cover that distance in one second:
    for one second:
            drive forward at speed of d / second
```

Open-loop control

- Unrealistic assumption of **instantaneous speed change** from 0 to desired d/s

- Even after reaching target speed, **friction and mechanical imperfection** will make it impossible to maintain d/s

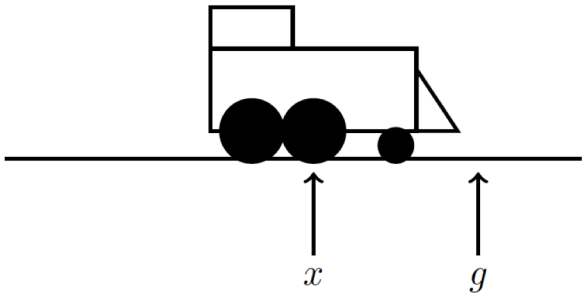
- Overtime **small errors** in speed will result in significant errors in the final position



Avoiding Open-loop Control Issues

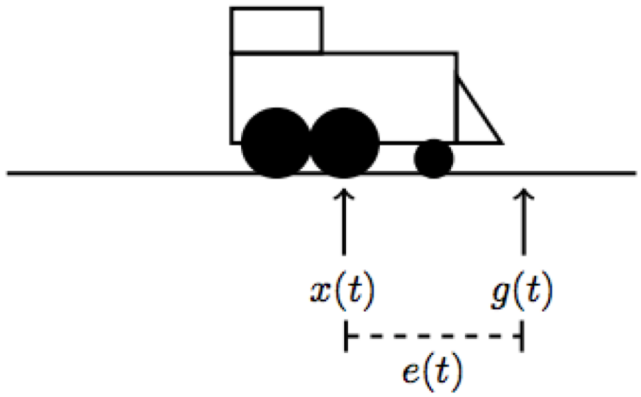
- **Continuously monitor** the current "**error**" in the system and **update the control signal** to **push the "error" toward zero**.

- Responds to **actual state** of the system (robot) and is able to **make adjustments** when systems fails to behave as expected.



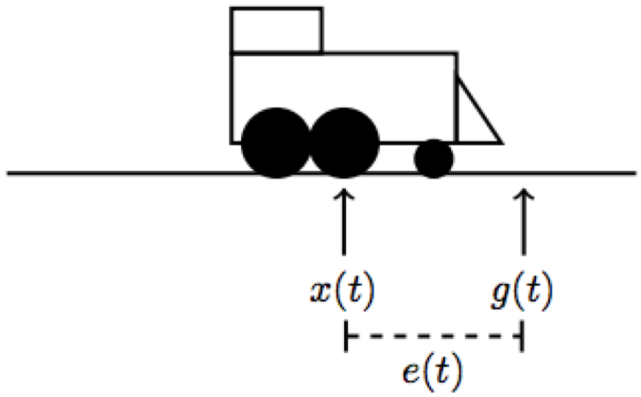
Proportional Control

- **Continuously monitor** the current "**error**" in the system and **update the control signal** to **push the "error" toward zero**.



Proportional Control

- **Continuously monitor** the current "**error**" in the system and **update the control signal** to **push the "error" toward zero**.



$x(t)$ - The state of a system at time t

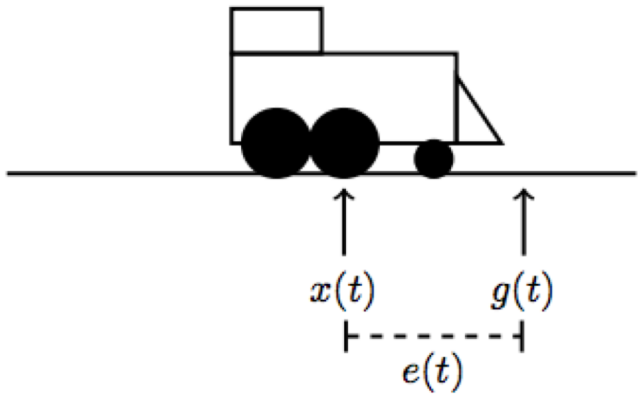
$g(t)$ - The goal state of a system at time t

$e(t)$ - Error at time t determine by $e(t) = g(t) - x(t)$

$u(t)$ - Control signal at time t

Proportional Control

- **Continuously monitor** the current "**error**" in the system and **update the control signal** to **push the "error" toward zero**.



$x(t)$ - The state of a system at time t

$g(t)$ - The goal state of a system at time t

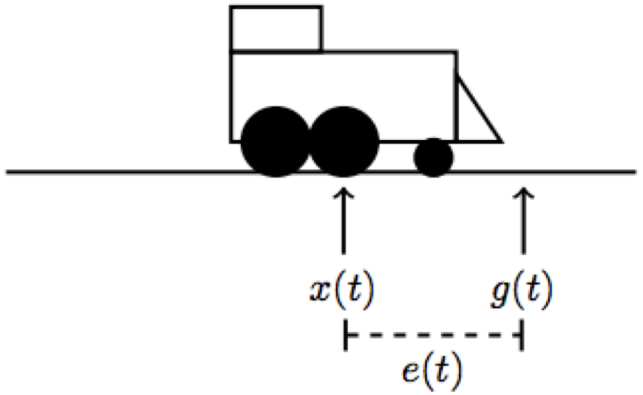
$e(t)$ - Error at time t determine by $e(t) = g(t) - x(t)$

$u(t)$ - Control signal at time t

K_p - **Gain** Controls how large the control signal will be for a particular error value.

$$u(t) = K_p e(t)$$

Proportional Control




```
def p_controller(robot, g, K_P):
    while true:
        e = g - robot.x
        u = K_P * e
        robot.move(u)
```

$u(t) = K_p e(t)$

$x(t)$ - The state of a system at time t

$g(t)$ - The goal state of a system at time t

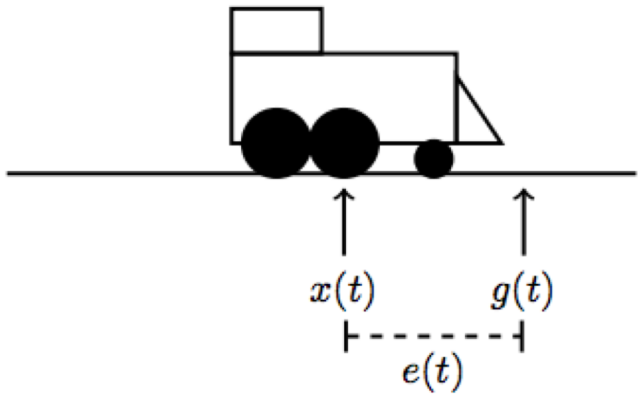
$e(t)$ - Error at time t determine by $e(t) = g(t) - x(t)$

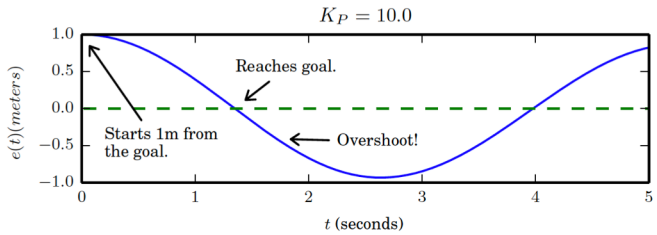
$u(t)$ - Control signal at time t

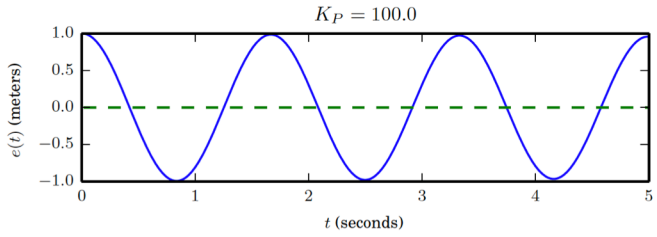
K_p - **Gain** Controls how large the control signal will be for a particular error value.

Proportional Control

- Overshoots the goal
- Over-corrects
- Moves back and forth indefinitely
- Changing Kp doesn't solve the problem



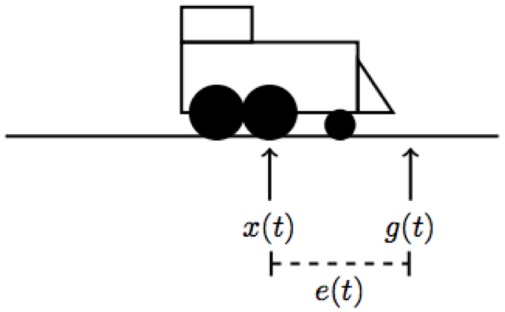


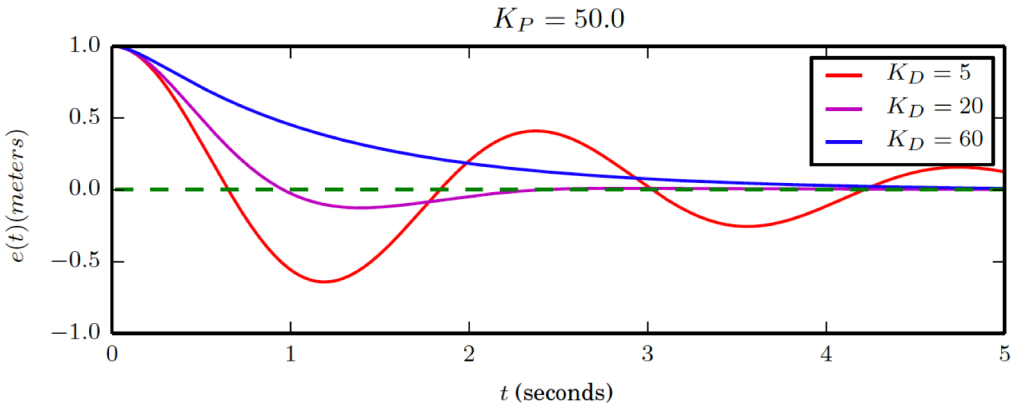


Proportional Derivative Control

- If error is **going down quickly**, **ease off** on the control signal

- If error is **going up quickly**, **increase** the control signal

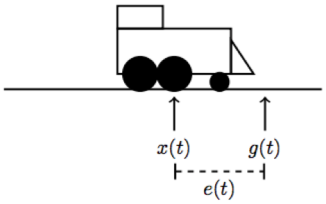




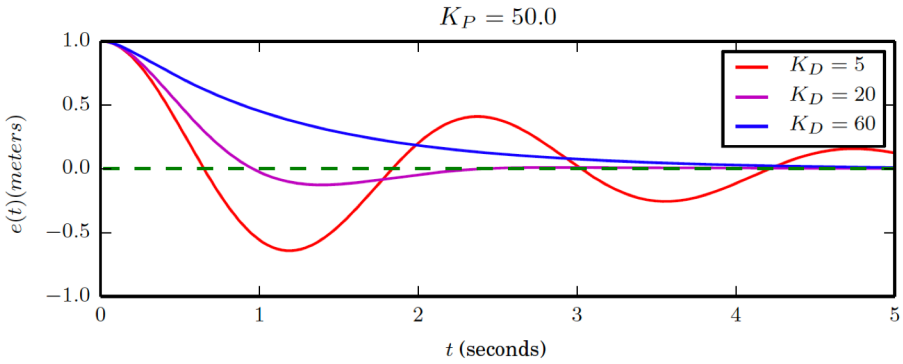
$$u(t) = K_p e(t) + K_d \frac{de(t)}{dt}$$

$$\frac{de(t)}{dt} \approx \frac{e(t) - e(t - \Delta t)}{\Delta t}$$

Proportional Derivative Control




```
def pd_controller(robot, g, K_P, K_D):

    # Keep track of previous error
    e_prev = g - robot.x

    while true:
        e = g - robot.x

        # Calculate approximate derivative
        dedet = (e - eprev) / robot.dt

        u = K_P * e + K_D * dedt
        robot.move(u)
        eprev = e
```

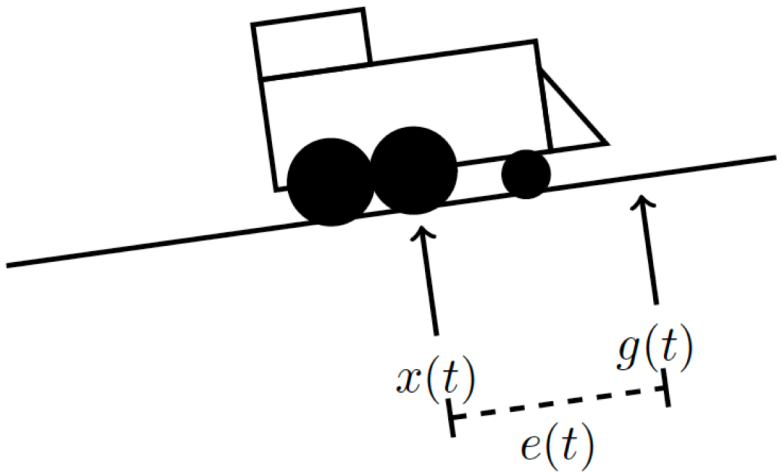


$$u(t) = K_p e(t) + K_d \frac{de(t)}{dt}$$

$$\frac{de(t)}{dt} \approx \frac{e(t) - e(t - \Delta t)}{\Delta t}$$

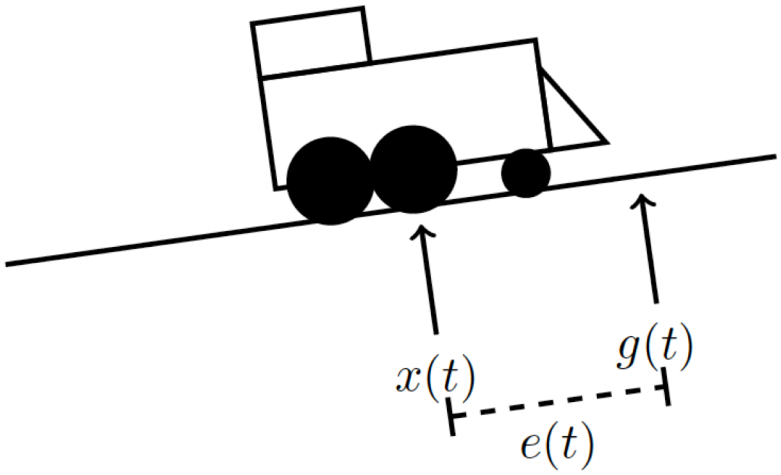
Proportional Control

```
def pd_controller(robot, g, K_P, K_D):

    # Keep track of previous error
    e_prev = g - robot.x

    while true:
        e = g - robot.x

        # Calculate approximate derivative
        dedet = (e - eprev) / robot.dt

        u = K_P * e + K_D * dedt
        robot.move(u)
        eprev = e
```



Proportional Integral Derivative Control

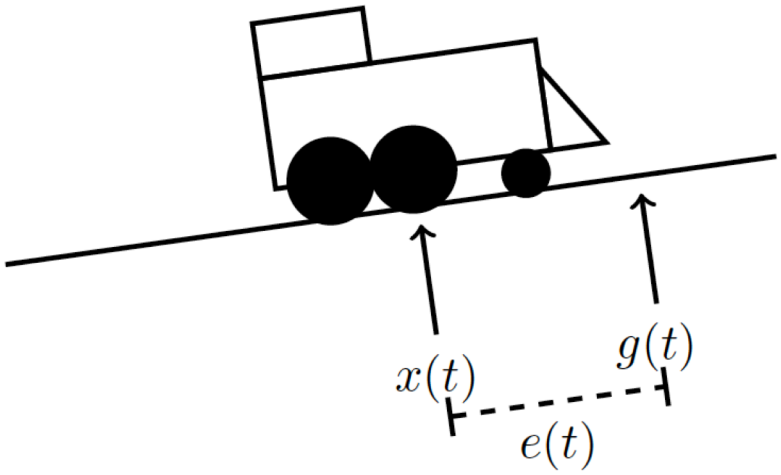
- Integral term allows the controller to **look backwards in time.**
- "stores up" the error that the system sees over time.
- As long as the error fails to reach zero, the integral term will steadily increase in magnitude.



$$u(t) = K_P e(t) + \overbrace{K_I \int_0^t e(\tau)d\tau}^{\text{integral term}} + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

Proportional Integral Derivative Control

- Integral term allows the controller to **look backwards in time.**
- "stores up" the error that the system sees over time.
- As long as the error fails to reach zero, the integral term will steadily increase in magnitude.

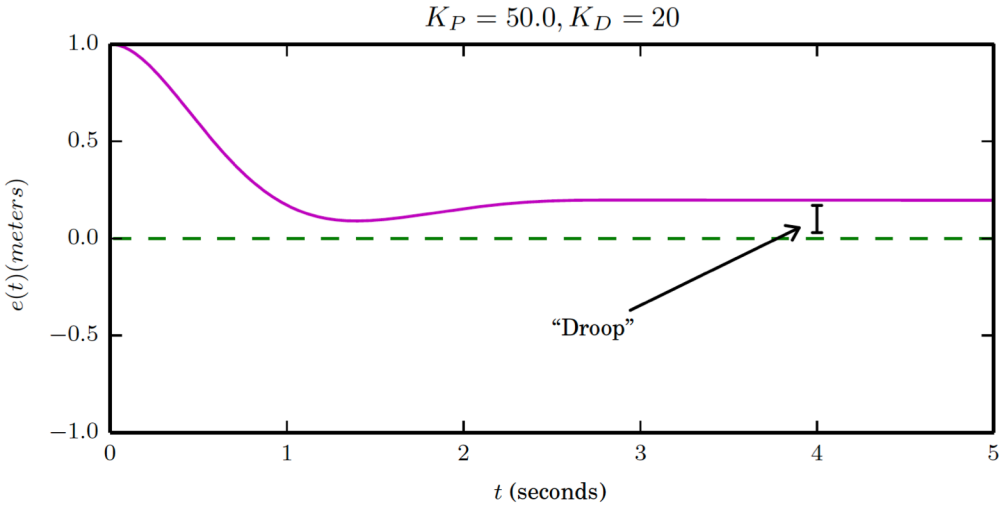


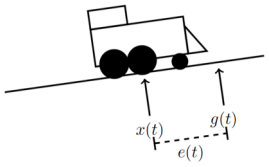
$$u(t) = K_P e(t) + \overbrace{K_I \int_0^t e(\tau)d\tau}^{\text{integral term}} + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

$$K_I \int_0^t e(t')dt' \approx \sum_{i=1}^{t/\Delta t} e(i\Delta t)\Delta t$$

Proportional Integral Derivative Control

- Integral term allows the controller to **look backwards in time.**
- "stores up" the error that the system sees over time.
- As long as the error fails to reach zero, the integral term will steadily increase in magnitude.





integral term

$$u(t) = K_P e(t) + \boxed{K_I \int_0^t e(\tau)d\tau} + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

$$K_I \int_0^t e(t')dt' \approx \sum_{i=1}^{t/\Delta t} e(i\Delta t)\Delta t$$

Proportional Integral Derivative Control

```python
def pid_controller (train , g, K_P , K_I , K_D ):

    e_prev = g - train .x  # Keep track of previous error
    e_sum = 0 # accumulator for integral term

    while True :

        # Current Error
        e = g - train .x

        # Integral Approximation
        e_sum = e_sum + e * train .dt

        # Derivative Approximation
        dedt = (e - e_prev ) / train .dt

        # Control signal
        u = K_P * e + K_I * e_sum + K_D * dedt

        # New Commmand
        train . throttle (u)

        # Store error
        e_prev = e
```